



Universidade Federal de Pernambuco  
Centro de Informática

Pós-Graduação em Ciência da Computação

wGEM: um *Framework* de Desenvolvimento  
de Jogos para Dispositivos Móveis

por

Carlos André Cavalcante Pessoa

**Dissertação de Mestrado**

Recife, Novembro de 2001



UNIVERSIDADE FEDERAL DE PERNAMBUCO  
CENTRO DE INFORMÁTICA

CARLOS ANDRÉ CAVALCANTE PESSOA

**wGEM: um *Framework* de Desenvolvimento  
de Jogos para Dispositivos Móveis**

*Este trabalho foi submetido à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação*

Orientador: Prof. Dr. Geber Lisboa Ramalho

Recife, 1 de Novembro de 2001



## AGRADECIMENTOS

A Deus, por me dar a vida e as condições para a realização deste trabalho.

Aos meus pais e irmãos, por todo o suporte me dado, especialmente durante minha formação acadêmica.

Ao meu professor e orientador Geber Ramalho, pelo apoio e dedicação à esta pesquisa.

Aos professores da banca examinadora, Paulo Borba e André Battaiola, pela dedicação em avaliar e contribuir da melhor forma para o melhoramento deste trabalho.

A todos os professores do Centro de Informática, pela excelente formação que tem dado aos seus alunos.

Aos integrantes do grupo de desenvolvimento de aplicações para celulares do CESAR, especialmente a Danielle Silva, Pedro Henrique e Rafael Palermo, pela contribuição dada ao desenvolvimento deste trabalho.

À Universidade Federal de Pernambuco, ao CNPq, e ao CESAR, pelo ambiente e apoio financeiro à realização desta pesquisa.

E a todos aqueles que de alguma forma contribuíram durante o mestrado, a escrita desta dissertação e o desenvolvimento do wGEM.



## RESUMO

O surpreendente avanço e diversidade dos jogos eletrônicos atuais demonstram o enorme sucesso desta área. Uma das tendências mais novas desta indústria é voltada para os novos dispositivos móveis, como celulares e *handhelds*, que adotaram a tecnologia Java 2 Micro Edition (J2ME) como arquitetura padrão para execução de software. Entretanto, dada a simplicidade de J2ME e seu curto tempo de criação, várias ferramentas de apoio à implementação de jogos usando J2ME precisam ser concebidas, em especial um *framework* de desenvolvimento de jogos. Este trabalho apresenta o *framework* wGEM, resultado de uma generalização e adaptação das características encontradas em *frameworks* de desenvolvimento de jogos para PCs às limitações dos dispositivos móveis e da tecnologia J2ME. wGEM, o primeiro *framework* do gênero que se tem notícia no mundo, facilita a implementação em escala industrial de jogos para dispositivos móveis e promove uma melhor qualidade deles, pois foi desenvolvido, testado e otimizado para realizar as tarefas fundamentais presentes nos jogos eletrônicos. Por fim, o wGEM representa o primeiro esforço do Centro de Informática da UFPE para dominar a tecnologia J2ME.

## ABSTRACT

The impressive advance and diversity of today's electronic games illustrate the tremendous success of this area. One of the trends in this industry is focused on the new mobile devices, as cell phones and handhelds, which have chosen the Java 2 Micro Edition technology (J2ME) as the standard software architecture. However, given the J2ME's simple resources and its recent emergence, support tools to the development of games using J2ME need to be designed, in special a framework for game development. This work presents the wGEM framework, as a result of a generalization and an adaptation of the features found in development frameworks for PC games to the limitations of the mobile devices and the J2ME technology. wGEM, the first announced framework of its sort, simplifies the implementation of mobile games in an industrial scale and promotes a better quality of them, since it was developed, tested and optimized to do the main tasks needed in electronic games. Finally, wGEM represents the first effort of the UFPE Informatics Center to master the J2ME technology.





# ÍNDICE

<b>1</b>	<b>INTRODUÇÃO.....</b>	<b>1</b>
1.1	MOTIVAÇÃO.....	2
1.2	OBJETIVOS .....	3
1.3	DESAFIOS .....	3
1.4	ESTRUTURA DA DISSERTAÇÃO.....	4
<b>2</b>	<b>JOGOS ELETRÔNICOS: CONCEITOS E EVOLUÇÃO .....</b>	<b>5</b>
2.1	CONCEITOS BÁSICOS.....	5
2.1.1	<i>Componentes de um Jogo.....</i>	<i>5</i>
2.1.2	<i>Ciclo de Atividades de um Jogo.....</i>	<i>6</i>
2.2	EVOLUÇÃO DOS JOGOS ELETRÔNICOS.....	8
2.2.1	<i>Tecnológica .....</i>	<i>8</i>
2.2.2	<i>Acadêmica .....</i>	<i>10</i>
2.2.3	<i>Comercial .....</i>	<i>14</i>
2.3	A ERA DOS JOGOS PARA DISPOSITIVOS MÓVEIS .....	14
2.3.1	<i>Soluções dependentes de plataforma.....</i>	<i>15</i>
2.3.2	<i>WAP.....</i>	<i>16</i>
2.3.3	<i>i-Mode .....</i>	<i>16</i>
2.3.4	<i>SMS.....</i>	<i>17</i>
2.3.5	<i>J2ME .....</i>	<i>17</i>
2.4	CONCLUSÕES .....	18
<b>3</b>	<b>DESENVOLVIMENTO DE JOGOS .....</b>	<b>21</b>
3.1	HISTÓRICO .....	21
3.2	ETAPAS DO DESENVOLVIMENTO DE JOGOS .....	25
3.2.1	<i>Game Design .....</i>	<i>25</i>
3.2.2	<i>Especificação da Arquitetura e Implementação do Jogo .....</i>	<i>27</i>
3.2.3	<i>Testes de Corretude e Jogabilidade.....</i>	<i>27</i>
3.3	BIBLIOTECAS MULTIMÍDIA .....	28
3.4	FRAMEWORKS DE DESENVOLVIMENTO DE JOGOS .....	29
3.4.1	<i>Requisitos .....</i>	<i>29</i>
3.4.2	<i>Implementação .....</i>	<i>30</i>
3.4.3	<i>Editor de Cenários.....</i>	<i>32</i>
3.5	FERRAMENTAS DE APOIO AO DESENVOLVIMENTO DE JOGOS.....	33
3.6	CONCLUSÕES .....	34
<b>4</b>	<b>JAVA 2 MICRO EDITION (J2ME).....</b>	<b>35</b>
4.1	HISTÓRICO .....	35

4.2	ARQUITETURA DE J2ME .....	39
4.2.1	<i>Máquina Virtual Java</i> .....	40
4.2.2	<i>Configuração</i> .....	41
4.2.2.1	CLDC .....	41
4.2.2.2	CDC .....	42
4.2.3	<i>Perfil</i> .....	43
4.2.3.1	MID Profile (MIDP) .....	44
4.2.3.2	Foundation Profile .....	45
4.3	FERRAMENTAS DE DESENVOLVIMENTO .....	46
4.3.1	<i>Instalação de Programas nos Dispositivos Móveis</i> .....	46
4.4	ADERÊNCIA MERCADOLÓGICA .....	48
4.5	CONCLUSÕES .....	49
<b>5</b>	<b>DESENVOLVIMENTO DE JOGOS EM J2ME .....</b>	<b>51</b>
5.1	RECURSOS DE J2ME PARA O DESENVOLVIMENTO DE JOGOS .....	51
5.1.1	<i>Suporte</i> .....	51
5.1.2	<i>Limitações</i> .....	52
5.2	UTILIZAÇÃO DE FRAMEWORKS EM DISPOSITIVOS MÓVEIS .....	53
5.2.1	<i>Vantagens</i> .....	53
5.2.2	<i>Desvantagens</i> .....	54
5.3	JOGOS RECOMENDÁVEIS PARA DISPOSITIVOS MÓVEIS .....	54
5.4	CONCLUSÕES .....	56
<b>6</b>	<b>WGEM.....</b>	<b>57</b>
6.1	DA CONCEPÇÃO AO ESTÁGIO ATUAL .....	57
6.2	REQUISITOS .....	58
6.2.1	<i>Motor</i> .....	58
6.2.2	<i>Editor de Cenários</i> .....	59
6.3	IMPLEMENTAÇÃO DO MOTOR .....	60
6.3.1	<i>Objeto do Jogo</i> .....	62
6.3.1.1	Representação Visual .....	63
6.3.1.2	Deteção de Colisão .....	65
6.3.2	<i>Mapa do Jogo</i> .....	66
6.3.2.1	Mapa baseado em cores .....	68
6.3.2.2	Mapa baseado em texturas .....	68
6.3.2.3	Mapa baseado em <i>tiles</i> .....	69
6.3.3	<i>Gerenciador de Objetos</i> .....	71
6.3.3.1	Repositório de objetos .....	72
6.3.3.2	Teste de interseção .....	73
6.3.4	<i>Gerenciador de Entrada</i> .....	74
6.3.5	<i>Gerenciador Gráfico</i> .....	75
6.3.6	<i>Gerenciador do Jogo</i> .....	76

6.3.6.1	Ciclo do jogo.....	77
6.3.6.2	Inicialização do cenário.....	78
6.3.7	<i>Gerenciador de Rede</i> .....	78
6.3.8	<i>Aplicação</i> .....	79
6.3.9	<i>Integração com o Editor de Cenários</i> .....	80
6.4	IMPLEMENTAÇÃO DO EDITOR DE CENÁRIOS .....	80
6.4.1	<i>Definição do Mapa</i> .....	81
6.4.2	<i>Definição dos Objetos</i> .....	81
6.4.3	<i>Definição do Cenário</i> .....	82
6.4.4	<i>Persistência de Cenários</i> .....	83
6.4.5	<i>Integração com o Motor</i> .....	84
6.5	CONCLUSÕES .....	85
<b>7</b>	<b>ESTUDOS DE CASO .....</b>	<b>87</b>
7.1	BREAKOUT .....	87
7.1.1	<i>Arquitetura do jogo</i> .....	87
7.1.1.1	Gerenciadores de objetos .....	88
7.1.1.2	Objetos do jogo .....	88
7.1.1.3	Mapa do jogo .....	88
7.1.1.4	Gerenciador de Entrada e Gráfico .....	88
7.1.1.5	Gerenciador do Jogo .....	89
7.1.1.6	Definição do cenário .....	89
7.2	SHIP .....	90
7.2.1	<i>Arquitetura do jogo</i> .....	90
7.2.1.1	Gerenciadores de objetos .....	90
7.2.1.2	Objetos do jogo .....	91
7.2.1.3	Mapa do jogo .....	91
7.2.1.4	Gerenciador de Entrada e Gráfico .....	91
7.2.1.5	Gerenciador do Jogo .....	92
7.2.1.6	Definição do cenário .....	92
7.3	ENDURO .....	93
7.4	AVALIAÇÃO DO RESULTADO.....	94
7.4.1	<i>Desempenho</i> .....	94
7.4.2	<i>Tamanho dos Jogos</i> .....	95
7.5	CONCLUSÕES .....	96
<b>8</b>	<b>CONCLUSÕES.....</b>	<b>97</b>
8.1	CONTRIBUIÇÕES.....	97
8.2	DIFICULDADES .....	97
8.3	TRABALHOS FUTUROS .....	98
<b>9</b>	<b>REFERÊNCIAS BIBLIOGRÁFICAS.....</b>	<b>99</b>



# ÍNDICE DE FIGURAS

FIGURA 2-1: CICLO DE ATIVIDADES DE UM JOGO.....	8
FIGURA 2-2: DOIS PDAS E UM HANDHELD, RESPECTIVAMENTE .....	16
FIGURA 3-1: CAMADAS DE SOFTWARE PARA EXECUÇÃO DE UM JOGO .....	23
FIGURA 3-2: EXEMPLO DE TRÊS CENÁRIOS PARA O JOGO BREAKOUT .....	32
FIGURA 4-1: CRESCIMENTO NO TAMANHO DA LINGUAGEM JAVA.....	36
FIGURA 4-2: SMART CARD E JAVA RING COM A TECNOLOGIA JAVACARD .....	37
FIGURA 4-3: OSCILOSCÓPIO DA EMPRESA TEKTRONIX BASEADO NA TECNOLOGIA EMBEDDED JAVA .....	37
FIGURA 4-4: DOIS SCREENPHONES E UM PDA SUPORTANDO PERSONALJAVA.....	38
FIGURA 4-5: MODELO DE CAMADAS DE J2ME.....	40
FIGURA 4-6: COBERTURA DE CLASSES DE CDC E CLDC EM RELAÇÃO À J2SE.....	43
FIGURA 4-7: ATUAL ARQUITETURA DE JAVA [58] .....	45
FIGURA 4-8: AMBIENTE DE DESENVOLVIMENTO PARA J2ME.....	46
FIGURA 4-9: PROCESSO DE INSTALAÇÃO DE PROGRAMAS EM DISPOSITIVOS.....	47
FIGURA 4-10: EXEMPLOS DE DISPOSITIVOS QUE SUPORTAM J2ME .....	48
FIGURA 6-1: COMUNICAÇÃO ENTRE OS MÓDULOS DO MOTOR DO WGEM .....	60
FIGURA 6-2: CLASSE GAMEOBJECT .....	63
FIGURA 6-3: CLASSES PARA REPRESENTAÇÃO VISUAL DOS OBJETOS DO JOGO .....	64
FIGURA 6-4: USO DE RETÂNGULOS PARA TESTAR COLISÃO ENTRE OBJETOS.....	65
FIGURA 6-5: EXEMPLOS DE FALHAS NA DETECÇÃO DE COLISÃO .....	66
FIGURA 6-6: TESTE DE COLISÃO BASEADO EM PROJEÇÃO SOBRE OS EIXOS.....	66
FIGURA 6-7: CLASSES DO WGEM PARA A IMPLEMENTAÇÃO DO MAPA DO JOGO .....	68
FIGURA 6-8: EXEMPLO DE IMAGEM PARA MAPAS BASEADOS EM TEXTURAS.....	69
FIGURA 6-9: EXEMPLO DE USO DE <i>TILES</i> NA MONTAGEM DE UM CENÁRIO .....	70
FIGURA 6-10: CLASSES PARA IMPLEMENTAÇÃO DO GERENCIADOR DE OBJETOS.....	72
FIGURA 6-11: ORGANIZAÇÃO DOS OBJETOS DE ACORDO COM A PERTINÊNCIA A UMA REGIÃO .....	73
FIGURA 6-12: EXEMPLO DE BUSCA POR UM OBJETO EM COLISÃO .....	74
FIGURA 6-13: EXEMPLO DE BOUDING BOX PARA BUSCA DE UM OBJETO EM COLISÃO.....	74
FIGURA 6-14: CLASSE IO.....	75
FIGURA 6-15: CLASSE GAMEMANAGER .....	77
FIGURA 6-16: ILUSTRAÇÃO DOS EVENTOS ENVOLVIDOS NO CICLO DO JOGO.....	78
FIGURA 6-17: CLASSE NETWORKMANAGER.....	79
FIGURA 6-18: CLASSE MIDLETGAME .....	80
FIGURA 6-19: MÓDULO DE INTEGRAÇÃO DO MOTOR COM O EDITOR DE CENÁRIOS.....	80
FIGURA 6-20: INTERFACE DO EDITOR PARA DEFINIÇÃO DO MAPA DO JOGO .....	81
FIGURA 6-21: PALETA DE OBJETOS DO EDITOR DE CENÁRIOS .....	82
FIGURA 6-22: DEFINIÇÃO DE OBJETOS ATRAVÉS DO EDITOR DE CENÁRIOS.....	82
FIGURA 6-23: ILUSTRAÇÃO DO MAPA OFERECIDA PELO EDITOR DE CENÁRIOS .....	83

FIGURA 6-24: INTERFACE DO EDITOR DE CENÁRIOS PARA A EDIÇÃO DA VELOCIDADE DOS OBJETOS .....	83
FIGURA 7-1: JOGO BREAKOUT .....	87
FIGURA 7-2: PALETA DE OBJETOS DO JOGO BREAKOUT .....	90
FIGURA 7-3: CENÁRIO DO JOGO BREAKOUT .....	90
FIGURA 7-4: JOGO SHIP .....	90
FIGURA 7-5: PALETA DE OBJETOS DO JOGO SHIP .....	93
FIGURA 7-6: CENÁRIO DO JOGO SHIP .....	93
FIGURA 7-7: JOGO ENDURO.....	93

# ÍNDICE DE TABELAS

TABELA 3-1: RECEITA BÁSICA PARA EQUIPES DE DESENVOLVIMENTO DE JOGOS NOS ANOS 80 [5].....	23
TABELA 3-2: PRINCIPAIS PAPÉIS DOS INTEGRANTES DE UMA FÁBRICA DE JOGOS .....	24
TABELA 3-3: RESUMO DOS PRINCIPAIS ELEMENTOS DE UM DOCUMENTO DE GAME DESIGN.....	26
TABELA 6-1: COMPONENTES E FUNCIONALIDADES PARA O MOTOR DO <i>FRAMEWORK</i> wGEM .....	59
TABELA 6-2: FUNCIONALIDADES DO EDITOR DE CENÁRIOS DO wGEM.....	60
TABELA 6-3: FORMATO DO ARQUIVO DE DESCRIÇÃO DE CENÁRIOS .....	84
TABELA 7-1: DICIONÁRIO DE OBJETOS DO JOGO BREAKOUT .....	89
TABELA 7-2: DICIONÁRIO DE OBJETOS DO JOGO SHIP.....	92
TABELA 7-3: PERFORMANCE DOS JOGOS DESENVOLVIDOS COM O wGEM .....	95
TABELA 7-4: TAMANHO DOS JOGOS DESENVOLVIDOS COM O wGEM .....	96





# 1 INTRODUÇÃO

A atual quantidade e diversidade de jogos eletrônicos é a prova maior de seu grande sucesso. Além da presença firme nos computadores pessoais, da enorme quantidade de videogames e de simuladores profissionais, essa forma de entretenimento há alguns anos já vem sendo utilizada em dispositivos bem simples, tais como os celulares, onde diversos jogos fazem parte dos programas que são desenvolvidos sob a coordenação do fabricante do dispositivo e que vem instalados no aparelho.

A utilização de jogos em dispositivos móveis aumentará consideravelmente devido a aderência desta indústria à uma arquitetura padrão e aberta que permite o desenvolvimento de aplicações para tais dispositivos pela comunidade interessada. Essa tecnologia, chamada J2ME (de Java 2 *Micro Edition*) [13], consiste na simplificação da linguagem e máquina virtual Java para tais dispositivos. Seu surgimento, em 2000, é o subsídio tecnológico básico para que o desenvolvimento de jogos para dispositivos móveis torne-se possível e similar ao que ocorre em PCs.

Infelizmente, como era de se esperar de uma tecnologia recente, ainda não há registro, à exceção deste trabalho, de um esforço de implementação, nem tão pouco de definição clara e abrangente do que seria um *framework* de jogos para dispositivos móveis, ferramenta considerada fundamental para um desenvolvimento profissional de jogos.

Este trabalho propõe um *framework* original para o desenvolvimento de jogos para dispositivos móveis. Tal *framework*, chamado wGEM<sup>1</sup>, é resultado de uma generalização e adaptação das características encontradas em *frameworks* de desenvolvimento de jogos para PCs, para conceber uma solução compatível com as limitações de memória e processamento encontradas nos dispositivos móveis e na tecnologia J2ME.

A seguir são apresentadas as principais motivações que resultaram no desenvolvimento deste trabalho, os objetivos mais importantes a serem atingidos, os principais desafios em sua realização e um anúncio da estrutura desta dissertação.

---

<sup>1</sup> O nome wGEM origina-se das iniciais para **W**ireless **G**ame **E**ngine for **M**obile Devices.

## 1.1 MOTIVAÇÃO

O desenvolvimento de um trabalho na área de jogos eletrônicos é muito motivante, seja ele o desenvolvimento de jogos ou ferramentas de apoio. Afinal, se jogar é tão bom, desenvolver jogos, para alguns profissionais da computação, é uma realização dupla. Dentre as motivações mais fortes, algumas merecem destaque:

- Sucesso comercial da área de jogos eletrônicos;
- Grandes desafios e multidisciplinaridade;
- Novas tecnologias.

O grande sucesso comercial que a área de jogos tem alcançado nas últimas décadas é sem dúvida um fator determinante para o surgimento de novas empresas e produtos. Por outro lado, para tais empresas terem um grande sucesso, é essencial o investimento em pesquisa, que na maioria das vezes envolve a academia.

Poucas são as áreas da computação que têm a multidisciplinaridade encontrada na área de jogos. Mesmo em jogos simples, diversas áreas precisam ser envolvidas, tais como, Computação Gráfica, Inteligência Artificial, Engenharia de Software, Redes de Computadores, Algoritmos e Estruturas de Dados, Computação Musical, entre outras fora da computação que em geral são necessárias, tais como Psicologia, Educação, Artes, etc. Essa situação resulta em uma enorme diversidade de problemas a serem explorados, sendo certamente uma motivação forte para diversas pesquisas acadêmicas.

Um fato bastante comum à área de jogos é a utilização, e muitas vezes a criação, das mais recentes tecnologias. Esse cenário sem dúvida é muito motivador, pois torna a área sempre atualizada com o mercado, o que é muito bem visto por empresas e profissionais da computação.

Além dos fatores anteriormente citados, este trabalho, em particular, envolve a área de dispositivos móveis, que também traz características motivadoras. Primeiramente pelas grandes perspectivas da área, como, por exemplo, o grande crescimento vivenciado pela telefonia móvel, e mais ainda por envolver uma tecnologia muito recente e inédita no CIn, que é a utilização de J2ME para o desenvolvimento de aplicações.

## 1.2 OBJETIVOS

Como dito anteriormente, o desenvolvimento profissional de jogos demanda a utilização de diversas ferramentas de apoio, em especial a utilização de um *framework*. Até onde sabemos, tal recurso, apesar de muito útil, ainda não existe para dispositivos móveis devido ao pouco tempo da tecnologia J2ME.

Os motores de jogos (termo usado para os *frameworks*) são utilizados como peça chave na construção de jogos de computador, pois são responsáveis pela execução das tarefas de mais baixo nível essenciais para os jogos. Um motor é, portanto, o responsável pela execução do jogo, cabendo ao desenvolvedor apenas inserir as características de cada jogo criado. As grandes vantagens de um motor, se construído baseado numa arquitetura geral e modular, é a sua reutilização numa enorme variedade de jogos e a melhoria da qualidade de software proporcionada, uma vez que o uso de um *framework* requer um certo grau de organização da aplicação de acordo com a arquitetura adotada pelo *framework*. Além disso, o uso de um motor pode ser feito de forma seletiva, de modo que cada projeto específico consiga atingir as suas necessidades particulares, não necessitando modificações no motor. Como resultado, o desenvolvimento de jogos torna-se muito mais fácil, pois os desenvolvedores ficam voltados para as características do próprio jogo - lógica, arte, inteligência, entre outras - e não mais para os processamentos de baixo nível.

Dada essa importância dos motores, e a sua ausência para a tecnologia J2ME, o objetivo deste trabalho é projetar, implementar e testar um *framework* para jogos em dispositivos móveis. Para tal, será necessário uma generalização da arquitetura de motores de jogos existentes para PCs, a concepção e implementação do *framework* utilizando J2ME, a implementação de alguns jogos para validação do trabalho e o desenvolvimento de um editor de cenários, ferramenta de apoio que em geral faz parte de um *framework*.

## 1.3 DESAFIOS

A construção de um motor de jogos para J2ME impõe uma série de desafios devido à simplicidade da linguagem e dos dispositivos. Hoje em dia, em pleno auge das placas aceleradoras gráficas para PCs, bibliotecas multimídia, computadores com DVDs e abundância de memória e processamento, o desenvolvimento de aplicações para dispositivos móveis pode ser avaliada como uma retomada aos quesitos de restrições de

memória, processamento, recursos gráficos, e outros, que existiam nos computadores de 10 anos atrás. Em se tratando de jogos, esta situação é similar à da era dos primeiros videogames, onde uma série de otimizações de código e artifícios eram necessários para que os jogos se tornassem viáveis para a tecnologia disponível.

Dessa forma, uma série de adaptações são necessárias aos algoritmos utilizados hoje em dia na implementação de *frameworks*, visto que tais soluções assumem a existência de certos recursos, que, como será mostrado, não são possíveis em J2ME.

Os desafios citados são agravados pela originalidade no uso J2ME, tecnologia totalmente inédita no CIn, se não no Brasil.

## **1.4 ESTRUTURA DA DISSERTAÇÃO**

Os assuntos abordados nesta dissertação giram em torno de três eixos principais –desenvolvimento de jogos, J2ME e o *framework* wGEM. Nos próximos dois capítulos o desenvolvimento de jogos é tratado. Primeiramente em uma visão conceitual, evolutiva e tecnológica – correspondendo ao capítulo 2, e em seguida, com um pouco mais de profundidade, em uma visão técnica, são consideradas as etapas de desenvolvimento e as tendências de implementação dos jogos – sendo este o capítulo 3.

No capítulo 4 será abordada a tecnologia J2ME, sendo importante para familiarizar aqueles que não conhecem esta nova linguagem e para mostrar as características e limitações dos dispositivos móveis.

O capítulo 5 realiza uma análise da viabilidade e conseqüências de se utilizar J2ME no desenvolvimento de jogos para dispositivos móveis.

Os capítulos 6 dedica-se ao wGEM. Nele é detalhada a concepção e implementação do wGEM, sendo de grande importância para que o leitor possa compreender a arquitetura e as soluções aplicadas para a sua construção.

O capítulo 7 é dedicado a alguns exemplos de uso do wGEM através de jogos desenvolvidos para a sua validação.

Finalmente, serão apresentadas as conclusões tiradas com o desenvolvimento deste trabalho e algumas sugestões de trabalhos futuros para o sistema.

## 2 JOGOS ELETRÔNICOS: CONCEITOS E EVOLUÇÃO

Este capítulo apresenta uma introdução sobre jogos eletrônicos necessária para familiarizar o leitor com este assunto que é a base deste trabalho. Inicialmente são mostrados os princípios básicos dos jogos eletrônicos, seguido de uma breve passagem sobre a evolução que esta área obteve nos últimos anos. Por fim, é realizada uma apresentação do estado da arte do foco específico desta pesquisa, que são os jogos para dispositivos móveis.

### 2.1 *CONCEITOS BÁSICOS*

Um jogo eletrônico pode ser definido como um sistema interativo que permite ao usuário experimentar, sem riscos, uma situação de conflito. Quase sempre, como pano de fundo dessa situação existe um enredo, as regras que o jogador deve seguir e os objetivos que ele deve se esforçar para atingir [3][28].

Tratando especificamente de jogos para computadores, este é um software com algumas características bastante particulares. Dentre estas, uma que chama a atenção é o contínuo processamento. Em geral, mesmo quando nenhuma interferência do jogador sobre o jogo ocorre, existe um processamento necessário para a atualização do estado do jogo e para sua apresentação para o jogador. Essa apresentação tenta retratar da forma mais realista possível a situação vivida no jogo. Isso implica, por exemplo, em estar constantemente desenhando imagens na tela, em tocar sons, vibrar algum dispositivo em contato com o jogador, enfim, utilizar todos os mecanismos possíveis de *feedback* para que o jogador sinta-se “dentro do jogo”.

#### 2.1.1 *Componentes de um Jogo*

Em termos de componentes, os jogos de computador são constituídos por três principais partes: o enredo, uma interface interativa e um motor (*framework*).

O *enredo* define o tema, a trama, e os objetivos do jogo, segundo os quais os usuários devem se esforçar para realizar uma série de atividades. A definição da trama não envolve só criatividade e pesquisa sobre o assunto, mas a interação com pedagogos, psicólogos, roteiristas, cineastas e especialistas no assunto a ser focado [3].

A *interface interativa* controla a comunicação entre o motor e o usuário, servindo de apresentação do jogo para este último. O desenvolvimento da interface envolve aspectos artísticos, cognitivos e técnicos. O valor artístico de uma interface está na capacidade que ela tem de valorizar a apresentação do jogo, atraindo usuários e aumentando a sua satisfação ao jogar. O aspecto cognitivo está relacionado à correta interpretação da informação pelo usuário. O aspecto técnico envolve fatores como desempenho e portabilidade [3].

O *motor* de um jogo é o seu sistema de controle, o mecanismo que coordena a reação do jogo em função das ações dos usuários e até dos próprios objetos do jogo que sejam dotados de um certo grau de inteligência e autonomia, os chamados NPCs (de *Non Player Character*). O motor é também o responsável por apresentar para o jogador o estado do jogo a cada instante, o que inclui a criação e desenho da interface visual do jogo, bem como a execução de sons e utilização de outros mecanismos de *feedback*.

### 2.1.2 *Ciclo de Atividades de um Jogo*

Como apresentado anteriormente, os jogos são aplicações que necessitam de um contínuo processamento. Para satisfazer tal requisito, um ciclo de atividades que se repete a todo instante enquanto o jogo é executado faz-se necessário. As atividades mais importantes deste ciclo são basicamente três:

- Atualização do estado do jogo;
- Apresentação do estado interno para o jogador;
- Sincronização.

No que diz respeito à *atualização* do estado do jogo, o principal objetivo é modificar o estado interno do jogo baseado na ocorrência de uma série de eventos e na lógica (regra) do jogo. O *estado interno* é a representação computacional de todos os objetos presentes no jogo. Por exemplo, em um jogo de corrida de carros, os carros são objetos, assim como a pista, a chuva, o vento, os obstáculos, os carros controlados pelo computador (NPCs), e outros. Já os *eventos* que ocasionam a atualização do estado interno são os mais variados possíveis e dependem muito de cada jogo. Podendo variar desde apenas o movimento do mouse, teclado ou *joystick* pelo jogador, até detecção de colisão entre objetos do jogo, recebimento de dados pela Internet, reconhecimento de comandos de voz, utilização de luvas e acessórios especiais, etc.

Em se tratando da fase de *apresentação*, o objetivo é dar um *feedback* para o jogador do estado interno do jogo. Como o estado interno pode mudar a todo instante, esta fase de apresentação serve basicamente para informar tais mudanças, pelo menos aquelas que o usuário supostamente deve perceber. Esta etapa do ciclo pode ser muito rica em detalhes quando se deseja dar maior sensação de realismo. Como comentado, o estado interno é uma representação computacional dos objetos do jogo (números indicando velocidade, altura, largura, posição, aceleração, etc.), cabendo à esta fase de apresentação dar uma representação humana e natural para este estado interno. Esta necessidade de realismo implica na utilização de diversos mecanismos de *feedback*. Indo desde o desenho de belíssimas imagens na tela e execução de sons até a utilização de músicas de acordo com o ambiente, renderização de desenhos tridimensionais, vibração de dispositivos em contato com o jogador, tais como luvas, roupas, *joysticks* e outros.

A fase de *sincronização* tem o objetivo de fazer com que o ciclo do jogo ocorra a uma taxa de apresentação satisfatória e o mais constante possível, diminuindo a influência da variação de desempenho apresentada pelos computadores. A idéia principal de tal sincronização é para que os jogos, em especial aqueles que retratam situações do mundo real, tais como os simuladores de vôo e jogos de corrida de carros, sejam visualizados como se fossem um filme.

A Figura 2-1, ilustrada a seguir, mostra o ciclo do jogo e resume as três atividades explicadas anteriormente.

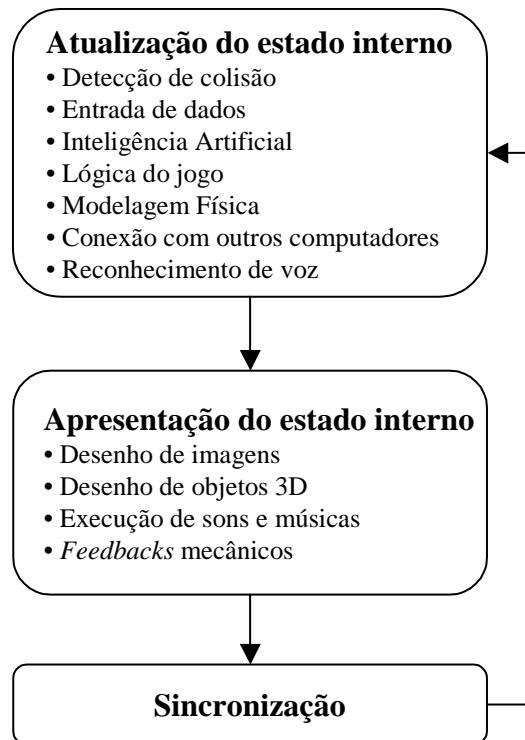


Figura 2-1: Ciclo de atividades de um jogo

## 2.2 EVOLUÇÃO DOS JOGOS ELETRÔNICOS

Certamente, um dos aspectos mais marcantes da computação é a rapidez de sua evolução. Esta evolução é marcada por constantes lançamentos de computadores cada vez mais velozes, ou por novas tecnologias, como a Internet, que se aproveitam desse avanço dos hardwares para construir soluções mais inovadoras. Isto tem sido um dos pontos cruciais para o constante crescimento que a área de jogos eletrônicos tem experimentado. A seguir, algumas das evoluções mais marcantes desta área são exploradas.

### 2.2.1 Tecnológica

Um fato que sempre marcou a área de jogos é a utilização total das mais recentes tecnologias e recursos multimídia disponíveis. Motivadas pelo excelente ganho de qualidade proporcionado por recursos de aceleração gráfica, sons 3D, e outros, os fabricantes de jogos estão sempre em busca de novidades a serem exploradas. Algumas das tecnologias atualmente em uso são:

- Placas de aceleração gráfica;
- Áudio interativo e 3D;



- Inserção de vídeos;
- Mecanismo de *feedback*;
- Bibliotecas multimídia;
- Diversidade de plataformas.

Com o baixo custo atual das *placas de aceleração gráfica*, a maioria dos jogos já adotam a sua utilização como requisito mínimo. Essas placas surgiram para atender a crescente demanda por recursos de computação gráfica, especialmente aqueles 3D [3]. Elas evitam uma sobrecarga da CPU com processamento necessário para cálculo e renderização de gráficos 3D, com isso, jogos inteiramente em ambientes 3D que antes eram muito lentos, passaram a ser viáveis.

Enquanto durante um bom tempo os jogos eram acompanhados de simples efeitos sonoros, atualmente a utilização de sons de boa qualidade e coerentes com o ambiente do jogo tornou-se um fator fundamental para aumentar a imersão do jogador e sucesso de alguns jogos. Visando melhorar o realismo dos jogos, uma técnica bastante comum atualmente é a utilização de áudio interativo [28]. Este recurso consiste em modificar as propriedades do som sendo executado de acordo com o estado do jogo. A interatividade varia desde a aplicação de simples efeitos aos sons, tais como mudança de volume, espectro, velocidade, e outros, até a execução de trilhas sonoras dinâmicas que acompanham o andamento do jogo. Um outro recurso sonoro também muito aplicado é a reprodução de sons 3D, que procuram reproduzir os sons do jogo levando em conta a espacialização das fontes sonoras [28].

Um outro resultado do uso de tecnologia em jogos é a utilização de pequenos vídeos que se encaixam no ambiente do jogo [28]. Além de aumentar consideravelmente o realismo gráfico dos jogos, especialmente se os vídeos forem gravações de situações reais, os vídeos eliminam um enorme trabalho de programação caso a mesma sequência tivesse que ser realizada por animação gráfica. Ao utilizar tais vídeos, a produção dos jogos passa a ser, em parte, tão complexa quanto a produção de filmes, já que muitos desses filmes envolvem artistas reais.

A utilização de dispositivos de *feedback* físico tem sido uma adição recente aos jogos eletrônicos. Em vez de apenas desenhar imagens e reproduzir sons, alguns jogos já são compatíveis com alguns *joysticks* especiais que vibram, ficam mais/menos sensíveis, de acordo com comandos enviados pelo jogo. O uso desses *joysticks* já é

bastante popular em jogos de carro e avião, onde o manche ou direção tem as propriedades alteradas em resposta a eventos acontecidos no ambiente do jogo. No caso da direção, por exemplo, a sua sensibilidade aumenta quando o carro está a alta velocidade, assim como treme quando o carro derrapa, etc.

O uso de *bibliotecas multimídia* é atualmente essencial ao desenvolvimento de jogos [5]. Bibliotecas como DirectX [25] e OpenGL [30] são largamente utilizadas por permitirem a integração dos jogos com os diversos tipos e marcas de periféricos utilizados nos PCs (placas de som, vídeo, *joysticks*, etc.), bem como prover diversos recursos úteis para o desenvolvimento e execução eficiente dos jogos [5].

Um outro grande avanço tecnológico a favor dos jogos decorre da *diversidade de equipamentos* eletrônicos que vem sendo criada. Um exemplo é o novo videogame da Microsoft, Xbox [26], a ser lançado em novembro de 2001. Visando entrar no mercado de videogames, a Microsoft adotou uma estratégia de atender as principais demandas dos fabricantes de jogos para consoles que não eram atendidas pelos principais videogames concorrentes. Após encontrar os requisitos dos maiores fabricantes, a Microsoft iniciou o desenvolvimento de uma máquina capaz de atender todas as demandas, de forma que os jogos do Xbox sejam capazes de realizar operações jamais possíveis antes. Outra categoria de dispositivos que tem despertado grande atenção aos desenvolvedores de jogos são os dispositivos móveis, tais como, celulares, *paggers* e PDAs<sup>1</sup>, devido a um enorme crescimento e evolução dessa categoria presenciado nos últimos anos. Tal avanço, por exemplo, torna viável hoje, pela primeira vez, o desenvolvimento de programas para esses dispositivos móveis pela comunidade interessada, de forma similar ao desenvolvimento para PCs. Essa situação, como será comentada na seção 2.3, contrasta com a tradição nesse segmento de mercado, em que os softwares eram desenvolvidos para plataformas proprietárias.

### 2.2.2 Acadêmica

Durante um bom tempo, realizar pesquisas acadêmicas na área de jogos eletrônicos trazia uma série de preconceitos. Na grande maioria das vezes essa situação era resultado de uma carência cultural das instituições de pesquisa, juntamente com a escassez de livros, congressos relacionados e, principalmente, especialistas na área

---

<sup>1</sup> PDA é a abreviação de Personal Digital Assistant, que é um tipo sofisticado de agenda eletrônica.

envolvidos em meios acadêmicos. Conseqüentemente, grande parte das soluções para problemas da área eram sigilos industriais, ficando longe do ambiente acadêmico.

Ao longo do tempo, felizmente, começou-se a observar que a interdisciplinaridade da área era um campo riquíssimo para pesquisas. Em poucas áreas da computação é possível explorar tantos domínios de conhecimento ao mesmo tempo. A seguir seguem alguns exemplos de áreas da computação que podem ser envolvidas:

- Computação Gráfica;
- Inteligência Artificial;
- Engenharia de Software;
- Computação Musical;
- Redes de Computadores;
- Algoritmos e Estruturas de Dados;
- Modelagem Física.

Uma das áreas de conhecimento mais requisitadas em jogos é a *Computação Gráfica*. Desde o surgimento dos jogos eletrônicos, quando os sons eram muito simples, não existiam vídeos e outras formas de apresentação do jogo, as técnicas da computação gráfica para o desenho e animação de *sprites*<sup>1</sup> já era a base para jogos em duas dimensões [3]. Hoje, com a crescente melhoria da qualidade dos jogos, a computação gráfica detém os maiores desafios da indústria de jogos, especialmente quando se trata da sua utilização em jogos 3D. Esses jogos exigem avançadas técnicas de renderização de objetos tridimensionais, iluminação de cenas, determinação de objetos não visíveis em um ambiente de acordo com o local do observador, e várias outras [82].

As técnicas da *Inteligência Artificial* são uma necessidade básica para jogos que tenham objetos controlados pelo computador (os NPCs). Estes objetos tornam os jogos muito menos monótonos, seja ao desafiarem o jogador ou a ajudá-lo em torno dos objetivos do jogo. Inicialmente apenas simples máquinas de estados eram suficientes para modelar o comportamento de um NPC. No entanto, em busca de jogos cada vez mais surpreendentes, conceitos muito mais avançados já são aplicados fortemente em alguns estilos de jogos. Áreas como agentes inteligentes, aprendizagem, planejamento

---

<sup>1</sup> Representação gráfica de um objeto do jogo através de imagens retangulares com regiões transparentes.

estratégico e métodos de busca já fazem parte dos mais tradicionais jogos eletrônicos [81].

A utilização da *Engenharia de Software* em jogos é uma tendência recente. Inicialmente, devido às exigências de performance sempre impostas aos jogos, as equipes de desenvolvimento ignoravam conceitos como reusabilidade e modularidade [6], levando ao ponto extremo de certos jogos serem desenvolvidos diretamente em linguagem de máquina, pois utilizar o Sistema Operacional era lento demais [5]. Com o passar o tempo, o crescimento considerável do tamanho, complexidade e dinheiro envolvidos nos projetos de jogos, esta área se sensibilizou da importância da utilização de linguagens de alto nível e dos conceitos da engenharia de software. Hoje, já é uma prática comum a utilização de *frameworks* e bibliotecas multimídia para o desenvolvimento de jogos, como será detalhado no próximo capítulo, quando será apresentada esta mudança metodológica ocorrida no desenvolvimento de jogos eletrônicos [5].

A busca por uma boa apresentação dos jogos eletrônicos também envolve a área da *Computação Musical*. Durante um bom tempo os jogos eram acompanhados de simples efeitos sonoros e músicas repetitivas como fundo musical. Com o avanço da computação, assim como o ocorrido com a computação gráfica, o uso de som em jogos passou por sérias evoluções. Hoje, técnicas como a geração automática de trilhas sonoras dinâmicas, que dependendo de situações e eventos ocorridos no jogo são criadas automaticamente, bem como a reprodução de sons 3D, que procuram reproduzir os sons do jogo levando em conta a espacialização das fontes sonoras, já são práticas comuns aos jogos eletrônicos [28].

Fruto do advento da Internet e das redes locais, a área de jogos eletrônicos passou também a envolver a área das *Redes de Computadores*. Quando os primeiros jogos em rede surgiram, o seu uso era limitado a jogos bem simples, em geral compartilhados por duas pessoas e requerendo muito pouca troca de dados. Atualmente o cenário é bem diferente e existe uma série de desafios a serem vencidos. Em jogos em tempo real, por exemplo, existe a contínua necessidade de troca de informações e consistência entre todos os usuários que estiverem jogando [28]. A maneira em que esses dados são trocados, bem como os protocolos utilizados, são de grande importância para a viabilidade do jogo, principalmente no que diz respeito a escalabilidade.

A área de *Algoritmos e Estruturas de Dados*, apesar de ser apenas base para o desenvolvimento de software em geral, tem uma importância fundamental para a

existência de aplicações com requisitos de performance tão extremos como os jogos eletrônicos. Desde o surgimento dos jogos, seja devido aos fortíssimos requisitos de velocidade e memória das plataformas de execução dos jogos, ou devido aos requisitos extremos de alguns jogos, os algoritmos e estruturas de dados convencionais utilizados em outras áreas da computação tornam-se inaplicáveis. Com isso, atualmente, especialmente em se tratando de jogos 3D, existem uma série de soluções e pesquisas em desenvolvimento dedicados para essa área de conhecimento [5].

Em muitos estilos de jogos, principalmente nos simuladores, há uma grande necessidade de introduzir nos jogos a *Modelagem Física* de certas entidades existentes no mundo real [85]. Apesar de inicialmente estes modelos terem sido bastante simples e muito artificiais, atualmente a busca por um maior realismo tem levado a um grande avanço da quantidade de modelos físicos sendo implementados em alguns jogos. Por exemplo, em um jogo de corrida de carros, a modelagem física do processo de derrapagem do carro em uma curva tem tornado os jogos muito mais realistas do que quando simplesmente fazia-se o carro efetuar uma trajetória curvilínea. De forma análoga, a modelagem da gravidade e condições climáticas, por exemplo, são fatores essenciais para o realismo atualmente encontrado em simuladores de voo. O simulador de voo *Microsoft Flight Simulator* vai ainda mais além, sendo capaz de comunicar-se através da Internet com um banco de dados meteorológico mundial para simular as mesmas condições climáticas presentes por onde o avião esteja voando.

Além das áreas citadas, o desenvolvimento de jogos pode envolver também uma grande interação com diversas áreas fora da computação. Em geral, áreas como Artes, Cinema e Música são bastante requisitadas. Além disso, especialmente durante a fase de elaboração do enredo do jogo, certos jogos podem necessitar também do envolvimento da Psicologia, Educação, História e Geografia [28].

Em razão de toda esta interdisciplinaridade, dentro e fora da computação, a visão acadêmica em torno da área de jogos eletrônicos tem mudado bastante. Enquanto numa primeira fase, a área não era muito explorada na academia, hoje, diversas universidades do mundo já dedicam cursos sobre desenvolvimento de jogos, vários livros já são dedicados para essa área de conhecimento e já existe um bom número de congressos sobre o tema. O CIN é o pioneiro, na América Latina pelo menos, a implementar um esforço com a introdução da disciplina Projeto e Implementação de Jogos [7], no entanto, ainda vai levar um tempo até ter-se uma boa cultura local sobre essa área.

Enquanto isso, alguns trabalhos acadêmicos foram desenvolvidos gerando teses de mestrado [8][9] e várias publicações, inclusive uma sobre esta pesquisa [28].

### ***2.2.3 Comercial***

Se por um lado a área de jogos eletrônicos ainda está em processo de disseminação na academia, pelo lado comercial esta área já é assunto de sucesso há muitos anos. Desde a década de 70, quando os primeiros jogos eletrônicos surgiram, o crescimento é contínuo. Hoje os números são surpreendentes. Pesquisas baseadas no mercado norte-americano mostram que 60% das pessoas acima de seis anos de idade (uma média de 145 milhões de pessoas) jogam em PCs ou videogames [55]. Em 2000, foram vendidos 219 milhões de jogos, o equivalente a dois jogos para cada casa dos EUA, sendo 90% dessa quantidade adquirida por pessoas com mais de 18 anos [55]. Com esses números, o mercado de jogos terminou o ano de 2000 gerando receitas de US\$ 6,2 bilhões, número maior do que a poderosa indústria cinematográfica [24]. Em pesquisa e desenvolvimento, a área recebeu US\$ 2,7 bilhões de investimento em 2000, um número maior até mesmo do que o investimento na indústria farmacêutica no mesmo ano [24].

Motivadas por esse grande sucesso, algumas empresas no Brasil começam a se especializar em desenvolvimento de jogos (ao contrário de empresas de distribuição de produtos importados). Atualmente existem seis, sendo duas delas em Pernambuco – Jynx [23] e Joystudios [22], ambas compostas por alunos e ex-alunos do CIn-UFPE.

## ***2.3 A ERA DOS JOGOS PARA DISPOSITIVOS MÓVEIS***

Pesquisas estimam que mais de 80% das pessoas da Europa utilizarão celular em 2005, e que mais de 1 bilhão de pessoas em todo o mundo utilizarão Internet em dispositivos móveis em 2003 [48]. Tais previsões, aliadas com o sucesso comercial dos jogos eletrônicos comentados na seção 2.2.3, demonstra o potencial mercadológico dos jogos para dispositivos móveis [51]. Essas perspectivas têm feito vários investidores de risco e operadoras de telefonia móvel apostar fortemente nesse mercado através de parcerias e criação de novas empresas no setor [52].

Hoje, a diversidade de equipamentos eletrônicos portáteis em que é possível a utilização de jogos é imensa. São diversos modelos de PDAs, celulares, *paggers*, videogames portáteis e vários outros equipamentos sendo lançados a cada dia permitindo esta forma de entretenimento. É interessante notar que apesar dessa

diversidade, a comunidade de desenvolvedores não pode, na maioria dos casos, construir aplicativos para tais aparelhos, apenas os próprios fabricantes dos dispositivos podem fazê-lo.

No entanto, após vários anos de arquiteturas fechadas e incompatíveis, os fabricantes de dispositivos como celulares, *paggers* e PDAs uniram-se em torno de viabilizar o desenvolvimento de aplicações para esses dispositivos da mesma forma que ocorre nos computadores pessoais. Em PCs, um software é desenvolvido para um dado sistema operacional, ou máquina virtual, como ocorre com Java, e é então capaz de ser executado em qualquer computador, não importando o fabricante do mesmo, bastando a presença do sistema operacional e/ou máquina virtual necessária. No entanto, até essa viabilidade chegar aos dispositivos móveis, por meio da tecnologia J2ME, outras formas de desenvolvimento de software, apesar de limitadas, foram introduzidas para permitir a criação de aplicações para tais dispositivos. Estas soluções serão explicadas a seguir, onde as soluções prévias a J2ME, e ainda em uso, mostram o contexto tecnológico atual da área.

### *2.3.1 Soluções dependentes de plataforma*

Já é possível, há um bom tempo, o desenvolvimento de programas para alguns dispositivos, embora sendo dependentes dos modelos e fabricantes dos aparelhos. O caso mais comum é o dos PDAs e Handhelds<sup>1</sup>, exemplificados na Figura 2-2 por modelos da Palm e HP, que possuem sistemas operacionais que permitem a instalação de novos programas. Utilizando compiladores específicos, o desenvolvimento de programas para tais dispositivos, geralmente utilizando a linguagem C, está ao alcance de qualquer desenvolvedor [41].

O grande problema desta abordagem é a falta de portabilidade, já que um programa desenvolvido para um sistema operacional utilizado em um determinado dispositivo, tal como o PalmOS [39], não pode ser utilizado (sem alteração ou recompilação) em um outro dispositivo baseado em um sistema operacional diferente, tal como o WindowsCE [40]. A Sun tentou contornar este problema com a introdução de versões reduzidas da plataforma JAVA, porém, como será mostrado na seção 4.1, tais soluções não tiveram a aceitação esperada.

---

<sup>1</sup> Espécie de computador portátil menor que um *laptop*



Figura 2-2: Dois PDAs e um Handheld, respectivamente

### 2.3.2 WAP

A primeira tecnologia capaz de trazer a portabilidade foi WAP (de *Wireless Application Protocol*) [42], criada para permitir o acesso a Internet através de dispositivos sem fio, especialmente os celulares. O seu uso requer que as informações na Internet sejam formatadas para um formato que possa ser mostrado na tela dos aparelhos portáteis, sendo basicamente necessária a conversão de documentos em formato HTML (*hypertext markup language*) para WML (de *wireless markup language*).

A tecnologia WAP, apesar de permitir que empresas possam desenvolver sistemas de informação para os dispositivos móveis, é muito limitada para o desenvolvimento de aplicações, já que ela não conta com uma linguagem de programação. Apesar de alguns *scripts* poderem ser adicionados às páginas, o desenvolvimento de jogos, dado o alto grau de interatividade e recursos gráficos necessários, torna-se muito difícil. Devido a esses problemas, os jogos em WAP são muito simples, baseados apenas em textos ou gráficos muito pobres. Além disso, desapontam pela lentidão e custo associados, já que é necessário que o aparelho esteja permanentemente conectado e que a troca de dados seja efetuada com o servidor WAP a cada evento ocorrido para que o estado do jogo seja atualizado. Mesmo assim, vários jogos para WAP têm tido sucesso, tal como o jogo de perguntas e respostas Buzztime [49], que em apenas 49 dias de lançamento conquistou mais de 130.000 usuários e foi responsável por mais de 1 milhão de minutos de ligações telefônicas [50].

### 2.3.3 i-Mode

A tecnologia i-Mode (de *information-mode*) [44] é semelhante à WAP. Foi criada no Japão, em 1999, pela NTT DoCoMo [45], empresa de telefonia móvel líder no país, com o mesmo intuito de fornecer acesso à Web pelos dispositivos móveis. Porém, o diferencial do i-Mode é que ele permite que o aparelho fique conectado o tempo todo



à Internet, sendo tarifado apenas pela quantidade de dados que seja transferida através do aparelho, ao contrário do WAP, onde o usuário paga pelo tempo que durar a ligação telefônica.

Por se tratar da mesma idéia do WAP, de basicamente permitir a visualização de conteúdo, as limitações encontradas no modelo i-Mode para o desenvolvimento de jogos são as mesmas presentes no WAP.

#### 2.3.4 SMS

Um dos serviços bastante utilizados atualmente em celulares é o recebimento e envio de mensagens de texto, similar à principal funcionalidade dos *paggers*. Por trás deste recurso está a tecnologia SMS (de *Short Messaging Service*) [43], responsável por um fluxo de mais de um bilhão de mensagens por mês na Europa [41].

O SMS é uma tecnologia muito limitada, permitindo que apenas textos de no máximo 160 caracteres sejam trocados entre dispositivos. Apesar disto, algumas empresas já desenvolveram jogos baseados apenas nessa troca de mensagens [46]. Alguns desses jogos são bem simples, como o jogo da forca, enquanto que em outros vários usuários ficam trocando mensagens para a criação de comunidades para competirem entre si em busca de certos objetivos.

#### 2.3.5 J2ME

Todos os problemas abordados nas tecnologias explicadas anteriormente implicam numa série de desafios aos desenvolvedores de software para dispositivos móveis. Um dos mais críticos é a ausência de *portabilidade*. Se um dado programa ou sistema de informação tiver que ser utilizado por um celular e PDA, por exemplo, é necessário que diversas versões sejam desenvolvidas para levar em conta as características de cada marca de celular ou PDA, pois não existe uma plataforma de execução de aplicações genérica. Além disso, as tecnologias WAP, SMS e i-Mode apresentam grandes *restrições*, não permitindo o desenvolvimento de aplicações complexas.

A solução para estes problemas é a recente tecnologia Java 2 Micro Edition (J2ME) [13], que consiste de uma linguagem de programação e máquina virtual Java simplificada e voltada para dispositivos móveis. O problema da *portabilidade* é solucionado através da implementação da máquina virtual para os diversos sistemas operacionais utilizados nos dispositivos, de forma semelhante ao que ocorre com a

tecnologia Java para PCs, enquanto que as *restrições* das demais tecnologias é largamente melhorada por um conjunto de recursos oferecidos por J2ME. Destacando-se a capacidade de criação de ricas interfaces gráficas, conexão remota com outros dispositivos, mecanismos de entrada e saída de dados e geração de programas a serem instalados nos dispositivos.

Com o surgimento de J2ME uma enorme diversidade de aplicações tornam-se viáveis. Em vez de simples navegação por sistemas de informação utilizando WAP ou i-Mode, onde o usuário tem que estar conectado (e é tarifado) o tempo todo, ou as trocas de mensagens instantâneas via SMS, J2ME é uma verdadeira linguagem de programação para dispositivos móveis. Por isso, permite a criação de programas a serem instalados nos dispositivos por qualquer usuário, seja por meio de *download* via cabo serial conectado a um PC ou via a própria rede de comunicação sem fio do aparelho. Esta capacidade de instalação de programas, aliada à crescente utilização de dispositivos móveis em todo o mundo, torna os dispositivos móveis um mercado tão promissor para a venda de aplicações quanto o de PCs.

Diante deste cenário, a escolha natural adotada neste trabalho para desenvolvimento do wGEM foi a tecnologia J2ME. O capítulo 4 é dedicado aos detalhes desta linguagem, o que mostrará de fato o seu enorme potencial. Entretanto, como será apresentado no capítulo 5, para poder produzir jogos na qualidade e velocidade demandadas, é preciso superar algumas limitações encontradas em J2ME, assim como construir ferramentas de apoio ao desenvolvimento.

## 2.4 CONCLUSÕES

Como visto durante este capítulo, a área de jogos eletrônicos é riquíssima para pesquisas, tem tido um enorme sucesso comercial e tem explorado (muitas vezes provocado) grandes avanços tecnológicos na computação. Dentre esses avanços, um de especial importância para este trabalho é a atual capacidade de desenvolvimento de aplicativos para dispositivos móveis através da tecnologia J2ME. Esperamos ter deixado claro o potencial dessa tecnologia diante do cenário atual encontrado na área de desenvolvimento para dispositivos móveis. Por se tratar do assunto que representa a viabilidade fundamental para nossa pesquisa, J2ME será apresentada em detalhes no capítulo 4.

Um resultado importante da evolução da área de jogos é abertura de algumas informações antes tratadas como segredos industriais. Dessa forma, alguns detalhes

sobre o processo de desenvolvimento de jogos e os principais requisitos destes programas podem ser enumerados. Tais informações, necessárias para a implementação dos componentes de um jogo que foram comentados neste capítulo, bem como a implementação do ciclo de atividades apresentado, serão apresentadas no capítulo seguinte, onde as tendências atuais no desenvolvimento de jogos serão apresentadas.



## 3 DESENVOLVIMENTO DE JOGOS

Durante este capítulo serão abordados diversos assuntos sobre o desenvolvimento de jogos para computadores. Inicialmente será apresentada a evolução histórica no processo de desenvolvimento de jogos, seguida de uma apresentação simplificada do que é atualmente o projeto de um jogo e suas etapas. Uma grande ênfase será dada à mudança cultural que houve em torno da aderência às boas práticas da engenharia de software e da importância da reusabilidade, que é realizada com o uso de *frameworks*. Por fim, dada a importância para esta pesquisa, um estudo mais detalhado sobre *frameworks* de desenvolvimento de jogos é apresentado, bem como uma classificação das ferramentas de apoio que em geral são utilizadas.

### 3.1 HISTÓRICO

Fazendo um retrospecto sobre o desenvolvimento de jogos de computadores, dos anos 80 até hoje houve uma grande mudança na metodologia de desenvolvimento. Em função das limitações encontradas nos computadores daquela época, o processo de programação envolvia uma constante preocupação em torno das restrições de processamento e memória existentes. Desenvolver um jogo era um verdadeiro trabalho artesanal. Para se ter uma idéia, todos os jogos tinham que ser desenvolvidos em *Assembler*, pois os programas produzidos por compiladores C eram lentos e grandes demais para a produção de jogos, enquanto que um programa desenvolvido em *Assembler* tinha sua performance e tamanho final sob total controle do programador. Esse método de desenvolvimento, interagindo diretamente com o hardware para obter a melhor performance possível, ficou conhecido pela expressão “*writing directly to the metal*” [5]. Naturalmente, isso trazia um enorme problema de portabilidade entre plataformas e dificuldade de produção dos jogos.

Com o avanço dos computadores e lançamento de compiladores cada vez mais avançados, a indústria aos poucos começou a adotar a linguagem C para o desenvolvimento de jogos. O primeiro jogo de sucesso a adotar essa estratégia foi o famoso *Doom*, que em 1993 provocou o início da era do desenvolvimento de jogos em C. No entanto, a mentalidade era ainda desenvolver cada jogo inteiramente desde o

início, com praticamente nenhuma reusabilidade de código construído anteriormente, especialmente de terceiros. Situação que ficou referenciada pela mentalidade NBH (de *Not Built Here*): o código dos outros é mais lento, pior e não presta [5].

Durante a era dos jogos em C, a principal plataforma de execução dos jogos era o sistema operacional DOS da Microsoft. Apesar do surgimento do sistema operacional Windows3.1, a execução dos jogos no ambiente Windows era lenta demais, o que fazia os jogos continuarem sendo executados no DOS. Mesmo com o lançamento do Windows95 a situação se manteve. Visando contornar esse problema a Microsoft desenvolveu o DirectX, uma biblioteca que permitia que os desenvolvedores de jogos tivessem acesso direto a certos recursos do computador sem a interferência do Windows. Além disso, o DirectX tornava a execução dos jogos mais rápida em ambiente Windows do que em DOS e garantia a compatibilidade com os diversos dispositivos presentes nos PCs (placas gráficas e de som, joysticks, etc.), o que evitava o desenvolvimento de inúmeros *drivers* pelos fabricantes de jogos. Com esse resultado, a Microsoft conseguiu não só a migração da indústria para o Windows95, mas também para o seu compilador Visual C++, que foi lançado na mesma época e repleto de otimizações que produziam programas menores e mais rápidos que qualquer outro compilador [5].

Essa situação promovida pela Microsoft, em se tratando do ambiente de desenvolvimento e execução de jogos, ainda retrata o cenário atual. O DirectX já está em sua versão 8 e o Visual C++ na versão 6. Ambos continuam sendo o ambiente padrão de desenvolvimento utilizado no mercado. A mudança mais radical que houve durante esse tempo foi o fim da mentalidade NBH. Da mesma forma que o uso dos sistemas operacionais e dos compiladores aumentaram consideravelmente a quantidade de operações realizadas automaticamente, a indústria de jogos passou a considerar essa situação também para reutilização de *frameworks* de desenvolvimento de jogos. A principal razão para essa mudança foi a pressão do mercado por novos jogos a uma grande velocidade. Como os jogos antigamente eram muito mais simples, as empresas ainda eram capazes de produzir bons resultados em pouco tempo, mesmo desenvolvendo cada jogo inteiramente. Porém, com o grande avanço tecnológico presenciado, hoje é impossível para uma empresa realizar, no tempo exigido, toda a implementação de computação gráfica, som, música, modelagem física, inteligência artificial e vários outros trabalhos envolvidos em um jogo sem a reutilização de *frameworks*. O uso de *frameworks*, bem como bibliotecas, pode ser visto como

abstrações do hardware ou sistema operacional que provêm um ambiente ideal à execução e ao desenvolvimento de um jogo. Essa visão é ilustrada na Figura 3-1.



Figura 3-1: Camadas de software para execução de um jogo

Como resultado dessa evolução metodológica no processo de desenvolvimento, os projetos de jogos passaram a ser muito mais organizados. Atualmente uma grande atenção é dada à arquitetura do jogo, que passou a levar em conta diversos pontos fundamentais da Engenharia de Software como reusabilidade, modularidade e abstração [6]. Com isto, a linguagem de programação também sofreu evoluções. Hoje, o padrão adotado é a linguagem C++, em razão dos benefícios proporcionados por ser uma linguagem orientada a objetos e por contar com o apoio do DirectX e da eficiência do ambiente Visual C++.

Em razão dessa evolução dos projetos de jogos e devido ao crescente mercado, o crescimento das equipes de desenvolvimento foi um fato inevitável. Além disso, a sua estrutura foi fortemente evoluída para lidar com a crescente necessidade de organização entre os membros, papéis mais bem definidos e canais de comunicação entre os grupos. Para se ter uma idéia dessa mudança, a Tabela 3-1 ilustra, de uma forma bastante irreverente, como eram formadas e conduzidas as equipes de desenvolvimento de jogos nos anos 80.

1. Encontre 5 programadores (Geral, IA, Gráficos, Som, etc.);
2. Eleja um *hacker* como líder;
3. Coloque-os numa sala pequena com alguns artistas a disposição;
4. Deixe cozinhar por 18 meses regando-os a pizza e coca-cola;
5. Dê um pouco mais de cozimento, e .... pronto!

Tabela 3-1: Receita básica para equipes de desenvolvimento de jogos nos anos 80 [5]

Hoje a situação é bem diferente. Um jogo profissional pode exigir uma equipe de centenas de pessoas. O jogo *StarCraft*, por exemplo, tinha uma equipe de 228 pessoas, sendo 53 em gerência, atividades comerciais e *design*, 16 em programação, 108 em artes e 51 em testes. Vale salientar que muitas dessas pessoas, da equipe de artes e testes, por exemplo, não participam do desenvolvimento do jogo do início ao fim. Na verdade elas têm um papel bem definido na empresa em que trabalham e participam neste papel em

diversos jogos sendo desenvolvidos ao mesmo tempo. Essa idéia, caracterizada como uma fábrica de software, tem sido aplicada em algumas empresas de jogos para tentar aproveitar ao máximo o paralelismo entre as equipes [5]. A Tabela 3-2 resume os principais papéis das pessoas envolvidas numa fábrica de jogos segundo nossa adaptação da configuração proposta por Andrew Rollings e Dave Morris [5].

Management & Design	<b>Software Planner</b>	Detalha os requisitos técnicos e esforço esperado para construção do jogo com base no <i>game design</i> (ver seção 3.2.1). Trabalha em conjunto com o <i>Game Designer</i> e <i>Lead Architect</i> .
	<b>Lead Architect</b>	Especifica os módulos do jogo em função dos requisitos técnicos detalhados pelo <i>Software Planner</i> .
	<b>Project Manager</b>	Gerencia interação entre membros da equipe e produz um planejamento para o desenvolvimento do jogo baseado nos esforços identificados pelo <i>Software Planner</i> e <i>Lead Architect</i> .
	<b>Game Designer</b>	Produz o documento de <i>game design</i> e trabalha na evolução do mesmo sempre que necessário.
Programming	<b>Lead Programmer</b>	Coordena a equipe de programação para garantir os prazos e a qualidade do jogo. É a interface de comunicação de sua equipe com o <i>Project Manager</i> para reportar problemas e estado do projeto, mas está envolvido com programação em 70% do tempo.
	<b>Programmer</b>	Implementa os módulos do jogo de acordo com as especificações definidas pelo <i>Software Planner</i> e <i>Lead Architect</i> . Pode ser especializado em áreas como IA, música, redes, etc, e trabalhar em mais de um projeto.
Art	<b>Lead Artist</b>	Papel semelhante ao <i>Lead Programmer</i> : detalha, atribui, faz ponte e cria.
	<b>Artist</b>	Responsável por implementar a arte gráfica do jogo (2D, 3D, cenário, animações digitais, etc.). Posição semelhante ao de programação, podendo também estar associado a vários projetos.
Music	<b>Musician</b>	Cria as trilhas sonoras. Geralmente trabalham isolados, mas quando existem trilhas interativas, ocorre uma maior interação com a equipe.
	<b>Sound Effect Technician</b>	Cria os efeitos sonoros de acordo com a especificação do <i>game design</i> .
Support & Quality Assurance	<b>QA Leader</b>	Elabora planos de teste e divide as atividades entre os <i>QA Technicians</i> . Interage com o <i>Project Manager</i> para elaborar os planos e reportar os resultados obtidos.
	<b>QA Technician</b>	Realiza testes de caixa branca e caixa preta dos módulos do jogo interagindo com os desenvolvedores responsáveis.
	<b>Play Testers</b>	Testam a jogabilidade. Podem ser pessoas da própria empresa ou de fora.
	<b>Support Technician</b>	Prepara e mantém a configuração dos computadores necessária para o projeto.

Tabela 3-2: Principais papéis dos integrantes de uma fábrica de jogos



Como consequência desse crescimento das equipes, aumento de responsabilidades com o mercado, compra de *frameworks* e vários outros fatores, o custo de produção dos jogos sofreu um enorme aumento. Atualmente, o custo considerado padrão para a produção de um jogo profissional básico é de 4 milhões de dólares. No entanto, em jogos mais sofisticados como o *Microsoft Flight Simulator*, essa quantia pode atingir a marca de 200 milhões de dólares<sup>1</sup>. Naturalmente, todo esse investimento exige que os projetos sejam muito bem gerenciados e especificados para que todos os riscos sejam diminuídos. As principais características desta especificação, bem como o restante do processo de desenvolvimento é apresentado a seguir.

## 3.2 ETAPAS DO DESENVOLVIMENTO DE JOGOS

O projeto de um jogo tem certas particularidades não encontradas em projetos de software em geral. Nesta seção serão abordadas as 3 principais fases do ciclo de desenvolvimento de um jogo: *game design*, especificação e implementação do jogo e testes.

### 3.2.1 Game Design

A primeira etapa no projeto de um jogo é a criação de um documento chamado *game design*. Ao contrário do projeto de um software tradicional, que inicia com a definição de um conjunto de requisitos levantados em entrevistas e análises de um dado problema [6], em jogos existe inicialmente a definição de um documento muito mais rico, correspondendo ao *game design*, que além de tratar dos requisitos do jogo (software), também define detalhes como o enredo, os personagens do jogo e os desenhos das principais cenas e objetos do jogo [5]. Para se ter uma idéia do nível de detalhes de um documento como esses, o número de páginas pode chegar a 200, como aconteceu com o jogo *Age of Empires*. Sendo assim, além de ser uma descrição detalhada das características do produto final, o documento de *game design* serve como uma visão unificada do jogo a ser compartilhada por toda a equipe de produção, indo desde o pessoal de marketing até desenvolvimento.

Além da importância ressaltada, um bom *game design* evita que erros sejam descobertos tardiamente, o que representa um custo muito grande para um jogo complexo que envolve uma grande quantidade de imagens, animações, vídeos, e outros

---

<sup>1</sup> Informação revelada por André Faure, diretor da divisão de jogos da Microsoft do Brasil, em palestra realizada no CESAR em setembro de 2001.

recursos. Em tais jogos, especialmente se forem envolvidos vídeos com pessoas encenando, uma mudança drástica realizada após alguns meses de trabalho pode representar a perda de toda a arte gráfica, sons, vídeos e outros produtos desenvolvidos. Em razão disso, a fase inicial no processo de desenvolvimento de um jogo é dedicada ao *game design*, uma vez que esta etapa é a mais adequada para a realização de mudanças, adição e remoção de recursos.

As principais informações de um *game design* são resumidas a seguir na Tabela 3-3. Para facilitar a compreensão, serão dados exemplos simples de cada item para o caso de um jogo de corrida de carros.

	Descrição	Exemplo: Fórmula 1
<b>Tema</b>	Explicação do que se trata o jogo através da definição do local, época, etc.	Campeonato Mundial de Fórmula 1.
<b>Estilo</b>	Definição do estilo do jogo, tal como ação, aventura, estratégia, simulação, educativo, passatempo, etc.	Ação. Subcategoria: esportes e velocidade.
<b>Backstory</b>	Explicação geralmente fornecida no início do jogo para situar o jogador com o seu objetivo, desafios, ambiente do jogo, etc.	Participar de campeonatos de fórmula 1 ao redor do mundo com o objetivo de terminar o ano com o maior número de pontos.
<b>Ambiente</b>	Define o ambiente físico onde o jogo acontece.	Autódromos de fórmula 1 espalhados pelo mundo.
<b>Personagens</b>	Representa as diversas entidades presentes no jogo com as quais o jogador interage e comanda diretamente.	Diversos carros que competem com o carro controlado pelo jogador.
<b>Regras</b>	Definição das regras do jogo.	Pontuação igual à oficial da Fórmula 1; Desgaste de pneus e consumo de combustível são simulados e requerem parada nos boxes.
<b>Aparência</b>	Definição da interface gráfica, sonora e mecânica do jogo. A especificação gráfica é definida através de rascunhos de desenhos que serão realizados futuramente. O formato de representação gráfica do jogo, 2D, 3D, etc., também é escolhido. Os sons, bem como qualquer mecanismo de <i>feedback</i> são também listados.	O jogo será 3D, com visualização igual a do piloto. Os carros terão desenhos similares aos da Fórmula 1, bem como as pistas, que serão réplicas das reais. Os sons emitidos serão sons reais de corridas e sua execução estará associada aos eventos naturais de batida, curva, etc.
<b>Níveis</b>	Todos os níveis do jogo, missões que o jogador deve cumprir ou diferentes níveis de dificuldades, devem ser especificados. Para cada nível são informados o ambiente, regras, personagens, pontuação, etc.	Os níveis corresponderão aos diversos autódromos do campeonato. Além disso, devem existir três níveis de dificuldade (fácil, médio e difícil).

Tabela 3-3: Resumo dos principais elementos de um documento de game design

### 3.2.2 Especificação da Arquitetura e Implementação do Jogo

Após a definição do *game design* segue a fase de especificação da arquitetura e implementação do jogo, embora o *game design* possa ser reconsiderado para evolução e correções. A primeira etapa é identificar, através do *game design*, todos os recursos necessários à implementação do jogo, tais como, utilização de bibliotecas multimídia, gráficos 3D, recursos de rede, inteligência artificial, modelagem física, editor de cenários, etc. Em seguida, dependendo dos recursos e prazos do projeto, a equipe pode optar por reutilizar suas próprias soluções aplicadas a jogos anteriores que sejam úteis ao projeto atual, desenvolver alguns dessas soluções ou comprar de empresas especializadas em venda de componentes.

Uma vez determinados todos os módulos a serem utilizados na construção do jogo, a especificação da arquitetura é realizada. Devido ao enorme reuso adotado atualmente pela indústria de jogos, a arquitetura é bastante influenciada pelo *framework* ou componentes utilizados.

Em seguida à definição da arquitetura adotada, a implementação do jogo segue concentrada na realização das características específicas do jogo sendo desenvolvido. Essa realização é alcançada por meio de adição de novos módulos ao *framework* utilizado, bem como a especialização de classes abstratas existentes nele que exijam uma consideração para cada jogo. Entre os objetivos dessa fase estão, por exemplo, a implementação dos objetos do jogo de acordo com as regras e do comportamento que os objetos devem ter em respostas a eventos acontecidos no jogo.

Além da implementação da lógica do jogo, nesta fase é também desenvolvida a arte gráfica, criação de sons, vídeos e quaisquer outros recursos necessários. Tais produções, em alguns casos, são terceirizadas para empresas especializadas nesses trabalhos.

### 3.2.3 Testes de Corretude e Jogabilidade

Uma vez terminada a implementação do jogo, e muitas vezes até durante, são realizados os testes do jogo para garantir uma boa qualidade. Assim como nos softwares em geral, são realizados testes de corretude para garantir que o jogo segue a sua especificação. No entanto, existe uma segunda fase de testes que visa averiguar se o jogo está adequado para o seu público alvo. Por exemplo, testando se o jogo está difícil/fácil demais, se a performance está boa, se a qualidade da apresentação do jogo (gráficos, sons, vídeos, etc.) está adequada, etc. Esta fase, chamada de teste de

jogabilidade, é fundamental e muito utilizada no desenvolvimento de jogos profissionais.

As pessoas envolvidas nos testes de jogabilidade dependem do orçamento do jogo e tamanho da empresa. Muitas vezes a pessoas da própria equipe de desenvolvimento são utilizadas, o que pode prejudicar o trabalho, visto que os participantes já são familiares ao projeto. Numa situação ideal, a empresa utiliza equipes especializadas na realização de testes de jogos, de dentro da empresa ou terceirizadas. Uma alternativa bem econômica e bastante praticada ultimamente é a divulgação de uma versão preliminar do jogo pela Internet para que seja avaliada por pessoas interessadas.

### 3.3 BIBLIOTECAS MULTIMÍDIA

A fase de implementação dos jogos é bastante acelerada pelo uso de bibliotecas multimídia destinadas a facilitar a integração dos jogos com os diversos tipos e marcas de periféricos utilizados nos PCs, bem como prover diversos recursos úteis para o desenvolvimento e execução eficiente dos jogos.

Como apresentado na evolução histórica do desenvolvimento de jogos, em um dado ponto a introdução do DirectX [25] resultou na aceleração da velocidade de execução dos jogos no ambiente Windows. Além disso, essa biblioteca, assim como o OpenGL [30], desenvolvido pela *Silicon Graphics*, tem a finalidade de proporcionar a portabilidade dos jogos diante das várias marcas de periféricos presentes em PCs. Essas próprias bibliotecas, suportadas pelos diversos fabricantes de dispositivos, utilizam os *drivers* específicos de cada hardware para evitar, entre outras, que o programador tenha que se preocupar com as particularidades de cada tipo de placa de som, vídeo, etc., uma vez que a biblioteca realiza esse tratamento e provê uma interface única e genérica de acesso.

Outro recurso interessante dessas bibliotecas é a capacidade de emular dispositivos não existentes em um dado computador, mas necessários em alguns jogos, como, por exemplo, as placas de aceleração gráfica 3D. Nesse caso, a biblioteca emula os recursos de aceleração para o jogo, porém repassa o trabalho para o processador do PC, em vez de usar uma placa gráfica.

Atualmente o DirectX é utilizado em praticamente todos os jogos e é constantemente atualizado pela Microsoft, bem como novos *drivers* são fornecidos pelos fabricantes de hardware. A grande vantagem do DirectX sobre o OpenGL decorre

do largo suporte a tipos de dispositivos encontrados no DirectX. Enquanto o OpenGL oferece apenas recursos para o processamento gráfico dos jogos, embora exista em fase de desenvolvimento o OpenAL [71], voltado para o suporte a áudio, o DirectX oferece suporte à gráficos 2d e 3d, áudio (inclusive 3D), entrada e saída de dados (inclusive com a tecnologia *forcefeedback*), vídeos digitais (em diversos formatos) e jogos em rede.

### 3.4 FRAMEWORKS DE DESENVOLVIMENTO DE JOGOS

Como resultado da crescente utilização da engenharia de software no desenvolvimento de jogos, a indústria começou a isolar os recursos utilizados na maioria dos jogos em *frameworks* (motores) de desenvolvimento. A idéia principal é permitir que os recursos comuns a quase todos os jogos sejam reutilizados para cada nova jogo criado, cabendo a cada jogo apenas a implementação de seus requisitos particulares. Como exemplo, praticamente toda a implementação do ciclo de atividades de um jogo que foi apresentado na seção 2.1.2 é provido automaticamente pelo *framework*.

Com o uso de *frameworks*, a criação de cada jogo torna-se mais rápida, simples e organizada, pois, na maioria das vezes, a utilização de um *framework* implica na estruturação do jogo de acordo com a arquitetura utilizada nele.

Devido a enorme adoção dessas ferramentas pela indústria de jogos, atualmente existem empresas especializadas somente na construção e venda de *frameworks*. Para se ter uma idéia, um *framework* pode chegar ao preço de 500 mil dólares, como é o caso do Unreal [73], ou de 200 mil dólares, no caso do LithTech [74].

A seguir são resumidos os principais requisitos de um *framework*, seguido das considerações necessárias para a implementação de uma ferramenta como estas. Este resumo serve de base para a concepção desse trabalho. Uma vez que o wGEM é um *framework* de desenvolvimento de jogos, embora que simplificado, já que é voltado para dispositivos muito limitados.

#### 3.4.1 Requisitos

Um *framework*, ou motor de jogos, é a peça chave no desenvolvimento de jogos de computador, tendo como principal objetivo executar uma grande parte das tarefas de baixo nível necessárias na maioria os jogos.

Alguns dos principais requisitos de um motor são [5][4][17]:

- Arquitetura modular, para que seja fácil a sua utilização e extensão para cada novo jogo;
- Bom nível de abstração, ao prover objetos com grande parte das funcionalidades automaticamente implementadas;
- Definição de objetos básicos, como o objeto do jogo e o mapa (cenário) do jogo.
- Detecção e tratamento adequado de eventos gerados por teclado, mouse, joysticks, etc;
- Implementação dos algoritmos básicos para desenho dos objetos do jogo, podendo ser 2D, 3D, etc;
- Funcionalidades úteis à jogos 3D, como iluminação, transformações geométricas, sistema de navegação, etc;
- Execução dos sons do jogo em resposta a eventos ocorridos;
- Implementação de algoritmos de IA para implementar os NPCs dos jogos;
- Implementação de algoritmos para troca remota de dados, gerenciamento de sessões, sincronização das informações compartilhadas do jogo, etc;
- Implementação de algoritmos para modelagem física de objetos.

Em consequência da grande variedade de requisitos dos *frameworks*, muitos procuram limitar-se a determinados estilos de jogos, como, por exemplo, motores para jogos 2D, 2½D, 3D, jogos de turno (Damas e Xadrez), etc. O *framework* construído como resultado desta pesquisa, por exemplo, envolve apenas o desenvolvimento de jogos 2D para dispositivos móveis.

### 3.4.2 Implementação

A implementação de um motor envolve diversos aspectos computacionais, a seguir são listados os principais:

- Escolha apropriada da linguagem de programação;
- Definição da arquitetura do motor;
- Desenvolvimento dos módulos.

A escolha da *linguagem de programação* ocorre principalmente em função da facilidade de uso, eficiência, portabilidade e, claro, viabilidade tecnológica de tal linguagem para o desenvolvimento de jogos. Um outro ponto a ser considerado é a

existência de bibliotecas de suporte ao desenvolvimento de jogos compatíveis com a linguagem escolhida. Devido a esses fatores, a grande parte dos *frameworks* para PCs são desenvolvidos em C++, bem como fazem uso das bibliotecas DirectX e/ou OpenGL.

A *arquitetura* de um motor corresponde exatamente à definição dos seus módulos e como é realizada a interação entre eles. Tais módulos são responsáveis por identificar e gerenciar todos os eventos que ocorrem ao longo do jogo, desde eventos gerados pelo usuário até aqueles causados por interação entre os objetos do jogo. Por isso, existem normalmente módulos para manipulação de objetos, renderização, sonorização, IA, rede e outros. Em geral, tais módulos refletem os requisitos do *framework* que foram identificados.

A qualidade da arquitetura é um dos aspectos mais importantes de um motor. Tal arquitetura torna-se crítica pelo fato de que um motor de jogos, em geral, contém um conjunto de classes abstratas que requerem, portanto, uma especialização para produzir cada jogo. Sendo assim, é de extrema importância que a arquitetura do motor seja clara o suficiente para que tal tarefa seja possível. Infelizmente, não há consenso sobre arquitetura de *frameworks* nem muitas publicações sobre esse tema, o que implica num grande esforço de análise e generalização de diversas soluções para compilar as características comuns (e corretas).

Uma vez definida a arquitetura do motor, os seus diversos *módulos* são então implementados. A quantidade de módulos depende muito da complexidade do *framework*. Os módulos responsáveis por atividades como, por exemplo, identificação de entrada de dados, são muito mais simples do que os responsáveis por transformações geométricas e renderização de objetos 3D ou de algoritmos de inteligência artificial. A implementação dos módulos, portanto, envolve a criação de algoritmos específicos para diversas tarefas. No módulo que representa o objeto do jogo, por exemplo, algoritmos básicos de deslocamento espacial, bem como a detecção de colisão com outros objetos do jogo são necessários. Em módulos que representem repositórios para objetos do jogo, por exemplo, há necessidade da definição e implementação de estruturas de dados muito eficientes em uso de memória e CPU.

Geralmente após (ou durante) a fase de implementação há sempre uma longa etapa de otimização. As otimizações realizadas no *framework* são de extrema importância, pois a sua performance impacta diretamente em todos os jogos construídos sobre o mesmo.

### 3.4.3 Editor de Cenários

Uma ferramenta bastante útil à criação dos jogos, e que em geral é desenvolvida para ser integrada a cada *framework*, é o editor de cenários. Esta ferramenta tem o papel de permitir, de forma visual e simples, a criação de níveis (missões ou cenários) para jogos construídos a partir do *framework* integrado ao editor. Para que isso ocorra, a interface do editor de cenários deve permitir que todas as informações que o *framework* precisa para criação de cada nível de um jogo possam ser especificadas. Uma vez que isso seja realizado, e um dado estágio seja definido, o editor exporta o cenário para um arquivo, cujo formato, contendo as informações especificadas, é compreendido pelo *framework*.

Para melhor entender a utilização dos editores de cenário, considere a Figura 3-2, que ilustra três diferentes níveis do jogo BreakOut. Nessas três configurações do jogo, a única diferença é a formação dos blocos na tela (o cenário do nível). O comportamento da bola, da raquete e dos blocos é o mesmo, e foi implementado no ato da criação do jogo e/ou *framework*. Com isso, assumindo que o editor seja capaz de permitir a especificação deste cenário e geração de arquivos descrevendo a formação criada, a partir desses arquivos, o *framework* é capaz de iniciar todos os objetos relacionados com cada nível do jogo.

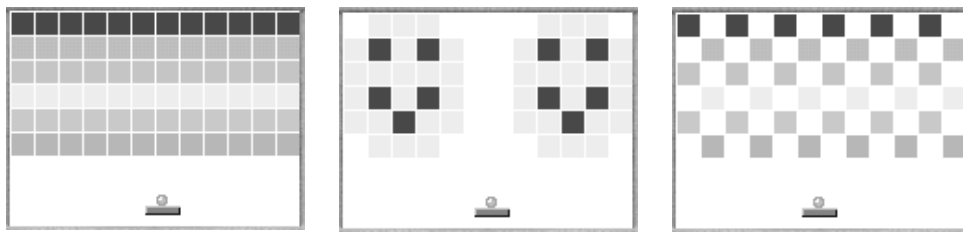


Figura 3-2: Exemplo de três cenários para o jogo BreakOut

Essa facilidade de criação visual de novas instâncias do jogo sem a necessidade de implementação extra no jogo ou no motor é a grande vantagem de uso dos editores de cenários. Principalmente pelo fato de que os editores de cenários são em geral utilizados pelo *Game Designer*, que não necessariamente sabe programar.

A forma de comunicação (acoplamento) entre o editor de cenários e o motor é um ponto muito importante. Pois toda a especificação do cenário realizada de forma simples e visual através do editor de cenários só tem valor se puder ser transferida para o motor.

Uma das formas de comunicação, quando o editor é desenvolvido baseando-se nos mesmos recursos de linguagem de programação presentes no motor, é utilizar no



editor as mesmas estruturas de dados adotadas no motor para representação dos objetos que definem os cenários. Nesses casos, uma vez que um dado cenário seja especificado, o editor de cenários pode utilizar as informações fornecidas para criar um objeto representando o cenário que seja o mesmo manipulado pelo motor. Quando este mecanismo é utilizado, a única coisa que o editor precisa fazer é serializar o objeto que representa o cenário para um arquivo que será lido pelo motor para que os devidos objetos sejam inicializados. Uma vantagem dessa abordagem é a forte integração entre o editor e o motor, o que muitas vezes pode até permitir que o cenário definido seja simulado no próprio editor de cenários.

Uma outra forma de comunicação é a definição de uma linguagem descritiva capaz de representar as informações necessárias para especificação de um cenário. A idéia é fazer com que todas as informações especificadas no editor sejam automaticamente traduzidas para uma linguagem intermediária (textual e descritiva) compreendida pelo motor. Utilizando essa abordagem, o editor de cenários basta então realizar a tradução do cenário definido para um arquivo texto que será lido pelo motor para a inicialização do cenário. A vantagem principal dessa forma de comunicação é a independência entre o editor e o motor, o que permite, por exemplo, que os dois sistemas sejam desenvolvidos de forma completamente independente, bastando a definição do formato do arquivo de descrição.

É interessante notar que um editor de cenários pode ser genérico (voltado para o *framework*) ou específico para um certo jogo. No caso dele ser genérico, apenas as características dos objetos que façam parte do *framework* são especificadas no editor. Quando a especialização do editor para um certo jogo é adotada, o editor é alterado para refletir também as características peculiares dos objetos manipulados pelo jogo.

### ***3.5 FERRAMENTAS DE APOIO AO DESENVOLVIMENTO DE JOGOS***

Um fato comum a todos os motores de jogos é a manipulação de imagens, sons, arquivos texto que descrevem os cenários do jogo, modelos 3D de objetos, etc. Portanto, é necessário o uso de ferramentas auxiliares para a criação e modelagem de tais arquivos.

Em se tratando de manipulação de imagens, existem vários editores que permitem o desenho de imagens pixel a pixel, aplicação de efeitos como textura e iluminação, exportação de imagens para diversos formatos de arquivos, etc. Entre eles

destacam-se o Jasc Paint Shop Pro 7 [31], Corel Photo-Paint 9.0 [32] e o Adobe Photoshop 6.0 [33].

Para a criação e edição dos efeitos sonoros e trilhas musicais utilizados em jogos, os processadores de sons são utilizados. Um dos melhores programas de processamento de sons digitais do mercado é o Sound Forge [34], apresentando uma enorme capacidade de processamento e facilidade de uso. No caso de sons baseados no formato MIDI, o seqüenciador mais popular e poderoso é o CakeWalk [35].

Para o desenvolvimento de jogos 3D é necessária a criação e animação de caracteres 3D e efeitos visuais. Este processo é realizado através do uso de ferramentas especializadas que contém os mais recentes efeitos e modelos tridimensionais existentes, facilitando a criação de objetos, animações e imagens para os jogos. Nesse mercado, os softwares 3D Studio Max 4 [36] e o Maya 3 [37] são os mais utilizados.

### 3.6 CONCLUSÕES

Como apresentado neste capítulo, o desenvolvimento de jogos tem certas peculiaridades. Dentre as quais está a criação do *game design*, cuja complexidade pode ser comparada em certos pontos a um roteiro de um filme. A boa qualidade do *game design* é sem dúvidas fundamental para o sucesso de um jogo. Assim como nos filmes, não basta apenas o uso de tecnologia de ponta, efeitos especiais, sons, etc. É necessário uma boa história, um cenário coerente, um forte envolvimento com o público, e outros fatores que são levados em consideração quando um *game design* é elaborado.

Outro ponto importante, resultado da crescente utilização da engenharia de software em jogos, é a tendência de utilização de *frameworks* de desenvolvimento. Um *framework* é um elemento essencial ao desenvolvimento dos jogos devido a sua capacidade de prover uma enorme quantidade de recursos úteis, reduzindo, portanto, a quantidade de código a ser escrito e a dificuldade de desenvolvimento dos jogos, bem como aumentando a organização final do projeto. Tudo isso acarreta, conseqüentemente, na aceleração do processo de desenvolvimento de jogos, fator crucial em um mercado cada vez mais exigente.

## 4 JAVA 2 MICRO EDITION (J2ME)

Este capítulo trata dos principais aspectos da linguagem Java 2 Micro Edition (J2ME). Esta nova linguagem, com aproximadamente um ano de criação, é uma simplificação da linguagem Java 2 Standard Edition (J2SE) realizada através da remoção e modificação de partes fundamentais de J2SE com o objetivo de criar um ambiente de execução de programas para dispositivos com grandes restrições de memória e processamento. Inicialmente é realizada uma apresentação histórica sobre o surgimento da linguagem. Em seguida são mostrados alguns detalhes da arquitetura da linguagem e a aderência mercadológica que ela tem alcançado.

### 4.1 *HISTÓRICO*

O surgimento de J2ME pode ser visto como uma retomada ao objetivo inicialmente definido pela Sun Microsystems quando a linguagem Java foi concebida. Em 1990, quando a linguagem ainda chamava-se Oak [2], o objetivo do projeto da Sun era definir uma linguagem para ser embutida em dispositivos bastante simples. No entanto, com os computadores pessoais (PCs) em alta, nada impedia que a linguagem fosse também utilizada em dispositivos complexos. Era apenas uma questão de implementar o suporte ao uso da linguagem nesses computadores. Uma vez que isso foi realizado, Java começou a ser utilizada e vem alcançando cada vez um maior público, sendo hoje uma das linguagens de programação mais utilizadas e de maior capacidade de desenvolvimento de grandes projetos.

Ao longo desses anos de sucesso de Java, tem ocorrido um inevitável e constante crescimento do número de recursos sendo oferecidos, e conseqüentemente, do tamanho do seu ambiente de execução e número de classes [1]. A cada nova versão da linguagem novos recursos são adicionados, tais como, Swing, Java 2D, Java 3D, RMI, etc. Tal crescimento é normal, visto que Java começou como uma linguagem pequena e simples, pois tinha como público alvo equipamentos também simples. Esse crescimento, ilustrado na Figura 4-1 por três versões de Java, mostra que cada vez mais a plataforma vem exigindo mais memória e capacidade de processamento.

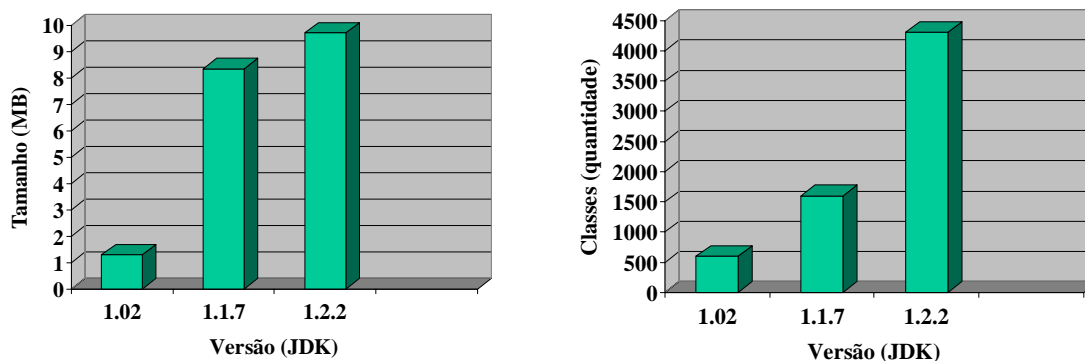


Figura 4-1: Crescimento no tamanho da linguagem Java

Diante dessa situação, o suporte a Java por dispositivos simples tornou-se impossível. Deixando esse segmento importante de mercado, liderado por celulares, *paggers* e PDAs, limitado a sua forma própria (e geralmente privada) de execução e desenvolvimento de programas. Tal situação inviabiliza que programas sejam desenvolvidos pela comunidade para tais dispositivos, e que até mesmo uma eventual atualização do software embutido nos aparelhos seja liberada pelos fabricantes para que os usuários mais cuidadosos a utilizem. Hoje é fácil de observar essa situação com os celulares, que vêm com um conjunto de programas previamente desenvolvidos pelo fabricante (utilizando o mais baixo nível de programação que diretamente manipula o sistema operacional do aparelho) e não permitem que o usuário instale novos softwares. Essa mesma observação também é válida para rádios, receptores de TV via satélite, fornos microondas, etc.

A primeira tentativa da Sun de contornar esse problema foi realizada em 1997, com o desenvolvimento de três plataformas simplificadas de Java:

- JavaCard;
- EmbeddedJava;
- PersonalJava.

A plataforma JavaCard [62] define um ambiente Java capaz de ser embutido em *smart cards*, juntamente com uma API de desenvolvimento de programas para esta plataforma. *Smart cards* são cartões de plástico do tamanho de um cartão de crédito dotados de um microprocessador, chips de memória e uma interface de comunicação. Através dessa interface, o cartão pode trocar as informações nele armazenadas com um outro dispositivo especial (leitor de *smart cards*) que em geral mantém conexão com um computador. Durante essa fase de troca de dados, os programas instalados no cartão

também podem solicitar dados ao leitor, realizar operações com eles e armazená-los na memória do cartão. Os *smart cards* são bastante utilizados atualmente como mecanismo de identificação eletrônica, bem como para controle financeiro, especialmente quando ele é utilizado também como cartão bancário e de crédito.

A tecnologia JavaCard também é utilizada com sucesso em anéis, chamados de *Java Rings* [75], com finalidade similar aos *smart cards*. A Figura 4-2 mostra um *smart card* e um *Java Ring* que utilizam a tecnologia JavaCard.



Figura 4-2: Smart Card e Java Ring com a tecnologia JavaCard

A plataforma EmbeddedJava [63] é voltada para fabricantes de sistemas embarcados. O seu objetivo é permitir que os fabricantes desses dispositivos possam utilizar a linguagem Java para o desenvolvimento do software a ser embutido em seu sistema (em lugar ao tradicional C ou *Assembler*). Naturalmente, a própria máquina virtual Java também é embutida, porém, o grande diferencial dessa tecnologia é que fica a critério de cada fabricante configurar que recursos da máquina virtual são necessários embutir no dispositivo de acordo com a sua capacidade. Sendo assim, a diversidade de equipamentos que podem suportar Embedded Java é imensa, já que as exigências do ambiente Java é amplamente configurável. Um caso de uso desta tecnologia, mostrado na Figura 4-3, é um osciloscópio desenvolvido pela empresa Tektronix [65].



Figura 4-3: Osciloscópio da empresa Tektronix baseado na tecnologia Embedded Java

A tecnologia PersonalJava [64] foi desenvolvida para permitir que um ambiente Java um pouco mais simples que o Java para PCs fosse definido de forma a ser suportado por dispositivos com poucos recursos. O foco dessa plataforma eram os

*screenphones*<sup>1</sup> e computadores portáteis sofisticados, que prontamente passaram a adotar a tecnologia em seus sistemas. Hoje, praticamente todos os sistemas operacionais utilizados em computadores portáteis e PDAs avançados já contam com uma máquina virtual para o PersonalJava. A Figura 4-4 mostra dois *screenphones*, fabricados pela Alcatel e SamSung, respectivamente, e um PDA (que também é telefone celular) desenvolvido pela Nokia que suportam a plataforma PersonalJava.



Figura 4-4: Dois screenphones e um PDA suportando PersonalJava

Apesar dessas três tecnologias terem aumentado a utilização de Java em dispositivos simples, a Sun observou que essa solução ainda tinha certas limitações que impediam a sua utilização como uma arquitetura padrão em alguns tipos de aparelhos [68]. Das três plataformas, apenas o JavaCard era considerado uma boa solução para o seu público-alvo, possibilitando que a plataforma pudesse ser utilizada como padrão do mercado de *smart cards*. O Embedded Java, por outro lado, como era modificado por cada fabricante de hardware que o utilizasse, eliminava a compatibilidade entre dispositivos baseados na tecnologia. Por isso, o desenvolvimento de programas não era aberto à comunidade, apenas aos próprios fabricantes dos dispositivos, que sabiam exatamente quais recursos da linguagem estavam presentes em seus aparelhos. Já o PersonalJava, apesar de exigir que cada dispositivo seguisse uma arquitetura padrão (proporcionando, portanto, portabilidade de programas entre dispositivos), era grande demais para ser utilizado em dispositivos simples como os celulares, *paggers* e alguns PDAs. Além disso, freqüentemente algumas adições de pacotes e APIs ao PersonalJava eram necessárias para ampliar o leque de dispositivos cobertos pela plataforma, como o JavaTV [66] e JavaPhone [67].

Vendo que era necessário fazer uma mudança mais radical em sua arquitetura para unificar todas essas diferentes tecnologias em uma arquitetura modular que pudesse se adequar de forma melhor aos diversos dispositivos existentes, a Sun formou

---

<sup>1</sup> Telefones avançados com uma tela de cristal líquido e acesso à Internet.

um consórcio com as maiores empresas de aparelhos eletrônicos para chegar a uma solução. Dessa iniciativa, a tradicional plataforma Java utilizada nos PCs foi dividida em três partes, na verdade três edições – Enterprise[12], Standard[11] e Micro[13], cada uma sendo direcionada para diferentes categorias de dispositivos.

- Java 2 Standard Edition (J2SE) , direcionada para as aplicações convencionais que são executadas em computadores domésticos ou servidores e que não requerem recursos avançados;
- Java 2 Enterprise Edition (J2EE), direcionada para as grandes aplicações comerciais que visam atender serviços requisitados por milhares de outros programas;
- Java 2 Micro Edition (J2ME), direcionada para o desenvolvimento de aplicações voltadas para dispositivos com capacidade limitada de processamento e memória.

Realizada essa divisão, a Sun apresentou J2ME oficialmente durante a conferência JavaOne em 1999 [21]. Porém, um grande trabalho ainda teria que ser realizado para a definição da arquitetura de J2ME, que entre outras coisas deveria ser capaz de englobar as plataformas Embedded Java e PersonalJava. Essa arquitetura, que levou quase um ano para ser concretizada, é explicada a seguir.

## ***4.2 ARQUITETURA DE J2ME***

Sem dúvida, uma característica de J2ME deveria ser a simplicidade. Dessa forma, uma grande parte do trabalho de definição da arquitetura da nova linguagem foi dedicado a considerar que componentes da linguagem Java original deveriam ser cortados, mantidos ou modificados. Durante essa fase, foi observado que muitas das características que estavam sendo atribuídas à J2ME não eram de comum necessidade para todos os tipos de dispositivos. Por exemplo, nem todo dispositivo utiliza uma tela gráfica como interface com o usuário, ou de forma equivalente, nem todos os dispositivos teriam a capacidade de estabelecer uma chamada telefônica. Se funcionalidades como essas fossem atribuídas à J2ME, o mesmo erro do passado estaria sendo cometido, uma vez que tais recursos seriam utilizados por alguns aparelhos (telefones celulares, por exemplo) enquanto que seria uma grande sobrecarga desnecessária para outros (como um rádio ou geladeira).

Com a observação dessa variedade de dispositivos, J2ME adotou uma arquitetura modular e escalável, consistindo de um conjunto de blocos que se complementam para representar as particularidades dos grupos de dispositivos

existentes. Como mostrado na Figura 4-5, J2ME define três camadas de software construídas sobre o sistema operacional presente em cada dispositivo.

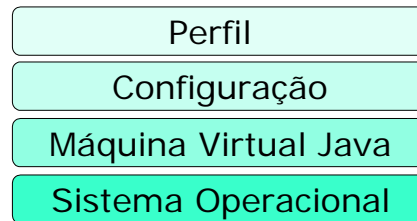


Figura 4-5: Modelo de Camadas de J2ME

Essas três camadas de software são apresentadas nas três seções seguintes.

#### 4.2.1 *Máquina Virtual Java*

Como comentado, J2ME é uma tecnologia definida por várias partes que se complementam para prover um ambiente compatível com a grande quantidade de dispositivos. Como em todas as tecnologias Java, na base de todas essas partes está uma máquina virtual com objetivo de garantir a portabilidade e execução de programas em qualquer dispositivo. A *portabilidade* é garantida desenvolvendo-se esta máquina virtual para cada dispositivo existente, levando em conta o sistema operacional presente e as características dos aparelhos. A *execução* de programas, bem como a execução da própria máquina virtual nesses dispositivos simples resultou num grande desafio de otimização para tornar esta máquina virtual o mais simples possível.

Diferentemente do termo JVM (de *Java Virtual Machine*), utilizado para referenciar as máquinas virtuais, em J2ME a máquina virtual é chamada de KVM (de *Kilobyte Virtual Machine*), em referência ao seu pequeno tamanho, usualmente menor que 128KB. A KVM não é apenas pequena, ela é uma implementação bastante otimizada da máquina virtual Java considerando todas as restrições de memória e processamento envolvidas nos pequenos aparelhos.

Como será apresentado nas próximas duas seções, os recursos oferecidos por J2ME dependem de que configuração e perfil é utilizado. Sendo assim, a própria máquina virtual também varia de acordo com os requisitos presentes em cada ambiente formado. Por isso, o termo KVM não é considerado muito correto, pois em alguns casos essa máquina alcança o tamanho de 2 MB, como será apresentado. Entretanto, o termo KVM foi adotado para representar a máquina virtual presente em sistemas utilizando J2ME.



### 4.2.2 Configuração

A camada de configuração define os recursos Java que fazem parte de uma larga categoria de dispositivos que compartilham certas características importantes, tais como capacidade de memória, comunicação, consumo de energia e interface com o usuário. Em outras palavras, uma configuração define o “menor denominador comum” dos recursos da plataforma Java que serão suportados em todos os dispositivos pertencentes à larga categoria considerada pela configuração.

Para evitar uma grande segmentação, apenas duas configurações existem, *Connected Device Configuration* – CDC[16] e *Connected Limited Device Configuration* - CLDC[15]. Essas configurações foram definidas a partir de fóruns de discussão com centenas de empresas, desenvolvedores e usuários de dispositivos que se mostraram interessados na definição dessas configurações. A seguir seguem as suas principais características.

#### 4.2.2.1 CLDC

A configuração CLDC é direcionada para dispositivos com grandes restrições de memória e processamento. Nesta categoria estão dispositivos como os celulares, *paggers* e os PDAs mais simples, cujas características comuns são apresentadas a seguir:

- Memória disponível para J2ME entre 128KB e 512KB;
- Conectividade à baixa qualidade e largura de banda (em torno de 9600 bps e intermitente, por ser sem fio);
- Grandes restrições de energia (fornecida através de bateria);
- Interface com o usuário muito limitada (em tamanho e resolução);
- Processador de 16 ou 32 bits e frequência em torno de 25 MHz.

Baseadas nessas restrições, um conjunto de quatro pacotes Java foi definido para contemplar os recursos igualmente suportados por todos os dispositivos abrangidos pela configuração CLDC – *java.lang*, *java.util*, *java.io* e *javax.microedition*. Com exceção do pacote *microedition*, os outros pacotes são subconjuntos dos pacotes equivalentes em J2SE. No entanto, todos eles foram totalmente redefinidos para remover classes desnecessárias, bem como métodos e atributos [58]. Os principais recursos oferecidos por todos esses pacotes são os seguintes:

- Suporte a tipos primitivos de dados, tais como, Boolean, Byte, String e Integer;
- Suporte a algumas estruturas de dados, tais como, Hashtable e Vector;
- Suporte a importantes recursos da linguagem J2SE, tais como utilização de *Threads*, *Exceptions* e *Garbage Collector*;
- Definição de um *framework* básico para comunicação de dados, sem no entanto implementá-lo.

#### 4.2.2.2 CDC

A configuração CDC tem o papel de cobrir os dispositivos com recursos superiores aos dispositivos tratados pela configuração CLDC, porém inferiores aos encontrados nos computadores pessoais, que já contam com o próprio J2SE. Nesta categoria estão dispositivos como os televisores, computadores de bordo/sistemas de navegação de carros e *screenphones*, cujas características comuns são apresentadas a seguir:

- Memória disponível para Java entre 2MB e 16MB;
- Conectividade à alta qualidade e largura de banda;
- Dispositivos fixos e com boa alimentação de energia;

De forma análoga à configuração CLDC, um conjunto de pacotes Java foi definido para contemplar os recursos igualmente suportados por todos os dispositivos abrangidos por CDC - *java.lang*, *java.util*, *java.net*, *java.io*, *java.text* e *java.security*. Todas as classes presentes nesses pacotes são idênticas às existentes em J2SE nos pacotes equivalentes. O CDC também engloba todos os pacotes presentes em CLDC, com o objetivo de manter a compatibilidade entre os dois. A Figura 4-6 ilustra a relação de pertinência entre os recursos de CLDC, CDC e J2SE [58]. Observar que tanto CDC quanto CLDC contém recursos adicionais à J2SE, como o pacote *javax.microedition*, por exemplo.

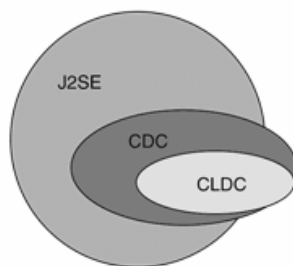


Figura 4-6: Cobertura de classes de CDC e CLDC em relação à J2SE

Devido a essa existência de duas configurações, duas máquinas virtuais diferentes foram especificadas com o objetivo de suportar as características existentes em cada configuração. Uma delas é a KVM já comentada, que é voltada para a configuração CLDC. A outra, chamada de CVM (de *Compact Virtual Machine*) [16] é direcionada para a configuração CDC.

### 4.2.3 Perfil

O objetivo da camada de perfil é especializar uma determinada configuração para uma parte do mercado largo que ela atinge. Cada perfil define a API voltada para demandas específicas de um segmento de mercado, tais como, carros, máquinas de lavar, celulares e televisores.

Um perfil é a camada mais próxima para os usuários e desenvolvedores de aplicações. Ela provê mecanismos como entrada de dados, interface com o usuário, persistência de dados e conectividade da forma ideal para o segmento de mercado abrangido pelo perfil.

Naturalmente, um programa escrito para um dado perfil tem a portabilidade garantida para qualquer dispositivo que suporte tal perfil. A idéia é que isso ocorra da mesma maneira que aplicações para PCs são desenvolvidas para um determinado sistema operacional sem preocupação sobre o fabricante do computador que irá executá-lo.

Por lidar com segmentos específicos de mercado, o número de perfis é bem maior que o de configurações. Atualmente existem dois perfis definidos, *Mobile Information Device Profile – MID Profile* [14], utilizando a configuração CLDC, e *Foundation Profile* [59], que utiliza a configuração CDC. Vários outros perfis estão em processo de desenvolvimento, também através de fóruns com toda a indústria interessada. Entre eles destaca-se o perfil *PDA Profile* [56], utilizando a configuração

CLDC e voltado para as demandas específicas dos PDAs. A seguir seguem maiores detalhes sobre o MIDP e *Foundation Profile*.

#### 4.2.3.1 MID Profile (MIDP)

Em função da grande exigência da indústria da telefonia móvel, o primeiro perfil da tecnologia J2ME a ser desenvolvido foi o MIDP, voltado para dispositivos móveis como os celulares e *paggers*. A sua especificação foi realizada em conjunto com mais de 20 empresas da indústria dos dispositivos móveis, resultando em um conjunto de pacotes Java que suplementam a configuração CLDC com diversos novos recursos, destacando-se:

- Definição de classes para criação da interface gráfica dos programas;
- Suporte à persistência de dados pelas aplicações, através de um pequeno banco de dados baseado em registros que é gerenciado pelo próprio MIDP;
- Implementação do protocolo HTTP para comunicação remota de dados utilizando a própria rede sem fio dos equipamentos;
- Representação da entidade aplicação, nomeada MIDlet, com um comportamento muito similar aos *applets* utilizados em J2SE, que deve utilizada pela aplicação para que possa ser executada pela KVM.

O perfil MIDP alcançou um sucesso enorme. Primeiramente pelo fato da Sun ter liberado uma ferramenta de desenvolvimento gratuita para CLDC/MIDP, como será mostrado na seção 4.3, que permitiu a emulação e desenvolvimento de programas em PCs. Segundo, ainda mais importante, foi a grande aderência mercadológica à esse padrão pela indústria de telefonia móvel, como será mostrado na seção 4.4. O sucesso foi tanto, que a Sun realizou uma implementação da KVM para o sistema operacional PalmOS, utilizado nos Palms, para dar suporte à CLDC e MIDP nesses PDAs [27]. Com isso, mesmo enquanto o perfil específico para PDAs não é finalizado, já é possível utilizar J2ME em Palms gratuitamente, bastando para isso instalar essa KVM disponibilizada para *download* pela Sun.

Apesar do suporte proporcionado pelo conjunto CLDC/MIDP, alguns recursos bastante importantes não são encontrados, entre eles destacam-se:

- Ausência de ponto flutuante, devido ao fato que operações envolvendo números reais são muito caras para a capacidade dos processadores encontrados nos dispositivos móveis cobertos pela plataforma;

- Ausência de reflexão de objetos, o que implica que programas não podem inspecionar o conteúdo de classes, objetos e métodos, para, por exemplo, realizar serialização de objetos e execução remota de métodos (RMI)[20];

#### 4.2.3.2 Foundation Profile

O *Foundation Profile* [59], também definido em conjunto com a indústria interessada, especifica um conjunto de pacotes Java que suplementam a configuração CDC com os principais pacotes ausentes nessa configuração, destacando-se o suporte a *sockets*, internacionalização e a implementação das classes ausentes nos pacotes *java.lang* e *java.io*.

O objetivo da Sun com o perfil *Foundation* é mais em servir como base para outros perfis mais avançados do que ser de fato utilizado isoladamente para o desenvolvimento de programas, apesar de ser possível [69]. Nesse sentido, atualmente existem alguns perfis bem interessantes em fase de especificação que poderão ser utilizados sobre o *Foundation Profile*. Um deles é chamado *Java Game Profile* [60], com o objetivo de prover os recursos necessários para o desenvolvimento de jogos 3D. Outro perfil a seguir essa estratégia é o *Personal Profile* [61], com o objetivo de prover um ambiente similar ao oferecido pela tecnologia PersonalJava.

A Figura 4-7 resume a atual situação da arquitetura geral da linguagem Java, ilustrando também com mais detalhes as máquinas virtuais, configurações e perfis de J2ME.

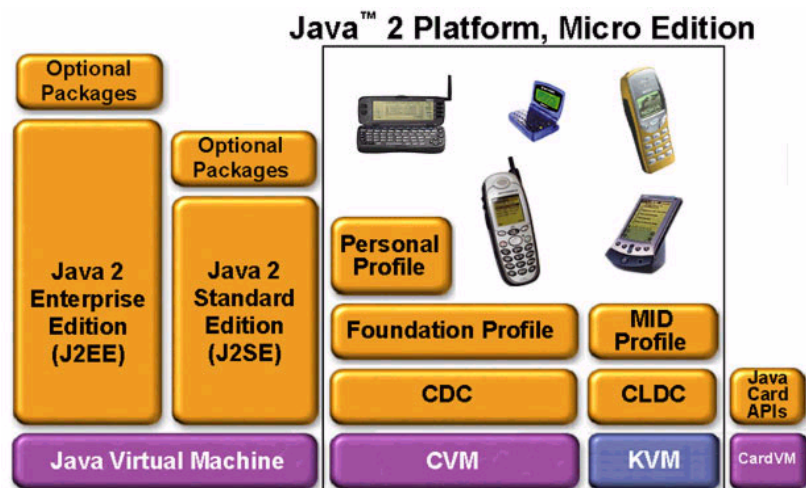


Figura 4-7: Atual arquitetura de Java [58]

### 4.3 FERRAMENTAS DE DESENVOLVIMENTO

Existem atualmente várias ferramentas para o desenvolvimento e execução de programas em J2ME [53]. Empresas como Borland, IBM e Sun já oferecem ambientes de desenvolvimento muito satisfatórios [53]. Dentre essas, a solução fornecida gratuitamente pela Sun é a versão de referência para a área [18]. Esta ferramenta, formada por um emulador (*J2ME Wireless Toolkit*) e uma ferramenta de desenvolvimento (*Forte for Java*), provê um ambiente visual e muito prático para o desenvolvimento e teste de programas em J2ME. Uma boa característica do ambiente da Sun é a capacidade de emulação de um programa em diversos tipos diferentes de dispositivos, o que é muito prático e evita que os programas tenham que ser testados nos dispositivos reais. A Figura 4-8 mostra este ambiente.



Figura 4-8: Ambiente de desenvolvimento para J2ME

#### 4.3.1 Instalação de Programas nos Dispositivos Móveis

Uma funcionalidade essencial de todos os ambientes de desenvolvimento para J2ME é a capacidade de geração de arquivos binários com todo o resultado da compilação de um programa. Ou seja, o arquivo que precisa ser transferido do emulador para ser instalado e executado no dispositivo real. O conteúdo desse arquivo compactado (JAR) é formado pelo resultado da compilação de todas as classes do projeto em desenvolvimento, bem como um arquivo descritor que contém informações sobre o projeto, tais como o nome da classe principal.

Para que a instalação desses programas nos dispositivos ocorra, os dispositivos devem implementar um mecanismo chamado *Java Application Manager* (JAM), cuja especificação é definida pelo perfil J2ME em uso pelo dispositivo. Um cenário típico de instalação de um programa é mostrado na Figura 4-9 [70]. Nesta situação, um usuário

de um celular com suporte à MIDP acessa um página na Internet (possivelmente utilizando WAP) que mostra os nomes de um conjunto de aplicações disponíveis para compra. Quando o usuário seleciona uma dessas através do aparelho, automaticamente é feito o *download* de um arquivo texto que descreve o conteúdo do programa (contendo poucos bytes) utilizando a rede sem fio do dispositivo. Este arquivo descritor informa para o dispositivo as informações básicas sobre a aplicação, como, por exemplo, o tamanho do programa e que configuração e perfil são necessários para a sua execução. Uma vez que o JAM tenha checado a possibilidade de instalação do programa (através das informações do arquivo descritor) o processo de *download* da aplicação é realizado. O JAM irá salvar a aplicação no dispositivo e adicioná-la à lista de programas instalados pelo usuário.

O modelo atual de uso do JAM impede que apenas novas classes sejam instaladas para um dado programa presente no dispositivo. Devido a essa limitação, se uma atualização de um programa for necessária, o *download* e instalação de todo o novo programa (novo arquivo JAR) é necessário.

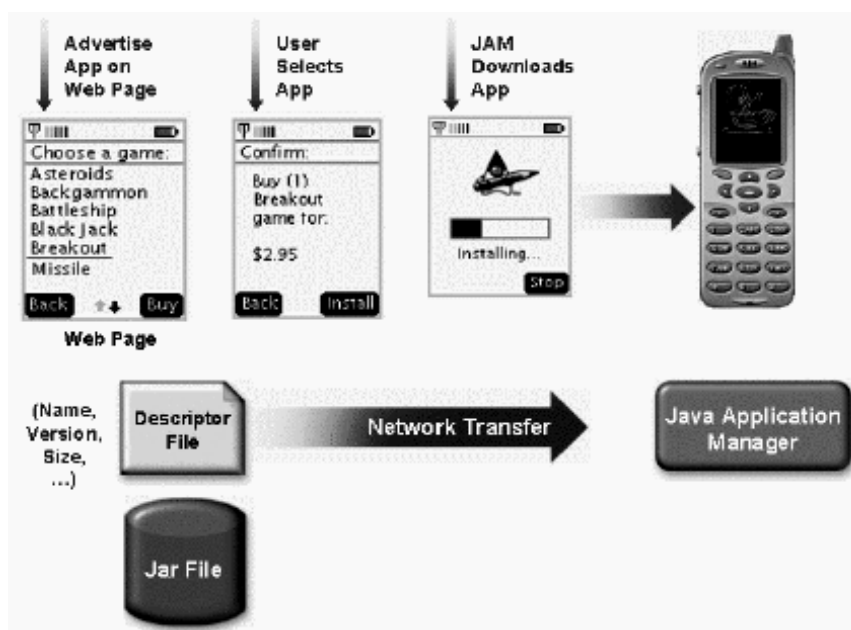


Figura 4-9: Processo de instalação de programas em dispositivos

Esse mesmo processo de instalação também pode ser feito via um cabo ligando o PC ao dispositivo. O processo de instalação é semelhante, com a diferença que o *download* é feito via o próprio cabo de conexão. Nesse caso, é necessária a utilização de um software para PCs fornecido pelo fabricante do dispositivo, bem como a compra do cabo de conexão.

## 4.4 ADERÊNCIA MERCADOLÓGICA

Com a definição completa da arquitetura de J2ME para dispositivos móveis (CLDC/MIDP), diversos fabricantes iniciaram imediatamente o desenvolvimento das máquinas virtuais Java para os seus sistemas operacionais, de forma que hoje, com pouco mais de um ano, diversos fabricantes de dispositivos móveis já oferecem, ou anunciam, dispositivos com suporte à J2ME. Empresas como Siemens, Nokia, NEC, Samsung, Motorola, Sony, Toshiba, Casio e Sharp já desenvolveram dispositivos com a tecnologia J2ME. O número de modelos de celulares já são quase 30, e esse número não para de crescer [54]. Só a Nokia anunciou que irá fabricar, somente em 2002, 50 milhões de aparelhos com suporte a J2ME, e que esse número será o dobro em 2003 [72]. Na Figura 4-10, alguns modelos de celulares que já suportam J2ME são ilustrados.

Uma máquina virtual Java para o sistema operacional utilizada nos Pams, PalmOS, também já foi desenvolvida [27], de forma que qualquer usuário de um Palm pode instalar gratuitamente esta máquina virtual e executar programas desenvolvidos para CLDC/MIDP.



Figura 4-10: Exemplos de dispositivos que suportam J2ME

A grande maioria dos celulares com J2ME só está disponível no Japão, onde há um maior avanço tecnológico das empresas de telefonia. Lá, onde essas novidades tecnológicas sempre promovem uma grande corrida às lojas, recentes pesquisas revelaram números surpreendentes de venda. O primeiro deles diz a venda destes novos aparelhos chegou a alcançar a taxa de 230 mil unidades somente na primeira semana [84]. O segundo dado, no entanto, é ainda de maior importância, pelo menos para a área



de jogos eletrônicos: 80% das aplicações instaladas pelos usuários japoneses são jogos [83]. Isto demonstra que a área de entretenimento para dispositivos móveis promete ser um grande nicho de mercado nessa nova tecnologia, além de um campo aberto à pesquisas, visto que não existem soluções para muitos problemas, como, por exemplo, a falta de suporte de J2ME a requisitos importantes para o desenvolvimento de jogos. Essa situação, como era de se esperar para uma tecnologia em seus primeiros anos de vida, traz grandes desafios, como serão explorados no capítulo 5.

## **4.5 CONCLUSÕES**

Como ilustrado ao longo deste capítulo, J2ME é resultado de um longo processo de evolução da Sun em busca de uma tecnologia capaz de permitir o desenvolvimento em massa de aplicativos voltados para dispositivos móveis. Além da investida tecnológica da Sun, a grande aceitação que J2ME tem alcançado entre os fabricantes de dispositivos e empresas da área mostra que essa tecnologia tem tudo para obter um grande sucesso. Uma análise dos recursos de J2ME para o desenvolvimento de jogos será apresentado no capítulo 5. Essa análise, bem como o resultado obtido com a implementação do wGEM, mostrará que de fato J2ME é uma excelente tecnologia e pode ser utilizada para o desenvolvimento de um grande universo de aplicações.



## 5 DESENVOLVIMENTO DE JOGOS EM J2ME

Dada a novidade da tecnologia J2ME, uma análise crítica sobre a sua utilização para o desenvolvimento de jogos e as conseqüências dessa escolha são abordadas neste capítulo.

Inicialmente o suporte e limitações de J2ME para o desenvolvimento de jogos é apresentado. Em seguida uma análise crítica sobre as vantagens e desvantagens em utilizar *frameworks* para dispositivos móveis é realizada. Por fim, uma reflexão sobre como J2ME deveria ser utilizado por um desenvolvedor de jogos, bem como os jogos considerados ideais para a tecnologia são abordados. Boa parte desta análise é fruto da nossa experiência adquirida ao longo do desenvolvimento do wGEM e de alguns jogos de teste.

### 5.1 RECURSOS DE J2ME PARA O DESENVOLVIMENTO DE JOGOS

Durante esta seção serão apresentados os principais recursos para o desenvolvimento de jogos que são suportados por J2ME, bem como certas limitações, e suas conseqüências, que impactam o desenvolvimento de jogos utilizando esta tecnologia.

#### 5.1.1 Suporte

Em se tratando do uso de J2ME para o desenvolvimento de jogos, mais precisamente da configuração CLDC e do perfil MIDP, uma observação rápida mostra a viabilidade de seu uso. A seguir seguem os principais pontos importantes que são suportados:

- Suporte à programação orientação a objetos (OO), uma vez que os conceitos encontrados em J2SE como Classes, Interfaces, Classes Abstratas e Pacotes, podem ser considerados para uma modelagem OO de um jogo ou *framework*;
- Suporte à manipulação da tela gráfica, fornecido pela classe *Canvas* do perfil MIDP, possibilitando o desenho de imagens e formas geométricas na tela do dispositivo;

- Suporte à manipulação do teclado dos dispositivos, que é um requisito fundamental para um jogo, dada a necessidade de interatividade com o jogador;
- Suporte à conectividade com outros dispositivos, possibilitando que jogos em rede sejam desenvolvidos através do protocolo HTTP fornecido pelo perfil MIDP.

### 5.1.2 Limitações

Porém, existem certas limitações bastante significativas. A seguir estão as mais críticas, além das consequências dessas limitações:

- Ausência de ponto flutuante, o que obriga o programador a adaptar algoritmos já conhecidos para possibilitar o trabalho com aproximações inteiras;
- Ausência de acesso à cor de um pixel da tela, tornando inviável a execução de algoritmos que dependem do acesso ou modificação do conteúdo gráfico de uma imagem ou da tela do dispositivo;
- Ausência de som, tornando os jogos mais monótonos e inviabilizando certos tipos de jogos que dependem de efeitos sonoros;
- Baixo poder de processamento dos dispositivos, em torno de 25 MHz, o que inviabiliza a implementação de jogos mais complexos, principalmente aqueles que necessitam de processamento em tempo real ou certos algoritmos de inteligência artificial;
- Pouca memória, em média 32 KB para execução de programas (memória RAM) e 128 KB para persistência de dados, prejudicando jogos que manipulam muitos objetos ou que precisam armazenar muitos dados em memória;
- Ausência de polígonos, suportando apenas a utilização de primitivas gráficas básicas (elipses, retângulos e linhas).

O problema do baixo poder de processamento dos dispositivos agrava-se ainda mais em razão dos programas em J2ME serem executados por uma máquina virtual JAVA, o que por si só já torna a execução mais lenta do que se o programa fosse compilado diretamente para a linguagem de máquina.

Diante dessas limitações, o desenvolvimento de jogos para dispositivos móveis impõe uma série de restrições e limitações. No entanto, a utilização de um *framework* pode suprir parte desses problemas, tais como a implementação de um mecanismo de desenho de polígonos ou a utilização de números reais.

## 5.2 UTILIZAÇÃO DE FRAMEWORKS EM DISPOSITIVOS MÓVEIS

Nós acreditamos que o uso de um *framework* para jogos em dispositivos móveis, assim como em PCs, é de enorme importância, por prover uma série de recursos úteis ao desenvolvimento profissional de jogos. Porém, devido às limitações dos dispositivos móveis, muitas vezes o uso de um *framework* pode trazer certos problemas, sendo necessária uma visão crítica sobre o custo e benefício associados à sua utilização. Essa análise é apresentada a seguir, onde as vantagens e desvantagens do uso de *frameworks* para jogos em dispositivos são listadas.

### 5.2.1 Vantagens

São várias as vantagens para utilização de um *framework* para o desenvolvimento, em escala industrial, de jogos para dispositivos móveis. Uma delas consiste no fato de que uma das tarefas mais pesadas em um jogo, que são armazenamento e gerenciamento dos objetos (tais como detecção de colisão entre eles e atualização), é implementada no *framework*. Considerando que tais tarefas podem ser bastante otimizadas por quem definiu o motor, o desenvolvedor de um jogo utilizando tal motor pode ter um ótimo ganho de performance em tais tarefas. De forma semelhante, tais recursos podem levar em conta as diversas restrições de memória dos dispositivos e utilizar estruturas de dados econômicas para armazenamento dos objetos do jogo, por exemplo.

Outro ponto importante é que o motor pode ser visto como um complemento para as deficiências encontradas na linguagem de programação para os dispositivos móveis, como a ausência de desenho de polígonos.

Um outra vantagem em utilizar um *framework* é que o mesmo incentiva a organização do projeto, uma vez que ele exige que o projeto adeque-se à arquitetura do *framework*, como apresentado no capítulo 3.

A importância de ferramentas de apoio, como editores de cenários, vem em um fator diferente. Considerando que novos níveis (estágios, missões, etc.) para um jogo podem ser criados mesmo após o jogo ter sido concluído e que os dispositivos móveis têm acesso à rede, tais níveis podem ser instalados pelo jogador, via dispositivo móvel, no futuro de forma bem simples. Isto traz duas vantagens. A primeira é a diminuição do tamanho dos jogos, caso os estágios sejam instalados separadamente via *download*. A segunda é que isso aumenta o interesse pelo jogo, pois caso o jogador já tenha concluído

todos os níveis instalados junto com o jogo ele terá a chance de instalar novos níveis e continuar jogando.

### 5.2.2 *Desvantagens*

Apesar das vantagens explicadas anteriormente serem muito importantes, existem algumas desvantagens que merecem destaque. Todas elas dizem respeito ao *overhead* provocado quando se utiliza um *framework*, uma vez que o motor deverá fazer parte do jogo quando esse tiver que ser instalado no dispositivo. Tal *overhead* torna os jogos em geral maiores do que se fossem criados inteiramente. Este crescimento é natural, pois uma vez que um motor precisa ser genérico, muitas vezes ele provê certas funcionalidades que não são necessárias em certos jogos. No entanto, essa situação pode ser melhorada se o motor for suficientemente modular de forma que o uso de seus módulos possa ser seletivo.

O *overhead* comentado é um fato comum até mesmo aos motores de jogos para PCs, no entanto, dada a enorme capacidade desses computadores, tais excessos não têm grandes efeitos colaterais. Isto não ocorre nos dispositivos móveis, pelo menos nos que já existem atualmente. Devido às limitações existentes, talvez a execução de um jogo baseado em um motor não seja suportada, enquanto que seria a execução do mesmo jogo feito sob medida para tais restrições – contento apenas os recursos necessários e especializados para cada jogo e sacrificando um pouco a engenharia de software para tornar o jogo menor e mais rápido.

## 5.3 *JOGOS RECOMENDÁVEIS PARA DISPOSITIVOS MÓVEIS*

Dadas as limitações dos dispositivos móveis apresentadas, bem como certos recursos não suportados por J2ME, é necessário que alguns cuidados sejam tomados durante a criação de jogos para dispositivos móveis. Estes cuidados são importantes para que o jogo seja adequado para a tecnologia e possa ser utilizado de fato em dispositivos móveis.

Se os recursos disponíveis são limitados, a primeira providência a ser tomada certamente é tornar o projeto simples. Em se tratando de jogos, isso implica em simplicidade em vários níveis - gráficos, lógica, regras, etc. A seguir seguem os pontos mais críticos:

- As regras do jogo para o usuário;
- A interface de entrada;

- Os recursos gráficos;
- A conectividade;
- A complexidade de execução do jogo.

Com respeito às *regras do jogo*, um cuidado especial deve ser dado à sua complexidade. Como os jogos em J2ME serão adquiridos por meio de *download*, não há manual do usuário. Sendo assim, as únicas instruções possíveis são algumas frases a serem mostradas na tela do dispositivo móvel. Portanto, a regra deve ser clara o suficiente para ser especificada em poucas palavras.

Outro ponto que precisa ser evitado é a utilização indevida da *interface de entrada* do dispositivo. Considerando que o jogador precisa segurar o dispositivo enquanto joga, não mais que dois botões devem precisar ser controlados pelo jogador ao mesmo tempo.

A escolha dos *recursos gráficos* utilizados poderá ter grande impacto em portabilidade e jogabilidade. Como cada dispositivo tem seu próprio tamanho de tela e resolução gráfica, cabe ao jogo se adaptar a essas diferenças. Uma boa prática é utilizar apenas imagens simples, em tamanho e cores, para que possam ser mostradas em qualquer dispositivo. Um outro ponto diz respeito ao abuso de recursos gráficos. Como a tela gráfica é pequena e os dispositivos são lentos, o número de objetos desenhados ao mesmo tempo deve ser muito bem observado.

Quanto a questão de *conectividade*, o cuidado necessário é quanto ao seu uso exagerado. Apesar do suporte ao protocolo HTTP de J2ME, a velocidade de conexão atual é muito baixa. Dessa forma, jogos que necessitem de conexão permanente e de boa qualidade, como jogos em rede e em tempo real, certamente não serão viáveis durante um bom tempo. Assim, a conectividade deve ser utilizada com muita cautela. Por exemplo, sendo utilizada apenas para enviar a pontuação do jogador para algum servidor de jogos, ou para fazer *download* de novos estágios do jogo via o mesmo servidor, situações que ocorrem com baixa frequência e totalmente sob o controle do jogador.

Uma outra recomendação é que os jogos devam ter uma *complexidade de execução simples*, pois farão uso de algoritmos rápidos e poderão ser implementados com um pequeno número de classes. Jogos complexos podem até ser executados em certos dispositivos, como um PDA, porém serão muito lentos em um celular, por exemplo.

Considerando todas as recomendações apresentadas, apenas alguns estilos de jogos tornam-se viáveis.

Considerando todas as limitações de J2ME, certamente, durante um bom tempo, o desenvolvimento de jogos 3D deve ser evitado. Sendo assim, o ideal é que os jogos sejam em ambientes 2D, e ainda sigam todos os cuidados apresentados anteriormente.

Jogos clássicos (PacMan, RiverRaid, Resta 1, etc.) são bastante indicados para J2ME, pois todos os pontos considerados nesta seção estão presentes nesses jogos.

Os jogos de turno (Dama, Xadrez, etc.) são também bons estilos de jogos para J2ME. Primeiro por serem simples (desconsiderando uma boa IA), desde que não envolvem o processamento contínuo necessário em jogos em tempo real, apenas em resposta à movimentação de peças. Segundo, por também serem jogos clássicos.

## **5.4 CONCLUSÕES**

Uma conclusão das diversas análises apresentadas ao longo deste capítulo é que J2ME é uma tecnologia que atualmente acompanha uma série de limitações e deve ser utilizada com muita cautela e disciplina, especialmente no que diz respeito ao desenvolvimento de jogos.

Uma consequência que vai além do desenvolvimento de jogos é que as aplicações desenvolvidas devem ser simples. Esta simplicidade, infelizmente, pode representar a troca de uma boa arquitetura de software por outra, menos genérica, organizada e ideal diante dos conceitos da engenharia de software, para que um dado programa seja viável. Essas lições, bastante observadas ao longo do desenvolvimento do wGEM, já são maciçamente ressaltadas nos livros sobre J2ME recentemente publicados [1][10]. Naturalmente, todos esses artifícios devem se tornar menos necessários em razão do avanço dos dispositivos, fato que é natural e deve levar pouco tempo para ocorrer.



## 6 wGEM

Este capítulo apresentará o *framework* wGEM, mostrando as partes que compõem o sistema e a abordagem de implementação de cada uma. Inicialmente será introduzido um breve histórico do seu desenvolvimento, seguido dos requisitos do motor e de seu editor de cenários. Por fim, os detalhes internos na implementação do sistema são apresentados.

### 6.1 DA CONCEPÇÃO AO ESTÁGIO ATUAL

A idéia do sistema surgiu no segundo semestre de 2000 por ocasião da escolha de projetos para a disciplina de graduação Projeto e Implementação de Jogos [7], voltada para a plataforma PC. Durante a mesma época estava surgindo também a linguagem J2ME, que despertou interesse do professor da disciplina, Geber Ramalho e do autor deste trabalho, definindo então uma alternativa de projeto que seria o desenvolvimento de um jogo para esta nova tecnologia.

Quando começamos o trabalho, em meados de outubro de 2000, ainda não existia um *framework* de desenvolvimento de jogos para J2ME, situação que, excetuando o wGEM, não mudou, pois não há relatos de nenhum outro *framework* do gênero.

O projeto de desenvolver um jogo era complexo, pois toda uma adaptação visando cobrir as limitações citadas na seção 5.1.2 teria que ser realizada. Por isso, o projeto da disciplina passou também a ser o Trabalho de Graduação do autor e incluiu o desenvolvimento de um motor de jogos para J2ME assim como um jogo para validação deste motor.

Durante essa fase, o projeto despertou interesse comercial do CESAR [38], que passou a financiar o trabalho já prevendo uma criação futura de um grupo de desenvolvimento de jogos para dispositivos móveis. Desse desenvolvimento surgiu o primeiro protótipo do wGEM [19].

Motivados pelo bom resultado alcançado durante o Trabalho de Graduação, da grande perspectiva da tecnologia, e do interesse do autor e do orientador na área de jogos, decidimos continuar o trabalho mesmo após a conclusão do curso de graduação.

A tendência natural foi então utilizar um curso de mestrado como forma de aprofundamento de diversos conceitos que fundamentam este trabalho, melhorando cada vez mais o sistema.

Neste período, o wGEM passou por grandes transformações. Destacando-se uma série de otimizações, inclusão de novas funcionalidades e o desenvolvimento de um editor de cenários. Esse *framework*, inédito para J2ME, resultou na submissão e apresentação de um tutorial no SIBGRAPI [28], que foi também publicado em uma revista nacional [29].

## 6.2 REQUISITOS

Durante esta seção apresentaremos os requisitos funcionais identificados para o wGEM. Conceitualmente, esses requisitos deveriam ser similares aos apresentados na seção 3.4, quando os requisitos gerais para um *framework* foram listados. No entanto, dadas as limitações de J2ME para o desenvolvimento de jogos apresentadas na seção 5.1, um processo de adaptação é necessário. A seguir serão apresentados o resultado desse processo, primeiramente para o motor e em seguida para o editor de cenários.

### 6.2.1 Motor

Dada a novidade de J2ME, não existe ainda um consenso sobre o que seria um motor de jogos padrão baseado nessa tecnologia. No entanto, após o estudo e adaptação das soluções para PCs (ver seção 3.4), nós definimos um conjunto mínimo de componentes e requisitos que tal motor deve exibir. Na Tabela 6-1 são listadas as principais funcionalidades que acreditamos estar presentes em um motor 2D baseado em J2ME.

É importante ressaltar que os requisitos de um motor são geralmente baseados nas tarefas envolvidas *durante* a execução do ciclo do jogo. Com isso, características como telas de abertura do jogo, janelas de escolha de dificuldade, estágios e várias outras, que são utilizadas *antes* ou *depois* do jogo ser iniciado, são implementadas pelos jogos desenvolvidos com o motor, e não pelo motor.

<b>Representação do Objeto do Jogo</b>	Os objetos do jogo (tais como, carro, bola, etc.) precisam de uma representação oferecida pelo motor que contenha tanto as informações necessárias para o seu gerenciamento (como posição, velocidade, dimensão e representação gráfica), quanto as funcionalidades necessárias para a execução do jogo (como detecção de colisão com outros objetos).
<b>Representação do Mapa do Jogo</b>	O motor deve prover a utilização de mapas <sup>1</sup> para facilitar o desenvolvimento de jogos com cenários complexos. Recursos como <i>scrolling</i> <sup>2</sup> e mapas baseados em texturas ou <i>tiles</i> <sup>3</sup> devem ser oferecidos para tornar simples a criação de jogos de plataforma no estilo Sonic© ou Super Mario©, por exemplo.
<b>Gerenciamento de Objetos</b>	O gerenciamento de objetos é uma das partes mais fundamentais de um motor. As suas responsabilidades envolvem a inicialização, o armazenamento e o acesso aos objetos do jogo que ele gerencia.
<b>Gerenciamento de Entrada</b>	Encarregado de identificar eventos ocorridos no teclado do dispositivo e encaminhá-los para o módulo responsável pelo gerenciamento do jogo.
<b>Gerenciamento Gráfico</b>	Responsável pela fase de apresentação gráfica do ciclo do jogo. O que envolve a coordenação do processo de desenho do mapa e dos objetos do jogo.
<b>Gerenciamento do Jogo</b>	O seu papel principal é implementar o ciclo do jogo, contando para isso com o apoio dos módulos de gerenciamento de objetos, do mapa do jogo, de entrada e gráfico.
<b>Gerenciamento de Rede</b>	Responsável por implementar as funcionalidades de <i>download</i> e <i>upload</i> de arquivos utilizando o protocolo HTTP suportado pelo perfil MIDP.
<b>Gerenciamento de Aplicação</b>	Encarregado de implementar as funcionalidades exigidas pela KVM às aplicações do perfil MIDP para que estas possam ser gerenciadas.
<b>Integração com o Editor de Cenários</b>	Deve oferecer funcionalidades necessárias para importação do arquivo produzido pelo editor de cenários.

Tabela 6-1: Componentes e funcionalidades para o motor do *framework* wGEM

### 6.2.2 Editor de Cenários

Baseado nos módulos identificados para o motor de jogos, os principais requisitos para o editor de cenários podem ser considerados. Na Tabela 6-2 são listadas as principais funcionalidades necessárias para tal editor.

<sup>1</sup> O mapa de um jogo 2D corresponde a um retângulo, geralmente maior do que a tela gráfica, sobre o qual os objetos do jogo estão localizados e se movem. Associado a cada mapa sempre existe uma janela visível, que é a região do mapa visível na tela gráfica.

<sup>2</sup> Scrolling representa a mudança de posição da janela visível do mapa.

<sup>3</sup> Uma técnica comum para a representação gráfica do mapa do jogo é dividi-lo em pequenas partes e associar pequenas imagens (tiles) a cada parte.

<b>Definição do Mapa</b>	Através da interface deve ser possível especificar o tipo de mapa utilizado no jogo (baseado em textura, <i>tiles</i> , etc.) além de características como altura, largura, imagem de textura e imagem para <i>tiles</i> .
<b>Definição dos Objetos</b>	O editor deve prover um mecanismo onde os objetos do jogo a serem utilizados em cada cenário possam ser definidos pelo usuário. Todos os tipos de representação gráfica de objetos utilizados no motor devem poder ser especificados (baseados em imagens, formas geométricas, etc.). Os objetos definidos devem ser adicionados a uma paleta de objetos que será utilizada para definição do cenário.
<b>Definição do Cenário</b>	A definição do cenário deve ser realizada utilizando recursos de <i>drag and drop</i> , permitindo que os objetos definidos e presentes na paleta de objetos sejam facilmente adicionados, removidos e deslocados dentro do cenário. Além disso, para cada objeto presente no cenário, características como velocidade do objeto deverão poder ser editadas.
<b>Persistência de Cenários</b>	O editor deve ser capaz de salvar e carregar cenários desenvolvidos, para facilitar a continuidade e reutilização do trabalho.
<b>Integração com o Motor</b>	A exportação dos cenários desenvolvidos para um arquivo em formato compatível com o motor deve ser provida.

Tabela 6-2: Funcionalidades do editor de cenários do wGEM

### 6.3 IMPLEMENTAÇÃO DO MOTOR

Nesta seção será apresentada a abordagem que adotamos para o desenvolvimento do motor de jogos do *framework* wGEM.

O primeiro passo foi a definição da arquitetura, cujo resultado é apresentado na Figura 6-1.

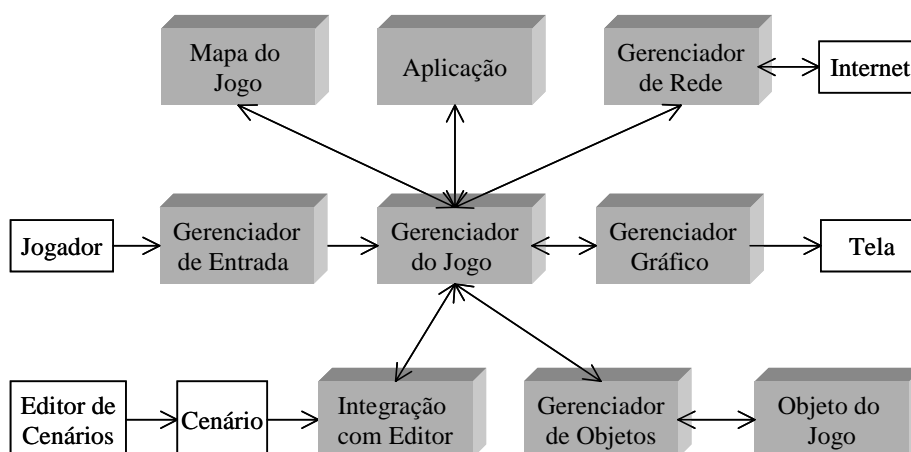


Figura 6-1: Comunicação entre os módulos do motor do wGEM

Esse modelo foi fruto de nossos estudos e generalizações de soluções para PCs e representa a nossa proposta de implementação e combinação dos componentes da Tabela 6-1 para produzir o motor de jogos para o wGEM. Entre os principais pontos dos sistemas equivalentes que estudamos para a construção do nosso modelo estão o

mapeamento dos requisitos da Tabela 6-1 para os módulos do motor [4] e a forma de cooperação entre eles para a implementação do ciclo do jogo [17].

Uma vez definida a arquitetura do motor, a próxima fase corresponde à implementação dos seus módulos, como apresentado na seção 3.4.2. Os princípios gerais de desenvolvimento de software de qualidade continuam aplicáveis à J2ME, especialmente os conceitos de reusabilidade e modularidade. No entanto, uma recomendação já bastante pregada em livros sobre J2ME é evitar o excesso de modularidade. Já que o grande número de classes tem forte impacto no tamanho final do programa, podendo assim comprometer o seu uso nos dispositivos<sup>1</sup>. Além disso, como já foi mencionado no capítulo 5, desenvolver aplicações, especialmente jogos, utilizando J2ME requer uma série de adaptações devido às limitações da plataforma. Isto leva a uma série de cuidados especiais que precisam ser tomados durante a implementação dos módulos. A seguir são listados alguns deles:

- Utilização de estruturas de dados eficientes;
- Otimização de partes críticas;
- Fraco acoplamento entre os módulos.

A utilização de uma *estrutura de dados* eficiente no motor é de extrema importância. Basta considerar, por exemplo, o gerenciador de objetos, que precisa armazenar e prover acesso rápido a todos objetos a ele associados. A utilização das estruturas de dados fornecidas por J2ME, como *hashtables* e vetores, não são adequadas, pois geram um excesso de uso de memória e processamento para a CPU.

Um esforço especial deve ser dedicado à *otimização* de partes do motor que sejam críticas para a velocidade de execução do jogo. Por exemplo, o algoritmo de detecção de colisão entre dois objetos deve ser executado muito rápido, dada a enorme frequência que ele é utilizado nos jogos.

Outro aspecto importante é *desacoplar* o máximo possível os componentes do motor para que o seu uso possa ser seletivo. Isto implica em menores programas e, conseqüentemente, menor espaço em memória. Como exemplo pode ser citado o módulo de rede do motor, que só é necessário em jogos em rede. Se não for o caso, o motor deve ser desacoplado o suficiente para que esse módulo não precise ser adicionado ao jogo.

---

<sup>1</sup> Já que alguns dispositivos com J2ME só permitem a instalação de programas de no máximo 10KB.

A seguir esses cuidados serão bastante exemplificados, quando as abordagens que utilizamos para a implementação dos módulos do motor serão apresentadas, assim como as principais informações sobre as classes do sistema.

### 6.3.1 *Objeto do Jogo*

A implementação do objeto do jogo envolve basicamente os seguintes problemas:

- Representação interna;
- Representação visual;
- Representação comportamental;
- Detecção de colisão com outros objetos.

A *representação interna* do objeto é formada por um conjunto de informações necessárias para sua manipulação pelo motor. Dentre essas informações estão a localização espacial do objeto relativa ao mapa do jogo (posição vertical e horizontal no mapa) e velocidade vertical e horizontal.

A *representação visual* do objeto define a forma em que ele deve ser apresentado graficamente na tela. No wGEM esta representação pode ser uma imagem, um retângulo, um círculo, um polígono ou uma imagem animada.

A *representação comportamental* do objeto indica como a sua representação interna e visual evoluem ao longo do jogo (por exemplo, como a sua posição e velocidade evoluem ao longo do jogo, quando ele deve ser destruído ou desativado, etc.). O comportamento padrão oferecido pelo objeto do jogo no wGEM consiste em apenas alterar a posição do objeto de acordo com a sua velocidade e posição atual. Naturalmente, cada tipo de objeto utilizado em um jogo pode redefinir esse comportamento através dos mecanismos de herança de classes e redefinição de métodos.

A capacidade de *detecção de colisão* com outros objetos do jogo é uma funcionalidade fundamental provida pelo wGEM. Isso se deve ao fato que a maioria dos eventos que ocasionam a mudança do estado interno dos objetos em um jogo tem relação direta com colisões detectadas.

A classe responsável pelo objeto do jogo é ilustrada na Figura 6-2. Atributos como `x`, `y`, `xVel` e `yVel` são a representação interna do objeto. O método

`paint(Graphics)`, responsável pode desenhar o objeto no `Graphics` recebido, apenas repassa a operação para o atributo `painter`, que desenha o objeto na tela de acordo com posição e representação visual do objeto. O método `update` implementa o comportamento do objeto, enquanto que o método `intersects(GameObject)` realiza a detecção de colisão com outro objeto.

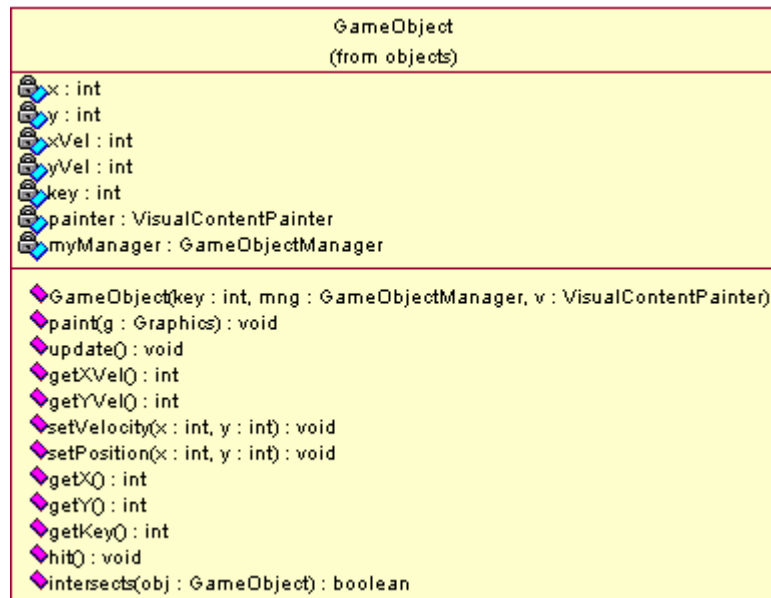


Figura 6-2: Classe `GameObject`

A seguir são detalhadas como as formas de representação visual dos objetos e a técnica de detecção de colisão foram implementadas.

### 6.3.1.1 Representação Visual

A representação visual de cada objeto é definida pela classe `VisualContentPainter` do wGEM. Esta classe é abstrata e define o método `paint(graphics, x, y)`, que precisa ser implementado por cada uma das classes que estendem esta classe base para desenhar a representação visual na posição especificada. Este método é chamado pelo objeto do jogo em resposta ao seu método `paint`. Neste método, ele apenas repassa a tarefa para o seu objeto `painter`, informando a sua localização para que o desenho ocorra na devida posição da tela. A Figura 6-3 ilustra a classe `VisualContentPainter` e todas as suas extensões oferecidas pelo wGEM para definir os tipos de representação gráfica disponíveis.

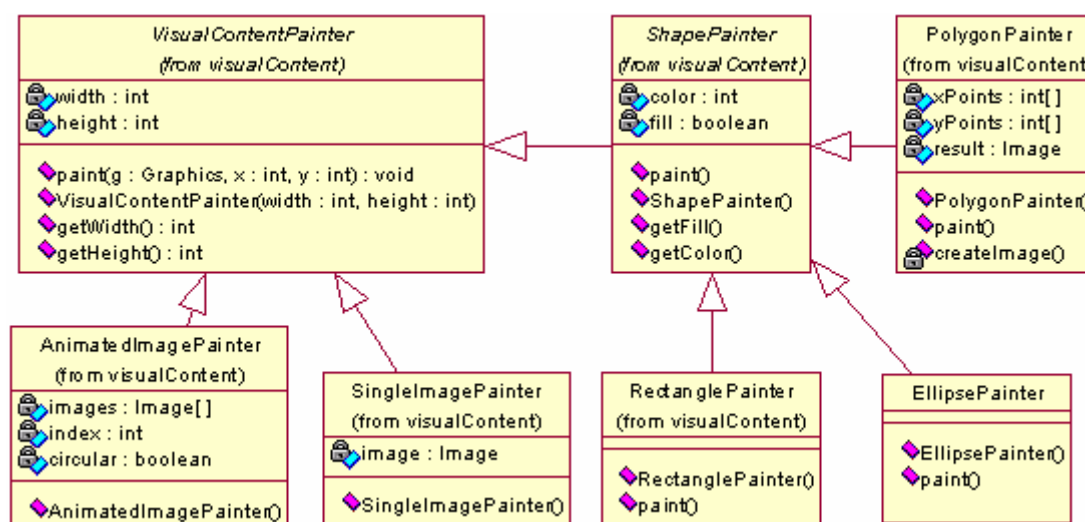


Figura 6-3: Classes para representação visual dos objetos do jogo

Uma das classes da Figura 6-3, *ShapePainter*, também é abstrata, servindo apenas para concentrar as propriedades comuns a todas as representações gráficas baseadas em formas geométricas, que são cor de desenho e tipo de pintura (cheia ou vazia). As classes *SingleImagePainter*, *RectanglePainter* e *EllipsePainter* são bastante simples, já que apenas repassam a tarefa para as classes do perfil MIDP que realizam esses tipos de desenho. A classe *AnimatedImagePainter* é um pouco mais complexa, pois tem o papel de animar e apresentar uma sequência de imagens (representando os quadros de uma animação de explosão, por exemplo) a uma dada taxa especificada pelo jogo. A classe *PolygonPainter* é certamente a classe mais complexa. Ela é capaz de desenhar polígonos, cheios ou vazios, baseado na definição dos seus vértices que são passados como argumento na criação do objeto. Para a sua implementação foi utilizado o algoritmo de *Bresenham* [76][77] para o desenho das arestas do polígono e o algoritmo *Boundary Fill* [76] para o preenchimento do seu interior quando ele for cheio. Esse processo manual de desenho de polígono é bastante custoso para ser realizado *on-line* durante a execução do jogo. Em função disso, o objeto *PolygonPainter*, tão logo é instanciado, cria o polígono e o desenha em uma imagem que será então utilizada em resposta a futuros pedidos de desenho do polígono. Como resultado dessa otimização, o uso de polígonos no wGEM oferece o mesmo desempenho de velocidade de uma imagem, já que a criação do polígono (que é lenta) é realizada *off-line* durante a inicialização do objeto, o que não impacta o ciclo de execução do jogo.



É interessante observar que a representação gráfica do objeto é independente do próprio objeto, já que ele, em resposta a uma chamada ao seu método `paint`, apenas redireciona essa tarefa para o seu objeto do tipo `VisualContentPainter`, que pode ser qualquer um dos ilustrados na Figura 6-3 e que foi passado como argumento para o construtor do objeto.

### 6.3.1.2 Detecção de Colisão

A técnica utilizada no wGEM para detecção de colisão entre dois objetos é bastante simples, e segue o que é usado mais comumente em jogos 2D [4][78]. Ela consiste na utilização de um retângulo imaginário, englobando cada objeto, cuja dimensão é a mesma da representação gráfica do objeto e a posição é definida pela posição do objeto (atributos  $x$  e  $y$ ). Então, o teste de colisão entre dois objetos do jogo é satisfeito se os seus respectivos retângulos se intersectarem.

Essa aproximação dos objetos por formas geométricas mais simples é bastante utilizada em jogos para permitir uma maior velocidade na detecção de colisão, uma vez que tal aproximação evita que um teste pixel a pixel entre imagens, ou que a interseção de formas geométricas complexas precise ser realizada. Em jogos 3D esta técnica também é aplicada, porém com a diferença que em vez de retângulos são utilizados esferas, cubos ou cilindros. Uma desvantagem natural dessa simplificação é que a sua corretude depende da forma geométrica real do objeto, como ilustrado na Figura 6-4, onde existem três casos em que os retângulos que aproximam os objetos colidem, e na verdade apenas no terceiro caso existe uma colisão real entre os objetos.



Figura 6-4: Uso de retângulos para testar colisão entre objetos

Realizada a aproximação dos objetos por retângulos, resta descobrir se os retângulos se intersectam. Para essa tarefa existem diversos algoritmos [4][78]. Testar se uma das arestas de um dos retângulos cruza com uma das arestas do outro é uma das maneiras. Outra menos complexa é verificar se um dos vértices de um dos retângulos pertence ao outro retângulo. No entanto, elas falham em certos casos especiais, como mostrado na Figura 6-5. Na ilustração à esquerda, o teste de cruzamento entre uma das

arestas do retângulo A com uma das arestas do retângulo B falha. A segunda ilustração mostra um exemplo de falha do teste de colisão baseado na pertinência de um dos vértices do retângulo A ao retângulo B.



Figura 6-5: Exemplos de falhas na detecção de colisão

Uma solução mais eficiente e correta é baseada no teste de interseção entre as projeções dos lados dos retângulos sobre os eixos vertical e horizontal [78]. Visto que, para haver interseção entre os retângulos, é preciso que as projeções também tenham pontos em comum. Além de ser correta, essa solução é muito eficiente, pois testar se dois intervalos de números tem algum ponto em comum é muito simples. A Figura 6-6 ilustra um exemplo do uso dessa técnica, bem como o algoritmo para teste de interseção. É interessante observar que esse algoritmo na verdade descobre se não há interseção entre os intervalos. Somente nos casos desses testes de não interseção falharem é que a colisão acontece. A grande vantagem disso é que nos casos da não existência de colisão entre os objetos (a grande maioria das vezes em que o algoritmo é executado) o algoritmo termina muito rápido.

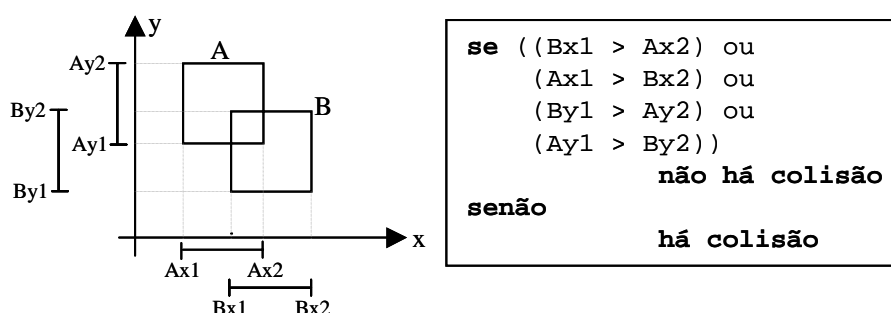


Figura 6-6: Teste de colisão baseado em projeção sobre os eixos

### 6.3.2 Mapa do Jogo

O mapa do jogo representa o “pano de fundo” sobre o qual os objetos do jogo se movimentam e são desenhados. Os problemas envolvidos em sua implementação são:

- Representação interna;
- Representação visual;
- Representação comportamental;

- Inicialização;
- *Scrolling*.

A *representação interna* do mapa é formada por um conjunto de informações como largura e altura do mapa, local da janela visível e velocidade horizontal e vertical do *scrolling*<sup>1</sup>.

A *representação visual* do mapa define como a sua região contida na janela visível deve ser apresentada graficamente para o jogador. No wGEM esta representação pode ser uma imagem, uma matriz de *tiles* ou um retângulo.

A *representação comportamental* representa como o mapa evolui ao longo do jogo. Esta evolução consiste basicamente na realização de *scrolling* para a direita, esquerda, etc. O comportamento automaticamente implementado pelo wGEM não realiza mudanças no mapa, devendo ser redefinido por cada jogo que precise de um comportamento especial.

A *inicialização* do mapa representa o processamento do arquivo exportado pelo editor de cenários para que o mapa seja inicializado devidamente.

A capacidade de *scrolling* do mapa é fundamental para os jogos de plataforma. Nesses jogos, como SuperMario© e Sonic©, o cenário de fundo do jogo muda à medida que o objeto controlado pelo jogador alcança novas regiões do mapa.

O conjunto de classes que implementam os tipos de mapas disponíveis no wGEM são ilustradas na Figura 6-7. A classe `Map`, abstrata, contém as características comuns aos três tipos de mapa disponíveis. Atributos como `mapWidth`, `mapHeight` e `visibleWindowLeft` são exemplos da representação interna do mapa. O método `paint` é abstrato e é implementado por cada estilo de mapa do wGEM para que o mesmo seja desenhado apropriadamente. O método `update` define o comportamento do mapa. O método `loadLevel` realiza a inicialização do mapa de acordo com o cenário definido pelo editor de cenários, que também é abstrato e implementado pelos estilos específicos de mapa. Métodos como `scrollUp`, e similares, são os responsáveis pelo *scrolling* do mapa.

---

<sup>1</sup> A velocidade horizontal ou vertical do *scrolling* indica a quantidade de deslocamento da janela visível para cada *scrolling* realizado nesses respectivos eixos.

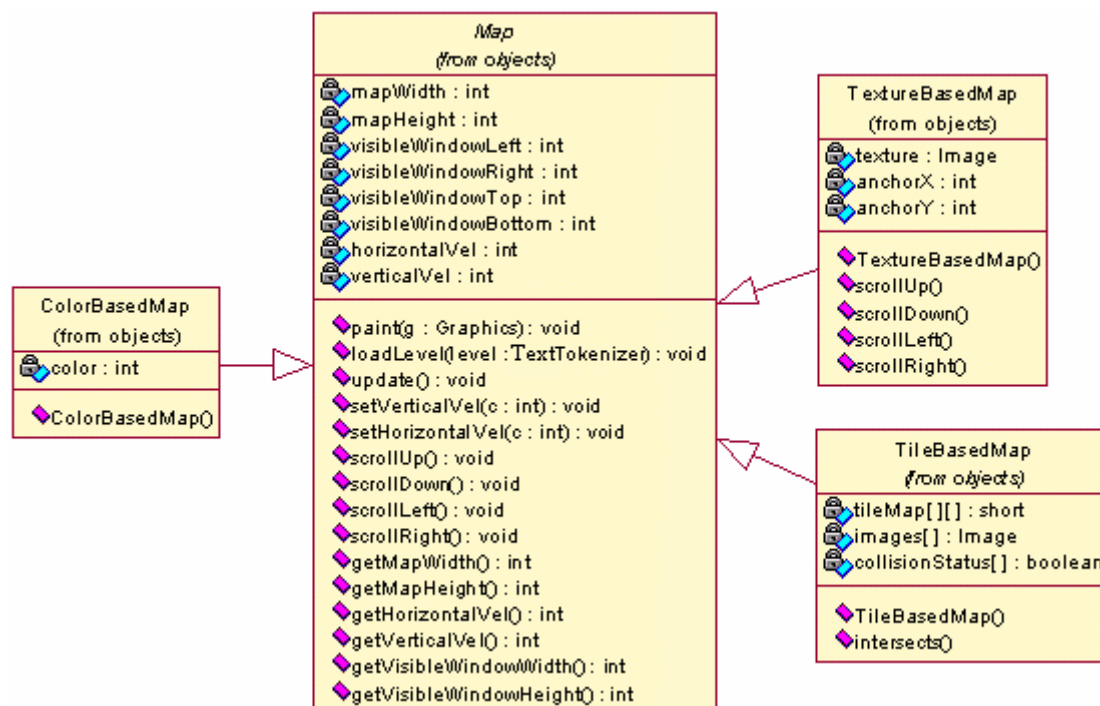


Figura 6-7: Classes do wGEM para a implementação do mapa do jogo

A seguir são detalhados os três tipos de mapas oferecidos pelo wGEM através da extensão da classe abstrata `Map`. A diferença central entre eles é apenas a implementação dos métodos abstratos `paint` e `loadLevel`.

### 6.3.2.1 Mapa baseado em cores

O mapa baseado em cores, implementado pela classe `ColorBasedMap` da Figura 6-7, é o mais simples dos mapas, servindo para jogos cujo cenário de fundo é apenas um retângulo de uma única cor (definida pelo atributo `color`). Por ser tão simples graficamente, o seu desenho é muito rápido, pois o único processamento envolvido é o desenho de um retângulo do tamanho e posição definidos pela janela visível. De forma análoga, a sua inicialização é muito simples, pois apenas a sua cor é necessária<sup>1</sup>.

### 6.3.2.2 Mapa baseado em texturas

Este estilo de mapa provê um mecanismo para a utilização de uma imagem para desenho da janela visível. Essa imagem deve ter a aparência de uma textura com características especiais, para que o desenho sucessivo de partes diferentes dela possa dar a impressão de *scrolling*. Para isso, a imagem deve ter o dobro do tamanho da janela visível do mapa e suas metades superior/inferior e esquerda/direita devem ser idênticas.

<sup>1</sup> Além das informações necessárias para a classe `Map`, como largura, altura, janela visível, etc.

Um exemplo de imagem com essas propriedades, dividida em quatro partes para ilustração da simetria, é apresentado na Figura 6-8. A idéia de utilização dessas imagens não é representar exatamente uma parte do mapa, já que a imagem não tem relação com a posição da janela visível. A idéia principal é apenas dar a impressão de deslocamento quando o *scrolling* acontece. Para isso, um *scrolling* para a direita, por exemplo, resulta em desenhar uma parte da imagem mais à direita do que a última desenhada, e assim por diante até que a extremidade direita da imagem seja alcançada. Nesse momento, devido a simetria da imagem, a parte esquerda da imagem volta a ser desenhada e todo o ciclo se repete. Esse tipo de animação é bastante comum em jogos de nave no espaço, onde a imagem desenhada consiste em uma série de estrelas em um fundo preto.

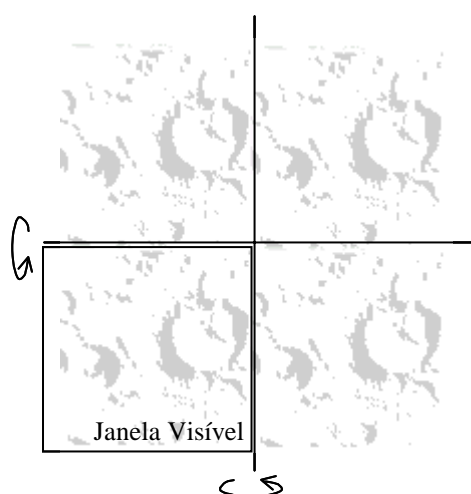


Figura 6-8: Exemplo de imagem para mapas baseados em texturas

A classe que representa este estilo de mapa, `TextureBasedMap`, é ilustrada na Figura 6-7. Os atributos `anchorX` e `anchorY` definem qual parte da imagem (representada pelo atributo `texture`) será desenhada para representar a janela visível. Para fazer com que essa imagem simule o *scrolling* da forma que foi explicada, esse mapa redefine os métodos de *scrolling* da classe `Map` para que os atributos `anchorX` e `anchorY` sejam atualizados adequadamente e a parte apropriada da textura seja desenhada no método `paint`. O método de inicialização desse mapa é similar ao mapa baseado em cores, com a diferença que ele lê a imagem que deve ser utilizada como textura.

### 6.3.2.3 Mapa baseado em *tiles*

O último, e mais complexo, estilo de mapa é baseado em *tiles*. Dos três tipos de mapa, este é de fato o único capaz de representar graficamente cenários complexos. Isto

é realizado pela divisão do mapa em pedaços (*tiles*) de tamanhos iguais e com associação de uma imagem a cada um deles. Essas imagens normalmente são reutilizadas por vários outros *tiles* para que o cenário seja montado. Para desenhar uma grande área gramada, por exemplo, apenas uma imagem de um retângulo com gramas é necessária para que todo o gramado seja desenhado por meio de vários *tiles* baseados nessa mesma imagem. Um exemplo claro dessa técnica é mostrado na Figura 6-9, onde um cenário é definido a partir de poucas imagens.

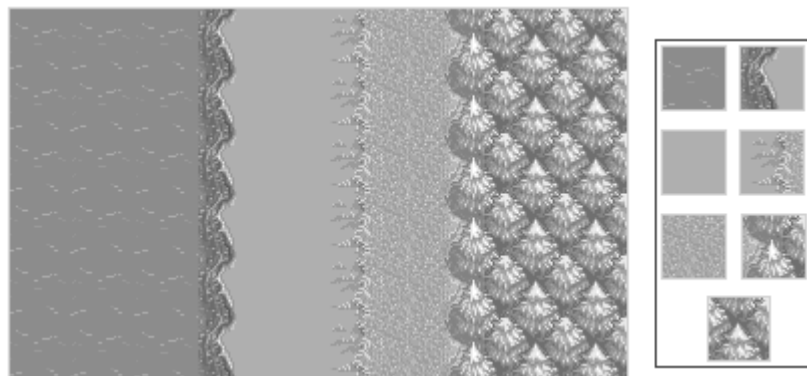


Figura 6-9: Exemplo de uso de *tiles* na montagem de um cenário

Com este estilo de mapa, o desenho da janela visível corresponde aos *tiles* pertencentes à janela, o que é simples de descobrir matematicamente, dada a igualdade entre os tamanhos dos *tiles*.

Em geral são também associadas informações extras aos *tiles*, como por exemplo, se os *tiles* devem permitir ou impedir que o “jogador” passe por cima deles, útil para *tiles* que representam objetos intransponíveis como pedra, madeira, etc.

A Figura 6-7 ilustra a classe `TileBasedMap`, provida pelo wGEM para este estilo de mapa. O atributo `images` armazena o conjunto de imagens utilizadas pelos *tiles* do mapa. O atributo `tileMap` é a matriz de *tiles* do mapa, cada um dos seus elementos armazena um índice do *array* `images` que define a imagem associada a ele. O atributo `collisionStatus` define se cada uma das imagens do *array* `images` representa uma área transponível ou não. O método `paint` descobre que *tiles* são visíveis e os desenha na tela, enquanto que o método `loadLevel` realiza a devida inicialização de todos os atributos do mapa de acordo com a especificação encontrada no arquivo do cenário. O método `intersects(GameObject)` informa se o objeto informado colide com algum *tile* intransponível.

Naturalmente, cada jogo pode estender essa classe e definir novas propriedades aos *tiles*, tal como dificuldade de deslocamento pelo *tile* (útil para diferenciar o deslocamento por água, terra, montanha, etc.).

### 6.3.3 Gerenciador de Objetos

O gerenciador de objetos tem objetivos muito importantes no wGEM, a seguir eles são listados:

- Armazenamento dos objetos do jogo;
- Atualização e desenho dos objetos visíveis;
- Teste de interseção entre os objetos visíveis e algum objeto especificado;
- Inicialização.

O *armazenamento dos objetos do jogo* requer uma estrutura de dados simples e eficiente para armazenamento e consulta aos objetos gerenciados por cada gerenciador do jogo. Como as estruturas de dados de J2ME não são muito adequadas, uma estrutura própria foi implementada e será explicada na próxima seção.

A *atualização e desenho dos objetos visíveis* representa o suporte oferecido pelo gerenciador de objetos ao gerenciador do jogo para que o ciclo do jogo seja implementado. Como cada gerenciador de objetos cuida de um grupo de objetos do jogo, a fase de atualização e desenho de todos os objetos visíveis (que estejam dentro da janela visível do mapa), envolve a participação de cada gerenciador de objetos, já que ele é a fachada de acesso a todos os objetos que ele gerencia.

O *teste de interseção entre os objetos visíveis e algum objeto especificado* é fundamental para a implementação dos jogos, tendo a função de descobrir e retornar algum dos objetos visíveis do gerenciador de objetos que colida com um dado objeto informado.

A *inicialização* do gerenciador de objetos consiste no processamento do arquivo gerado pelo editor de cenários para que todos os objetos gerenciados por ele sejam criados de acordo com as características especificadas.

A classe responsável por implementar o gerenciador de objetos, `GameObjectManager`, é ilustrada na Figura 6-10. O atributo `objects` representa o repositório dos objetos armazenados por cada gerenciador de objetos, cuja estrutura interna será explicada na próxima seção. Os atributos `boxLeft` e similares são utilizados

para a implementação do método `getIntersection(GameObject)`. Os métodos `paint(Graphics)` e `update` realizam as chamadas desses respectivos métodos em todos os objetos do repositório que estejam visíveis no mapa. O método `getIntersection(GameObject)`, cuja implementação será detalhada na seção 6.3.3.2, tem o papel de retornar algum dos seus objetos visíveis que colida com o objeto passado como parâmetro. A inicialização do gerenciador é definida pelo método `loadLevel`, que implementa a inicialização dos objetos de acordo com o arquivo exportado pelo editor. Durante essa inicialização, cada objeto especificado é criado através do método `createGameObject`. Como este método apenas pode criar um objeto do tipo `GameObject`, e nos jogos alguns gerenciadores podem lidar com objetos especiais (que estendem a classe `GameObject`), este método é escrito separadamente para que possa ser reescrito pelos gerenciadores de objetos dos jogos que necessitem criar os seus objetos especiais nesse momento.

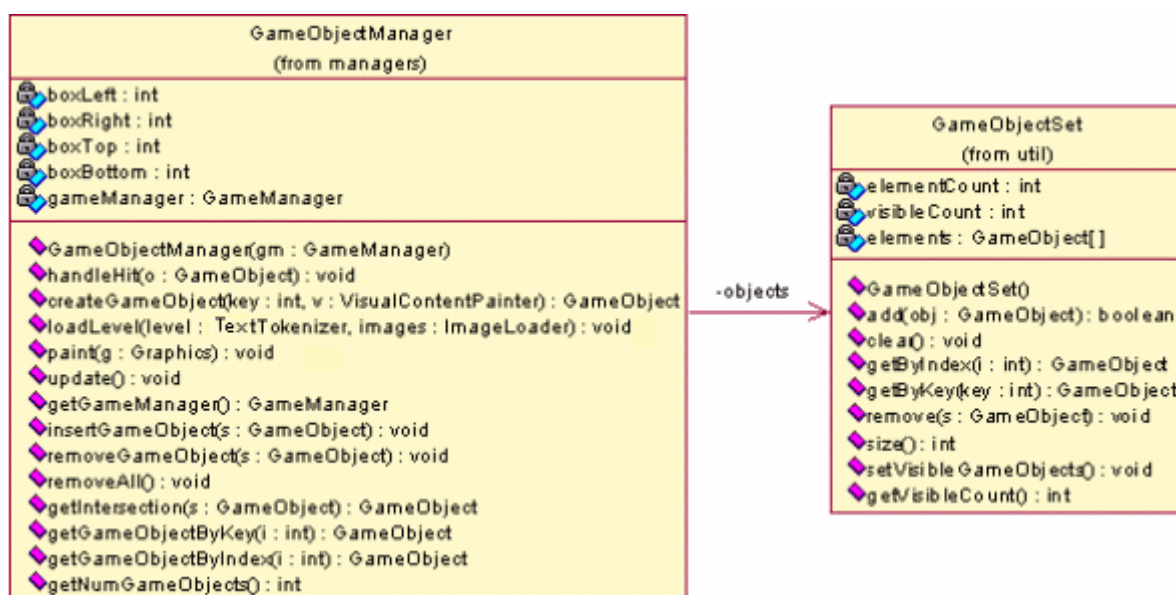


Figura 6-10: Classes para implementação do gerenciador de objetos

### 6.3.3.1 Repositório de objetos

O repositório de objetos, implementado pela classe `GameObjectSet` da Figura 6-10, provê métodos para inserção, remoção e busca de elementos. A sua implementação utiliza um *array* dinâmico, que traz os benefícios de um *array* (acesso imediato a objetos em uma dada posição) bem como de uma lista encadeada (crescimento dinâmico). Essa estrutura também é reorganizada automaticamente a cada objeto removido para manter todos os objetos armazenados continuamente.



Outra funcionalidade provida pela classe `GameObjectSet` é o método `setVisibleGameObjects`, que tem a função de ordenar todos os seus objetos para que os que estejam dentro de uma dada região retangular sejam movidos para o início da estrutura. Esse processo, exemplificado na Figura 6-11, é utilizado para agilizar o acesso a todos os objetos que estejam internos à janela visível do mapa, situação necessária para os métodos `paint`, `update` e `getIntersection`. Para isso, antes da execução de cada ciclo do jogo, o repositório é ordenado para que o gerenciador de objetos não precise mais se preocupar com visibilidade ou não de seus objetos. Com isso, a execução do método `paint`, `update` e `getIntersection` só precisa executar a iteração entre os  $n$  primeiros objetos do repositório (onde  $n$  é o número de objetos visíveis que foram deslocados para o início da estrutura pelo método `setVisibleGameObjects`). Essa otimização torna-se muito importante especialmente quando existe um grande número de objetos armazenados no repositório e apenas uma pequena parte deles é visível a cada ciclo do jogo (situação muito comum em jogos de plataforma).

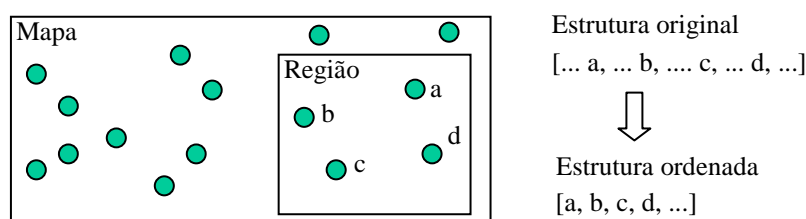


Figura 6-11: Organização dos objetos de acordo com a pertinência a uma região

### 6.3.3.2 Teste de interseção

A implementação do método `getIntersection(GameObject)` conta com outra otimização bastante significativa. A busca de um objeto dentro de um conjunto que colida com um dado objeto sendo testado, função do método `getIntersection`, é bastante explorada na literatura e existem diversas soluções para evitar a busca sequencial por todos os objetos do conjunto até que um deles colida com o objeto em teste [79]. Essas soluções em geral baseiam-se em organizar os objetos a serem testados em estruturas de dados hierárquicas de acordo com relações geométricas entre os objetos, como à esquerda de, acima de, etc., para acelerar o processo de busca de forma similar ao alcançado por uma árvore de busca binária, onde um dado ramo da árvore pode ser ignorado se alguma relação não for satisfeita pela raiz daquele ramo. No entanto, essas estruturas exigem bastante memória e processamento para manter a sua

hierarquia atualizada todas as vezes que um dos objetos se move ou é removido da árvore, o que é um grande problema para as limitações de J2ME.

Adotamos no wGEM uma estratégia mais simples, e que também evita, em alguns casos, que o gerenciador de objetos realize o teste de colisão entre todos os seus objetos visíveis e o objeto sendo testado. Para compreender nossa abordagem, considere a Figura 6-12, em que o gerenciador de quadrados precisa descobrir (na área da janela visível) se o círculo colide com algum dos quadrados. Nesta figura, fica claro que não há colisão. Porém, o gerenciador dos objetos quadrados, não tendo como saber disso, precisaria testar a colisão com todos os quadrados dentro da janela visível.

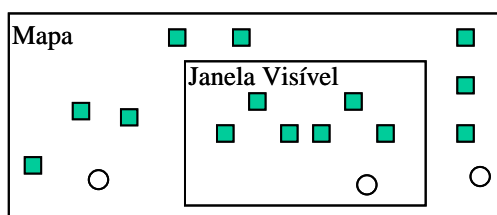


Figura 6-12: Exemplo de busca por um objeto em colisão

A otimização consiste em utilizar um retângulo em cada gerenciador de objetos que englobe minimamente todos os objetos visíveis. A dimensão e posicionamento de tal retângulo são atualizados antes de cada ciclo do jogo durante a fase em que os objetos visíveis são detectados. Um exemplo desse retângulo para o caso apresentado na Figura 6-12 é ilustrado na Figura 6-13. Com essa técnica, chamada de *bouding box* ou *bouding volume* [82], o gerenciador pode eliminar casos triviais de não colisão sem precisar do teste individual com todos os objetos visíveis. Pois, se o objeto não colidir com a área definida pelo retângulo, ele não colidirá com nenhum objeto visível. Somente nos casos da colisão com o *bouding box* ocorrer é que o teste individual com cada objeto visível é realizado.

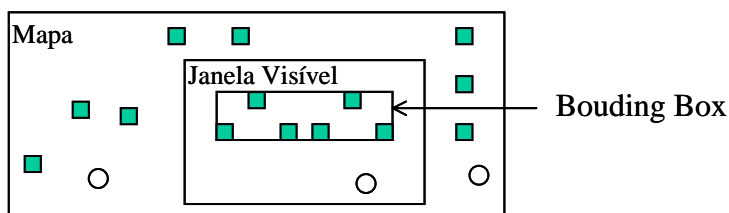


Figura 6-13: Exemplo de bouding box para busca de um objeto em colisão

### 6.3.4 Gerenciador de Entrada

A tarefa básica do gerenciador de entrada é receber eventos relacionados ao pressionamento de teclas do dispositivo e encaminhar o evento para o gerenciador do

jogo. Como as ações a serem tomadas em resposta a esses eventos dependem de cada jogo, o tratamento realizado pelo gerenciador do jogo deve ser especializado para cada jogo.

A classe responsável pelo gerenciamento de entrada, `IO`, é ilustrada na Figura 6-14. Por razões da arquitetura de J2ME, a classe `Canvas`, responsável por identificar teclas pressionadas no teclado do dispositivo é também a classe que provê acesso para desenho na tela. Por isso, a classe `IO`, ao estender a classe `Canvas` para escutar os eventos de pressionamento de teclas, tem também que ser a classe responsável por renderizar o cenário do jogo. Assim, o gerenciador gráfico, que será apresentado na próxima seção, também é implementado pela classe `IO`.

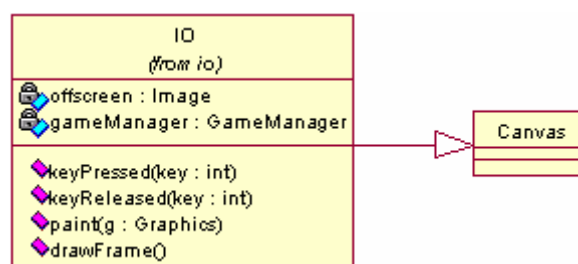


Figura 6-14: Classe `IO`

Dos métodos da classe `IO`, apenas `keyPressed(key)` e `keyReleased(key)`, são envolvidos com o gerenciamento de entrada. Eles detectam que uma tecla foi pressionada ou liberada e simplesmente chamam métodos similares que existem no gerenciador do jogo. Cabendo então ao gerenciador do jogo em cada jogo reescrever tais métodos para realizar as ações devidas.

### 6.3.5 Gerenciador Gráfico

O gerenciador gráfico é o responsável pela apresentação gráfica do jogo. Já apresentado na Figura 6-14, o seu processamento é restrito basicamente ao método `paint(Graphics)`, que é a reescrita do método `paint(Graphics)` padrão da classe `Canvas`. O corpo desse método na classe `IO` consiste em executar o método `paint` de todos os gerenciadores de objetos do jogo (que por sua vez vai executar o método `paint` de seus objetos visíveis), bem como do mapa do jogo. Como o método `paint` só é executado quando o método `repaint` da classe `Canvas` é chamado, a classe `IO` oferece o método `drawFrame`, que apenas realiza a chamada ao método `repaint`. Com esse suporte, a etapa de apresentação gráfica do ciclo do jogo implementada pelo

gerenciador do jogo precisa apenas utilizar o método `drawFrame` para que seja realizada.

O gerenciador gráfico automaticamente utiliza *double buffering* [4], para que todos os objetos desenhados durante a execução do método `paint` ocorram em uma área de memória (correspondendo ao atributo `offscreen`) e só após desenho completo do *frame* o resultado seja desenhado na tela do dispositivo. Isto traz uma série de vantagens, entre elas evita que os objetos sejam desenhados na tela de forma seqüencial, o que pode ser percebido pelo usuário e comprometer a qualidade do jogo. Note que com o uso de *double buffering* cada frame ainda é gerado seqüencialmente, porém em uma imagem secundária. Somente ao término do processo é que o resultado é mostrado na tela, e de uma única vez.

### 6.3.6 Gerenciador do Jogo

A gerenciamento do jogo envolve basicamente:

- Armazenamento do estado interno do jogo, que consiste em armazenar o conjunto de gerenciadores de objetos e o mapa do jogo;
- Implementação do ciclo do jogo, que, devido a modularidade do wGEM, é bem simples, pois basta repassar o trabalho para os gerenciadores de objetos e para o gerenciador gráfico;
- Inicialização do cenário, que também é bastante simples, pois quase todo o trabalho será realizado pelos gerenciadores de objetos e mapa do jogo.

A classe que implementa o gerenciador do jogo, `GameManager`, é ilustrada na Figura 6-15. Os atributos `gameObjectManagerSet` e `map` representam o estado interno do jogo. Entre os outros atributos destacam-se o gerenciador de entrada e gráfico, `io`, e a aplicação, `midletGame`. Os métodos `keyPressed(key)` e `keyReleased` são abstratos e requerem um tratamento especial para cada jogo. Eles são chamados pela classe `IO` em resposta aos respectivos eventos ocorridos com o teclado do dispositivo. Os métodos `pauseGame` e `continueGame` servem para parar e continuar o ciclo do jogo, cuja implementação é explicada na seção seguinte. Os métodos `startGame` e `destroyGame` são chamados no ato da inicialização e fim do jogo. Eles também são abstratos, cabendo a cada jogo implementar alguma coisa que seja útil ao jogo, como, por exemplo, salvar o estado do jogo no método `destroyGame` para uma futura continuação. O método `startLevel` inicia o jogo baseado no cenário inicializado em memória pelo método `loadLevel`, cuja implementação é explicada na seção 6.3.6.2.

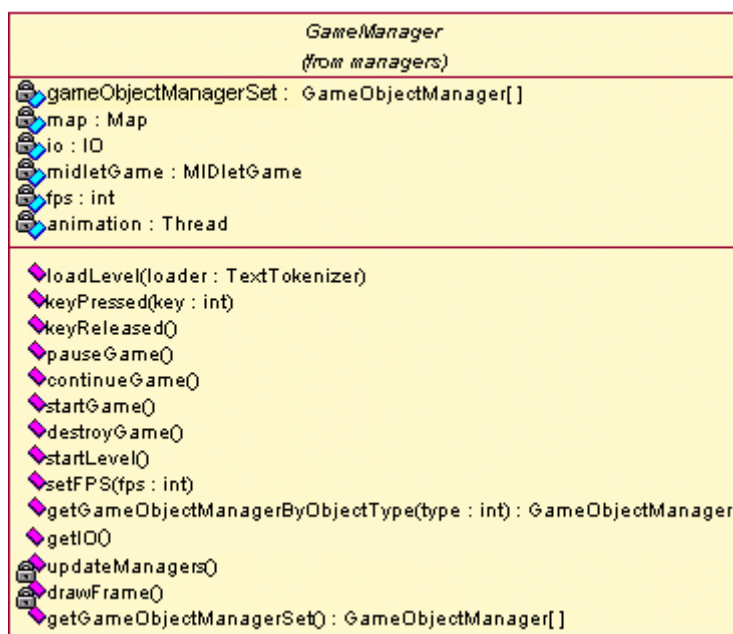


Figura 6-15: Classe GameManager

### 6.3.6.1 Ciclo do jogo

Para a implementação do ciclo do jogo é utilizada a classe `Thread` de J2ME, que realiza o *loop* que é executado a todo instante. A utilização de um *thread* para implementação do ciclo do jogo, representado pelo atributo `animation` da Figura 6-15, também facilita o processo de sincronização explicado na seção 2.1.2. Este *thread* tenta manter a taxa de *frames* por segundo que foi especificada pelo jogo através do método `setFPS` da classe `GameManager`. Para isso, o tempo gasto para a execução das duas tarefas do ciclo do jogo (atualização e apresentação gráfica) é medido para cada ciclo do jogo e comparado com a velocidade especificada. Baseado nessa comparação, o ciclo pode recomeçar imediatamente ou pode realizar uma pausa a fim de manter a taxa de apresentação desejada. Caso essa pausa seja necessária, basta suspender o *thread* de execução pelo tempo necessário para que a velocidade esperada seja alcançada.

Para facilitar o entendimento, um diagrama com a seqüência de operações realizadas pelo *thread* para a implementação do ciclo do jogo é ilustrado na Figura 6-16.

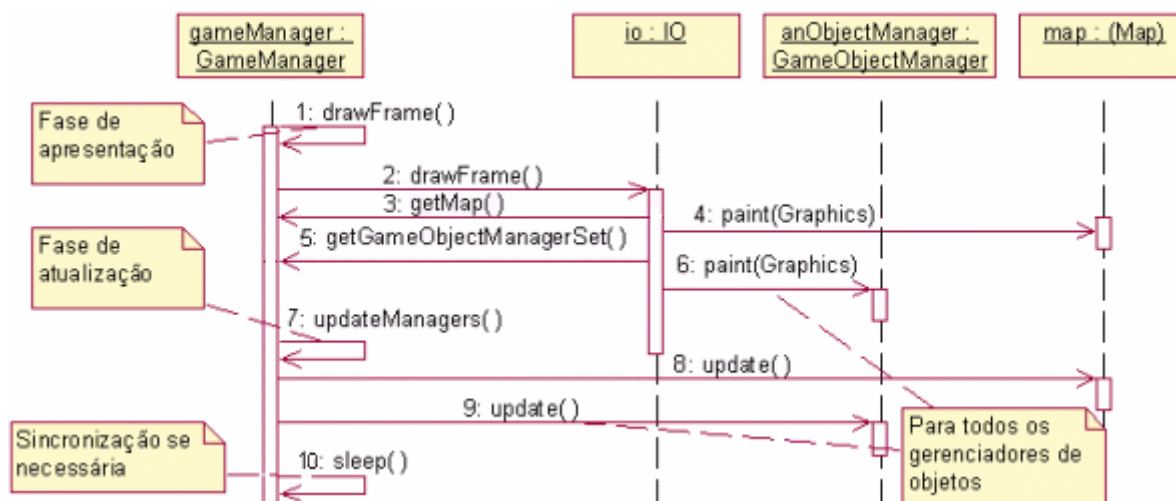


Figura 6-16: Ilustração dos eventos envolvidos no ciclo do jogo

É importante ressaltar que o uso de um *thread* para a implementação do ciclo do jogo, juntamente com os métodos `pauseGame` e `resumeGame`, promovem uma boa maneira para o desenvolvimento de jogos em turno. Para isso, bastaria associar os eventos de mudanças de turno dos jogos com a parada e continuação do ciclo do jogo. Com isso, por exemplo, a atualização gráfica só ocorreria quando as mudanças ocasionadas pelas jogadas no turno fossem realizadas, o que evitaria o redesenho desnecessário.

### 6.3.6.2 Inicialização do cenário

A inicialização do cenário é fornecida pelo método `loadLevel`, que realiza uma leitura do arquivo gerado pelo editor de cenários e repassa a tarefa para os gerenciadores de objetos e mapa do jogo. Durante este processo, o gerenciador do jogo precisa saber qual dos gerenciadores de objetos utilizados no jogo deve processar um grupo de um determinado tipo de objeto encontrado no arquivo do cenário (o formato do arquivo será explicado na seção 6.4.5). Para realizar a associação com o gerenciador apropriado, o gerenciador do jogo também define o método abstrato `getGameManagerByObjectType` que precisa ser implementado pela classe que estende o gerenciador do jogo em cada jogo, para que a devida associação entre tipos de objetos e gerenciadores desses tipos de objetos seja possível.

### 6.3.7 Gerenciador de Rede

O gerenciador de rede do wGEM, implementado pela classe `NetworkManager`, ilustrada na Figura 6-17, dispõe de métodos para *download* e *upload* de arquivos

utilizando o protocolo HTTP suportado pelo MIDP. O método `downloadFile(url)` recebe como argumento um endereço na Internet, faz o *download* do arquivo e retorna o seu conteúdo em um *array* de bytes. O método `requestService(serviceURL)` tem o papel de realizar requisições de serviços a *servlets*. Com este método, apenas a especificação da URL do *servlet* é necessária para a realização de *upload* e *download* de dados para ele. Os dados retornados pelo *servlet* também são retornados como um *array* de bytes.

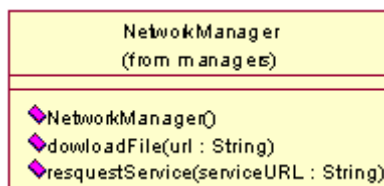


Figura 6-17: Classe NetworkManager

### 6.3.8 Aplicação

Assim como todo *applet* para J2SE precisa estender a classe `Applet` para que seja reconhecido como um programa a ser executado em um *browser*, no perfil MIDP cada aplicação tem que ter alguma classe que estenda a classe `MIDlet` para que seja executada pelo dispositivo.

A classe `MIDlet` possui três métodos abstratos que permitem à máquina virtual Java iniciar, pausar e destruir um determinado `MIDlet`. Dessa forma, o desenvolvedor, ao estender a classe `MIDlet`, deve implementar esses métodos para que sua aplicação seja executada da forma apropriada.

Para facilitar esse processo, o wGEM oferece a classe `MIDletGame`, ilustrada na Figura 6-18, que estende a classe `MIDlet` e implementa os métodos abstratos necessários através de chamadas aos métodos `startGame`, `pauseGame/resumeGame` e `DestroyGame` da classe `GameManager`. Em outras palavras, basta o desenvolvedor criar uma classe que estenda esta classe para que o jogo torne-se um `MIDlet`.

É interessante observar que é esta camada de aplicação, basicamente, que transforma o wGEM em um *framework* de jogos para J2ME. Da forma em que o wGEM foi desenvolvido, não haveria grandes dificuldades em construir outro módulo de aplicação, assim como fazer pequenos ajustes aos demais módulos, para transformá-lo em um *framework* de jogos 2D para J2SE, sendo esta transformação, inclusive, uma das nossas propostas de trabalhos futuros desta pesquisa.

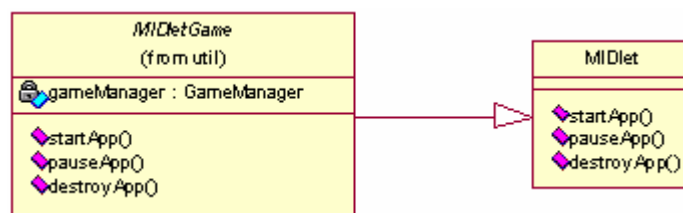


Figura 6-18: Classe MIDletGame

### 6.3.9 Integração com o Editor de Cenários

Este módulo é necessário pela falta de suporte de J2ME a certos recursos necessários à abertura e leitura do arquivo gerado pelo editor de cenários. Como explicado, o gerenciador do jogo, os gerenciadores de objetos e o mapa do jogo fazem uso da leitura do arquivo que define o cenário. Porém, em J2ME não existe uma classe que dê suporte ao *parsing* de arquivos texto, que é necessário para o processamento do arquivo produzido pelo editor. Por essa razão, o wGEM oferece a classe `TextTokenizer` que é capaz de produzir *tokens* encontrados em um texto. A classe `TextTokenizer` utiliza uma interface definida pelo wGEM, `TextLoader`, para obter os caracteres do texto a ser realizado o *parsing*. O wGEM oferece duas classes que implementam essa interface, `JARTextLoader` e `ByteArrayTextLoader`, para textos provenientes de arquivos e *array* de bytes, respectivamente. Essas duas classes atendem, portanto, tanto os casos do arquivo texto fazer parte do arquivo JAR do jogo, quanto os casos do arquivo ter sido obtido através de *download*. A Figura 6-19 ilustra todas essas classes apresentadas.

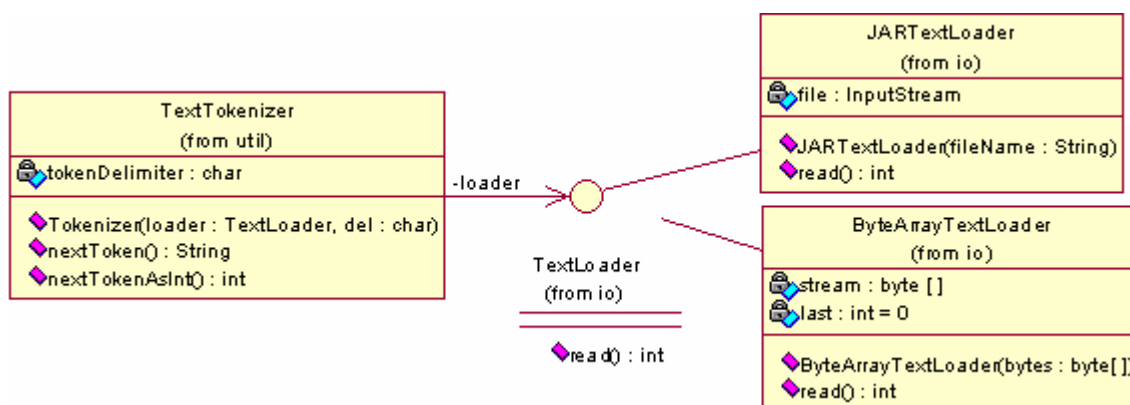


Figura 6-19: Módulo de integração do motor com o editor de cenários

## 6.4 IMPLEMENTAÇÃO DO EDITOR DE CENÁRIOS

Nesta seção será apresentado o editor de cenários do *framework* wGEM. Para a sua implementação foi escolhido o ambiente Delphi 5.0, da Imprise Corporation, que



oferece grande facilidade e rapidez no desenvolvimento de programas para PCs com os requisitos gráficos do editor de cenários.

Nas próximas seções serão apresentados os resultados obtidos na implementação de cada um dos requisitos apontados na Tabela 6-2. A seção 6.4.5 é de especial importância, pois explica como é a integração entre o editor e o motor.

### 6.4.1 Definição do Mapa

A definição do mapa envolve a especificação das seguintes informações:

- Estilo de mapa, que é selecionado entre as três opções disponíveis no wGEM (baseado em cores, textura ou *tiles*);
- Representação gráfica do mapa, cujas informações dependem do estilo de mapa escolhido (cor de fundo, imagem de textura, etc.);
- Dimensão do mapa, que é especifica através da sua largura e altura;
- Janela visível, que é especificada pela sua altura, largura, velocidade vertical e horizontal e posição inicial.

A Figura 6-20 ilustra a interface do editor para a especificação do mapa.

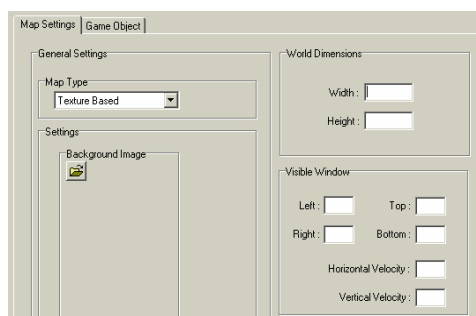


Figura 6-20: Interface do editor para definição do mapa do jogo

### 6.4.2 Definição dos Objetos

A definição dos objetos a serem utilizados no jogo requer as seguintes funções:

- Ilustração da paleta de objetos;
- Criação, edição e remoção de objetos.

A *ilustração da paleta de objetos* tem o papel de mostrar para o usuário do editor, que normalmente é o *game designer*, os objetos já definidos por ele. Essa ilustração, apresentada na Figura 6-21, mostra todos os objetos através da representação gráfica do objeto, do seu nome, e da quantidade de objetos de cada tipo já adicionados ao cenário.

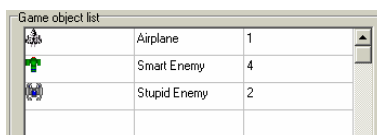


Figura 6-21: Paleta de objetos do editor de cenários

A *criação, edição e remoção de objetos* é realizada através da paleta de objetos. Nela é possível remover ou editar qualquer objeto, bem como adicionar novos, através de menus ativados com o clique do botão direito do *mouse* sobre a paleta. A edição ou inserção de objetos leva para uma janela de definição de objetos. Nesta janela, ilustrada na Figura 6-22, é possível definir o tipo do objeto (dentro de um conjunto de tipos presentes no dicionário de objetos do cenário<sup>1</sup>), tipo de representação gráfica dele (entre as disponíveis no wGEM), bem como as informações necessárias para a especificação de cada tipo de representação gráfica (imagens associadas, cor, preenchimento vazio ou cheio, etc).

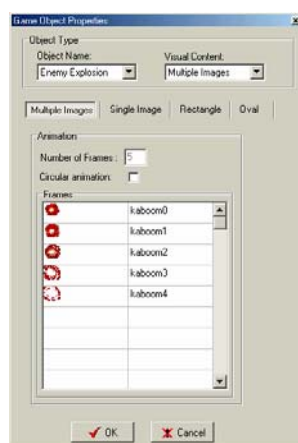


Figura 6-22: Definição de objetos através do editor de cenários

### 6.4.3 Definição do Cenário

A definição do cenário necessita das seguintes funcionalidades:

- Ilustração do cenário;
- Inserção, edição e remoção de objetos no cenário.

A *ilustração do cenário* mostra para o *game designer* a representação gráfica do cenário em definição. Além de mostrar os objetos sobre o mapa, a própria representação

---

<sup>1</sup> Esse dicionário é informado pelo game designer no início da definição do cenário para que o editor saiba os nomes e códigos dos objetos que podem ser definidos para cada jogo. O seu formato é descrito na seção 6.4.5

gráfica do mapa é também realizada, como é ilustrado na Figura 6-23, em que um mapa baseado em texturas é desenhado, permitindo que o usuário veja o resultado da textura sendo aplicada ao mapa. A janela visível também é desenhada, como pode ser observada na parte inferior da Figura 6-23.



Figura 6-23: Ilustração do mapa oferecida pelo editor de cenários

A *inserção, edição e remoção de objetos no cenário* é muito fácil. Para a inserção, basta o usuário arrastar com o *mouse* qualquer objeto da paleta de objetos para dentro do mapa que o mesmo será inserido na posição em que foi solto. A edição da posição do objeto no mapa também é realizada com o uso do *mouse*. A velocidade horizontal e vertical do objeto é editada através de um duplo clique sobre o objeto, que leva para a janela mostrada na Figura 6-24, onde o objeto também pode ser removido.

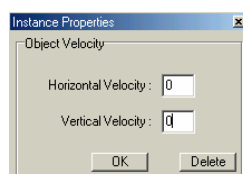


Figura 6-24: Interface do editor de cenários para a edição da velocidade dos objetos

#### 6.4.4 *Persistência de Cenários*

A persistência dos cenários desenvolvidos é muito importante e envolve a ação de salvar o cenário em um arquivo e carregá-lo a partir do mesmo. A abordagem para esse problema foi semelhante a adotada para a integração entre o editor e o motor, que consiste na transformação do cenário em um arquivo texto que descreve as propriedades do mesmo. De forma equivalente, a leitura desse arquivo é realizada para que o cenário previamente definido possa ser recriado.

### 6.4.5 Integração com o Motor

Durante a seção 3.4.3 foram apresentadas duas das principais abordagens utilizadas para a integração entre o editor de cenários e o motor de jogos.

A abordagem de o editor serializar os seus objetos que definem o cenário para que o motor possa lê-los e inicializar os seus devidos objetos, apesar de ser tecnicamente simples, não é viável para J2ME por não haver suporte à leitura de objetos serializados. Sendo assim, a abordagem utilizada foi a transformação do cenário em um arquivo descritor, cujas informações são apresentadas (em ordem de ocorrência no arquivo) na Tabela 6-3.

<b>Tipo do Mapa</b>	0 se TextureBasedMap, 1 se ColorBasedMap ou 2 se TileBasedMap.
<b>Representação Gráfica do Mapa</b>	Nome da imagem de textura, se TextureBasedMap, RGB da cor do mapa, se ColorBasedMap ou, se TileBasedMap, número de imagens usadas ( <i>ni</i> ); nomes das imagens ( <i>ni</i> nomes); informação sobre transponibilidade das imagens ( <i>ni</i> números 0 ou 1); número de linhas ( <i>l</i> ) e colunas ( <i>c</i> ) de tiles; índice da imagem utilizada em cada tile ( <i>lxc</i> números entre 1 e <i>ni</i> ).
<b>Dimensão do Mapa</b>	Largura e altura.
<b>Janela Visível</b>	Velocidade horizontal, vertical; posição da janela (esquerda, direita, topo e base).
<b>Objetos Definidos</b>	Número de tipos de objetos definidos; Para cada tipo objeto: código do objeto, representação gráfica do objeto (formato explicado a seguir), instâncias do objeto colocadas no cenário (formato explicado a seguir).
<b>Representação Gráfica de um Objeto</b>	Tipo de representação gráfica (0 para múltiplas imagens, 1 para uma imagem, 2 para retângulo, 3 para elipse e 4 para polígono); conteúdo da representação gráfica (número de imagens e nome da cada imagem para representação gráfica do tipo 0; nome da imagem para tipo 1; largura, altura, RGB da cor e tipo de preenchimento, vazio ou cheio, para tipo 2 ou 3; número de pontos do polígono, definição dos pontos e tipo de preenchimento para tipo 4).
<b>Instâncias dos Objetos</b>	Número de instâncias; Para cada instância: posição horizontal, vertical e velocidade horizontal e vertical do objeto.

Tabela 6-3: Formato do arquivo de descrição de cenários

Como ilustrado na Tabela 6-3, os objetos do jogo que foram definidos no editor são identificados na descrição do arquivo por códigos, enquanto que quando o *game designer* escolheu o tipo do objeto no editor, essa escolha foi realizada pelo nome do tipo do objeto, como apresentado na seção 6.4.2. Essa associação entre nomes e códigos é definida por um dicionário de objetos que é informado pelo *game designer* ao editor antes da definição do cenário. Esse arquivo, cujo formato é um sequência de pares (nome do objeto, código do objeto) representa todos os tipos de objetos que podem ser introduzidos no cenário para um determinado jogo (no próximo capítulo existem exemplos deste arquivo). É baseado nesse código do objeto que o jogo é capaz de

associar cada tipo de objeto ao devido gerenciador de objetos, como foi apresentado na seção 6.3.6.2.

Para que essa associação por códigos possa ser realizada, o dicionário de objetos precisa ser definido pelo *game designer*, pois ele tem o domínio sobre todos os objetos pertencentes ao jogo, bem como pode associar códigos a cada um deles. Uma vez que essa associação seja realizada, o jogo pode ser implementado, bem como os cenários, pois cada objeto definido no editor será devidamente compreendido pelo jogo.

Outro ponto importante é que a apenas os nomes das imagens utilizadas no cenários são adicionados ao arquivo descritor. Com isso, quando o motor realiza a inicialização do cenário ele precisa de tais imagens para realmente iniciar o cenário. Em geral, essas imagens fazem parte do arquivo JAR do jogo, porém, o motor também pode conseguir tais imagens remotamente pelo módulo de rede do wGEM. Nesse caso, ao realizar o *download* de cada imagem, o jogo poderia também armazená-las no disco rígido dos dispositivos para uso futuro.

## 6.5 CONCLUSÕES

O presente capítulo teve o objetivo de dar uma visão detalhada da nossa proposta de *framework* para jogos em J2ME. Alguns detalhes de implementação revelaram a preocupação que tivemos em otimizar o sistema tanto em uso de memória e CPU. Essas otimizações, embora tenham sido apresentadas como as técnicas atualmente em uso no wGEM, foram um resultado de um longo processo de amadurecimento. Artifícios como o algoritmo de detecção de colisão baseado em projeções, a utilização de *bouding boxes*, o desenho *off-line* de polígonos, e vários outros, só vieram a ser aplicados depois que os problemas com as outras abordagens foram identificados, como os erros na detecção de colisão por pertinência de vértices, a construção e desenho dos polígonos *on-line*, etc.

Esperamos ter mostrado que o wGEM é de fato um sistema que apresenta as características fundamentais para o desenvolvimento de jogos em J2ME. O capítulo seguinte é um complemento deste, onde são apresentados alguns jogos que demonstram o uso do wGEM e servem também como uma forma de validação deste *framework*.



## 7 ESTUDOS DE CASO

Durante este capítulo serão apresentados jogos que desenvolvemos para ilustrar o uso do wGEM bem como para avaliar o seu desempenho em emuladores e dispositivos reais. Inicialmente serão apresentados dois jogos para dar uma visão detalhada de como o wGEM pode ser utilizado na prática para o desenvolvimento de um jogo. Em seguida será relatada uma experiência recente de uso do sistema que exemplifica claramente como o wGEM facilita o processo de desenvolvimento de jogos para J2ME. Finalmente, uma avaliação do desempenho dos jogos desenvolvidos será apresentada.

### 7.1 *BREAKOUT*

O primeiro jogo implementado foi o clássico BreakOut, ilustrado na Figura 7-1, onde o jogador tem o objetivo de destruir um conjunto de blocos utilizando uma bola que é controlada por uma raquete. O usuário deve então controlar a raquete para evitar que a bola saia do campo de jogo (parte inferior) e ao mesmo tempo tentar dar uma direção à bola que provoque o choque com algum bloco.



Figura 7-1: Jogo BreakOut

#### 7.1.1 *Arquitetura do jogo*

Durante essa seção serão mostradas como as classes do wGEM, bem como o editor de cenários, foram utilizados para a implementação do BreakOut.

#### 7.1.1.1 Gerenciadores de objetos

Seguindo a abordagem proposta pelo wGEM, cada jogo deve ter diversos gerenciadores de objetos, cada um responsável por objetos similares. Dessa forma, os seguintes gerenciadores de objetos foram modelados para o BreakOut:

- Gerenciador da bola, que armazena todas as bolas presentes no jogo (pode haver mais de uma em jogo ao mesmo tempo);
- Gerenciador da raquete, que armazena a raquete controlada pelo jogador;
- Gerenciador dos blocos, que armazena todos os blocos do jogo.

#### 7.1.1.2 Objetos do jogo

Como no BreakOut existem apenas três tipos de objetos, três classes estendem a classe `GameObject` do wGEM para representar a *bola*, o *bloco* e a *raquete*.

- A *bola* redefine o método de atualização (`update`) da classe `GameObject` para realizar as ações necessárias, que são o teste de fim de jogo, colisão com a raquete, com os blocos e com a parede, assim como tomar as ações coerentes, como refletir a trajetória da bola, solicitar a remoção de um bloco colidido, etc;
- A *raquete* apenas define dois métodos necessários à sua movimentação pelo jogador que são utilizados pelo gerenciador do jogo especializado para o jogo BreakOut, que ao detectar que a tecla esquerda ou direita do teclado foi acionada (pelo método `keyPressed`) repassa o ocorrido para a raquete;
- O *bloco* não tem um comportamento especial com relação a classe `GameObject`, sendo definido como uma nova classe apenas para facilitar a implementação de uma possível evolução no comportamento deste objeto para algum mais rico.

#### 7.1.1.3 Mapa do jogo

Como o BreakOut não tem necessidade de *scrolling* do mapa, já que ele tem a mesma dimensão da tela do dispositivo, o mapa baseado em cores do wGEM foi utilizado.

#### 7.1.1.4 Gerenciador de Entrada e Gráfico

O gerenciador gráfico e de entrada do BreakOut estende a classe `IO` do wGEM para adicionar dois comandos na tela do dispositivo. Um deles permite que uma bola adicional seja incluída no jogo, e o outro reinicia o jogo.



### 7.1.1.5 Gerenciador do Jogo

O gerenciador do jogo do BreakOut agrupa todos os gerenciadores de objetos do jogo (como atributos de classe) e implementa os métodos abstratos definidos pela classe `GameManager` do `wGEM`.

Nos métodos `keyPressed` e `keyReleased` ele identifica se a tecla representa a seta para esquerda ou direita e repassa o evento para que a raquete movimente-se de acordo. Outro método que merece destaque é o `startGame`. Nele, o arquivo gerado pelo editor de cenários é aberto e o método `loadLevel` da classe base é executado, levando à carga do cenário em memória. Em seguida o método `startLevel` é chamado, o que leva à inicialização do ciclo do jogo.

### 7.1.1.6 Definição do cenário

Como explicado na seção 6.4.5, para definir um cenário utilizando o editor é preciso definir o dicionário de objetos do jogo. Para o BreakOut este dicionário é ilustrado na Tabela 7-1.

Bloco 1
Raquete 2
Bola 3

Tabela 7-1: Dicionário de objetos do jogo BreakOut

Com essa definição, o editor pode ser inicializado levando o usuário a definir qualquer um desses três tipos de objetos. Da mesma forma, a implementação do método `getGameManagerByObjectType` no gerenciador do jogo do BreakOut é capaz de associar objetos com esses códigos aos devidos gerenciadores de objetos.

A definição gráfica do bloco e raquete utiliza a representação gráfica baseada em retângulos, enquanto que a bola utiliza a representação de elipse. A Figura 7-2 ilustra a paleta de objetos construída com o editor, enquanto que a Figura 7-3 ilustra um cenário definido com esses objetos. Este mesmo cenário foi exportado para arquivo e utilizado no jogo, produzindo o resultado que foi apresentado na Figura 7-1.

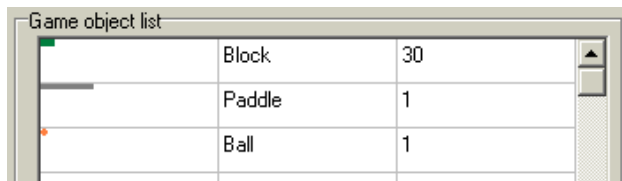
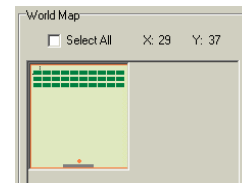


Figura 7-2: Paleta de objetos do jogo BreakOut

Figura 7-3: Cenário do jogo  
BreakOut

## 7.2 SHIP

O segundo jogo implementado foi denominado Ship, um jogo de nave no estilo RiverRaid, onde o jogador tem o objetivo de controlar uma nave por um campo contendo algumas naves inimigas, destruir o máximo de inimigos possível utilizando as armas da nave e coletar alguns itens de energia e armas especiais pelo caminho. O resultado é ilustrado na Figura 7-4.



Figura 7-4: Jogo Ship

### 7.2.1 Arquitetura do jogo

Durante essa seção será explicada a arquitetura do jogo Ship alcançada com a utilização do wGEM e do editor de cenários.

#### 7.2.1.1 Gerenciadores de objetos

Os seguintes gerenciadores de objetos foram modelados para o Ship:

- Gerenciador da nave, que armazena a nave do jogador;
- Gerenciador dos inimigos, que armazena as naves inimigas;
- Gerenciador dos tiros da nave do jogador, que armazena os objetos que representam os tiros que a nave do jogador disparou;
- Gerenciador das balas dos inimigos, idem ao caso anterior;

- Gerenciador das explosões das naves inimigas, que armazena os objetos que representam as explosões destas naves;
- Gerenciador das explosões da nave do jogador, idem ao caso anterior;
- Gerenciador dos itens do jogo, que armazenam os itens que podem ser coletados pela nave do jogador.

#### 7.2.1.2 Objetos do jogo

Para o jogo Ship existem 8 tipos de objetos. A seguir eles são apresentados:

- Nave do jogador, que redefine o método de atualização (`update`) da classe `GameObject` para realizar as ações necessárias, que são a identificação de colisão entre ela e os inimigos, a borda da tela e os itens a serem coletados, assim como responder aos comandos do jogador (movimentação e tiro);
- Inimigo inteligente, que ao se tornar visível no mapa começa a seguir a nave do jogador e atirar contra ele;
- Inimigo simples, que apenas movimenta-se na tela sem seguir ou atirar;
- Item, que se divide em quatro tipos diferentes itens que podem ser coletados pelo jogador (energia e três tipos de arma);
- Tiro do jogador, que permite a visualização do tiro e detecta colisão entre ele e os inimigos, tomando as providências devidas;
- Tiro do inimigo, idem ao caso anterior;
- Explosão da nave inimiga, que permite a visualização do efeito de explosão da nave inimiga;
- Explosão da nave do jogador, idem ao caso anterior.

#### 7.2.1.3 Mapa do jogo

Como no jogo Ship existe *scrolling* de tela, o mapa baseado em texturas do wGEM é utilizado como base para a definição de um mapa cujo comportamento especial é a realização de *scrolling* vertical para cima a cada ciclo do jogo.

#### 7.2.1.4 Gerenciador de Entrada e Gráfico

O gerenciador gráfico e de entrada do Ship estende a classe `IO` do wGEM adicionando dois comandos na tela do dispositivo para terminar e reiniciar o jogo.

### 7.2.1.5 Gerenciador do Jogo

O gerenciador do jogo do Ship é semelhante ao existente no jogo BreakOut, com a diferença que nos métodos `keyPressed` e `keyReleased` ele identifica se a tecla representa a seta para esquerda, direita, cima e baixo e repassa o evento para que a nave do jogador movimente-se de acordo, assim como ao identificar que a tecla pressionada representa o tiro, ele repassa o ocorrido para o gerenciador de tiros da nave para que o tiro aconteça.

### 7.2.1.6 Definição do cenário

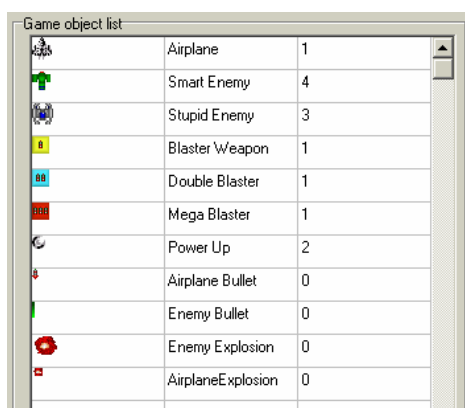
O dicionário de objetos utilizado no Ship é ilustrado na Tabela 7-2.

Nave do jogador	0
Inimigo inteligente	1
Inimigo simples	2
Item de arma de um tiro	3
Item de arma de dois tiros	4
Item de arma de três tiros	5
Item de energia	6
Tiro da nave do jogador	7
Tiro da nave inimiga	8
Explosão da nave do jogador	9
Explosão da nave inimiga	10

Tabela 7-2: Dicionário de objetos do jogo Ship

Da mesma forma que ocorreu no BreakOut, a implementação do método `getGameobjectManagerByObjectType` no gerenciador do jogo do Ship realiza a associação devida entre objetos com esses códigos e os devidos gerenciadores.

A definição gráfica desses objetos utiliza a representação gráfica baseada em imagens simples, com exceção das explosões, que utilizam animação de imagens. A Figura 7-5 ilustra a paleta de objetos construída com o editor, enquanto que a Figura 7-6 ilustra uma parte do cenário definido com esses objetos. Foi este o cenário utilizado para produzir a imagem apresentada na Figura 7-4.



Game object list		
	Airplane	1
	Smart Enemy	4
	Stupid Enemy	3
	Blaster Weapon	1
	Double Blaster	1
	Mega Blaster	1
	Power Up	2
	Airplane Bullet	0
	Enemy Bullet	0
	Enemy Explosion	0
	Airplane Explosion	0

Figura 7-5: Paleta de objetos do jogo Ship

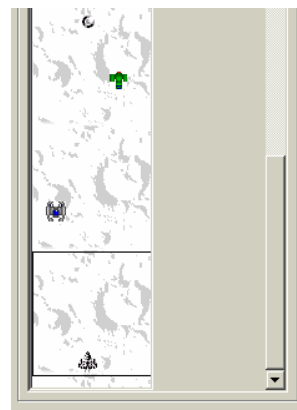


Figura 7-6: Cenário do jogo Ship

### 7.3 ENDURO

O jogo Enduro, do clássico jogo de corrida de carros da era Atari, ilustrado na Figura 7-7, teve o seu desenvolvimento um pouco diferente dos dois jogos relatados anteriormente. Os jogos BreakOut e Ship foram implementados durante a fase de desenvolvimento e testes do wGEM, enquanto que o jogo Enduro foi criado pela equipe do CESAR de desenvolvimento de aplicações para celulares, sem a participação direta do autor deste trabalho.



Figura 7-7: Jogo Enduro

A equipe do CESAR, que em 4 de outubro de 2001 iniciou o desenvolvimento deste jogo de corrida de carros, concluiu a primeira versão em apenas um dia de trabalho envolvendo duas pessoas. A razão para tamanha rapidez vem do fato que o recurso de *scrolling* de mapa do wGEM é a base para jogos de corrida de carros (2D e simples). Uma vez que esse recurso já é fornecido pelo motor, que o editor de cenários permite facilmente a criação do cenário (os obstáculos na pista, a textura que representa a pista, etc.), e que o jogo não conta com NPCs muito sofisticados, quase tudo é fornecido automaticamente pelo wGEM, bastando apenas a definição da arte gráfica e a implementação das (poucas) características particulares do jogo.

Essa facilidade proporcionada ao jogo Enduro pode ser estendida também para praticamente todos os jogos de plataforma, já que as características de todos eles são basicamente as mesmas.

Por ter uma estrutura bastante similar aos jogos BreakOut e Ship, não ilustraremos aqui a arquitetura detalhada do jogo Enduro. Em vez disto, apresentamos a seguir uma avaliação do desempenho desses três jogos com o uso do wGEM.

## 7.4 AVALIAÇÃO DO RESULTADO

Para a avaliação dos jogos desenvolvidos com o wGEM, consideramos os quesitos tamanho e desempenho dos jogos. Nas seguintes seções estão os resultados obtidos.

### 7.4.1 Desempenho

O parâmetro que utilizamos para avaliar o desempenho dos jogos foi a taxa de *frames* por segundo alcançada. O valor dessa taxa considerado ideal para os jogos é de 24 fps [4], pois quando uma sequência de imagens (*frames* do jogo) é apresentada a esta frequência, o olho humano percebe essa animação discreta (sequência de *frames*) como se fosse contínua<sup>1</sup>.

Para os experimentos, definimos no motor a taxa de apresentação em 24 fps e monitoramos, para cada jogo, a real capacidade de produção de *frames*. Uma vez que devido a fase de sincronização do ciclo do jogo, caso a produção de *frames* ultrapasse o valor especificado (24, no caso), a suspensão do *thread* de execução mantém sempre a mesma taxa.

O desempenho alcançado no ambiente da Sun que utilizamos para o desenvolvimento dos jogos foi excelente. Porém, o resultado obtido não reflete o desempenho que o jogo teria em um real dispositivo, pois o emulador da Sun utiliza todo o poder de processamento do PC para executar os programas. Para se ter uma idéia, nos três jogos desenvolvidos, a taxa de *frames* alcançada foi de 1000 fps. Entretanto, esse valor é importante para dar uma idéia de quão rápido seriam estes jogos caso o wGEM fosse um motor para J2SE.

Para uma melhor avaliação do desempenho, adotamos a avaliação dos jogos em emuladores de Palms, uma vez que a implementação da KVM para o sistema

---

<sup>1</sup> Desde que as diferenças entre as imagens adjacentes da sequência sejam pequenas.

operacional PalmOS, além do próprio emulador de Palms, são disponíveis para *download* na Internet. Nesse ambiente, os resultados obtidos foram bem mais reais, resultando em uma taxa de apresentação média de 30 fps para os três jogos desenvolvidos.

Os testes que realizamos no emulador do Palm foram repetidos com sucesso e performance similar em dois modelos de Palms e em um celular (em fase de desenvolvimento) com suporte a J2ME conseguido pelo CESAR.

Um resultado importante, observado na execução do jogo BreakOut, foi a baixa performance observada quando a bola entra na região definida pelo *bouding box* do gerenciador de blocos. Como nesse momento o teste de não colisão com esta região falha, o teste seqüencial com todos os blocos é realizado. Com isso, quando o cenário que desenvolvemos tinha um grande número de blocos, cerca de 30, o jogo chegou a apresentar uma performance de 5 fps dentro do *bouding box* e 30 fora dele. Esse resultado mostra que o número de objetos visíveis (ao mesmo tempo) de um gerenciador de objetos deve ser cuidadosamente observado em razão do baixo poder de processamento dos dispositivos.

A Tabela 7-3 resume os resultados obtidos.

	Performance (FPS)			
	Emulador Sun	Emulador de Palm	Palm	Celular
<b>BreakOut</b>	1000	20-50	6-33	5-24
<b>Ship</b>	1000	50-70	20-33	16-22
<b>Enduro</b>	1000	60-100	30-50	19-24

Tabela 7-3: Performance dos jogos desenvolvidos com o wGEM

### 7.4.2 Tamanho dos Jogos

O tamanho dos programas em J2ME é um fator importante a ser considerado devido a limitada capacidade dos dispositivos. Na Tabela 7-4 resumimos o tamanho final de cada jogo, a divisão desse tamanho entre o motor<sup>1</sup>, o jogo e os arquivos de recursos para os jogos, bem como o número de linhas de código dos jogos.

<sup>1</sup> O tamanho do motor em cada jogo varia por que somente os módulos necessários são adicionados.

	Tamanho (KB)				Linhas de Código do Jogo
	Motor	Jogo	Recursos do Jogo	Total	
<b>BreakOut</b>	21.3	8.6	0.1	30.0	306
<b>Ship</b>	22.7	23.0	8.3	54.0	794
<b>Enduro</b>	22.1	5.5	2.5	30.1	207

Tabela 7-4: Tamanho dos jogos desenvolvidos com o wGEM

## 7.5 CONCLUSÕES

Consideramos os resultados obtidos com os jogos desenvolvidos bastante satisfatórios como validação do wGEM. Com eles foi possível aplicar o modelo de arquitetura oferecido pelo wGEM de forma bastante simples e prática, além de utilizar com sucesso a integração entre o editor de cenários e o motor. No quesito performance, consideramos satisfatória a execução dos jogos nos dispositivos reais. Em se tratando da facilidade de desenvolvimento proporcionada, a experiência com o jogo Enduro é um exemplo concreto para esse aspecto, mostrando inclusive a tamanha importância de um *framework* para a aceleração do processo de desenvolvimento de jogos.

Outro ponto de destaque é a facilidade de desenvolvimento de novos cenários realmente alcançada com a utilização do editor de cenários. Como mostrado neste capítulo, uma vez que os objetos em uso em um determinado jogo tenham sido definidos pelo dicionário de objetos, novos níveis facilmente são definidos através do editor de cenários e integrados ao jogo, da mesma forma que a representação gráfica dos objetos e do mapa pode ser alterada facilmente e sem a necessidade de programação (ou recompilação). Isto não só encurta o processo de integração dos cenários com o jogo, como também promove grande independência entre o desenvolvimento do jogo (realizado pelos desenvolvedores de software) e o desenvolvimento dos cenários do jogo (realizado pelos *game designers*).



## 8 CONCLUSÕES

Neste capítulo serão apresentadas as principais contribuições desta pesquisa, as dificuldades encontradas em sua realização e alguns trabalhos futuros.

### 8.1 CONTRIBUIÇÕES

Acreditamos que a contribuição mais relevante deste trabalho é o estudo e desenvolvimento originais de um *framework* para jogos 2D baseado em J2ME, onde buscamos ao mesmo tempo qualidade e eficiência.

O desenvolvimento de um editor de cenários representa também um resultado que agrega valor ao wGEM. Basta imaginar o trabalho manual que seria necessário caso todos os cenários tivessem que ser especificados diretamente junto com o jogo através de comandos de programação.

Outro resultado importante é o módulo de conexão remota oferecida pelo wGEM (apresentado na seção 6.3.7). Isto, aliado com a capacidade do motor de ler os cenários dos jogos a partir de arquivos gerados pelo editor de cenários, permite que os jogos já instalados nos dispositivos possam evoluir consideravelmente com apenas o *download* de novos cenários produzidos pelo editor.

### 8.2 DIFICULDADES

A primeira dificuldade encontrada nesta pesquisa foi certamente a ausência de literatura atual e de qualidade sobre o funcionamento detalhado de jogos 2D, especialmente sobre *frameworks* para essa categoria de jogos. Só há poucos anos os pesquisadores começaram a empenhar-se academicamente na área de jogos. Por isso, a maioria das contribuições dadas estão relacionadas com jogos 3D, que representam o estado da arte na indústria de jogos para PCs. Além disso, os recursos assumidos pelas soluções para PCs ficam longe da realidade existente nos dispositivos móveis e em J2ME.

Um outro desafio foi a complexidade encontrada no desenvolvimento de um *framework* para jogos 2D. Primeiro, pela necessidade de prover uma solução geral, o que requer mais tempo para projeto e testes. Segundo, pelas fortes exigências de

desempenho para tais sistemas. Finalmente, pelas limitações técnicas da tecnologia J2ME, o que nos levou a uma grande fase de adaptação de soluções aplicáveis em jogos 2D para PCs, além de exigir a implementação de diversos módulos que normalmente já são oferecidos pela linguagem de programação ou bibliotecas para PCs.

### 8.3 TRABALHOS FUTUROS

Como continuação do nosso trabalho, diversas maneiras de evoluir o wGEM podem ser consideradas. A seguir são apresentados trabalhos de *extensão* e *otimização* deste *framework*.

No que diz respeito à *extensão* do wGEM, alguns módulos podem ser adicionados ao sistema, assim como alguns deles podem ser complementados. A seguir são listadas algumas sugestões:

- Permitir a especificação de objetos com representação gráfica baseada em polígonos e mapas baseados em *tiles* na interface do editor de cenários;
- Adicionar um módulo ao motor que ofereça objetos que tenham comportamentos inteligentes bem definidos e que possam ser reutilizados, como perseguição e fuga de um determinado objeto especificado;
- Definir classes de exceções que melhorem a informação dada pelo motor sobre problemas ocorridos em seu funcionamento;
- Definir um novo módulo de aplicação e realizar os ajustes necessários para tornar o motor compatível com J2SE, como apresentado na seção 6.3.8.

Em se tratando da *otimização* do wGEM, acreditamos que o trabalho deva ser focado no gerenciador de objetos. Como apresentado na seção 6.3.3, este gerenciador utiliza um repositório de objetos para armazenar todos os seus objetos. No entanto, este modelo produz um sério *overhead* quando é preciso descobrir (a cada ciclo do jogo) os objetos do repositório que estão visíveis, ou que colidem com um dado objeto, uma vez que é necessário o acesso a todos os objetos do repositório. Para acelerar esse processo, apresentamos na seção 6.3.3 as soluções de ordenação dos objetos da estrutura antes de cada ciclo do jogo e utilização de *bounding box*. No entanto, acreditamos que soluções alternativas precisam ser estudadas para buscar possíveis ganhos de performance.

## 9 REFERÊNCIAS BIBLIOGRÁFICAS

- [1] Giguère, E., *Java 2 Micro Edition*, Wiley Computer Publishing, 2000.
- [2] Sun Microsystems, *A Brief History of the Green Project*, URL: <http://java.sun.com/people/jag/green/index.html>. Consultado em 29 de julho de 2001.
- [3] Battaiola, A., *Jogos por Computador – Histórico, Relevância Tecnológica e Mercadológica, Tendências e Técnicas de Implementação*, Curitiba, SBC 2000.
- [4] Fan, J. et al. *Black Art of Java Game Programming*. Waite Group Press, 1996
- [5] Rollings, A., Morris, D., *Game Architecture and Design*, Coriolis Group Books, 1999
- [6] Sommerville, I., *Software Engineering*, Addison-Wesley, 2000.
- [7] Ramalho, G., *Projeto e Implementação de Jogos*, URL: <http://www.cin.ufpe.br/~compint/jogos.html>. Consultado em 2 de agosto de 2001.
- [8] Silva, D., Ramalho, G., *Atores Sintéticos em Jogos de Aventura: o Projeto enigma no Campus*, Dissertação de Mestrado, Pernambuco, Dezembro, 2000.
- [9] Madeira, C., Ramalho, G., Ferraz, C., *FORGE V8: Um framework para o desenvolvimento de jogos de computador e aplicações multimídia*, Dissertação de Mestrado, Pernambuco, Julho, 2001.
- [10] Feng, Y., Zhu, J., *Wireless Java Programming with J2ME*, Sams Publishing, 2001.
- [11] Java 2 Platform, Standard Edition (J2SE), URL: <http://java.sun.com/j2se/>. Consultado em 6 de agosto de 2001.
- [12] Java 2 Platform, Enterprise Edition (J2EE), URL: <http://java.sun.com/j2ee/>. Consultado em 6 de agosto de 2001.
- [13] Java2 Platform, Micro Edition (J2ME), URL: <http://java.sun.com/j2me/>. Consultado em 6 de agosto de 2001.
- [14] Mobile Information Device Profile (MIDP), URL: <http://java.sun.com/products/midp/>. Consultado em 6 de agosto de 2001.
- [15] CLDC and the K Virtual Machine (KVM), URL: <http://java.sun.com/products/cldc/>. Consultado em 6 de agosto de 2001.
- [16] CDC and the CVM Virtual Machine, URL: <http://java.sun.com/products/cdc/>. Consultado em 6 de agosto de 2001.

- [17] Filion, D., *Structure of Games*, URL: <http://members.nbci.com/armagammon/articles/structure.htm>. Consultado em 6 de agosto de 2001.
- [18] Java 2 Platform Micro Edition, Wireless Toolkit, URL: <http://java.sun.com/products/j2mewtoolkit/>. Consultado em 6 de agosto de 2001.
- [19] Pessoa, C., Ramalho, G., *wGEM: Um Motor para Desenvolvimento de Jogos para Dispositivos Móveis*. Centro de Informática – UFPE, 2000. URL: <http://www.cin.ufpe.br/~tg/2000-2/>. Consultado em 7 de agosto de 2001.
- [20] Java Remote Method Invocation (RMI), URL: <http://java.sun.com/products/jdk/rmi>. Consultado em 7 de agosto de 2001.
- [21] JavaOne 1999, URL: <http://java.sun.com/javaone/javaone99/>. Consultado em 7 de agosto de 2001.
- [22] Joystudios, URL: <http://www.joystudios.com.br> . Consultado em 9 de agosto de 2001.
- [23] Jynx Playware, URL: <http://www.jynx.com.br>. Consultado em 9 de agosto de 2001.
- [24] Interactive Digital Software Association (IDSA), URL: <http://www.idsa.com>. Consultado em 9 de agosto de 2001.
- [25] DirectX Home, URL: <http://www.microsoft.com/directx/>. Consultado em 9 de agosto de 2001.
- [26] The Xbox Console, URL: <http://www.microsoft.com/xbox/>. Consultado em 9 de agosto de 2001.
- [27] Mobile Information Device Profile (MIDP) for Palm OS, URL: <http://java.sun.com/products/midp/palmOS.html>. Consultado em 9 de agosto de 2001.
- [28] Feijó, B., Dreux, M., Ramalho, G., Battaiola, A., Pessoa, C., *Desenvolvimento de Jogos em Computadores e Celulares*, Florianópolis, SIBGRAPI 2001.
- [29] Revista de Informática Teórica e Aplicada - RITA, URL: <http://www.inf.ufrgs.br/~revista/>. Consultado em 16 de agosto de 2001.
- [30] OpenGL, URL: <http://www.opengl.org>. Consultado em 8 de agosto de 2001.
- [31] Jasc Paint Shop Pro 7, URL: <http://www.jasc.com/products/psp/>. Consultado em 24 de agosto de 2001.
- [32] Corel Corporation, URL: <http://www.corel.com>. Consultado em 24 de agosto de 2001.
- [33] Adobe Photoshop 6, URL: <http://www.adobe.com/products/photoshop/main.html>. Consultado em 24 de agosto de 2001.
- [34] Sound Forge, URL: <http://www.sonicfoundry.com>. Consultado em 24 de agosto de 2001.

- [35] Cakewalk, URL: <http://www.cakewalk.com>. Consultado em 24 de agosto de 2001.
- [36] 3D Studio Max, URL: <http://www.discreet.com/products/>. Consultado em 24 de agosto de 2001.
- [37] Maya 3, URL: <http://www.aliaswavefront.com/en/Home/homepage.html>. Consultado em 24 de agosto de 2001.
- [38] Centro de Estudos e Sistemas Avançados do Recife, URL: <http://www.cesar.org.br>. Consultado em 24 de agosto de 2001.
- [39] PalmOS, URL: <http://www.palmos.com/>. Consultado em 3 de setembro de 2001.
- [40] Windows CE, URL: <http://www.microsoft.com/windows/embedded/ce/default.asp>. Consultado em 1 de outubro de 2001.
- [41] Anywhereyougo.com, URL: <http://www.anywhereyougo.com/>. Consultado em 3 de setembro de 2001.
- [42] WAP, URL: <http://www.gsmworld.com/technology/wap.html>. Consultado em 1 de outubro de 2001.
- [43] SMS, URL: <http://www.gsmworld.com/technology/sms.html>. Consultado em 1 de outubro de 2001.
- [44] I-Mode, URL: <http://www.nttdocomo.com/i/service/home.html>. Consultado em 1 de outubro de 2001.
- [45] NTT DoCoMo, URL: <http://www.nttdocomo.com>. Consultado em 3 de setembro de 2001.
- [46] Phone2Play, URL: <http://www.phone2play.com>. Consultado em 3 de setembro de 2001.
- [47] Fieldem, T., Orubeondo, A., *J2ME and WAP: Together forever?*, URL: <http://www.javaworld.com/javaworld/jw-12-2000/jw-1201-iw-j2me.html>. Consultado em 1 de setembro de 2001.
- [48] Briggs, J., *J2ME: The next major games platform?*, URL: [http://www.javaworld.com/javaworld/jw-03-2001/jw-0309-games\\_p.html](http://www.javaworld.com/javaworld/jw-03-2001/jw-0309-games_p.html). Consultado em 1 de setembro de 2001.
- [49] Buzztime, URL: <http://www.buzztime.com>. Consultado em 3 de setembro de 2001.
- [50] WapDrive, *Buzztime announces the success of its WAP game*, URL: [http://www.wapdrive.com/DOCS/wap\\_news/02\\_17\\_2001.html](http://www.wapdrive.com/DOCS/wap_news/02_17_2001.html). Consultado em 3 de setembro de 2001.
- [51] USAToday, *Cell phone games expected to ring off hook*, URL: <http://www.usatoday.com/life/cyber/tech/review/games/2001-03-26-phone-games.htm>. Consultado em 3 de setembro de 2001.
- [52] Red Herring, *VC Whispers: High-stakes games, Venture capitalists play a game of chance with wireless investments*, URL: [http://redherring.com/index.asp?layout=special\\_report\\_gen&doc\\_id=1450019145&](http://redherring.com/index.asp?layout=special_report_gen&doc_id=1450019145&)

- [channel=70000007&rh\\_special\\_report\\_id=6600000066](#). Consultado em 3 de setembro de 2001.
- [53] Micro Java Network, *Developer Tools*, URL: <http://www.microjava.com/developer/tools>. Consultado em 6 de setembro de 2001.
- [54] JavaMobiles, URL: <http://www.javamobiles.com/>. Consultado em 7 de setembro de 2001.
- [55] Diário de Pernambuco, *Game ganha espaço feito gente grande*, URL: [http://www.pernambuco.com/diario/2001/08/07/info1\\_0.html](http://www.pernambuco.com/diario/2001/08/07/info1_0.html). Consultado em 7 de setembro de 2001.
- [56] Java Community Process, *PDA Profile for the J2ME Platform*, URL: <http://jcp.org/jsr/detail/075.jsp>. Consultado em 7 de setembro de 2001.
- [57] Java Community Process, *Personal Profile for the J2ME Platform*, URL: <http://java.sun.com/products/foundation>. Consultado em 7 de setembro de 2001.
- [58] White, J., JavaPro, *Big Plans for J2ME*, URL: <http://www.java-pro.com/upload/free/features/javapro/2001/05may01/jw0105/jw0105-1.asp>. Consultado em 8 de setembro de 2001.
- [59] Foundation Profile, URL: <http://java.sun.com/products/foundation/>. Consultado em 8 de setembro de 2001.
- [60] Java Community Process, *Java Game Profile*, URL: <http://jcp.org/jsr/detail/134.jsp>. Consultado em 8 de setembro de 2001.
- [61] Java Community Process, *Personal Profile*, URL: <http://jcp.org/jsr/detail/062.jsp>. Consultado em 8 de setembro de 2001.
- [62] Java Card, URL: <http://java.sun.com/products/javacard/>. Consultado em 9 de setembro de 2001.
- [63] Embedded Java, URL: <http://java.sun.com/products/embeddedjava/>. Consultado em 9 de setembro de 2001.
- [64] Personal Java, URL: <http://java.sun.com/products/personaljava/>. Consultado em 9 de setembro de 2001.
- [65] Tektronix Uses EmbeddedJava Technology to Enhance Instruments, URL: <http://java.sun.com/products/embeddedjava/success/tektronix.html>. Consultado em 9 de setembro de 2001.
- [66] Java TV, URL: <http://java.sun.com/products/javatv/>. Consultado em 9 de setembro de 2001.
- [67] Java Phone, URL: <http://java.sun.com/products/javaphone/>. Consultado em 9 de setembro de 2001.
- [68] Micro Java Network, *From PersonalJava to J2ME: Some Introductory Ideas*, URL: [http://www.microjava.com/articles/perspective/personaljava\\_j2me?content\\_id=1440](http://www.microjava.com/articles/perspective/personaljava_j2me?content_id=1440). Consultado em 9 de setembro de 2001.

- [69] Giguere, E., *Connected Device Configuration and the Foundation Profile*, URL: <http://developer.java.sun.com/developer/products/wireless/foundation/ttpps/cdcfoundation/>. Consultado em 9 de setembro de 2001.
- [70] Day, B., *Developing Wireless Applications using the Java 2 Platform, Micro Edition*, URL: <http://developer.java.sun.com/developer/products/wireless/getstart/articles/wirelessdev/wirelessdev.pdf>. Consultado em 9 de setembro de 2001.
- [71] OpenAL, URL: <http://www.openal.org>. Consultado em 13 de setembro de 2001.
- [72] EETimes.com, *Cell phone makers jump for Java*, URL: <http://www.eetimes.com/story/OEG20010607S0102>. Consultado em 13 de setembro de 2001.
- [73] Unreal Engine, URL: <http://www.unrealengine.com/>. Consultado em 1 de outubro de 2001.
- [74] LithTech, URL: <http://www.lithtech.com/>. Consultado em 1 de outubro de 2001.
- [75] Knuckletop Computing: The Javatm Ring, URL: <http://java.sun.com/features/1998/03/rings.html>. Consultado em 9 de setembro de 2001.
- [76] Fundamental Algorithms for Raster Graphics, URL: <http://www.acs.ilstu.edu/docs/ACS356/LectureNotes/module3.4.htm>. Consultado em 24 de setembro de 2001.
- [77] Foley, J.D. et al., *Introduction to Computer Graphics*, Addison-Wesley, 1997.
- [78] Amato J., *Collision Detection*, URL: <http://www.gamedev.net/reference/articles/article735.asp>. Consultado em 24 de setembro de 2001.
- [79] Bobic, N., *Advanced Collision Detection Techniques*, URL: [http://www.gamasutra.com/features/20000330/bobic\\_01.htm](http://www.gamasutra.com/features/20000330/bobic_01.htm). Consultado em 27 de setembro de 2001.
- [80] Roberts, D., *Collision Detection: Getting the most out of your collision tests*. URL: <http://www.ddj.com/articles/1995/9513/9513a/9513a.htm>. Consultado em 27 de setembro de 2001.
- [81] Woodcock, S., *Game AI: The State of the Industry*, URL: [http://www.gamasutra.com/features/20001101/woodcock\\_01.htm](http://www.gamasutra.com/features/20001101/woodcock_01.htm). Consultado em 2 de outubro de 2001.
- [82] Eberly, D. H., *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*, Morgan Kaufmann Publishers, 2001.
- [83] Sakamura, K., *A Java-enabled evolution*, IEEE Computer Society, URL: <http://www.computer.org/micro/mi2001/pdf/m4002.pdf>. Consultado em 3 de outubro de 2001.

- [84] Heiss, J., *BIG IN JAPAN*, URL: <http://java.sun.com/features/2001/03/docomo.html>.  
Consultado em 3 de outubro de 2001.
- [85] Lamothe, A., *Tricks of the Windows Game Programming Gurus*, Sams, 1999.