Pós-Graduação em Ciência da Computação

AVALIAÇÃO EXPERIMENTAL DE DETECÇÃO DE COLISÃO PARA JOGOS J2ME

Por

ERIC BRUNO PERAZZO MARIZ

Dissertação de Mestrado

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, JANEIRO/2006

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ERIC BRUNO PERAZZO MARIZ

"AVALIAÇÃO EXPERIMENTAL DE DETECÇÃO DE COLISÃO PARA JOGOS J2ME"

ESTE TRABALHO FOI APRESENTADO A PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DE PERNAMUCO COMO REQUISITO PARCIAL PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIA DA COMPUTAÇÃO.

ORIENTADOR: GEBER RAMALHO

CO-ORIENTADOR: ANDRÉ SANTOS

RECIFE, JANEIRO/2006

Mariz, Eric Bruno Perazzo
Avaliação experimental de detecção de colisão para jogos J2ME / Eric Bruno Perazzo Mariz. – Recife : O Autor, 2006.
x, 105 folhas : il., fig.

Dissertação (mestrado) – Universidade Federal de Pernambuco. CIn. Ciência da Computação, 2006.

Inclui bibliografia e apêndices.

1. Ciência da computação – Engenharia de software empírica. 2. Jogos móveis – Detecção de colisão – Avaliação experimental. 3. Breakout e Space Invaders – Implementação em J2ME. I. Título.

004.05	CDU (2.ed.)	UFPE
004.2	CDD (22.ed.)	BC2006-123

Resumo

Apesar da avaliação experimental ser uma abordagem aceita e bem difundida para validação científica na maioria das disciplinas, apenas recentemente ela tem sido sistematicamente usada em Engenharia de Software, para com o intuito de examinar experimentalmente abordagens de desenvolvimento. Neste contexto, o crescente mercado de jogos para dispositivos móveis tem uma alta demanda por pesquisas na área de Engenharia de Software Empírica, devido à necessidade de utilização de técnicas de desenvolvimento adequadas às limitações de memória e processamento destes dispositivos. Infelizmente, muito pouco tem sido feito ou relatado na literatura a respeito de avaliações empíricas de técnicas de desenvolvimento de jogos móveis, que são o principal tipo de aplicação móvel hoje em dia. Assim, o objetivo desta dissertação é fornecer um estudo experimental comparativo entre diferentes técnicas de detecção de colisão, função recorrente e muito freqüente, para jogos móveis usando a linguagem J2ME, que é o padrão atual de desenvolvimento.

São adotadas três métricas para servir de base na análise comparativa dos resultados: A performance em quadros (*frames*) por segundo; o percentual do tempo total gasto nos métodos mais relevantes; e o tamanho do código-fonte. Algumas técnicas de detecção de colisão em duas dimensões são implementadas em 2 jogos (O *Breakout* e o *Space Invaders*) como estudo de caso. As técnicas foram executadas tanto em emulador quanto em celulares. A análise dos resultados obtidos identifica, com base nas métricas de comparação, qual técnica de detecção melhor se aplica para cada um dos dois jogos escolhidos. Exemplificando, a partir dos resultados podemos confirmar que o jogo Breakout possui uma boa performance quando o mesmo é implementado com ladrilhos. Diferentemente, o jogo Space Invaders, que não possui características de um jogo baseado em ladrilhos, demonstrou um resultado bastante insatisfatório no uso desta técnica quando comparado com as outras.

Palavras-chave: Engenharia de Software Empírica, Detecção de Colisão, Avaliação Experimental, J2ME, Jogos Móveis.

Abstract

Despite Experimental evaluation is an accepted and widespread approach for scientific validation in most scientific disciplines, only recently it has been systematically used in Software Engineering, with the objective to experimentally examine development approaches. In this scenario, there has been a growing demand for research in the area of Empirical Software Engineering for mobile games, due to the needs of suitable development techniques that fit the processing and memory limitations of such devices. Unfortunately, little has been made or reported on the literature regarding empirical evaluation of development techniques for mobile games, which are the main kind of mobile applications nowadays. Thus, the objective of this dissertation is to provide a comparative experimental study among different collision detection techniques, a very frequent and recurrent function, for mobile games using J2ME, the current standard development language.

Three variables are used as the basis for the comparative analysis of the results: The performance in *frames per second*; the percentage of time spent in most relevant methods; and the source code size. Some 2D collision detection techniques are implemented in two games (*Breakout* and *Space Invaders*) as a case study. The techniques were executed in an emulator and in cell phones. Considering the comparison variables, the analysis of the results identifies which is the best technique for each game. For instance, based on the results, one can confirm that the Breakout game has a good performance when implemented with a tile-based collision detection technique. Space Invaders, which has no characteristics of a tile-based game, presented a very unsatisfactory result when implemented with tiles in comparison with the other techniques.

Keywords: Empirical Software Engineering, Collision Detection, Experimental Evaluation, J2ME, Mobile games.

Glossário

Nesta seção são descritos algumas abreviações e acrônimos utilizados neste trabalho e necessários ao bom entendimento do mesmo.

2D	Indicativo de “duas dimensões”. Geralmente usado para designar que um determinado jogo ou característica é em duas dimensões.
3D	Indicativo de “três dimensões”.
Bomba	Usada para designar o ataque de uma nave inimiga (*Non Player Character*) no jogo *Space Invaders*.
Fps	*Frames* por segundo. Número de iterações (*ticks*) executado no jogo em um segundo.
GPS	*Global Positioning System*. Sistema que identifica a posição de um dado objeto em coordenadas no globo terrestre.
J2ME	Java 2 Micro Edition.
J2SE	Java 2 Standard Edition.
MIDlet	Definição de uma aplicação MIDP. Também usado para designar a classe que é o ponto de partida de execução de uma aplicação MIDP.
MIDP	Mobile Information Device Profile.
NCSL	*Non-Comment Source Lines*. Número de linhas sem comentários e sem espaços em branco.
NPC	*Non Player Character*. Definição para personagens de um jogo que não são controlados pelo jogador. Ex: Naves inimigas, monstros.
Profiler	Ferramenta do Wireless Toolkit utilizada para obter resultados de performance.
SDK	Software Development Kit
Tick	É a unidade de medida do tempo de execução de uma iteração completa do laço principal do jogo, ou seja, 1 tick é o tempo que leva para o jogo atualizar os objetos, fazer cálculos necessários e atualizar a tela.
Tiro	Usada para designar o ataque da nave do jogador no jogo *Space Invaders*.
Treatment	A técnica ou ferramenta que se deseja avaliar (comparada com outra técnica ou ferramenta).

Sumário

1 Introdução 1

2 A Plataforma J2ME 4

2.1 Uma Visão Geral 4

2.2 Estrutura de um programa J2ME 6

2.3 O Wireless Toolkit 8

2.3.1 Coletando Dados de Performance 9

2.3.1.1 Utilizando o *Profiler* 10

2.4 Conclusão do capítulo 11

3 Metodologia 12

3.1 Avaliação Experimental em Engenharia de Software 12

3.1.1 Projeto e Análise em Engenharia de Software 13

3.1.1.1 Etapas no Processo de Experimentação 15

3.2 Proposta 17

3.3 Visão geral dos jogos escolhidos 19

3.3.1 Uma consideração inicial 19

3.3.2 O Breakout 19

3.3.3 O Space Invaders 21

3.4 Conclusão do capítulo 22

4 Detecção de Colisão 23

4.1 Detecção de Colisão em Jogos 23

4.1.1 Tipos de Detecção de Colisão 24

4.2 Alguns Conceitos sobre Jogos 24

4.3 Técnicas de Detecção de Colisão 27

4.3.1 Detecção Padrão (*Object-to-objcet*) 27

4.3.2 Região de Influência 29

4.3.3 Ordenação por Eixo (Axis Sorting) 30

4.3.4 Gerenciador de Objetos 33

4.3.5 Detecção pelo Método do Setor 35

4.3.6 Detecção de Colisão Baseada em Ladrilhos 36

4.4 Conclusão do Capítulo 38

5 Implementação e Resultados 40
5.1 Estudos de caso 40
5.1.1 Implementação do Breakout 40
5.1.2 Implementação do Space Invaders 44
5.2 Configurações Experimentais 47
5.2.1 Configuração do Ambiente 47
5.2.2 Configuração de Variáveis 48
5.2.3 Etapas para a Execução do Experimento 49
5.3 Resultados 51
5.3.1 Resultados do Breakout 52
5.3.1.1 Métrica 1: Número de quadros por segundo 52
5.3.1.2 Métrica 2: Tamanho de código 59
5.3.1.3 Métrica 3: Percentual de execução dos métodos relevantes 59
5.3.2 Análise dos Resultados para o Jogo Breakout 62
5.3.2.1 Instabilidade dos resultados do Breakout 63
5.3.2.2 Efeito Colateral da Interface Gráfica 65
5.3.3 Resultados do Space Invaders 65
5.3.3.1 Métrica 1: Número de quadros por segundo 66
5.3.3.2 Métrica 2: Tamanho de código 72
5.3.3.3 Métrica 3: Percentual de execução dos métodos relevantes 73
5.3.4 Análise dos Resultados para o Jogo Space Invaders 75
5.4 Conclusão do Capítulo 76

6 Conclusões 78
6.1 Contribuições 78
6.2 Dificuldades encontradas 79
6.3 Trabalhos futuros 80
6.3.1 A Evolução do J2ME (MIDP 2.0) 80
6.3.2 Outras considerações 82

Apêndice A: Funções de Detecção de Colisão (Breakout) 84
A.1 Detecção padrão 84
A.2 Região de influência 84
A.3 Ordenação pelo eixo x desconsiderando a região de influência 85
A.4 Ordenação pelo eixo x com região de influência 85
A.5 Ordenação pelo eixo y desconsiderando a região de influência 86

A.6 Ordenação pelo eixo y com região de influência 86
A.7 Gerenciador de objetos (com quatro objetos por gerenciador) 87
A.8 Gerenciador de objetos (com oito objetos por gerenciador) 87
A.9 Baseada em ladrilhos pequenos (6 x 4) 88
A.10 Baseada em ladrilhos grandes (6 x 12) 90

Apêndice B: Funções de Detecção de Colisão (Space Invaders) 93

B.1 Detecção padrão 93
B.2 Região de influência 93
B.3 Ordenação pelo eixo x desconsiderando a região de influência 94
B.4 Ordenação pelo eixo x com região de influência 95
B.5 Ordenação pelo eixo y desconsiderando a região de influência 96
B.6 Ordenação pelo eixo y com região de influência 97
B.7 Método do setor 98
B.8 Baseada em ladrilhos 98

Lista de Figuras

Figura 2.1: Edições da linguagem Java. 4
Figura 2.2: Arquitetura do J2ME. 5
Figura 2.3: Componentes de uma *suite* J2ME. Figura retirada da introdução ao Wireless Toolkit 1.04 (Guia do usuário). 6
Figura 2.4: Tela da aplicação "*Hello World*". 8
Figura 2.5: Procedimentos de desenvolvimento e teste de uma aplicação J2ME. Imagem retirada da introdução ao Wireless Toolkit 1.04 (Guia do usuário). 9
Figura 2.6: Resultados do tempo de execução dos métodos de uma aplicação mostrados pela ferramenta *Profiler*. 10
Figura 3.1: Diagrama de classes inicialmente definida para a implementação das técnicas de detecção de colisão. 19
Figura 3.2: (a) Tela original do jogo Breakout. (b) Tela do jogo após modificação para dar suporte aos experimentos. 20
Figura 3.3: Tela do jogo Space Invaders. 22
Figura 4.1: Exemplo de um retângulo de circunscrição de um objeto (*bounding Box*). 24
Figura 4.2: Seqüência de imagens que formam um *sprite* de um personagem andando. 25
Figura 4.3: (a) Exemplo de um ladrilho isométrico. (b) Exemplo de terreno construído a partir de vários ladrilhos. 26
Figura 4.4: O Jogo *Snake* baseado em ladrilhos. Cada quadrado é um ladrilho que pode representar diferentes objetos do jogo, como o queijo, a parede, o corpo da cobra, etc. 26
Figura 4.5: Ilustração da interseção entre dois retângulos de circunscrição. 28
Figura 4.6: Verificação de colisão padrão. A cada iteração do jogo, verifica-se todas as possibilidades de colisão. Não há eliminação de testes. 29
Figura 4.7: Primeiro se verifica se a bola entrou na região de influência. Em caso positivo, a detecção de colisão é aplicada entre a bola e todos os tijolos. 30
Figura 4.8: Implementação da técnica de região de influência para o Space Invaders. 30
Figura 4.9: Objetos ordenados pelo eixo x. 31
Figura 4.10: Indexando pelo eixo x. A partir do 9º tijolo não é preciso mais verificar colisão, pois os tijolos possuem uma coordenada x maior do que a coordenada x da bola mais sua largura. (Ordenação pelo eixo x). 33

Figura 4.11: A partir da 13º nave inimiga não é mais necessário testar colisão, pois as naves possuem uma coordenada x maior que a coordenada x mais a largura do tiro. (Ordenação pelo eixo x). 33
Figura 4.12: Técnica de gerenciadores de objetos. (a) 4 gerenciadores por 4 objetos para o nível 1; (b) 2 gerenciadores por 8 objetos para o nível 1; (c) 12 gerenciadores por 4 objetos para o nível 2; (d) 6 gerenciadores por 8 objetos para o nível 2. 35
Figura 4.13: Os objetos 1 e 2 estão presentes num mesmo setor e devem ser testados entre si. Assim como os objetos 3 e 4, onde o objeto 3 está presente em dois setores. Objeto 5 faz parte de 4 setores. 36
Figura 4.14: (a) Representação gráfica do mapa do jogo; (b) Representação do deslocamento contínuo dentro de um ladrilho vazio. (x1, y1) é o ponto relativo atual de deslocamento dentro do ladrilho. 37
Figura 4.15: Supondo que a bola esteja com movimento vertical ascendente. Na próxima atualização, a bola colidirá com o tijolo, pois a mesma passaria a ocupar um ladrilho ocupado por um tijolo. 38
Figura 4.16: O jogo Space Invaders implementado em ladrilhos. O fundo do jogo mostra o mapa de ladrilhos, onde estes ladrilhos possuem o tamanho das bombas (e tiros). Ou seja, as naves inimigas e do jogador ocupam vários ladrilhos. 38
Figura 5.1: Diagrama de classes da implementação das técnicas de detecção de colisão para o Breakout. 42
Figura 5.2: Diagrama de classes da implementação das técnicas de detecção de colisão para o Space Invaders. 45
Figura 5.3: Parte da saída gerada pelo console do Eclipse ao final de execução de um teste para a técnica padrão do Breakout. Estão sendo mostrados aqui apenas 10 objetos do primeiro nível. 50
Figura 5.4: Saída gerada numa tela do emulador. 50
Figura 5.5: Média dos valores de fps para o nível 1 do jogo Breakout (executado no PC). 53
Figura 5.6: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 1 do jogo Breakout (executado no PC). 53
Figura 5.7: Média dos valores de fps para o nível 2 do jogo Breakout (executado no PC). 54
Figura 5.8: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 2 do jogo Breakout (executado no PC). 54

Figura 5.9: Média dos valores de fps para o nível 1 do jogo Breakout (executado no celular MC60). 55
Figura 5.10: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 1 do jogo Breakout (executado no celular Siemens MC60). 55
Figura 5.11: Média dos valores de fps para o nível 2 do jogo Breakout (executado no celular MC60). 56
Figura 5.12: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 2 do jogo Breakout (executado no celular Siemens MC60). 56
Figura 5.13: Média dos valores de fps para o nível 1 do jogo Breakout (executado no celular 6100). 57
Figura 5.14: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 1 do jogo Breakout (executado no celular Nokia 6100). 57
Figura 5.15: Média dos valores de fps para o nível 2 do jogo Breakout (executado no celular 6100). 58
Figura 5.16: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 2 do jogo Breakout (executado no celular Siemens MC60). 58
Figura 5.17: Comparação do tamanho de código para as técnicas implementadas no jogo Breakout. 59
Figura 5.18: Percentual de execução do método `drawWorld`. 60
Figura 5.19: Percentual de execução do método `updateWorld`. 61
Figura 5.20: Percentual de execução do método `checkIntersection`. 61
Figura 5.21: Colisão da bola com dois tijolos em *frames* consecutivos 64
Figura 5.22: Média dos valores de fps para o nível 1 do jogo Space Invaders (executado no PC). 66
Figura 5.23: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 1 do jogo Space Invaders (executado no PC). 67
Figura 5.24: Média dos valores de fps para o nível 2 do jogo Space Invaders (executado no PC). 68

Figura 5.25: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 2 do jogo Space Invaders (executado no PC). 68
Figura 5.26: Média dos valores de fps para o nível 1 do jogo Space Invaders (executado no celular MC60). 69
Figura 5.27: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 1 do jogo Space Invaders (executado no celular Siemens MC60). 69
Figura 5.28: Média dos valores de fps para o nível 2 do jogo Space Invaders (executado no celular MC60). 70
Figura 5.29: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 2 do jogo Space Invaders (executado no celular Siemens MC60). 70
Figura 5.30: Média dos valores de fps para o nível 1 do jogo Space Invaders (executado no celular 6100). 71
Figura 5.31: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 1 do jogo Space Invaders (executado no celular Nokia 6100). 71
Figura 5.32: Média dos valores de fps para o nível 2 do jogo Space Invaders (executado no celular 6100). 72
Figura 5.33: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 2 do jogo Space Invaders (executado no celular Nokia 6100). 72
Figura 5.34: Comparação do tamanho de código para as técnicas implementadas no jogo Space Invaders. 73
Figura 5.35: Percentual de execução do método drawWorld. 73
Figura 5.36: Percentual de execução do método updateWorld. 74
Figura 5.37: Percentual de execução do método checkIntersection. 74

1 Introdução

Uma linha de pesquisa que vem ganhando força em computação recentemente é a chamada Engenharia de Software Empírica [6], onde se procura, nos moldes do que se faz tradicionalmente na área de redes de computadores, realizar avaliações experimentais de abordagens de desenvolvimento [7][8][9].

No entanto, no campo específico de jogos não há muitos trabalhos que utilizem este tipo de estudo, exceto em computação gráfica, que emprega estudos empíricos na validação de algoritmos, a exemplo do trabalho de "Avaliação Comparativa de Visualização e Resultados Experimentais" por Hualin Zhou [3]. Particularmente na área de jogos para dispositivos móveis, a literatura não relata estudos desta natureza. Isto seria de grande relevância, primeiramente porque as grandes limitações de processamento, memória e display dos dispositivos móveis, demandam claramente este tipo de pesquisa. Em segundo lugar, porque se trata de uma indústria promissora: o montante investido em jogos móveis em 2003 foi de US$ 436,4 milhões e se expandirá para algo em torno dos US$ 4,4 bilhões em 2006 e 9,34 bilhões em 2008 [5].

De fato, devido à inerente deficiência de memória, processamento e *display* dos dispositivos móveis, faz-se necessário que os jogos desenvolvidos para estes dispositivos sejam otimizados [27], para tornar viável sua jogabilidade. Dentre as possibilidades de otimização a serem consideradas, pode-se citar: a otimização da função responsável por atualizar a tela, a otimização dos recursos computacionais pensando no melhor custo/benefício em termos de performance e uso de memória, a otimização da técnica de detecção de colisão, entre outras possibilidades. O algoritmo de detecção de colisão, em particular, possui um importante papel, visto que o mesmo tem uso recorrente e muito freqüente em praticamente todos os tipos de jogos. Isto nos motivou a desenvolver este trabalho.

Baseado no princípio de que jogos móveis precisam ter este nível de otimização, decidimos realizar um trabalho que servisse de base analítica, permitindo que desenvolvedores possam escolher a melhor técnica de detecção de colisão, ou mesmo uma técnica híbrida, para o tipo de jogo a ser desenvolvido. Nossa motivação foi reforçada pela inexistência atual de trabalho similar.

Um outro motivo, não menos importante, foi ajudar a difundir o estudo sobre Engenharia de Software Empírica, adotando sua teoria como princípio básico para a realização dos experimentos, ajudando assim a diminuir eventuais erros relacionados a uma má elaboração e estruturação do ambiente e do processo dos testes. É importante mencionar que a avaliação empírica ainda não é largamente utilizada em Engenharia de Software.

A idéia inicial era desenvolvermos diferentes tipos de detecção de colisão para diferentes tipos de jogos com características de jogabilidade distintas. Desenvolvemos onze diferentes técnicas (técnicas efetivas e variações) de detecção de colisão indo das mais simples como a técnica de detecção padrão até técnicas mais complexas como as baseadas em ladrilhos. Algumas dessas técnicas foram propostas nossas identificadas com base nas características dos jogos escolhidos. Ficamos restritos ao domínio 2D porque o alvo do estudo é o de jogos móveis, que em sua esmagadora maioria são em duas dimensões. Apenas recentemente alguns jogos 3D começaram a surgir, mas para um mercado ainda bastante restrito, pois os celulares com capacidade computacional para executar estes jogos ainda são bastante caros.

No entanto, devido à complexidade e à necessidade de uma quantidade de tempo razoável para se implementar os algoritmos e executar os testes para análise dos mesmos, decidimos utilizar só dois estudos de caso: os jogos Breakout e Space Invaders. Além de serem clássicos, estes jogos foram escolhidos por terem características distintas, como a existência ou não de mobilidade dos objetos na tela.

Um dos primeiros passos a serem tomados na consolidação da abordagem é a definição das métricas para dar suporte à análise comparativa entre as diferentes técnicas de detecção de colisão. Para tal, as seguintes variáveis foram adotadas:

- número de quadros (*frames*) por segundo;
- percentual do tempo total gasto para os métodos mais relevantes;
- tamanho de código.

O número de quadros por segundo (*fps*) é a taxa com que os quadros que formam uma animação são calculados e desenhados para dar o aspecto de movimento, por exemplo, uma taxa de 10 quadros por segundo indica que 10 quadros da animação são mostrados a cada segundo. A partir de 24 fps a visão humana tem a ilusão de que a animação é contínua. O sistema de TV padrão NTSC (americano) opera a 30 *fps*, já o sistema PAL opera a 25 *fps* [35][36].

O percentual do tempo total gasto dá uma idéia do quanto foi despendido com o processamento da detecção de colisão em função do tempo total decorrido durante o experimento.

O tamanho do código é medido pelo número de linhas de código (não inclusas as linhas em branco e os comentários). Quanto maior o código-fonte, maior o tamanho final do MIDlet. Esta métrica é importante para avaliar se um incremento no tamanho final de um MIDlet (visto que o mesmo irá rodar em um ambiente com poucos recursos) representa um benefício quando comparada com as outras métricas. Isto é, se um aumento no tamanho do código, com o objetivo de melhoria de performance, representa um ganho satisfatório.

Uma métrica bastante utilizada na análise de performance de um algoritmo é o tempo gasto na execução do método que o implementa [15][20][21]. Como o tempo gasto na execução dos métodos de detecção de colisão é da ordem de microssegundos, não pudemos utilizar esta métrica, pois a API do J2ME não disponibiliza um método que retorne o tempo decorrido em microssegundos, apenas em milisegundos.

Decidimos rodar o experimento tanto no emulador (para PC) quanto em celulares. O número de quadros por segundo é a única métrica que é possível utilizar para gerar resultados a partir de aparelhos celulares. O emulador disponibiliza uma estatística em função dos métodos das classes que compõem o jogo. Algo que não é possível se obter por meio dos celulares. Uma das variáveis geradas nesta estatística é a do percentual do tempo gasto por um método em função do tempo total de execução do jogo. Decidimos utilizar esta métrica para alguns métodos que classificamos como relevantes na implementação das técnicas de detecção de colisão para auxiliar na análise comparativa das mesmas.

2 A Plataforma J2ME

2.1 Uma Visão Geral

J2ME ou *Java 2 Micro Edition* [33] é uma plataforma usada para desenvolver aplicações Java para dispositivos tais como, PDA's (*Personal Digital Assistant*), telefones celulares, sistemas embarcados, etc. De acordo com a Sun, "J2ME foi projetada para dispositivos com memória, display e processamento limitados.". O J2ME é um subconjunto do J2SE [34] cujas principais diferenças concentram-se na camada de interface e de armazenamento de dados. Abaixo segue um diagrama com as edições do Java existentes atualmente:

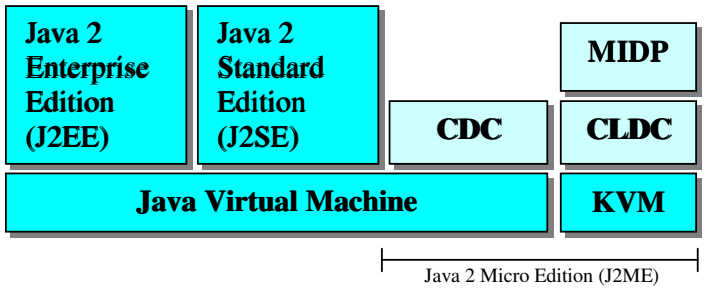


Figura 2.1: Edições da linguagem Java.

Observando a imagem acima pode-se notar dois lados distintos, a *Java Virtual Machine* (JVM) e a *Kilobyte Virtual Machine* (KVM). A JVM roda no PC, no servidor, e contém a biblioteca e a API completas do Java 2. A KVM é um subconjunto dessa biblioteca e API. Para maiores detalhes sobre *Virtual Machines*, ver [4][37][38].

O J2ME é dividido em Configurações. Uma Configuração provê o mais básico conjunto de bibliotecas que deve estar presente em cada implementação de um ambiente J2ME. Uma das Configurações é o CDC (*Connected Device Configuration*) [28]. O CDC é um subconjunto das classes do J2SE, que disponibiliza as funcionalidades do Java não relacionadas à interface gráfica, ou ao modo como as aplicações são carregadas ou ativadas (estas são funcionalidades dos *Profiles*). Exemplos de dispositivos que suportam o CDC são: *smart communicators, pagers, Personal Digital Assitants (PDAs)*, entre outros.

Uma outra Configuração é a CLDC (*Connected Limited Device Configuration*) [29][30]. Esta é uma configuração mais limitada que a CDC. Isto é, ela disponibiliza um subconjunto bem mais enxuto das classes do J2SE, para que seja suportada por

dispositivos com capacidade bem mais limitada de processamento e memória. *Palmtops*, dispositivos embutido de propósito específico, e Celulares são exemplos de dispositivos que utilizam esta Configuração. Estes dispositivos devem obedecer a uma configuração mínima de:

- 128 KB de memória para rodar o Java;
- 32 KB de memória para alocação em tempo de execução;
- interface restrita com o usuário;
- pouca energia, tipicamente um dispositivo com bateria;
- conectividade de rede, tipicamente sem fio, uma faixa de banda pequena e acesso intermitente.

Profiles são extensões de configurações. Eles disponibilizam as bibliotecas necessárias para o desenvolvedor escrever programas para um certo tipo de dispositivo. Como exemplo pode-se citar o MIDP (*Mobile Information Device Profile*) [31][32] que define API's para: componentes de interface com o usuário, armazenamento persistente, rede e temporizadores e entrada e tratamento de evento. As API's foram desenvolvidas levando em consideração as limitações do tamanho da tela e de memória dos dispositivos móveis.

A Figura abaixo mostra a arquitetura geral para o J2ME:

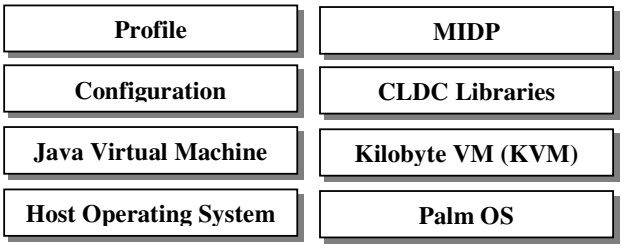


Figura 2.2: Arquitetura do J2ME.

Profiles representam a camada mais alta da arquitetura, disponibilizando os métodos necessários para escrever programas para um conjunto específico de dispositivos.

Configurações representam a camada de suporte à camada de *profiles*, responsáveis pela comunicação com a API de baixo nível do dispositivo móvel. Esta camada é a principal responsável pela implementação nativa das funcionalidades disponibilizadas na camada superior.

Existem alguns simuladores para rodar o J2ME, como o *Wireless Toolkit* da Sun, porém os softwares desenvolvidos para os simuladores nem sempre refletem o comportamento exato quando executados nos dispositivos.

2.2 Estrutura de um programa J2ME

Como dito anteriormente, a API de mais alto nível utilizada por desenvolvedores é o MIDP. Os programas desenvolvidos utilizando-se esta API são conhecidos como MIDlets.

Uma *suite* é um pacote contendo dois arquivos, um arquivo texto chamado JAD que contém informações específicas à aplicação, e um arquivo binário chamado JAR. O JAR nada mais é que um arquivo compactado, utilizando o algoritmo ZIP, contendo todos os recursos necessários à execução da aplicação, incluindo um ou mais MIDlets, arquivos de imagens, texto e um arquivo especial chamado Manifest que, assim como o JAD, contém informações sobre a aplicação (Ver Figura 2.3). A diferença entre os dois é que o JAD pode conter dados customizados do usuário, do tipo chave-valor. Estes parâmetros customizados podem ser acessados pela aplicação por meio de um método da classe MIDlet chamado `getAppProperty`, onde passando-se a chave correspondente (uma String), obtém-se o valor correspondente àquela chave, caso a mesma esteja presente no JAD, caso contrário, o valor nulo é retornado.

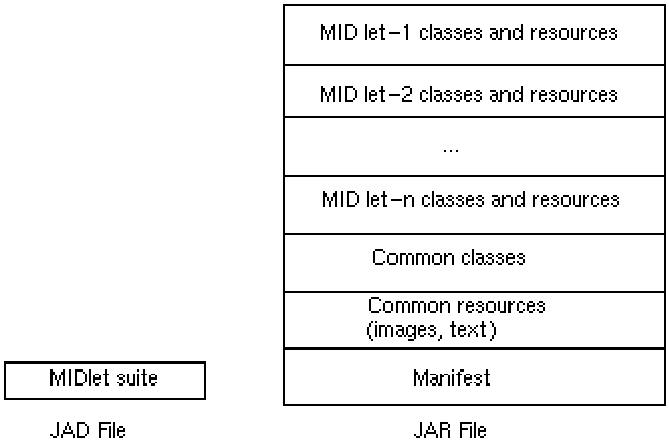


Figura 2.3: Componentes de uma *suite* J2ME. Figura retirada da introdução ao Wireless Toolkit 1.04 (Guia do usuário).

Abaixo segue um exemplo simples de um MIDlet. O programa, quando executado, deverá mostrar na tela a frase “Hello World” e um comando “Sair” deverá

estar disponível no lado inferior da tela, onde uma vez executado deverá encerrar a aplicação (Figura 2.4).

```
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;

public class HelloMIDlet extends MIDlet implements CommandListener
{
  private Form mainForm;
  private boolean isPaused;

  public HelloMIDlet() {
    mainForm = new Form("Hello World");
    this.isPaused = false;
  }

  public void startApp() {
    if (!this.isPaused) { // chamado pela primeira vez
      mainForm.append(new StringItem(null, "Hello World"));
      mainForm.addCommand(new Command("Sair", Command.EXIT, 0));
      mainForm.setCommandListener(this);
      Display.getDisplay(this).setCurrent(mainForm);
    } else { // resumindo a aplicação
      Display.getDisplay(this).setCurrent(mainForm);
    }
  }

  public void pauseApp() { }

  public void destroyApp(boolean unconditional) {
    super.notifyDestroyed();
  }

  public void commandAction(Command c, Displayable d) {
    this.destroyApp(false);
  }
}
```

Toda aplicação J2ME deve ter pelo menos uma classe que estende da classe abstrata "javax.microedition.midlet.MIDlet". Essa classe possui três métodos abstratos que precisam ser implementados: `startApp`, `pauseApp` e `destroyApp`. O método `startApp` é o ponto de partida para a execução da aplicação e é chamado pela KVM para dar início à aplicação. O método `pauseApp` é executado quando ocorre algum evento externo à aplicação que a faz parar de executar, como a recepção de uma ligação por exemplo. Ele deve ser responsável por armazenar o estado da aplicação antes da parada de sua execução para sua posterior recuperação, onde o método `startApp` é chamado novamente para continuar a execução da aplicação. E finalmente o método `destroyApp` deve ser executado antes da finalização da aplicação, onde duas de suas funções, entre possíveis outras, são: armazenar os dados persistentes que ainda não foram salvos e chamar o método `notifyDestroyed` para avisar a KVM que a aplicação pode ser finalizada.

Figura 2.4: Tela da aplicação "*Hello World*".

2.3 O Wireless Toolkit

O Wireless Toolkit é uma ferramenta para o desenvolvimento de aplicações móveis usando a linguagem de programação Java. Ele é um ambiente de emulação para desenvolver aplicações especificamente para dispositivos móveis que suportam o CLDC/MIDP.

O Wireless Toolkit pode ser usado tanto isoladamente quanto integrado a ambientes de desenvolvimentos. Atualmente existem diversos IDEs (*Integrated Deveolpment Environment*) que suportam a ferramenta, como o Borland JBuilder[1], o Metrowerks CodeWarrior Wireless Studio[2], o Eclipse[3], entre outros.

As fases de desenvolvimento de uma aplicação MIDP consistem em (Figura 2.5):

- criação/Edição dos arquivos fontes (arquivos ".java");
- compilação desses arquivos fontes em *bytecodes* (arquivos ".class");
- preverificação dos arquivos compilados;
- execução e depuração.

O ambiente de desenvolvimento Eclipse foi utilizado no projeto junto com a integração do *Wireless Toolkit* fazendo uso de um *plugin*. Para maiores detalhes sobre as ferramentas utilizadas e suas configurações, ver Seção 5.2. A seguir é descrito como usar a ferramenta de gerenciamento de performance do *Wireless Toolkit*, que servirá para coletar os dados da métrica de percentual de tempo gasto dos métodos relevantes.

---

[1] http://www.borland.com/us/products/jbuilder/index.html

[2] http://www.microdevnet.com/nokia/toolkits/cww/

[3] http://www.eclipse.org/

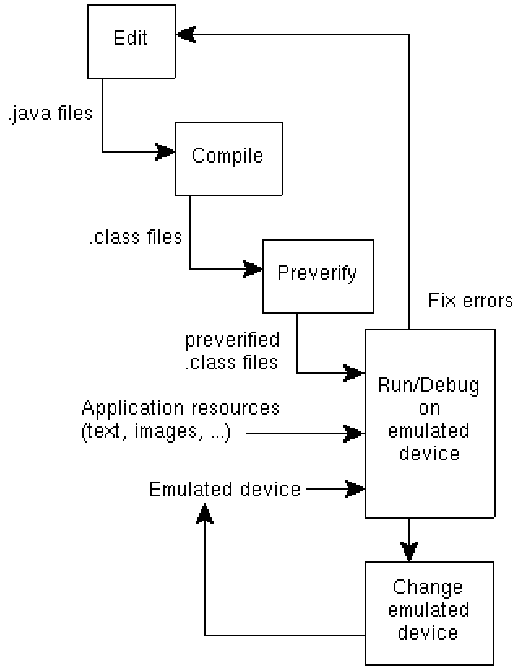


Figura 2.5: Procedimentos de desenvolvimento e teste de uma aplicação J2ME. Imagem retirada da introdução ao Wireless Toolkit 1.04 (Guia do usuário).

2.3.1 Coletando Dados de Performance

As ferramentas de gerenciamento de performance do J2ME *Wireless Toolkit* são utilizadas principalmente para examinar vários aspectos relativos às aplicações, tendo em vista a melhoria da velocidade de execução e do uso de memória dos MIDlets. Neste caso, é feito uso destas ferramentas para coletar dados que possam nos fornecer uma análise comparativa entre as técnicas de detecção de colisão. O *Wireless Toolkit* fornece as seguintes ferramentas de análise de performance:

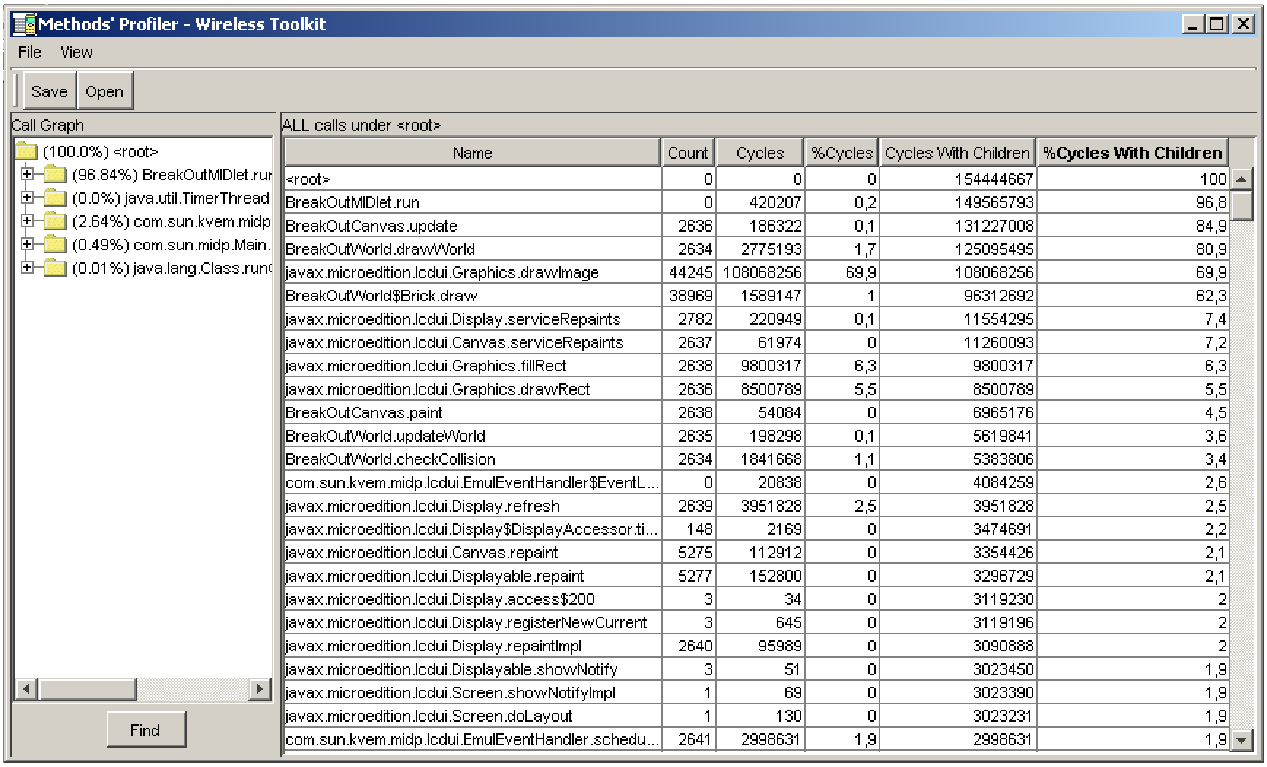
- *Profiler*: Permite monitorar o tempo de execução e a freqüência de uso de todos os métodos da aplicação.
- Monitor de Memória: Permite que o uso de memória da aplicação seja examinado.
- Monitor de Rede: Disponibiliza o monitoramento de tráfego de informação entre o dispositivo e a rede.
- Emulador de Velocidade: Permite ajustar a velocidade de atualização da tela para refinar a renderização gráfica. Além disso, ele permite que seja ajustada a velocidade de execução dos *byte-codes* pela velocidade de transmissão de dados pela rede.

O *Profiler* é mostrado na seção seguinte, pois é a única ferramenta que interessa ao escopo deste trabalho.

2.3.1.1 Utilizando o *Profiler*

O *profiler* analisa o tempo de execução dos métodos da aplicação e é bastante usado para identificar onde os problemas em potencial, como os gargalos, podem existir para que se possam fazer os ajustes finos.

Ele disponibiliza dois tipos de informações, uma lista hierárquica dos métodos da aplicação, e informações sobre o tempo de execução dos métodos e a quantidade de vezes que eles foram chamados. A Figura 2.6 mostra um exemplo de resultado da ferramenta.



Name	Count	Cycles	%Cycles	Cycles With Children	%Cycles With Children
<root>	0	0	0	154444667	100
BreakOutMIDlet.run	0	420207	0,2	149565793	96,8
BreakOutCanvas.update	2636	186322	0,1	131227008	84,9
BreakOutWorld.drawWorld	2634	2775193	1,7	125095495	80,9
javax.microedition.lcdui.Graphics.drawImage	44245	108068256	69,9	108068256	69,9
BreakOutWorld$Brick.draw	38969	1589147	1	96312692	62,3
javax.microedition.lcdui.Display.serviceRepaints	2782	220949	0,1	11554295	7,4
javax.microedition.lcdui.Canvas.serviceRepaints	2637	61974	0	11260093	7,2
javax.microedition.lcdui.Graphics.fillRect	2638	9800317	6,3	9800317	6,3
javax.microedition.lcdui.Graphics.drawRect	2636	8500789	5,5	8500789	5,5
BreakOutCanvas.paint	2638	54084	0	6965176	4,5
BreakOutWorld.updateWorld	2635	198298	0,1	5619841	3,6
BreakOutWorld.checkCollision	2634	1841668	1,1	5383806	3,4
com.sun.kvem.midp.lcdui.EmulEventHandler$EventL...	0	20838	0	4084259	2,6
javax.microedition.lcdui.Display.refresh	2639	3951828	2,5	3951828	2,5
javax.microedition.lcdui.Display$DisplayAccessor.ti...	148	2169	0	3474691	2,2
javax.microedition.lcdui.Canvas.repaint	5275	112912	0	3354426	2,1
javax.microedition.lcdui.Displayable.repaint	5277	152800	0	3296729	2,1
javax.microedition.lcdui.Display.access$200	3	34	0	3119230	2
javax.microedition.lcdui.Display.registerNewCurrent	3	645	0	3119196	2
javax.microedition.lcdui.Display.repaintImpl	2640	95989	0	3090888	2
javax.microedition.lcdui.Displayable.showNotify	3	51	0	3023450	1,9
javax.microedition.lcdui.Screen.showNotifyImpl	1	69	0	3023390	1,9
javax.microedition.lcdui.Screen.doLayout	1	130	0	3023231	1,9
com.sun.kvem.midp.lcdui.EmulEventHandler.schedu...	2641	2998631	1,9	2998631	1,9

Figura 2.6: Resultados do tempo de execução dos métodos de uma aplicação mostrados pela ferramenta *Profiler*.

As colunas da janela de dados mostram as seguintes informações:

- *Name*: O nome completo do método (*fully qualified name*).
- *Count*: A quantidade de vezes que um determinado método foi chamado durante a execução da aplicação.
- *Cycles*: O tempo de execução de um método (não leva em consideração o tempo de execução dos métodos chamados por este método).

- *%Cycles*: O percentual de tempo gasto na execução de um método em relação ao tempo total que o programa executou (não leva em consideração o tempo de execução dos métodos chamados por este método).
- *Count with children*: A quantidade de vezes que um método e seus descendentes foram chamados durante a execução da aplicação.
- *%Count with children*: O tempo percentual gasto executando o método e todos os seus descendentes em relação ao tempo que o programa ficou executando.

Para efeito da coleta de dados para a métrica, é levada em consideração apenas a última informação, que representa um percentual do tempo total gasto pelos métodos mais relevantes (e seus descendentes). Os métodos relevantes são discutidos mais adiante na seção que detalha a implementação dos MIDlets.

Esta ferramenta, para coletar os dados necessários, introduz um atraso na execução do MIDlet como um todo, mas que é distribuído uniformemente entre os métodos, o que não representa um problema, já que estes dados são usados de forma comparativa.

2.4 Conclusão do capítulo

O J2ME foi a plataforma escolhida para o desenvolvimento do nosso trabalho por ser, atualmente, o padrão de mercado. A grande maioria dos jogos móveis é desenvolvida usando-se a linguagem Java.

O *Wireless Toolkit* é utilizado como ambiente de desenvolvimento e emulador para obtenção das métricas estabelecidas. Além do emulador de celular, que utilizamos para obter o número médio de quadros por segundo das técnicas de detecção de colisão, utilizamos também o *Profiler* para obter o percentual do tempo total gasto na execução dos principais métodos concernentes às técnicas de detecção de colisão.

3 Metodologia

Para atingir os objetivos, é preciso definir uma metodologia para a elaboração dos experimentos a serem realizados, bem como uma análise para garantir a validade dos resultados obtidos. Visando este aspecto, estudamos uma linha de pesquisa relativamente nova na Engenharia de Software como um todo, mas há tempos bastante utilizada e bem difundida por diversas ciências em vários campos de estudos, a validação de um determinado objeto de estudo por meio de experimentação.

Certamente não podemos utilizar a experimentação como método definitivo para provar uma teoria, mas sem dúvida podemos utilizá-la para provar que uma determinada teoria não é válida, simplesmente achando um contra-exemplo que invalida essa teoria.

3.1 Avaliação Experimental em Engenharia de Software

Segundo Walter Tichy [12], cientistas da computação defendem a falta de experimentação por meio de uma grande variedade de argumentos. Alguns argumentos sugerem que a experimentação é inapropriada, muito difícil, desnecessária, e até mesmo prejudicial. O fato é que muitas teorias em Ciência da Computação, qualquer que seja o objeto de estudo, não foram testadas. Por exemplo, programação funcional, programação orientada a objetos, e métodos formais são considerados objetos de melhoria de produtividade do programador, ou de melhoria de qualidade do sistema, ou ambas. No entanto, pouco estudo tem sido feito com o objetivo de validar essa afirmação.

Podemos notar claramente a necessidade de se provar a veracidade dessas alegações, ou pelo menos de se tentar estabelecer argumentos convincentes. É de se admirar que essas importantes alegações não tenham sido sistematicamente testadas, mesmo elas existindo há mais de 30 anos e tendo em vista o esforço empenhado no desenvolvimento de linguagens de programação e técnicas formais. A experimentação é, sem dúvida, um caminho para se alcançar o nível de argumentação necessário para continuar confiando em afirmações, pelo menos até que algum contra-exemplo seja encontrado e estas sejam derrubadas, propiciando o aparecimento de novas alegações.

Um exemplo da disparidade a respeito da experimentação entre a computação e qualquer outra ciência "mais tradicional" é bem citada por Tichy no artigo: "*Experimental Evaluation in Computer Science: A Quantitative Study*" [46], onde ele classifica 400 artigos. Numa amostra aleatória de todos os artigos publicados pela ACM em 1993, o estudo mostrou que 40% dos que precisavam de algum tipo de avaliação empírica não a tinham. Em jornais (*journals*) sobre software, esse valor chegou a 50%. O mesmo estudo também analisou um jornal não relacionado à Ciência da Computação, o *Optical Engineering*, e constatou que apenas 15% dos artigos que necessitavam, não tinham uma avaliação quantitativa.

3.1.1 Projeto e Análise em Engenharia de Software

O uso de uma determinada técnica, ferramenta ou método deve ser considerado por meio de um embasamento científico, em vez de simplesmente adotar sua utilização porque definem que estes são melhores que outros existentes no mercado, sem levar em consideração dados realísticos. Isto é, não se deve utilizar uma determinada ferramenta ou técnica apenas porque ela virou moda de mercado, por exemplo. É necessário levar em consideração um estudo prévio que demonstre, ou que pelo menos dê indícios, que a ferramenta, método ou técnica que se está avaliando é realmente melhor do que outra. Para isto, Shari Pfleeger [13] discute dois dos principais tipos de avaliação utilizados: o estudo de caso e o experimento formal.

O primeiro passo da investigação é saber o que se quer investigar. O objeto da investigação é que nos ajudará a escolher a técnica que melhor se aplica à situação. Esse objeto pode ser expresso como uma hipótese da investigação. Por exemplo, "Usar o método de projeto ABC produz um software de melhor qualidade que usar o método XYZ". Baseado na hipótese, um estudo de caso seria avaliar um "*snapshot*" da organização usando o método ABC, e um experimento formal seria fazer uma comparação cuidadosamente controlada entre o uso do método ABC e o uso do método XYZ. Vale também salientar que no caso da elaboração da hipótese, o mesmo deve ser definido em termos quantificáveis, facilitando assim a análise para indicar se a hipótese é confirmada ou rejeitada. Por exemplo, ao invés de estabelecer a hipótese como "Usar o método de projeto ABC produz um software de melhor qualidade que usar o método XYZ", seria melhor definir "qualidade" em termos quantitativos, que poderia ser em relação ao número de defeitos encontrados. Desta forma, a hipótese poderia ser redefinida para "O código produzido usando-se o método de projeto ABC tem um

menor número de defeitos por mil linhas de código que o código produzido usando-se o método XYZ".

Uma vez que a hipótese tenha sido estabelecida, deve-se verificar que variáveis podem afetar sua veracidade, e quanto controle se tem sobre cada uma delas. Este grau de controle sobre as variáveis é um fator discriminatório entre o estudo de caso e experimentos formais. Um estudo de caso é preferível quando se está examinando eventos onde comportamentos relevantes não podem ser manipulados. Por exemplo, ao se investigar o efeito de um método de projeto na qualidade final de um software, mas não há como se ter o controle sobre quem está usando que método de projeto, então provavelmente o estudo de caso será a melhor opção para documentar os resultados. Experimentos formais são utilizados quando se pode manipular o comportamento diretamente, precisamente e sistematicamente. Baseado no exemplo anterior, caso se possa controlar quem usa o método ABC, quem usa o método XYZ, e quando e onde eles são usados, então um experimento formal é possível.

Esta diferença entre estudo de caso e experimentos formais pode ser mais rigorosamente estabelecida considerando-se a noção de *variável de estado*. Uma variável de estado é um fator que pode influenciar o resultado da avaliação final de um dado projeto. Por exemplo, uma hipótese pode envolver os efeitos de uma linguagem na qualidade final do código. "Linguagem" neste caso é uma variável de estado, e um experimento envolveria a utilização de vários projetos que utilizassem o maior número de linguagens possível. Já o estudo de caso levaria apenas em consideração a linguagem mais utilizada na organização, ou seja, no estudo de caso escolhe-se um valor para a variável em questão e no experimento procura-se levar em consideração o máximo de valores possíveis para a variável.

Em um determinado experimento podem haver várias variáveis de estado. Neste caso, quando se quer identificar os efeitos de uma delas em relação a uma dada hipótese, deve-se fazer os valores das outras possíveis variáveis permanecerem os mesmos entre os experimentos. Por exemplo, tomando-se como referência a linguagem de programação como a variável de estado a ser verificada para identificar uma variação na produtividade do projeto, outras variáveis que podem afetar a produtividade, como a experiência na aplicação que está sendo desenvolvida, ambiente de programação, tipo de problema, entre outras, devem permanecer com o mesmo valor entre os experimentos.

Em geral, para se saber quando usar um estudo de caso e quando usar um experimento formal, identifica-se o nível de controle sobre as variáveis que podem afetar a verdade de uma hipótese. Se o nível for alto, então se pode trabalhar com um experimento. Caso não haja controle sobre essas variáveis, então se deve utilizar um estudo de caso.

3.1.1.1 Etapas no Processo de Experimentação

Shari Pfleeger [14] estabelece os seguintes passos na preparação de um experimento formal:

- concepção;
- projeto;
- preparação;
- execução;
- análise;
- disseminação e tomada de decisão.

A fase de *concepção* inclui o tipo de análise descrito na seção anterior: para identificar se o experimento formal é a técnica de avaliação mais apropriada para o que se quer investigar. Daí, deve-se estabelecer o objetivo do estudo de forma clara e precisa, que pode incluir mostrar que uma determinada ferramenta ou método é superior de alguma forma à outra, ou, de maneira similar, pode-se querer mostrar que um determinado fator pode afetar a qualidade de um produto, ou que uma determinada variável pode afetar o resultado gerado por um software. Independentemente da escolha do objetivo, ele deve ser determinado de tal forma que seja avaliado de forma clara ao final do experimento.

Na fase de *projeto*, dado que o objetivo tenha sido devidamente estabelecido, uma hipótese formal deve ser criada a partir desse objetivo. Freqüentemente são utilizados dois tipos de hipóteses: a hipótese nula e a hipótese experimental (ou alternativa). A hipótese nula não assume nenhuma diferença entre os tratamentos (*treatments*). Isto é, entre métodos, ferramentas, técnicas, ambientes, ou outros objetos que estejam sendo estudados, em relação a variável que está sendo analisada (como produtividade, qualidade ou custo). Já a hipótese alternativa assume uma diferença significativa entre dois tratamentos. Por exemplo, suponha que se queira saber se a técnica de desenvolvimento de software A produz um código de melhor qualidade que seu processo de desenvolvimento atual. A hipótese pode ser formulada da seguinte forma:

1. *Hipótese nula*: Não há diferença na qualidade do código entre código desenvolvido utilizando-se a técnica A e código produzido com o processo de desenvolvimento de software atual.
2. *Hipótese alternativa*: A qualidade do código produzido utilizando-se a técnica A é melhor que a do código produzido utilizando-se o processo de desenvolvimento atual.

As duas hipóteses podem ser utilizadas e ambas são assumidas como verdadeiras a menos que os dados resultantes do experimento indiquem o contrário.

Supondo que se queira testar um determinado método ou ferramenta, o experimento consistirá de uma série de testes desse método ou ferramenta que se quer avaliar. O documento de projeto deverá descrever como estes testes serão organizados e executados. A terminologia tratamento é usada para descrever, neste caso, a nova ferramenta ou método que se deseja avaliar. Na execução de qualquer teste individual, apenas um tratamento é usado. Um teste individual é às vezes chamado de amostra (*trial*) e um experimento é formalmente definido como um conjunto de amostras.

A fase de *preparação* envolve estabelecer o necessário para a execução dos testes. Algumas das ações a serem tomadas nesta fase são: verificar que ferramentas precisam ser adquiridas, treinar o pessoal que será envolvido na execução dos testes, verificar se o hardware necessário precisa ser configurado de certa maneira, etc. Se possível, propiciar uma execução simplificada do experimento com um pequeno conjunto de pessoas pode ser útil para garantir que o plano está completo e que todas as instruções necessárias à execução dos testes foram bem entendidas pelas pessoas que farão parte do experimento.

Finalmente, a fase de *execução* pode ser efetuada. Deve-se seguir os passos indicados no plano definido na fase anterior e coletar os dados das métricas que foram estabelecidas no plano. Deve-se ter cuidado durante a execução para garantir que os dados estejam consistentes, para viabilizar uma comparação confiável dos resultados.

A fase de *análise* possui duas partes. Na primeira, deve-se rever todas as medidas coletadas para se ter certeza que elas são válidas e úteis. Na segunda é feita uma análise para identificar se a hipótese é suportada ou recusada pelos resultados do experimento, ou seja, uma resposta para a questão estabelecida inicialmente na pesquisa é obtida.

Ao final da fase de análise, deve-se verificar como as características analisadas afetaram os resultados finais. Na fase de *disseminação e tomada de decisão*, é importante documentar as conclusões tiradas do experimento para que outras pessoas

possam replicá-lo e confirmar essas conclusões de maneira similar. As conclusões devem ser documentadas de forma clara, descrevendo todos os problemas encontrados durante a execução dos experimentos. Por exemplo, se o pessoal mudou durante o desenvolvimento do projeto, ou se uma ferramenta sofreu algum tipo de atualização. Tudo isso tem que ser levado em consideração e documentado.

3.2 Proposta

A proposta central deste trabalho é prover um estudo experimental sobre os diferentes tipos de detecção de colisão para jogos móveis. Subseções anteriores mostraram os aspectos que devem ser levados em consideração quando se deseja realizar um experimento. Como o objetivo é utilizar apenas dois tipos de jogos para realizar os experimentos, pode-se considerar que neste nível de análise, o projeto é um estudo de caso, pois leva em consideração apenas uma pequena amostra dos diversos tipos de jogos existentes e que poderiam ser desenvolvidos utilizando-se a plataforma J2ME. No entanto, entrando no foco do trabalho, a análise das técnicas de detecção de colisão, pode-se considerá-lo como um experimento formal, e é este aspecto que será abordado.

Visto que se trata de um experimento formal, os fatores relevantes com base no que foi relatado na seção anterior serão analisados. Em primeiro lugar, aplicou-se a técnica de detecção de colisão como a única variável de estado adotada nos experimentos. Isto implica dizer que o código na implementação das diferentes técnicas de detecção de colisão, nos dois jogos utilizados, deve ser o mesmo, ou pelo menos o mais similar possível entre as implementações. Idealmente, apenas a técnica utilizada deveria ser variada, para que qualquer variação nos dados coletados fosse atribuída à técnica utilizada. No entanto, para algumas técnicas, foi necessário fazer algumas modificações em outras porções do código, como será visto em capítulo posterior. Desta forma, o objetivo de avaliar comparativamente as diferentes técnicas em função das métricas adotadas é atingido.

Estabelecendo claramente, o objetivo é analisar que técnica de detecção de colisão melhor se aplica a um dado tipo de jogo. Além da análise comparativa, um resultado quantitativo que possa prover ao desenvolvedor a possibilidade de escolha de uma determinada técnica (ou mesmo uma solução híbrida, dependendo do tipo de jogo e de suas características inerentes) será disponibilizado.

Baseado no objetivo descrito acima e levando-se em consideração o conceito de hipótese nula descrita na seção anterior, a hipótese pode ser definida como: “Não há diferença de performance entre as técnicas de detecção de colisão consideradas”. No entanto, como foi dito anteriormente, a hipótese deve ser refinada ao máximo em termos quantitativos para que se possa ter uma maior precisão no momento da avaliação. Desta forma, tomando como base as métricas descritas na Introdução, a hipótese pode ser redefinida como: “Não há diferença no número de quadros por segundo ou do percentual do tempo de execução gasto nas funções mais relevantes entre as técnicas de detecção de colisão consideradas, bem como não há diferença no tamanho final do código entre as técnicas”. Partindo deste pressuposto, o experimento deve ser executado e, baseado na análise de seus resultados, a hipótese deve ser suportada ou rejeitada.

Entrando no mérito das etapas do processo de experimentação, foram estabelecidos: o objetivo, a atividade correspondente à etapa de concepção, e a hipótese, referente à etapa de projeto. O próximo passo é definir um documento de projeto que contenha as informações necessárias à execução dos testes. A própria dissertação será usada para documentar essas informações na seção de Configurações Experimentais. A fase da preparação consiste na configuração do ambiente onde os testes serão executado. Estes serão realizados utilizando o emulador *Wireless Toolkit* da Sun para os testes em PC e utilizando celulares que suportam o MIDP 1.0 [25]. Desta forma, o emulador precisa ser configurado como descrito na seção 5.2.1 e os celulares precisam estar com os MIDlets carregados e configurados (recomenda-se carregar e rodar um de cada vez para evitar problemas de memória nos celulares) para a execução do experimento. Após se configurar o ambiente adequadamente, os MIDlets poderão ser executados, gerando assim os dados necessários para a próxima etapa, a análise. Nesta etapa, a hipótese estabelecida poderá ser confirmada ou recusada, identificando em caso de recusa uma solução para o objetivo definido. No caso da etapa de disseminação e tomada de decisão, uma conclusão sobre os dados obtidos será produzida, reportando as informações necessárias. Com base nesta conclusão, outras pessoas poderão dar continuidade ou refinar o experimento afim de torná-lo mais completo. Além disto, será possível fornecer a base empírica para que desenvolvedores possam tomar uma decisão sobre que tipo de detecção de colisão pode ser adotado, levando-se em consideração o tipo de jogo sendo desenvolvido e suas características.

A seguir é mostrada uma visão geral sobre os jogos que dão suporte a este trabalho, para que se possa falar, nas seções subseqüentes, sobre as técnicas que foram implementadas para cada um dos dois jogos.

3.3 Visão geral dos jogos escolhidos

3.3.1 Uma consideração inicial

Inicialmente foi adotada uma estrutura de implementação das diferentes técnicas para um dado tipo de jogo baseada no *Strategy pattern* [2][1] (a Figura 3.1 mostra a estrutura de implementação inicialmente definida). Isto foi feito para que se pudesse isolar a porção de código que seria dinamicamente testada, aumentando a confiabilidade dos resultados da análise.

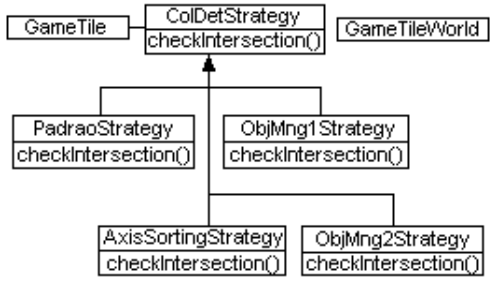


Figura 3.1: Diagrama de classes inicialmente definida para a implementação das técnicas de detecção de colisão.

No entanto, como esta estruturação faria com que as técnicas fossem implementadas todas numa mesma *suite* (Ver seção 2.2), o impacto destas técnicas na variável *tamanho de código* deixaria de ser analisado. Levando-se isto em consideração, cada técnica foi desenvolvida em uma *suite* independente. No entanto, ainda dentro do princípio de tentar maximizar o código comum. O mesmo código foi adotado na implementação dos diferentes tipos de detecção de colisão para um dado jogo, exceto para os métodos relevantes à técnica de detecção, cujo objetivo é tratado de forma detalhada mais adiante.

3.3.2 O Breakout

O Breakout é um dos dois jogos utilizados como estudo de caso para a implementação das técnicas de detecção de colisão. Ele foi escolhido por ser um jogo

simples, bastante conhecido, por ter a maioria de seus objetos estáticos e por se prestar bem à utilização de diversas técnicas de detecção de colisão 2D.

A seguir é descrito o conceito do jogo, bem como as características de desenvolvimento envolvidas.

São dois os objetos que constituem o jogo: a bola e o tijolo. Os tijolos ficam dispostos num esquema pré-determinado no início do jogo, o objetivo é fazer a bola colidir com estes tijolos fazendo-os desaparecer. Quando todos os tijolos são eliminados da tela, inicia-se uma nova fase com tijolos dispostos de forma diferente, em maior quantidade, com níveis maiores de resistência à colisão, ou uma miscelânea dessas características. O jogo original possui um objeto retangular embaixo na tela (uma espécie de "raquete", ver Figura 3.2 (a)), que é usado pelo jogador para fazer a bola refletir, sendo que quando a bola atinge o lado inferior da tela num ponto onde esse objeto retangular não alcança, o jogador perde uma "vida" e assim continuamente até o jogador perder todas as "vidas" ou atingir o último nível do jogo.

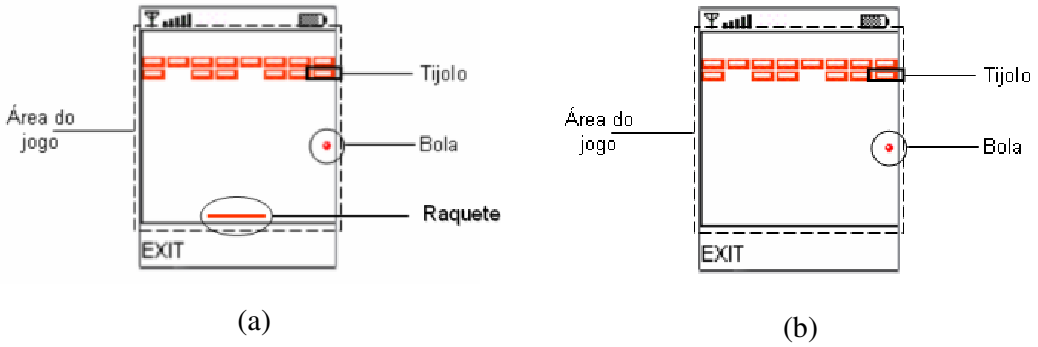


Figura 3.2: (a) Tela original do jogo Breakout. (b) Tela do jogo após modificação para dar suporte aos experimentos.

Um dos alicerces no processo de avaliação experimental é tornar os experimentos automatizados, ou seja, sem intervenção humana, de tal forma que o mesmo possa ser reproduzido diversas vezes gerando os mesmos resultados obtidos do experimento feito originalmente. Para efeito da análise experimental de desempenho dos algoritmos de detecção de colisão, as características de interação com o usuário tiveram que ser cortadas. Devido a esta modificação, a bola, ao colidir com uma das quatro paredes, atualiza automaticamente sua trajetória. Ou seja, o único ponto de intervenção externa com o jogo foi excluído (a raquete). Desta forma, a bola, ao colidir com o lado inferior da área de jogo, sofre reflexão, como mostrado na Figura 3.2 (b). Além disso, o ângulo e velocidade iniciais da bola tiveram que ser estabelecidos por meio de tentativa e erro.

Isto teve que ser feito para que não fosse caracterizada uma trajetória de repetição constante (caso ocorrido em algumas das tentativas). Esta trajetória faria com que a bola permanecesse colidindo apenas com as paredes e não atingisse mais os tijolos. Sem a destruição completa dos tijolos, o experimento não poderia acabar, já que o fator necessário para mudança de fase é a completa destruição dos tijolos que estão na tela.

Uma decisão de implementação foi disponibilizar duas fases (níveis) para o jogo, uma com 16 tijolos e outra com 48 tijolos para todos as técnicas implementadas, para uma melhor análise da técnica em função do número de objetos na tela. O resultado esperado é que com a diminuição do número de objetos na tela, diminua também o número de testes de detecção de colisão. Esta diminuição levaria a um ganho de desempenho, característica que permite uma análise comparativa com as outras técnicas.

3.3.3 O Space Invaders

O outro jogo escolhido para a realização dos experimentos foi o Space Invaders. Ele foi escolhido basicamente pelas mesmas características do jogo Breakout, porém com uma característica distinta deste, os objetos do Space Invaders são dinâmicos. Abaixo segue uma breve descrição de como funciona o jogo.

O jogador controla uma nave que fica localizada na parte inferior da tela e se movimenta para a esquerda ou para a direita. As naves inimigas ou NPCs (*Non-Player Characters*) ficam localizadas na parte superior e laçam *bombas* (daqui em diante adotaremos esta nomenclatura para designar o ataque de uma nave inimiga) na nave do jogador na tentativa de destruí-la. Além disso, as naves inimigas movimentam-se também na vertical em direção a nave do jogador, podendo levar a uma colisão e uma conseqüente destruição da nave do jogador. O objetivo do jogador é atacar (utilizaremos a nomenclatura *tiro* para designar o ataque da nave do jogador) as naves inimigas até destruí-las por completo, levando a uma mudança de fase. Após um determinado número de fases, o jogador chega ao chefe do estágio, onde terá que destruí-lo para avançar de estágio.

Como foi dito na seção 3.3.2, é necessário tirar todos as intervenções do usuário (jogador) para que o teste seja automatizado e passível de uma repetição fiel fazendo com os mesmos resultados sejam obtidos. Desta forma, foi necessário tirar o controle da nave por parte do jogador inserindo um controle automático do movimento horizontal da mesma, ou seja, a nave fica se movimentando de um lado para o outro automaticamente. A direção da nave é invertida ao colidir com as paredes laterais.

Adicionalmente a nave do jogador foi programada para realizar ataques dentro de um intervalo pré-determinado. Outras duas características relevantes das modificações feitas foram: a remoção do movimento vertical das naves inimigas e a remoção dos diversos níveis devido a sua complexidade e irrelevância para o escopo deste trabalho.

Assim como no Breakout, também foram implementados dois níveis diferentes, um com 10 naves inimigas e outro com 20, com o intuito de analisar o comportamento das técnicas em função do número de objetos.

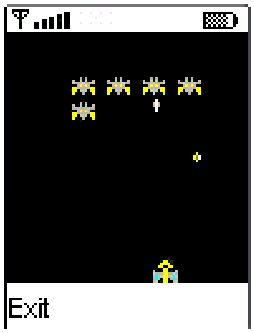


Figura 3.3: Tela do jogo Space Invaders.

3.4 Conclusão do capítulo

Com o intuito de prover uma base científica para o nosso trabalho, utilizamos uma área relativamente nova na Engenharia de Software, a Engenharia de Software Empírica. Utilizamos alguns estudos sobre metodologia de experimentação para dar um fundamento mais sólido aos experimentos e servir como um guia para trabalhos relacionados que necessitam de algum tipo de experimentação.

Devido à complexidade inerente do trabalho, decidimos escolher dois jogos para implementar diferentes técnicas de detecção de colisão. Os jogos escolhidos foram o *Breakout* e o *Space Invaders* por serem jogos de fácil implementação e bastante conhecidos. Além destas características em comum, estes dois jogos possuem uma característica distinta, o movimento dos objetos no decorrer do jogo.

4 Detecção de Colisão

Um problema fundamental na simulação de sistemas físicos é o problema de detecção de colisão que determina quando dois corpos entram em contato. O percentual do tempo computacional gasto com detecção de colisão para determinados tipos de sistemas que utilizam esta modelagem física é superior a 90% [11]. Muitos dos algoritmos de detecção de colisão usados atualmente foram desenvolvidos para *robot motion planning*, e posteriormente adotados para o uso em sistemas de Realidade Virtual (*VR*) e simuladores físicos [15]. Apesar da vasta aplicabilidade das técnicas de detecção de colisão, o objetivo deste trabalho é focar especificamente a área de jogos 2D, pois como falamos anteriormente, a capacidade de processamento atual da maioria dos dispositivos móveis não suporta o desenvolvimento de jogos 3D com um nível satisfatório mínimo de jogabilidade.

4.1 Detecção de Colisão em Jogos

O problema de detecção de colisão está presente desde os primórdios dos jogos eletrônicos. Onde tanto os mais simples jogos 2D (como o *Breakout*, o *Tetris*, o *Pinball*, entre outros) quanto os mais avançados jogos 3D existentes atualmente (como *Doom*, *Unreal Tournament*, *Counter Strike*, entre outros) precisam de algum tipo de verificação de colisão. Os jogos 3D usam complexos algoritmos de verificação de colisão baseados em polígonos, cuja complexidade de implementação é de magnitude superior à implementação de checagem de colisão para simples jogos 2D como os citados acima.

Com o avanço da capacidade computacional dos PCs, os jogos para computador também evoluíram. Com esta evolução os programadores passaram a adicionar mais e mais complexidade algorítmica para simular o mundo com maior precisão. Além do fator computacional, diversos jogadores passaram a não mais aceitar simulações grosseiras da realidade, a exemplo dos primeiros simuladores de vôo, onde um jogador conseguia voar através de um pico de uma montanha e atravessá-lo ileso, ou mesmo de jogos de ação, onde o jogador tinha a experiência de ser atingido por um projétil que sequer estava próximo a ele.

Principalmente devido a esta grande diferença computacional entre os PCs e os dispositivos móveis existentes atualmente, as técnicas de detecção de colisão 3D não fazem parte do escopo desta dissertação. Apesar de viáveis em termos de implementação, estas técnicas não são passíveis de utilização em jogos móveis, pois degradam bastante a performance desse tipo de aplicação, tornando-o inviável em termos de jogabilidade.

4.1.1 Tipos de Detecção de Colisão

Para um jogo em duas dimensões, há basicamente dois tipos de detecção de colisão, a detecção baseada em *pixel*, onde o princípio básico é a verificação de sobreposição dos *pixels* que formam as imagens dos objetos, e a detecção baseada em circunscrição (*bounding*), ou seja, os objetos são inscritos em elementos geométricos, como retângulos (*bounding box*, ver a Figura 4.1) ou círculos (*bounding sphere*), havendo colisão quando os dois elementos de circunscrição se intersectam.

As técnicas de detecção de colisão utilizadas neste trabalho são técnicas de eliminação de testes de colisão, que têm como objetivo diminuir o número de testes de colisão entre objetos do jogo. No entanto, quando dois objetos precisam ser testados para saber se há colisão entre eles, estas técnicas usam o teste de colisão entre os retângulos de circunscrição dos objetos.

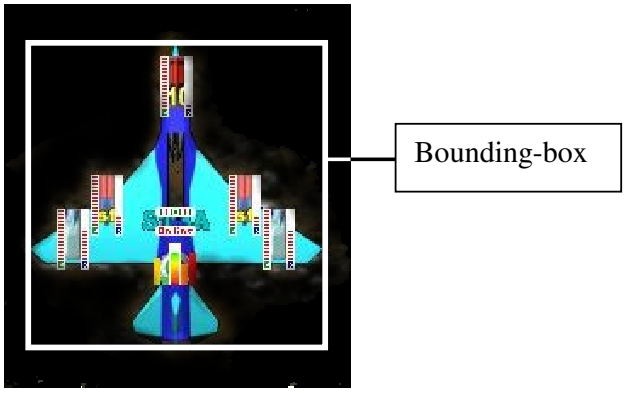


Figura 4.1: Exemplo de um retângulo de circunscrição de um objeto (*bounding Box*).

4.2 Alguns Conceitos sobre Jogos

Este tópico aborda alguns conceitos básicos sobre jogos para o bom entendimento deste trabalho.

Em jogos (ou em qualquer outra aplicação gráfica), a tela normalmente consiste de diferentes objetos, que podem ou não estar conectados entre si. Tipicamente uma

camada (*layer*) agrupa um conjunto de objetos. A quantidade de camadas utilizadas pode variar de um jogo para outro dependendo da necessidade. Por exemplo, um determinado desenvolvedor pode querer agrupar os objetos de um jogo específico em três camadas: uma para objetos considerados base (que dizem respeito ao pano de fundo do jogo), como a grama, lagos, montanhas, etc; Outra camada para objetos considerados intermediários, como árvores, rochas, mesas, entre outros; E uma última contendo objetos considerados de primeira instância, como espadas, pessoas, monstros, etc. Da mesma forma, um outro desenvolvedor pode querer utilizar apenas duas camadas, uma para objetos de plano de fundo e outra para objetos móveis, como pessoas, monstros, escadas, etc. Ou ainda um desenvolvedor pode querer utilizar uma camada para cada objeto da tela. Concluindo, as camadas são gerenciadas de forma independente.

Um *Sprite* é uma coleção de imagens 2D (quadros) mostrados em seqüência, com um atraso temporal entre um quadro e outro. *Sprites* são usados para representar diversos objetos do jogo, como pessoas se movendo, espaçonaves, cadeiras, mísseis, cursores de *mouse* animados, etc Figura 4.2. A seqüência de imagens abaixo mostra um exemplo de *sprite*. No decorrer do tempo, quando uma imagem é sobreposta pela seguinte na seqüência, o jogador tem a sensação visual de que a personagem está andando.

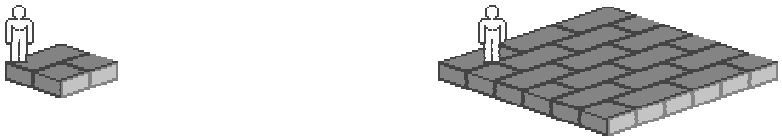
Figura 4.2: Seqüência de imagens que formam um *sprite* de um personagem andando.

Em alguns casos, *sprites* podem possuir características avançadas, como sofrer movimentos de rotação, ser criados semitranslúcidos, ser configurados para executar a animação em ordem reversa de quadros, entre diversas outras características.

Jogos baseados em *ladrilhos* [39][42][43] são simplesmente jogos construídos sobre um determinado tipo de grade (*grid*) com personagens se movimentando de ladrilho em ladrilho. Cada ladrilho representa um determinado tipo de terreno permitindo a construção de um mundo onde o jogo se passa. Os ladrilhos podem estar agrupados em camadas (*layered tiles*), começando por grama e terra que cobre a janela do jogo, passando por árvores e construções que são sobrepostos ao que foi desenhado anteriormente.

Ladrilhos podem ter outras propriedades além do tipo de terreno que eles representam. Por exemplo, ladrilhos de transição podem mudar o tipo de terreno durante o jogo como resposta a certos eventos. Jogadores ou NPCs podem se movimentar com diferentes velocidades dependendo do tipo de terreno do ladrilho. Como resposta a entrada do jogador dentro de uma construção, a camada de ladrilhos que representa o telhado pode desaparecer enquanto o jogador estiver dentro da construção, entre outras características.

Os jogos baseados em ladrilhos são utilizados em dois tipos de jogos, os 2D Figura 4.4 e os chamados jogos isométricos [41] Figura 4.3. Projeções isométricas são, superficialmente falando, uma falsa representação 3D construída usando-se projeções ortográficas, ou projeções em perspectiva.



(a) (b)

Figura 4.3: (a) Exemplo de um ladrilho isométrico. (b) Exemplo de terreno construído a partir de vários ladrilhos.

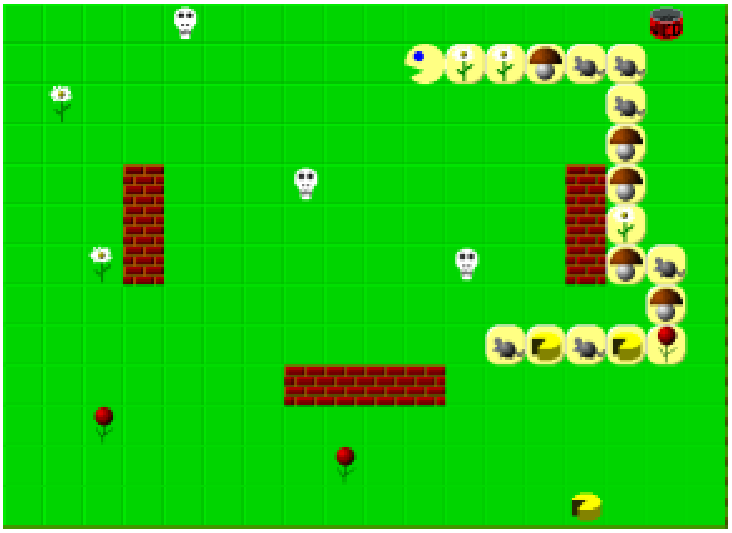


Figura 4.4: O Jogo *Snake* baseado em ladrilhos. Cada quadrado é um ladrilho que pode representar diferentes objetos do jogo, como o queijo, a parede, o corpo da cobra, etc.

Os tipos de jogos citados são baseados unicamente em ladrilhos, onde todo o plano de fundo do jogo é definido como ladrilhos, bem como as personagens do jogo, cujo movimento se dá de ladrilho em ladrilho. Há, no entanto, jogos híbridos, onde o plano de fundo do jogo é definido como ladrilhos, mas as personagens do jogo possuem

um movimento livre de acordo com a resolução da tela. Na outra extremidade existem os jogos sem ladrilhos (ou textura de fundo), onde as personagens se movimentam livremente segundo a resolução da tela, e onde o plano de fundo também é definido em função da resolução.

Um *mapa de ladrilhos* [40] é a representação do mundo (objetos e plano de fundo) para jogos baseados em ladrilhos. Tipicamente ele é uma matriz, onde cada célula da matriz corresponde a um ladrilho. Cada objeto do jogo tem um (ou mais de um) valor associado que representa este objeto no mapa. Por exemplo, uma espada pode ter um valor 1 associado a ela. Isto implica dizer que todo valor 1 dentro da matriz representa uma espada e nenhum outro tipo de objeto. No caso do plano de fundo, as diferentes regiões podem ser representadas por diferentes valores. Por exemplo, o céu pode ser representado pelo valor 2, a grama pelo valor 3, a montanha pelo valor 4, e assim por diante.

4.3 Técnicas de Detecção de Colisão

Dadas as limitações impostas pelo MIDP 1.0, não foram levadas em consideração algumas técnicas de detecção de colisão. Esta API do MIDP não especifica nenhuma função que lê os *pixels* de uma imagem. Isto inviabiliza o uso de técnicas de detecção baseadas em *pixel*, cuja precisão é bem maior que as técnicas baseadas em retângulo circunscrito. No entanto, a técnica de leitura de *pixel* é mais lenta devido a uma necessidade de maior processamento na verificação de colisão (ver seção 4.1.1).

A seguir são descritas todas as técnicas de detecção de colisão implementadas, tanto no Breakout quanto no Space Invaders, e utilizadas nos experimentos.

4.3.1 Detecção Padrão (*Object-to-objcet*)

Esta técnica de detecção de colisão utiliza um algoritmo simples de verificação de colisão baseado em retângulo circunscrito [10]. Isto é, cada objeto ou *sprite* (bola, tijolo) do jogo é representado, em termos de implementação, por um retângulo que o envolve. O algoritmo consiste na verificação de sobreposição dos retângulos dos dois objetos em questão. Esta técnica simplesmente leva em consideração apenas as eliminações possíveis de acordo com as regras do jogo. Exemplificando, no caso do Breakout, não é necessário fazer testes de colisão entre os tijolos do jogo, apenas entre a bola e os tijolos. No caso do Space Invaders, não é necessário fazer testes de colisão

```
boolean intersects(Rect rect1, Rect rect2) {
    if ((rect1.getXPos() + rect1.getWidth()) < rect2.getXPos())
        return false;
    if (rect1.getXPos() > (rect2.getXPos() + rect2.getWidth()))
        return false;
    if ((rect1.getYPos() + rect1.getHeight()) < rect2.getYPos())
        return false;
    if (rect1.getYPos() > (rect2.getYPos() + rect2.getHeight()))
        return false;
    return true;
}
```

entre as naves inimigas e a nave amiga (do jogador). Segue o algoritmo genérico de teste entre dois retângulos de circunscrição:

A Figura abaixo ilustra a interseção entre dois retângulos de circunscrição.

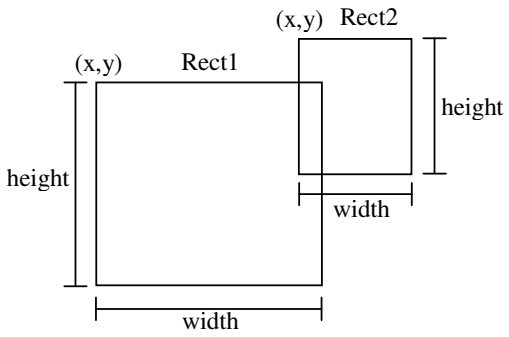


Figura 4.5: Ilustração da interseção entre dois retângulos de circunscrição.

`Rect` é o objeto que representa o retângulo circunscrito. No caso do BreakOut, a cada ciclo a bola é comparada com todos os tijolos para verificar se a mesma colidiu com algum tijolo. Em caso positivo o tijolo desaparece (é destruído) e a bola modifica sua trajetória.

No caso do Space Invaders, a cada ciclo verifica-se se houve colisão entre todas as bombas e a nave do jogador. Neste caso, ocorre apenas a verificação de colisão, onde a única conseqüência é a destruição da bomba que colidiu (no jogo original a nave do jogador explodiria e perderia uma vida, característica que foi retirada da implementação dos experimentos, pelas razões discutidas). Verifica-se também a colisão entre todos os tiros com todas as naves inimigas. Neste caso, havendo colisão, esta nave inimiga é destruída e o tiro desaparece da tela.

O potencial desta técnica é razoavelmente limitado, pois como foi mencionado antes, ela se limita a eliminar testes de colisão baseado apenas nas regras do jogo. Desta forma, considerando que todas as técnicas de detecção de colisão devem eliminar os

testes de colisão com base nas regras do jogo, esta técnica não apresenta nenhum tipo efetivo de eliminação de teste de colisão.

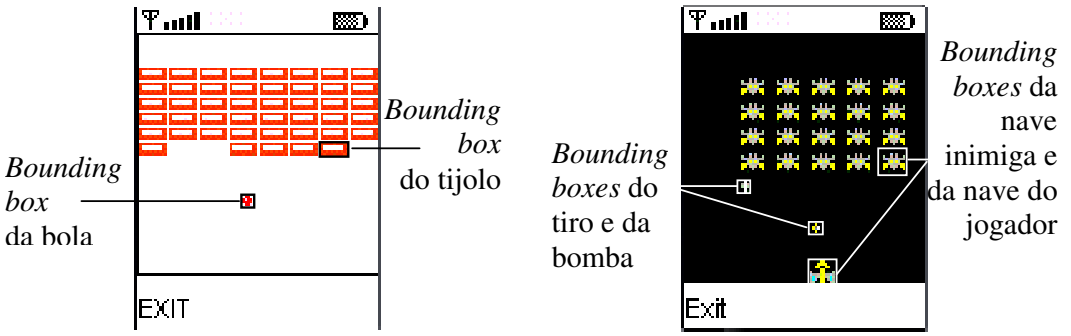


Figura 4.6: Verificação de colisão padrão. A cada iteração do jogo, verifica-se todas as possibilidades de colisão. Não há eliminação de testes.

4.3.2 Região de Influência

Esta técnica baseia-se numa característica específica de jogos que possuem os objetos do jogo sempre dispostos numa mesma região da tela, não havendo deslocamento destes objetos fora da área referida. Esta área foi denominada de *Região de Influência*. Esta é uma técnica que foi sugerida por nós levando-se em consideração a característica citada.

No caso do Breakout, ela leva em consideração que os tijolos estão dispostos sempre numa mesma região da tela não havendo deslocamento dos mesmos, ou seja, a verificação de colisão não precisa ser efetuada se a bola não estiver dentro dessa área. Desta forma, esta técnica visa otimizar a detecção padrão, eliminando os testes de colisão entre a bola e os tijolos quando a bola não estiver dentro dessa região de influência. Como a região de influência é um retângulo, para verificar se a bola está nessa área, basta testar se ela entrou total ou parcialmente usando-se o mesmo algoritmo da seção anterior. A Figura 4.7 ilustra a técnica.

O caso do Space Invaders é um pouco diferente. As naves inimigas se movimentam por toda a tela no jogo original. No entanto, como o movimento vertical das naves foi retirado, para o escopo dos experimentos, elas passaram a se movimentar apenas na horizontal de um lado para o outro. Isto permitiu que a técnica fosse implementada para este jogo, fazendo da região onde as naves se movimentam a região de influência. A Figura 4.8 ilustra o que foi descrito.

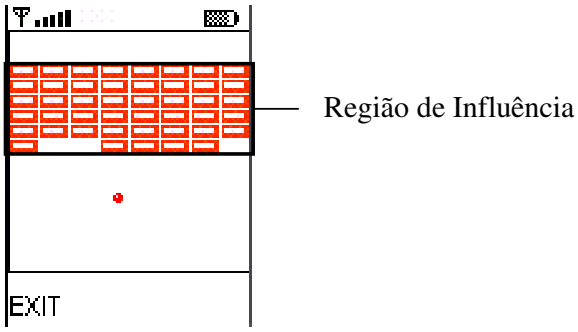


Figura 4.7: Primeiro se verifica se a bola entrou na região de influência. Em caso positivo, a detecção de colisão é aplicada entre a bola e todos os tijolos.

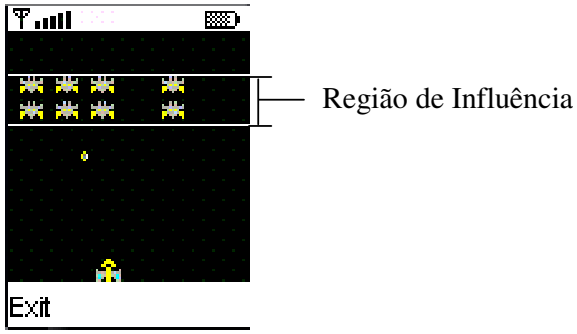


Figura 4.8: Implementação da técnica de região de influência para o Space Invaders.

Esta técnica permite um ganho de performance considerável em relação à técnica de detecção padrão, pois os testes de colisão são feitos entre a região de influência e o objeto de interesse (ou seja, apenas um teste de colisão a cada iteração do jogo). Os testes de colisão entre o dado objeto de interesse (objeto que está sendo testado) e outros objetos do jogo só são realizados se este objeto estiver dentro da região de influência.

4.3.3 Ordenação por Eixo (Axis Sorting)

Quando vários objetos estão na tela de uma vez, dificilmente eles estão no mesmo local. Esta técnica de detecção de colisão se propõe a eliminar testes de colisão ordenando os objetos numa lista, a cada iteração do jogo, de forma ascendente segundo as coordenadas do eixo x ou y dos objetos [10]. É preciso ter cuidado com a escolha do algoritmo de ordenação, pois dependendo da escolha, isto pode deteriorar bastante o tempo total gasto na detecção de colisão. Por exemplo, se o *quicksort* for escolhido e os objetos a serem ordenados já estiverem praticamente ordenados, o mesmo pode mostrar o seu pior caso, n^2, dependendo da implementação do *quicksort*.

Para checar por colisões, deve-se simplesmente testar um dado objeto com os objetos subseqüentes na lista. O algoritmo deve parar de testar quando um dos objetos

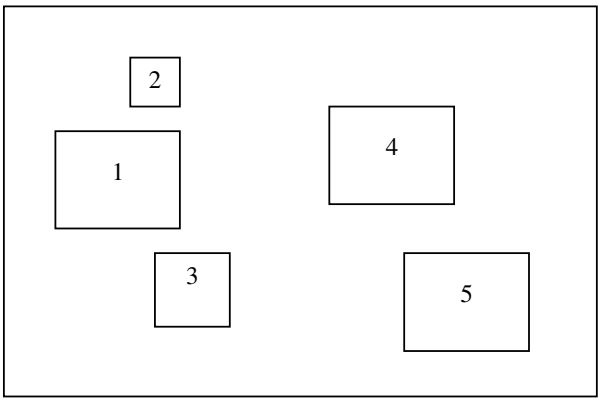
da lista possuir uma coordenada x (ou y) (dependendo do eixo escolhido para a ordenação) maior do que a coordenada x (ou y) do primeiro objeto da comparação mais sua largura, no caso do eixo x, ou sua altura, no caso do eixo y. O pseudo-código abaixo ilustra o algoritmo:

```
Entrada: SpriteList (A lista dos objetos do jogo ordenada pelo eixo x)
Saída: void
AxisSortingCollisionTest
begin
    Sprite s1, s2;

    s1 := SpriteList.head;
    while (s1 != ListEnd)
    begin
        s1.Right := s1.Left + s1.Width;
        /* Compara s1 com os próximos elementos da lista até que */
        /* a extremidade esquerda de um desses objetos seja maior */
        /* do que a extremidade direita de s1 */

        s2 := s1.Next;
        while (s2 != ListEnd AND (s1.Right > s2.Left))
        begin
            CheckIntersection(s1, s2);
            s2 := s2.Next;
        end
        s1 := s1.Next;
    end
end
```

Por exemplo: supondo que se tenha cinco objetos ordenados por suas coordenadas do eixo x. Primeiro o algoritmo testaria o objeto #1 com o objeto #2, depois com o objeto #3. No objeto #4, o teste deveria parar porque o objeto #4 está localizado na coordenada de valor 40 e a extremidade direita do objeto #1 está localizada na coordenada de valor 20. Dado que o objeto após o objeto #4 na lista tem uma coordenada x maior do que a do objeto #4, não há necessidade de checar colisão com o restante dos objetos da lista. A Figura 4.9 ilustra o exemplo dado.



Número	Coord. x	Largura
1	10	10
2	15	5
3	18	8
4	40	10
5	45	10

Figura 4.9: Objetos ordenados pelo eixo x.

No caso do Breakout, a técnica é implementada levando-se em consideração os dois eixos, ou seja, ela é implementada ordenando os objetos pelo eixo x e também pelo eixo y, avaliando ambos os resultados. Além da variação dos eixos, esta técnica também foi combinada com a técnica de região de influência. Isto é, ela é implementada em quatro variações:

- uma indexando pelo eixo x e não considerando a região de influência;
- a segunda, indexando pelo eixo x e levando em consideração a região de influência;
- a terceira, indexando pelo eixo y e não levando em consideração a região de influência;
- e a quarta, levando em consideração a região de influência e indexando pelo eixo y.

A Tabela 4.1 mostra as variações da técnica.

Técnica	Eixo de ordenação	Região de influência
1	X	Sim
2	Y	Não
3	X	Sim
4	Y	Não

Tabela 4.1: Variações implementadas da técnica de ordenação por eixo.

A Figura 4.10 mostra um exemplo da técnica não levando em consideração a região de influência e com indexação pelo eixo x.

As mesmas variações implementadas para o Breakout também foram implementadas para o Space Invaders. No caso deste jogo, como pode haver vários tiros ao mesmo tempo na tela, verifica-se a colisão para cada tiro separadamente. Para cada tiro é feito o teste de colisão entre este e cada nave inimiga da lista, até encontrar uma nave que possua a coordenada x maior que a coordenada x do tiro mais sua largura, ou que se chegue ao fim da lista. Esse procedimento é executado isoladamente para cada tiro até esgotar as possibilidades. A Figura 4.11 ilustra o mencionado.

No caso da nave do jogador, como há apenas uma nave, a detecção de colisão continua sendo feita diretamente, ou seja, verifica-se, para cada bomba, se a mesma colidiu com a nave do jogador.

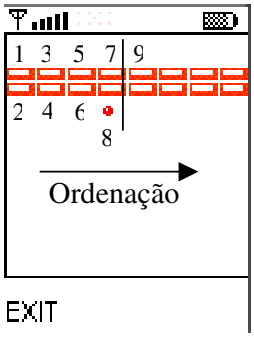


Figura 4.10: Indexando pelo eixo x. A partir do 9º tijolo não é preciso mais verificar colisão, pois os tijolos possuem uma coordenada x maior do que a coordenada x da bola mais sua largura. (Ordenação pelo eixo x).

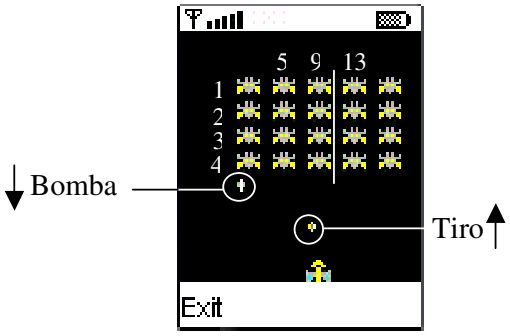


Figura 4.11: A partir da 13º nave inimiga não é mais necessário testar colisão, pois as naves possuem uma coordenada x maior que a coordenada x mais a largura do tiro. (Ordenação pelo eixo x).

A técnica de ordenação por eixo tem um grande potencial quando existem muitos objetos na tela. Deve-se levar em consideração a distribuição dos objetos em relação aos eixos para saber qual dos dois eixos deve ser levado em consideração no momento da implementação. Implementamos a técnica para os dois eixos para uma avaliação comparativa entre eles.

4.3.4 Gerenciador de Objetos

Esta técnica de detecção de colisão se propõe a eliminar colisões utilizando gerenciadores que agrupam objetos, fazendo uso de algum critério [18]. Desta forma, a verificação de colisão é feita entre um dado objeto e um gerenciador. Caso o gerenciador identifique que há possibilidade de colisão entre o dado objeto e um dos objetos do gerenciador, então o nível de comparação passa a ser entre o dado objeto e cada objeto do gerenciador. A técnica se trata de uma extensão da idéia de retângulos circunscritos, onde um gerenciador nada mais é que um retângulo que engloba um determinado grupo de objetos.

Não implementamos esta técnica para o jogo Space Invaders porque as naves possuem movimento. Para cada iteração do jogo seria necessário atualizar o gerenciador.

Como há diversas possibilidades de combinação de número de objetos com número de gerenciadores, utilizamos o mesmo algoritmo para duas combinações:

- 4 gerenciadores por 4 objetos (Figura 4.12 (a)) e 2 gerenciadores por 8 objetos (Figura 4.12 (b)) para o nível 1;
- 12 gerenciadores por 4 objetos (Figura 4.12 (c)) e 6 gerenciadores por 8 objetos (Figura 4.12 (d)) para o nível 2.

Neste caso, os limites que agrupam os objetos (tijolos) de um dado gerenciador são estabelecidos por um retângulo circunscrito. Desta forma, quando a verificação de colisão é feita, utiliza-se o algoritmo de detecção de colisão de retângulos entre a bola e cada um dos gerenciadores. Se houver interseção, a verificação então é feita entre a bola e cada um dos tijolos que fazem parte do gerenciador para saber se houve colisão.

Exemplificando a técnica, na Figura 4.12 (a) há 4 gerenciadores (G1, G2, G3 e G4) por 4 tijolos para o nível 1. No momento de se verificar a detecção de colisão, a bola é testada (usando-se o algoritmo de verificação de interseção descrito na seção 4.3.1) com cada um dos 4 gerenciadores. Se houver interseção, faz-se a verificação de colisão entre a bola e cada um dos tijolos que fazem parte do gerenciador em questão. A idéia é exatamente a mesma para as outras variações de número de gerenciadores por número de tijolos.

Esta técnica pode ser utilizada quando os objetos são estáticos e distribuídos de tal forma que permita o agrupamento dos objetos, como mostrado no Breakout. Se os objetos forem distribuídos de forma muito irregular, o objeto de interesse (objeto que está sendo testado) pode estar intersectando todos os gerenciadores (ou grande parte deles), degradando a performance da técnica. Isto porque os testes seriam feitos para todos os objetos (ou para a grande maioria) convergindo para a técnica padrão. Portanto, o potencial desta técnica é limitado devido a grande dependência do tipo de jogo e da distribuição dos objetos do mesmo.

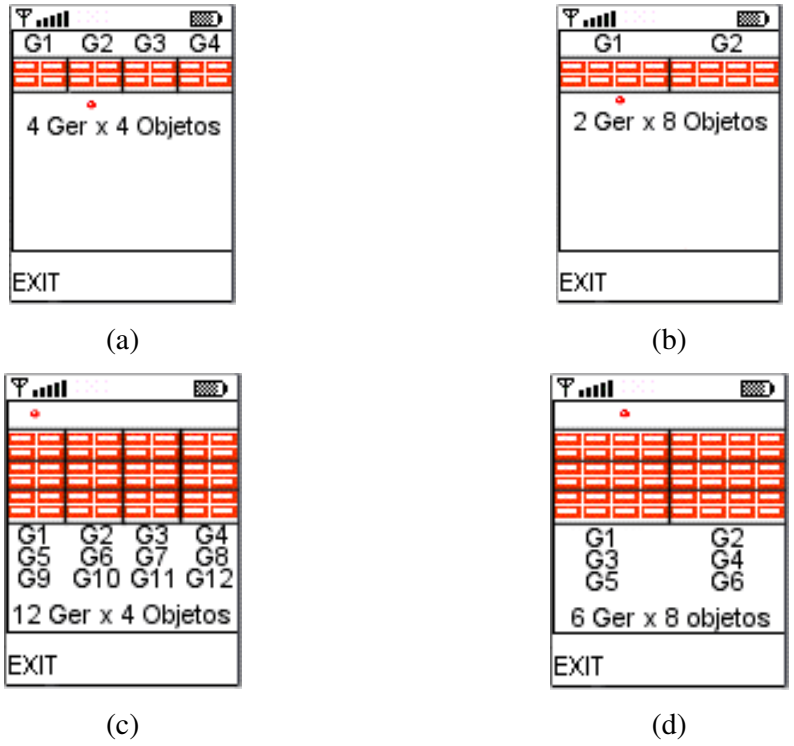


Figura 4.12: Técnica de gerenciadores de objetos. (a) 4 gerenciadores por 4 objetos para o nível 1; (b) 2 gerenciadores por 8 objetos para o nível 1; (c) 12 gerenciadores por 4 objetos para o nível 2; (d) 6 gerenciadores por 8 objetos para o nível 2.

4.3.5 Detecção pelo Método do Setor

O método do setor divide a área do jogo em setores [10]. No caso mais simples, a área é dividida em quadrantes. Cada objeto é adicionado a um ou mais quadrantes dependendo de suas coordenadas na tela. Em seguida, executa-se o teste de colisão para todos os objetos dentro de um dado setor, para todos os setores. Assumindo que os objetos estão bem distribuídos por toda a tela, a maioria dos setores irá conter apenas alguns objetos ou mesmo nenhum, diminuindo o número de testes de colisão.

Assumindo que o tamanho dos setores é maior que o maior objeto passível de verificação de colisão, um dado objeto pode estar contido em até quatro setores no pior caso (ver Figura 4.13), quando suas coordenadas estiverem presentes em quatro setores diferentes. Este método é implementado apenas para o Space Invaders, já que o Breakout possui os objetos estáticos.

Esta técnica possui a limitação dos objetos precisarem estar esparsamente distribuídos. Caso os objetos estejam agrupados próximos em um mesmo local, a performance da técnica pode ser comprometida. Além disso, o custo computacional de se atualizar os objetos em função dos setores pode ser maior que o benefício da

eliminação dos testes de colisão. Principalmente em se tratando de dispositivos móveis, com tela reduzida e cujos jogos geralmente não possuem tantos objetos presentes ao mesmo tempo na tela.

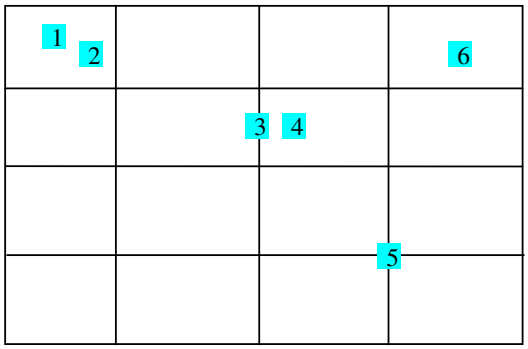


Figura 4.13: Os objetos 1 e 2 estão presentes num mesmo setor e devem ser testados entre si. Assim como os objetos 3 e 4, onde o objeto 3 está presente em dois setores. Objeto 5 faz parte de 4 setores.

4.3.6 Detecção de Colisão Baseada em Ladrilhos

Um jogo baseado em ladrilhos (*tiles*) é um jogo cuja superfície em que se joga (o mundo) é representado por pequenas seções chamadas ladrilho [19]. Para maiores informações sobre algumas características básicas de jogos, ver a seção 4.2.

No caso da detecção de colisão baseada em ladrilhos [44][45], não somente a forma de verificação de colisão é diferenciada, como também o jogo precisa ser reestruturado para suportar os ladrilhos. Isto é, a lógica do jogo precisa ser estruturada para trabalhar com mapa de ladrilhos, onde os objetos do jogo são representados por valores nos ladrilhos. O jogo deve passar a trabalhar com ladrilhos como unidade de deslocamento, ao invés de *pixels* como em um jogo não baseado em ladrilhos. A conversão da posição do ladrilho em *pixels* é feita apenas no momento da atualização da tela. Resumindo, a lógica que representa o mundo do jogo é diferente da dos jogos não baseados em ladrilhos.

Nesta técnica, um dado objeto que ocupa um determinado ladrilho não precisa ser testado com todos os outros objetos presentes na área do jogo, mas apenas com os objetos que ocupam ladrilhos próximos a ele, eliminando bastante a quantidade de testes de colisão.

Para o Breakout esta técnica é implementada em duas versões:

- uma versão onde o ladrilho tem o tamanho da bola, ou seja, um tijolo ocupa mais de um ladrilho.

- e outra cujo ladrilho possui o tamanho de um tijolo, isto é, cada tijolo ocupa um ladrilho.

Para a primeira versão, o jogo é completamente baseado em ladrilhos, ou seja, a bola possui um deslocamento discreto movimentando-se de ladrilho em ladrilho. Desta forma, para se verificar a colisão, basta testar se a bola, após o movimento da iteração corrente, atinge um ladrilho que não estava vazio, ou seja, possui um tijolo (ver Figura 4.15). Havendo a colisão, a bola sofre reflexão em sua trajetória e o tijolo que sofreu colisão desaparece.

Já no caso da segunda versão, o jogo é dividido em quadrantes do tamanho de um tijolo, formando o mapa do jogo (ver Figura 4.14 (a)). Os dois únicos tipos de ladrilhos do mapa são: o ladrilho com tijolo e o ladrilho vazio. A bola é um objeto dinâmico que se movimenta dentro dos ladrilhos, já que a mesma possui dimensões menores que as dimensões de um ladrilho. Neste caso, as coordenadas da bola em relação ao ladrilho em que ela está presente são atualizadas a cada iteração do jogo. No momento de verificação de colisão, testa-se se há colisão entre a bola e um dos oito tijolos (ladrilhos) que fazem fronteiras com o ladrilho que a bola está presente. Além disso, como a bola possui um deslocamento contínuo dentro do tijolo (ver Figura 4.14 (b)), a mesma precisa alcançar um dos quatro lados do tijolo para haver possibilidade de colisão. Esta versão não é 100% baseada em ladrilhos, já que a bola não faz parte do mapa do jogo, ela é apenas um objeto com deslocamento contínuo.

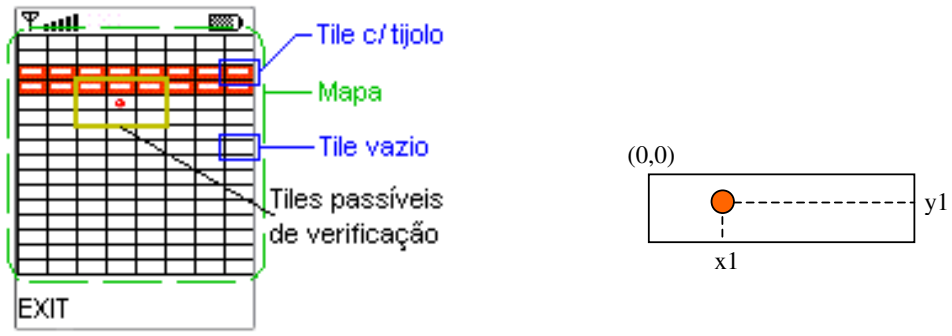


Figura 4.14: (a) Representação gráfica do mapa do jogo; (b) Representação do deslocamento contínuo dentro de um ladrilho vazio. (x1, y1) é o ponto relativo atual de deslocamento dentro do ladrilho.

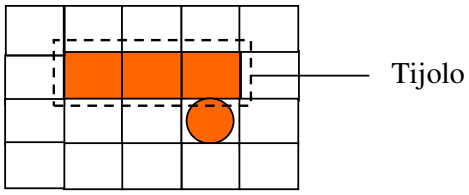


Figura 4.15: Supondo que a bola esteja com movimento vertical ascendente. Na próxima atualização, a bola colidirá com o tijolo, pois a mesma passaria a ocupar um ladrilho ocupado por um tijolo.

Para o Space Invaders, o jogo é implementado com o ladrilho do tamanho do menor objeto, que no caso são a bomba e o tiro (a Figura 4.16 ilustra o jogo). O movimento dos objetos do jogo é feito de ladrilho em ladrilho.

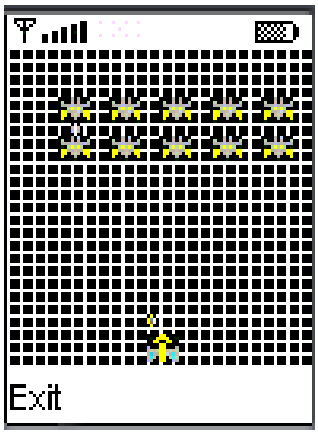


Figura 4.16: O jogo Space Invaders implementado em ladrilhos. O fundo do jogo mostra o mapa de ladrilhos, onde estes ladrilhos possuem o tamanho das bombas (e tiros). Ou seja, as naves inimigas e do jogador ocupam vários ladrilhos.

A técnica baseada em ladrilhos tem um grande potencial, uma vez que os testes de colisão são bem simples de implementar. Isto é, a maior dificuldade está na atualização dos objetos do jogo, tornando simples a identificação de colisão. No entanto, esta técnica, por estar intrinsecamente ligada ao tipo de jogo, só deve ser usada em jogos que podem ser naturalmente estruturados em ladrilhos. Por exemplo, o Space Invaders não é um jogo que pode ser naturalmente estruturado em ladrilhos. Isto é, o esforço de implementação, os resultados de performance e a qualidade final do jogo podem não justificar o desenvolvimento deste jogo baseado em ladrilhos em relação a outras técnicas de detecção de colisão.

4.4 Conclusão do Capítulo

A detecção de colisão desempenha um papel fundamental no desenvolvimento de jogos hoje em dia. Com a evolução da capacidade computacional dos PCs, os jogadores

passaram a exigir um nível muito maior na simulação dos jogos. Para se atingir este nível de exigência, a escolha do algoritmo de detecção de colisão deve ser feita de tal forma que permita um bom desempenho para não prejudicar a jogabilidade e também possua uma boa precisão para não desapontar os usuários.

Como o objetivo é disponibilizar um estudo sobre detecção de colisão para dispositivos móveis, não levamos em consideração algumas técnicas de detecção de colisão, como as técnicas 3D e as que se baseiam em leitura de *pixel*, pela deficiência computacional e da plataforma de desenvolvimento para estes dispositivos. O estudo foi focado nas seguintes técnicas, por serem bastante conhecidas e usadas quando o assunto é detecção de colisão 2D:

- Detecção Padrão (*Object-to-objcet*)
- Região de Influência
- Ordenação por Eixo (Axis Sorting)
- Gerenciador de Objetos
- Detecção pelo Método do Setor
- Detecção de Colisão Baseada em Ladrilhos

Também foram levadas em consideração técnicas híbridas compostas pelas técnicas citadas, como a técnica de ordenação por eixo com e sem região de influência. Cada técnica é explicada em detalhes para que se suas implementações possam ser posteriormente entendidas. Estas técnicas foram implementadas para os dois jogos levados em consideração pelo experimento, o *Breakout* e o *Sapce Invaders*.

5 Implementação e Resultados

Os tópicos a seguir falam sobre detalhes da implementação das técnicas de detecção de colisão para os dois jogos escolhidos. É exposta uma visão geral da estrutura dos jogos e do suporte às técnicas de detecção de colisão, objetivando maximizar o código comum. Isto permite, como foi mencionado anteriormente, uma comparação justa entre as diferentes técnicas. Além dos detalhes de implementação, são mostradas as configurações necessárias para a reprodução do experimento, de tal forma que outras pessoas que tenham acesso a esta dissertação possam chegar aos mesmos resultados obtidos. Uma análise sobre os resultados obtidos é também realizada.

5.1 Estudos de caso

5.1.1 Implementação do Breakout

Para o jogo Breakout, as seguintes técnicas de detecção de colisão (e variações) foram implementadas:

- Detecção padrão (*Object to object*);
- Região de influência;
- Ordenação pelo eixo x sem região de influência;
- Ordenação pelo eixo x com região de influência;
- Ordenação pelo eixo y sem região de influência;
- Ordenação pelo eixo y com região de influência;
- Gerenciador de objetos (com quatro objetos por gerenciador);
- Gerenciador de objetos (com oito objetos por gerenciador);
- Baseada em ladrilhos (ladrilho de 6x12 [altura x largura]);
- Baseada em ladrilhos (ladrilho de 6x4 [altura x largura]).

Como foi explicado na seção 3, com base nos estudos sobre experimentação, apenas a variável de interesse (a técnica de detecção de colisão) deve ser responsável por variações nos resultados do experimento, eliminando qualquer outro fator que também possa levar a uma variação nos resultados. O código comum entre as implementações das técnicas é um fator bastante importante e necessário para se atingir este objetivo, pois qualquer variação de código que não tenha como meta a implementação da técnica pode levar a uma diferença nos resultados, que não refletiria

uma variação (desses resultados) em função da variável em questão. Isto é, qualquer variação no código que não tenha a ver com a implementação de uma determinada técnica pode comprometer os resultados do experimento. Como foi mencionado na seção 3.3.1, inicialmente o padrão *Strategy* foi utilizado para a implementação de todas as técnicas em apenas um MIDlet, isso facilitava bastante a questão do código comum, pois era necessário apenas implementar diferentes classes para as técnicas de detecção, onde o resto do código permanecia o mesmo. No entanto, esta abordagem não permitia identificar, ou pelo menos dificultava a medição, da variável *tamanho de código*. Desta forma, cada técnica foi implementada em um MIDlet separado. Para se fazer isto, é necessário se ter cuidado na replicação do código, para evitar que qualquer mudança possa levar a uma inconsistência dos resultados.

Na Figura 5.1 pode-se observar o diagrama de classes da implementação das técnicas de detecção de colisão para o *Breakout*. O `BreakOutMIDlet` é a classe que estende da classe básica MIDlet do MIDP, que é o ponto de partida de execução da aplicação. Uma das principais funções desta classe é a execução das iterações do teste, ou seja, ela possui o laço principal de execução da aplicação. A classe `BreakOutCanvas` é responsável, principalmente, pelo gerenciamento da parte gráfica da aplicação. Ela estende da classe abstrata `Canvas` do MIDP e implementa a função `paint` dessa classe abstrata, que é responsável pela atualização da tela a cada iteração da aplicação. Este método basicamente chama a função de atualização da tela, `drawWorld`, definida na interface `WorldInterface` que é específica à implementação de cada técnica de detecção de colisão. Isto é, cada técnica possui a liberdade de implementar o modo como a tela deve ser atualizada em função de sua implementação do "mundo" do jogo, refletida na classe `BreakOutWorld`. Na verdade, apenas a técnica baseada em ladrilhos possui sua implementação diferente das demais. Isto é, para todas as outras técnicas de detecção de colisão, a implementação do método de atualização da tela é a mesma. A classe `BreakOutWorld` é a implementação do "mundo" do jogo que implementa a interface `WorldInterface`, que possui os seguintes métodos:

- `public void loadWorld(int level);`
- `public void updateWorld();`
- `public void drawWorld(Graphics g);`
- `public boolean isGameOver().`

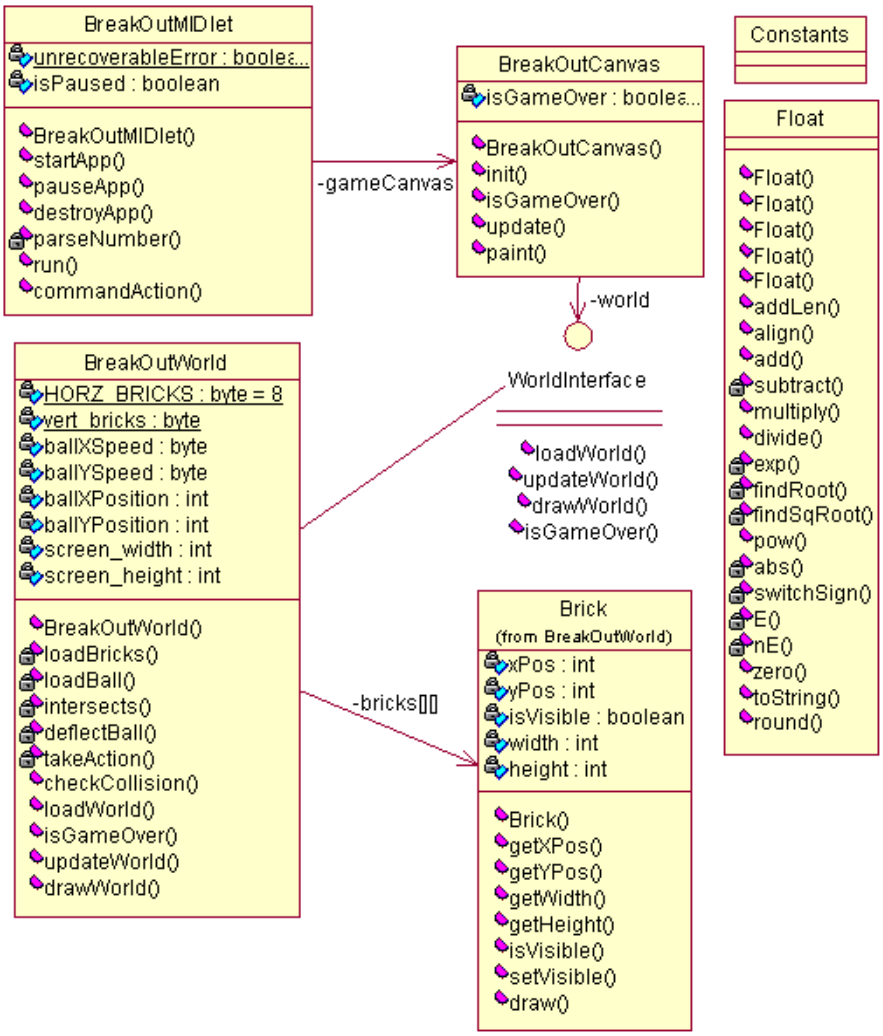


Figura 5.1: Diagrama de classes da implementação das técnicas de detecção de colisão para o Breakout.

O método `loadWorld` é responsável por carregar os objetos que fazem parte do jogo para os dois níveis de cada iteração. Desta forma, este método é executado duas vezes por iteração. O método `updateWorld` é responsável por atualizar as coordenadas dos objetos do jogo. Após a chamada deste método é feita a verificação de colisão entre os objetos usando o método `checkCollision`. Uma vez que os objetos tenham suas novas coordenadas definidas, o método `drawWorld` é chamado para a atualização da tela. O método `isGameOver` é chamado para saber se o nível de uma iteração terminou, ou seja, ele verifica se todos os tijolos da tela foram destruídos, em caso positivo, passa-se para o segundo nível da iteração ou inicia-se uma nova iteração. A classe `Brick` modela um tijolo e contém informações como a imagem associada ao

objeto, a largura e altura, a posição relativa do tijolo e o estado de visibilidade do objeto. Um `BreakOutWorld` possui o conjunto de tijolos que formam um nível.

Além das classes mencionadas acima, outras duas classes fazem parte da implementação do jogo, a classe `Constants`, que possui constantes e variáveis estáticas de uso global pela aplicação, e a classe `Float`, que é utilizada pelo `BreakOutMIDlet` para realizar os cálculos da média do número de quadros por segundo entre as iterações, já que a API do MIDP não disponibiliza manipulação de tipos básicos com ponto flutuante. O número de quadros por segundo é calculado pela aplicação contando-se o número de quadros entre uma colisão e outra e dividindo-se o valor encontrado pelo tempo gasto entre as colisões. Desta forma, é possível obter o número de quadros por segundo para um determinado número de tijolos na tela. Isto é, obtém-se a performance em *fps* por número de tijolos, identificando assim a variação de performance em termos do número de objetos (em termos do número de verificações de colisão feitas). Quanto menor o número de objetos na tela, menor é o número de detecções realizadas.

A classe `BreakOutWorld` possui todo o código relevante à implementação das técnicas, ou seja, nas outras classes não há modificações, permanecendo o mesmo código para todos os MIDlets. Na verdade, para todas as implementações das técnicas, exceto a da baseada em ladrilhos, apenas o código que verifica a colisão (cuja implementação encontra-se no método `checkCollision`) é que sofre modificação entre as implementações. Desta forma, seria possível restringir ainda mais a variação do código entre as implementações a apenas esse método. Entretanto essa abordagem exclui a implementação da técnica baseada em ladrilhos. Como o objetivo é propiciar uma porção de código que seja comum a todas as implementações, é necessário manter o limite entre o código comum e código específico de cada técnica no nível de implementação do mundo do jogo, dado que a técnica baseada em ladrilhos possui uma forma distinta intrínseca de implementação. Não obstante, o código da implementação do mundo do jogo para as outras técnicas permanece o mesmo, possibilitando-se fazer, caso haja interesse, uma comparação apenas entre as técnicas não baseadas em ladrilhos em função apenas do método específico de detecção de colisão. O apêndice A mostra as implementações dos métodos de detecção de colisão para as diversas técnicas implementadas no Breakout.

A implementação do Breakout disponibiliza uma variável de configuração no arquivo JAD (ver seção 2.2). A variável "NUM_ITERATIONS" permite a quem estiver executando os testes configurar de forma fácil (isto é, sem necessitar recompilar o código) o número de iterações que o MIDlet precisa executar. O valor que esta variável pode assumir é um valor inteiro positivo não nulo.

Como foi dito na seção 3.3.2, o Breakout foi reestruturado para retirar as características de interação com o usuário. Sendo necessário para que o experimento pudesse ser repetido outras vezes gerando os mesmos resultados inicialmente obtidos. A única característica presente no jogo era a raquete que o jogador usava para fazer a bola refletir em direção aos tijolos. Esta raquete foi retirada e a bola, ao atingir a parte inferior da tela, passou ser refletida automaticamente, da mesma forma como ela é refletida ao se chocar com as paredes laterais ou a parte superior da tela.

5.1.2 Implementação do Space Invaders

Para o Space Invaders, as seguintes técnicas de detecção de colisão (e variações) foram implementadas:

- Detecção padrão (*Object to object*);
- Região de influência;
- Ordenação pelo eixo x sem região de influência;
- Ordenação pelo eixo x com região de influência;
- Ordenação pelo eixo y sem região de influência;
- Ordenação pelo eixo y com região de influência;
- Método do setor;
- Baseada em ladrilhos (ladrilho de 4x4 [largura x altura]).

Como dito anteriormente, o Space Invaders possui características distintas do Breakout. Desta forma, alguns tipos de detecção de colisão que foram implementados no Breakout, e que não fazem sentido no Space Invaders, não foram implementados e vice-versa. Por exemplo, não faz sentido falar em Gerenciador de Objetos para o Space Invaders, pois os objetos que fazem parte deste jogo são dinâmicos. Assim como, devido a esta mesma característica, não faz muito sentido implementar o método do setor para o Breakout, pois os tijolos possuem suas posições sempre estáticas na tela.

Segue abaixo o esquema de classes utilizado na implementação do Space Invaders.

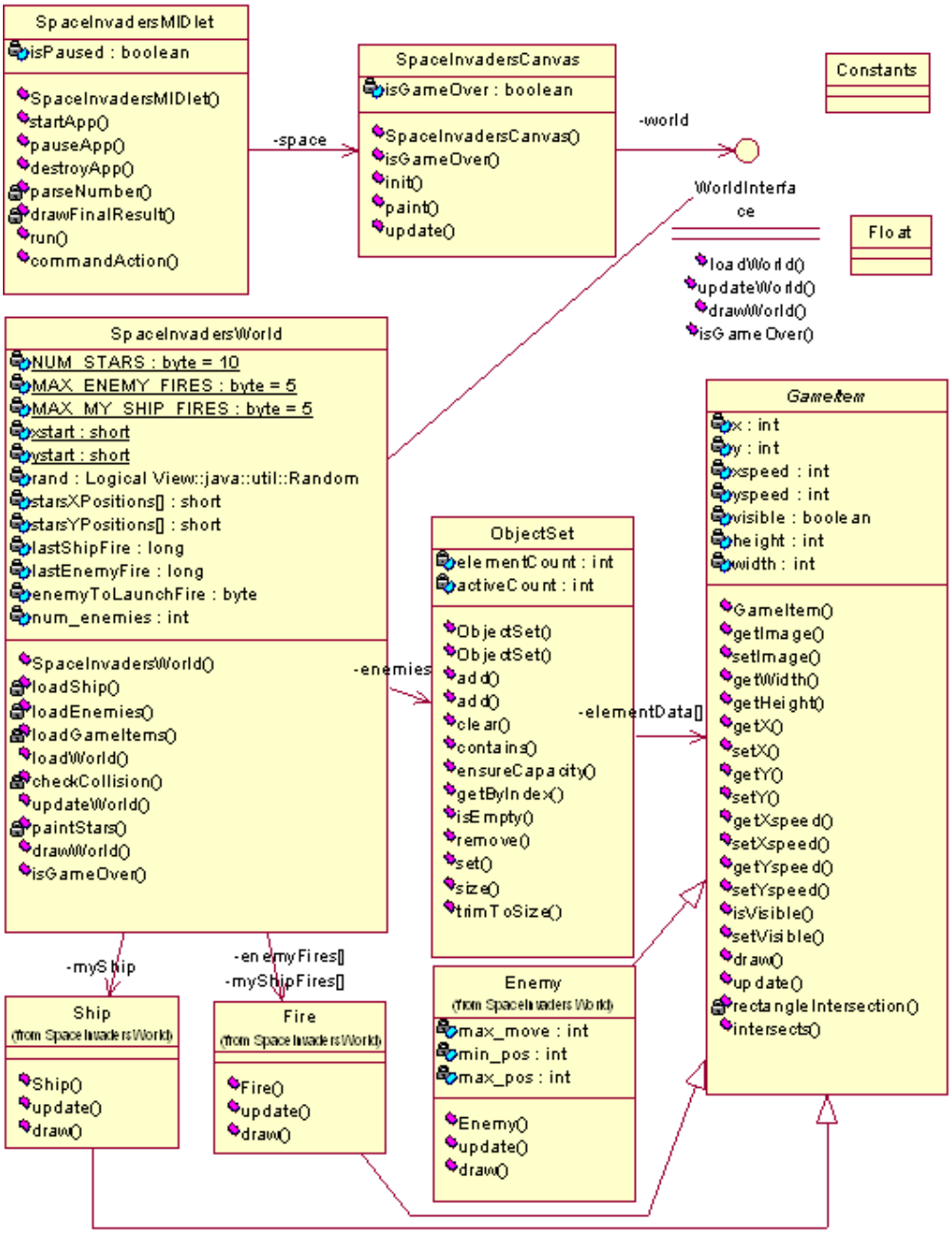


Figura 5.2: Diagrama de classes da implementação das técnicas de detecção de colisão para o Space Invaders.

O esquema utilizado na implementação do Space Invaders não é muito diferente do esquema utilizado no Breakout. A classe `SpaceInvadersMIDlet` é a classe que estende do MIDlet e que implementa o controle de execução da aplicação (controle dos níveis e iterações). Assim como no Breakout, cada iteração também foi implementada

com dois níveis, um com dez naves inimigas e outro com o dobro. A classe `SpaceInvadersCanvas` é responsável pelo gerenciamento da parte gráfica do jogo (ela estende da classe abstrata `Canvas` do MIDP). Também como no Breakout, a implementação do mundo deve ser feita implementando-se os métodos definidos na interface `WorldInterface`, possibilitando assim a implementação das diversas técnicas apenas modificando a implementação da classe `SpaceInvadersWorld`, que contém a porção de código específica à implementação do mundo do jogo e da técnica de detecção de colisão.

Devido à complexidade de implementação deste jogo ser um pouco maior em relação ao Breakout, introduzimos algumas outras classes que se fizeram necessárias para uma aplicação mais bem estruturada do ponto de vista da metodologia OO. A classe `ObjectSet` é uma classe auxiliar que permite manipular um conjunto de objetos. No caso específico da implementação do Space Invaders, ela é utilizada para armazenar o conjunto de naves inimigas. A classe `GameItem` é utilizada para evitar replicação de código para os diversos objetos do jogo (fogo amigo e inimigo, o inimigo e a nave do jogador), pois estes itens do jogo possuem características comuns, como velocidade, posição relativa na tela, altura e largura, uma imagem associada, etc. Desta forma, os objetos do jogo estendem dessa classe para herdar o código comum a eles, necessitando implementar apenas a parte específica de cada objeto. As classes `Fire`, `Enemy` e `Ship` dizem respeito, respectivamente, às implementações dos objetos do jogo: tiro e bomba, a nave inimiga e a nave do jogador (ou nave amiga). O apêndice B mostra as implementações dos métodos de detecção de colisão para as diversas técnicas implementadas no Space Invaders.

Assim como no Breakout, a implementação do Space Invaders também disponibiliza uma variável configurável no arquivo JAD. A variável "NUM_ITERATIONS" para indicar o número de iterações que o MIDlet precisa executar. O valor que esta variável pode assumir é um valor inteiro positivo não nulo.

Um desafio tanto para a implementação do Breakout quanto para a do Space Invaders em relação à técnica baseada em ladrilhos é implementar o movimento dos objetos mantendo compatibilidade da velocidade e do deslocamento com a implementação das outras técnicas. Isto porque a unidade de deslocamento desta técnica é o ladrilho, enquanto que a unidade de deslocamento dos objetos para as outras técnicas é o *pixel*. Em relação à implementação do Space Invaders, por exemplo, a

velocidade horizontal das naves inimigas é de um *pixel*. As naves se movimentam de um em um *pixel* horizontalmente a cada ciclo do jogo. Para obter este mesmo efeito na implementação baseada em ladrilhos, levando-se em consideração que o ladrilho da implementação é de quatro *pixels* de largura por quatro de altura e que as naves só podem se movimentar ladrilho a ladrilho, faz-se necessário implementar o controle das iterações de tal forma que as naves só possam se deslocar de um ladrilho a cada quatro iterações do jogo, fazendo com que sua velocidade final permaneça a mesma das implementações das outras técnicas. Um problema introduzido por meio deste método é o efeito do movimento discreto dos objetos, devido ao deslocamento de vários *pixels* (tamanho do ladrilho) de uma única vez por ciclo do jogo.

Para que o jogo ficasse automático, o controle da nave amiga pelo jogador foi retirado. Como foi discutido na seção 3.3.3, o Space Invaders passou a ter o movimento da nave amiga controlado automaticamente pelo jogo. Seu movimento passou a ser constante e repetitivo de um lado a outro da área do jogo com ataque temporizado (i.e., a nave emite tiros em intervalos regulares pré-definidos).

5.2 Configurações Experimentais

Esta seção cobre, como foi mencionado na seção 3.2, as configurações necessárias para a reprodução do experimento. Essas informações de configuração consistem em ferramentas utilizadas, valores de variáveis e etapas de execução do experimento.

5.2.1 Configuração do Ambiente

O primeiro passo é identificar, instalar e configurar as ferramentas necessárias à execução dos testes. As ferramentas utilizadas para desenvolvimento e para rodar o experimento com suas respectivas versões foram:

- *J2SDK*, versão 1.4.2.
- *Eclipse Platform*, versão 3.0.1.
- *Plugin EclipseME*, versão 0.7.0.
- *Wireless Toolkit*, versão 1.0.4.

O *Wireless Toolkit* da Sun é o emulador mais largamente utilizado atualmente para executar aplicações J2ME e por isso foi escolhido como ferramenta básica para rodar os testes. Decidimos utilizar o Eclipse como plataforma de desenvolvimento por ser uma ferramenta grátis, estável, de fácil aprendizagem e cada vez mais utilizada para

desenvolvimento de aplicações Java. Apesar de ser uma ótima ferramenta de desenvolvimento, sua escolha só foi possível graças ao *plugin* que permite a integração entre o *Eclipse* e o *Wireless Toolkit*.

As informações de instalação das ferramentas estão nos *sites* de *download* relacionados. É preciso instalar a plataforma Eclipse de acordo com o especificado na seção de instalação da ferramenta, depois instalar o *Wireless Toolkit*, e por último seguir as instruções de instalação do *plugin* descritas no *site*, configurando as informações necessárias dentro do Eclipse para que haja a integração entre ele e o *Wireless Toolkit*. Uma vez que o ambiente esteja instalado, basta configurar o eclipse para carregar o ambiente de desenvolvimento dos *MIDlets*, a partir daí eles estarão prontos para serem executados.

Além das ferramentas acima citadas, usadas para executar o experimento no PC, utilizamos também dois celulares diferentes:

- Nokia 6100;
- Siemens MC60.

Ambos suportam o MIDP 1.0. O Nokia 6100 possui uma tela de 128x128 *pixels* com *display* colorido de 4096 cores. Já o Siemens MC60 possui uma tela de 101x80 pixels também com *display* colorido de 4096 cores.

5.2.2 Configuração de Variáveis

Como mencionado nos detalhes de implementação dos dois jogos, uma variável de configuração foi disponibilizada nos arquivos JAD para indicar o número de iterações que o teste deve ser executado, para ambos os jogos. Os seguintes valores são utilizados para rodar os testes:

- Os seguintes valores foram utilizados para a variável “NUM_ITERATIONS”:
    - 30 para execução no emulador sem verificação de performance;
    - 15 para execução no celular;
    - 5 para execução no emulador para verificar o percentual do tempo gasto na execução de um método usando o *profiler*.

Estes valores devem ser modificados (ou incluídos, caso não estejam presentes) no arquivo JAD de cada MIDlet antes da execução do mesmo. Decidimos por 30 iterações para a execução no emulador por ser um número razoável para se obter um valor aceitável. Cada execução das 30 iterações no emulador (rodando num Pentium III 800 MHz) leva em torno de 40 minutos, o que nos impediu de aumentar este número,

pois qualquer incremento não traria grandes benefícios nos resultados e poderia trazer um aumento razoável no tempo gasto na execução do experimento. Vale salientar que cada execução das 30 iterações teve que ser feita para cada uma das técnicas de detecção de colisão. Como o poder de processamento dos celulares é bem menor que o do PC, resolvemos adotar 15 iterações para manter basicamente o mesmo tempo de execução. No caso do percentual de tempo gasto pelos métodos relevantes, o número de iterações não influencia tanto quanto no caso do número de *fps*. Desta forma, resolvemos adotar apenas 5 iterações para prolongar um pouco mais o tempo total de execução do experimento a fim de minimizarmos possíveis variações em função da execução de outros processos pelo PC.

5.2.3 Etapas para a Execução do Experimento

Após a instalação e configuração do ambiente e definição dos valores das variáveis de tempo de execução, o experimento pode ser executado. As etapas da execução são guiadas em função das métricas a serem observadas.

O número de quadros por segundo é a única métrica que deve ser executada tanto no ambiente do PC quanto no ambiente do celular. Considerando o ambiente do PC, deve-se executar cada MIDlet dos dois jogos usando-se o Eclipse. Cada MIDlet deve executar os dois níveis implementados para cada uma das iterações, cujo valor foi definido na seção anterior. Ao final da última iteração, o MIDlet gera o resultado tanto para o console do Eclipse quanto para uma tela no emulador. O resultado gerado para o console é mais detalhado, pois mostra, além da média do número de quadros por segundo, os valores do tempo e número de quadros executados para cada iteração por número de objetos. A Figura 5.3 ilustra uma parte (apenas 7 dos 16 objetos do nível 1) da saída de console gerada com cinco iterações para o jogo Breakout com a técnica de detecção padrão. A Figura 5.4 ilustra essa mesma execução de cinco iterações com a saída para uma tela no próprio emulador. Vale observar que os valores de tempo e número de quadros para cada iteração não são mostrados, apenas os resultados finais, que são a média aritmética de cada iteração. Para o ambiente do celular, o MIDlet deve ser carregado no dispositivo e executado conforme as especificações do dispositivo em questão. Após a completa execução das iterações da aplicação, o MIDlet deve gerar a mesma tela que foi gerada para o emulador, com as informações do resultado final do teste.

A métrica do tempo proporcional de execução das funções relevantes só pode ser obtida por meio do emulador *Wireless Toolkit*, já que este disponibiliza um *profiler*, onde é possível obter os resultados para esta métrica. O MIDlet deve ser executado pelo Eclipse com o *profiler* do *Wireless Toolkit* habilitado. Ao final da execução, deve-se fechá-lo para que a tela de resultados do *profiler* apareça, deve-se, então, coletar os valores para a variável em questão e para os métodos relevantes.

```
Date: 3/1/2005
Number of iterations: 5
Collision type: Object-to-object

(Time in milisec, FPS) for each iteration  [ average FPS ]
--------------------------------------------------------------------------------
Level 1 (16 objects)
16 objects: (1242;  19,32) (1342;  17,88) (1332;  18,01) (1281;  18,73) (1262;  19,01)   [  18,59 ]
15 objects: (2614;  17,59) (2463;  18,67) (2513;  18,30) (2534;  18,15) (2394;  19,21)   [  18,38 ]
14 objects: (2544;  18,86) (2384;  20,13) (2564;  18,72) (2544;  18,86) (2483;  19,33)   [  19,18 ]
13 objects: (  50;  20,00) (  50;  20,00) (  50;  20,00) (  50;  20,00) (  50;  20,00)   [  20,00 ]
12 objects: (2243;  21,84) (2313;  21,18) (2183;  22,44) (2273;  21,55) (2324;  21,08)   [  21,62 ]
11 objects: (2494;  24,05) (2664;  22,52) (2674;  22,43) (2734;  21,94) (2623;  22,87)   [  22,76 ]
10 objects: ( 230;  34,78) ( 340;  23,52) ( 341;  23,46) ( 350;  22,85) ( 351;  22,79)   [  25,48 ]
```

Figura 5.3: Parte da saída gerada pelo console do Eclipse ao final de execução de um teste para a técnica padrão do Breakout. Estão sendo mostrados aqui apenas 10 objetos do primeiro nível.

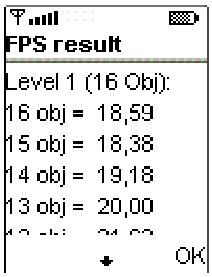


Figura 5.4: Saída gerada numa tela do emulador.

São três os métodos relevantes para comparação entre as técnicas de detecção de colisão:

- `updateWorld`;
- `drawWorld`;
- `checkIntersection`;

O método `updateWorld`, como dito antes, é responsável por atualizar as coordenadas de todos os objetos da tela antes da atualização gráfica. Este método permite comparar o percentual do tempo de execução gasto na atualização dos objetos entre as diversas técnicas, pois todas, sem exceção, implementam este método definido na interface de implementação do mundo do jogo (`worldInterface`). Este método é

relevante porque é possível obter por meio dele uma idéia do tempo gasto em praticamente todo cálculo computacional de tempo de execução feito pelo jogo.

O método `drawWorld` é responsável pela atualização gráfica dos objetos na tela, isto é, ele é o responsável por chamar as funções gráficas de desenho de imagens, de retângulos, de limpar a tela, etc. Este método tem relevância porque permite a visualização do percentual do tempo total gasto com a atualização gráfica dos objetos que, no caso dos celulares, é responsável pelo maior tempo de execução, principalmente devido a baixa qualidade e a pouca quantidade de recursos que possibilitem uma execução rápida, diferentemente dos PCs, que atualmente contam com placas aceleradoras gráficas e outros dispositivos que aceleram a comunicação e o desempenho do monitor.

Finalmente, o método `checkIntersection`, que é responsável pela verificação de colisão, foi levado em consideração porque permite obter uma comparação direta da performance entre as diferentes técnicas de detecção de colisão. O único problema em relação a este método é que ele não pode ser usado para uma comparação entre a técnica baseada em ladrilhos e as outras, pois, como foi dito em seção anterior, a técnica baseada em ladrilhos possui uma implementação do mundo de forma bastante diferente inviabilizando uma comparação justa, como é feita, por exemplo, entre as técnicas não baseadas em ladrilhos. Isto porque, para estas técnicas, a implementação do `checkIntersection` é praticamente a única modificação que existe entre elas. Desta forma, num âmbito mais global, deve-se levar em consideração o método `updateWorld`, pois ele inclui a chamada a este método, levando, então, em consideração o tempo gasto na verificação de colisão e de atualização dos objetos.

A métrica do tamanho de código (medido em número de linhas de código) deve ser obtida usando qualquer ferramenta que faça contagem do número de linhas de código sem levar em consideração os comentários e as linhas em branco. Para cada técnica (MIDlet), deve-se obter o número de linhas de código de todos os arquivos e somar os valores, gerando, então, o tamanho de código usado na implementação técnica.

5.3 Resultados

Nesta seção são mostrados os resultados obtidos pela execução dos testes (MIDlets) para as métricas estabelecidas em função dos dois jogos escolhidos.

5.3.1 Resultados do Breakout

Para todos os resultados mostrados nas seções seguintes, o seguinte mapeamento dos nomes das técnicas utilizadas deve ser considerado:

- x-influ – Ordenação pelo eixo x desconsiderando-se a região de influência;
- x+influ – Ordenação pelo eixo x com região de influência;
- y-influ – Ordenação pelo eixo y desconsiderando-se a região de influência;
- y+influ – Ordenação pelo eixo y com região de influência;
- ladr6x12 – Baseada em ladrilhos (ladrilho de 6x12);
- gerenc4 – Gerenciador de objetos (com quatro objetos por gerenciador);
- gerenc8 – Gerenciador de objetos (com oito objetos por gerenciador);
- obj/obj – Padrão (*Object-to-object*);
- reg_influ – Região de influência;
- ladr6x4 – Baseada em ladrilhos (ladrilhos de 6x4).

5.3.1.1 Métrica 1: Número de quadros por segundo

Os resultados obtidos para os níveis 1 e 2 das técnicas implementadas para o Breakout encontram-se nas figuras que se seguem. O título na parte superior do gráfico identifica o nível e o ambiente onde o experimento foi executado. O eixo y indica o número de quadros por segundo (*fps*) e o eixo x indica o número de objetos (No caso do Breakout, tijolos) presentes na tela. Do lado direito do gráfico encontra-se a legenda das técnicas de detecção de colisão. O gráfico mostra o número de quadros por segundo em função da quantidade de objetos presentes na tela. Para esta métrica, quanto maior o valor, mais expressivo é o resultado.

Com base no gráfico da Figura 5.6, pode-se perceber que a técnica padrão (*Object-to-object*) mostrou o resultado menos significativo na média. Isto é, a técnica padrão alcançou valores de número de quadros por segundo em função do número de objetos menores que as outras técnicas. Por outro lado, a técnica baseada em ladrilhos de 6x12 (ladr6x12) mostrou os resultados mais significativos, mantendo valores de *fps* acima dos demais na maioria das vezes.

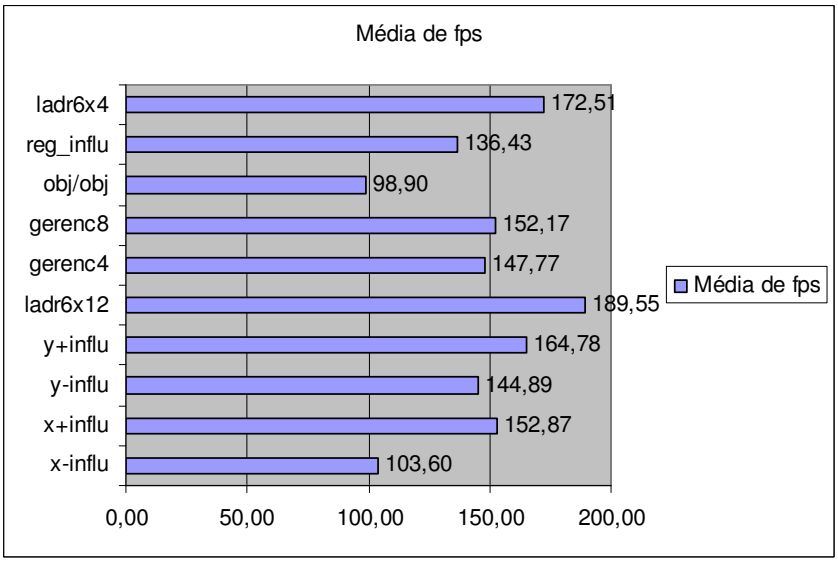


Figura 5.5: Média dos valores de fps para o nível 1 do jogo Breakout (executado no PC).

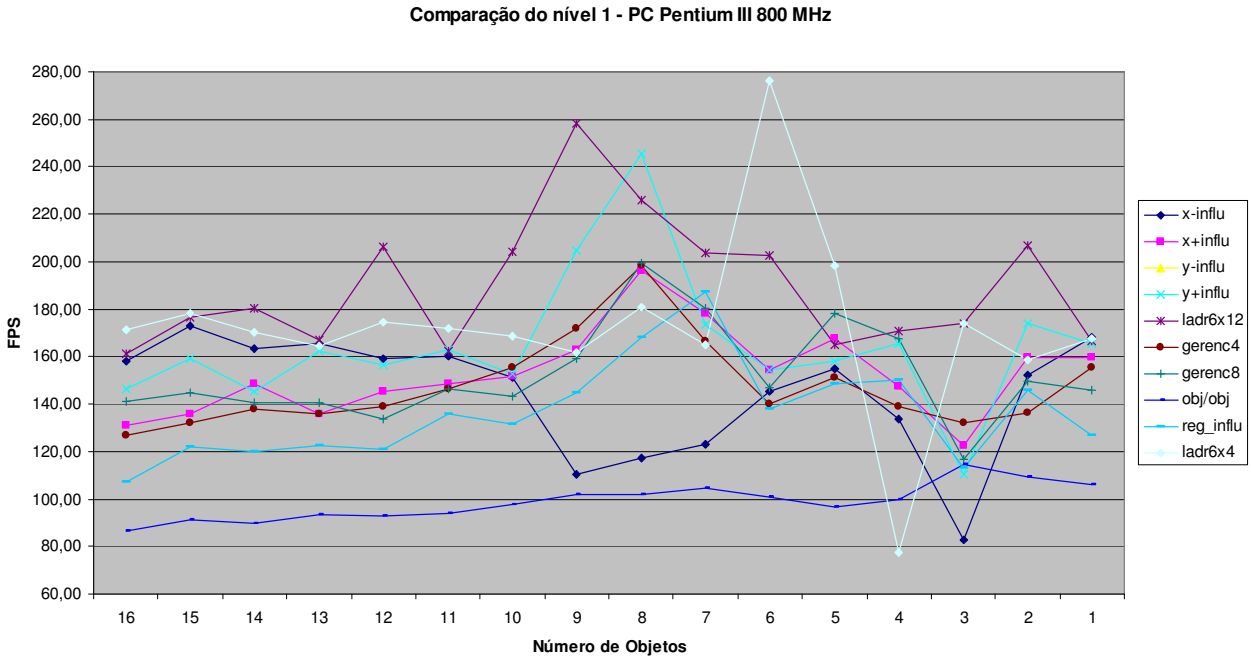


Figura 5.6: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 1 do jogo Breakout (executado no PC).

O gráfico da Figura 5.8 mostra que as duas técnicas baseadas em ladrilho obtiveram os resultados mais expressivos, com a técnica baseada em ladrilhos de 6x4 (ladr6x4) mostrando um valor ligeiramente maior, porém com uma instabilidade (variação de *fps* em função do número de objetos) ligeiramente maior que a técnica de

ladrilhos de 6x12 (ladr6x12). A técnica padrão, confirmando os resultados do gráfico do nível 1, mostrou o resultado menos significativo na média.

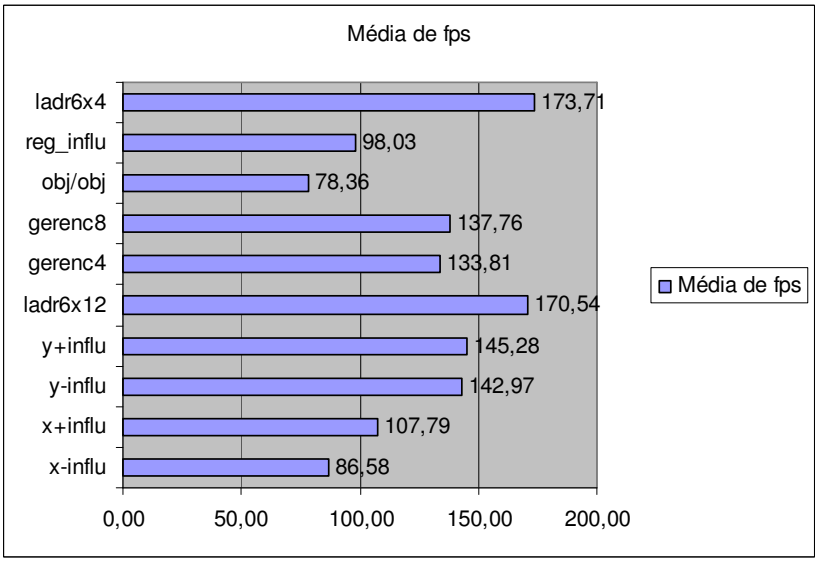


Figura 5.7: Média dos valores de fps para o nível 2 do jogo Breakout (executado no PC).

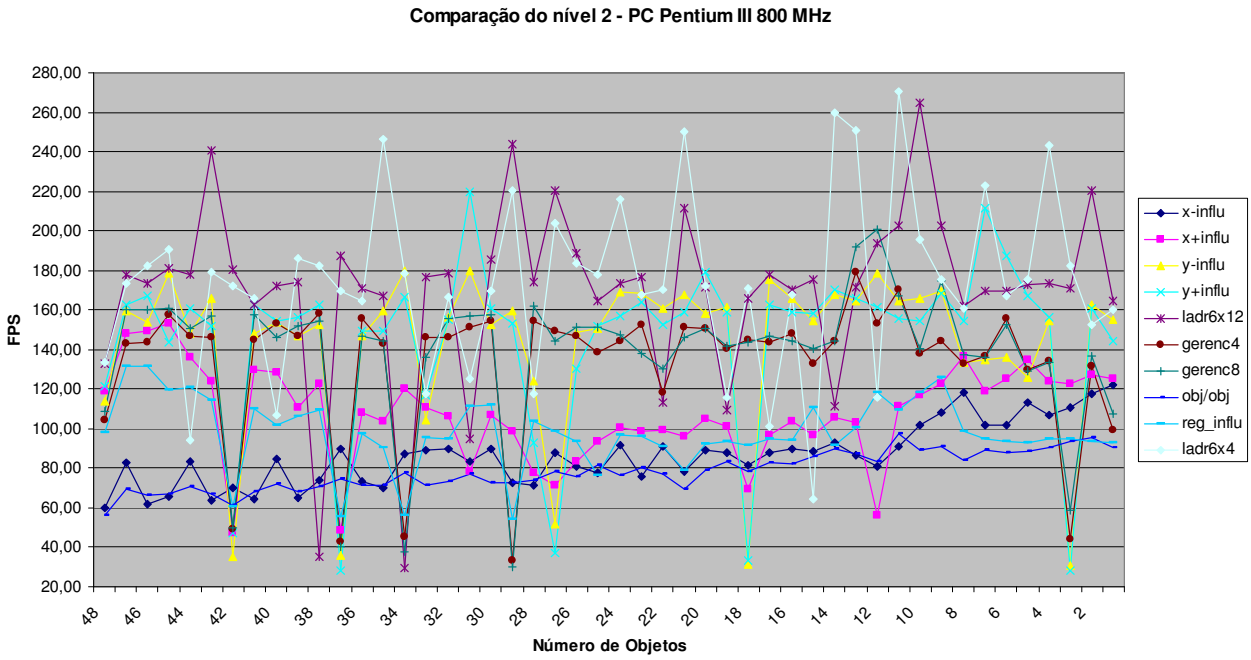


Figura 5.8: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 2 do jogo Breakout (executado no PC).

Os resultados da Figura 5.10 reforçam os resultados dos gráficos obtidos no emulador em PC. Pode-se observar que a técnica padrão não mostrou um resultado

significativo. As demais técnicas mostraram resultados bastante aproximados entre si, algumas um pouco mais instáveis que as outras. A técnica baseada em ladrilhos de 6x12 (ladr6x12) mostrou um valor um pouco mais expressivo que as demais. Já a técnica baseada em ladrilhos de 6x4 (ladr6x4) mostrou um resultado um pouco menos significativo que as técnicas de gerenciamento de objetos (gerenc4 e gerenc8).

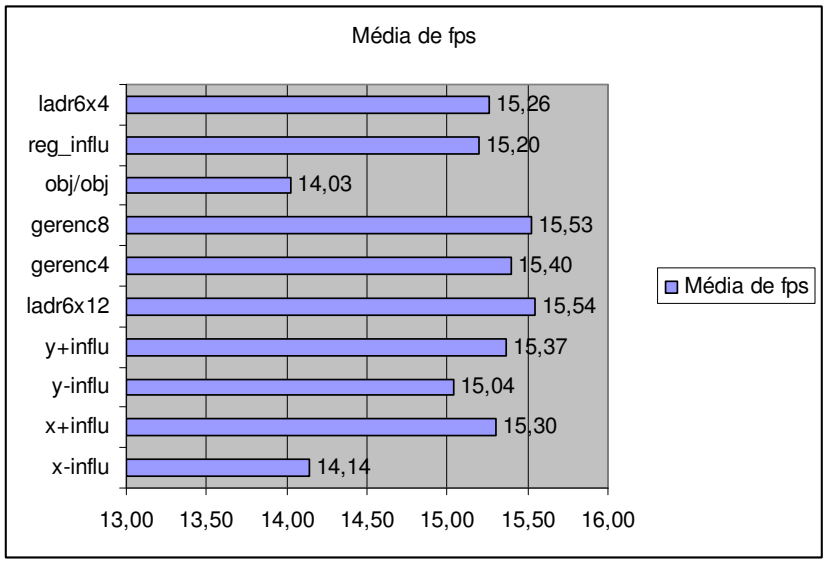


Figura 5.9: Média dos valores de fps para o nível 1 do jogo Breakout (executado no celular MC60).

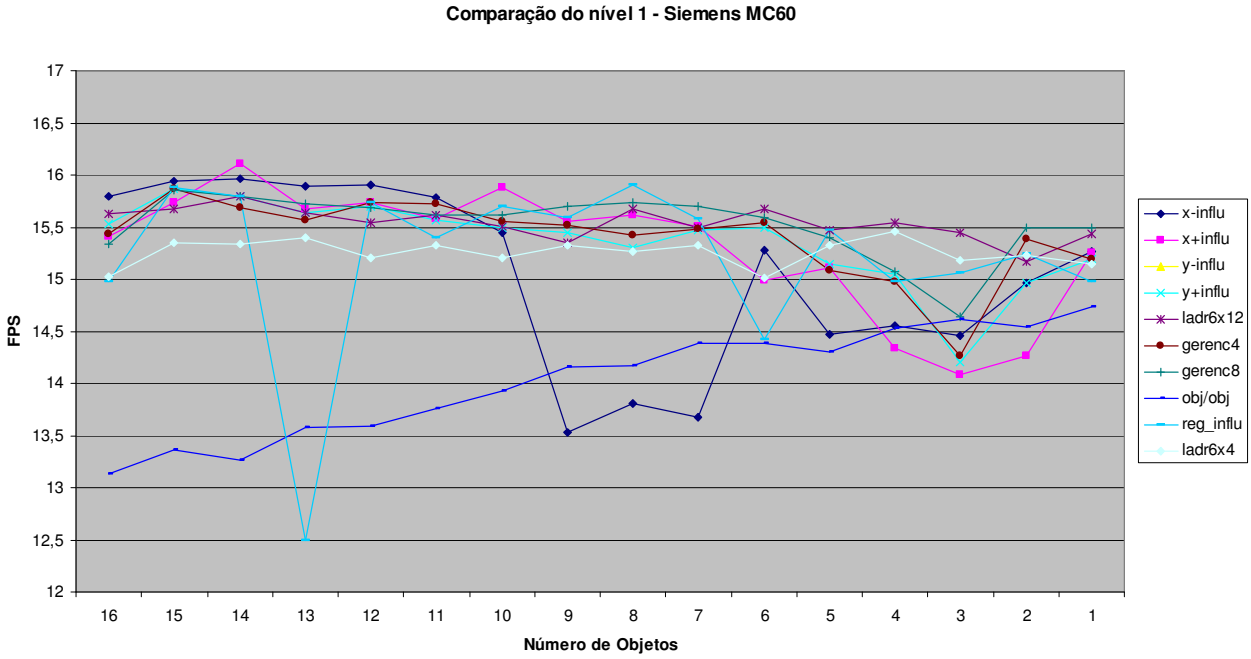


Figura 5.10: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 1 do jogo Breakout (executado no celular Siemens MC60).

Os resultados para o nível 2 do celular Siemens MC60 Figura 5.12 também confirmam que o resultado de maior expressividade é o da técnica de ladrilhos de 6x12 (ladr6x12) e o que demonstrou um valor menor foi o da técnica padrão.

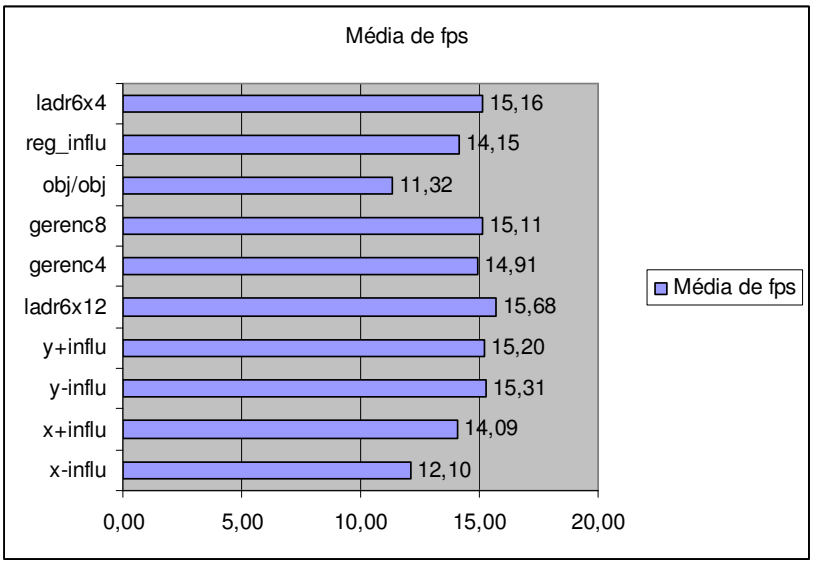


Figura 5.11: Média dos valores de fps para o nível 2 do jogo Breakout (executado no celular MC60).

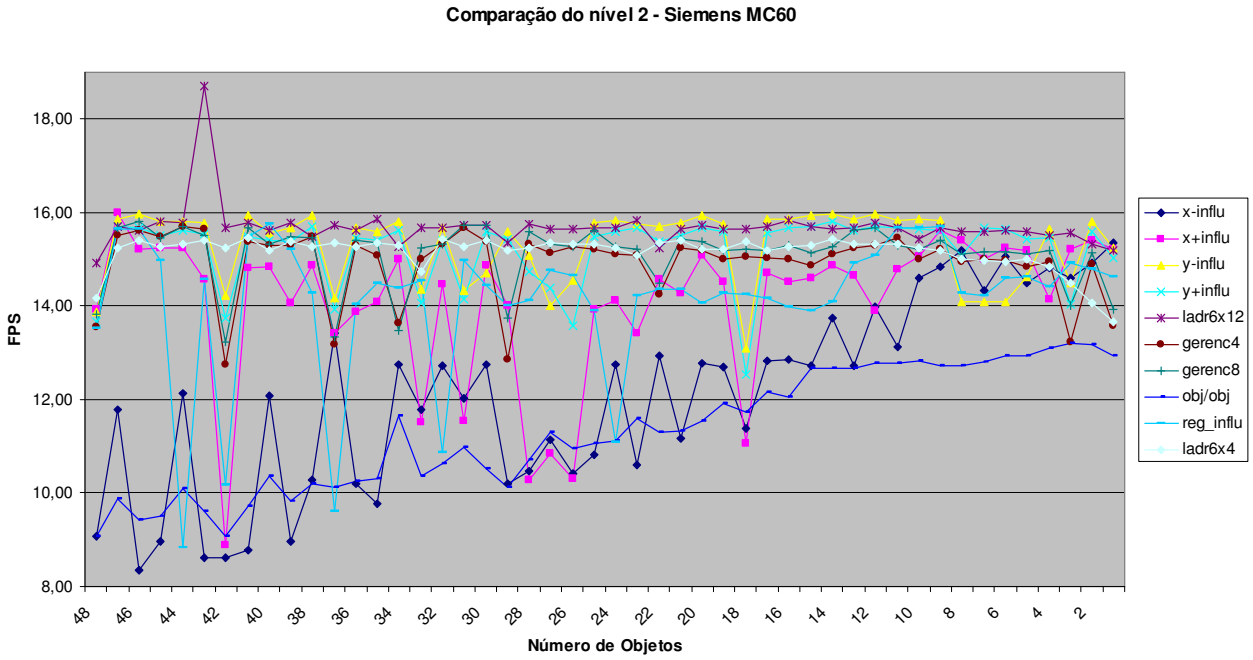


Figura 5.12: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 2 do jogo Breakout (executado no celular Siemens MC60).

Os resultados obtidos para o nível 1 do celular Nokia 6100 Figura 5.14 foram bastante parecidos para todas as técnicas. Considerando-se a média dos valores, a técnica baseada em ladrilhos de 6x12 (ladr6x12) mostrou o resultado mais significativo. Já o resultado de menor valor ficou com a técnica de ordenação pelo eixo x desconsiderando a região de influência.

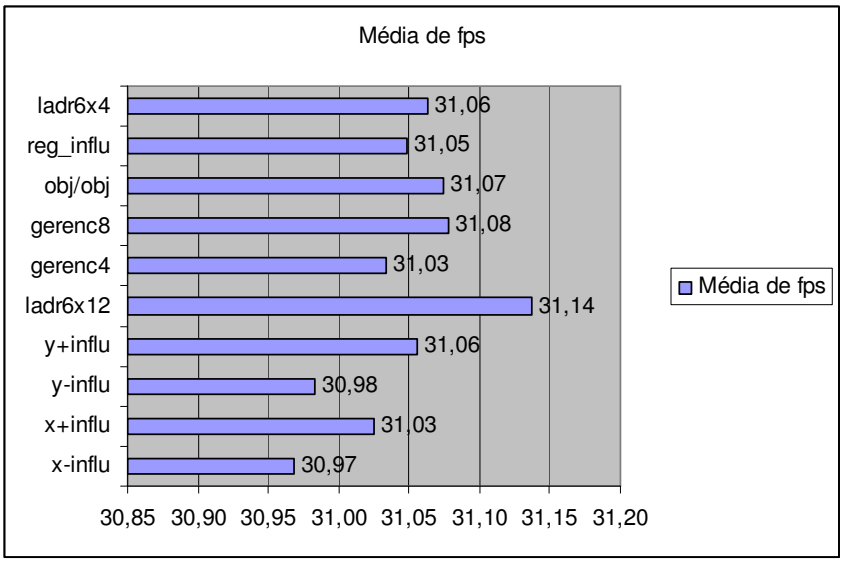


Figura 5.13: Média dos valores de fps para o nível 1 do jogo Breakout (executado no celular 6100).

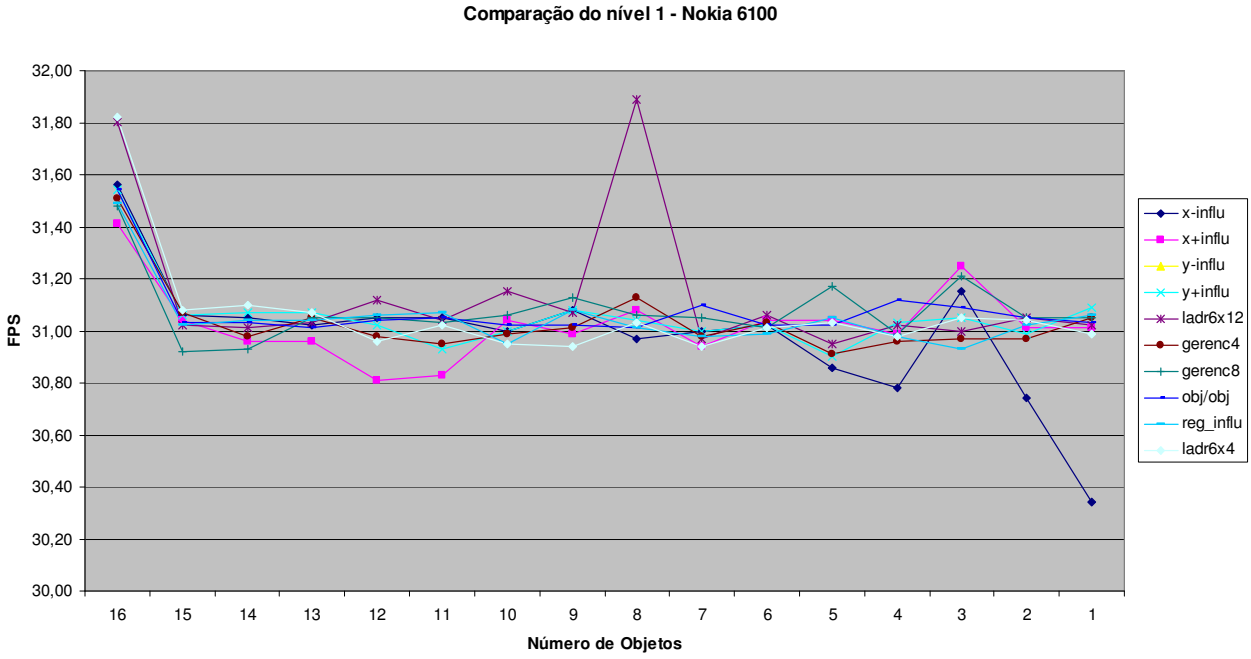


Figura 5.14: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 1 do jogo Breakout (executado no celular Nokia 6100).

O nível 2 também mostrou resultados bastante próximos entre as diferentes técnicas. Pode-se observar que a técnica de ordenação pelo eixo x desconsiderando a região de influência mostrou o resultado menos significativo. Para o resultado de maior expressividade, as duas técnicas baseadas em ladrilhos e as duas técnicas de gerenciamento de objetos tiveram valores bastante parecidos.

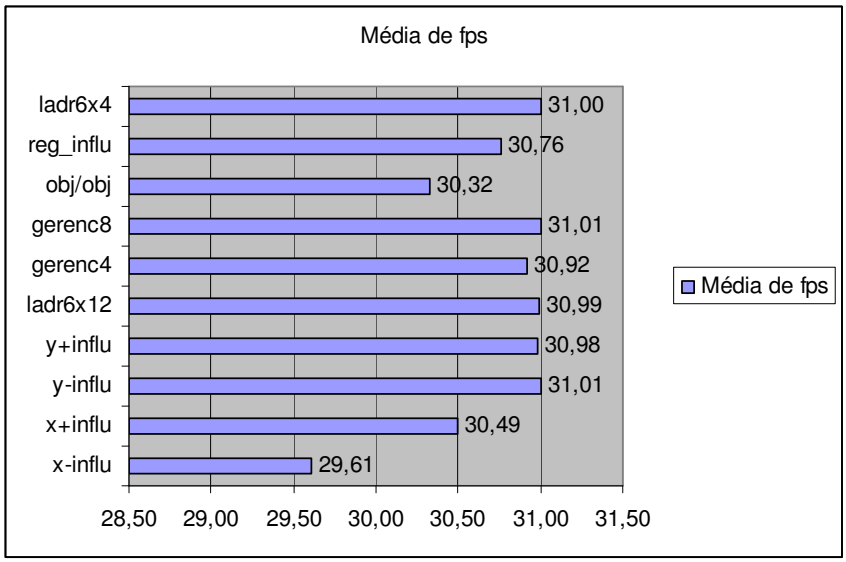


Figura 5.15: Média dos valores de fps para o nível 2 do jogo Breakout (executado no celular 6100).

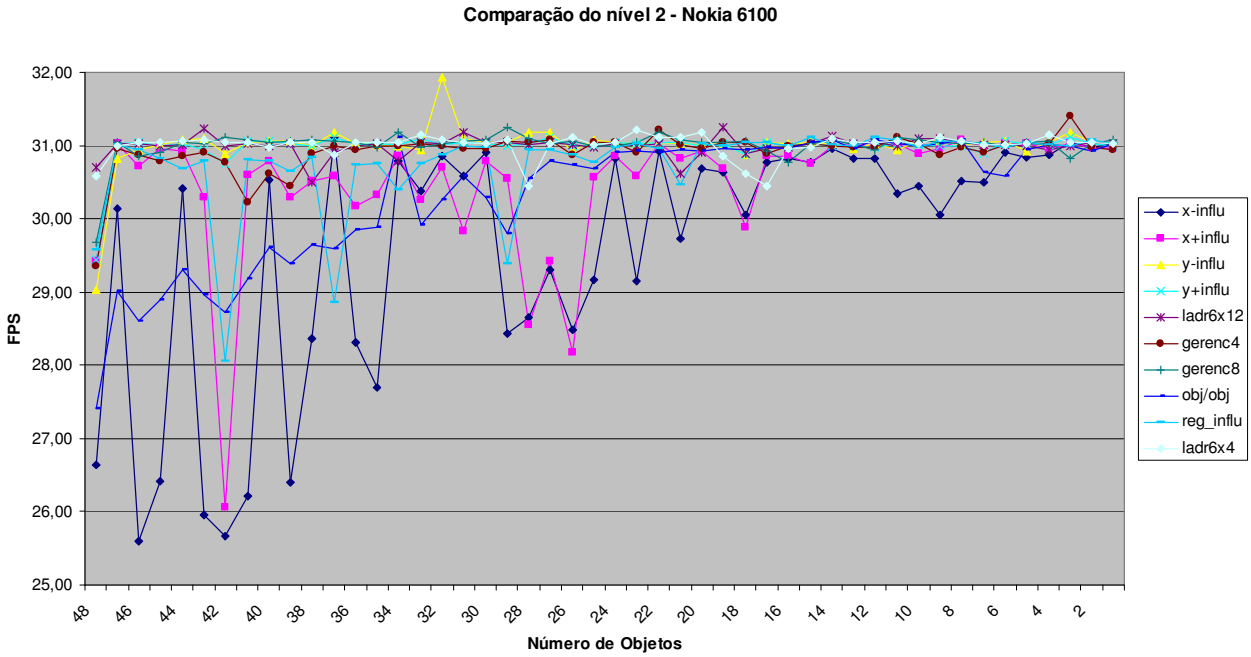


Figura 5.16: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 2 do jogo Breakout (executado no celular Siemens MC60).

5.3.1.2 Métrica 2: Tamanho de código

A Figura 5.17 demonstra comparativamente os valores de tamanho de código obtidos para cada técnica implementada. Foram consideradas as linhas de código sem linhas em branco e sem linhas de comentário. O eixo x indica o número de linhas total da implementação do MIDlet.

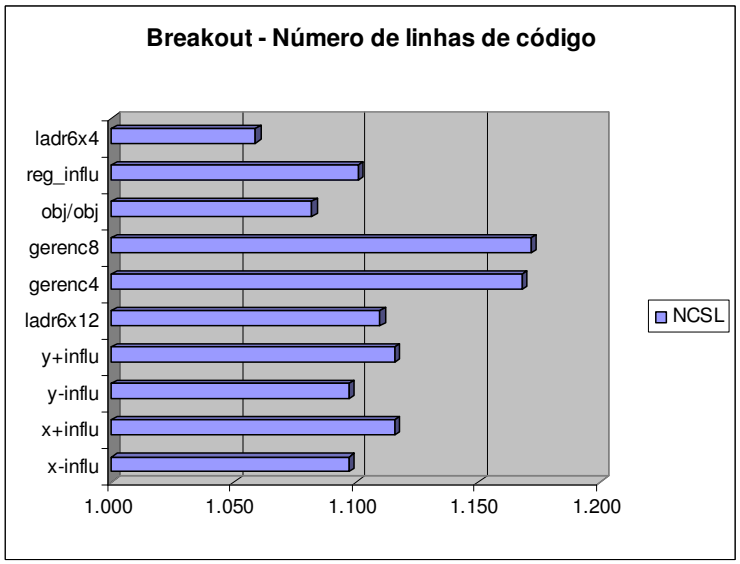


Figura 5.17: Comparação do tamanho de código para as técnicas implementadas no jogo Breakout.

Com base nos valores mostrados no gráfico, pode-se observar que a técnica baseada em ladrilho de 6x4 foi a técnica que precisou de menos linhas de código para ser implementada. Considerando-se esforço de implementação em função do tamanho do código-fonte, pode-se dizer que as técnicas de gerenciamento de objetos foram as que necessitaram de maior esforço de implementação. Não necessariamente o tamanho de código-fonte está relacionado à complexidade de implementação. Ainda assim, esta métrica é importante como foi visto no final da Introdução.

5.3.1.3 Métrica 3: Percentual de execução dos métodos relevantes

A seguir são mostrados os valores obtidos com a ferramenta *profiler* do *Wireless Toolkit*. Na parte superior do gráfico está o nome do método avaliado. O eixo y indica a técnica e o eixo x indica o percentual do tempo total de execução do MIDlet que foi gasto com o método em questão. O valor percentual leva em consideração também o tempo gasto com os métodos chamados pelo método em questão.

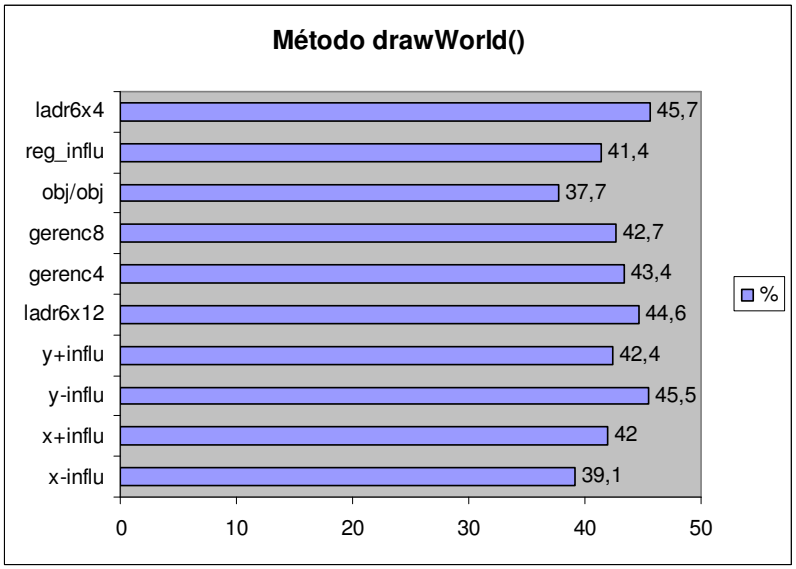


Figura 5.18: Percentual de execução do método `drawWorld`.

O método `drawWorld` possui basicamente a mesma implementação para todas as técnicas. Até mesmo as técnicas baseadas em ladrilhos possuem praticamente a mesma implementação das demais. Desta forma, a variação do tempo gasto neste método é em função dos outros métodos relevantes, que é a parte do código que diferencia os MIDlets uns dos outros. Isto é, comparativamente falando, quando uma técnica A mostra um valor maior que o da técnica B, isto significa que o método de detecção de colisão da técnica B levou mais tempo para ser executado, pois este método de atualização da tela possui a mesma implementação para as duas técnicas.

Exemplificando, pode-se observar que os valores do resultado indicam que a técnica padrão (*Object-to-object*) apresenta o resultado menos expressivo, pois precisou gastar mais tempo na detecção de colisão, visto que um percentual menor foi gasto no método de atualização da tela. Apenas 37,7% do tempo total de execução do MIDlet foi gasto neste método, indicando que foi preciso gastar um maior tempo no método de atualização dos objetos. Isto, por sua vez, indica que a técnica padrão consome mais processamento para a detecção de colisão que as demais.

Por meio dos resultados do gráfico da Figura 5.19, pode-se confirmar o que foi dito. A técnica padrão gastou o maior tempo para atualização dos objetos no decorrer da execução do MIDlet. As técnicas baseadas em ladrilho mostraram um ótimo resultado, indicando que suas implementações consomem menos processamento para a atualização dos objetos incluindo a verificação de colisão.

O método `checkIntersection` (Figura 5.20) foi levado em consideração na tentativa de fazer uma análise comparativa somente da porção do código que trata da detecção de colisão. No entanto, como foi explicado anteriormente, as técnicas baseadas em ladrilho possuem uma implementação do mundo diferente das outras técnicas. A verificação de colisão para estas técnicas foi implementada junto com a implementação da atualização dos objetos. Não foi possível separar de forma confiável (i.e., susceptível a uma comparação justa) a parte do código que trata da atualização dos objetos da parte que trata da detecção de colisão.

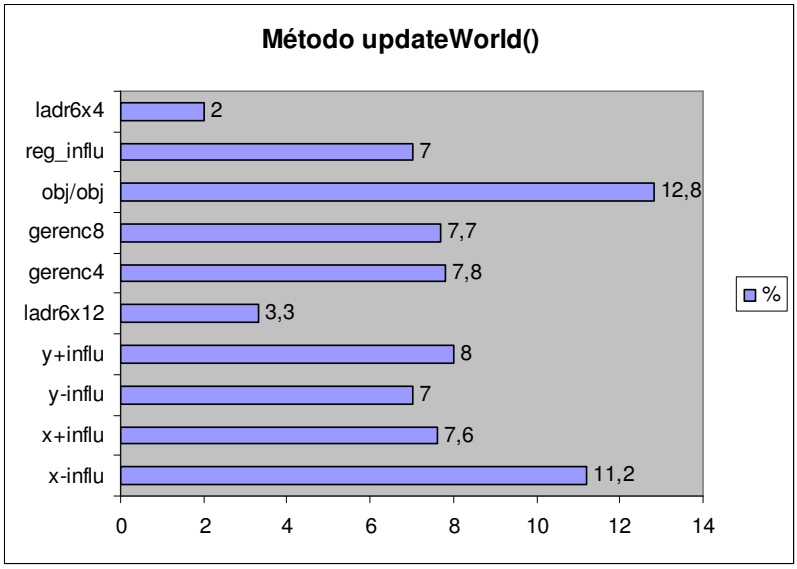


Figura 5.19: Percentual de execução do método `updateWorld`.

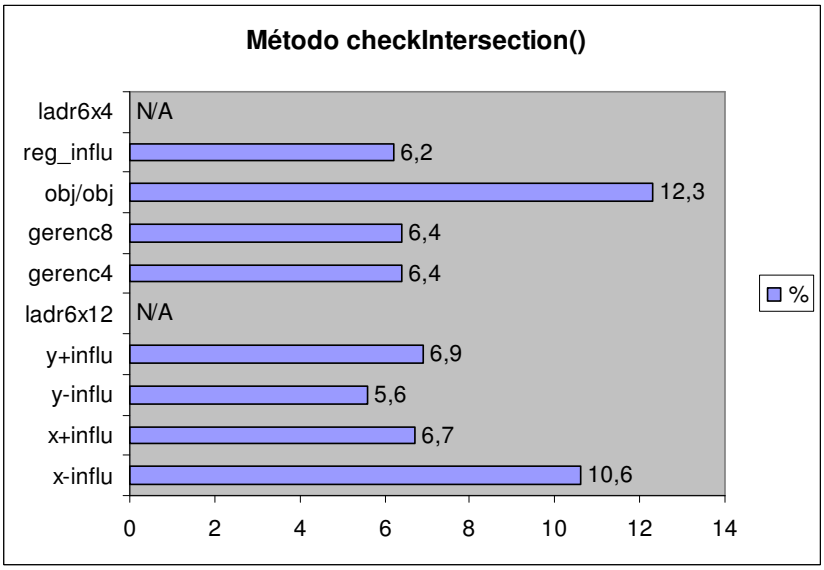


Figura 5.20: Percentual de execução do método `checkIntersection`.

Este método não pôde ser levado em consideração para uma análise comparativa entre todas as técnicas. Porém, desconsiderando-se apenas as técnicas baseadas em ladrilho, a comparação entre as demais é confiável. Pode-se observar a ineficiência, em relação às outras técnicas, da técnica padrão.

5.3.2 Análise dos Resultados para o Jogo Breakout

Como se pode perceber a partir dos resultados mostrados na seção anterior, a hipótese estabelecida na seção 3.2 não é sustentada:

> *Hipótese*: Não há diferença no número de quadros por segundo ou do percentual do tempo de execução gasto nas funções mais relevantes entre as técnicas de detecção de colisão consideradas, bem como não há diferença no tamanho final do código entre as técnicas.

As variações nos resultados entre as implementações das diferentes técnicas mostram que algumas técnicas são melhores que outras em função das métricas levadas em consideração.

De acordo com os resultados obtidos para o *Breakout*, pode-se notar a supremacia das implementações baseadas em ladrilho (ladr6x12 e ladr6x4). As duas técnicas apresentaram os resultados mais expressivos tanto para a métrica de quadros por segundo quanto para o percentual gasto nos métodos relevantes. Além do bom resultado para a métrica de número de linhas de código identificado na implementação da técnica ladr6x4, o que significa que o código está mais enxuto. Não é possível comparar os resultados em função do método específico de detecção de colisão (`checkIntersection`), pois como foi dito anteriormente, não se pode fazer uma comparação justa entre as técnicas, já que a implementação do mundo do jogo, para a técnica baseada em ladrilhos, é diferenciada das demais implementações.

No entanto, pode-se levar em consideração os resultados obtidos pelo método `updateWorld`, já que este é o método de mais alto nível comum a todas as implementações (i.e., o código desde o início da aplicação até o momento que este método é chamado é o mesmo para todas as implementações) e pode ser usado na comparação entre todas as técnicas. Pode-se notar, de acordo com o resultado deste método, que as duas técnicas (ladr6x12 e ladr6x4) possuem os resultados mais significativos no cálculo de atualização das coordenadas dos objetos do jogo. Com base na análise do *fps* e da métrica do percentual de tempo dos métodos relevantes, pode-se afirmar que as técnicas baseadas em ladrilho mostraram a performance mais

significativa para a implementação de detecção de colisão, com a técnica de ladrilhos de 6x4 mostrando valores ligeiramente mais expressivos que a técnica de ladrilhos de 6x12.

Apesar da técnica de ladrilhos de 6x4 mostrar uma performance um pouco melhor, por ser uma implementação 100% baseada em ladrilhos, o movimento da bola pela tela se mostra um tanto quanto discretizado em relação as demais técnicas. Desta forma, a técnica de ladrilhos de 6x12 seria a mais indicada para a implementação, pois apresenta resultados bastante satisfatórios (tanto quanto a de ladrilhos de 6x4) e, por ser um jogo híbrido, permite um movimento suave da bola pela tela, o que garante maior jogabilidade para o usuário final.

Complementando a análise, podemos identificar com base nos resultados que a técnica padrão (cuja verificação de colisão é feita entre todos os objetos) deve ser evitada. É melhor dar preferência a alguma das técnicas analisadas ou mesmo a uma solução híbrida, permitindo que haja uma maior eliminação de testes de colisão e aumentando a performance final do jogo.

Foi possível constatar pelos gráficos do número de quadros por segundo que não houve um incremento uniforme dos valores em função do número de objetos, mas sim uma volatilidade devido principalmente a uma otimização da atualização da interface gráfica, discutida na seção abaixo. Para boa parte das técnicas, os resultados variam em torno de um valor de forma constante, tanto para o nível 1 quanto para o nível 2. Também não foi constatada grande diferença entre os resultados apresentados pelo emulador ou pelos celulares. Apenas a diferença já prevista dos valores entre o emulador e os celulares. Entre os dois celulares, o Nokia 6100 mostrou resultados mais significativos no geral que o Siemens MC60.

5.3.2.1 Instabilidade dos resultados do Breakout

Os valores obtidos para a métrica de *frames* por segundo, como pôde ser constatado pelos resultados da seção 5.3.1.1, em todos os ambientes e nos dois níveis, para o Breakout se mostraram bastante instáveis, variando bastante ao longo do número de objetos presentes na tela. Isto se deve principalmente pela forma como o cálculo do número de *fps* é feito e pelas regras do jogo em si.

O número de *frames* por segundo é calculado dividindo-se o número de *frames* (entre duas colisões consecutivas) pelo tempo obtido entre as duas colisões. Com isto, obtém-se o valor de *fps* para um determinado número de tijolos na tela. Quando dois

tijolos estão muito próximos, a colisão pode acontecer no intervalo de apenas um frame. Desta forma o número de *frames* por segundo passa a depender especificamente do tempo que levou entre as duas colisões. O exemplo abaixo mostra numericamente como o efeito pode acontecer.

Supondo dois tijolos vizinhos diagonalmente, como os mostrados na figura 5.21 (tijolo 1 e tijolo 2), se a bola bate no tijolo 2 no frame x e depois no tijolo 1 no frame x+1, e ainda supondo que o tempo entre estes dois fatos foi de 70 ms, o valor de fps entre as duas colisões será de 1 frame/0,07 s = 14,28 frames/segundo. Agora supondo novamente o mesmo cenário, mas supondo que o tempo foi de 30 ms, o cálculo do número de *fps* será de 33,33 frames/segundo. Como há diversos cenários como o explicado anteriormente, há uma inevitável variação brusca dos valores obtidos para o número de *fps*. Como o valor do tempo é o tempo real entre as colisões e levando em consideração que diversos fatores influenciam neste tempo durante a execução do jogo, os valores de *frames* por segundo ao longo do número de tijolos presentes na tela sofrerão uma variação acentuada.

O ideal seria que fosse possível calcularmos um número razoável de *frames* entre as colisões para que fosse possível obter valores mais homogêneos. Contudo, para que isto fosse possível, precisaríamos alterar o comportamento original do jogo para permitir um maior número de *frames* entre colisões. Por exemplo, poderíamos fazer com que a bola saísse sempre do fundo da tela em direção ao tijolo e calcular o número de *frames* por segundo até o momento da colisão. Porém, achamos por bem manter as características originais do jogo, levando a resultados mais condizentes com a realidade.

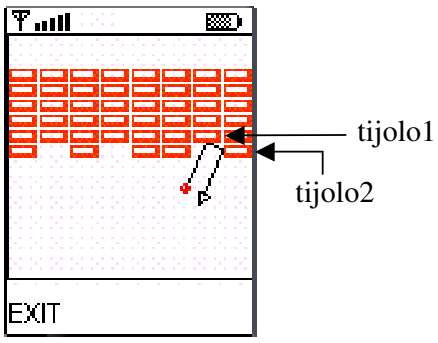


Figura 5.21: Colisão da bola com dois tijolos em *frames* consecutivos

5.3.2.2 Efeito Colateral da Interface Gráfica

Com base na característica do jogo Breakout de ter os objetos estáticos, decidimos implementar uma otimização na atualização da interface gráfica do jogo. A otimização consiste na atualização apenas dos objetos afetados a cada ciclo. A forma mais simples de atualizar os objetos de um jogo a cada ciclo é apagar a tela inteira e desenhá-los de acordo com as novas posições, para todos os objetos. A otimização para o Breakout faz com que apenas a bola (que sofre atualização contínua durante o jogo) seja atualizada, e quando houver uma colisão com algum tijolo, este também seja apagado da tela. Isto implica dizer que os outros tijolos não são atualizados, diminuindo assim o número de chamadas às funções que manipulam a interface gráfica.

A partir desta otimização, descobrimos que a variação em termos de otimização da interface gráfica é tão relevante quanto a variação em função da técnica de detecção de colisão. Não esperávamos que esse impacto fosse tão relevante. Não há relatos indicando o grau de importância da otimização da parte gráfica em relação à escolha de técnicas de detecção de colisão para jogos móveis. O fato da importância da interface gráfica não deve ser levado em consideração como algo que invalide ou diminua a importância da escolha de uma boa técnica de detecção de colisão. Isto pode ser observado como um efeito colateral positivo deste trabalho, pois determina que os desenvolvedores devem dar a devida atenção na forma como o jogo deve ser graficamente atualizado. Uma vez que isto tenha sido definido de forma otimizada, a escolha de uma boa técnica de detecção de colisão passa a ser o próximo passo em busca da melhor performance possível.

No entanto, este efeito colateral acontece quando há possibilidade de poupar chamadas às funções gráficas, o que acontece quando o jogo apresenta objetos que não variam de posição. Se o jogo apresenta uma variação quase constante dos objetos, não há como otimizar a atualização da interface gráfica, fazendo com que a escolha da técnica de detecção de colisão seja a principal forma de ganho de performance.

5.3.3 Resultados do Space Invaders

Para todos os resultados mostrados nas seções seguintes, o seguinte mapeamento dos nomes das técnicas utilizadas deve ser considerado:

- x-influ – Ordenação pelo eixo x desconsiderando-se a região de influência;
- x+influ – Ordenação pelo eixo x com região de influência;
- y-influ – Ordenação pelo eixo y desconsiderando-se a região de influência;

- y+influ – Ordenação pelo eixo y com região de influência;
- obj/obj – Padrão (*Object-to-object*);
- setor – Método do setor;
- ladr4x4 – Baseada em *tiles* (*tile* de 4x4).
- reg_influ – Região de Influência.

5.3.3.1 Métrica 1: Número de quadros por segundo

Os resultados obtidos para os níveis 1 e 2 de cada técnica implementada para o Space Invaders encontram-se nas figuras que se seguem. O tipo de gráfico é o mesmo utilizado na apresentação dos resultados do jogo Breakout.

A partir do gráfico da Figura 5.23, pode-se perceber que a técnica do setor mostrou o resultado menos significativo, onde os valores no gráfico estiveram, para quase todos os valores de número de objetos, abaixo dos apresentados pelas outras técnicas. A técnica de região de influência mostrou os resultados mais significativos para o nível 1, seguida da técnica de ordenação pelo eixo y com região de influência (y+influ) e das outras técnicas de ordenação.

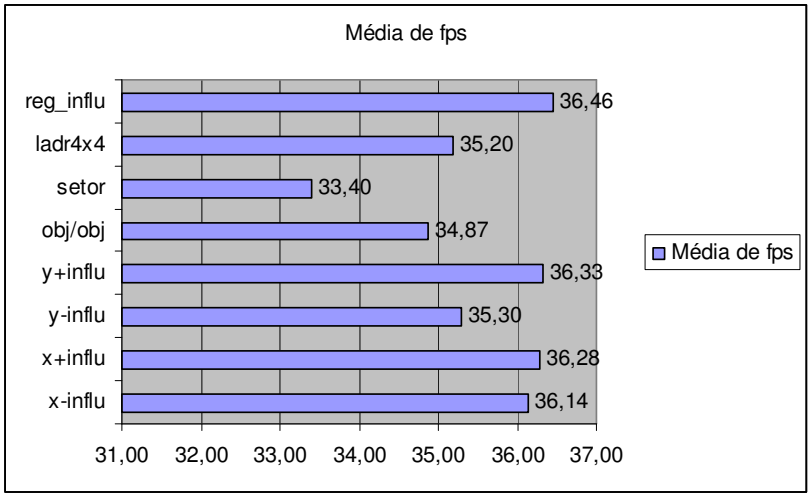


Figura 5.22: Média dos valores de fps para o nível 1 do jogo Space Invaders (executado no PC).

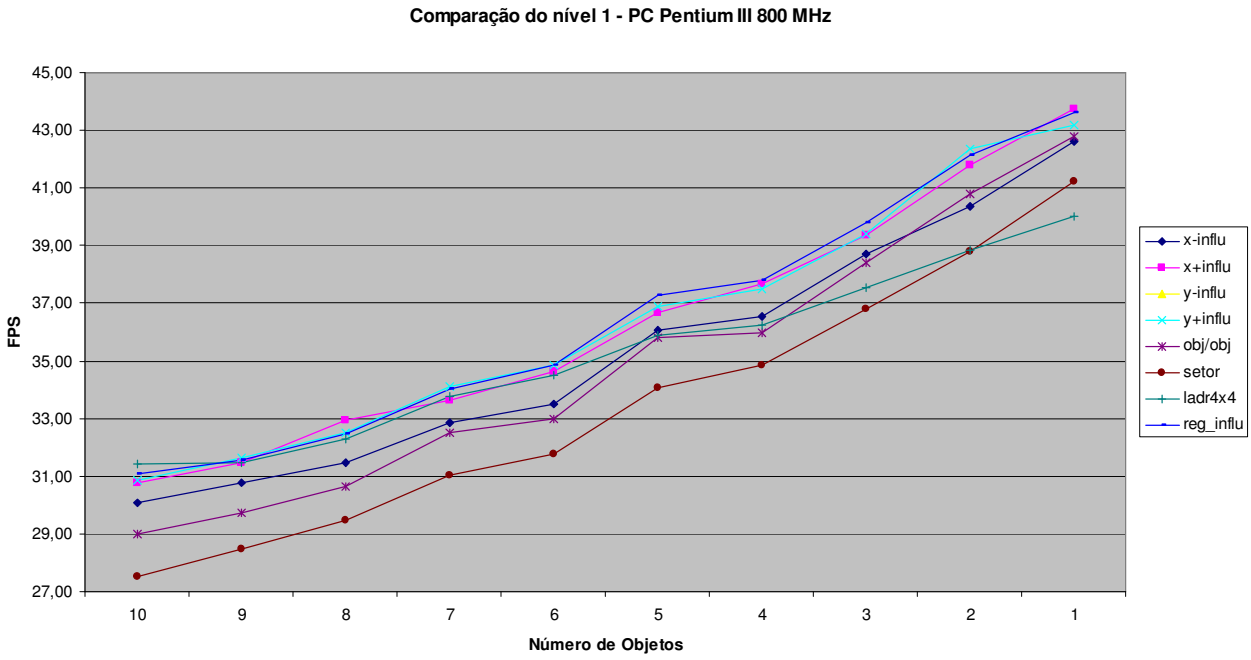


Figura 5.23: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 1 do jogo Space Invaders (executado no PC).

Para o nível 2 (Figura 5.25), o mesmo resultado do nível 1 pode ser constatado em relação ao resultado de menor valor. A técnica do setor se mostrou insatisfatória quando comparada às outras técnicas. Mostrando os resultados mais expressivos estão: a técnica de ordenação pelo eixo y levando em consideração a região de influência (y+influ); e a técnica de ordenação pelo eixo x sem levar em consideração a região de influência (x-influ). Ambas com valores bastante próximos à técnica de ordenação pelo eixo x levando em consideração a região de influência (x+influ) e à técnica de região influência (reg_influ).

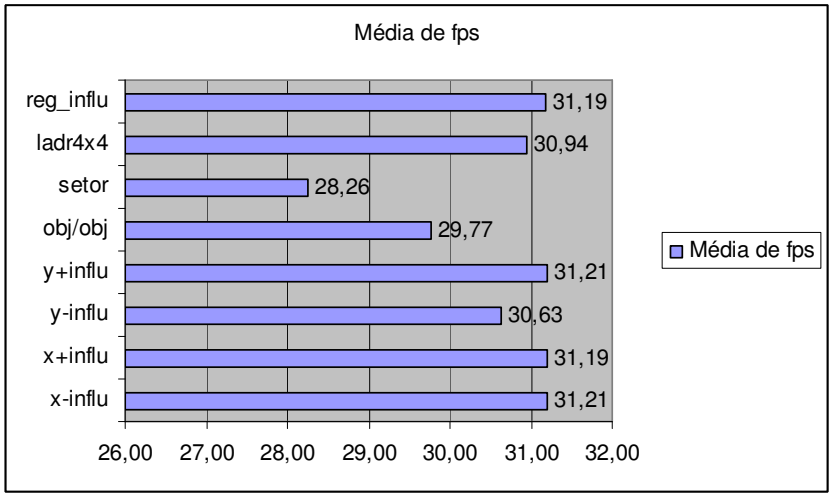


Figura 5.24: Média dos valores de fps para o nível 2 do jogo Space Invaders (executado no PC).

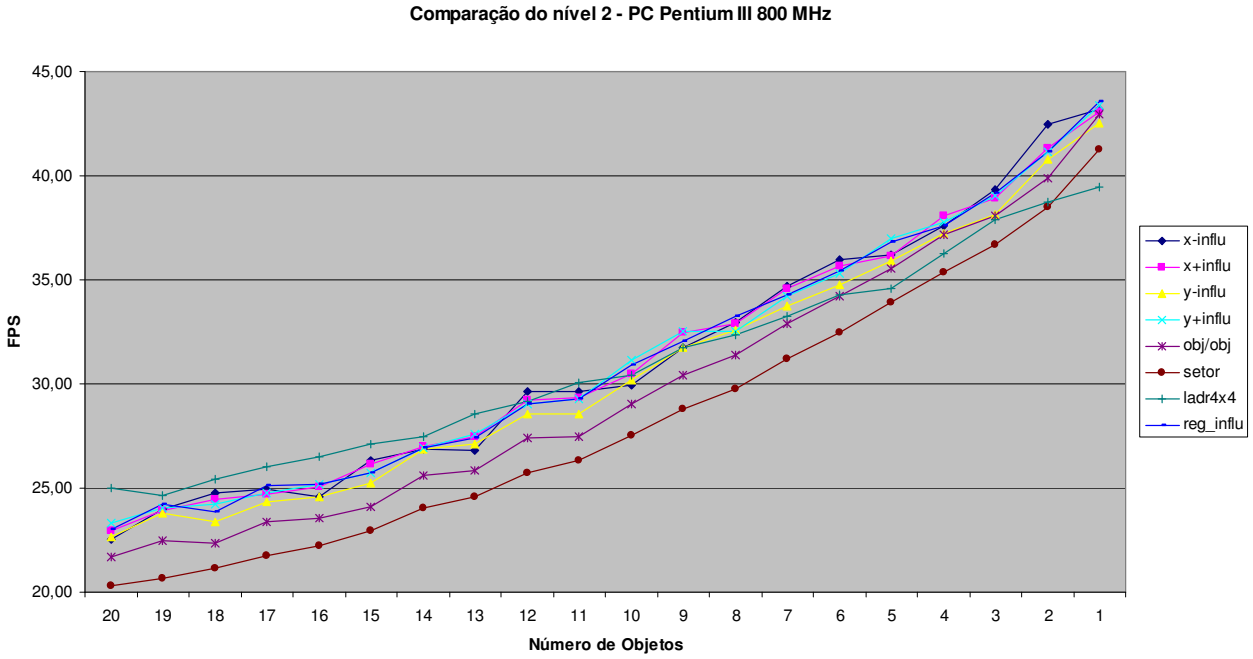


Figura 5.25: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 2 do jogo Space Invaders (executado no PC).

A técnica baseada em ladrilhos de 4x4 se mostrou bastante inferior às outras técnicas no gráfico da Figura 5.27. A técnica de região de influência apresentou, ao longo do gráfico de fps, os resultados mais significativos com valores bastante próximos aos valores de outras técnicas, como: a de ordenação pelo eixo y com região de influência (y+influ); e a de ordenação pelo eixo x com região de influência (x+influ).

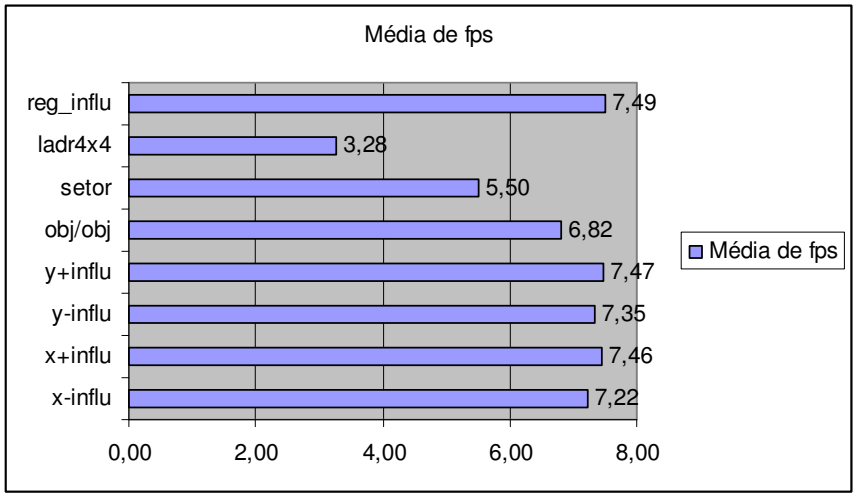


Figura 5.26: Média dos valores de fps para o nível 1 do jogo Space Invaders (executado no celular MC60).

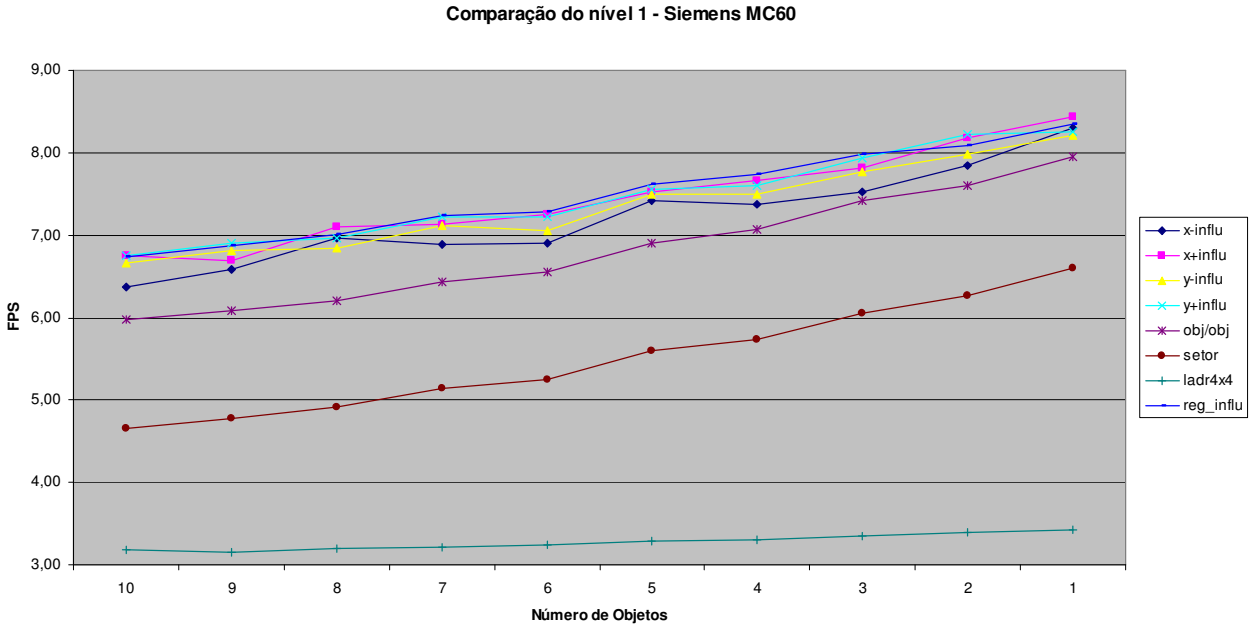


Figura 5.27: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 1 do jogo Space Invaders (executado no celular Siemens MC60).

Confirmando o que foi constatado nos resultados do nível 1, a técnica baseada em ladrilhos de 4x4 apresentou os menores valores (Figura 5.29). A técnica de ordenação pelo eixo y com região de influência apresentou os resultado mais significativos com valores bastante próximos às técnicas já mencionadas para os resultados do nível 1 (as outras de ordenação e a de região de influência).

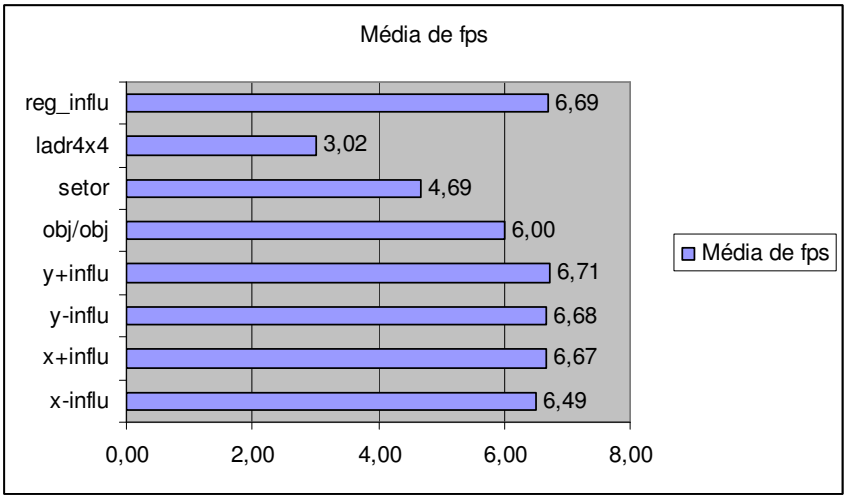


Figura 5.28: Média dos valores de fps para o nível 2 do jogo Space Invaders (executado no celular MC60).

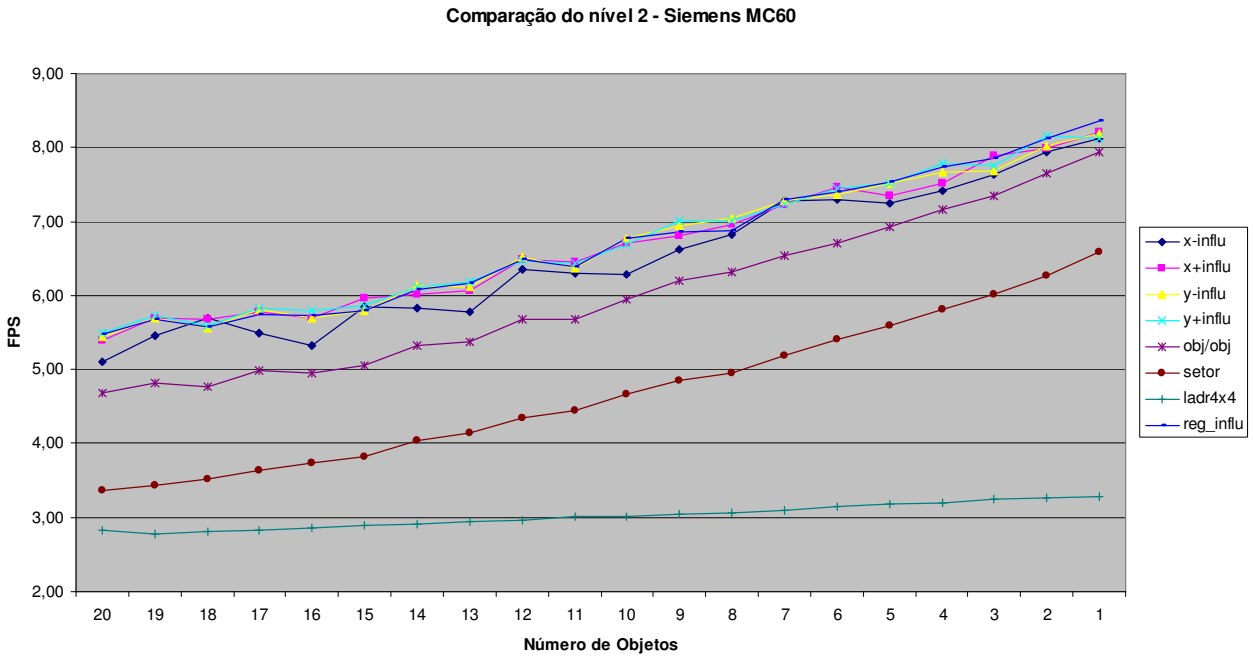


Figura 5.29: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 2 do jogo Space Invaders (executado no celular Siemens MC60).

Com base no gráfico gerado a partir dos dados obtidos do Nokia 6100 (Figura 5.31), confirma-se a ineficiência da técnica baseada em ladrilhos frente às demais técnicas. A técnica de região de influência apresenta ótimos resultados, seguida de perto pelas técnicas de ordenação.

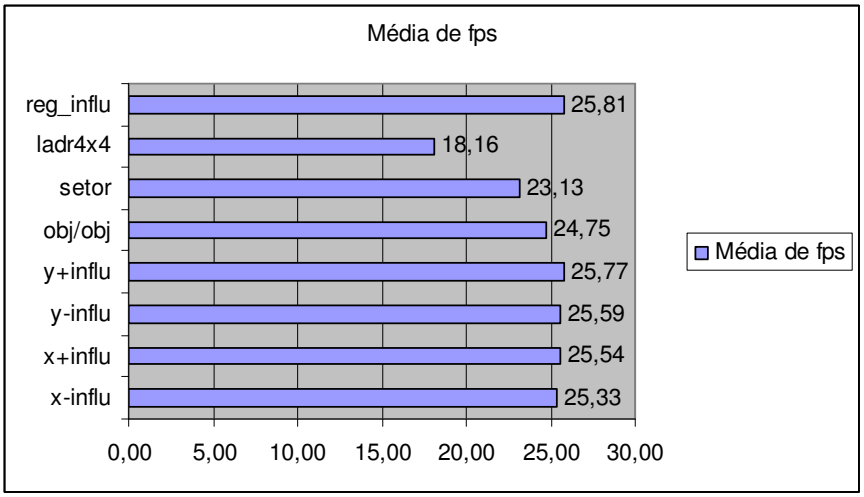


Figura 5.30: Média dos valores de fps para o nível 1 do jogo Space Invaders (executado no celular 6100).

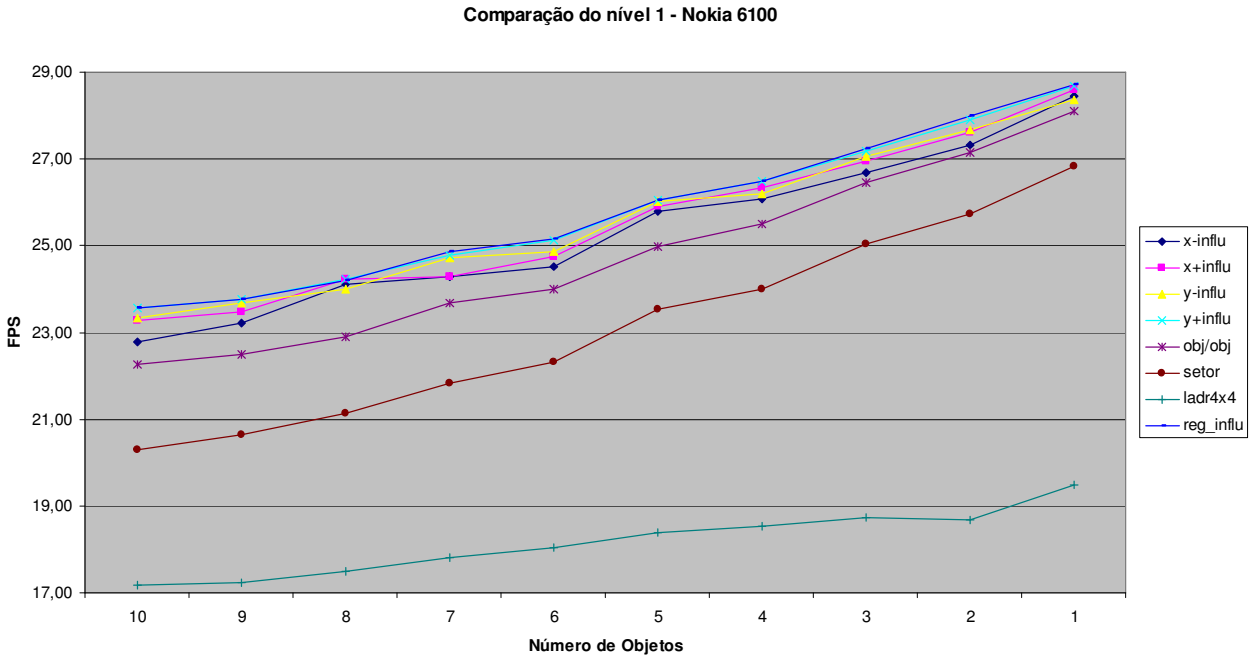


Figura 5.31: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 1 do jogo Space Invaders (executado no celular Nokia 6100).

Também para o nível 2 dos valores obtidos do Nokia 6100, os resultados apontam que a técnica baseada em ladrilhos foi menos eficiente que as outras técnicas. A técnica de ordenação pelo eixo y com região de influência (y+influ) se mostrou ligeiramente mais eficiente que a técnica de região de influência e que outras técnicas de ordenação, apresentando os resultados mais significativos.

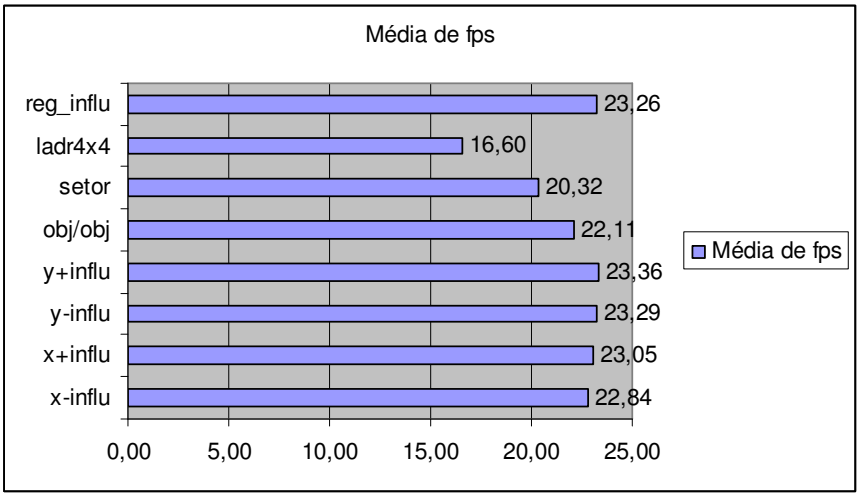


Figura 5.32: Média dos valores de fps para o nível 2 do jogo Space Invaders (executado no celular 6100).

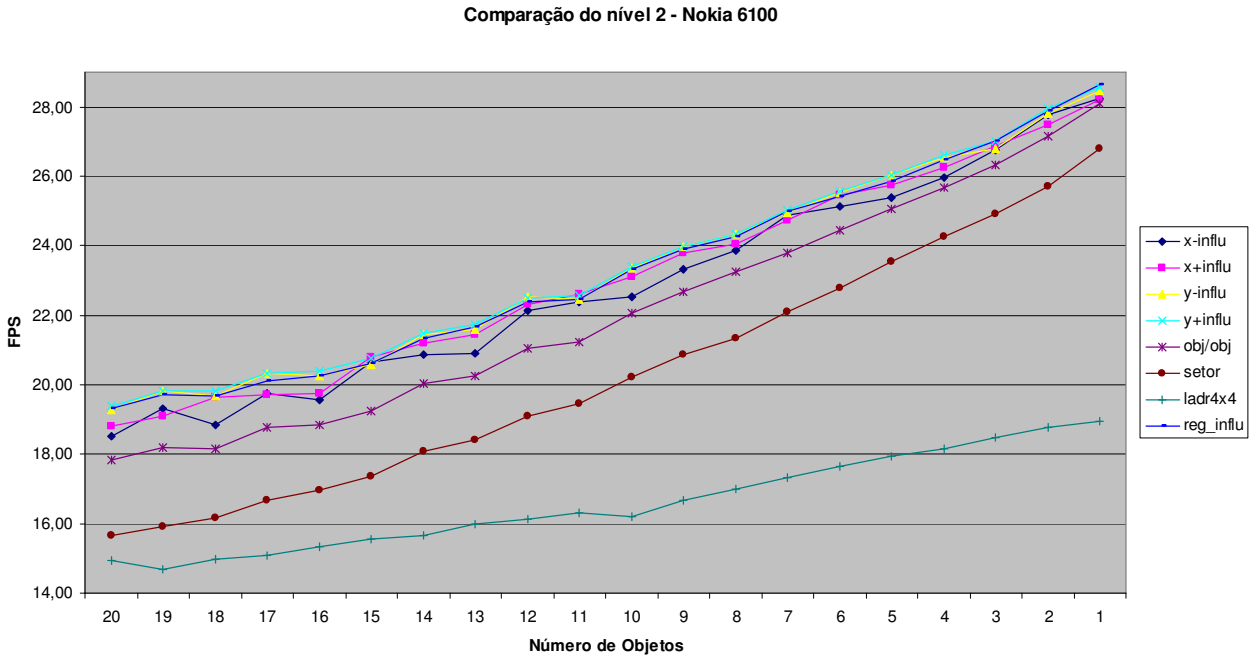


Figura 5.33: Resultado comparativo do número de quadros por segundo entre as técnicas de detecção de colisão para o nível 2 do jogo Space Invaders (executado no celular Nokia 6100).

5.3.3.2 Métrica 2: Tamanho de código

As figuras abaixo demonstram comparativamente os valores de tamanho de código obtidos para cada técnica implementada. Vale salientar que o valor não está necessariamente ligado a complexidade de implementação, porém é um bom indicativo.

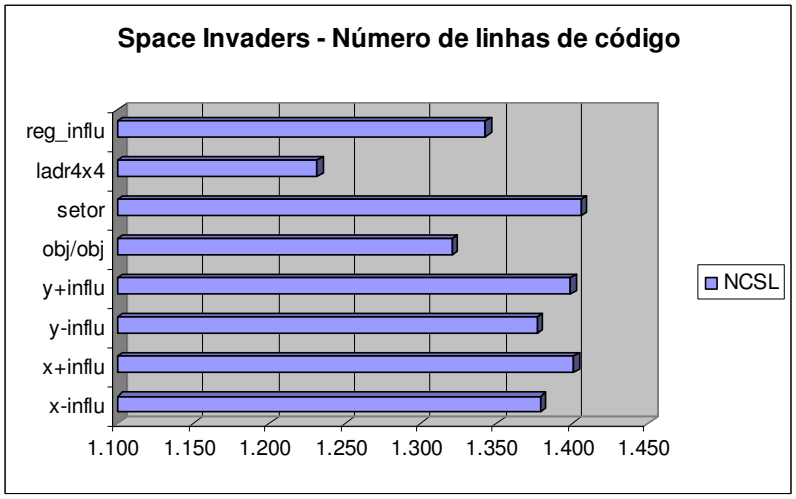


Figura 5.34: Comparação do tamanho de código para as técnicas implementadas no jogo Space Invaders.

Segundo os dados do gráfico da Figura 5.34, a técnica baseada em ladrilhos mostrou o resultado de menor valor em termos de número de linhas de código. A técnica do setor apresentou o maior número de linhas de código, com valor próximo ao das técnicas de ordenação com região de influência.

5.3.3.3 Métrica 3: Percentual de execução dos métodos relevantes

Seguem abaixo os valores obtidos com a ferramenta *profiler* do *Wireless Toolkit*.

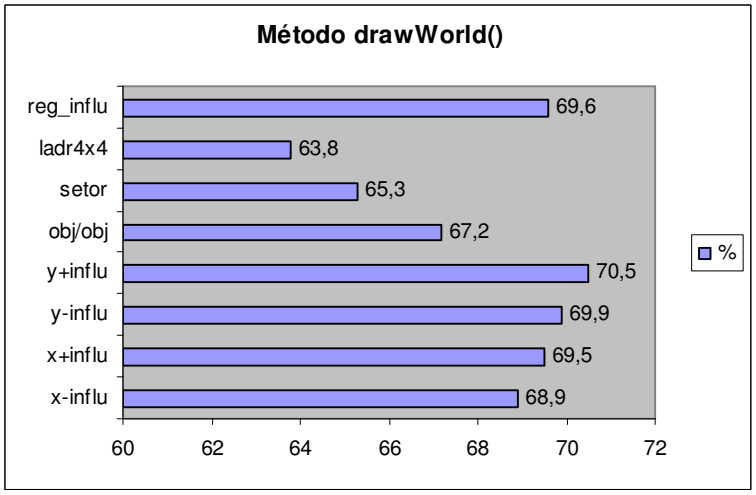


Figura 5.35: Percentual de execução do método `drawWorld`.

De acordo com o gráfico acima, a técnica baseada em ladrilhos apresentou o resultado menos expressivo. Como foi discutido na seção 5.3.1.3, este método possui praticamente a mesma implementação para todos os MIDlets, levando a conclusão que a técnica baseada em ladrilhos precisou gastar mais tempo na atualização dos objetos do

jogo. Ainda segundo os valores do gráfico, a técnica de ordenação pelo eixo y com região de influência apresentou o melhor resultado, com valor bem próximo à técnica de região de influência e às outras técnicas de ordenação.

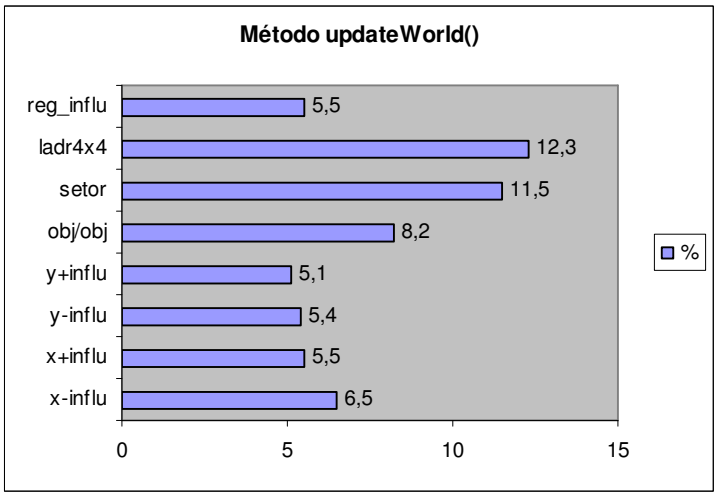


Figura 5.36: Percentual de execução do método `updateWorld`.

Com base no gráfico, confirmando o que foi dito no parágrafo anterior, pode-se perceber a ineficiência da técnica baseada em ladrilhos quando comparada com as outras, seguida de perto pela técnica do setor. Por outro lado, a técnica de ordenação pelo eixo y com região de influência apresentou uma maior eficiência em relação às outras.

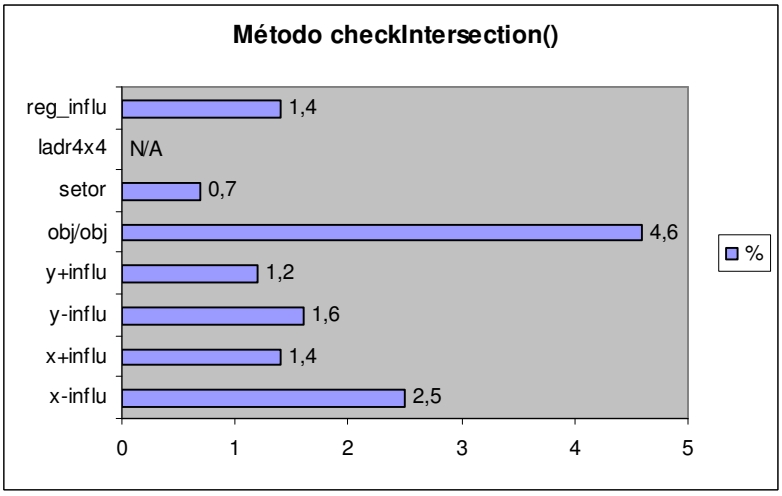


Figura 5.37: Percentual de execução do método `checkIntersection`.

Como foi discutido antes, a implementação do código responsável apenas pela detecção de colisão não foi dissociada da implementação da atualização dos objetos do

jogo na técnica baseada em ladrilho. Desta forma, o resultado para esta técnica não pode ser levado em consideração comparativamente. Desconsiderando-se esta técnica, o método do setor apresentou o resultado mais significativo como era de se esperar, pois a carga computacional está na atualização dos objetos em função dos setores. Uma vez que cada objeto esteja identificado no devido setor, a verificação de colisão é bastante simples, bastando comparar os objetos dentro do mesmo setor. Isto é bastante eficiente no caso do Space Invaders, dado que os objetos estão dispostos de forma dispersa na tela diminuindo bastante o número de testes de colisão. A técnica que apresentou o resultado mais expressivo depois do método do setor foi a técnica de ordenação pelo eixo x com região de influência.

5.3.4 Análise dos Resultados para o Jogo Space Invaders

Assim como na análise dos resultados do jogo Breakout, a hipótese do experimento não foi sustentada também para o jogo Space Invaders. As variações nos resultados entre as implementações das diferentes técnicas mostram que algumas técnicas são melhores que outras em função das métricas levadas em consideração.

As técnicas de ordenação por eixo e a técnica de região de influência apresentaram os melhores resultados de *fps*, na média. Tanto para o nível 1 quanto para o nível 2. Além dos melhores resultados, no geral, na métrica de performance dos métodos relevantes. A técnica de ordenação pelo eixo y com região de influência (y+influ) se mostrou a técnica de maior eficiência no geral com resultado um pouco melhor que as outras técnicas de ordenação e que a técnica de região de influência. Com base no gráfico do número de linhas de código, pode-se perceber que foi necessário acrescentar uma certa complexidade para implementar as técnicas de ordenação, mas o resultado final em função disto é satisfatório.

A técnica baseada em ladrilhos apresentou os resultados menos expressivos no geral. Apesar de não ter demonstrado grandes problemas no número de quadros por segundo para os testes executados no emulador em PC, os resultados foram contundentes na execução do experimento nos celulares. Isto, de certa forma, já era esperado, pois o Space Invaders não é um jogo com características intrínsecas para implementação com ladrilhos. Os resultados mostraram que foi necessário gastar mais tempo com processamento para atualizar os objetos que as outras técnicas, degradando a performance final do jogo.

Os resultados também mostraram que o método do setor não é uma técnica tão boa quando comparada às técnicas de ordenação e de região de influência. A partir dos resultados do método `updateWorld`, pode-se verificar que o custo computacional para a atualização dos objetos só perde para a técnica baseada em ladrilhos. Em conseqüência disto, a taxa de *fps* mostrou resultados pouco satisfatórios para esta técnica. Isto se deve principalmente à necessidade de atualização dos objetos em função dos setores a cada iteração do jogo. Apesar do bom resultado para o método específico de detecção de colisão, esta técnica mostrou uma performance inferior às outras técnicas no geral.

Os resultados dos gráficos de *fps* mostraram valores com incremento uniforme em função do número de objetos, indicando um aumento de performance à medida que o número de objetos diminui. Isto é verdade para todas as técnicas analisadas e para os dois níveis. O aumento de performance inversamente proporcional ao número de objetos também se mostrou inversamente proporcional para os dois níveis. Isto é, o número de quadros por segundo verificado no nível 2 (vinte objetos) quando uma dada técnica chega ao décimo objeto é praticamente o mesmo mostrado pela mesma técnica no início do nível 1 (quando os dez objetos ainda estão presentes na tela), mantendo a proporção do valor de *fps* em função do número de objetos. Assim como no Breakout, os valores obtidos pelo Nokia 6100 foram mais significativos que os obtidos pelo Siemens MC60. Exceto pelos valores absolutos de *fps*, não houve maiores diferenças entre os resultados apresentados pelo emulador e pelos celulares.

De acordo com o que foi explicado na análise dos resultados do Breakout, não implementamos qualquer otimização na atualização da interface gráfica do Space Invaders. Isto porque todos os objetos do jogo são atualizados de posição a cada ciclo do jogo. A tela é redesenhada por completo a cada ciclo. Desta forma, a detecção de colisão empenha um importante papel na melhoria de performance do jogo.

5.4 Conclusão do Capítulo

O principal objetivo desta dissertação é prover uma avaliação comparativa entre as diferentes técnicas de detecção de colisão escolhidas. Para que isto seja viável e principalmente confiável, idealmente apenas o código que reflete a detecção de colisão em si deve variar entre as implementações das diferentes técnicas nos dois jogos levados em consideração. Para tal, foi utilizada uma API com os seguintes métodos:

drawWorld, updateWorld, loadWorld e isGameOver. Apenas as implementações destes métodos variam para as diferentes técnicas. Idealmente, apenas alguns métodos que envolvem a detecção de colisão em si deveriam ser diferentes. No entanto, a implementação da técnica baseada em ladrilhos forçou a necessidade de uma API que permitisse a variação na implementação da forma como o mundo do jogo é construído.

Levando isto em consideração, os dois jogos foram estruturados para garantir o máximo de código comum entre as implementações das diferentes técnicas para garantir a confiabilidade dos resultados.

Como já foi mencionado anteriormente, um dos alicerces da experimentação é a viabilidade de replicação do mesmo, permitindo que outros consigam chegar aos mesmos resultados obtidos inicialmente. Para que isto fosse possível, foi necessário explicar todo o procedimento para replicação do experimento de tal forma que fosse possível chegar aos mesmos resultados obtidos quando de sua primeira execução.

Por meio dos resultados obtidos, atingimos um dos objetivos da dissertação que era prover uma avaliação comparativa entre as diferentes técnicas de detecção de colisão levadas em consideração. Esta avaliação permitirá que outros desenvolvedores possam se basear para a melhor escolha da técnica de detecção em suas aplicações ou permitir-lhes fazer um estudo comparativo semelhante levando em consideração outra possíveis técnicas de detecção de colisão ou ainda outros possíveis jogos com características distintas.

6 Conclusões

Neste capítulo são apresentadas as contribuições deste trabalho, as principais dificuldades encontradas durante seu desenvolvimento e algumas considerações para trabalhos futuros.

6.1 Contribuições

Com base na metodologia utilizada e nos resultados do experimento, podemos afirmar que os objetivos iniciais desta dissertação foram alcançados. A principal contribuição foi a iniciativa de um estudo pioneiro experimental sobre um tema atualmente relevante (Detecção de Colisão para jogos móveis), onde há uma demanda considerável por pesquisa, como foi mencionado no início deste trabalho.

Além disto, os resultados obtidos a partir deste trabalho podem servir como um guia para que desenvolvedores de jogos móveis possam se basear na escolha de uma determinada técnica (ou mesmo uma solução híbrida) de detecção de colisão para jogos que possuam características semelhantes às dos jogos aqui analisados. Além também de permitir que este mesmo trabalho seja estendido por outras pessoas interessadas no intuito de se adequar as suas necessidades. Isto caracteriza uma contribuição não menos importante que a citada anteriormente. Vale salientar que algumas das técnicas foram propostas por nós, não havendo relato similar na literatura, o que valoriza ainda mais como contribuição à comunidade de desenvolvedores.

Com base no estudo desenvolvido sobre experimentação (Seção 3), outras pessoas que necessitam desenvolver um trabalho que requer, de alguma maneira, um estudo empírico podem utilizar este trabalho para servir de base para um processo experimental. Isto contribui diretamente na disseminação do estudo sobre engenharia de software empírica.

Outra contribuição também relevante foi o fato dos resultados mostrarem uma grande importância para o processamento da parte gráfica em um jogo J2ME. Isto implica dizer que o desenvolvedor deve dar a devida atenção no momento de implementar o código que atualiza a interface gráfica, pois o ganho de performance na escolha de uma boa solução de detecção de colisão para um jogo pode ser ínfimo se o processamento da interface gráfica for negligenciado.

6.2 Dificuldades encontradas

Esta seção mostra as principais dificuldades e obstáculos encontrados durante o desenvolvimento deste trabalho, além de algumas críticas.

A API do MIDP 1.0 não disponibiliza um método que mede o tempo em microssegundos, apenas em milisegundos (usando-se o método `currentTimeMillis` da classe `System`). Desta forma, não foi possível medir o tempo gasto dentro dos métodos que detectam colisão, pois estes executam em microssegundos, o que é uma perda de uma excelente métrica utilizada em alguns trabalhos do gênero [15][17].

Um problema encontrado em relação à métrica de performance dos métodos relevantes é que o *profiler* do *Wireless Toolkit* leva em consideração não apenas o código da aplicação, mas também o código do próprio WTK. Desta forma, foi necessário adotar um padrão para a execução do teste de maneira uniforme. Isto é, a execução do MIDlet deveria começar de um determinado ponto (de maneira uniforme entre as técnicas) e terminar sua execução num outro ponto também comum entre as execuções das implementações. Como os MIDlets iniciam sua execução de forma automática, o ponto inicial de execução é o mesmo para todas as implementações. Para o ponto final determinamos que este seria o aparecimento da tela de resultados na aplicação, ou seja, quando esta tela aparece no simulador, o responsável pelo teste deve fechar o simulador naquele exato momento para parar sua execução, habilitando assim os resultados do *profiler*. Caso não fosse imposto este controle, os resultados seriam afetados, já que o código do simulador também é levado em consideração pelo *profiler*, distorcendo o resultado final.

Outro obstáculo encontrado e que falamos previamente é o grande impacto do método de atualização gráfica do MIDP (o `paint` da classe `Canvas`) na performance de execução, que não invalida a aplicabilidade dos resultados deste trabalho. Podemos até considerar este obstáculo como um efeito colateral positivo deste trabalho, pois confirma a relevância que tem a parte gráfica no desenvolvimento de um jogo móvel.

Mais um exemplo de dificuldade que enfrentamos foi a possibilidade de uma grande variação dos resultados no cálculo do número de quadros por segundo no simulador em função da performance do PC. Se o PC tiver uma performance de processamento muito elevada, os valores entre colisões (principalmente no caso do

Breakout) podem apresentar grandes variações, deteriorando os resultados desta métrica.

A configuração final para a correta execução dos experimentos também necessitou de grande esforço. Foi preciso empenhar bastante tempo para projetar e implementar os MIDlets de tal forma que o código comum fosse maximizado para permitir uma comparação justa entre as técnicas. Também foi preciso empenhar um certo esforço para refinar os algoritmos de detecção de colisão ao longo do desenvolvimento. Alguns ajustes nos algoritmos foram feitos com base em execução e reavaliação. Considerando que foram 3 métricas, 2 jogos, 18 MIDlets no total e 3 ambientes de teste (emulador e 2 celulares), foi necessário gastar um tempo razoável para rodar os experimentos, obter os resultados e processá-los para que fosse possível fazer uma análise. Além disso, também não foi fácil conseguir os celulares para executar os experimentos.

6.3 Trabalhos futuros

Esta seção cobre alguns aspectos que podem ser levados em consideração para a extensão deste trabalho.

6.3.1 A Evolução do J2ME (MIDP 2.0)

Na época que decidimos fazer este trabalho, o MIDP já estava sendo amplamente utilizado em boa parte do mundo na sua versão 1.0 [25][31], mas sua utilização no Brasil era modesta e quase inexistente. Nesta mesma época, a versão 2.0 da API [26][32] já estava em sua fase de testes e expansão nos países tecnologicamente mais avançados, principalmente em países da Ásia, como o Japão e a Coréia do Sul. Durante o desenvolvimento da dissertação, a versão 2.0 já estava sendo suportada por uma vasta gama de celulares e outros dispositivos em vários países do mundo, mas estes aparelhos ainda não estavam disponíveis em larga escala no Brasil, época em que a versão inicial do MIDP estava predominando aqui. Atualmente, tanto no Brasil como em diversos países do mundo, as operadoras de telefonia móvel já disponibilizam celulares que suportam a versão 2.0 do MIDP.

A versão 2.0 traz algumas melhorias importantes, como suporte a multimídia, interface com usuário melhorada, novos protocolos de conectividade e um melhor sistema de segurança baseado em assinatura de MIDlets [24]. Outra melhoria foi a

adição de um pacote nativo de suporte ao desenvolvimento de jogos [23][22], visto que este tipo de aplicação foi um dos mais amplamente desenvolvidos por diversos desenvolvedores autônomos e empresas ao redor do mundo, seguido de outros tipos de aplicações como as que fazem uso da mobilidade do celular, como aplicações de mapas e localização geográfica baseado em GPS. A nova API de jogos auxilia os desenvolvedores deste tipo de aplicação a desenvolver interfaces mais rapidamente e com uma melhor usabilidade, auxiliando a economizar recursos do dispositivo, como memória. A API também ajuda a reduzir o tamanho final do arquivo JAR gerado. Com MIDP 1.0, os desenvolvedores de jogos tinham que fazer suas próprias rotinas gráficas para obter uma melhor performance. Isto levava a uma desvantagem do aumento do tamanho do código gerado, e conseqüentemente do tamanho do arquivo JAR, além de possivelmente levar a um código mais "pobre". Levar o código destas rotinas para dentro do MIDP faz com que os arquivos JAR diminuam de tamanho, já que as mesmas passam a fazer parte do MIDP, além de disponibilizar aos desenvolvedores rotinas de boa qualidade (em termos de implementação). Esta API de jogos disponibiliza classes que implementam conceitos básicos de desenvolvimento de jogos, como *sprites*, camadas e ladrilhos, e facilitam a construção conceitual do projeto do jogo, além de facilitar o seu desenvolvimento. A idéia básica da API é que os jogos consistem de camadas (*layers*). O plano de fundo do jogo poderia estar em uma camada, enquanto a estrada poderia estar em outra, e a nave do herói poderia fazer parte de outra camada (como um *sprite*) (ver seção 4.2). Em vários jogos, a área de jogabilidade é freqüentemente maior que o tamanho da tela. Movimentar-se por esta área deslizando a tela pode não ser um trabalho de codificação tão simples. A API de jogos provê uma janela de visão, que possibilita o controle de toda essa área de jogabilidade de maneira simplificada.

A API pode ser encontrada no pacote `javax.microedition.lcdui.game` e possui cinco novas classes, são elas: GameCanvas, Layer, LayerManager, Sprite e TiledLayer. A classe GameCanvas é uma classe abstrata que provê a base para interface do jogo. Esta classe tem dois benefícios sobre a classe Canvas, ela possui um buffer off-screen, e ela possui a funcionalidade de capturar o estado da tecla do dispositivo. A classe Layer é uma classe abstrata que representa um objeto do jogo, como um NPC, por exemplo. Sprite e TiledLayer são classes herdadas de Layer. A classe LayerManager gerencia diversas subclasses de Layer e as imprime na tela em uma ordem específica. Uma Sprite é um Layer que pode conter diversos quadros

armazenados em uma Image. Com Sprite, é possível se usar partes de uma imagem como quadros e fazer uma seqüência de quadros para criar uma animação. Outra funcionalidade da classe Sprite é que ela possui a funcionalidade de checar colisão entre ela e outras Sprites ou TiledLayers. A classe TiledLayer é um pouco parecida com a classe Sprite, mas é principalmente usada para modelar planos de fundo (ou backgrounds), estradas ou outro tipo similar de área ampla. TiledLayer consiste de uma grade de células, que pode ser preenchida com imagens ou ladrilhos. Desta forma, o plano de fundo ou cenário é construído de pequenas imagens.

São dois os principais fatores que têm impacto sobre este trabalho com o surgimento desta nova versão da API:

- a adição de suporte a jogos, que sugere que os MIDlets utilizados neste trabalho sejam reestruturados;
- a disponibilidade de uma função que lê os *pixels* de uma imagem.

Este último item tem uma relevância muito grande em relação a este trabalho, pois, como foi dito anteriormente, o tipo de detecção baseado em *pixel* não foi levado em consideração por não haver um método que pudesse ler os *pixels* de uma imagem para o MIDP 1.0. Com a inclusão deste método (chamado `getRGB` e disponibilizado na classe `Image`), o trabalho pode ser estendido para levar em consideração este tipo de detecção de colisão. Desta forma, as técnicas utilizadas neste trabalho podem ser reimplementadas utilizando-se a leitura de *pixels* das imagens para uma detecção de colisão mais precisa, possibilitando assim uma avaliação das técnicas implementadas com e sem leitura de *pixel*. Exceto as técnicas baseadas em ladrilhos, que usam naturalmente a verificação *bounding box*, não permitindo a implementação de verificação baseada em *pixel*. Isto possibilita saber se vale à pena ou não agregar esse tipo de detecção (com um conseqüente refinamento da verificação de colisão) para uma determinada técnica, ou ainda para qual (ou quais) das técnicas o custo-benefício é suficiente considerando-se este refinamento da detecção.

6.3.2 Outras considerações

Uma extensão implicitamente definida neste trabalho é a ampliação dos resultados, ou seja, a inclusão de novas técnicas de detecção de colisão e a inclusão de outros jogos com características distintas das dos jogos que foram implementados neste trabalho.

Outra extensão em potencial é definir outras métricas para consolidar de forma mais ampla os resultados. Por exemplo, o trabalho de Evin Levey [16] propõe algumas métricas para avaliação de técnicas de detecção de colisão, as quais podemos citar: performance, escalabilidade e facilidade de implementação.

A performance já foi explorada por meio do número de quadros por segundo, mas a escalabilidade e a facilidade de implementação são métricas que poderiam ser levadas em consideração para tornar o trabalho mais amplo. No entanto, algum critério mínimo deve ser definido para uma análise em função da facilidade de implementação, pois este é bastante subjetivo.

A qualidade gráfica também poderia ser utilizada como parâmetro de análise comparativa. A técnica baseada em ladrilho (não híbrida), por exemplo, deixa o jogo com movimentos mais discretos, prejudicando a jogabilidade. Isto também é um fator importante na escolha final da técnica.

Apêndice A: Funções de Detecção de Colisão (Breakout)

A.1 Detecção padrão

Implementação do método que detecta colisão para a técnica: “Detecção Padrão”.

```
public void checkCollision()
{
    boolean intersection = false;

    for (int i = 0; i < this.bricks.length && !intersection; i++) {
        for (int j = 0; j < this.bricks[0].length && !intersection;
j++)
        {
            if (this.bricks[i][j].isVisible() &&
                this.intersects(this.bricks[i][j]))
            {
                this.takeAction(bricks[i][j]);
                intersection = true;
            }
        }
    }
}
```

A.2 Região de influência

Implementação do método que detecta colisão para a técnica: “Região de Influência”.

```
public void checkCollision()
{
    if (isBallInBricksBox()) {
        boolean intersection = false;

        for (int i = 0; i < this.bricks.length && !intersection; i++)
{
            for (int j=0; j<this.bricks[0].length && !intersection;
j++)
            {
                if (this.bricks[i][j].isVisible() &&
                    this.intersects(this.bricks[i][j]))
                {
                    this.takeAction(bricks[i][j]);
                    intersection = true;
                }
            }
        }
    }
}
```

A.3 Ordenação pelo eixo x desconsiderando a região de influência

Implementação do método que detecta colisão para a técnica: “Ordenação pelo eixo x sem levar em consideração a região de influência”.

```
public void checkCollision()
{
    boolean end = false;
    boolean intersection = false;
    Brick brick;

    for (int i = 0; i < this.sortedBricks.size()
                    && !end && !intersection; i++)
    {
        brick = (Brick)this.sortedBricks.elementAt(i);

        if (brick.getXPos() >
            (this.ballXPosition + Constants.ballImg.getWidth()))
        {
            end = true;
        } else {
            if (this.intersects(brick)) {
                this.takeAction(brick);
                this.sortedBricks.removeElementAt(i);
                intersection = true;
            }
        }
    }
}
```

A.4 Ordenação pelo eixo x com região de influência

Implementação do método que detecta colisão para a técnica: “Ordenação pelo eixo x com região de influência”.

```
public void checkCollision()
{
    if (this.isBallInBricksBox()) {
        boolean end = false;
        boolean intersection = false;
        Brick brick;

        for (int i = 0; i < this.sortedBricks.size()
                        && !end && !intersection; i++)
        {
            brick = (Brick)this.sortedBricks.elementAt(i);

            if (brick.getXPos() >
                (this.ballXPosition + Constants.ballImg.getWidth()))
            {
                end = true;
            } else {
                if (this.intersects(brick)) {
                    this.takeAction(brick);
                    this.sortedBricks.removeElementAt(i);
                    intersection = true;
                }
            }
```

```
        }
    }
}
```

A.5 Ordenação pelo eixo y desconsiderando a região de influência

Implementação do método que detecta colisão para a técnica: “Ordenação pelo eixo y sem levar em consideração a região de influência”.

```
public void checkCollision()
{
    boolean end = false;
    boolean intersection = false;
    Brick brick;

    for (int i = 0; i < this.sortedBricks.size()
                    && !end && !intersection; i++)
    {
        brick = (Brick)this.sortedBricks.elementAt(i);

        if ((brick.getYPos() + brick.getHeight()) <
            this.ballYPosition)
        {
            end = true;
        } else {
            if (this.intersects(brick)) {
                this.takeAction(brick);
                this.sortedBricks.removeElementAt(i);
                intersection = true;
            }
        }
    }
}
```

A.6 Ordenação pelo eixo y com região de influência

Implementação do método que detecta colisão para a técnica: “Ordenação pelo eixo y com região de influência”.

```
public void checkCollision()
{
    if (this.isBallInBricksBox()) {
        boolean end = false;
        boolean intersection = false;
        Brick brick;

        for (int i = 0; i < this.sortedBricks.size()
                        && !end && !intersection; i++)
        {
            brick = (Brick)this.sortedBricks.elementAt(i);

            if ((brick.getYPos() + brick.getHeight()) <
                this.ballYPosition)
            {
                end = true;
            } else {
                if (this.intersects(brick)) {
                    this.takeAction(brick);
```

```
                    this.sortedBricks.removeElementAt(i);
                    intersection = true;
                }
            }
        }
    }
}
```

A.7 Gerenciador de objetos (com quatro objetos por gerenciador)

Implementação do método que detecta colisão para a técnica: “Gerenciador de objetos (com quatro objetos por gerenciador)”.

```
public void checkCollision()
{
    if (isBallInBricksBox()) {
        boolean intersection = false;
        int intersectedOM = -1;
        // verifies which ObjectManager is being affected
        for (int i = 0; i < BreakOutWorld.num_obj_mang &&
             intersectedOM == -1; i++)
        {
            if (this.objectManagers[i].
                    intersects(this.ballXPosition,
                               this.ballYPosition,
                               Constants.ballImg.getWidth(),
                               Constants.ballImg.getHeight()))
            {
                intersectedOM = i;
            }
        }
        if (intersectedOM != -1) {
            Brick[] bricks =
                this.objectManagers[intersectedOM].getBricks();
            // verifies which brick (from the affected manager)
            // is colliding with the ball
            for (int i = 0; i < bricks.length && !intersection; i++) {
                if (bricks[i].isVisible() &&
this.intersects(bricks[i]))
                {
                    this.takeAction(bricks[i]);
                    intersection = true;
                }
            }
        }
    }
}
```

A.8 Gerenciador de objetos (com oito objetos por gerenciador)

Implementação do método que detecta colisão para a técnica: “Gerenciador de objetos (com oito objetos por gerenciador)”.

```
public void checkCollision()
{
    if (isBallInBricksBox()) {
        boolean intersection = false;
        int intersectedOM = -1;
        // verifies which ObjectManager is being affected
        for (int i = 0; i < BreakOutWorld.num_obj_mang &&
```

```
             intersectedOM == -1; i++)
        {
            if (this.objectManagers[i].
                    intersects(this.ballXPosition,
                               this.ballYPosition,
                               Constants.ballImg.getWidth(),
                               Constants.ballImg.getHeight()))
            {
                intersectedOM = i;
            }
        }
        if (intersectedOM != -1) {
            Brick[] bricks =
                this.objectManagers[intersectedOM].getBricks();
            // verifies which brick (from the affected manager)
            // is colliding with the ball
            for (int i = 0; i < bricks.length && !intersection; i++) {
                if (bricks[i].isVisible() &&
this.intersects(bricks[i]))
                {
                    this.takeAction(bricks[i]);
                    intersection = true;
                }
            }
        }
    }
}
```

A.9 Baseada em ladrilhos pequenos (6 x 4)

Implementação do método que detecta colisão para a técnica: “Baseada em ladrilhos pequenos”.

```
/**
 * Remove um determinado tijolo do jogo (destroi)
 */
private void removeBrick(byte xTile, byte yTile)
{
    BreakOutWorld.map[yTile][xTile] = BreakOutWorld.EMPTY_TILE;
    this.brickYTileToDeleted = yTile;

    if ((xTile % 3) == 0) {
        BreakOutWorld.map[yTile][xTile + 1] =
        BreakOutWorld.map[yTile][xTile + 2] =
            BreakOutWorld.EMPTY_TILE;
        this.brickXTileToDeleted = xTile;
    } else if ((xTile % 3) == 1) {
        BreakOutWorld.map[yTile][xTile - 1] =
        BreakOutWorld.map[yTile][xTile + 1] =
            BreakOutWorld.EMPTY_TILE;
        this.brickXTileToDeleted = (byte)(xTile - 1);
    } else {
        BreakOutWorld.map[yTile][xTile - 1] =
        BreakOutWorld.map[yTile][xTile - 2] =
            BreakOutWorld.EMPTY_TILE;
        this.brickXTileToDeleted = (byte)(xTile - 2);
    }
}

/**
 * Atualiza as coordenadas verticais dos objetos
```

```
 */
private void updateVertBallCoord()
{
    if (this.ballYTile - 1 < 0) { // collision with left wall
        this.ballYSpeed *= -1;
        this.ballYTile = 1;
    } else if (this.ballYTile + 1 > BreakOutWorld.map.length - 1) {
        this.ballYSpeed *= -1;
        this.ballYTile = (byte)(BreakOutWorld.map.length - 2);
    } else {
        this.ballYTile += this.ballYSpeed;

        if (BreakOutWorld.map[this.ballYTile][this.ballXTile] ==
            BreakOutWorld.BRICK_TILE)
        {
            this.removeBrick(this.ballXTile, this.ballYTile);
            this.ballYSpeed *= -1;
            this.ballYTile += this.ballYSpeed;
            Constants.wasCollision = true;
        }
    }
}

/**
 * Atualiza as coordenadas horizontais dos objetos
 */
private void updateHorizBallCoord()
{
    if (this.ballXTile - 1 < 0) { // collision with left wall
        this.ballXSpeed *= -1;
        this.ballXTile = 1;
    } else if (this.ballXTile + 1 > BreakOutWorld.map[0].length - 1) {
        this.ballXSpeed *= -1;
        this.ballXTile = (byte)(BreakOutWorld.map[0].length - 2);
    } else {
        this.ballXTile += this.ballXSpeed;

        if (BreakOutWorld.map[this.ballYTile][this.ballXTile] ==
            BreakOutWorld.BRICK_TILE)
        {
            this.removeBrick(this.ballXTile, this.ballYTile);
            this.ballXSpeed *= -1;
            this.ballXTile += this.ballXSpeed;
            Constants.wasCollision = true;
        }
    }
}

/**
 * Atualiza os objetos do jogo
 */
public void updateWorld()
{
    this.lastBallXPos = this.ballXRPosition;
    this.lastBallYPos = this.ballYRPosition;
    this.brickXTileToDeleted = this.brickYTileToDeleted = -1;

    if (this.speed_control == 1) {
        this.speed_control = 0;

        this.updateVertBallCoord();
        if (!Constants.wasCollision)
            this.updateHorizBallCoord();
```

```
    } else {
        this.speed_control++;
    }

    this.ballXRPosition = (byte)this.getRealX(this.ballXTile);
    this.ballYRPosition = (byte)this.getRealY(this.ballYTile);
}
```

A.10 Baseada em ladrilhos grandes (6 x 12)

Implementação do método que detecta colisão para a técnica: "Baseada em ladrilhos grandes".

```
/**
 * Destroi um tijolo. Substitui o ladrilho "Tijolo" pelo ladrilho
 * "Vazio".
 */
private void destroyBrick(byte yTile, byte xTile)
{
    BreakOutWorld.map[yTile][xTile] = BreakOutWorld.EMPTY_TILE;
    this.brickXTileToDeleted = xTile;
    this.brickYTileToDeleted = yTile;
    Constants.wasCollision = true;
}

/**
 * Verifica colisao entre objetos
 */
private void verifyCollision(byte position)
{
    switch(position) {
        case 0:
            if (BreakOutWorld.map[this.ballYTile - 1][this.ballXTile]
!=
                BreakOutWorld.EMPTY_TILE)
            { // collision with brick
                this.ballYOffset = 0;
                this.ballYSpeed *= -1;
                this.destroyBrick((byte)(this.ballYTile-1),
                    this.ballXTile);
            } else {
                this.ballYTile -= 1;
                this.ballYOffset += BreakOutWorld.TILE_HEIGHT;
            }
            break;
        case 1:
            if (BreakOutWorld.map[this.ballYTile + 1][this.ballXTile]
!=
                BreakOutWorld.EMPTY_TILE)
            { // collision with brick
                this.ballYOffset = (byte)(BreakOutWorld.TILE_HEIGHT -

Constants.ballImg.getHeight());
                this.ballYSpeed *= -1;
                this.destroyBrick((byte)(this.ballYTile+1),
                    this.ballXTile);
            } else {
                if (this.ballYOffset > BreakOutWorld.TILE_HEIGHT) {
                    this.ballYTile += 1;
                    this.ballYOffset -= BreakOutWorld.TILE_HEIGHT;
                }
```

```
            }
            break;
        case 2:
            if (BreakOutWorld.map[this.ballYTile][this.ballXTile - 1]
!=
                BreakOutWorld.EMPTY_TILE)
            { // collision with brick
                this.ballXOffset = 0;
                this.ballXSpeed *= -1;
                this.destroyBrick(this.ballYTile,
                    (byte)(this.ballXTile-1));
            } else {
                this.ballXTile -= 1;
                this.ballXOffset += BreakOutWorld.TILE_WIDTH;
            }
            break;
        case 3:
            if (BreakOutWorld.map[this.ballYTile][this.ballXTile + 1]
!=
                BreakOutWorld.EMPTY_TILE)
            { // collision with brick
                this.ballXOffset = (byte)(BreakOutWorld.TILE_WIDTH -

Constants.ballImg.getWidth());
                this.ballXSpeed *= -1;
                this.destroyBrick(this.ballYTile,
                    (byte)(this.ballXTile+1));
            } else {
                if (this.ballXOffset > BreakOutWorld.TILE_WIDTH) {
                    this.ballXTile += 1;
                    this.ballXOffset -= BreakOutWorld.TILE_WIDTH;
                }
            }
            break;
        default:
            break;
    } // end of switch
}

/**
 * Atualiza as coordenadas verticais dos objetos
 */
private void updateVertBallCoord()
{
    this.ballYOffset += this.ballYSpeed;

    if (this.ballYOffset < 0) { // change tile and update offset
        if ((this.ballYTile - 1) < 0) { // collision with top wall
            this.ballYOffset = 0;
            this.ballYSpeed *= -1;
        } else {
            this.verifyCollision((byte)0);
        }
    } else if ((this.ballYOffset + Constants.ballImg.getHeight()) >
               BreakOutWorld.TILE_HEIGHT)
    {
        if ((this.ballYTile + 1) > (BreakOutWorld.map.length - 1))
        { // collision with bottom wall
            this.ballYOffset = (byte)(BreakOutWorld.TILE_HEIGHT -
                                      Constants.ballImg.getHeight());
            this.ballYSpeed *= -1;
        } else {
            this.verifyCollision((byte)1);
```

```
        }
    }
}

/**
 * Atualiza as coordenadas horizontais dos objetos
 */
private void updateHorizBallCoord()
{
    this.ballXOffset += this.ballXSpeed;

    if (this.ballXOffset < 0) { // change tile and update offset
        if ((this.ballXTile - 1) < 0) { // collision with left wall
            this.ballXOffset = 0;
            this.ballXSpeed *= -1;
        } else {
            this.verifyCollision((byte)2);
        }
    } else if ((this.ballXOffset + Constants.ballImg.getWidth()) >
                BreakOutWorld.TILE_WIDTH)
    {
        if ((this.ballXTile + 1) > (BreakOutWorld.map[0].length - 1))
        { // collision with right wall
            this.ballXOffset = (byte)(BreakOutWorld.TILE_WIDTH -
                                      Constants.ballImg.getWidth());
            this.ballXSpeed *= -1;
        } else {
            this.verifyCollision((byte)3);
        }
    }
}

/**
 * Atualiza os objetos do jogo
 */
public void updateWorld()
{
    this.lastBallXPos = this.ballXRPosition;
    this.lastBallYPos = this.ballYRPosition;
    this.brickXTileToDeleted = this.brickYTileToDeleted = -1;

    this.updateHorizBallCoord();
    if (!Constants.wasCollision)
        this.updateVertBallCoord();

    this.ballXRPosition =
        (byte)(this.getRealX(ballXTile)+this.ballXOffset);
    this.ballYRPosition =
        (byte)(this.getRealY(ballYTile)+this.ballYOffset);
}
```

Apêndice B: Funções de Detecção de Colisão (Space Invaders)

B.1 Detecção padrão

Implementação do método que detecta colisão para a técnica: “Detecção Padrão”.

```
public void checkCollision()
{
    Enemy enemy;
    for(int i=0; i < Constants.MAX_NUM_OBJECTS; i++) {
        enemy = (Enemy)this.enemies.getByIndex(i);
        if (enemy.isVisible()) {
            for(int j=0; j<SpaceInvadersWorld.MAX_MY_SHIP_FIRES; j++)
{
                if (this.myShipFires[j].isVisible()) {
                    if (enemy.intersects(this.myShipFires[j])) {
                        // collision
                        enemy.setVisible(false);
                        this.myShipFires[j].setVisible(false);
                        this.num_enemies--;
                        Constants.wasCollision = true;
                    }
                }
            }
        }
    }
    for(int i=0; i < SpaceInvadersWorld.MAX_ENEMY_FIRES; i++) {
        if (this.enemyFires[i].isVisible()) {
            if (this.enemyFires[i].intersects(this.myShip)) {
                this.enemyFires[i].setVisible(false);
            }
        }
    }
}
```

B.2 Região de influência

Implementação do método que detecta colisão para a técnica: “Região de Influência”.

```
public void checkCollision()
{
    Enemy enemy;
    for(int j=0; j<SpaceInvadersWorld.MAX_MY_SHIP_FIRES; j++) {
        if (this.myShipFires[j].isVisible() &&
            this.insideInfluenceRegion(this.myShipFires[j]))
        {
            for(int i=0; i < Constants.MAX_NUM_OBJECTS; i++) {
                enemy = (Enemy)this.enemies.getByIndex(i);
                if (enemy.isVisible()) {
                    if (enemy.intersects(this.myShipFires[j])) {
                        // collision
                        enemy.setVisible(false);
                        this.myShipFires[j].setVisible(false);
```

```
                        this.num_enemies--;
                        Constants.wasCollision = true;
                    }
                }
            }
        }
    }
    // collision detection between the ship and an enemy's fire
    for(int i=0; i < SpaceInvadersWorld.MAX_ENEMY_FIRES; i++) {
        if (this.enemyFires[i].isVisible()) {
            if (this.enemyFires[i].intersects(this.myShip)) {
                this.enemyFires[i].setVisible(false);
            }
        }
    }
}
```

B.3 Ordenação pelo eixo x desconsiderando a região de influência

Implementação do método que detecta colisão para a técnica: "Ordenação pelo eixo x desconsiderando a região de influência".

```
pubilc void checkCollision()
{
    Enemy enemy = null;
    boolean done;
    Fire[] sortedEnemyFires = this.sortEnemyFires(this.enemyFires);

    for (int i = 0; i < SpaceInvadersWorld.MAX_MY_SHIP_FIRES; i++) {
        if (this.myShipFires[i].isVisible()) {
            done = false;
            for(int j=0; j<Constants.NUM_ENEMY_LINE && !done; j++) {
                for(int k=0; k<4 && !done; k++) {
                    enemy = (Enemy)this.enemies.getByIndex(
                            k * Constants.NUM_ENEMY_LINE + j);
                    if (enemy.isVisible()) {

                        if (enemy.getX() > (this.myShipFires[i].getX()
+
                            this.myShipFires[i].getWidth()))
                        {
                            done = true;
                        } else {
                            if (this.myShipFires[i].intersects(enemy))
{
                                // collision
                                enemy.setVisible(false);
                                this.myShipFires[i].setVisible(false);
                                this.num_enemies--;
                                Constants.wasCollision = true;
                                break;
                            }
                        }
                    }
                }
            }
        }
    }
    if (sortedEnemyFires != null) {
        int size = sortedEnemyFires.length;
```

```
        done = false;

        for (int i = 0; i < size && !done ; i++) {
            if ((myShip.getX() + myShip.getWidth()) <
                sortedEnemyFires[i].getX())
            {
                done = true;
            } else {
                if (myShip.intersects(sortedEnemyFires[i])) {
                    sortedEnemyFires[i].setVisible(false);
                }
            }
        }
    }
}
```

B.4 Ordenação pelo eixo x com região de influência

Implementação do método que detecta colisão para a técnica: "Ordenação pelo eixo x com região de influência".

```
public void checkCollision()
{
    Enemy enemy = null;
    boolean done;
    Fire[] sortedEnemyFires = this.sortEnemyFires(this.enemyFires);

    for (int i = 0; i < SpaceInvadersWorld.MAX_MY_SHIP_FIRES; i++) {
        if (this.myShipFires[i].isVisible() &&
            this.insideInfluenceRegion(this.myShipFires[i]))
        {
            done = false;
            for(int j=0; j<Constants.NUM_ENEMY_LINE && !done; j++) {
                for(int k=0; k<4 && !done; k++) {
                    enemy = (Enemy)this.enemies.getByIndex(
                            k * Constants.NUM_ENEMY_LINE + j);
                    if (enemy.isVisible()) {

                        if (enemy.getX() > (this.myShipFires[i].getX()
+
                            this.myShipFires[i].getWidth()))
                        {
                            done = true;
                        } else {
                            if (this.myShipFires[i].intersects(enemy))
{
                                // collision
                                enemy.setVisible(false);
                                this.myShipFires[i].setVisible(false);
                                this.num_enemies--;
                                Constants.wasCollision = true;
                                break;
                            }
                        }
                    }
                }
            }
        }
    }
    if (sortedEnemyFires != null) {
        int size = sortedEnemyFires.length;
```

```
        done = false;

        for (int i = 0; i < size && !done ; i++) {
            if ((myShip.getX() + myShip.getWidth()) <
                sortedEnemyFires[i].getX())
            {
                done = true;
            } else {
                if (myShip.intersects(sortedEnemyFires[i])) {
                    sortedEnemyFires[i].setVisible(false);
                }
            }
        }
    }
}
```

B.5 Ordenação pelo eixo y desconsiderando a região de influência

Implementação do método que detecta colisão para a técnica: “Ordenação pelo eixo y desconsiderando a região de influência”.

```
public void checkCollision()
{
    Enemy enemy = null;
    boolean done;
    Fire[] sortedEnemyFires = this.sortEnemyFires(this.enemyFires);

    for (int i = 0; i < SpaceInvadersWorld.MAX_MY_SHIP_FIRES; i++) {
        if (this.myShipFires[i].isVisible()) {
            done = false;
            for(int j=0; j<Constants.MAX_NUM_OBJECTS && !done; j++) {
                enemy = (Enemy)this.enemies.getByIndex(
                        Constants.MAX_NUM_OBJECTS - j - 1);
                if (enemy.isVisible()) {

                    if (enemy.getY() + enemy.getHeight() <
                        this.myShipFires[i].getY())
                    {
                        done = true;
                    } else {
                        if (this.myShipFires[i].intersects(enemy)) {
                            // collision
                            enemy.setVisible(false);
                            this.myShipFires[i].setVisible(false);
                            this.num_enemies--;
                            Constants.wasCollision = true;
                            break;
                        }
                    }
                }
            }
        }
    }
    if (sortedEnemyFires != null) {
        int size = sortedEnemyFires.length;
        done = false;

        for (int i = 0; i < size && !done ; i++) {
            if ((myShip.getX() + myShip.getWidth()) <
                sortedEnemyFires[i].getX())
```

```
            {
                done = true;
            } else {
                if (myShip.intersects(sortedEnemyFires[i])) {
                    sortedEnemyFires[i].setVisible(false);
                }
            }
        }
    }
}
```

B.6 Ordenação pelo eixo y com região de influência

Implementação do método que detecta colisão para a técnica: "Ordenação pelo eixo y com região de influência".

```
public void checkCollision()
{
    Enemy enemy = null;
    boolean done;
    Fire[] sortedEnemyFires = this.sortEnemyFires(this.enemyFires);

    for (int i = 0; i < SpaceInvadersWorld.MAX_MY_SHIP_FIRES; i++) {
        if (this.myShipFires[i].isVisible() &&
            this.insideInfluenceRegion(this.myShipFires[i]))
        {
            done = false;
            for(int j=0; j<Constants.MAX_NUM_OBJECTS && !done; j++) {
                enemy = (Enemy)this.enemies.getByIndex(
                        Constants.MAX_NUM_OBJECTS - j - 1);
                if (enemy.isVisible()) {

                    if (enemy.getY() + enemy.getHeight() <
                        this.myShipFires[i].getY())
                    {
                        done = true;
                    } else {
                        if (this.myShipFires[i].intersects(enemy)) {
                            // collision
                            enemy.setVisible(false);
                            this.myShipFires[i].setVisible(false);
                            this.num_enemies--;
                            Constants.wasCollision = true;
                            break;
                        }
                    }
                }
            }
        }
    }
    if (sortedEnemyFires != null) {
        int size = sortedEnemyFires.length;
        done = false;

        for (int i = 0; i < size && !done ; i++) {
            if ((myShip.getX() + myShip.getWidth()) <
                sortedEnemyFires[i].getX())
            {
                done = true;
            } else {
                if (myShip.intersects(sortedEnemyFires[i])) {
```

```
                    sortedEnemyFires[i].setVisible(false);
                }
            }
        }
    }
}
```

B.7 Método do setor

Implementação do método que detecta colisão para a técnica: “Método do Setor”.

```
public void checkCollision()
{
    Enemy enemy = null;
    byte[] sectors;
    int size;

    size = this.myShipFires.length;
    for(int i=0; i < size; i++) {
        if (this.myShipFires[i].isVisible()) {
            sectors = this.myShipFires[i].getSectors();
            for(int j=0, index; j<4 && sectors[j] != -1; j++) {
                index = sectors[j];
                for(int k=0; k<10 && this.sectors[index][k] != null;
k++)
                {
                    if (this.sectors[index][k] instanceof Enemy) {
                        enemy = (Enemy)this.sectors[index][k];
                        if (this.myShipFires[i].intersects(enemy)) {
                            // collision
                            enemy.setVisible(false);
                            this.myShipFires[i].setVisible(false);
                            this.num_enemies--;
                            Constants.wasCollision = true;
                            break;
                        }
                    }
                }
            }
        }
    }
    for (int i = 0; i < this.enemyFires.length; i++) {
        if (this.enemyFires[i].isVisible()) {
            // collision detection between the ship and an enemy's
fire
            if (this.myShip.intersects(enemyFires[i])) {
                this.enemyFires[i].setVisible(false);
            }
        }
    }
}
```

B.8 Baseada em ladrilhos

Implementação do método que detecta colisão para a técnica: “Baseada em ladrilhos”.

```
/**
 * Verifica se houve colisao.
 */
private void verifyCollision() {
```

```
    int index = 0;
    for(int i=4; i < this.line; i++) {
        for(int j=0; j<SpaceInvadersWorld.X_TILES; j++) {
            if (this.bulletsMap[i][j] ==
                SpaceInvadersWorld.MYSHIP_FIRE_TILE)
            {
                if (this.map[i][j] != SpaceInvadersWorld.EMPTY_TILE) {
                    if (this.map[i][j]==
                        SpaceInvadersWorld.ENEMY_TILE[0])
                    {
                        for(int a=0; a<2; a++)
                            for(int b=0; b<3; b++)
                                this.map[i+a][j+b] =
                                    SpaceInvadersWorld.EMPTY_TILE;
                    } else if (this.map[i][j]==
                        SpaceInvadersWorld.ENEMY_TILE[1])
                    {
                        for(int a=0; a<2; a++)
                            for(int b=-1; b<2; b++)
                                this.map[i+a][j+b] =
                                    SpaceInvadersWorld.EMPTY_TILE;
                    } else if (this.map[i][j]==
                        SpaceInvadersWorld.ENEMY_TILE[2])
                    {
                        for(int a=0; a<2; a++)
                            for(int b=-2; b<1; b++)
                                this.map[i+a][j+b] =
                                    SpaceInvadersWorld.EMPTY_TILE;
                    } else if (this.map[i][j]==
                        SpaceInvadersWorld.ENEMY_TILE[3])
                    {
                        for(int a=-1; a<1; a++)
                            for(int b=0; b<3; b++)
                                this.map[i+a][j+b] =
                                    SpaceInvadersWorld.EMPTY_TILE;
                    } else if (this.map[i][j]==
                        SpaceInvadersWorld.ENEMY_TILE[4])
                    {
                        for(int a=-1; a<1; a++)
                            for(int b=-1; b<2; b++)
                                this.map[i+a][j+b] =
                                    SpaceInvadersWorld.EMPTY_TILE;
                    } else if (this.map[i][j]==
                        SpaceInvadersWorld.ENEMY_TILE[5])
                    {
                        for(int a=-1; a<1; a++)
                            for(int b=-2; b<1; b++)
                                this.map[i+a][j+b] =
                                    SpaceInvadersWorld.EMPTY_TILE;
                    }
                    this.bulletsMap[i][j] =
                        SpaceInvadersWorld.EMPTY_TILE;
                    Constants.wasCollision = true;
                    this.num_enemies--;
                }
            }
        }
    }
}

/**
 * Atualiza os objetos do jogo
 */
```

```
public void updateWorld()
{
    // update existent fires
    for(int i=0; i<SpaceInvadersWorld.Y_TILES; i++) {
        for(int j=0; j<SpaceInvadersWorld.X_TILES; j++) {
            // update
            if (this.bulletsMap[i][j] ==
                SpaceInvadersWorld.MYSHIP_FIRE_TILE)
            {
                this.bulletsMap[i][j] = SpaceInvadersWorld.EMPTY_TILE;
                if (i > 0) {
                    if (this.bulletsMap[i-1][j] ==
                        SpaceInvadersWorld.EMPTY_TILE)
                    {
                        this.bulletsMap[i-1][j] =
                            SpaceInvadersWorld.MYSHIP_FIRE_TILE;
                    } else {
                        this.bulletsMap[i-1][j] =

SpaceInvadersWorld.ENEMY_AND_SHIP_FIRE_TILE;
                    }
                }
            } else if (this.bulletsMap[i][j] ==
                SpaceInvadersWorld.ENEMY_AND_SHIP_FIRE_TILE)
            { // update
                this.bulletsMap[i][j] = SpaceInvadersWorld.EMPTY_TILE;
                if (i > 0)
                    this.bulletsMap[i-1][j] =
                        SpaceInvadersWorld.MYSHIP_FIRE_TILE;
                if (i < (SpaceInvadersWorld.Y_TILES-1))
                    this.bulletsMap[i][j] =
                        SpaceInvadersWorld.ENEMY_FIRE_TILE;
            }
        }
    }
    for(int i=SpaceInvadersWorld.Y_TILES-1; i>=0; i--) {
        for(int j=0; j<SpaceInvadersWorld.X_TILES; j++) {
            if (this.bulletsMap[i][j] ==
                SpaceInvadersWorld.ENEMY_FIRE_TILE)
            { // update
                this.bulletsMap[i][j] = SpaceInvadersWorld.EMPTY_TILE;
                if (i < (SpaceInvadersWorld.Y_TILES-2)) {
                    if (this.bulletsMap[i+2][j] ==
                        SpaceInvadersWorld.EMPTY_TILE)
                    {
                        this.bulletsMap[i+2][j] =
                            SpaceInvadersWorld.ENEMY_FIRE_TILE;
                    } else {
                        this.bulletsMap[i+2][j] =

SpaceInvadersWorld.ENEMY_AND_SHIP_FIRE_TILE;
                    }
                }
            } else if (this.bulletsMap[i][j] ==
                SpaceInvadersWorld.ENEMY_AND_SHIP_FIRE_TILE)
            { // update
                this.bulletsMap[i][j] = SpaceInvadersWorld.EMPTY_TILE;
                if (i > 0)
                    this.bulletsMap[i][j] =
                        SpaceInvadersWorld.MYSHIP_FIRE_TILE;
                if (i < (SpaceInvadersWorld.Y_TILES-2))
                    this.bulletsMap[i+2][j] =
                        SpaceInvadersWorld.ENEMY_FIRE_TILE;
```

```
            }
        }
    }

    // update enemies
    this.enemy_speed_control++;
    if (this.enemy_speed_control == 4) {
        this.enemy_move_control--;

        if (this.enemy_move_control < 0) {
            this.enemy_speed *= -1;
            this.enemy_move_control = 4;
        }

        this.enemy_speed_control = 0;
        this.updateEnemies();
    }

    // every MY_SHIP_FIRE_INTERVAL, a new bullet is fired by the ship.
    if (this.shipFireIteration == Constants.MY_SHIP_FIRE_INTERVAL) {
        this.shipFireIteration = 0;
        this.bulletsMap[21][this.current_x_tile] =
            SpaceInvadersWorld.MYSHIP_FIRE_TILE;
    } else {
        this.shipFireIteration++;
    }

    // every ENEMY_FIRE_INTERVAL, a new bullet is fired by an enemy.
    if (this.enemyFireIteration == Constants.ENEMY_FIRE_INTERVAL) {
        int currentEnemy = 0;
        boolean end = false;

        this.enemyFireIteration = 0;

        for(int i=4; i < 8 && !end; i++) {
            for(int j=0; j<SpaceInvadersWorld.X_TILES && !end; j++) {
                if ((this.map[i][j] ==
SpaceInvadersWorld.ENEMY_TILE[0]))
                {
                    if (currentEnemy == this.enemyToLaunchFire) {
                        this.bulletsMap[i+2][j+1] =
                            SpaceInvadersWorld.ENEMY_FIRE_TILE;
                        end = true;
                    } else {
                        currentEnemy++;
                    }
                }
            }
        }
        if (this.num_enemies > 0)
            this.enemyToLaunchFire = (byte)
                ((this.enemyToLaunchFire + 1) % this.num_enemies);

    } else {
        this.enemyFireIteration++;
    }

    // update ship
    this.ship_speed_control++;
    if (this.ship_speed_control == 4) {
        this.ship_move_control--;

        if (this.ship_move_control < 0) {
```

```
            this.ship_speed *= -1;
            this.ship_move_control = 20;
        }

        this.ship_speed_control = 0;
        this.updateShip();
    }
    this.verifyCollision();
}
```

Referências Bibliográficas

[1]. Perazzo, E., Ramalho, G., e Santos, A. *Avaliação Experimental de Detecção de Colisão para Jogos J2ME*. WJogos, Curitiba/PR, Brasil, 2004.

[2]. Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Element of Reusable Object-Oriented Software*, Addison-Wesley, pp. 315-323, Boston, 1995.

[3]. Zhou, H., Chen, M., Webster, M. *Comparative Evaluation of Visualization and Experimental Results Using Image Comparison Metrics*, Conference on Visualization '02, Boston, 2002.

[4]. Sun. *Virtual Machines*, http://java.sun.com/, 2004.

[5]. Peng Li, S. *Wireless Games The Next Internet Goldmine*, http://www.wi-fiplanet.com/news/article.php/1445341, 2004.

[6]. Zelkowitz, S., Wallace, D. *Experimental Validation in Software Engineering*, Conference of Empirical Assessment & Evaluation in Software Engineering, Keele University, Staffordshire, U.K, 1997.

[7]. Desilva, S., Das, S. *Experimental Evaluation of a Wireless Ad Hoc network*, 9th International Conference on Computer Communications and Networks (IC3N), Las Vegas, October 2000.

[8]. Chandra, P., Fisher, A., Kosak, C., and Steenkiste, P. *Experimental Evaluation of ATM Congestion Control Mechanisms*, IEEE Infocom, 1997.

[9]. Kotz, D., Newport, C., Gray, R., Liu, J., Yuan, Y. and Elliott, C. *Experimental Evaluation of Wireless Simulation Assumptions*, MSWiM'04, Italy, 2004.

[10]. Roberts, D. *Getting the most out of your collision tests*, http://www.ddj.com/documents/s=983/ddj9513a/9513a.htm, 2004.

[11]. Mirtich, Brian Vicent, *Impulse-based Dynamic Simulation of Rigid Body Systems*, PhD thesis, University of California, Berkeley, 1996.

[12]. Tichy, W. *Should Computer Scientists Experiment More?* University of Karlsruhe. 1998.

[13]. Pfleeger, S. *Experimental Design and Analysis in Software Engineering. Part 1: The Language of Case Studies and Formal Experiments*.

[14]. Pfleeger, S. *Experimental Design and Analysis in Software Engineering. Part 2: How to Set Up an Experiment*.

[15]. Reggiani, M., Mazzoli, M., Caselli, S. *An Experimental Evaluation of Collision Detection Packages for Robot Motion Planning*.

[16]. Levey, E., Peters, C., O'Sullivan, C. *New Metrics for Evaluation of Collision Detection Techniques*.

[17]. Zachmann, G. *Benchmarking Collision Detection Algorithms*. http://web.informatik.uni-bonn.de/II/ag-klein/people/zach/coldet/

[18]. Fan, J., Ries E., Tenitchi, C. *Black Art of Java Game Programming*. Waite Group Press. 1996.

[19]. Lamothe, A., Ratcliff, J., Seminatore, D. *Tricks of the Game Programming Gurus*. Sams Pub. 1994.

[20]. Wang, W. Yi-King Choi. Bin Chan. Myung-Soo Kim. Jiaye Wang. *Efficient Collision Detection for Moving Ellipsoids Based on Simple Algebraic Test and Separating Planes*. Technical report TR-2002-16, Department of Computer Science and Information Systems, University of Hong Kong, 2002.

[21]. J.T. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, K. Zikan. *Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs* IEEE Transactions on Visualization and Computer Graphics, March 1998, Volume 4, Number 1. Pages 21-36.

[22]. Hamer, C., *Using the MIDP 2.0 Game API*. Disponível em http://www.javaworld.com/javaworld/jw-08-2004/jw-0809-j2me.html, 2005.

[23]. Kontio, M., *The Game API*. Disponível em http://java.sys-con.com/read/37511.htm, 2005.

[24]. Knudsen, J., *Understanding MIDP 2.0's Security Architecture*. Disponível em http://developers.sun.com/techtopics/mobility/midp/articles/permissions/, 2005.

[25]. *MIDP 1.0 Style Guide*. Disponível em http://java.sun.com/j2me/docs/alt-html/midp-style-guide7/, 2002.

[26]. Bloch C., Wagner, A., *MIDP 2.0 Style Guide for the Java 2 Platform, Micro Edition*. Addison-Wesley Professional, 2003.

[27]. Câmara, C., Santos, A. e Ramalho, G., *Whole Program Optimizations of Java 2 Micro Edition Bytecode*, CIn-UFPE, 2004.

[28]. Sun Microsystems, *CDC – Connected Device Configuration, v1.0a*. JCP Specification, JSR 036. Disponível em http://jcp.org/aboutJava/communityprocess/final/jsr036/, 2005.

[29]. Sun Microsystems, *CLDC – Connected, Limited Device Configuration, v1.0a*. JCP Specification, JSR 030. Disponível em http://jcp.org/aboutJava/communityprocess/final/jsr030/, 2005.

[30]. Sun Microsystems, *CLDC – Connected, Limited Device Configuration, v1.1*. JCP Specification, JSR 139. Disponível em http://jcp.org/aboutJava/communityprocess/final/jsr139/, 2005.

[31]. Sun Microsystems, *MIDP – Mobile Information Device Profile, v1.0*. JCP Specification, JSR 037. Disponível em http://jcp.org/aboutJava/communityprocess/final/jsr037/, 2005.

[32]. Sun Microsystems, *MIDP – Mobile Information Device Profile, v2.0*. JCP Specification, JSR 118. Disponível em http://jcp.org/aboutJava/communityprocess/final/jsr118/, 2005.

[33]. Sun Microsystems, *J2ME - Java 2 Platform, Micro Edition*. Disponível em http://java.sun.com/j2me/, 2005.

[34]. Sun Microsystems, *J2SE - Java 2 Platform, Standard Edition*. Disponível em http://java.sun.com/j2se/, 2005.

[35]. AviSynth, *Interlacing and Deinterlacing*. Disponível em http://www.avisynth.org/InterlacingAndDeinterlacing, 2005.

[36]. Divx.com. *NTSC, PAL, and Interlace Explained*. Disponível em http://www.divx.com/support/guides/guide.php?gid=10, 2005.

[37]. Lindholm, T., Yellin, F, *The Java Virtual Machine Specification*. Disponível em http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html, 2005.

[38]. Venners, B., *Inside The Java Virtual Machine*. McGraw-Hill, Second Edition, January 2000.

[39]. Michael, D., *Tile/Map-Based Game Techniques: Handling Terrain Transitions*. Disponível em http://www.gamedev.net/reference/articles/article934.asp, 1999.

[40]. Adams, J., *Smooth Scrolling a Tile Map*. Disponível em http://www.gamedev.net/reference/articles/article743.asp, 1999.

[41]. Gambone, D., *Isometric Tiles*. Disponível em http://www.gamedev.net/reference/articles/article738.asp, 1999.

[42]. McIntosh, J., *Tile Graphics Techniques*. Disponível em http://www.gamedev.net/reference/articles/article727.asp, 1999

[43]. Palmer, C., Taylor, G., *Tile Based Games FAQ*. Disponível em http://www.gamedev.net/reference/articles/article728.asp, 1999.

[44]. Hufsky, F., *Tilebased collision detection and response*. Disponível em http://www.maybe-phil.net/websites/no-skill/jnrdev/en/jnrdev1/, 2004.

[45]. ACZ, *Tile-Based Collision Detection*. Disponível em http://ti89.acz.org/collision.htm, 1999.

[46]. Lukowicz, P., Tichy, W., Heinz, E., Prechelt, L. *Experimental Evaluation in Computer Science: A Quantitative Study*. Journal of Systems and Software. Vol. 28. Páginas 9-18. 1995.

Dissertação de Mestrado apresentada por **Eric Bruno Perazzo Mariz** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título **"Avaliação Experimental de Detecção de Colisão para Jogos J2ME"**, orientada pelo **Prof. André Luis de Medeiros Santos** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Silvio de Barros Melo
Centro de Informática / UFPE

Prof. Ricardo Masa Ferreira Lima
Escola Politécnica / UPE

Prof. André Luis de Medeiros Santos
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 19 de julho de 2005.

Prof. JAELSON FREIRE BRELAZ DE CASTRO
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.