UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

FÍSICA DE CORPOS RÍGIDOS PARA O MOTOR DE JOGOS M3GE NA PLATAFORMA J2ME

ERNANI CRISTIANO SIEBERT

BLUMENAU
2010

2010/2-13

ERNANI CRISTIANO SIEBERT

FÍSICA DE CORPOS RÍGIDOS PARA O MOTOR DE JOGOS M3GE NA PLATAFORMA J2ME

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Dalton Solano dos Reis - Orientador

BLUMENAU
2010

2010/2-13

FÍSICA DE CORPOS RÍGIDOS PARA O MOTOR DE JOGOS M3GE NA LINGUAGEM J2ME

Por

ERNANI CRISTIANO SIEBERT

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Dalton Solano dos Reis, M.Sc. – Orientador, FURB
Membro:	Prof. Paulo Cesar Rodacki Gomes, Dr. – FURB
Membro:	Prof. Mauro Marcelo Mattos, Dr. – FURB

Blumenau, 13 de dezembro de 2010

Dedico este trabalho a minha família, pelo apoio recebido durante os anos de graduação.

AGRADECIMENTOS

Ao professor Dalton Solano dos Reis, pela orientação e apoio no desenvolvimento deste trabalho.

O homem deve criar as oportunidades e não somente encontrá-las.

Francis Bacon

RESUMO

Este Trabalho de Conclusão de Curso descreve a implementação de um módulo de física de corpos rígidos para o motor M3GE, desenvolvido na plataforma J2ME para dispositivos na configuração CLDC e perfil MIDP, que corresponde à maioria dos celulares com visor colorido. Para a detecção da colisão, são utilizados objetos com o formato de caixa, usando-se algoritmos que fazem o teste de vértice com chão e vértice com plano. São descritas ainda as forças aplicadas quando os corpos colidem e resultando no deslocamento linear e angular.

Palavras-chave: Jogos. Celular. Motor. Física. Colisão. Corpos rígidos. J2ME. M3G. M3GE.

ABSTRACT

This work describes the implementation of a rigid bodies module for the engine M3GE, developed in the platform J2ME for CLDC configuration and MIDP profile, which corresponds to most mobile phones with color display. For the collision detection, are used objects having box shape, utilizing algorithms that test vertex with floor and vertex with plane. Also are described the forces applied when the bodies collide resulting in displacement linear (translation) and angular (rotation).

Key-words: Games. Cell phone. Engine. Physic. Collision. Rigid bodies. J2ME. M3G. M3GE.

LISTA DE ILUSTRAÇÕES

Figura 1 – Componentes do Java Micro Edition 20
Figura 2 - Módulos implementados no M3GE 22
Figura 3 – Orientação como diferença entre eixos de coordenadas 24
Quadro 1 – Velocidade angular 24
Quadro 2 – Aceleração angular 24
Figura 4 – Trajetória circular das partículas durante a rotação 25
Quadro 3 – Comprimento do arco no deslocamento angular 25
Quadro 4 – Velocidade linear em volta de um eixo 25
Quadro 5 – Aceleração linear tangencial 26
Figura 5 – Aceleração centrípeta resultante da gravidade 26
Quadro 6 – Aceleração centrípeta 26
Quadro 7 – Velocidade tangencial como vetor de magnitude e direção 26
Quadro 8 – Aceleração tangencial e centrípeta como vetores 26
Quadro 9 – Massa total na forma integral 27
Quadro 10 – Massa total como soma das massas individuais 27
Figura 6 – Centro de massa em um objeto composto 28
Quadro 11 – Centro de massa em cada coordena 28
Figura 7 – Resistência a rotação detende da distancia ao eixo que uma força é aplicada 29
Quadro 12 – Momento de inércia como integral 29
Quadro 13 – Momentos de inércia para as formas mais comuns 29
Quadro 14 – Exemplo de calculo do cilindro retangular 30
Quadro 15 – Distância ao quadrado de cada componente do centro de gravidade 30
Quadro 16 – Teorema dos eixos aplicado a cada componente do corpo 30
Quadro 17 – Momento de inércia total de um ponto em volta do eixo neutro 30
Quadro 18 – Conservação de momento 32
Quadro 19 – Energia cinética linear e angular 32
Quadro 20 – Coeficiente de restituição 32
Figura 8 – Impulso alterando ambas velocidade linear e angular 33
Quadro 21 – Impulso linear 33
Quadro 22 – Velocidade após impacto 33
Quadro 23 – Velocidade no ponto de contato 33

Quadro 24 – Formula combinada para impulso linear e angular 34
Quadro 25 – Velocidade linear e angular 34
Figura 9 – Força de atrito causando resistência à movimentação 34
Quadro 26 – Impulso total 34
Quadro 27 – Calculo da nova velocidade angular 35
Quadro 28 – Velocidades linear e angular 35
Quadro 29 – Vetor tangente 35
Figura 10 – Trajetória alterada pela rotação quando uma bola de tênis atinge o solo 35
Figura 11 - Diagrama de casos de uso 38
Quadro 30 – Caso de uso 01 - Lê informações do cenário 38
Quadro 31 – Caso de uso 02 - Aplica forças simuladas 38
Quadro 32 – Caso de uso 03 - Detecta possíveis colisões 39
Quadro 33 – Caso de uso 04 - Resolve colisões 39
Figura 12 – Diagrama de classes 40
Figura 13 – Classes `Matriz`, `Vetor`, `Quaternion` e suas dependências 41
Figura 14 – Classes `Collision`, `Edge` e suas dependências 42
Figura 15 – Classe `Object3DInfo` e suas dependências 43
Figura 16 – Classe `Psysics` e suas dependências 45
Figura 17 – Diagrama de seqüência 47
Quadro 34 – Rotina de integração 49
Quadro 35 – Verificação de colisão 51
Quadro 36 – Verificações de colisão feitas para a *bounding box* 52
Quadro 37 – Constantes de retorno para os métodos de colisão 52
Quadro 38 – Dados necessários para resposta de colisão 52
Figura 18 – Configuração do dispositivo a ser emulado 53
Figura 19 – Como executar o jogo de exemplo 54
Figura 20 – Opções para carregar o cenário de demonstração 55
Figura 21 – Objetos em queda e colisão com o solo e permanecendo em contato 56
Figura 22 – Colisão entre dois cubos 57
Figura 23 – Mudança de câmera 58
Quadro 39 – Teste 1 - tempos obtidos 59
Figura 24 – Teste 1 - relação do tempo com a quantidade de cubos 60
Quadro 40 – Teste 2 - tempos obtidos 60

Figura 25 – Teste 2 - relação do tempo com a quantidade de cubos..60
Quadro 41 – Teste 3 - tempos obtidos..61
Figura 26 – Teste 3 - relação do tempo com a quantidade de cubos..61
Quadro 42 – Teste 4 - tempos obtidos..61
Figura 27 – Teste 4 - relação do tempo com a quantidade de cubos..61
Figura 28 – Relação entre os testes de desempenho..62
Figura 29 – Teste 5 - fórmula pra o consumo de memória..62
Quadro 43 – Teste 5 - consumos obtidos...62
Figura 30 – Teste 5 - relação do uso de memória com a quantidade de cubos........................62
Figura 31 – Teste 6 - fórmula pra o consumo de memória..63
Quadro 44 – Teste 6 - consumos obtidos...63
Figura 32 – Teste 6 - relação do uso de memória com a quantidade de cubos........................63
Figura 33 – Teste 7 - Fórmula pra o consumo de memória..63
Quadro 45 – Teste 7 - consumos obtidos...63
Figura 34 – Teste 7 - relação do uso de memória com a quantidade de cubos........................64
Figura 35 – Teste 8 - Fórmula pra o consumo de memória..64
Quadro 46 – Teste 8 - consumos obtidos...64
Figura 36 – Teste 8 - relação do uso de memória com a quantidade de cubos........................64
Figura 37 – Relação entre os testes de memória...65

LISTA DE SIGLAS

CDC - *Connected Device Configuration*

CLDC - *Connected Limited Device Configuration*

CPU - *Central Processing Unit*

FPS - *First Person Shooter*

FPS – Frames Por Segundo

GJK - *Gilbert Johnson Keerthi*

GUI - *Graphical User Interface*

JCP – *Java Community Process*

JVM - *Java Virtual Machine*

LCP - *Linear Complementarity Problem*

M3G - *Mobile 3D Graphics API*

M3GE - *Mobile 3D Game Engine*

MIDP - *Mobile Information Device Profile*

OBJ - *OBJect file Wavefront Technologies*

PDA - *Personal Digital Assistant*

PPU - *Physics Processing Unit*

RAM - *Random Access Memory*

RMI - *Remote Method Invocation*

ROM - *Read Only Memory*

LISTA DE SÍMBOLOS

α – Alpha (minúscula), aceleração angular

μ – Mi (minuscula), coeficiente de atrito

Ω – Omega (maiúscula), orientação

ω – Omega (minúscula), velocidade angular

SUMÁRIO

1 INTRODUÇÃO **15**
1.1 OBJETIVOS DO TRABALHO 16
1.2 ESTRUTURA DO TRABALHO 16
2 FUNDAMENTAÇÃO TEÓRICA **18**
2.1 MOTORES DE JOGOS 18
2.2 JAVA 2 MICRO EDITION (J2ME) 19
2.3 MOBILE 3D GRAPHICS API 21
2.3.1 Mobile 3D Graphics Engine 21
2.4 DINÂMICA DE CORPOS RÍGIDOS 23
2.5 DETECÇÃO DE COLISÃO 23
2.6 CONCEITOS DE CINEMÁTICA PARA CORPOS RÍGIDOS 24
2.6.1 Rotação Angular 24
2.6.2 Movimento angular 25
2.6.3 Aceleração centrípeta 26
2.7 PROPRIEDADES DE MASSA 27
2.7.1 Massa e massa total 27
2.7.2 Centro de Massa 27
2.7.3 Momento de inércia 28
2.8 RESPOSTA DE COLISÃO 31
2.8.1 IMPACTO 32
2.8.2 IMPULSO LINEAR E ANGULAR 33
2.8.3 ATRITO 34
2.9 TRABALHOS CORRELATOS 35
3 DESENVOLVIMENTO **37**
3.1 REQUISITOS PRINCIPAIS DO MÓDULO A SER DESENVOLVIDA 37
3.2 ESPECIFICAÇÃO 37
3.2.1 Diagrama de Casos de Uso 37
3.3 DIAGRAMA DE CLASSES 39
3.3.1 As Classes Vetor, Matriz e Quaternion 40
3.3.2 As Classes `Collision` e `Edge` 41
3.3.3 A Classe `Object3DInfo` 42

3.3.4 A Classe `Physics` 44
3.3.5 Diagrama de Sequência 46
3.4 IMPLEMENTAÇÃO 47
3.4.1 Técnicas e ferramentas utilizadas 47
3.4.1.1 Integração 48
3.4.1.2 Detecção de colisão 49
3.4.2 Operacionalidade da implementação 53
3.5 RESULTADOS E DISCUSSÃO 58
4 CONCLUSÕES 66
4.1 EXTENSÕES 66
REFERÊNCIAS BIBLIOGRÁFICAS 68

1 INTRODUÇÃO

Desde tempos incontáveis o homem tem inventado jogos para se socializar, divertir, medir habilidades e exercitar o raciocínio e criatividade. Com o advento da tecnologia os jogos também evoluíram e passaram para meios eletrônicos. A atual portabilidade e integração de funções adicionais a dispositivos inicialmente voltados para outros fins, como celulares e *palms*, permitiu que os jogos também fossem difundidos nestes meios, ao alcance de cada vez mais pessoas.

Antes da metade da década de 1990, os jogos eram tipicamente escritos como entidades singulares. Os maiores limitadores eram as restrições de memória e a necessidade de fazer uso otimizado do hardware de vídeo. Além disso, o rápido avanço das plataformas de hardware significava que a maior parte do código não poderia ser reutilizada.

O cenário atual dos dispositivos móveis assemelha-se muito ao cenário existente na década de 80, em relação a qualidade gráfica e complexidade, quando os jogos começaram a ser difundidos para computadores domésticos (importante notar a diferença entre computadores e consoles, este último têm plataforma de hardware totalmente dedicadas).

O termo *game engine* (motor de jogos) surgiu com a popularização dos *First Person Shooter* (FPSs) quando os jogos começaram a ser desenvolvidos com a abordagem de separar regras específicas das entidades básicas que podiam ser licenciadas.

Motores de jogos também são freqüentemente utilizados por outros tipos de aplicações como demonstrações de produtos, visualizações arquiteturais e modelagem de ambientes (ELDAWY, 2006). Um motor realístico pode inclusive melhorar o aprendizado implementando regras do mundo real que podem ser usadas tanto para a educação, ensinando jovens como pensarem, quanto para treinamento médico e militar através de simuladores (BERBERICH, WRITER, 2007).

Assim como a separação entre o conteúdo específico e o motor permitiu que as equipes se especializassem e crescessem, o contínuo refinamento dos motores de jogos criaram distintas divisões e atualmente pode-se dizer que as duas principais são: motor gráfico, responsável pela renderização das imagens e motor de física, que realizar operações matemáticas de forma mais rápida.

Este trabalho pretende implementar um módulo para o motor Mobile 3D Game Engine (M3GE) (PAMPLONA, 2005) que adiciona física de corpos rígidos para que desenvolvedores e projetistas de jogos tenham mais recursos disponíveis e possam se concentrar em outras

etapas como enredo e interface. Levando em conta que este trabalho é uma extensão, ele adiciona um gerenciador de física que trata da movimentação dos objetos carregados nele, detectando colisões e aplicando a devida resposta em deslocamento linear (translação) e angular (rotação).

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um módulo para o motor M3GE que adicione física de corpos rígidos em 3D.

Os objetivos específicos do trabalho são:

a) associar um padrão em forma de caixa para detectar a colisão entre objetos carregados no motor;
b) detectar colisões e extrair as informações necessárias para o tratamento delas;
c) aplicar a resposta de colisão para determinar o deslocamento linear e angular;
d) obter uma taxa aceitável de Frames Por Segundo (FPS) para o máximo de corpos gerenciados.

1.2 ESTRUTURA DO TRABALHO

O capítulo 2 contém a fundamentação teórica, expondo os tópicos mais importantes para o desenvolvimento do presente trabalho.

A seção 2.1 apresenta uma das principais funcionalidades de um motor de jogos. A seção 2.2 apresenta a arquitetura da plataforma J2ME e as configurações existentes para cada família de dispositivos. A seção 2.3 apresenta o pacote M3G, utilizado para desenvolvimento de aplicações em 3D, e em seguida na seção 2.3.1, são apresentadas as funcionalidades do motor de jogos M3GE.

A seção 2.4 traz alguns conceitos do que se trata a dinâmica de corpos rígidos, e a seção 2.5 apresenta como é resolvida a detecção de colisão. A seção 2.6 apresenta conceitos comuns relacionados a cinemática de corpos rígidos. As seções 2.6.1, 2.6.2, 2.6.3, descrevem respectivamente a rotação angular, movimento angular e aceleração centrípeda.

A seção 2.7 apresenta todas propriedades de massa utilizadas na resposta da colisão. As seções 2.7.1, 2.7.2, 2.7.3, apresentam respectivamente a massa e massa total, centro de massa, e momento de inércia.

A seção 2.8 apresenta as forças que estão envolvidas no cálculo da resposta de colisão. As seções 2.8.1, 2.8.2, 2.8.3, 2.8.4 apresentam respectivamente o momento de impulso, impacto, impulso linear e angular e atrito.

Na seção 2.9 são relatados os trabalhos correlatos e as principais características que se relacionam com o presente trabalho.

O capítulo 3 apresenta a especificação e implementação do módulo. No capítulo 4 são apresentadas as conclusões deste trabalho bem como possíveis extensões em futuros trabalhos.

2 FUNDAMENTAÇÃO TEÓRICA

Para entendimento do trabalho, são apresentados os conceitos envolvidos desde o mais geral até especificamente o que foi implementado, começando com o que são os motores de jogos e sua arquitetura; a plataforma J2ME; o motor desenvolvido por Pamplona (2005) e conceitos de física de corpos rígido utilizados nos algoritmos de detecção e resposta de colisão. No final são apresentados cinco trabalhos correlatos.

2.1 MOTORES DE JOGOS

Os motores de jogos implementam funcionalidades comuns à maioria dos jogos, permitindo que sejam reutilizadas a cada novo jogo criado (WARD, 2008), como:

a) gerenciador principal: coordena os demais componentes;
b) gerenciador de entrada: recebe e identifica os eventos de entrada e os envia para o gerenciador principal;
c) gerenciador gráfico: transforma o modelo do estado atual do jogo em uma visualização para o usuário;
d) gerenciador de som: execução de sons a partir de eventos no jogo;
e) gerenciador de inteligência artificial: gerencia o comportamento dos objetos desenvolvidos pelo *designer*;
f) gerenciador de múltiplos jogadores: trata da comunicação dos jogadores, independentemente do meio físico em que se encontram;
g) gerenciador de objetos: carrega, controla o ciclo de vida, salva e destrói um grupo de objetos do jogo. O objeto do jogo possui dados relevantes para uma entidade que faça parte do jogo (como avião, monstro, etc.), controlando posição, velocidade, dimensão, detecção de colisão, entre outros;
h) gerenciador do mundo: armazena o estado atual do jogo e para isso utiliza os gerenciadores de objetos;
i) editor de cenários: em geral, uma ferramenta externa que descreve um estado inicial do jogo para cada um de seus níveis.

Entretanto, nenhum jogo de respeito atualmente pode passar sem um bom motor de

física. No inicio, a física era implementada com cada efeito específico para aquele título. A dificuldade de conseguir uma aparência realística, mais a necessidade de quase o mesmo efeito jogo após jogo, encorajou desenvolvedores a procurar uma solução geral que pudesse ser reutilizada. Portanto, pode-se definir um motor de física como uma parte comum de código que sabe sobre física em geral, mas não é programada com as especificidades do cenário de cada jogo. É basicamente uma grande calculadora que faz a matemática necessária, mas não sabe o que precisa ser simulado (MILLINGTON, 2007, p. 2-3).

2.2 JAVA 2 MICRO EDITION (J2ME)

O J2ME tem todas as características para satisfazer as necessidades que o desenvolvimento de software para dispositivos mais limitados impõe - baixo poder de processamento e pouca memória disponível (SUN MICROSYSTEM, 2009a).

Outro problema ao se desenvolver para dispositivos móveis é a extensa interação e parceria com os fabricantes de dispositivos, e para resolve-lo, foi especificada o J2ME (BATTAIOLA et al., 2001, p. 37-38). A plataforma J2ME é uma versão simplificada das APIs do Java e da sua máquina virtual para dispositivos móveis, trazendo grandes resultados nas áreas de automação comercial e industrial. Mas inicialmente descartado para desenvolvimento de jogos por ser muito lento, sendo adotadas para este tipo de desenvolvimento as linguagens nativas dos dispositivos, por serem mais rápidas e com mais recursos gráficos.

J2ME é um conjunto de tecnologias e especificações que possibilita o desenvolvimento de software para sistemas e aplicações embarcadas, ou seja, rodam num dispositivo de propósito específico como celulares, *Personal Digital Assistants* (PDAs), controles remotos e outros, que têm como alvo disponibilizar uma Java Virtual Machine (JVM) com APIs definidas através da *Java Community Process* (JCP) e baseiam-se nas necessidades da sua arquitetura computacional (SUN MICROSYSTEM, 2009a).

A arquitetura é definida pelo J2ME em um modelo de três camadas embutidas sobre o sistema operacional: configuração, perfil e pacotes opcionais.

A camada de configuração define a JVM e conjunto de classes para uma família de dispositivos:

a) *Connected Device Configuration* (CDC): dispositivos com maior capacidade

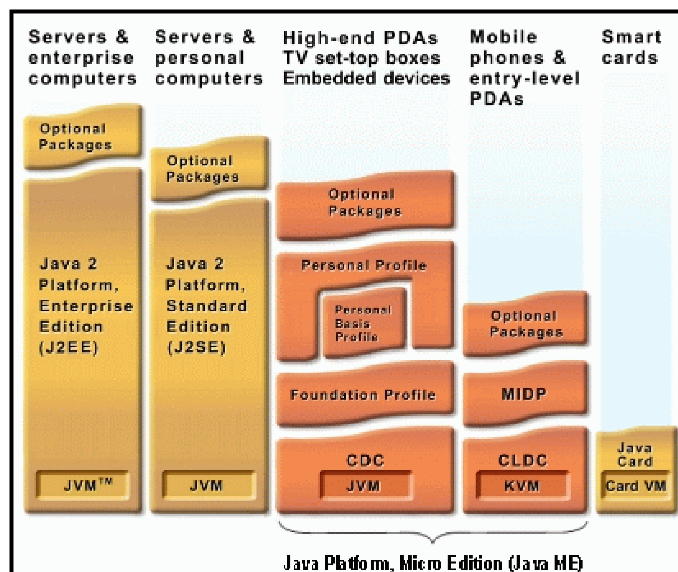
computacional (do inglês *high-end consumer devices*), como exemplos, ambientes para *set-top boxes* de TVs a cabo, dispositivos *wireless high-end* e sistemas automotivos;

b) *Connected Limited Device Configuration* (CLDC): dispositivos de menor capacidade computacional (do inglês *low-end consumer devices*), usualmente *wireless*, tais como, ambiente para telefones celulares, *pagers* e computadores portáteis.

Outra camada é a de perfil, que consiste em um conjunto de classes que possibilita aos desenvolvedores de software implementarem as aplicações de acordo com as características dos dispositivos, as quais são:

a) *Mobile Information Device Profile* (MIDP): ambiente para aplicações em dispositivos *wireless* sobre a CLDC;
b) *foundation profile*: perfil de base para dispositivos em rede sem *Graphical User Interface* (GUI) sobre CDC;
c) *personal basis RMI Profiles*: suporte básico a gráficos e *Remote Method Invocation* (RMI) para dispositivos CDC.

A terceira camada é a de pacotes opcionais, composta por um conjunto de APIs para tecnologias específicas. A Figura 1 representa um resumo dos componentes da tecnologia J2ME e como se relacionam com outras tecnologias Java.



Fonte: Sun Microsystem (2009a).
Figura 1 – Componentes do Java Micro Edition

2.3 MOBILE 3D GRAPHICS API

A Mobile 3D Graphics API (M3G) é um pacote adicional para J2ME voltado para a classe de dispositivos CLDC, em particular para o perfil MIDP. Esta interface é orientada a objeto e consiste de 30 classes que podem ser usadas para desenhar complexas cenas animadas tridimensionais. É flexível o suficiente para uma grande faixa de aplicações, incluindo jogos, mensagens animadas, visualização de produtos, proteção de tela e interfaces customizáveis (SUN MICROSYSTEMS, 2009b).

Dispositivos móveis podem caracterizar uma grande variedade de capacidades gráficas. Alguns dispositivos alvos podem ter visor preto e branco em resolução 96X64, enquanto outros como o Ipod podem ter uma tela de 480X320 com 24 bits de cores. No futuro, até resoluções e definição de cores mais altas são esperadas. Similarmente, a velocidade do processador pode variar de dezenas a centenas de MHz. A API deve acomodar tais diferenças, e ser capaz de tirar toda vantagem do incremento das capacidades especiais de hardware, como aceleradores 3D. A arquitetura da API permite ser implementada completamente dentro do software ou tomar vantagem do hardware presente no dispositivo (SUN MICROSYSTEMS, 2009b).

Os requisitos principais da API são:

a) habilitar o uso de gráficos 3D numa variedade de diferentes aplicações;
b) não assumir a existência de dispositivos de grande poder de processamento;
c) não impor limites no tamanho ou complexidade do conteúdo 3D;
d) suportar recursos sofisticados para plataformas de baixo poder de processamento;
e) ocupar pouco espaço em *Random Access Memory* (RAM) e *Read Only Memory* (ROM);
f) suportar o modo de acesso preservado (grafos de cena) e imediato.

2.3.1 Mobile 3D Graphics Engine

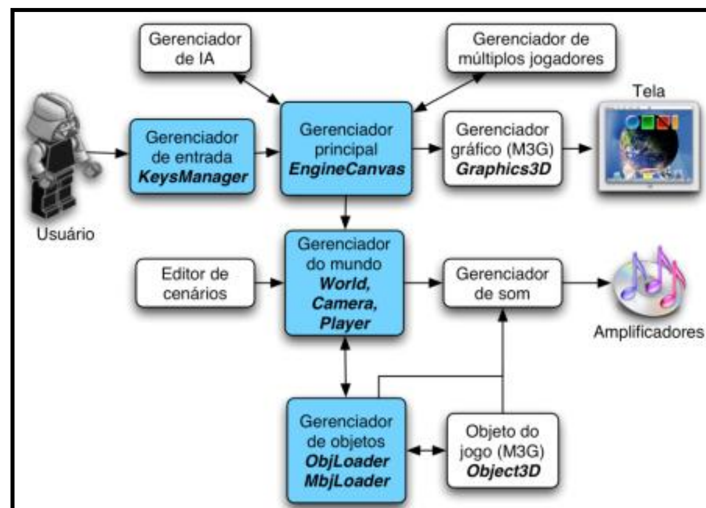
Mobile 3D Graphics Engine (M3GE) é um motor de jogos que utiliza as bibliotecas gráficas da M3G. Inicialmente concebida como um protótipo, possui apenas alguns módulos implementados de um motor padrão tais como: importação e renderização de ambientes 3D,

criação de câmeras, tratamento de eventos, movimentação de personagens no cenário e tratamento de colisão (PAMPLONA, 2005).

A M3GE já possui as seguintes funcionalidades:

a) define um formato de arquivo Mobile oBJect file (MBJ) que facilita a importação de modelos 3D através de um utilitário que converte arquivos do formato OBJect file (OBJ) Wavefront Technologies, que é considerado um formato padrão por ser suportado em várias ferramentas de modelagem 3D;
b) troca de câmeras no cenário;
c) movimentação de personagens no cenário;
d) uma forma simples de detecção de colisão;
e) modelo de eventos.

O motor M3GE foi projetado anexo à M3G, significando que se pode acessar diretamente a M3G na aplicação, se necessário. Isso dá maior flexibilidade na implementação dos jogos. O motor foi dividido em dois grandes componentes: o responsável pela leitura dos arquivos de entrada e o *core*, além de um terceiro componente de classes utilitárias (GOMES; PAMPLONA, 2005, p. 39). A Figura 2 ilustra a arquitetura da M3GE, seus principais componentes e classes.



Fonte: Gomes e Pamplona (2005, p. 2).

Figura 2 - Módulos implementados no M3GE

Pamplona (2005, p. 77) concluiu que é sempre necessária a preocupação com algoritmos velozes e alguns dos objetivos mais próximos são melhorar as rotinas de colisão e implementar simulação física de corpos rígidos, como pulo do personagem jogador, gravidade e o arremesso de objetos.

2.4 DINÂMICA DE CORPOS RÍGIDOS

O domínio de estudo da mecânica esta dividido em estática, que estuda os corpos em repouso, e dinâmica, que foca nos corpos em movimento e o efeito das forças sobre eles (BOURG, 2002, p. 52-53). Dentro da dinâmica, existem domínios até mais específicos chamados cinemática, que foca no movimento dos corpos independente das forças, e cinética, que considera ambos movimento e forças que afetam ou agem no corpo em movimento (BOURG, 2002, p. 49).

Um corpo rígido é composto de um sistema de partículas que permanecem numa distância fixa sem relativa translação ou rotação entre elas, ou seja, não sofre deformação. Quando um corpo rígido rotaciona, geralmente rotaciona através do eixo que passa através do seu centro de massa, a menos que o corpo esteja articulado em outro ponto através do qual é forçado a rotacionar.

2.5 DETECÇÃO DE COLISÃO

A detecção de colisão é um problema de geometria, envolvendo a questão se e onde dois ou mais objetos colidem. Cada objeto no jogo pode estar colidindo com outro objeto, e cada um dos pares precisam ser checados. Estes dois problemas chave (muitas colisões possíveis e a checagem dispendiosa) têm soluções independentes, as quais são (MILLINGTON, 2007, p. 231):

a) *coarse collision detection*: tenta-se encontrar um grupo de objetos com probabilidade de contato um com o outro. É um processo tipicamente rápido e usa regras de manuseio e estruturas de dados especializadas para eliminar a maioria das checagens de possíveis colisões;
b) *fine collision detection*: a segunda etapa considera as colisões candidatas e faz a checagem para determinar exatamente onde eles estão em contato.

Após dois ou mais objetos colidirem, deve-se tratar a resposta da colisão. Nesta etapa são cruciais o cálculo da massa, centro de massa e momento de inércia, chamadas de propriedades da massa.

2.6 CONCEITOS DE CINEMÁTICA PARA CORPOS RÍGIDOS

Neste capítulo serão apresentados alguns conceitos comuns usados na física, mas que são particularmente importantes para o entendimento deste trabalho.

2.6.1 Rotação Angular

A orientação, Ω, é definida como a diferença angular entre os dois conjuntos de eixos de coordenadas (local e global), como representada na Figura 1Figura 3. Isso é chamado ângulo de Euler e a unidade para deslocamento angular são radianos (rad) (BOURG, 2002, p. 50).

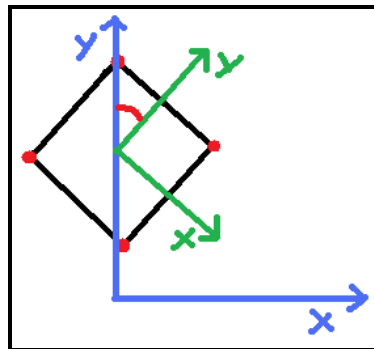


Figura 3 – Orientação como diferença entre eixos de coordenadas

A taxa com que Ω (orientação) muda, é velocidade angular, ω. Sua unidade é radianos por segundo (rad/s), como mostra o Quadro 1.

$$\omega = d\Omega/dt$$

Fonte: Bourg (2002, p. 50).

Quadro 1 – Velocidade angular

Do mesmo modo, a taxa com que ω (velocidade angular) muda é a aceleração angular, α, medida como radianos por segundo ao quadrado (rad/s²), como mostra o Quadro 2.

$$\alpha = d\omega/dt = d^2\Omega/dt^2$$

Fonte: Bourg (2002, p. 50).

Quadro 2 – Aceleração angular

2.6.2 Movimento angular

Quando um corpo rotaciona num dado eixo, todo ponto dele se movimenta numa trajetória circular em volta do eixo de rotação, causando movimento linear de cada partícula formando o corpo, como representado na Figura 4. Esse movimento linear é em adição ao movimento linear do centro de massa do corpo (BOURG, 2002, p. 51). E para rastrear a orientação de um corpo conforme ele rotaciona num plano tridimencional (3D), são necessários três sistemas de coordenadas locais fixadas no seu centro de massa (BOURG, 2002, p. 49).

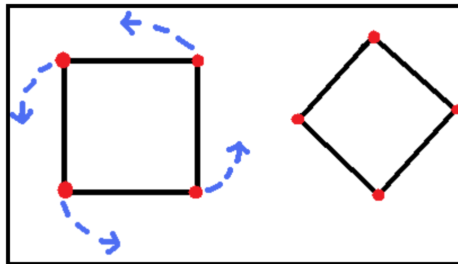


Figura 4 – Trajetória circular das partículas durante a rotação

A extensão do arco da trajetória circulada por uma partícula no corpo rígido é uma função da distância do eixo de rotação à partícula e deslocamento angular. Usa-se c para denotar o comprimento do arco e r para denotar a distância do eixo de rotação para a partícula, como mostra o Quadro 3.

$$c = r\Omega$$

Fonte: Bourg (2002, p. 51).

Quadro 3 – Comprimento do arco no deslocamento angular

Se derivar essa fórmula com respeito ao tempo, obtém-se uma equação relacionando a velocidade linear das partículas à medida que elas se movem ao longo de seu caminho circular para a velocidade angular do corpo, como mostra o Quadro 4.

$$dc/dt = r\, d\Omega/dt$$

$$v = r\omega$$

Fonte: Bourg (2002, p. 52).

Quadro 4 – Velocidade linear em volta de um eixo

Essa velocidade como um vetor, é tangente ao caminho circular percorrido pela partícula. Derivando a equação de velocidade linear, revela-se a fórmula para aceleração linear tangencial como uma função da aceleração angular, como mostra o Quadro 5.

$$a_t = r\,\alpha$$

Fonte: Bourg (2002, p. 52).
Quadro 5 – Aceleração linear tangencial

2.6.3 Aceleração centrípeta

Há também outro componente da aceleração da partícula que resulta da rotação do corpo, chamado aceleração centrípeta. Esse componente é perpendicular ao caminho circular da partícula e é sempre direcionado a partir do eixo de rotação, como mostra a Figura 5. Para aceleração centrípeta existem duas fórmulas possíveis, mostradas no Quadro 6.

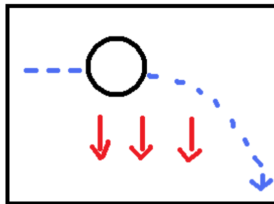


Figura 5 – Aceleração centrípeta resultante da gravidade

$$a_n = v^2/r$$

$$a_n = r\omega^2$$

Fonte: Bourg (2002, p. 53).
Quadro 6 – Aceleração centrípeta

Se extrair o produto vetorial da velocidade angular e do eixo de rotação até a partícula, tem-se o vetor da velocidade tangencial com ambas magnitude e direção da velocidade tangencial, como mostra o Quadro 7.

$$v = \omega \times r$$

Fonte: Bourg (2002, p. 54).
Quadro 7 – Velocidade tangencial como vetor de magnitude e direção

Para determinar os vetores para aceleração tangencial e centrípeta, são necessárias as equações mostradas no Quadro 8.

$$a_n = \omega \times (\omega \times r)$$
$$a_t = \alpha \times \omega$$

Fonte: Bourg (2002, p. 54).
Quadro 8 – Aceleração tangencial e centrípeta como vetores

2.7 PROPRIEDADES DE MASSA

As propriedades de massa cruciais são a massa, centro de massa e momento de inércia, pois são funções dela o movimento de um corpo e a resposta para uma dada força (BOURG, 2002, p. 5).

2.7.1 Massa e massa total

É a medida da quantidade de matéria num corpo. Pode-se pensar numa medida da resistência a mudança na posição (movimento).

A massa total de um corpo é a soma das massas de todas as partículas formando o corpo. Para um corpo de densidade uniforme a massa é expressa pela integral mostrada no Quadro 9.

$$\mathrm{m} = \int pdV = \int \mathrm{dV}$$

Fonte: Bourg (2002, p. 6).

Quadro 9 – Massa total na forma integral

Mas ao invés de tomar o volume integral para encontrar a massa, simplesmente divide-se em corpos componentes de massa facilmente calculável e soma-se a massa de todos componentes para chegar à massa total (BOURG, 2002, p. 6-7), como mostra o Quadro 10.

$$\mathrm{W_{total}} = W_{corpo1} + W_{corpo2} + \ ...$$

Fonte: Bourg (2002, p. 7).

Quadro 10 – Massa total como soma das massas individuais

2.7.2 Centro de Massa

Centro de massa é um ponto cuja massa do corpo está equilibradamente distribuída e através do qual qualquer força pode agir sem resultar em rotação, como mostrado na Figura 6.

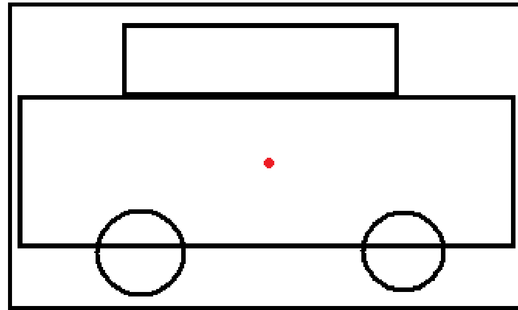


Figura 6 – Centro de massa em um objeto composto

Para obtê-la, divide-se o corpo num infinito numero pequenas quantidades de massa com o centro de cada massa especificada relativas ao eixo de coordenadas de referência. Depois obtenha o "primeiro momento" de cada massa, que é a massa vezes a distância do eixo de referência, então soma-se todos esses momentos (BOURG, 2002, p. 5-7).

Por último, divide-se a soma dos momentos pela massa total do corpo, sujeitando a coordenada do centro de massa em relação ao eixo de referência. Esse cálculo deve ser feito uma vez para cada dimensão (BOURG, 2002, p. 5-7), como mostra o Quadro 11.

$$X_{cg\,body} = \{(X_{cg\ corpo\ 1})(W_{corpo\ 1}) + (X_{cg\ corpo\ 2})(W_{corpo\ 2}) + \cdots\} / W_{total}$$
$$Y_{cg\,body} = \{(Y_{cg\ corpo\ 1})(W_{corpo\ 1}) + (Y_{cg\ corpo\ 2})(W_{corpo\ 2}) + \cdots\} / W_{total}$$
$$Z_{cg\,body} = \{(Z_{cg\ corpo\ 1})(W_{corpo\ 1}) + (Z_{cg\ corpo\ 2})(W_{corpo\ 2}) + \cdots\} / W_{total}$$

cg: centro de massa

Fonte: Bourg (2002, p. 13).

Quadro 11 – Centro de massa em cada coordena

2.7.3 Momento de inércia

É uma medida quantitativa da distribuição radial de massa de um corpo ao longo de um eixo especifico de rotação. Mede a resistência do corpo ao movimento de rotação.

Aqui, o ponto onde se aplica a força, ou alavanca, é a distância perpendicular do eixo de coordenada em volta do qual se quer calcular o momento de inércia, para o centro de massa, mostrado na Figura 7. O segundo momento, então é produto da massa vezes essa distância ao quadrado. Então quando calculando o momento de inércia em torno de um dado eixo, essa distância, `r`, está nos planos adjacentes (BOURG, 2002, p. 8-9), como mostra o Quadro 12.

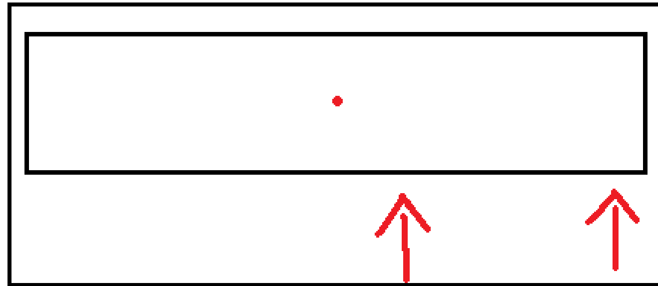


Figura 7 – Resistência a rotação detende da distancia ao eixo que uma força é aplicada

$$I_{xx} = \int r_x^2\, dm = \int (y^2 + z^2)\, dm$$

$$I_{yy} = \int r_y^2\, dm = \int (z^2 + x^2)\, dm$$

$$I_{zz} = \int r_z^2\, dm = \int (x^2 + y^2)\, dm$$

Fonte: Bourg (2002, p. 9).

Quadro 12 – Momento de inércia como integral

Na prática, o momento de inércia é aproximado usando fórmulas para formas simples de densidade uniforme. Dividindo-se o corpo em componentes menores, considerando o nível de precisão desejado (BOURG, 2002, p. 10-11), como mostra o Quadro 13.

Cilindro circular:

$$I_{xx} = I_{yy} = \left(\frac{1}{4}\right) mr^2 + \left(\frac{1}{12}\right) ml^2$$

$$I_{zz} = (1/2)mr^2$$

Envoltório de cilindro circular:

$$I_{xx} = I_{yy} = \left(\frac{1}{2}\right) mr^2 + \left(\frac{1}{12}\right) ml^2$$

$$I_{zz} = mr^2$$

Cilindro retangular:

$$I_{xx} = \left(\frac{1}{2}\right) m\,(a^2 + l^2)$$

$$I_{yy} = \left(\frac{1}{12}\right) m\,(b^2 + l^2)$$

$$I_{zz} = \left(\frac{1}{12}\right) m\,(a^2 + b^2)$$

Esfera:

$$I_{xx} = I_{yy} = I_{zz} = (2/5)mr^2$$

Envoltório de esfera:

$$I_{xx} = I_{yy} = I_{zz} = (2/3)mr^2$$

Fonte: Bourg (2002, p. 10).

Quadro 13 – Momentos de inércia para as formas mais comuns

O primeiro passo é calcular o momento de inércia de cada componente em volta de seu próprio eixo neutro utilizando a fórmula correspondente sua forma geométrica (BOURG, 2002, p. 12), como exemplifica o Quadro 14.

$$I_{o\ componente\ 1} = \left(\frac{m}{12}\right)(w^2 + L^2)$$
$$I_{o\ componente\ 2} = \left(\frac{m}{12}\right)(w^2 + L^2)$$
$$I_{o\ componente\ n} = \left(\frac{m}{12}\right)(w^2 + L^2)$$

Fonte: Bourg (2002, p. 12).
Quadro 14 – Exemplo de calculo do cilindro retangular

Depois é preciso usar o teorema dos eixos paralelos para transferir esses momentos para o eixo neutro do corpo, localizado no centro de gravidade calculado previamente. Para fazer isso, a distância do centro de gravidade para o centro de gravidade de cada componente deve ser encontrada (BOURG, 2002, p. 12-15). Antes de aplicar o teorema é necessário saber a distância ao quadrado de cada componente do centro de gravidade, como mostra o Quadro 15.

$$d^2_{componente\ 1} = (x_{cg\ componente\ 1} - X_{cg})^2 + (y_{cg\ componente\ 1} - Y_{cg})^2$$
$$d^2_{componente\ 2} = (x_{cg\ componente\ 2} - X_{cg})^2 + (y_{cg\ componente\ 2} - Y_{cg})^2$$
$$d^2_{componente\ n} = (x_{cg\ componente\ n} - X_{cg})^2 + (y_{cg\ componente\ n} - Y_{cg})^2$$

Fonte: Bourg (2002, p. 14).
Quadro 15 – Distância ao quadrado de cada componente do centro de gravidade

Agora aplicando o teorema dos eixos paralelos, mostrado no Quadro 16.

$$I_{cg\ componente\ 1} = I_o + md^2$$
$$I_{cg\ componente\ 2} = I_o + md^2$$
$$I_{cg\ componente\ n} = I_o + md^2$$

Fonte: Bourg (2002, p. 14).
Quadro 16 – Teorema dos eixos aplicado a cada componente do corpo

Finalmente pode-se obter o momento de inércia total do corpo em volta de seu próprio eixo neutro assumindo a contribuição `Icg` de cada componente (BOURG, 2002, p. 15), como mostra o Quadro 17.

$$I_{cg\ total} = I_{cg\ componente\ 1} + I_{cg\ componente\ 2} + I_{cg\ componente}$$

Fonte: Bourg (2002, p. 16).
Quadro 17 – Momento de inércia total de um ponto em volta do eixo neutro

Porém, na dinâmica de corpos rígidos em 3D o corpo pode rotacionar em torno de

qualquer eixo, não necessariamente um dos eixos de coordenada.

2.8 RESPOSTA DE COLISÃO

Resposta de colisão é um problema de física envolvendo o movimento de dois ou mais objetos depois que eles colidiram. Quando um corpo rígido rotaciona, geralmente rotaciona através do eixo que passa através do seu centro de massa, a menos que o corpo esteja articulado em outro ponto através do qual é forçado a rotacionar (BOURG, 2002). Lidar com corpos rígidos envolve dois aspectos distintos:

a) rastrear a translação do centro de massa;
b) rastrear a rotação do corpo.

O tratamento apresentado neste trabalho baseia-se nos princípios de impacto Newtonianos, onde os corpos são tratados como rígidos, independente da construção e matéria. Ou seja, é uma idealização onde eles não se deformam.

Existem dois tipos de problemas em cinética, um tipo é quando se sabe ou se pode estimar aceleração do corpo usando cinemática, e deve-se resolver a(s) forca(s) agindo no corpo. O outro tipo é quando se sabe ou pode-se estimar a(s) força(s) agindo no corpo, e precisa-se resolver a aceleração resultante do corpo (subsequentemente sua velocidade e deslocamento). Para determinar a resposta da colisão lida-se com o segundo tipo. BOURG (2002) descreve o procedimento geral para resolver a resposta de colisão, como sendo:

a) calcular as propriedades de massa do corpo (massa, centro de massa e momento de inércia);
b) identificar e quantificar todas forças e momentos agindo no corpo;
c) tomar o vetor força de todas forças e momentos;
d) resolver as equações de movimento para aceleração linear e angular;
e) integrar em relação ao tempo para encontrar velocidade linear e angular;
f) integrar de novo em relação ao tempo para encontrar o deslocamento linear e angular.

2.8.1 IMPACTO

O impacto conta com o principio de Newton da conservação de momento, que declara que quando um sistema de corpos rígidos colidem, a quantidade de movimento é conservado. Isso significa que para corpos de massa constante, a soma de suas massas vezes suas respectivas velocidades antes do impacto serão iguais após o impacto, como mostra o Quadro 18.

$$m_1 v_{1-} + m_2 v_{2-} = m_1 v_{1+} + m_2 v_{2+}$$

Fonte: Bourg (2002, p. 89).

Quadro 18 – Conservação de momento

Como assume-se que o corpo não deforma, não há perda de energia cinética. O que classifica a colisão como do tipo elástica em oposição à inelástica.

A energia cinética é função da velocidade do corpo e sua massa, e a energia cinética angular, uma função da inércia do corpo e velocidade angular, como mostra o Quadro 19.

$$KE_{linear} = (1/2)mv^2$$

$$KE_{angular} = (1/2)I\omega^2$$

Fonte: Bourg (2002, p. 90).

Quadro 19 – Energia cinética linear e angular

O grau de elasticidade do impacto usado por BOURG (2002, p. 90), é a relação entre a velocidade relativa de separação à velocidade relativa de aproximação dos objetos colidindo, onde e é o coeficiente de restituição e é função do material do objeto, construção e geometria. Para colisões perfeitamente inelásticas, e vale zero, e para perfeitamente elásticas, e vale 1, como mostra o Quadro 20.

$$e = -(v_{1+} - v_{2+})/(v_{1-} - v_{2-})$$

Fonte: Bourg (2002, p. 90).

Quadro 20 – Coeficiente de restituição

Quando a linha de ação de impacto é perpendicular às superfícies de colisão, então é sem atrito. Quando a velocidade dos corpos está paralela a linha de ação, o impacto é dito direto. Quando a linha de ação passa através do cento de massa, a colisão é dita ser central. O impacto direto central ocorre quando as linhas de ação passam através do centro de massa e suas velocidades estão paralelas. Quando as velocidades não estão paralelas o impacto é obliquo.

2.8.2 IMPULSO LINEAR E ANGULAR

Após ocorrer o impacto entre dois corpos, ele dará origem a duas forças diferentes, que como mostra a Figura 8, serão funções tanto da intensidade como da distancia do eixo de rotação.

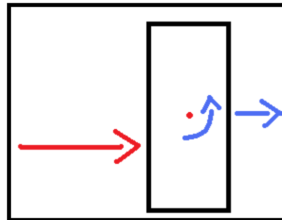


Figura 8 – Impulso alterando ambas velocidade linear e angular

Em simulações de tempo real na qual objetos, especialmente de formas arbitrárias, podem colidir, é melhor uma aproximação mais geral onde são representados por formas geométricas mais simples.

Lidando com esferas, a única fórmula de impulso necessária é para impulso linear, na qual permite calcular as novas velocidades dos objetos após o impacto, mostrada no Quadro 21.

$$J = m(v_+ - v_-)$$

Fonte: Bourg (2002, p. 96).
Quadro 21 – Impulso linear

Pode–se determinar a velocidade de cada corpo após impacto com a fórmula do Quadro 22.

Para corpo 1: $v_{1+} = J/m_1 + v_{1-}$
Para corpo 2: $v_{2+} = -J/m_2 + v_{2-}$

Fonte: Bourg (2002, p. 96).
Quadro 22 – Velocidade após impacto

Para rotaciona-los é necessário uma nova equação para impulso que inclui efeitos angulares. Nesse caso a velocidade no ponto de contato é função de ambas velocidade linear e angular, como mostrado no Quadro 23.

$$v_p = v_g + (\omega \times r)$$

Fonte: Bourg (2002, p. 97).
Quadro 23 – Velocidade no ponto de contato

A formula para J que considera ambos efeitos linear e angular, na qual pode usar para encontrar a velocidade angular e linear de cada corpo imediatamente após impacto, é mostrada no Quadro 24.

$J = -v_r\,(e+1)/\{1/m_1 + 1/m_2 + n\cdot\{(r_1 \times n)/I_1] \times r_1 + n\cdot\{(r_2 \times n)/I_2] \times r_2\}$ v_r: velocidade relativa ao longo da linha de ação no ponto de impacto; n: vetor ao longo da linha de ação apontando para fora do corpo.

Fonte: Bourg (2002, p. 98).

Quadro 24 – Formula combinada para impulso linear e angular

Com essa nova formula para "J", pode-se calcular a mudança na velocidade linear e angular dos objetos envolvendo a colisão usando as formulas do Quadro 25.

$$v_{1+} = v_{1-} + (Jn)/m_1$$
$$v_{2+} = v_{2-} + (-Jn)/m_2$$
$$\omega_{1+} = \omega_{1-} + (r_1 \times Jn)/I_{cg}$$
$$\omega_{2+} = \omega_{2-} + (r_2 \times -Jn)/I_{cg}$$

Fonte: Bourg (2002, p. 98).

Quadro 25 – Velocidade linear e angular

2.8.3 ATRITO

Atrito age entre as superfícies para resistir à movimentação. Exceto num impacto direto, por um momento muito breve de contato os objetos irão experimentar uma força de atrito que age tangencialmente as superfícies de contato, exemplificado na Figura 9. Não muda apenas a velocidade na direção tangencial, como também cria um momento (torque) nos objetos, tendendo a mudar suas velocidades angulares. Esse impulso tangencial combinado com o impulso normal resulta em uma linha efetiva de ação do impulso total de colisão que não é mais perpendicular a superfície de contato, como mostra o Quadro 26.

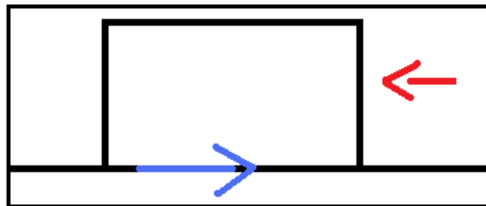


Figura 9 – Força de atrito causando resistência à movimentação

$\tan ø = F_f / F_n = \mu$
F_f: força de atrito tangencial
F_n: força normal de impacto

Fonte: Bourg (2002, p. 98).

Quadro 26 – Impulso total

Desde que a proporção do atrito tangencial à força de colisão normal é igual ao coeficiente de atrito, pode-se ainda relacioná-la ao ângulo ø.

Em adição à força de atrito mudando a velocidade linear do corpo na direção tangencial, ela também muda a velocidade angular do copo. Desde que a força de atrito está

agindo na superfície do corpo a alguma distância de seu cento de gravidade, isso cria um momento (torque) ao redor do centro de gravidade que o faz girar, como mostra o Quadro 27.

$$Impulso = I_{cg}/(\mu r)(\omega_{+} - \omega_{-})$$
$$\omega_{+} = (Impulso)(\mu r)/I_{cg} + \omega_{-}$$

Fonte: Bourg (2002, p. 99).

Quadro 27 – Calculo da nova velocidade angular

Para calcular as novas velocidades linear e angular, usa-se as fórmulas do Quadro 28.

$$\mathrm{v}_{1+} = \mathrm{v}_{1-} + [Jn + (\mu J)]/m_1$$
$$\mathrm{v}_{2+} = \mathrm{v}_{2-} + [-Jn + (\mu J)t]/m_2$$
$$\omega_{1+} = \omega_{1-} + \{ (r_1 \cdot [Jn + (\mu J)t] \} / I_{cg}$$
$$\omega_{2+} = \omega_{2-} + (r_2 \cdot -Jn)/I_{cg}$$

t: vetor tangente a superfície de colisão

Fonte: Bourg (2002, p. 100).

Quadro 28 – Velocidades linear e angular

Pode-se calcular o vetor tangente, t, sabendo-se os vetores normal e a velocidade relativa no mesmo plano do vetor normal, como mostra o Quadro 29.

$$t = (n + v_r) \cdot n)$$
$$t = t/|t|$$

Fonte: Bourg (2002, p. 100).

Quadro 29 – Vetor tangente

Para muitos problemas, pode-se negligenciar atrito, desde que o efeito é pequeno em comparação ao efeito do impulso normal. Porém, para alguns tipos de problemas, o atrito é crucial. Por exemplo, a trajetória de uma bola de golf ou de tênis depende fortemente do giro imposto pela colisão com o taco ou a raquete, como mostrado na Figura 10.

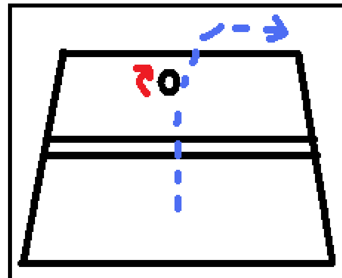


Figura 10 – Trajetória alterada pela rotação quando uma bola de tênis atinge o solo

2.9 TRABALHOS CORRELATOS

Existem tanto motores voltados para simulações físicas (conhecidos como motores de física) quanto motores de jogos com algumas implementações de física incluídas, a grande maioria, porém, disponível apenas para computadores *desktop*.

Newton Game Dynamics é uma API *middleware* livre, mas de código fechado, usada em aplicações 3D, que prima pela precisão sobre a velocidade usando um método determinístico ao invés do tradicional baseado em *Linear Complementarity Problem* (LCP) ou iteração (NEWTON GAME DYNAMICS, 2007).

PhysX é um motor de física *middleware* proprietário que inclui suporte para as placas de vídeo da NVidia de mesmo nome, que possuem uma *Physics Processing Unit* (PPU) que faz cálculos específicos e aliviam o processamento da *Central Processing Unit* (CPU). É usado pelo *framework* Unreal Engine 3 (UNREAL TECHNOLOGY, 2008).

Bullet é em *middleware* de código aberto usado pelo Blender Game Engine e que também é suportado pelo Crystal Space. É usada largamente em jogos tanto para os PC, XBox 360, PlayStation 3 e Wii. Suas formas de colisão incluem esfera, caixa, cilindro, cone, cálice convexo e não convexo e malha de triângulos. Implementa detecção de colisão convexa Gilbert Johnson Keerthi (GJK), teste de colisão *Solid Sweep*, detecção de colisão discreta e contínua, além de possuir física de corpos macios (BULLET PHYSICS, 2005).

Para a plataforma J2ME, atualmente existem apenas dois motores de jogos que incluem simulação física, Emini e DyMiX, mas eles funcionam apenas para aplicações em 2D, nos dispositivos MIDP, e sendo ambos comerciais.

O motor Emini que além de ser bastante otimizado, é o que possui o maior número de recursos como: partículas, detecção de colisão, resposta de colisão, gravidade, atrito, formas customizadas, juntas, molas e eventos (ADENSAMER, 2010). O motor DyMiX simula apenas corpos rígidos simples como retângulos e círculos, mas é rápido por usar apenas operações com ponto fixo e pode integrar o motor de física Bloft para adicionar física de corpos macios (MACNY, 2010).

3 DESENVOLVIMENTO

Esse capítulo apresenta como foi desenvolvido o módulo para o motor de jogos M3GE.

3.1 REQUISITOS PRINCIPAIS DO MÓDULO A SER DESENVOLVIDA

O módulo para física de corpos rígidos e colisão deverá:

a) ser desenvolvido na plataforma Java Micro Edition (Requisito Não Funcional - RNF);
b) ser compatível com dispositivos *CLDC* (RNF)
c) ser compatível com o motor M3GE (Requisito Funcional - RF);
d) armazenar informações sobre os objetos desenhados na tela (RF);
e) atualizar as forças envolvidas na simulação (RF);
f) detectar colisões que ocorrem entre os objetos e armazenar as informações resultantes (RF);
g) atualizar a posição dos objetos na tela, resultando em rotação e translação (RF).

3.2 ESPECIFICAÇÃO

O sistema apresentado utiliza alguns dos diagramas da Unified Modeling Language (UML). Foi utilizada uma versão *demo* da ferramenta Enterprise Architect versão 8.0 para a elaboração dos diagramas de casos de uso, de classe e de sequência.

3.2.1 Diagrama de Casos de Uso

A Figura 11 apresenta o diagrama de Casos de Uso do módulo para o motor M3GE, que adiciona efeitos da dinâmica de corpos rígidos. Como os métodos são acessados

internamente pelo próprio motor, ele é representado como sendo o ator.

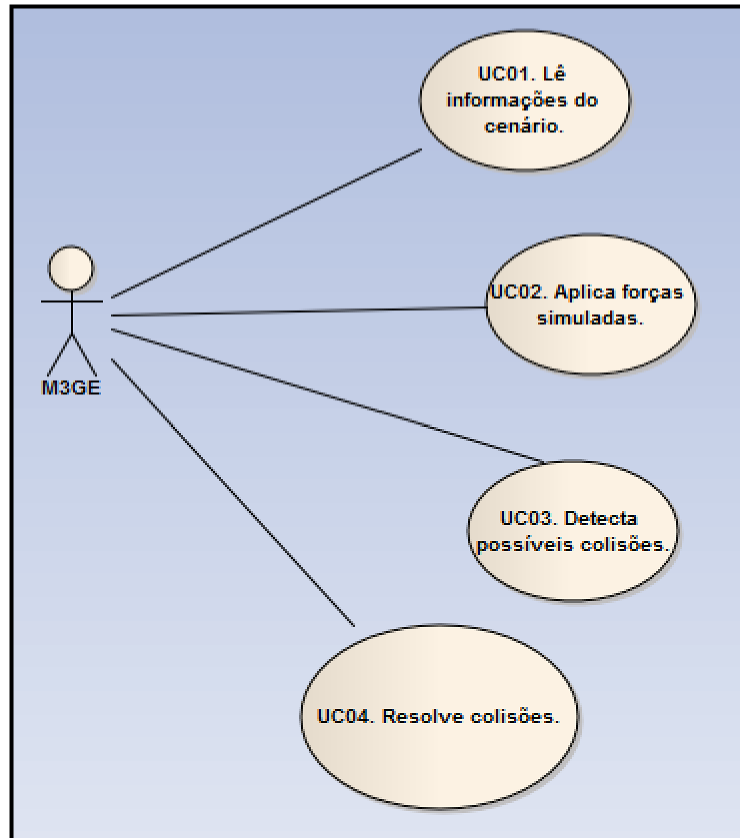


Figura 11 - Diagrama de casos de uso

O Quadro 30, Quadro 31, Quadro 32 e Quadro 33 descrevem cada um dos casos de uso com mais detalhes.

UC01 – Lê informações do cenário.	
Pré-condições	O modelo do cenário 3D deve ter sido carregado pela classe `Loader`.
Cenário Principal	1. O motor faz uma chamada ao método `loadBodies`, que percorre o grafo de cena armazenando uma referência aos objetos do tipo `Mesh`.
Pós-condições	A classe `CollisionDetection` armazena uma lista de objetos `Object3Dinfo` com informações dos objetos carregados.

Quadro 30 – Caso de uso 01 - Lê informações do cenário

UC02 – Aplica forças simuladas.	
Pré-condições	UC01.
Cenário Principal	1. O motor faz uma chamada ao método `stepSimulation`; 2. O motor calcula as forças que os objetos são submetidos, chamando a função `calcObjectForces;` 3. O motor calcula a posição final dos objetos dentro da função `stepSimulation .`
Exceção	No passo 1, caso algum objeto esteja em contato com o chão, aplica-se uma forca contrária para que ele pare de cair.
Pós-condições	As informações contidas nos objetos `Object3DInfo,` são atualizadas.

Quadro 31 – Caso de uso 02 - Aplica forças simuladas

UC03 – Detecta possíveis colisões.	
Pré-condições	UC02.
Cenário Principal	1. O motor, através da função `checkGroundPlaneContacts`, verifica se algum objeto colide com o chão; 2. O motor faz uma chamada ao método `checkForCollisions` que verifica-se se existe há colisão entre os objetos comparando cada objeto com sua *bound box*.
Exceção	No passo 1, caso algum objeto esteja penetrando em outro objeto, recalcula-se a posição original de todos objetos e testa a colisão com metade do tempo original.
Pós-condições	A classe `CollisionDetection` armazena uma lista de objetos `Collision` contendo os pontos de contato.

Quadro 32 – Caso de uso 03 - Detecta possíveis colisões

UC04 – Resolve colisões.	
Pré-condições	UC03.
Cenário Principal	1. O motor chama o método `resolveCollisions` e aplica a força de impulso angular e linear nos pontos de colisão, guardados na estrutura `Collisions`; 2. O motor atualiza a posição final dos objetos.
Pós-condições	As novas posições dos objetos são atualizadas nos objetos `Object3DInfo`.

Quadro 33 – Caso de uso 04 - Resolve colisões

3.3 DIAGRAMA DE CLASSES

Entre as classes já existentes no motor M3GE, foram tanto alteradas as classes já existentes como foram criadas novas para atender as necessidades do presente módulo desenvolvido. Na Figura 13 é apresentado um diagrama com as classes utilizadas pelo motor de física, fazendo-se um distinção entre as classes Alteras, Auxiliares, e de Operações. A seguir as classes são apresentadas com mais detalhes.

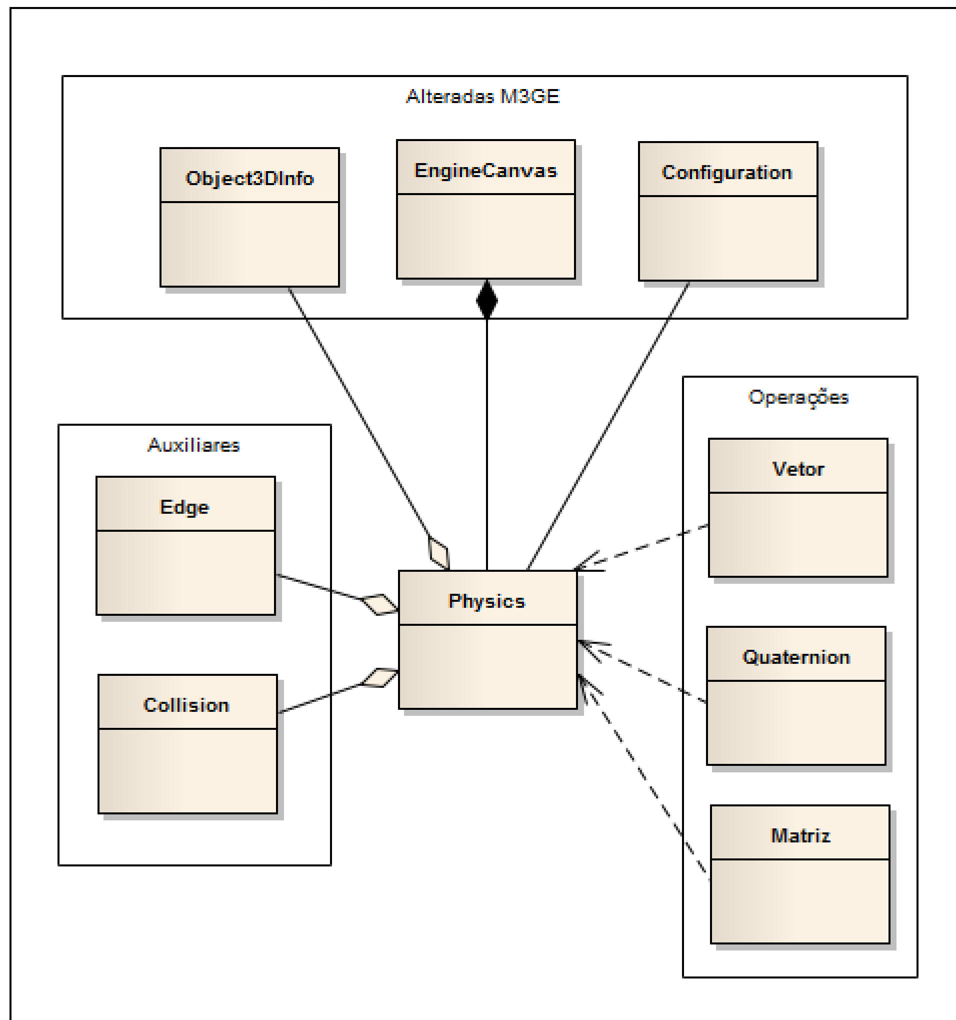


Figura 12 – Diagrama de classes

3.3.1 As Classes Vetor, Matriz e Quaternion

A três classes `Matriz`, `Vetor` e `Quaternion` mostradas na Figura 13, além de utilizadas para armazenar a estrutura que representam, possuem métodos estáticos responsáveis por todas operações com vetores, matrizes e quatérnions. Na classe `Quaternion` foram adicionadas várias constantes com valor de precisão Double apenas para uso interno das funções `aTan2()`, `asen()` e `acos()`.

Figura 13 – Classes `Matriz`, `Vetor`, `Quaternion` e suas dependências

3.3.2 As Classes `Collision` e `Edge`

A classes `Collision` e `Edge` mostradas na Figura 14 servem de estruturas auxiliares que armazenam dados temporariamente. A classe `Edge` simplesmente armazena dois índices nas variáveis `a` e `b`, que representa os pontos de uma aresta. Na classe `Collision`, são armazenados os dados resultantes de uma colisão, necessários para o cálculo da resposta.

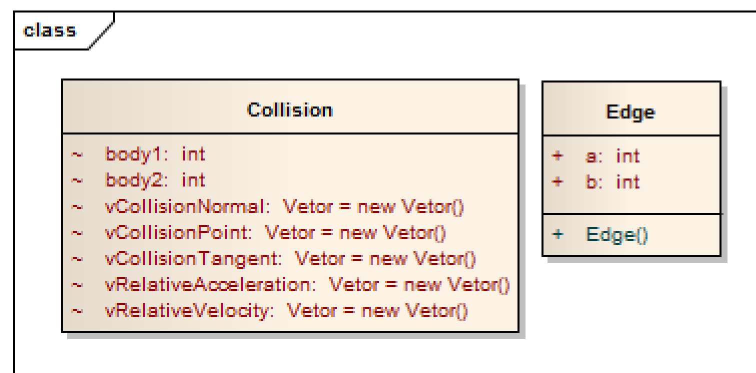


Figura 14 – Classes `Collision`, `Edge` e suas dependências

3.3.3 A Classe `Object3DInfo`

A Classe `Object3DInfo` já era existente na M3GE, e era associado um objeto dessa classe para cada nodo do grafo de cena. A estrutura foi aproveitada e foram adicionadas variáveis que guardam o estado atual do objeto, necessárias para calcular seu deslocamento a cada passo de tempo, como a aceleração, velocidade, aceleração angular, velocidade angular, posição e orientação, mostradas na Figura 15.

Figura 15 – Classe `Object3DInfo` e suas dependências

3.3.4 A Classe `Physics`

A classe `Physics`, mostrada na Figura 16, é onde estão localizados os principais métodos. O método `stepSimulation`, responsável pela integração, os método `checkBoxCollision`, que faz uma checagem do tipo *bounding box*, e o método `resolveCollisions`, que aplica a força resultante de cada um dos pontos de colisão. Também contém o método `checkGroundPlaneContact` que verifica se os corpos entraram em contato com o solo, aplicando uma força contrária para que parem de cair. Por motivo de simplificação, e utilizarem variáveis que são compartilhadas, os métodos que fazem a detecção e resposta de colisão não foram separados em classes diferentes.

Outro método presente nesta classe, e que é essencial para a integração com a M3GE é `updatePosition`, que integra o sistema de orientação e posicionamento armazenados na classe `Object3DInfo` com o grafo de cena utilizado pela biblioteca M3G, portanto deve ser chamando sempre antes de cada frame ser desenhado, caso contrário a posição dos objetos gerenciado não será alterada na tela.

Nesta classe estão presentes todas as constantes que podem ser alteradas antes ou até mesmo durante a execução de um jogo, como gravidade, coeficiente de restituição, e tolerância para colisão.

Figura 16 – Classe `Psysics` e suas dependências

3.3.5 Diagrama de Sequência

O diagrama de sequência, apresentado na Figura 17, dá uma visão de como ocorre a chamada aos principais métodos necessário para simular dinâmica de corpos rígidos durante a execução do motor M3GE.

Primeiramente, classe `Midlet`, implementada pelo desenvolvedor, deve criar uma instância da classe `EngineCanvas`, onde serão instanciadas o restante das classes utilizadas pela M3GE, incluindo a classe `Physics`.

Após a classe `EngineCanvas` ter sido instanciada, o método `createScene` monta o grafo de cena e o método `loadBodies` presente da classe `Physics`, varre este grafo de cena procurando objetos da classe `Mesh` - representando uma malha de polígonos - que serão carregados na variável `bodies` e posteriormente, o desenvolvedor pode incluir ou excluir aqueles que serão gerenciados pela classe `Physics`.

A função `createScene` da classe `Engine` canvas, ao ser invocada pelo desenvolvedor, instancia um objeto `Timer`. Este objeto faz uma chamada periódica à função `stepSimulation`, da classe `Physics` com um valor configurado dentro da mesma. O método `stepSimulation` faz uma chamada primeiro à `checkGroudPlaneContacts` - responsável pelo tratamento do contato com o chão, depois à `checkForCollision`, que verifica se ocorrem colisões, e caso existam, são tratadas no método `resolveCollision`. Por fim, o método `updateBodies` atualiza a posição dos objetos no grafo de cena através de funções nativas da M3G.

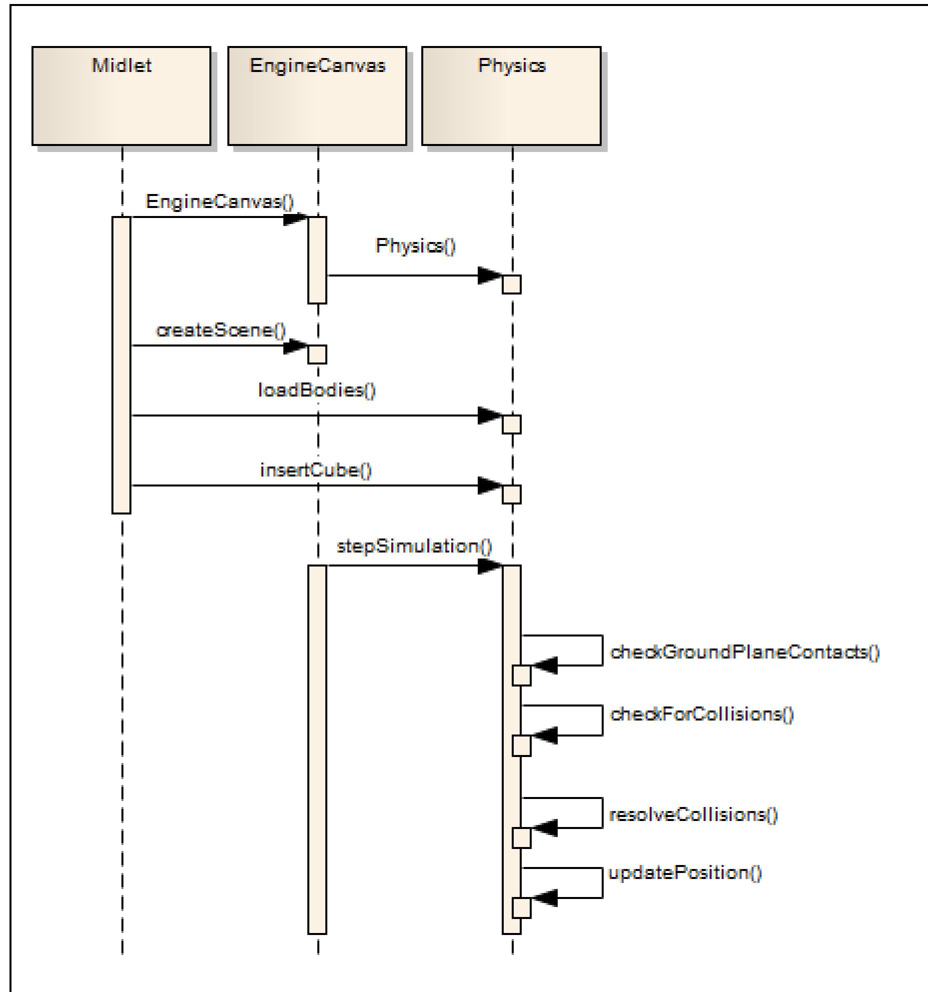


Figura 17 – Diagrama de seqüência

3.4 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas e a operacionalidade da implementação.

3.4.1 Técnicas e ferramentas utilizadas

O ambiente de desenvolvimento utilizado para a implementação do módulo foi o Eclipse versão 3.4.2., configurado para o tipo de dispositivo CLDC e perfil MIDP. Esta ferramenta de desenvolvimento foi escolhida por oferecer compatibilidade com o projeto já disponível do motor M3GE. Os principais algoritmos utilizados são os de integração,

detecção de colisão e resposta de colisão, descritos a seguir.

3.4.1.1 Integração

A função `stepSimulation` começa chamando o método `calcObjectForces`, na linha 10, que calcula as forças agindo sobre os corpos. Então entra num *loop*, linha 11, até que todas colisões sejam resolvidas, não haja penetração ou um limite de tentativas for atingida. Dentro deste *loop*, após a posição dos corpos ser atualizada, é feita uma chamada ao método `checkForCollisions`, que retorna `1` para colisão, `-1` para penetração, `0` para não colisão e `2` para contato.

Caso seja verificada uma penetração, nas linha 23, o tempo decorrido da atualização dos objetos, `dt`, é diminuído pela metade e a posição original recalcula num intervalo de tempo cada vez menor, até que não haja penetração.

```
1  public void stepSimulation(float dtime)
2  {
3              …
4              Int                  check = NOCOLLISION;
5              Int                  c = 0;
6              boolean        pencheck = DOPENCHECK;
7              boolean checkAgain = false;
8
9              // Calculate all of the forces and moments:
10             calcObjectForces();
11     do
12     {
13             // Integrate
14             for(i=0; i<bodies.length; i++)
15             {
16                          …
17             }
18
19             // Handle collisions
20             check = checkForCollisions(pencheck);
21             if(check == COLLISION)
22                          resolveCollisions();
23             else if(check == PENETRATING)
24             {
25                          dt = dt/2;
26             …
27                          c++;
28                          if(c < 3)
29                                  chekAgain = true;
30                          else
31                          {
32                                  pencheck = false;
33                                  checkAgain = true;
34                          }
35             }
36             else{
37              chekAgain = false;
38             }
39     }while(chekAgain);
40 }
```

Quadro 34 – Rotina de integração

3.4.1.2 Detecção de colisão

O método `checkForCollisions`, mostrado no Quadro 35 começa assumindo que não haverá colisão. Dentro do `for`, na linha 8, cada corpo é comparado com os demais. Primeiramente é feito um rápido teste com o maior comprimento como se uma esfera o estivesse envolvendo. Se as esferas que envolve os dois corpos estiverem encostando, testadas na linha 18, nenhum teste adicional é necessário entre eles. Caso estejam, antes é verificado se

há penetração, na linha 16. Se também não houver penetração, na linha 20 é feita uma chamada ao método `checkBoxCollision`, que verifica se as duas caixas que envolvem os objetos (*bounding box*) estão colidindo.

Como a penetração tem um tratamento diferente, onde o tempo do passo é diminuído, o método só prossegue se não houver penetração, mostrado na linha 40. Cada corpo agora é verificado se está colidindo com o chão, na linha 44. Se estiver apenas colidindo, os dados da colisão são armazenados em `collisionData`, linha 54, e o método retorna o *status* do copo.

```
1   public int checkForCollisions(boolean pencheck)
2   {
3           int   status = NOCOLLISION;
4   …
5           int check = NOCOLLISION;
6   …
7           // check object collisions with each other
8           for(i=0; i<bodies.length; i++)
9           {
10                  for(j=0; j<bodies.length; j++)
11                          if(j!=i) // don't check object against itself
12                          {
13                                  // do a bounding sphere check first
14                                  d =
15  Vetor.minus(bodies[i].vPosition,bodies[j].vPosition);
16                                  if(d.magnitude() < (bodies[i].fRadius +
17  bodies[j].fRadius))
18                                  {// possible collision
19                                          if(pencheck)
20                                                  check = isPenetrating(i, j);
21                                          if(check == PENETRATING)
22                                          {
23                                                  status = PENETRATING;
24                                                  break;
25                                          } else {
26                                                  check =
27  checkBoxCollision(collisionData, i, j, k);
28                                                  if(check == COLLISION)
29                                                  {
30                                                          status = COLLISION;
31                                                          k++;
32                                                  }
33                                          }
34                                  }
35                          }
36                          if(status == PENETRATING)
37                                  break;
38          }
39
40          if(status != PENETRATING)
41          {
42                  // check object collisions with the ground
43                  for(i=0; i<bodies.length; i++)
44                  {
45                          if(pencheck)
46                                  check = isPenetratingGround(i);
47                          if(check == PENETRATING)
48                                  if(check == COLLISION)
49                                  {
50                                          status = COLLISION;
51                                          k++;
52                                  }
53                          }
54                  }
55          }
56          return status;
57  }
```

Quadro 35 – Verificação de colisão

Para o cálculo da colisão, são chamados dois testes diferentes, mostrados no Quadro 36, são verificados os contatos com os pares vértice-vértice e vértice-plano.

```
1 status1 = checkVertexFaceCollisions(collisionData, body1, body2);
2 status2 = checkEdgeEdgeCollisions(collisionData, body1, body2);
```

Quadro 36 – Verificações de colisão feitas para a *bounding box*

Em aresta-vértice é calculado o vetor normal (com módulo 1, apenas para indicar direção), que é sempre perpendicular a aresta. Em `checkEdgeEdgeCollisions`, calcula-se uma linha entre os centros de gravidade e é tomado o vetor normal paralelo a elas.

Os métodos determinam o exato ponto de contato, porém não lidam com múltiplos pontos de contato. O ponto de colisão é simplesmente a coordenada do vértice envolvido, em coordenadas globais que para serem úteis no cálculo do deslocamento angular, são novamente convertidas em coordenadas locais. Ainda, determina-se a velocidade relativa, entre os pontos de impacto, que são função da velocidade linear e angular. Para determinar se um vértice em consideração está de fato colidindo com uma aresta ou face, deve-se verificar se a distância do vértice está dentro da sua tolerância de colisão, `tol`, e se os pontos de contato estão se movendo em sentido oposto um ao outro, `vrt`.

Depois da verificação vértice-vértice e vértice-plano, verifica-se a penetração, que é uma verificação do tipo ponto no plano, para determinar se algum vértice de um polígono está em contato com uma das faces do outro polígono.

Caso os método para detecção sejam trocados pelo desenvolvedor, ele deverá substituir os métodos que atribuem o `status`, no Quadro 36 ou adicionar mais status a serem testados, respeitando apenas os valor de retorno, mostrados no Quadro 37 e os dados armazenando os dados necessários para a resposta – corpos envolvidos, vetor normal da colisão, ponto de colisão, velocidade relativa e tangente da colisão - mostrados no Quadro 38.

```
1 private static final int NOCOLLISION = 0;
2 private static final int COLLISION = 1;
3 private static final int PENETRATING = -1;
4 private static final int CONTACT = 2;
```

Quadro 37 – Constantes de retorno para os métodos de colisão

```
1 collisionData[k].body1 = body1;
2 collisionData[k].body2 = body2;
3 collisionData[k].vCollisionNormal = n;
4 collisionData[k].vCollisionPoint = v1[i];
5 collisionData[k].vRelativeVelocity = Vr;
6 collisionData[k].vCollisionTangent =
7 Vetor.crossProduct(Vetor.crossProduct(n,Vr),n);
```

Quadro 38 – Dados necessários para resposta de colisão

3.4.2 Operacionalidade da implementação

Para o desenvolvimento de aplicativos na plataforma J2ME, a SUN (2010) disponibiliza o Java Wireless Toolkit, que deve estar instalado no computador. Para executar uma aplicação de exemplo, deve-se importar o projeto do motor M3GE dentro do ambiente de desenvolvimento Eclipse, e em configurações do projeto, o dispositivo que será emulado deve estar configurado para CLDC, como mostra a Figura 18.

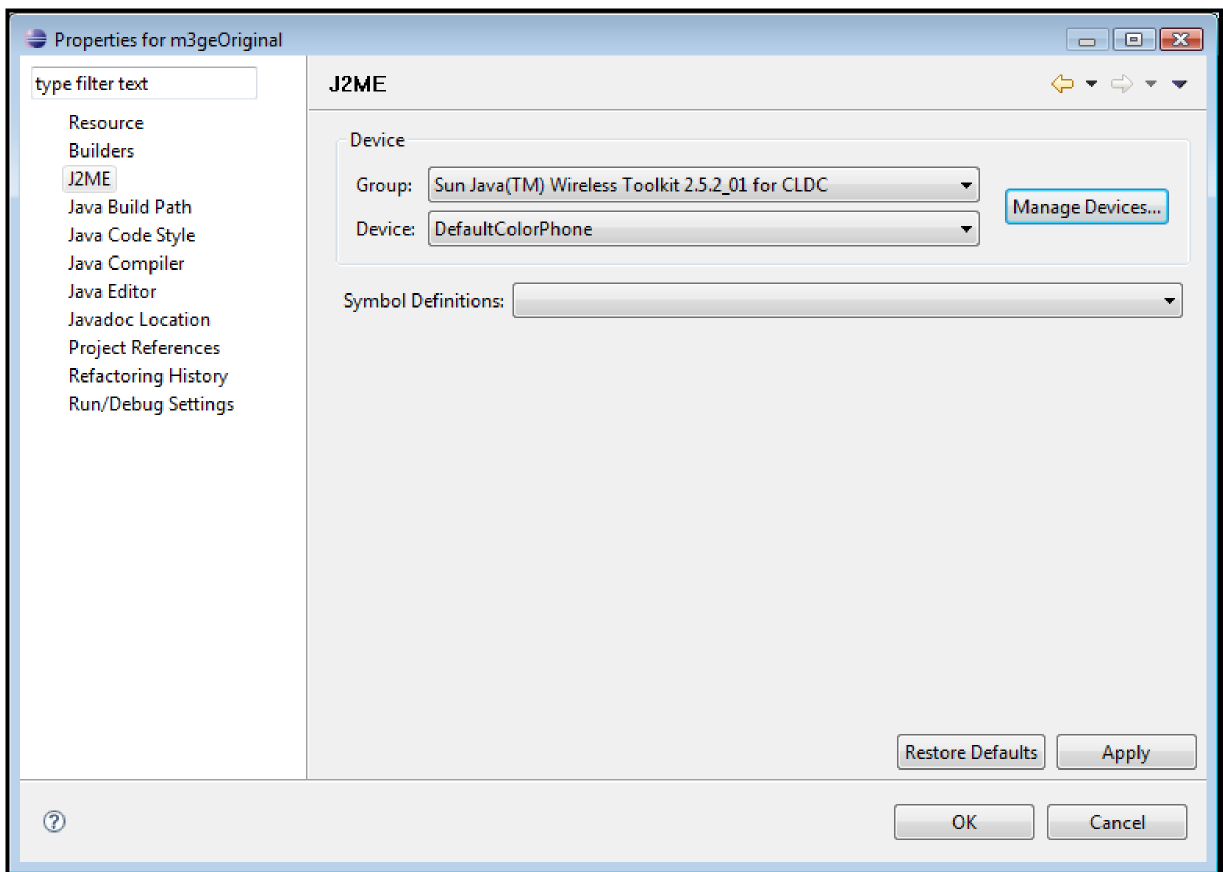


Figura 18 – Configuração do dispositivo a ser emulado

Na raiz do projeto existe uma subpasta chamada `examples`, dentro da pasta do exemplo `Multiplayer` existe a classe nomeado como `MultiplayerMidlelt`. As classes nomeadas com `Midlet` são responsáveis pela execução e lógica dos jogos e devem fazer a comunicação entre motor e jogo. Clicando com o botão direito na Classe `MultiplayerMidlet`, no menu de contexto, dentro da opção *Run As*, deve aparecer a opção *Emulated j2ME Midlet*, como ilustrado na Figura 19.

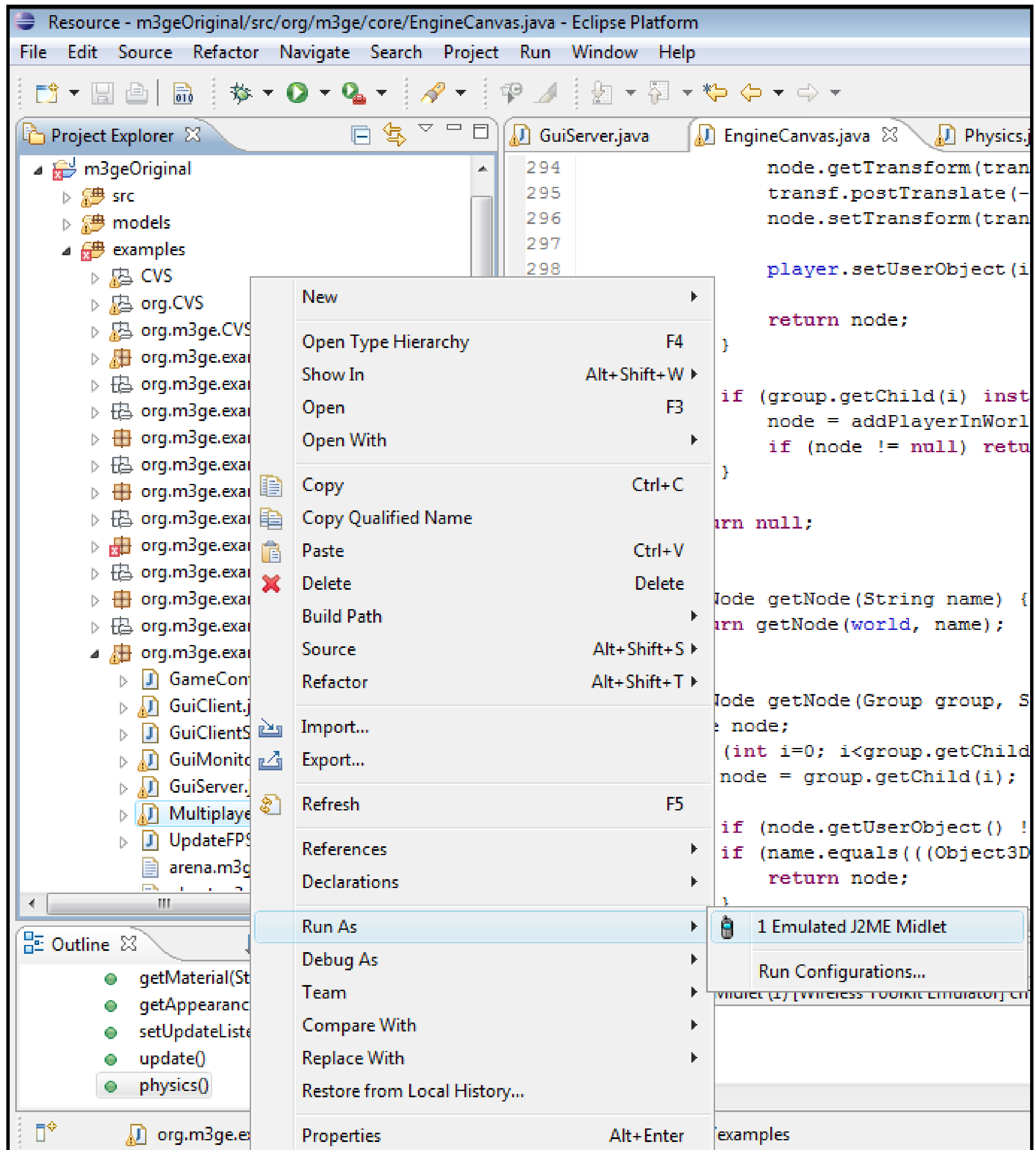


Figura 19 – Como executar o jogo de exemplo

Para executar a simulação de exemplo não é necessário fazer nenhuma configuração adicional, na tela que irá aparecer pode-se selecionar a opção *Server* e em seguida *Start Game*, como mostrado Figura 20.

Figura 20 – Opções para carregar o cenário de demonstração

Após a tela ser carregada, os objetos que estiverem em contato com o chão permanecem no lugar, enquanto os que estiverem suspensos começam a cair devido a força da gravidade, como demonstra a Figura 21. Se houver colisão entre eles ela também será tratada, como mostrado na Figura 22.

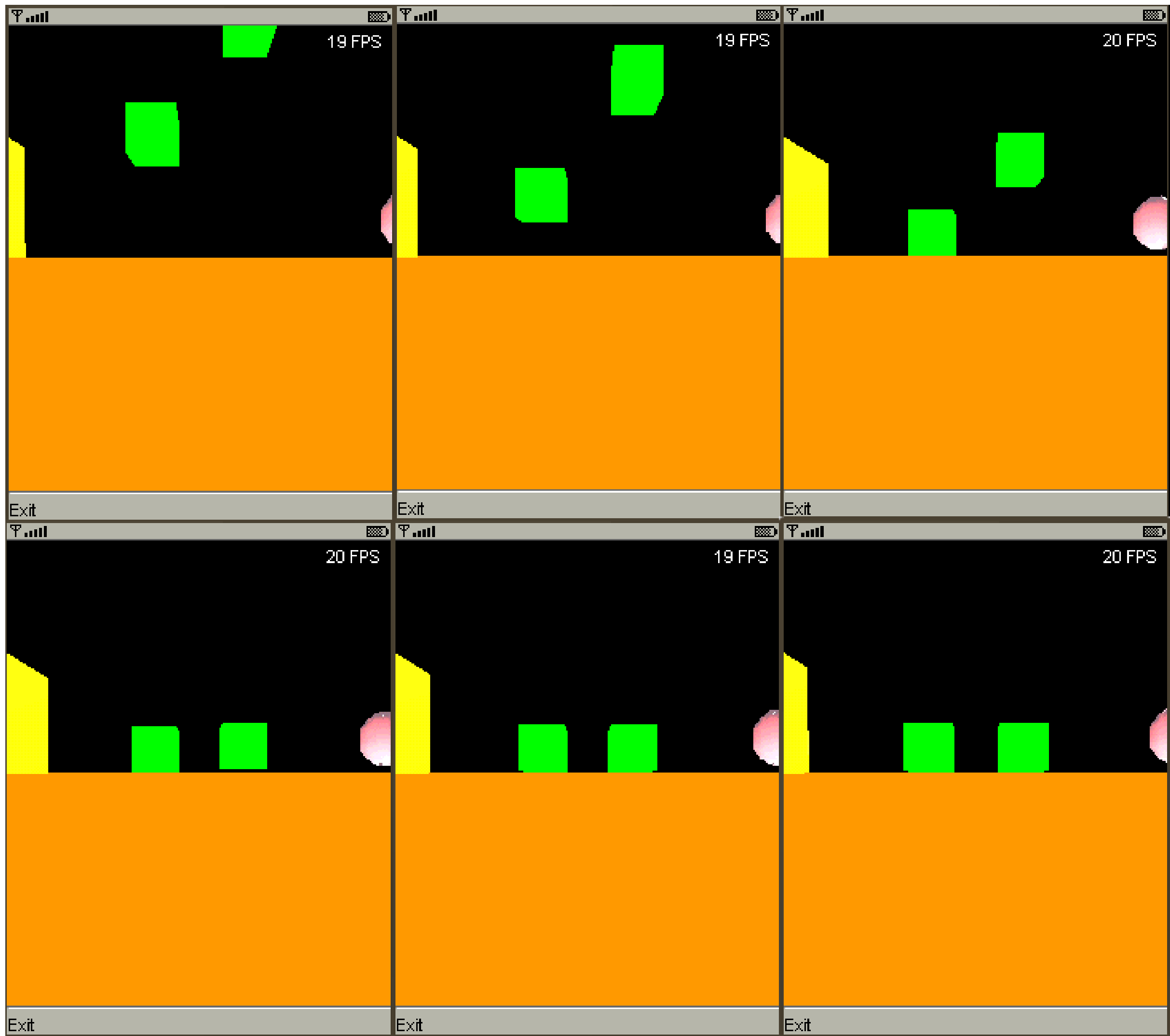


Figura 21 – Objetos em queda e colisão com o solo e permanecendo em contato

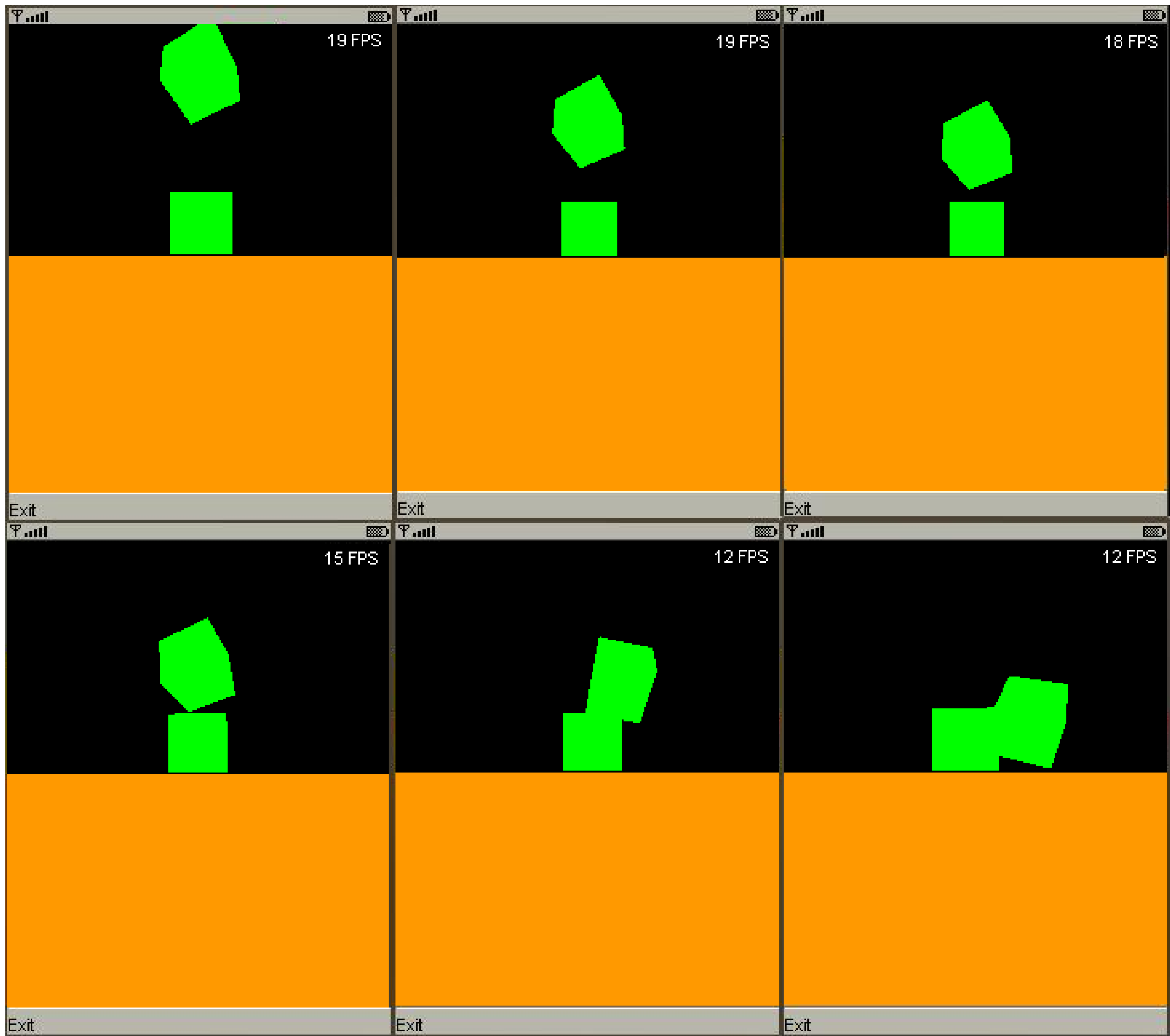


Figura 22 – Colisão entre dois cubos

A interface do teclado ainda permite, através de funções anteriormente implementadas por PAMPLONA (2005), que a posição da câmera seja alterada livremente, como demonstrado na Figura 23. As setas direcionais permitem que a câmera movimente-se para frente, para trás e gire para esquerda e direita. As teclas numéricas 1 e 4, permitem que se olhe para cima e para baixo. Enquanto as teclas 2 e 3 permitem uma movimentação lateral para esquerda e direita.

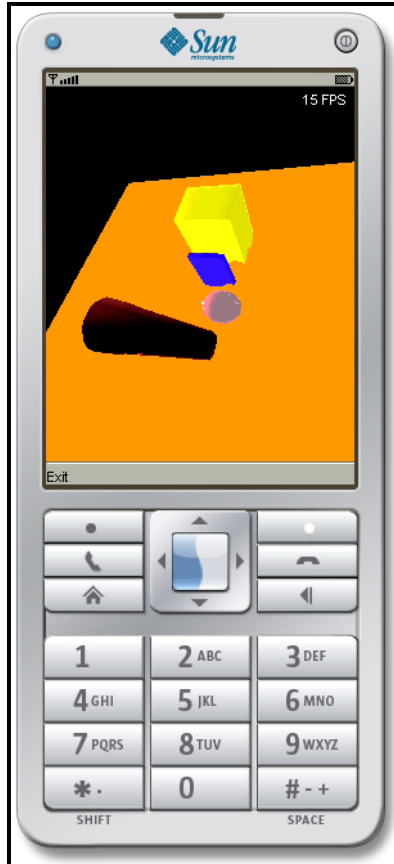


Figura 23 – Mudança de câmera

3.5 RESULTADOS E DISCUSSÃO

No presente trabalho tem-se o desenvolvimento de um módulo capaz de adicionar física de corpos rígidos para o motor de jogos M3GE. As classes presentes neste trabalho foram as desenvolvidas por PAMPLONA (2006), e apenas a classe `EngineCanvas` foi modificada para incluir uma chamada à classe `Physics`. As demais classes presentes no trabalho são auxiliares para operações entre vetores matrizes e quatérnions.

Ao se iniciar o projeto notou-se que a velocidade de processamento era um dos fatores críticos a ser tratado. Nos testes foram utilizados o emulador incluso no Wireless toolkit for CLDC disponibilizado pela SUN (2010), foi mantida a atualização de 20 (FPS) presente na implementação original, e o intervalo de tempo usado para atualização pelo motor de física foi de 15 milissegundos, um valor que permitiu permitiu que não ocorressem penetrações dos corpos simulados e ao mesmo tempo deixou a simulação mais estável.

Para a realização dos testes foram tomados dois cenários, que foram nomeados:

a) face-chão: onde são testados vários objetos colidindo com o chão;
b) face-vértice: existe contato entre o plano de uma das faces do cubo com a aresta de outro cubo.

A simulação realizada pode ainda ser dividida em duas etapas, que podem ser analisadas independentemente, que são:

a) detecção de colisão: detecta se os corpos testados estão em contato um com o outro e calcula os dados necessários para a Resposta;
b) resposta de colisão: toma os dados extraídos pela detecção e calcula uma nova trajetória para os corpos testados.

Definidos os cenários de teste, as duas etapas foram analisadas em relação ao uso de memória teórica e desempenho prático, resultando nos seguintes testes:

a) teste 1: cenário face-chão, etapa de detecção de colisão; desempenho prático;
b) teste 2: cenário face-chão; etapa de resposta de colisão; desempenho prático;
c) teste 3: cenário face-vértice; etapa de detecção de colisão; desempenho prático;
d) teste 4: cenário face-vértice; etapa de resposta de colisão; desempenho prático;
e) teste 5: cenário face-chão; etapa de detecção de colisão; uso de memória teórica;
f) teste 6: cenário face-chão; etapa de resposta de colisão; uso de memória teórica;
g) teste 7: cenário face-vértice; etapa de detecção de colisão; uso de memória teórica;
h) teste 8: cenário face-vértice; etapa de resposta de colisão; uso de memória teórica.

Como objeto de teste foi modelado um cubo com 8 (oito) vértices, interligados por uma malha de 16 (dezesseis) triângulos. Observa-se que para os testes de desempenho cada teste foi repedido 5 vezes e o tempo apresentou uma variação de 15%, sendo que o valor apresentado é uma média dos tempos testados. E para os testes de consumo de memória, os valores foram calculados em byte, convertidos para Kbytes e o valor foi considerado apenas duas casas decimais.

No Teste 1, aumenta-se gradativamente a quantidades de cubos que colidiam com o chão, mas distantes o suficiente para colidirem entre si. O valores de tempo médio obtidos aparecem no Quadro 39, e o aumento do tempo de processamento se mostrou linear, como mostra a Figura 24.

Cubos	01	02	03	04	05	06	07	08	09	10
Tempo (ms)	0*	2	7	9	13	17	20	22	25	28
*tempo foi menor que 1ms										

Quadro 39 – Teste 1 - tempos obtidos

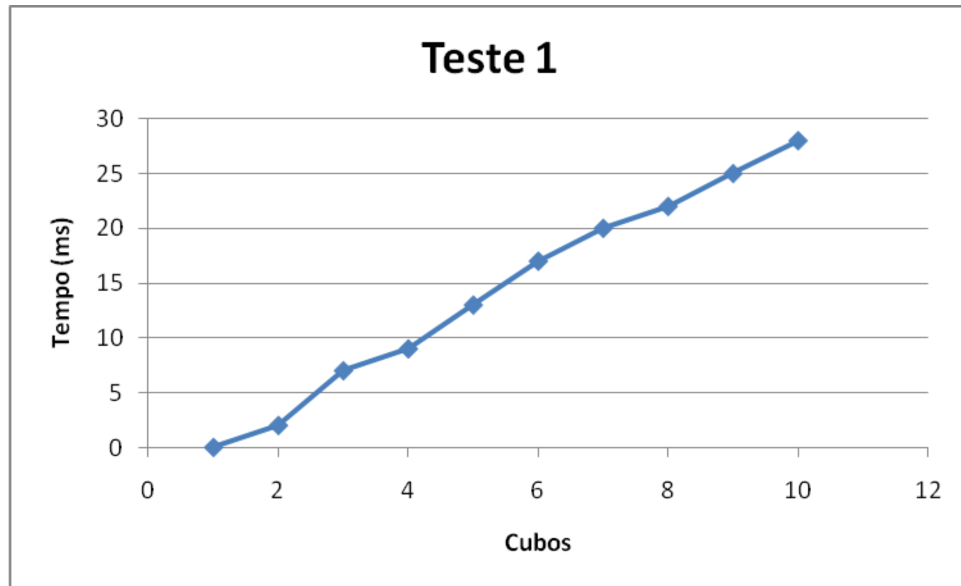


Figura 24 – Teste 1 - relação do tempo com a quantidade de cubos

O Teste 2, mediu o tempo para processar a resposta quando varia-se a quantidade de cubos que colidem com o chão. Os valores de tempo obtidos do Quadro 40 e mostrados na Figura 25, indicam que é um processo rápido e varia pouco quando aumenta-se a quantidade de cubos, obtendo-se tempos de processamento (em milissegundos) iguais mesmo quando a quantidade de cubos colidindo é aumentada.

Cubos	01	02	03	04	05	06	07	08	09	10
Tempo (ms)	01	01	01	01	01	02	02	02	02	03

Quadro 40 – Teste 2 - tempos obtidos

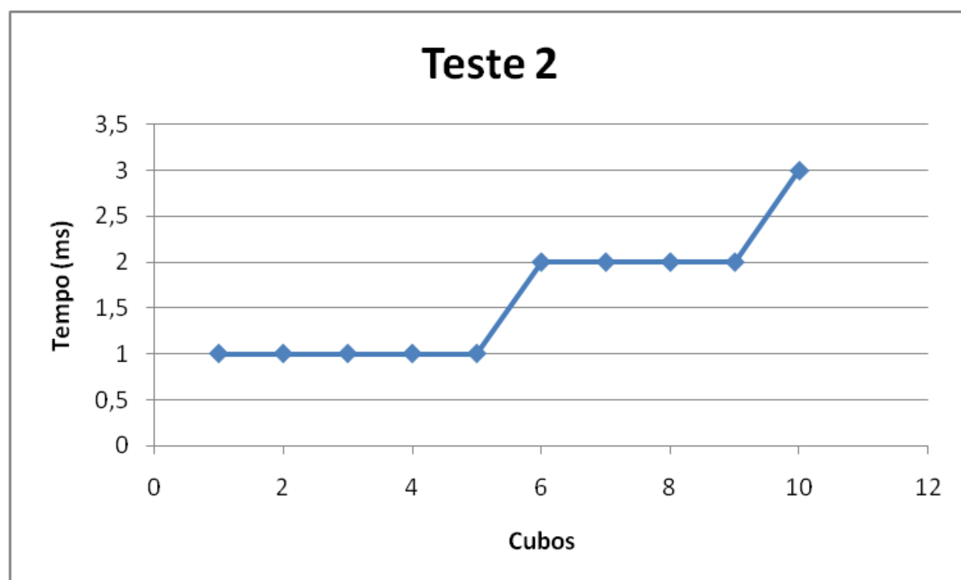


Figura 25 – Teste 2 - relação do tempo com a quantidade de cubos

No Teste 3, foi medido o tempo para processar a detecção de colisão quando aumentava-se a quantidade de cubos empilhado de forma que colidissem entre si. Os tempos de processamento obtidos no Quadro 40 e mostrados na Figura 25, indicam uma inflação na medida em que a quantidade de cubos aumenta.

Cubos	1	2	3	4	5	6	7	8	9	10
Tempo (ms)	1	3	6	10	12	20	38	48	70	82

Quadro 41 – Teste 3 - tempos obtidos

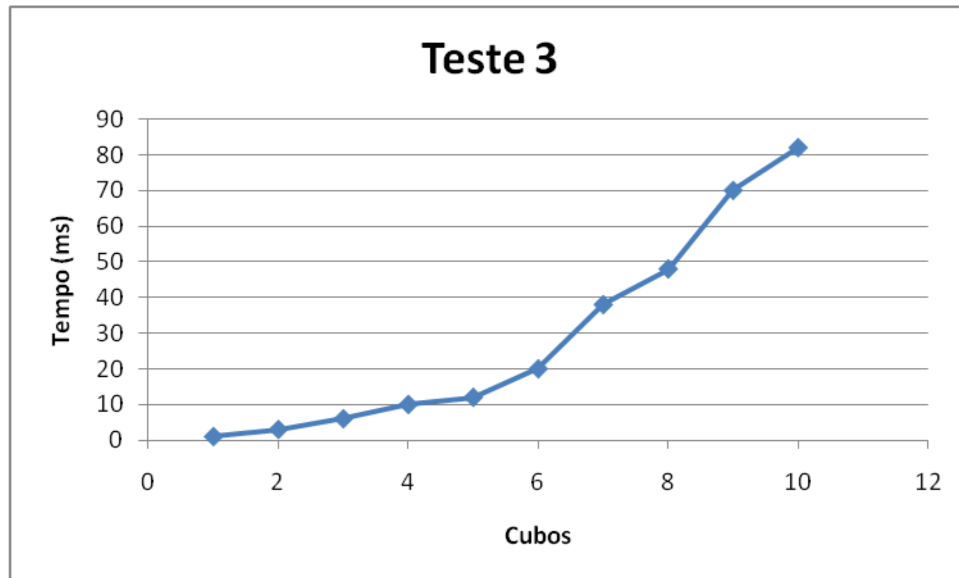


Figura 26 – Teste 3 - relação do tempo com a quantidade de cubos

O Teste 4, medindo o tempo para se processar a resposta de colisão quando os cubos estão empilhados, mostrou-se linear como indicam os valores obtidos no Quadro 42 e na Figura 27.

Cubos	01	02	03	04	05	06	07	08	09	10
Tempo (ms)	1	3	5	7	9	11	13	15	17	19

Quadro 42 – Teste 4 - tempos obtidos

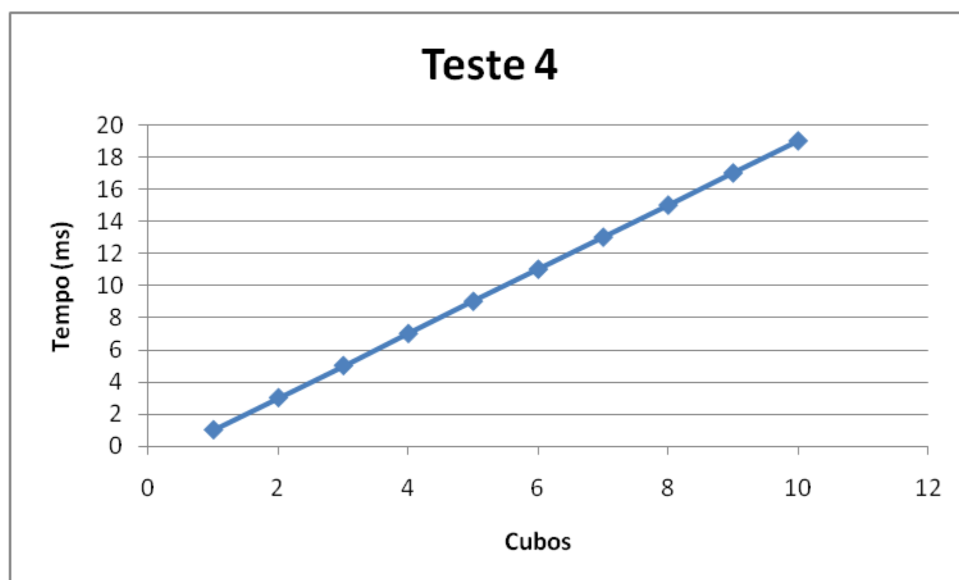


Figura 27 – Teste 4 - relação do tempo com a quantidade de cubos

Relacionando todos os testes de desempenho, na Figura 28, pode-se notar que o teste de maior custo de processamento foi o Teste 3, onde é executado o algoritmo que testa as colisões entre vértices e faces.

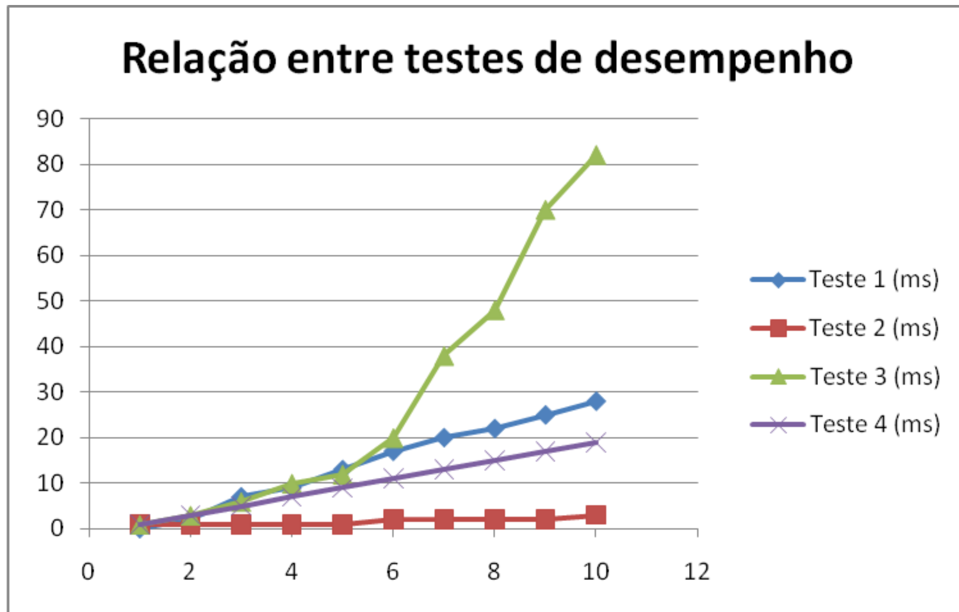


Figura 28 – Relação entre os testes de desempenho

Para o Teste 5, que mediu o consumo de memória quando os cubos colidem entre si, chegou-se à formula apresentada na Figura 29. Os valores calculados no Quadro 43 e na Figura 30, indicam um consumo linear.

Consumo de memória = (1.012 * cubos) + 36

Figura 29 – Teste 5 - fórmula pra o consumo de memória

Cubos	1	2	3	4	5	6	7	8	9	10
Memória (bytes)	1,02	2,01	3	3,98	4,97	5,96	6,95	7,94	8,92	9,91

Quadro 43 – Teste 5 - consumos obtidos

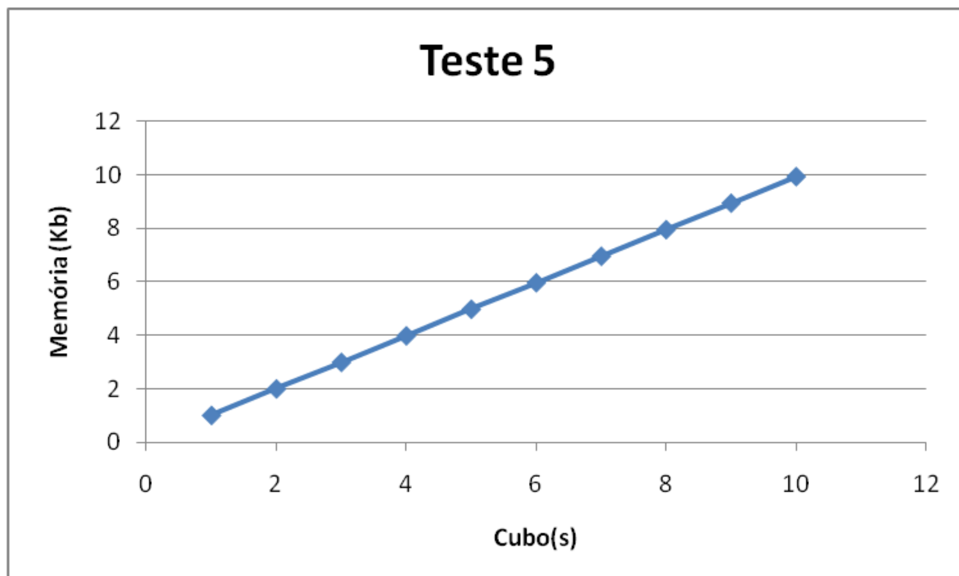


Figura 30 – Teste 5 - relação do uso de memória com a quantidade de cubos

No Teste 6, que mediu o consumo de memória da resposta á colisão dos cubos com o chão, chegou-se à formula apresentada na Figura 31. Os valores calculados no Quadro 44 e presentes na Figura 30, mostram que o uso da memória mantêm-se linear a medida que a quantidade de cubos é aumentada.

Consumo de memória = 404 * cubos

Figura 31 – Teste 6 - fórmula pra o consumo de memória

Cubos	1	2	3	4	5	6	7	8	9	10
Memória (Kb)	0,39	0,78	1,18	1,57	1,97	2,36	2,76	3,15	3,55	3,94

Quadro 44 – Teste 6 - consumos obtidos

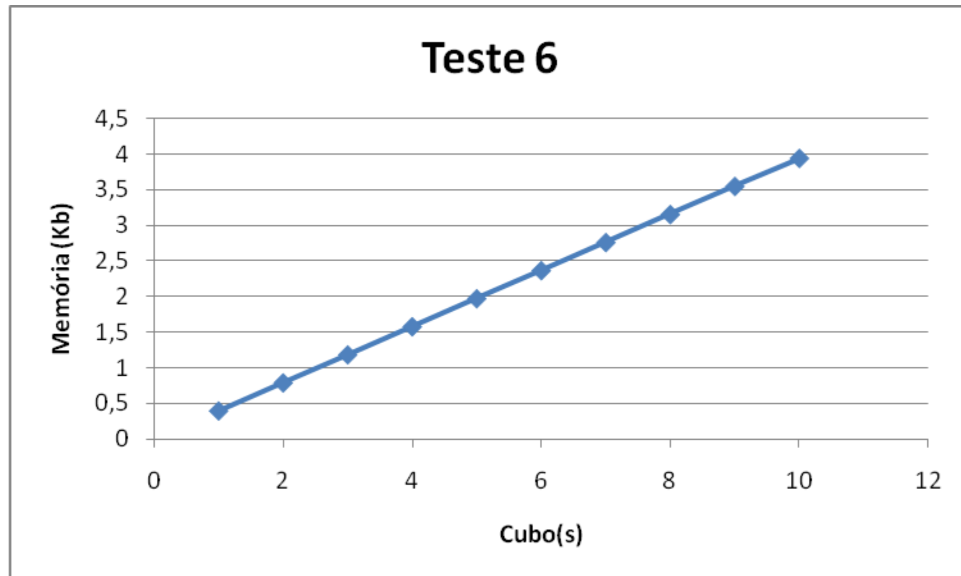


Figura 32 – Teste 6 - relação do uso de memória com a quantidade de cubos

Para o Teste 7, que mediu o consumo de memória quando os cubos colidem com o chão, chegou-se à formula apresentada na Figura 33. Os valores calculados no Quadro 45 e presentes na Figura 30, mostram uma inflação no uso da memória a medida que aumenta-se a quantidade de cubos.

Consumo de memória = {[(cubos * (16396 * cubos))-16396]} + (1012 * cubos) + 36

Figura 33 – Teste 7 - Fórmula pra o consumo de memória

Cubos	1	2	3	4	5	6	7	8	9	10
Memória (Kb)	1,02	50,04	131,09	244,16	389,25	566,37	775,51	1.016,68	1.289,86	1.595,07

Quadro 45 – Teste 7 - consumos obtidos

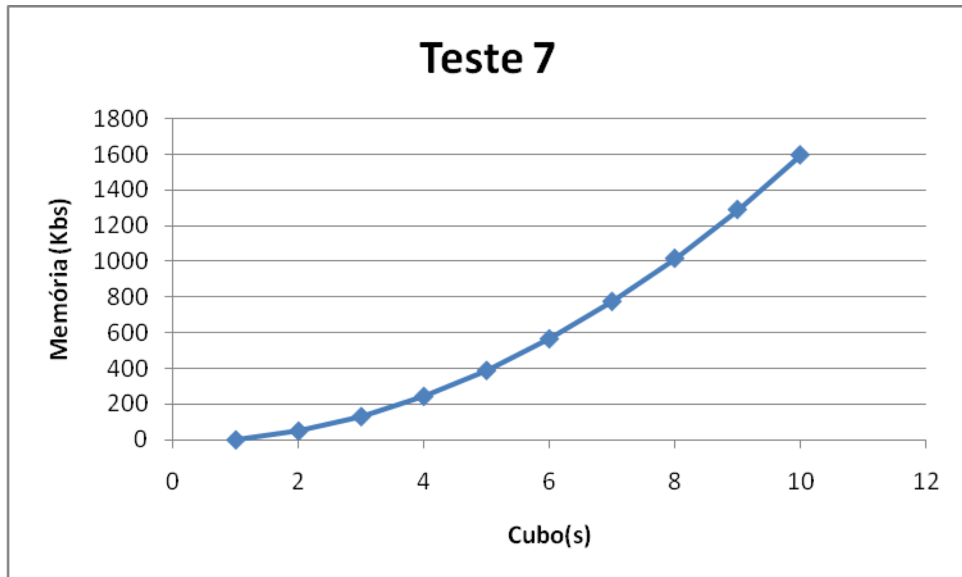


Figura 34 – Teste 7 - relação do uso de memória com a quantidade de cubos

No Teste 8, que mediu o consumo de memória para a resposta à colisão dos cubos entre si, chegou-se à formula apresentada na Figura 35, que é a mesma obtida no Teste 6. Apesar dos cenários e tipo de colisão testados serem diferentes o numero de colisões simultâneas detectadas foi igual aos do Teste 6, não alterando o uso de memória, como mostrado no Quadro 46 e Figura 36.

Consumo de memória = 404 * cubos

Figura 35 – Teste 8 - Fórmula pra o consumo de memória

Cubos	1	2	3	4	5	6	7	8	9	10
Memória (Kb)	0,39	0,78	1,18	1,57	1,97	2,36	2,76	3,15	3,55	3,94

Quadro 46 – Teste 8 - consumos obtidos

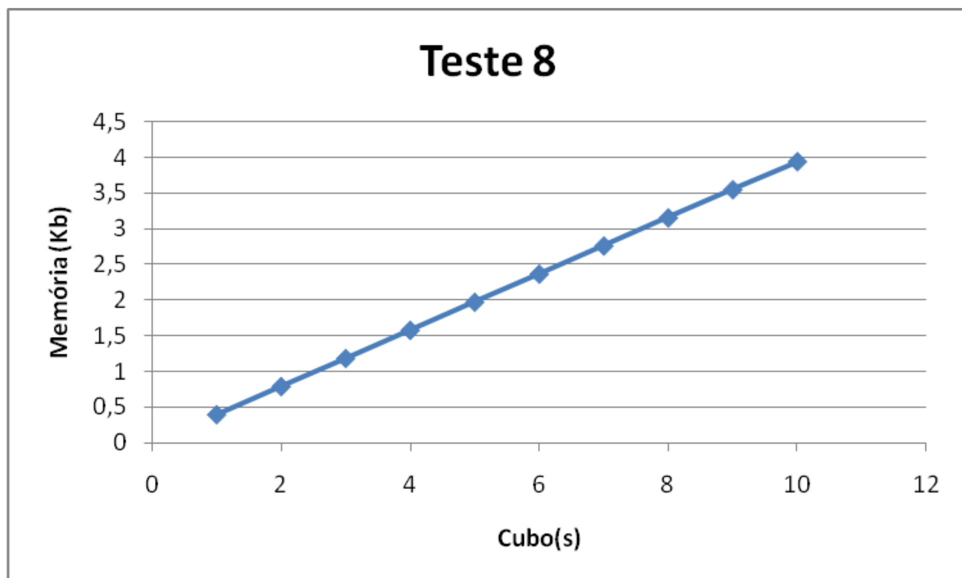


Figura 36 – Teste 8 - relação do uso de memória com a quantidade de cubos

Ao relacionar-se todos os testes de memória, na Figura 37, nota-se que o teste que obteve maior uso de memória foi o Teste 7, onde é executado o teste de colisão entre os

vértices e faces.

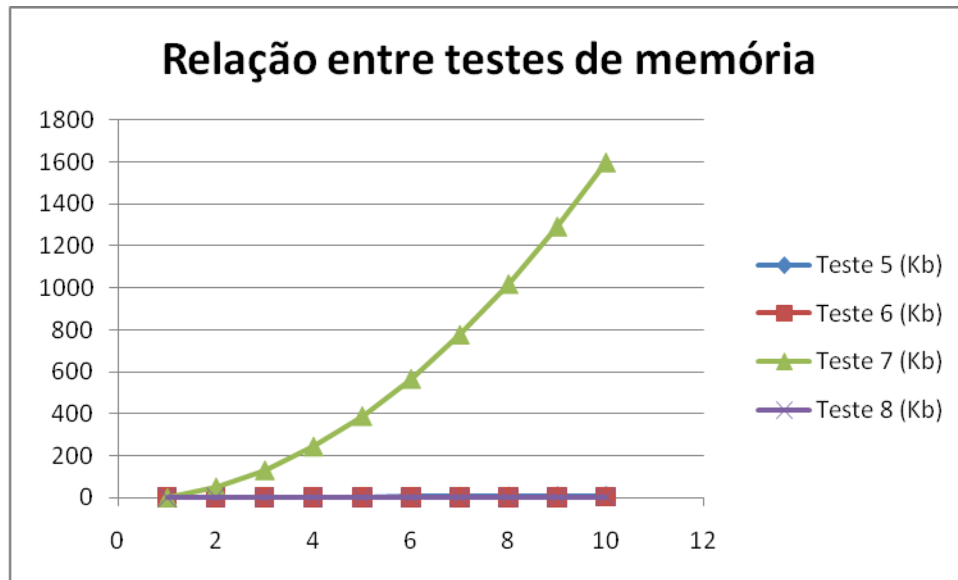


Figura 37 – Relação entre os testes de memória

Uma das dificuldades encontradas tanto durante os testes quanto no desenvolvimento, foi que o *debug* para do ambiente utilizado para a plataforma ainda não funciona perfeitamente para a plataforma utilizada, algumas soluções foram pesquisadas em fóruns na internet com usuários que tiveram o mesmo problema, mas nenhuma solução se mostrou efetiva, optando-se por utilizar o modo console do próprio ambiente para verificar o valor das variáveis quando necessário.

A plataforma J2ME possui ainda uma biblioteca matemática limitada e não estão disponíveis as funções trigonométricas `acos()`, `asin()`, `atan()` e `atan2()`, que foram necessárias para transformar os ângulos no formato de quatérnions para ângulos de Euler, usados pelo método da `postRotate`, da M3G, que rotaciona os objetos na tela. A solução encontrada foi usar uma implementação dessas funções encontrada no fórum da NOKIA (2006).

A natureza do trabalho, que passa pelo campo da física, trouxe consigo o grau de complexidade alta para os cálculos matemáticos, físicos e gráficos para tratamento de colisão. Para isso foram utilizados os livros Physics for Game Developes, de BOURG (2002) e Game Pysics Engine Development, de MILLINGTON (2007).

4 CONCLUSÕES

O presente trabalho descreveu o desenvolvimentor um módulo capaz adicionar dinâmica de corpos rígidos ao motor M3GE. A implementação do módulo teve com base o método apresentado por BOUG (2002) e o código fonte dos exemplos implementados por ele na linguagem C++, disponíveis online O'REILLY (2001), foram cruciais, poupando tempo e tornando o projeto viável.

O uso de dinâmica de corpos rígidos em 3D se mostrou viável, e mostra-se um estudo relevante levando em conta a ausência de motores de física em 3D que tenham sido desenvolvidos para a plataforma J2ME, bem como falta de livros específicos voltados para este assunto.

apesar das limitações inerentes da plataforma J2ME, sendo que nem todos objetos do cenário precisam ser gerenciados pelo módulo de física e poderiam ser ativados ou desativados dependendo de certas condições, como eventos ou a distância que se encontram do jogador. Mas considerando que desempenho do teste de colisão entre vértices e planos consumiu significativo tempo de processamento, pode-se inferir que outros testes com um maior número de comparações ou que utilizan-se de formas mais complexas ocasionem numa perda significativa de FPS, portanto o desempenho dos algoritmos de detecção deve ser observado.

4.1 EXTENSÕES

A campo de estudo dos motores de física é bastante abrangente, possibilitando que ao módulo apresentado muitas alternativas de continuação tanto na área da detecção de colisão, quanto da resposta.

De forma a melhorar a detecção, a extensão mais próxima seria um método que detecte também o contato entre os pares de arestas, mas outras formas para detecção poderiam ser utilizadas, como esferas, cones, formas côncavas e convexas, que podem ser testadas entre si ou usadas para compor objetos com formas mais complexas.

Outros efeitos comumente presentes em motores de física ainda podem ser adicionados como física para corpos macios, corpos deformáveis, *ragdolls* (simulação dos movimentos de

um corpo humano) e sistema de partículas para simular efeitos como névoa, fogo e explosões.

Seria importante o estudo de técnicas para otimização do código de modo a aumentar os FPS, que pode ser com alguma técnica que diminua o número de comparações para verificar a colisão e/ou métodos mais eficientes. Pode-se também verificar modos de simplificar as equações de forma a usarem operações que demandem menor tempo de processamento. Além da possibilidade de usar números inteiros ao invés de números de ponto flutuante.

REFERÊNCIAS BIBLIOGRÁFICAS

ADENSAMER, Alexander. **Emini physics engine**. Vienna, 2010. Disponível em: <http://emini.at>. Acesso em: 20 mar. 2010.

BATTAIOLA, André L. et al. Desenvolvimento de jogos em computadores e celulares. **Revista de Informática Teórica e Aplicada**, [S.l.], v. 8, n. 2, p. 7-46, out. 2001.

BERBERICH, Steve; WRITER, Staff. **Video games starting to get serious**. Gaithersburg, 2007. Disponível em: <http://www.gazette.net/stories/083107/businew11739_32356.shtml>. Acesso em: 9 mar. 2009.

BOURG, David M. **Physics for game developers**. Sebastopol: O'Relly, 2002.

BULLET PHYSICS. **Bullet collision detection and physics library**. [S.l.], 2005. Disponível em: <http://www.bulletphysics.com/mediawiki-1.5.8/index.php?title=Features>. Acesso em: 23 mar. 2009.

ELDAWY, Mohamed. **Report on use of middleware in games**. Madison, 2006. Disponível em: <http://adlcommunity.net/file.php/36/GrooveFiles/Games%20Madison/report%20Middleware.pdf>. Acesso em: 7 mar. 2009.

GOMES, Paulo C. R.; PAMPLONA, Vitor F. **M3GE:** um motor de jogos 3D para dispositivos móveis com suporte a Mobile 3D Graphics API. São Paulo, 2005. Disponível em: <http://vitorpamplona.com/deps/papers/2005_SBGAMES_M3GE.pdf>. Acesso em: 1 mar. 2009.

MACNY, Ondraj. **DyMiX engine**. Pragua, 2010. Disponível em: <http://dymix.hardwire.cz>. Acesso em: 20 mar. 2010.

MILLINGTON, Ian. **Game physics engine development**. Amsterdam: Morgan Kaufmann, 2007.

NEWTON GAME DYNAMICS. [S.l.], 2007. Disponível em: <http://www.newtondynamics.com >. Acesso em: 23 mar. 2009.

NOKIA . **Forum Nokia**. [S.l.], 2006. Disponível em: <http://discussion.forum.nokia.com/forum/showthread.php?72840-Maths-acos-asin-atan>. Acesso em: 20 mar. 2010.

O'REILLY . **Index to example programs discussed in physics for game developers**. [S.l.], 2001. Disponível em: <http://examples.oreilly.com/9780596000066>. Acesso em: 20 mar. 2010.

PAMPLONA, Vitor F. **Um protótipo de motor de jogos 3d para dispositivos móveis com suporte a especificação mobile 3d graphics api for J2me**. 2005. 83 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

SUN MICROSYSTEMS. **Java ME platform overview**. [S.l.], 2009a. Disponível em: <http://java.sun.com/j2me/overview.html>. Acesso em: 16 mar. 2009.

______. **JSR 184**: Mobile 3D Graphics API for J2ME. [S.l.], 2009b. Disponível em: <http://jcp.org/en/jsr/detail?id=184>. Acesso em: 14 mar. 2009.

______. **Sun java wireless toolkit for CLDC**. [S.l.], 2010. Disponível em: <http://java.sun.com/products/sjwtoolkit/download.html>. Acesso em: 10 mar. 2010.

UNREAL TECHNOLOGY. [S.l.], 2008. Disponível em: <http://www.unrealtechnology.com/>. Acesso em: 16 mar. 2009.

WARD, Jeff. **What is a game engine?** [S.l.], 2008. Disponível em: <http://www.gamecareerguide.com/features/529/what_is_a_game_.php>. Acesso em: 16 mar. 2009.