UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

UM PROTÓTIPO DE MOTOR DE JOGOS 3D PARA DISPOSITIVOS MÓVEIS COM SUPORTE A ESPECIFICAÇÃO MOBILE 3D GRAPHICS API FOR J2ME

VITOR FERNANDO PAMPLONA

BLUMENAU
2005

2005/I-41

VITOR FERNANDO PAMPLONA

UM PROTÓTIPO DE MOTOR DE JOGOS 3D PARA DISPOSITIVOS MÓVEIS COM SUPORTE A ESPECIFICAÇÃO MOBILE 3D GRAPHICS API FOR 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. Paulo C. Rodacki Gomes, Dr. –
Orientador

BLUMENAU
2005

2005/I-41

UM PROTÓTIPO DE MOTOR DE JOGOS 3D PARA DISPOSITIVOS MÓVEIS COM SUPORTE A ESPECIFICAÇÃO MOBILE 3D GRAPHICS API FOR J2ME

Por

VITOR FERNANDO PAMPLONA

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

Presidente: Prof. Paulo C. Rodacki Gomes, Dr. – Orientador, FURB

Membro: Prof. Maurício Capobianco Lopes, Msc. - FURB

Membro: Prof. Mauro Marcelo Mattos, Dr. - FURB

BLUMENAU
2005

AGRADECIMENTOS

À Aquele que não conheço, mas espero conhecer.

Ao meu falecido pai por apontar o caminho certo diversas vezes durante o curso.

À minha mãe e ao meu irmão que sempre me apoiaram.

Ao meu orientador, Dr. Paulo Cesar Rodacki Gomes, por confiar em mim.

Ao meu amigo, Marcelo Eduardo M. de Oliveira, consultor de jogos do Instituto Nokia de Tecnologia pelas consultorias cedidas.

Aprecie o passado, devore o presente e sinta o futuro.

RESUMO

Este trabalho descreve a construção de um motor de jogos 3D para dispositívos móveis ou limitados utilizando a especificação *Mobile 3D Graphics API for J2ME*. O motor permite o desenvolvimento de jogos complexos e de alta qualidade gráfica nos aparelhos de maneira rápida o suficiente para criar interatividade com o jogador. Além do motor de jogos, descreve a implementação um pequeno demonstrativo de suas características para a validação do trabalho e certificação da tecnologia, o qual foi testado em meio físico, e não apenas simulado.

Palavras Chave: Jogos. Computação gráfica. Motor de jogos. Java 2 Micro Edition. JSR-184. M3G.

ABSTRACT

This work describes the construction of a 3D game engine to mobile or limited devices using the Mobile 3D Graphics API for J2ME specification. The engine allows the development of high quality graphics and complex games to mobile devices in a way to gain velocity. A small demonstration is described to test the game engine characteristics and functions, and it will validate the java tecnology for mobile devices. The tests were on simulated and real devices.

Key-Words: Games. Graphic computer. Game engine. Java 2 Micro Edition. JSR-184. M3G.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

Há muitos anos o homem cria jogos para se divertir. O jogo sempre foi sinônimo de competição, avaliando quem é o mais forte ou o mais rápido. A era tecnológica os evoluiu, criando ambientes complexos tanto para reprodução visual quanto para resolução do enredo. Atualmente, um novo setor está chamando a atenção, um segmento que está se desenvolvendo com muita rapidez, que tende a ultrapassar o volume de produção dos jogos atuais. Esta é a área de jogos para dispositivos móveis. Um mercado muito rico, grande e diversificado (BATTAIOLA et al., 2001) .

Os jogos para celulares disponíveis ao mercado podem ser comparados com os jogos existentes no final dos anos 80. Jogos simples, em duas dimensões e sem utilizar muitos recursos gráficos. Estimativas indicam que, em alguns anos, os tradicionais jogos eletrônicos executados em micro-computadores estarão rodando em celular, *palm top*, *pocket pc* e tantos outros modelos (AARNIO, 2004). Esta evolução criará novos mercados e, um deles, é o desenvolvimento de arquiteturas e ferramentas para facilitar o desenvolvimento desses jogos.

Dentre as novas necessidades, um requisito peculiar e muito conhecido pelos engenheiros de software, principalmente de jogos, se sobressai: os *motores de jogos*. Esses motores são bibliotecas de desenvolvimento responsáveis pelo gerenciamento do jogo, das imagens, do processamento de entrada de dados e outras funções. São tão importantes que estão em, praticamente, todos os jogos para micro-computadores, controlando a estrutura do jogo e seu ciclo de vida.

Para encontrar os desafios deste novo mundo, este trabalho descreve a construção de um motor de jogos 3D de forma que desenvolvedores e projetistas de jogos, também chamados de *game designers*, tenham uma infra-estrutura de software básica para a criação dos jogos para dispositivos móveis.

1.1 ESCOPO E PROBLEMA

As tradicionais rotinas gráficas de renderização, textura e tratamento de imagens exigem um grande poder de processamento e um alto uso de memória, o que tornaria jogos e aplicações gráficas inviáveis para dispositivos móveis. Esse, sem dúvida, foi um dos maiores desafios na implementação deste trabalho: a utilização de algorítmos exigentes implementando-os em uma linguagem interpretada e com limitados recursos de hardware.

Atualmente existem poucos motores capazes de atuar com velocidade e precisão em dispositivos móveis. A diferença entre a maioria dos motores de jogos e este trabalho está em uma recente padronização que será comentada na fundamentação teórica - o uso da biblioteca de funções gráficas nativas para a plataforma Java 2 Micro Edition (SUN MICROSYSTEMS, 2004a) chamada de *Mobile 3D Graphics API* (M3G) (NOKIA, 2003).

O segmento de jogos para celulares ou dispositivos limitados está em sua fase inicial levando este trabalho a ser considerado pioneiro, além da possibilidade de abrir várias portas para o mundo da computação gráfica em dispositivos móveis.

1.2 OBJETIVO

Este trabalho possui três objetivos gerais: adquirir experiência prática com desenvolvimento de jogos, analisar o uso do java para desenvolvimento em dispositivos móveis e verificar como os dispositivos móveis se comportam com jogos complexos e rotinas 3D.

Para alcançar esses objetivos, é proposto um protótipo de um motor de jogos 3D para dispositivos móveis chamado de *Mobile 3D Game Engine* (M3GE). Este protótipo deve implementar um algoritmo leitor de um arquivo de modelo 3D, carregar todos os dados definidos nele e montar uma visualização do cenário para o jogador. Deve possuir formas predefinidas e configuráveis de movimentação de personagem, de atualização da visão em um espaço de tempo e de câmeras para apresentação do mundo.

O trabalho também apresenta a implementação de um jogo simples utilizando as propriedades criadas no motor de jogos e, o que permitiu avaliar a implementação, velocidade e a portabilidade do motor de jogos construído.

Este trabalho foi testado em um celular Siemens modelo CX65 (SIEMENS AG, 2005a), um dos primeiros celulares a implementarem a especificação M3G no Brasil.

1.3 ESTRUTURA DO TEXTO

Esta monografia divide-se em fundamentação teórica, desenvolvimento e conclusões.

No capítulo de fundamentação teórica são mostradas todas as informações necessárias para esclarescer o leitor e dar o início da análise e ao desenvolvimento. No capítulo 3 é apresentada a metodologia de desenvolvimento e os documentos de análise, assim como os algoritmos criados e adaptados. Por fim, a conclusão retrata a opinião do autor sobre o tema e sua implementação, considerando ou não o java como ambiente de desenvolvimento e execução para softwares que utilizam rotinas 3D.

2 FUNDAMENTAÇÃO TEÓRICA

Dividiu-se este tópico em oito seções para facilitar o entendimento do assunto. Primeiramente apresenta-se um estado da arte e comenta-se a história dos jogos em dispositivos móveis. Logo após, define-se cada uma das tecnologias a serem utilizadas no trabalho e os trabalhos correlatos a este.

2.1 UM POUCO DE HISTÓRIA

Mobilidade, esta é a palavra do futuro. O que era gigantesco já ficou minúsculo, o que é minúsculo será microscópico daqui a alguns anos. Assim como aconteceu com outros equipamentos, os celulares e os *Personal Digital Assistant* (PDA) estão conquistando espaço no mercado. A tecnologia tornou-se essencial, assumindo o posto de braço direito dos homens.

Há poucos anos os celulares e similares ganharam uma tela mais delicada, com uma maior resolução e coloridas, ou seja, preparada para os usuários exigentes que viriam. Junto com ela, surgiam os sistemas operacionais e as linguagens de programação para dispositivos móveis. Assim uma nova era para a informática. Inicialmente, os recursos escassos não permitiam grandes feitos. A conexão, via cabos, com micro-computadores era ruim, não se tinha a mobilidade desejada. Nos últimos anos essa tecnologia evoluiu, rompeu as barreiras de hardware e software e, finalmente, permitiu as pequenas e grandes empresas desenvolverem seus projetos.

Juntamente com os celulares modernos, surgiram os jogos para dispositivos móveis. Leves, sem muitos detalhes e absolutamente fracos, os jogos para celulares, hoje, podem ser comparados com os jogos *desktop* lançados no final dos anos 80. Em 2000, já se previa que a história dos jogos se repetiria para os celulares. Da mesma maneira que alguns anos atrás, eles irão ganhar espaço, melhorar seu *design* e evoluir para ambientes 3D e *on-line*.

Paralelamente, as linguagens de programação evoluíam. Java, a linguagem interpretada criada em 1995 pela Sun Microsystems (SUN MICROSYSTEMS, 2005), dava seus primeiros

passos. Em 1998, com a criação do *Java Community Process* (JCP) (JCP, 2004a) a tecnologia Java deixa de ser propriedade da Sun e passa a ser propriedade de um grupo de especificação, do qual qualquer empresa poderia pertencer.

O JCP criou as *Java Specification Request* (JSR) (JCP, 2004b), especificações claras, concisas e livres, que determinam os padrões de desenvolvimento, novas implementações ou revisões de uma implementação existente no Java. Estes documentos permitem a outras empresas participarem ativamente do desenvolvimento da tecnologia Java, aumentando o foco tecnológico e abrindo espaço para que a tecnologia possa ser usada em todos os lugares.

A união entre dispositivos móveis e a tecnologia Java trouxe grandes resultados nas áreas de automação comercial e industrial, surgindo, no mercado, muitos sistemas com interfaces em celulares e PDAs. Porém, no desenvolvimento de jogos, o Java foi descartado por ser muito lento. A adoção para este tipo de desenvolvimento é maior em linguagens nativas dos dispositivos por serem mais rápidas e com mais recursos gráficos. Em 2002, lançou-se uma JSR com o objetivo de criar rotinas gráficas velozes e práticas para substituir as implementações das linguagens nativas, a M3G (NOKIA, 2003). Essa especificação mudou os ânimos da maioria dos investidores, voltando a colocar o Java na linha de jogos para dispositivos móveis.

Battaiola et al. (2001, p. 1) diz que “Jogos por computador têm apresentado um surpreendente grau de evolução tecnológica nos últimos anos. O grande interesse no desenvolvimento deste tipo de software se deve ao seu atrativo comercial, cujo mercado mundial movimenta dezenas de bilhões de dólares.”.

2.2 DISPOSITIVOS MÓVEIS

A computação móvel é caracterizada por um dispositivo móvel, com capacidade de processamento, em um ambiente sem fio. Exemplos de dispositivos móveis são: *Personal Digital Assistants* (PDAs), telefones celulares e smartcards.

Esses dispositivos suportam uma carga de processamento muito fraca, apresentam baixo potencial de memória, são, geralmente, mono-tarefa e com grandes limitações gráficas, tornando o desenvolvimento de aplicações muito mais difícil que em ambientes *desktop*. Mesmo

assim, dispositivos móveis estão começando a ser utilizados para entrada de dados nos mais variados sistemas, aumentando a mobilidade e, em conseqüência, a produtividade de muitas pessoas.

Hoje em dia os mais conhecidos e utilizados dispositivos são os celulares, que evoluíram tanto que não podem mais ser considerados como simples telefones. Os produtos lançados recentemente nos Estados Unidos e na Europa já possuem agenda, e-mail, internet, fotografia, e, é claro, jogos (NOKIA, 2005).

Em uma pesquisa realizada em 30 de junho de 2004, o IBGE (2004) apontou que, só no Brasil, "telefones celulares já são o sétimo produto em volume de vendas, com R$7,5 bilhões em 2002 ". Em contexto global, o número de celulares que rodam Java chega a 350 milhões (SUN MICROSYSTEMS, 2004d), isso desconsiderando vários modelos que não possuem o Java instalado.

A maioria dos aparelhos recentemente lançados possui entre 10 e 60 *megabytes* de memória compartilhada. Ou seja, qualquer aplicação deverá dividir esse volume de memória com o sistema operacional do celular. Em *palm tops* este valor já ultrapassou os 128MB e tende a crescer muito mais.

2.3 JOGOS ELETRÔNICOS

Segundo Battaiola et al. (2001, p. 4), "Um jogo de computador pode ser definido como um sistema composto de três partes básicas: enredo, motor e interface interativa. O sucesso de um jogo está associado à combinação perfeita destes componentes.".

O enredo trata do ritmo do jogo: temas, tramas e objetivos. A interface interativa é o meio que o jogador se comunica com o jogo, e também o meio em que o jogo mostra o seu estado atual. O motor é o ponto central, é ele que controla a reação de um jogo perante uma ação do usuário. Este motor é descrito na seção 2.4.

Atualmente existem vários tipos de jogos, mas basicamente se dividem em dois grandes grupos, baseados na forma com que o usuário interage com o jogo:

a) jogos em terceira pessoa: são jogos em que o usuário vê a cena. Na literatura essa

visão é chamada de God's Eye, ou o olhar de Deus. Geralmente é uma visão de cima, abrangendo toda a cena envolvida. Nesses casos o personagem é denominado de Ator;

b) jogos em primeira pessoa: são jogos em que a visão do jogador é a mesma do personagem em jogo. Nesse tipo de jogo, o usuário não consegue, por exemplo, ver quem se encontra atrás de seu personagem. Nestes casos o personagem é denominado de Avatar.

Conforme Battaiola et al. (2001), define-se cena como o que está no escopo de atuação do jogador de acordo com o tipo de jogo que é representado. Para desenvolver um jogo, além das preocupações com imagens e rotinas gráficas, criar um enredo envolvente não é simples. Para solucionar uma parte dos problemas desenvolvem-se os motores de jogos, que serão tratados mais adiante.

2.4 OS MOTORES DE JOGOS 3D

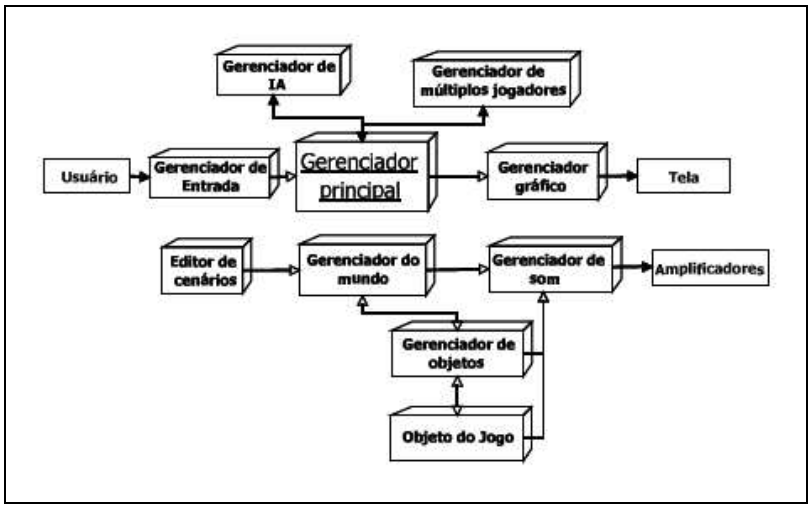
Os jogos para desktop já utilizam recursos muito avançados, chegando ao ponto de separar as regras e a visualização do jogo. De acordo com o padrão de projeto *Model View Control* (MVC) (SUN MICROSYSTEMS, 2004c), modelo, visão e controle devem estar nitidamente separados em qualquer aplicação. Para garantir as boas práticas no desenvolvimento de jogos, foram criados os motores de jogos 3D ou, em inglês, *3D game engines*.

Segundo Eberly (2001, p. xxvii, tradução nossa), "Um motor de jogo deve tratar as questões de gerenciamento do grafo de cena através de uma interface que provê de forma eficiente a entrada da camada exibidora, podendo esta ser baseada em hardware ou software". Uma arquitetura de motor de jogos 3D completa pode ser observada na fig. 2.1.

Os componentes da fig. 2.1 são:

a) gerenciador de entrada: identifica eventos de entrada de dados e envia para o gerenciador principal para definir o que será feito com o dado;

b) gerenciador gráfico: transforma um modelo matemático do estado atual do jogo em uma visualização para o usuário, mostrando somente o que o jogo permite para aquela cena. Aqui se encontram todas as rotinas de gerenciamento gráfico de um



Fonte: Battaiola et al. (2001, p. 12)

Figura 2.1 – Arquitetura de um motor de jogos 3D

jogo. Em dispositivos móveis isso é implementado pela M3G;

c) gerenciador de som: execução de sons a partir de eventos no jogo. No Java a especificação que cuida dessa parte é JSR-135 (SUN MICROSYSTEMS, 2004e): *Mobile Media API*;

d) gerenciador de inteligência artificial: gerencia o comportamento dos objetos desenvolvidos pelo designer. Geralmente é desenvolvido em linguagens lógicas como Prolog;

e) gerenciador de múltiplos jogadores: faz o tratamento de vários jogadores e da comunicação entre eles independentemente do meio físico que se encontram;

f) gerenciador de objetos: carrega, controla o ciclo de vida, salva e destrói um grupo de objetos do jogo. Em geral, um jogo possui vários gerenciadores de objetos que, além de suas funções normais, ainda precisam se comunicar;

g) objeto do jogo: possui dados relevantes para uma entidade que faça parte do jogo, como avião, monstro, parede, etc. Esta parte do motor controla a posição, velocidade, dimensão, detecção de colisão, o próprio desenho, entre outros;

h) gerenciador do mundo: armazena o estado atual do jogo e para isso utiliza os gerenciadores de objetos. Em geral, um editor de cenários descreve um estado inicial do jogo para cada nível do mesmo;

i) editor de cenários: ferramenta externa que cria mundos para serem carregados pelo

gerenciador de mundos;

j) gerenciador principal: indica qual objeto vai processar qual informação.

Levando em conta que o gerenciador gráfico e o gerenciador de som já estão prontos, e que neste trabalho não serão abordados questões de inteligência artificial, múltiplos jogadores e editor de cenários. O que será implementado é um gerenciador de entrada, um gerenciador de objetos, um gerenciador de mundo e o gerenciador principal.

Atualmente os motores de jogos 3D *open source* mais conhecidos são: Crystal Space (CRYSTAL SPACE, 2004) e Genesis3D (ECLIPSE ENTERTAINMENT, 2004). Os mais conhecidos proprietários são: 3D Game Studio (CONITEC CORPORATION, 2004) e Fly3D (PARALELO COMPUTAÇÃO, 2004) (BATTAIOLA et al., 2001, p. 30).

2.4.1 Gerenciador de objetos

O gerenciador de objetos, basicamente, carrega os objetos do jogo a partir de um arquivo e controla seus ciclos de vida (BATTAIOLA et al., 2001).

A carga de objetos é identificada pela leitura de um arquivo texto ou binário, que mantém informações sobre o desenho dos componentes visuais e pode, ao ser lido, montar uma cena, um componente do jogo ou efetuar algumas configurações. Normalmente os motores possuem funções que permitem carregar um grande número de formatos de objetos, facilitando o trabalho para o *game designer*.

O fluxo de um jogo é planejado, desenvolvido e sustentado pelo programador, porém o controle sobre os objetos de um jogo é do motor 3D. Uma das muitas utilidades do gerenciador de objetos é que ele pode compartilhar a memória facilmente entre todo o jogo, diminuindo o volume de memória ocupado e, algumas vezes, aumentando a velocidade do processamento.

Na prática, em um motor simples, o gerenciador de objetos invoca métodos que devem ser implementados pelo usuário. Estes métodos indicarão ações ou eventos acontecidos no jogo, por exemplo: tiros, colisões e movimentação.

2.4.2 Gerenciador gráfico

Uma estrutura chamada *grafo de cena* armazena todas as informações necessárias para que uma cena possa ser montada e renderizada pelo gerenciador gráfico. A estrutura é composta de nós interligados na forma de árvore, portanto, com uma única raiz e sem ciclos. Utiliza-se um grafo de cena para agilizar as rotinas de renderização, separando os objetos em grupos e sub-grupos de forma a permitir que objetos mais complexos sejam formados por objetos simples. Com este agrupamento é possível manipular um componente visual complexo como um único objeto, simplificando o código e obtendo um melhor desempenho computacional.

Cada nó de um grafo de cena pode ser representado por um grupo de outros nós ou um objeto, que, por sua vez, pode ser um conjunto de pontos ou uma imagem. O conjunto de pontos, geralmente chamado de *Mesh*, possui informações suficientes para que seja criado um objeto completo na cena. Por exemplo, um *Mesh* de círculo, possui o ponto central, o raio e a textura ou a cor da imagem. Caso seja um polígono, ao invés de um ponto central e um raio é utilizado um conjunto de pontos e um conjunto de formação de faces, onde cada ponto é relacionado com seus vizinhos criando a face do objeto. Objetos animados, imagens 2D, luzes e câmeras também são representados como nós no grafo de cena.

O grupo difere do nó pois a sua função não é desenhar alguma imagem, mas sim controlar objetos próximos ou que atuam em conjunto no espaço gráfico. Quando um único grupo é alterado, o gerenciador gráfico não precisa calcular toda a cena para ser mostrada na tela, o único objeto afetado é o grupo e, portanto, só ele e seus sub-grupos serão redesenhados.

Para serem utilizados com o máximo de vantagem, Eberly (2001, p. 143, tradução nossa) indica que “os nodos devem possuir informações espaciais e semânticas sobre os objetos que eles representam.”. Os nós de um grafo de cena podem receber instruções de rotação, translação e escala. Cada uma dessas instruções é executada no nó indicado e em toda a sua árvore descendente de nós. Por exemplo, em jogos de primeira pessoa é comum ter um grupo chamado de personagem, que possui todo o desenho do personagem em conjunto com uma câmera, que imita o olho do personagem. Se o personagem se mover, a câmera deve seguí-lo, então, coloca-se a câmera dentro do grupo do personagem e, ao rotacionar ou mover, o evento é computado

para o grupo inteiro, incluindo a câmera.

Para ganhos de desempenho computacional, o grafo de cena implementa o corte de visão, ou em inglês, *culling*. Esta característica permite ao renderizador conhecer o estado visual do nó e se não puder ser visto pelo jogador, o gerenciador gráfico irá ignorá-lo, assim como seus filhos. Uma boa programação sobre o grafo de cena pode aumentar consideravelmente a velocidade de um jogo.

2.5 JOGOS EM JAVA PARA DISPOSITIVOS MÓVEIS

Normalmente, os celulares e PDAs já são vendidos com alguns programas. Se o cliente desejar aumentar a sua gama de possibilidades outros podem ser encontrados na internet, geralmente na página do fabricante. Esses sistemas são desenvolvidos em parceira com os fabricantes dos dispositivos, exigindo muita interação entre os dois e, com isso, dificultando a participação da comunidade (BATTAIOLA et al., 2001). Para solucionar esse problema, foi especificado o *Java 2 Micro Edition* (J2ME) (SUN MICROSYSTEMS, 2004a).

O J2ME é a versão do Java para dispositivos móveis. Esta versão é especialmente preparada para aparelhos com recursos escassos e contém tudo o que é necessário para desenvolver aplicações portáteis. O J2ME ainda se divide em vários grupos (SUN MICROSYSTEMS, 2004a):

a) dispositivos móveis: que compreende os PDAs em geral, palm tops, pockets pc, set-top boxes, entre outros;
b) dispositivos móveis limitados: celulares e outros dispositivos com baixo processamento;
c) demais sistemas embarcados: outros sistemas embarcados com algumas bibliotecas específicas como JavaTV, *Java 2 Enterprise Edition* (J2EE) Client e Java Card.

Esta divisão de grupos permite mais flexibilidade da tecnologia, conseguindo assim atender a todos os dispositivos, tendo eles um baixo ou alto nível de processamento. O J2ME também disponibiliza opções como: programação orientada a objetos, manipulação de telas, leitura dos teclados e conectividade que, muitas vezes, não são encontradas em outras plataformas.

Pelo lado negativo, o J2ME traz algumas limitações como: ausência de aritmética de ponto flutuante, não permite acessar diretamente os *pixels* da tela, ausência de som, baixo poder de processamento e ausência de primitivas gráficas do tipo polígono.

Até pouco tempo atrás, o J2ME era considerado muito lento e limitado. Mas os novos celulares e a tecnologia Java evoluíram e conseguiram resolver muitos de seus problemas. Por este motivo, o desenvolvimento de aplicações para celulares aumentou consideravelmente nos últimos anos. Atualmente é uma das tecnologias com mais investimento e da qual ainda espera-se muito (AARNIO, 2004).

A Nokia, fabricante de dispositivos móveis, liderou a JSR-184, chamada de Mobile 3D Graphics API (M3G) (NOKIA, 2003) que foi iniciada em 2002 e concluída em dezembro de 2003. A M3G padroniza o desenvolvimento de bibliotecas para montar e visualizar estruturas 3D. Espera-se que esta JSR seja implementada por todos os fabricantes de dispositivos móveis limitados, com modificações e extensões apropriadas para os seus aparelhos, aumentando as possibilidades de desenvolvimento com J2ME.

2.6 A MOBILE 3D GRAPHICS API FOR J2ME

A M3G, segundo Mahmoud (2004, tradução nossa), ”define rotinas de baixo e alto nível para tornar eficiente e criar iteratividade de gráficos 3D para dispositivos com pouca memória e poder de processamento, sem suporte de hardware ou para operações com pontos flutuantes”. Embora a definição faça menção a dispositivos sem suporte de hardware para 3D, praticamente, só os dispositivos que implementam alguma função nativa da *Application Programming Interface* (API) 3D conseguem uma velocidade de renderização aceitável. Os celulares Nokia, por exemplo, utilizam uma implementação nativa da OpenGL ES (KHONOS GROUP, 2004), uma versão simplificada da OpenGL, para aumentar a velocidade de carga, das rotinas de renderização e detecção de colisão. As outras fabricantes estão implementando estruturas similares. A evolução da tecnologia é rápida criando aparelhos com o triplo de processamento em questão de meses (JBENCHMARK, 2005).

A M3G foi especificada para as versões *Mobile Information Device Profile* (MIDP) 2.0 e *Connected Limited Device Configuration* (CLDC) 1.1 (SUN MICROSYSTEMS, 2004a). O

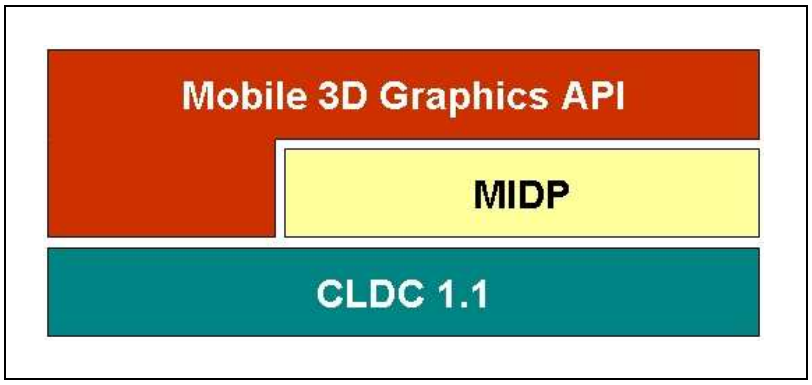
CLDC é uma configuração que define os recursos da máquina virtual e as bibliotecas principais para J2ME, enquanto o MIDP é um perfil para dispositivos portáteis definindo APIs como a de interface com o usuário, redes e conectividade, armazenamento, entre outros. Quando esta monografia estava sendo escrita estes recursos só existiam em celulares caros, deixando uma grande parcela da população com versões mais antigas do Java em seus celulares e impossibilitando o uso da API 3D.

O grupo que definiu a JSR 184 (M3G), definiu um conjunto das capacidades que a API deve suportar:

a) trabalhar em *retained-mode*, importando os grafos de cena de algum lugar, ou em *immediate-mode*, permitindo ao desenvolvedor criar seus próprios grafos de cena;
b) a API deve importar *Meshes* (Objetos 3D), texturas e grafos de cena;
c) os dados devem estar em formato binário para diminuir o tamanho do armazenamento e a transmissão;
d) deve ser possível implementar a API sobre a OpenGL ES, sem recursos de ponto flutuante de hardware;
e) deve implementar algum meio de existir valores com ponto flutuante para evitar erros de imagem;
f) ROM e RAM ocupadas devem ser mínimas. A API deve ocupar menos de 150 KB;
g) deve implementar garbage collection;
h) deve ser interoperável com outras API Java, especialmente o MIDP.

A API está definida e deve ser implementada dentro do pacote *javax.microedition.m3g* contendo 30 classes que estão brevemente definidas abaixo. As classes podem ser divididas em 6 grupos: básicas, nós de grafo de cena, carga de arquivos e classes de baixo nível, atributos visuais, modificadoras, manipuladoras de animação e colisão.

A arquitetura final esta exposta na fig. 2.2.



Fonte: Mahmoud (2004)

Figura 2.2 – Arquitetura M3G

2.6.1 Básicas

O grupo de classes básicas são as classes principais da M3G, que se responsabilizam por desenhar na interface toda a estrutura do grafo de cena e servirem como base para implementações mais específicas estando dentro do conjunto da API ou não. Participam deste grupo apenas duas classes:

a) *Graphics3D*: é criada apenas uma vez para cada máquina virtual. Mantém o contexto e renderiza o mundo de duas formas *retained-mode*, onde redesenha um mundo (*World*) completo, ou em *immediate-mode*, onde redesenha partes de grafos de cena de acordo com os parâmetros do desenvolvedor;

b) *Object3D*: classe base de todos os objetos visíveis da M3G. Possui métodos, como o *setUserObject*, que são importantes para armazenar informações personalizadas sobre os objetos.

2.6.2 Nós do grafo de cena

Este grupo de classes constitui-se de classes básicas e especializadas que podem compor um grafo de cena. Algumas delas produzem objetos estáticos e outras objetos dinâmicos. Elas são:

a) *Node*: nó de um grafo de cena. Classe base para *Camera*, *Light*, *Mesh*, *Sprite3D* e *Group*;

b) *World*: nó de cena especial para o objeto raiz do grafo;

c) *Camera*: permite projetar um modelo 3D para 2D de acordo com uma posição de origem e uma perspectiva;

d) *Light*: acrescenta luzes e sombras ao modelo;

e) *Sprite3D*: representa uma imagem 2D como um Objecto 3D;

f) *Mesh*: representa um objeto 3D definido como uma superfície formada por uma malha de polígonos;

g) *Group*: mantém um grupo de outros nós;

h) *SkinnedMesh*: nó próprio para trabalhar com esqueletos e interligação de vértices em uma estrutura;

i) *MorphingMesh*: um objeto 3D que possui alteração na distribuição dos seus pontos 3D no mundo.

2.6.3 Carga de arquivos e classes de baixo nível

Esta categoria de classes detém os objetos utilizados para a carga de um grafo de cena, determinando pontos, faces e cores. São eles:

a) *Loader*: carrega um grafo de cena ou qualquer *Object3D* a partir de um arquivo binário com formato M3G;

b) *IndexBuffer*: define a forma de conexão entre vértices para formar um objeto geométrico;

c) *TriangleStripArray*: determina quais pontos de uma estrutura 3D devem formar uma face;

d) *VertexArray*: array para armazenar os vértices, vetores normais, texturas e cores;

e) *VertexBuffer*: armazena vários *VertexArrays* para constituir um objeto 3D.

É possível trabalhar só com essas classes ao invés de criar um grafo de cena. Mas, embora isto acrescente muita velocidade e baixo uso de memória, o desenvolvedor acaba ficando preso na sua modelagem 3D, pois estas classes não permitem que os vértices sejam alterados. O único meio de interagir com o desenho é convertendo todos esses dados para um grafo de cena completo.

2.6.4 Atributos visuais

Nesta categoria, encontram-se as classes que mantém informações de cores, texturas e demais atributos visuais para os objetos tridimensionais definidos na seção 2.6.2. Estão definidas abaixo:

a) *Appearance*: um conjunto de componentes que define atributos para a renderização de objetos 3D como um *Mesh* ou um *Sprite3D*;

b) *Background*: define atributos para limpar o campo de visão. Por exemplo permite configurar a cor ou imagem a ser exibida no fundo, independente do local da câmera;

c) *CompositingMode*: semelhante ao *Appearance*, mas encapsula informações por pixels ao invéis de informações globais. Atua compondo uma ou mais cores;

d) *Fog*: classe para controlar nativamente névoas no jogo;

e) *Image2D*: uma imagem 2D, muito usada para texturas ou como fundo do jogo;

f) *Material*: semelhante ao *Appearance*, encapsula atributos para luz dependendo da incidência sobre o objeto;

g) *Texture2D*: semelhante ao *Appearance*, mas com dados relevantes para texturas;

h) *PolygonMode*: uma classe semelhante a aparência, mas com informações a nível de polígono.

Todas elas atuam em conjunto com o grafo de cena, cada um em seu caso. Elas foram separadas do grafo de cena para evitar o uso de memória desnecessária, visto que boa parte dos nós não farão referência a atributos, pois serão grupos de nós. Não há necessidade de instanciá-las.

2.6.5 Modificadoras

O grupo de classes modificadoras são as classes responsáveis por alterações nos modelos tridimensionais. Todas as rotinas de transformação são feitas pelas duas classes apresentadas abaixo:

a) *Transform*: possui uma matriz de ordem 4 para efetuar transformações de escala, rotação e translação nos objetos 3D. Toda a transformação de objetos do grafo de

cena é feita por esta classe, pois é mais rápida e permite que o objeto volte ao estado normal facilmente;

b) *Transformable*: classe base para todos os ítens que podem ser alterados pelo *Transform*.

Embora as classes que especializam a classe *Node* já contenham métodos para as 3 operações básicas, não é conveniente usá-las, visto que alteram diretamente o modelo 3D. A classe *Transform* aumenta as possibilidades de operações sobre o modelo e o tempo adicional para usá-la é insignificante.

2.6.6 Manipuladoras de animações

As manipuladoras de animações são classes de um grupo especial criado para agilizar o processamento e facilitar a utilização de animações nos jogos. No geral, elas são instanciadas com um conjunto de estados do modelo 3D e ficam em uma repetição pré-determinada e intermitente destes estados. Elas são:

a) *AnimationController*: controla a posição, velocidade e peso de uma sequência de animações;

b) *AnimationTrack*: associa um *KeyframeSequence* com um *AnimationController* alterando alguma propriedade da animação diretamente;

c) *KeyframeSequence*: cria uma animação como uma sequência de *keyframes*.

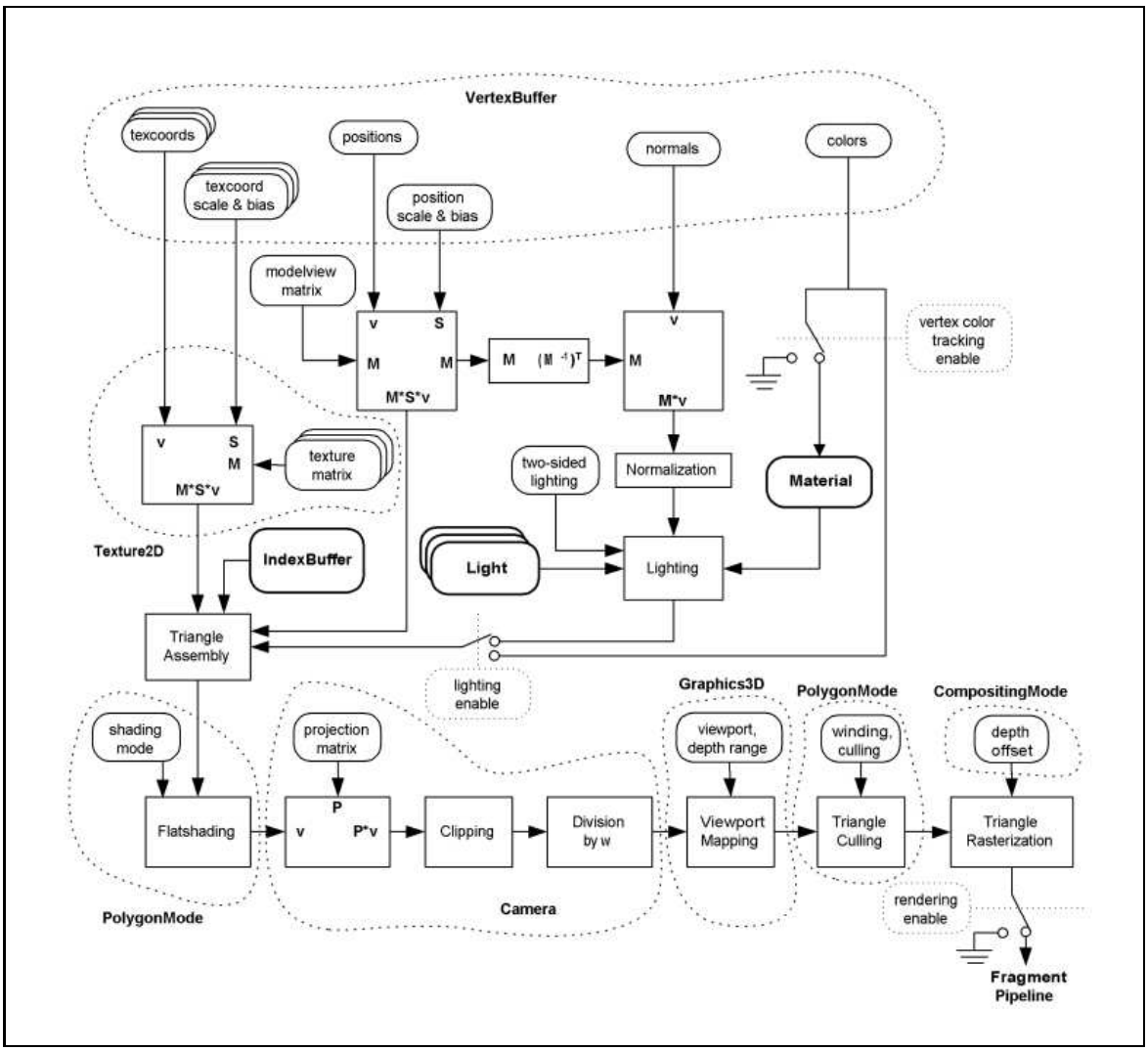
2.6.7 Colisão

A colisão é feita pela classe *Group*, no método *pick* que retorna uma instância da classe *RayIntersection*. Nela, encontra-se armazenado o objeto da colisão, objeto de origem, vetor de direção e distância entre origem e destino. Este método é utilizado pelas rotinas de detecção de colisão implementadas neste trabalho e descritas na seção 3.9.

Segundo a especificação M3G, algorítmo do método *pick* deve tratar o *culling* de objetos, onde objetos não visiveis em relação a câmera, ou ao estado do grafo atual, não devem ser processados pelo cálculo. Por experiência prática o *pick* é um dos algoritmos mais lentos da M3G e deve receber melhorias muito em breve.

2.6.8 Unindo as classes

Na fig. 2.3 é apresentada, de forma completa, toda a estrutura da biblioteca M3G. Como mencionado, o *VertexBuffer* mantém todos os dados do modelo e as demais classes trabalham com ele. Os vetores normais e as cores são utilizadas para determinar diretrizes para que as luzes (*Light*) possam ser desenhadas. Adicionando os vértices e texturas, devidamente processados, com as luzes, o montador de triângulos (*Triangle Assembly*) pode organizar toda a estrutura e repassar para as rotinas de projeção. Nesta fase já são desconsiderados todos os objetos que a câmera atual não está mostrando. Em seguida a classe *Graphics3D* entra em ação mapeando o modelo 3D para o *viewport*, uma área bidimensional predefinida pelo desenvolvedor e visualizada pelo jogador, renderizando a estrutura.

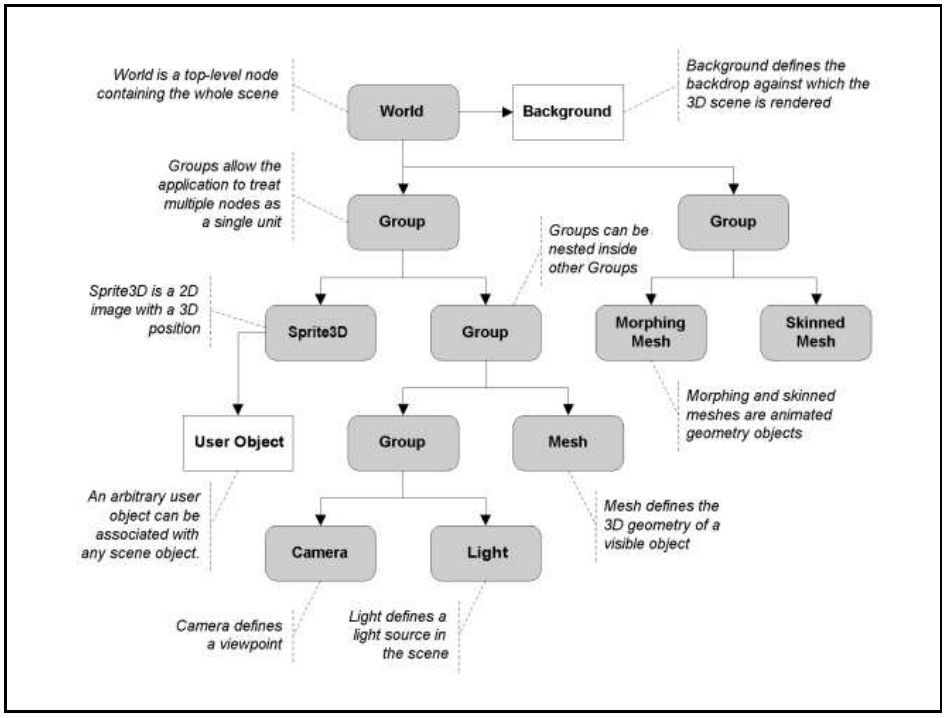


Fonte: Nokia (2003)

Figura 2.3 – Especificação da M3G

2.6.9 O grafo de cena da M3G

Segundo (NOKIA, 2003, p. 204, tradução nossa), "um grafo de cena é contruído a partir de uma hierarquia de nós. Em um grafo de cena completo, todos os nós são ligados entre si com uma raiz comum que é denominada *mundo*."Um exemplo de grafo de cena pode ser visto na fig. 2.4.



Fonte: Nokia (2003, p. 204)

Figura 2.4 – Um grafo de cena

O primeiro nó indica o mundo, sendo que abaixo dele é montada toda a cena. O mundo possui, opcionalmente, um *Background*, uma cor ou imagem de fundo. O *Background* não é considerado um nó, mas sim um atributo do mundo, ou seja, não participa do grafo de cena e não possui uma estrutura hierárquica e, portanto, não pode ser utilizado em qualquer outro local do grafo de cena.

Os grupos concatenam nós na estrutura, determinando que objetos devem ser tratados de maneira equivalente. Os nós se dividem entre os vários nomes apresentados na seção 2.6.2, incluindo câmeras e luzes.

2.6.10 O formato de arquivos M3G

A M3G possui um formato próprio para carregar um modelo tridimensional mas, o seu uso está comprometido devido a falta de suporte dado a este arquivo pelas ferramentas de modelagem e, devido ao fato do arquivo ser gravado em formato binário, o que impede que seja alterado em qualquer editor de textos.

No entando, este formato é o mais rápido e mais leve para ser carregado, visto que está definido com a hierarquia de classes da M3G, ou seja, não é necessário nenhum processamento adicional para encontrar valores e preencher o modelo 3D. Eles já estarão prontos para serem adicionados aos objetos.

Algumas ferramentas testadas, como o *M3G Exporter* (M3G EXPORTER, 2005) para o 3D Studio Max (DISCREET, 2005) e o *Mascot Capsule Toolkit for M3G* (HI CORP, 2005), apresentaram muita instabilidade e falta de recursos durante as rotinas de exportação, além de serem *plugins* para ferramentas proprietárias e de alto custo.

2.7 O FORMATO DE ARQUIVOS WAVEFRONT - ARQUIVOS OBJ E MTL

Por Otto (2002, tradução nossa) , ”modelar é o processo de descrever um objeto ou cena que possa gerar uma imagem”. A cena normalmente é modelada em ferramentas específicas como o 3D Studio Max (DISCREET, 2005) ou o Blender (BLENDER FOUNDATION, 2005). Estas ferramentas exportam o modelo em vários formatos, incluindo o escolhido para este trabalho, o Wavefront (O’REILLY & ASSOCIATES INC, 1996).

Wavefront é um formato padrão para armazenar modelos 3D. Foi criado para ser utilizado com o *Wavefront Advanced Visualizer* da Viewpoint DataLabs. Hoje em dia esta especificação é reconhecida no mundo todo possuindo uma excelente aceitação de mercado. Alguns *designers* consideram o Wavefront um padrão de comunicação entre as ferramentas de modelagem, visto que é um dos únicos formatos suportados por várias ferramentas de desenho 3D. É com este arquivo que o motor de jogos desenhará toda a cena.

Os arquivos exportados são arquivos texto e sua extensão padrão é a .obj. Eles suportam polígonos e estruturas geométricas livres como superfícies e curvas. O presente trabalho con-

```
# <algum texto>
v <float><float><float>
vn <float><float><float>
vt <float><float>
g <nome do grupo>
mtllib <arquivo>
usemtl <nome de um material>
f <int[/int[/int]]>[<int[/int[/int]]>[<int[/int[/int]]>]]
...
```

Quadro 2.1 – Especificação de um arquivo Wavefront Obj

sidera somente as estruturas poligonais definidas no arquivo, sendo que as outras são ignoradas pela classe leitora.

O padrão Wavefront também utiliza-se de outro arquivo. Este arquivo tem a extensão .mtl e armazena cores e texturas do ambiente ou de um grupo de objetos 3D.

A estrutura do arquivo não é complexa, mas tem muitos detalhes. Basicamente, consiste em um item por linha, sendo o tipo deste item definido pela primeira palavra encontrada. Serão tratados os tipos: **v**, **vn**, **vt**, **f**, **g**, **mtllib** e **usemtl** além do comando de comentário **#**. Um modelo de arquivo obj está descrito no quadro 2.1.

O caractere **#** representa um comentário de linha representando que nenhuma informação desde o caractere **#** até o fim de linha deva ser considerada. O parâmetro **v** representa um vértice tridimensional e é acompanhado pelos valores **x**, **y** e **z** respectivamente. Estes vértices são enumerados seqüencialmente iniciando de 1 até o final do arquivo formando um índice para a leitura das faces.

O indentificador **vn** é um vetor tridimensional normal (x,y,z) e, assim como os vértices, são enumerados seqüencialmente iniciando de 1 até o final do arquivo. O vetor normal é associado a um único vértice pelo seu número seqüencial. É baseado nele que a M3G executa as rotinas de visualização e de luz e sombra, entre outras.

O indentificador **vt** é um vetor textura, que também é numerado seqüencialmente, iniciando de 1 até o final do arquivo. Este vetor é interpretado de maneira diferente da convencional pela M3G, onde o primeiro float é o eixo **x** do modelo, porém o segundo é o eixo **y** ao

```
newmtl <nome >
Ka <float><float><float>
Kd <float><float><float>
illum [1,2]
Ks <float><float><float>
d <float>
Ns <float>
Tr <float>
map_Kd <arquivo>
```

Quadro 2.2 – Especificação de um arquivo Wavefront MTL

contrário. Ou seja, o primeiro aumenta da esquerda para a direita, enquanto o segundo aumenta de baixo para cima. Visto esta diferença, a classe responsável por ler este arquivo é responsável por inverter os números encontrados na segunda posição.

O identificador **g** determina um grupo, ou seja, tudo que é lido após o **g** pertencerá a um determinado grupo, como a definição de textura e o conjunto de faces que dá forma a um polígono. Todas as faces lidas até o surgimento de um **g** são colocadas em um grupo sem nome, para que seja possível a renderização. O nome do grupo é importante para que o usuário do motor localize seus objetos.

O identificador **f** indica uma face triangular. Somente faces triangulares são interpretadas pela classe leitora deste arquivo. Os números inteiros que seguem representam os números seqüenciais de vértices, vetores normais e vetores de textura respectivamente separados por "/". Para a importação da M3GE é necessário que os números representando os três vértices de cada ponto sejam iguais. Ou seja, se a face ligar os pontos 1, 2 e 5, os vetores normais e vetores de textura também serão 1, 2 e 5.

O identificador **mtllib** indica que é necessário importar um outro arquivo antes de dar prosseguimento da leitura deste. O arquivo possui o formato de um WaveFront MTL (MCNAMARA, 2000) e deve ter a extensão .mtl e indica texturas e as cores dos objetos importados. Estas estruturas são utilizadas pelos objetos 3D indicando como devem ser pintadas as suas faces. O quadro 2.2 mostra a estrutura de um arquivo MTL.

O identificador **usemtl** indica que o grupo atual está utilizando um material com o nome

que está em anexo a instrução. O mesmo nome deverá estar especificado no arquivo MTL. Caso este material não existir, uma exceção é lançada. Ao iniciar um novo grupo o material é zerado, devendo ser novamente referenciado pela usemtl.

No arquivo MTL, a instrução **newmtl** indica a construção de um novo material. Todas as declarações abaixo dele até o aparecimento de outro newmtl estarão sendo armazenadas no mesmo material. A instrução **Ka** define a cor que influencia a luz ambiente quando o objeto permanecer na sombra. Seu padrão é 0.2, 0.2 e 0.2. Já a **Kd** define a cor de difusão do material e será utilizada quando o objeto for iluminado parcialmente pela luz ambiente. O seu padrão é 0.8, 0.8 e 0.8. O **Ks** indica a cor que será utilizada caso o objeto seja totalmente iluminado. O seu padrão é 1.0, 1.0 e 1.0. As cores estão em uma faixa de 0 a 1 e são convertidas em cores RGB.

A instrução **d** indica o valor *alpha* do material. Este valor é adicionado antes de cada cor apresentada acima, formando um formato de cor ARGB. O valor padrão é 1.0. A instrução **illum** indica a utilização ou não da cor de objeto iluminado. Caso o número seja 1 esta cor não é informada, caso seja 2 deve ser informada. O identificador **Tr** indica a transparência do material, variando de completamente transparente onde vale 0 até ser opaco com valor 1. Há uma certa confusão entre os valores de **d** e de **Tr**, sendo que algumas ferramentas exportam os dois valores trocados.

O comando **map_Kd** importa um arquivo de imagem que está no mesmo diretório do atual. Esta imagem representa a textura desenhada nos objetos. A imagem deve ser quadrada medindo lateralmente um número na potência de dois (1, 2, 4, 8, 16, etc.). Esta é uma exigência da M3G e não foi tratada neste trabalho e, portanto, o desenvolvedor de jogos deve utilizar as imagens apropriadas.

2.8 CONSIDERAÇÕES FINAIS E TRABALHOS CORRELATOS

Dentre os módulos vistos na seção 2.4 os seguintes serão trabalhados durante o desenvolvimento da M3GE, porém, conforme indicação nos objetivos e requisitos do trabalho, não serão totalmente implementados:

a) gerenciador principal;

b) gerenciador de entrada;

c) gerenciador gráfico;

d) gerenciador de mundo;

e) gerenciador de objetos.

A tecnologia Java, o J2ME e a M3G criam uma boa portabilidade, mesmo que limitada a dispositivos com CLDC 1.1 pela M3G, o que exclui *palm tops*, *pockets pc* e outros PDAs. Até o presente momento, o autor deste trabalho desconhece trabalhos publicados livremente com o mesmo tema deste. Assim, Eberly (2001) será utilizado como base teórica para a aquitetura do motor. A diferença é que o motor não será para ambiente desktop como Eberly constrói em seu livro, mas sim móvel.

O doutor Andrew Davison está construíndo um livro sobre a API M3G e está publicando os rascunhos dos capítulos do livro em sua página na internet para avaliação do público. No primeiro capítulo (DAVISON, 2004) ele utiliza as bibliotecas de renderização 3D do Java para *desktop* gerando trechos de código, em linguagem java, para a construção do modelo via *arrays*. Como o capítulo trabalha com arquivos wavefront será a base teórica mais próxima da parte de leitura e criação do grafo de cena da M3G definido neste trabalho.

E, por fim, Nokia (2004b) será utilizado como base teórica e exemplo prático para os possíveis problemas que podem ser encontrados nos dispositivos móveis, tais como: pouca memória, baixo processamento e necessidade de uso de algoritmos especiais para computação móvel.

3 DESENVOLVIMENTO

A M3GE foi desenvolvida seguindo as três etapas básicas de construção de software: análise, implementação e testes. Não foi utilizada nenhuma metodologia de mercado como *Extreme Programming* (WUESTEFELD, 2001), *Scrum* (CONTROL CHAOS, 2005) ou o Processo Unificado (IBM, 2005), o que permitiu ao autor deste trabalho a mobilidade para quebrar regras destas metodologias e desenvolver um software utilizando o modelo de processo em espiral, com vários protótipos para testes.

Quando se programa para dispositivos limitados, é necessário seguir a máxima chamada *Keep It Simple, Stupid* (KISS) (WIKIPEDIA, 2005), que prega que algo só deve ser feito se realmente é necessário, nada de previsões ou tentativas. Isto tudo para que a velocidade e o tamanho do motor não comprometa o jogo.

A análise de todo o projeto seguiu os parâmetros da orientação a objetos utilizando alguns diagramas da *Unified Modeling Language* (UML) (OBJECT MANAGEMENT GROUP, 2004), entre eles o diagrama de componentes e de sequência. A ferramenta para construir esses diagramas foi a Jude Bamboo 1.3 (EIWA SYSTEM MANAGEMENT, INC, 2005).

Para o desenvolvimento foi utilizado a IDE livre Eclipse (ECLIPSE FOUNDATION, 2004) com o *Sun Wireless Toolkit* (WTK) (SUN MICROSYSTEMS, 2004b) e o *plugin* EclipseME (ECLIPSEME TEAM, 2005) que disponibiliza facilidades para trabalhar com emuladores de celulares. Para testes foi utilizado a implementação de referência da M3G da Nokia (NOKIA, 2004a), que já vem com emuladores próprios e o *Siemens Mobility Toolkit* (SMTK) (SIEMENS AG, 2005b) com emuladores para o Siemens CX65 (SIEMENS AG, 2005a), além do próprio WTK.

Para a modelagem 3D, foi utilizada a ferramenta Art of Illusion (AoI) (EASTMEN, 2005) que, além de prover todos os recursos necessários, é um software livre simples e de fácil utilização. Ele exporta os arquivos *Wavefront* que serão utilizados no trabalho. Para alguns testes com a M3G foi utilizada uma versão *trial* do 3D Studio Max (DISCREET, 2005).

O desenvolvimento foi iniciado no sistema operacional Microsoft Windows XP, e concluído com o linux Fedora Core 2. A migração de sistema operacional foi feita assim que a Sun (SUN MICROSYSTEMS, 2005) lançou a versão do WTK para o ambiente Linux. Para escrever a monografia foi utilizado o preparador de documentos Latex (LATEX PROJECT TEAM, 2005) com o editor de textos Kile (KILE, 2005)

A seguir estarão listados os processos de desenvolvimento assim como a lógica do material desenvolvido.

3.1 REQUISITOS PRINCIPAIS

O motor de jogos deve cumprir os seguintes requisitos:

a) carregar e desenhar um ambiente virtual a partir de um arquivo de configurações: dado um arquivo de configuração de ambiente, com os pontos desenhados em um espaço 3D, o sistema deve ler este arquivo e gerar as imagens. O Formato do arquivo é o *Wavefront*, descrito na seção 2.7;

b) troca de câmeras no cenário: permitir que o desenvolvedor crie câmeras para visualização do cenário em diferentes pontos, garantindo uma maior interação do usuário final. O número de câmeras é indefinido e cada câmera deve assumir posições iniciais pré estabelecidas pelo desenvolvedor;

c) movimentação de personagens no cenário: permitir que os personagens se movimentem pelo cenário, a partir de uma entrada de dados informada pelo usuário. A movimentação deve acontecer em todos os sentidos e ângulos de um ambiente 3D;

d) portabilidade: oferecida pela tecnologia Java;

e) velocidade: o maior desafio é construir um software rápido e, ao mesmo tempo, portável.

Ao finalizar o trabalho, devido a ganhos com tempo de desenvolvimento, outros requisitos foram implementados, a detecção de colisão entre personagem e objetos da cena, descrito na seção 3.9, um módulo de eventos e um aplicativo para preparar o arquivo Obj para a importação nos celulares também foi criado.

3.2 SOLUÇÃO PROPOSTA: MOBILE 3D GAME ENGINE

Como pode ser visto na fig. 3.1, a M3GE foi projetada para ser utilizada como uma API anexada a M3G. Ou seja, mesmo usando a M3GE é possível utilizar a M3G diretamente. As duas bibliotecas interagem entre si, proporcionando ao desenvolvedor do jogo flexibilidade e velocidade quando necessária.

Figura 3.1 – Visão Geral da Mobile 3D Game Engine

O projeto M3GE foi dividido em dois grandes componentes: o responsável pela leitura de um arquivo *Wavefront* (O'REILLY & ASSOCIATES INC, 1996) e o responsável pelo motor de jogos ou *core*, como pode ser visto na fig. 3.2.

O *core* possui todas as funções básicas necessárias para que o jogo possa ser executado. É sobre ele que estão as implementações de cada desenvolvedor, adicionando lógica e o enredo ao jogo. Este componente é dividido em oito classes: Configuration, Cameras, CollisionDetection, EngineCanvas, KeysManager, Player, UpdateListener, UpdateSceneTimerTask.

Em comparação com uma arquitetura de motor de jogos 3D, como apresentado na seção 2.4, a M3GE implementa os módulos do motor de jogos nas classes listadas na fig. 3.2. A fig. 3.3 apresenta um diagrama de implantação da UML, onde mostra como as classes do motor de jogos implementado neste trabalho, se comunicam com as classes disponibilizadas pela API M3G.

Quando o usuário pressiona uma tecla, esta informação é repassada para a classe *KeysManager*, que atua como gerenciador de entrada. Após processado, a informação passa para a classe *EngineCanvas* que é o gerenciador principal do motor de jogos 3D. A *EngineCanvas* trabalha com instâncias de *Player*, *Cameras* e *World*, foram construídas separando as re-

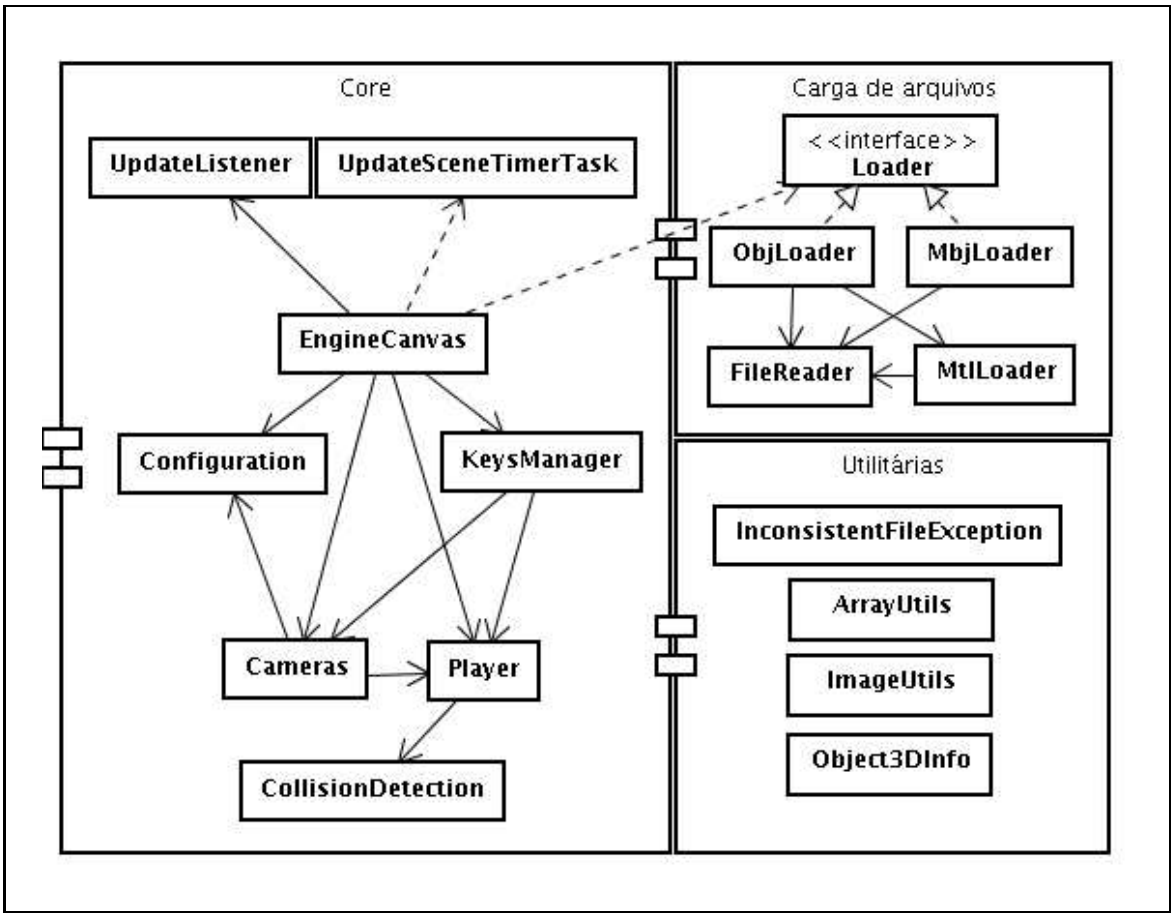


Figura 3.2 – A arquitetura Mobile 3D Game Engine

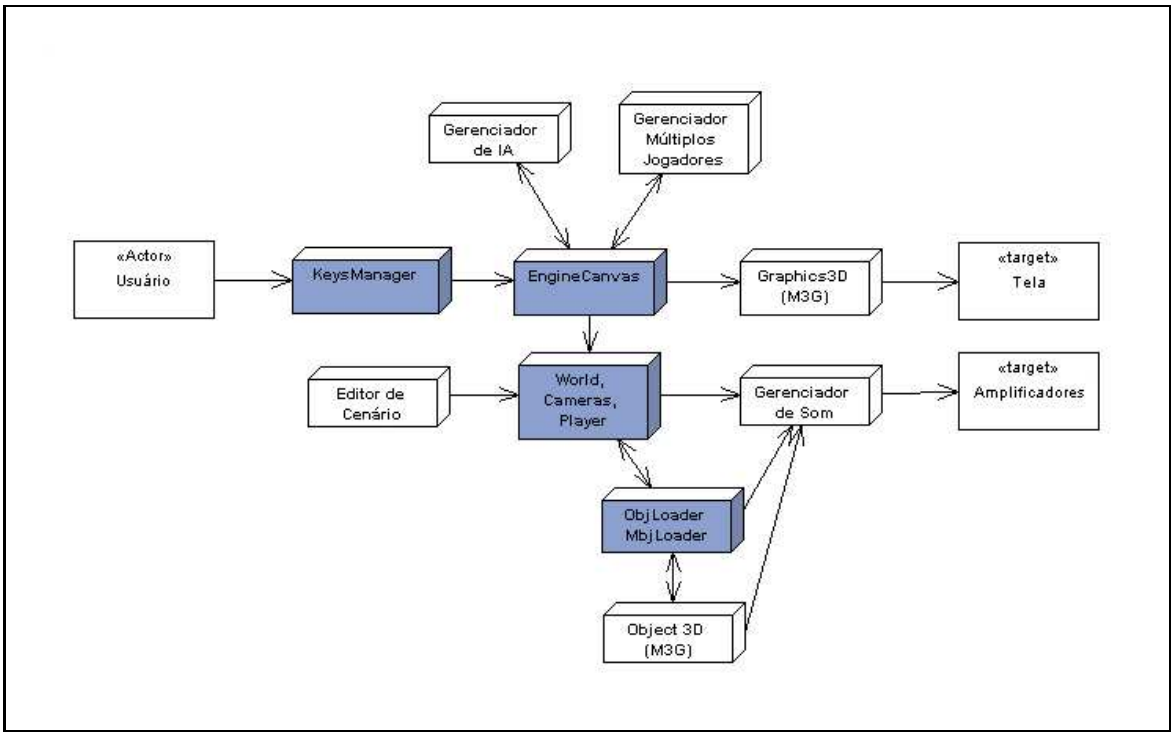


Figura 3.3 – A arquitetura da M3GE comparada a arquitetura padrão de um motor de jogos

sponsabilidades de um único gerenciador de mundo, mas atuando em conjunto.

As classes *ObjLoader* e *MbjLoader* atuam criando instâncias de *Object3D*, os nós de grafo de cena, e gerenciando o seu ciclo de vida. Portanto atuam como dois gerenciadores de objetos. O gerenciador gráfico já é implementado pela M3G, a classe *Graphics3D*, desenha o modelo 3D em um visualizador 2D.

Os componentes escuros são implementados neste trabalho exceto a classe World, que é disponibilizada pela M3G. As classes implementadas estão especificadas adiante.

3.2.1 A classe Cameras

Esta classe é responsável por manter toda a estrutura de câmera do jogo, carregando as configurações do arquivo, gerenciando posicionamento de cada câmera no mundo e identificar qual a câmera atualmente utilizada para renderizar as imagens;

O diagrama de classe fig. 3.4 mostra todos os métodos e atributos usados na classe. As atributos inteiras *height* e *width* armazenam a informação do tamanho da tela, e os atributos *world*, *player* e *conf* armazenam o primeiro nó do grafo de cena, o nó do jogador e a classe de configuração. Todos esses atributos são preenchidas no construtor.

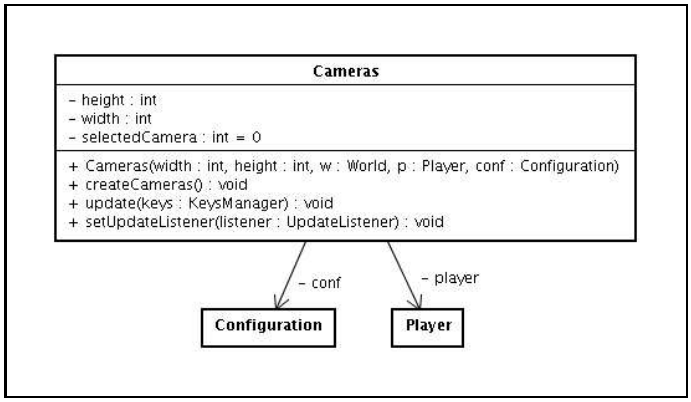


Figura 3.4 – Classe *Cameras* e suas dependências

O atributo *cameras* armazena um vetor com todas as câmeras declaradas no arquivo de propriedades pelo desenvolvedor do jogo. A variável *selectedCamera* armazena a camera selecionada pelo usuário. Este valor inteiro é um índice para o vetor *cameras*. O atributo listener mantém uma classe implementada pelo desenvolvedor do jogo, que é invocada assim

que a troca de câmeras ocorrer.

O método *createCameras()* busca a declaração de todas as câmeras do atributo *conf* e as instancia, configurando-as conforme requisitado pelo desenvolvedor. Ao final da rotina, ela configura a primeira câmera lida como a câmera ativa. É neste método que configura-se a câmera relativa ao mundo ou ao jogador, adicionando-a no nó de grafo de cena *world* ou *player* respectivamente. Se não houver câmera definida, uma excessão é lançada informando ao desenvolvedor que é necessário uma câmera.

O método *update(KeysManager keys)* é invocado pelo ciclo principal do jogo, explicado na seção seção 3.5, e atua verificando se a tecla pressionada é a tecla de troca de câmeras e trocando a câmera caso seja verdadeiro. Após trocar de câmera, a rotina invoca o método *camUpdated* do atributo *listener*, indicando a troca de câmera para o desenvolvedor.

O método *setUpdateListener(UpdateListener listener)* atua configurando a variável listener de acordo com a repassada no parâmetro. É a única maneira de adicionar um *listener* neste objeto.

3.2.2 A classe Configuration

Esta classe é responsável por carregar um arquivo de configurações do motor. Neste arquivo estão informados todos os dados que não podem ser adicionados no arquivo *Wavefront*, sejam referentes a *engine* ou as rotinas de renderização. Na seção 3.7 será visto como o arquivo de propriedades, definido neste trabalho, é formado.

O diagrama de classe da fig. 3.5 está mostrando as dependências e as possíveis operações que a classe *Configuration* pode executar. As constantes estáticas definem as chaves para o arquivo de propriedades. Cada uma delas será explicada na seção 3.7. O construtor desta classe recebe uma *String file*, ele utiliza esta *String* para localizar o arquivo de propriedades, caso não encontre as configurações padrões serão carregadas e uma mensagem de erro é escrita na saída padrão.

O método *putInKeys(String tableKey, int engineValue)* é utilizado pelo construtor e tem a função de configurar chaves não declaradas no arquivo de propriedades. Caso o arquivo não

Figura 3.5 – Classe *Configuration* e suas dependências

seja informado, ou não exista, as chaves serão incluídas através deste método. A rotina, verifica se a chave, passada no parâmetro *tableKey*, não foi importada, caso verdadeiro a rotina adiciona esta chave com o valor da variável *engineValue* nas configurações do motor.

O método *toEngine(int keyCode)* converte uma tecla pressionada pelo usuário para uma tecla utilizada no motor de jogos. Este método é invocado a cada vez que o usuário pressiona uma tecla, consultando as configurações e determinando que ação representa esta tecla.

Os métodos *getInt*, *getFloat* e *get* retornam o valor definido pelo desenvolvedor do jogo no arquivo de configurações, para a chave passada no parâmetro. Este método é utilizado por muitas classes e retorna instâncias de inteiro, ponto flutuante e *string* respectivamente.

A função *getCameraPosition(int camera)* retorna um array com todas as informações sobre uma câmera. As três primeiras posições do array são os valores **x**, **y**, e **z** onde a câmera se encontra. As próximas três posições são os valores **ax**, **ay** e **az**, determinando a posição que a câmera está virada. Os três finais são o **fovy**, o **far** e o **near**, que determinam o ângulo de visão, o quão longe e o quão próximo a câmera deverá visualizar. Este método é chamado pela classe *Cameras* para estabelecer as configurações iniciais.

A função *isCollisionDetection()* retorna um valor boleano identificando se o desenvolvedor do jogo utilizará ou não a detecção de colisão implementada no motor de jogos. A detecção de colisão será discutida na seção 3.9.

A função *getDefaults()* retorna uma lista com todas as configurações padrão utilizadas no motor de jogos.

3.2.3 A classe CollisionDetection

Esta classe, que está representada na fig. 3.6, é responsável pelo cálculo de colisão do personagem com os outros objetos visíveis na cena. O cálculo é demonstrado na seção 3.9.

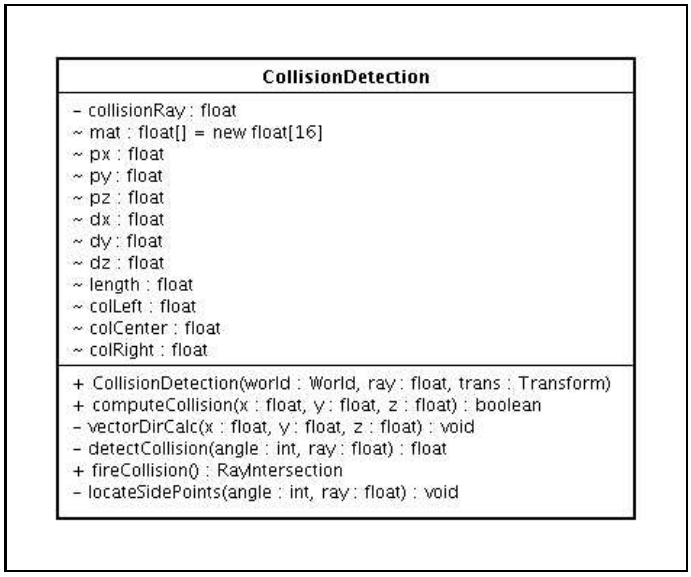


Figura 3.6 – Classe *CollisionDetection* e suas dependências

Esta classe é instanciada uma única vez para todos os cálculos de colisão. Os atributos *playerTrans*, *collisionRay* e *world* mantém, respectivamente, a classe que transforma nó do grafo de cena do personagem, o raio de colisão e a o mundo, o primeiro nó do grafo de cena. As três são configuradas pelo construtor. Os outros atributos são variáveis globais, utilizadas pelas funções definidas abaixo, e declarados desta maneira para melhorar o processamento, evitando declarações de variáveis dentro das funções.

A função *computeCollision(float x, float y, float z)* é a principal função desta classe. Ela faz o cálculo determinando se o jogador, indo para a direção apontada em **x**, **y** e **z** nos parâmetros colide ou não com algum objeto visível.

O método interno *vectorDirCalc(float x, float y, float z)* calcula a posição final após o jogador andar na direção **x**, **y** e **z** passada no parâmetro. Esta função existe, pois o método *pick*, que determina a colisão entre um ponto e um objeto 3D e é implementado pela M3G, necessita, além do ponto de partida do teste, um ponto absoluto para informar direção a testar. A função computeCollision chama este método antes de qualquer processamento. O resultado desta função, o ponto (x,y,z) de destinho e a hipotenusa entre origem e destino ficam armazenadas nas variáveis globais *dx*, *dy*, *dz* e *lenght* respectivamente. As posições de origem são colocadas nas variáveis globais *px*, *py* e *pz*.

A função *detectCollision(int angle, float ray)* calcula a colisão com os valores definidos na função *vectorDirCalc* para um ângulo de um raio de colisão. O ângulo servirá para calcular os três pontos definidos na fig. 3.6. O raio é a distância máxima para que uma colisão seja detectada nos três pontos.

O método *fireCollision()* determina a posição de origem, a posição de destino para em seguida calcular a colisão entre o ponto central do jogador e qualquer objeto em sua visão. Este método é invocado quando a tecla **FIRE** é pressionada. Este método retorna uma instância de *RayIntersecion*, classe da M3G responsável por manter dados de colisão.

O método *locateSidePoints(int angle, float ray)* localiza os pontos laterais do cálculo de colisão de acordo com o ângulo passado no parametro *angle* e a distância para o ponto central, que é passada no parâmetro *ray*.

3.2.4 A classe EngineCanvas

Esta é a classe principal da *engine*. Ela é a responsável pelo gerenciamento do ciclo de vida todos os objetos, pela chamada do *KeysManager* a cada tecla pressionada, pela renderização da imagem utilizando a M3G, pela carga do arquivo de configurações da *engine*, pela criação de câmeras independentes ou não, pela chamada ao componente de carga de arquivos *Wavefront*, e pela criação de uma classe que chama a renderização da tela em um ciclo de tempo determinado.

O diagrama de classe na fig. 3.7 mostra a classe *EngineCanvas* e as suas dependências.

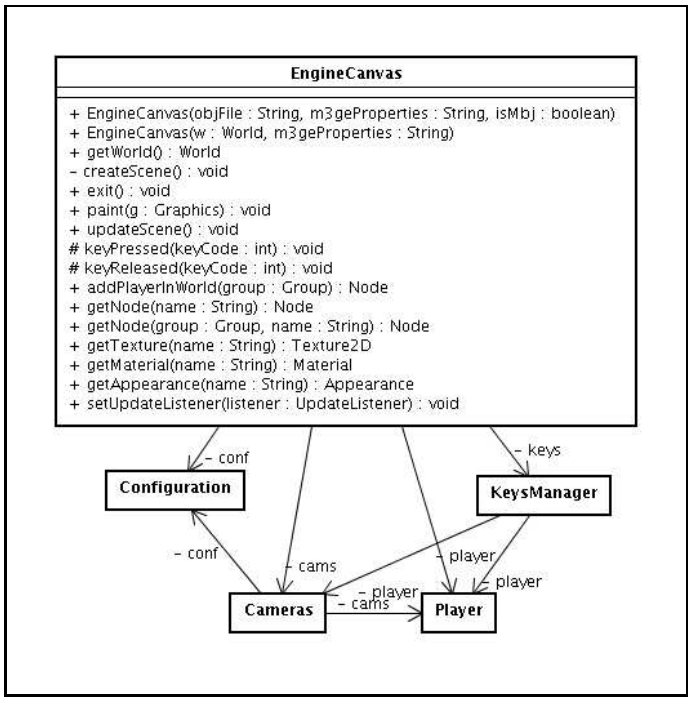


Figura 3.7 – Classe *EngineCanvas* e suas dependências

O atributo *g3d* é uma instância da classe *Graphics3D*, e é utilizado para renderizar o modelo 3D. O atributo *world* é o nó inicial do grafo de cena. O atributo *light* é a luz padrão do ambiente, e o *lightTransform* é o objeto que posicionará a luz no lugar desejado dentro do modelo 3D.

O atributo *conf* é instanciado durante o método construtor e mantém todas as configurações disponíveis a esta classe. O atributo *player* representa o nó do grafo de cena cujo seus filhos são a estrutura de um personagem do jogo, incluindo as câmeras. O atributo *cams* mantém as câmeras do jogo e controla a visualização atual do jogador, e o *keys* as teclas pressionadas.

O atributo *refreshTimer* é um relógio que invocará uma instância de *UpdateSceneTimerTask* em uma frequência determinada pelo desenvolvedor do jogo no arquivo de configurações. O atributo *updateListener* é o atributo que manterá o objeto de atualização criado pelo desenvolvedor do jogo, e informado a esta classe pelo método *setUpdateListener()*.

O artibuto *model* pode ser uma instância de *ObjLoader* ou *MbjLoader*, dependendo da escolha do desenvolvedor. Este atributo carregará o modelo de arquivos e manterá consigo todas as informações do modelo.

Os construtores da classe *EngineCanvas* carregam um modelo de arquivo ou já recebem

o modelo instanciado. Em comum entre os dois construtores está a chamada da carga do arquivo de configuração, e a chamada ao método *createScene()*, onde são instanciados todos os atributos declarados, preparando o ambiente para a execução do jogo.

O método *exit()* cancela a execução do relógio, preparando o jogo para ser desligado.

A função *updateScene()* atualiza todo o modelo 3D, baseando-se nas telas atualmente pressionadas pelo usuário e informa ao *listener* do desenvolvedor de jogos, que um *update* foi invocado. Ao final da rontina, chama o método *repaint()*, onde o modelo tridimensional é convertido para 2D e apresentado ao jogador.

Os métodos *keyPressed* e *keyReleased* são invocados automaticamente quando o jogador faz alguma ação. Ambos redirecionam o processamento para o atributo *keys*.

A função *addPlayerInWorld(Group group)* pesquisa por um nó chamado "PLAYER"no grafo de cena, caso encontre, ela o substitui pelo atributo *player*, que é um grupo de nós, e adiciona o nó PLAYER como filho do grupo *player*. A partir deste ponto, o personagem principal do jogo está preparado para receber ações.

As funções *getNode(String name)* e *getNode(Group group, String name)* retornam nó do grafo de cena que possui o nome igual ao parâmetro. Caso não encontre, a função retorna *null*. A comparação entre o parâmetro e o nome do nó é *case-sensitive*. Estas funções existem caso do desenvolvedor do jogo deseje alterar ou consultar o grafo de cena diretamente.

As funções *getTexture*, *getMaterial* e *getAppearance* retornam instâncias de Textura, Material e Appearance respecivamente. Estas instâncias são carregadas com um nome a partir do arquivo Mtl. É via esta função que o desenvolvedor poderá utilizar de recursos já criados em novos nós para o grafo de cena.

3.2.5 A classe KeysManager

A classe *KeysManager* controla as teclas digitadas pelo usuário. Tem a função de verificar se existem métodos atribuídos a cada tecla pressionada e, quando existir, chamar os objetos responsáveis pela ação de acordo com cada tecla.

O diagrama na fig. 3.8 mostra os atributos e métodos da classe *KeysManager*.

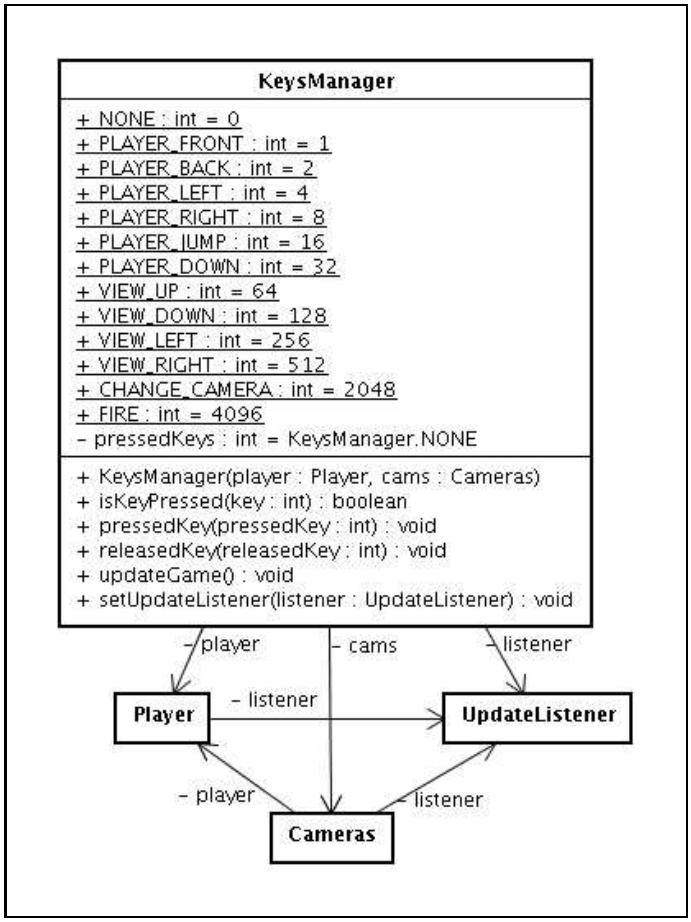


Figura 3.8 – Classe *KeysManager* e suas dependências

Os atributos estáticos são códigos em potência de 2 criando uma mascara de bits e, portanto, registrando em uma única variável do tipo inteiro o estado de cada tecla pressionada pelo jogador. O atributo *pressedKeys* é a variável que mantém o estado das teclas. Os atributos *cams* e *player* são utilizados durante o *update*, onde verifica-se a tecla pressionada e, caso necessário, chama algum método deles.

O atributo *listener* é a classe de eventos implementada pelo desenvolvedor do jogo, ela é invocada ao pressionar ou liberar cada tecla, e é configurada pelo método *setUpdateListener*.

O método *isKeyPressed* verifica se uma determinada tecla está pressionada. Os métodos *pressedKey* e *releasedKey* pressionam e liberam uma tecla no atributo *pressedKeys*. Estes dois métodos são chamados quando a *EngineCanvas* recebe algum evento de tecla pressionada ou liberada da máquina virtual java.

A função *updateGame()* invoca *cams* e *player* para atualizarem seus estados. Esta função é chamada pelo relógio da classe *EngineCanvas*.

3.2.6 A classe Player

Esta classe é o personagem principal do jogo, o personagem que o jogador controla. É responsável por manter e atualizar a posição, ângulo e tamanho, assim como armazenar todo o modelo 3D do personagem. O *Player* é um grupo de nós do grafo de cena da M3G, e, com isso, pode manter o desenho do personagem em seus nós filhos. O *Player* também é responsável por chamar as rotinas de teste de colisão quando necessárias.

No diagrama contido na fig. 3.9 mostra a especificação da classe Player e suas dependencias. O construtor da classe *Player* requer uma instância de *Configuration* e uma de *World*. O objeto da classe *Configuration* será consultado para buscar informações de colisão ativada, raio de colisão, poder de pulo e velocidade do personagem. Estes dados são armazenados nos atributos: *collision*, *collisionRay*, *jump* e *move*. No construtor também é criado uma instância de *CollisionDetection* que será responsável por calcular a colisão entre objetos durante todo o jogo.

O atributo *rotationAngle* guarda o ângulo de rotação em relação a posição inicial do per-

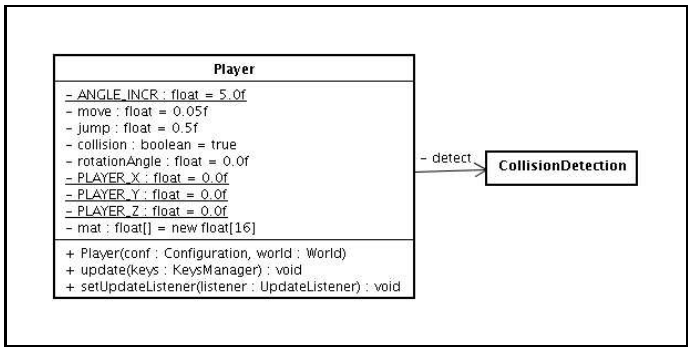


Figura 3.9 – Classe *Player* e suas dependências

sonagem. Este valor é necessário caso o desenvolvedor necessite saber para onde o personagem está virado sem consultar o grafo de cena. O atributo *trans* é o transformador principal, ele faz o personagem andar, virar, subir e descer no grafo de cena.

O atributo *listener* é a classe de eventos implementada pelo desenvolvedor do jogo e será invocada quando o personagem se mover ou atirar. As constantes estáticas *ANGLE_INCR*, *PLAYER_X*, *PLAYER_Y* e *PLAYER_Z*, mantém o ângulo de rotação a cada movimento do personagem, e a posição inicial em x, y e z.

O método *update(KeysManager keys)* movimenta o personagem de acordo com as teclas pressionadas, testando colisão entre os objetos e a colisão do tiro com algum objeto no modelo, caso seja esta a ação.

3.2.7 A classe UpdateListener

Esta é classe que o desenvolvedor de jogos deve implementar caso haja a necessidade de tratar alguma ação nas rotinas de renderização, ou alguma lógica de jogo a adicionar em eventos. Isto permite, por exemplo, que o desenvolvedor de jogos detalhe informações para os jogadores, escrevendo-as diretamente na tela.

No diagrama de classe disponível na fig. 3.10 é possível conhecer todas os eventos disponíveis para o desenvolvedor de jogos. O método *camUpdated(int now)* é invocado quando o jogador troca de câmera, informando no parâmetro qual é a câmera atual. O método *playerMoved(float x, float y, float z)* é invocado quando o jogador se movimenta. No parâmetro estão a posição x, y e z atual do personagem.

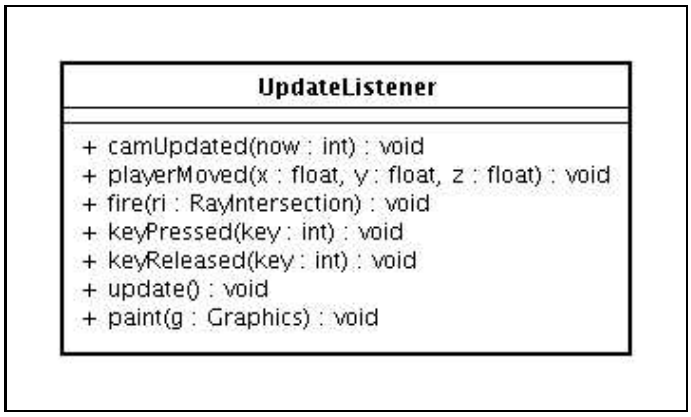


Figura 3.10 – Classe *UpdateListener*

O método *fire(RayIntersection ri)* é invocado quando o jogador atira e acerta algum objeto 3D. O parâmetro é uma classe da M3G onde podem ser encontradas todos os dados relativos a colisão. Os métodos *keyPressed(int key)* e *keyReleased(int key)* são invocados quando o jogador pressiona e solta uma tecla, no parâmetro encontra-se o código da tecla definido como constante pública na classe *KeysManager*.

O método *update()* é invocado antes de atualizar o modelo 3D para visualização do jogador e o método *paint(Graphics g)* é invocado após desenhar o modelo 3D no *display* do celular. Este método permite ao desenvolvedor de jogos adicionar algum objeto bidimensional a visialização.

3.2.8 As classes UpdateSceneTimerTask e Object3DInfo

A classe UpdateSceneTimerTask é responsável pela chamada às rotinas de desenho do motor em um determinado tempo. Seu único método *run()* consiste em chamar o método *update()* do atributo *canvas*.

A Object3DInfo é uma classe que armazena o ponto central de cada objeto tridimensional e o seu nome nos atributos *centerX*, *centerY*, *centerZ* e *name* respectivamente.

3.2.9 O componente Carga de Arquivos

O segundo componente, denominado “Carga de Arquivos”é responsável pela carga do arquivo *Wavefront* Obj e Mtl ou pelo arquivo Mbj, que será especificado na seção 3.8, e suas

conversões para um grafo de cena da M3G. Este componente é chamado pelo *core* caso o desenvolvedor opte por utilizar estas rotinas para carregar o seu mundo. As principais classes pertencentes a este grupo são a *ObjLoader*, que carrega um arquivo .obj (*Wavefront* Obj) e a *MtlLoader* que carrega os arquivos de texturas (*Wavefront* Mtl) e a *MbjLoader* que carrega um arquivo .mbj. A *FileReader* é uma classe especial para facilitar a leitura dos arquivos e a *InconsistentFileException*, no componente "Utilitárias"é uma excessão lançada quando algum erro é detectado no arquivo.

O terceiro componente, denominado "Utilitárias"possui classes que todos os outros componentes, incluindo implementações externas a M3GE, podem fazer uso. São classes que só possuem métodos estáticos para agilizar a programação, como buscas e rotinas para preenchimento de *arrays*.

A demonstração do jogo, não incluída no diagrama, faz parte do componenete "Exemplos", onde existe outras aplicações para mostar o uso do motor de jogos.

3.3 O DESENVOLVIMENTO DO PRIMEIRO JOGO

A M3GE é de fácil uso, de maneira que, para carregar um mundo e permitir que um personagem ande sobre ele, basta criar uma instância da classe *EngineCanvas* passando o endereço do arquivo *Wavefront* e o endereço de um arquivo de propriedades, descrito na seção 3.7. Na fig. 3.11 está descrito o processo de carga de um jogo, sendo que das classes descritas na imagem, somente a classe *MapMidlet* é criada pelo usuário.

O *MIDLet*, que é a classe principal de uma aplicação no J2ME, cria um objeto de *EngineCanvas*. A partir deste ponto, a *engine* carrega todas as suas configurações, carrega o arquivo obj, cria a cena, e retorna um objeto da classe *Canvas*, permitindo que o desenvolvedor configure este *canvas* como o principal da MIDlet e, portanto, renderizando-a na tela para o jogador.

O método *getWorld* que existe na *EngineCanvas* permite ao desenvolvedor do jogo modificar ou adicionar nós de cena em seu grafo. Para buscar um determinado nó foi implementada a função *getNode*, que tem como parâmetro o nome do nó e que busca em todo o grafo de cena

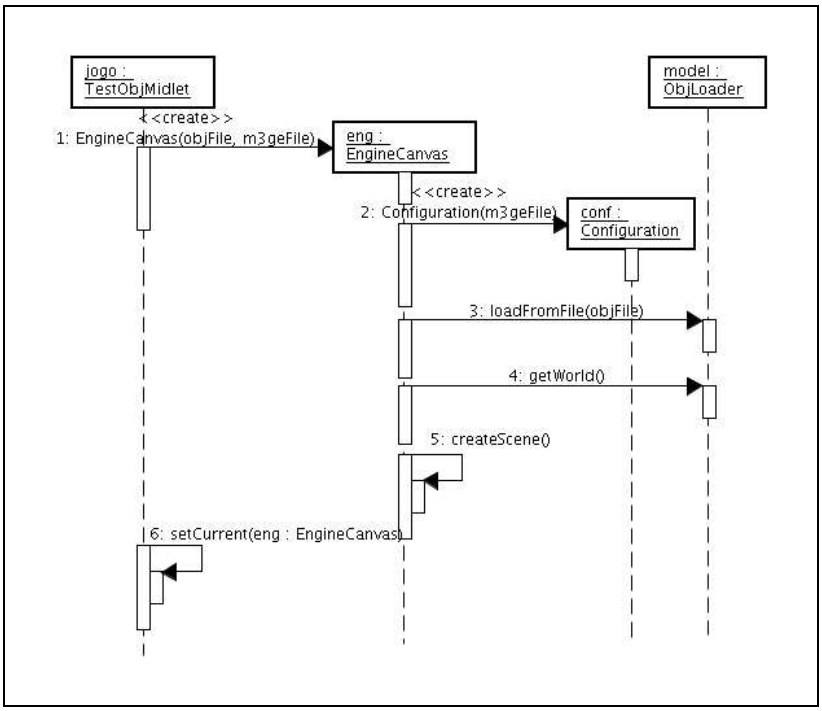


Figura 3.11 – Criando um jogo

um nó com o mesmo nome, retornando-o. Métodos semelhates a este são os *getTexture*, o *getMaterial* e o *getAppearance*, que consultam o grafo retornando objetos de *Texture2D*, *Material* e *Appearance* respectivamente.

A classe *EngineCanvas* herda características da classe *Canvas*, portanto ela mesma desenha o jogo. É importante lembrar que, em alguns celulares, é necessário chamar o método *exit* ao fechar a aplicação. Este método cancela a *thread UpdateSceneTimerTask* que faz a atualização da tela num período de tempo. Caso este método não seja invocado, a *thread* continua executando até que o celular seja desligado.

Para atuar sobre eventos acontecidos dentro da Engine, deve-se utilizar a classe *UpdateListener* como será explicado na seção 3.5.

3.4 CARREGANDO UM ARQUIVO WAVEFRONT

Carregar um arquivo *Wavefront*, definido na seção 2.7 e citado como exemplo no quadro 3.1, onde desenha um triângulo em duas dimensões, pode ser simples, mas como o ambiente é executado em dispositivos limitados, a leitura do arquivo e a montagem do grafo de cena não podem exigir muita memória e o processamento. Para agilizar a leitura do arquivo, é

utilizado um *array* de *byte* com 1024 posições como buffer, assim, ao invés de ler de *byte* em *byte*, o motor lerá por *kilobyte*.

Para que a importação de um arquivo Obj seja viável, a sua definição foi extendida adicionando as seguintes características, algumas delas limitadas pela própria M3G:

a) vértices, vetores normais, e vetores de textura devem ser igualmente sequenciados, e devem estar em mesmo número, ou seja, o vetor normal 4 e o vetor textura 4 devem referenciar o mesmo vértice 4;
b) os vetores normais não declarados no arquivo terão o valor 0, 0 e 1 para os eixos X, Y e Z respectivamente. Estes valores foram escolhidos pois normalmente o jogador inicia o jogo com a visão voltada para o lado negativo do eixo Z, e assim, estes valores tornam o objeto 3D visível a este jogador;
c) ao ler as faces, somente o valor dos vértices é considerado, visto que os vetores de textura e normais devem estar na mesma posição;
d) os vértices devem obrigatoriamente ser 3D, assim como os vetores normais;
e) as faces devem ser triangulares, ou seja, devem ter exatamente três pontos de referência;
f) para agilizar a leitura todos os dados de textura e cores devem ser colocados num único arquivo MTL;
g) as imagens de textura devem ter seu tamanho em uma potência de 2, ou seja, 2, 4, 8, 16, 32, 64, etc;
h) o grupo com o nome PLAYER é o objeto 3D que é modificado de acordo com as ações do jogador (Andar, virar, pular, etc).

A carga completa do grafo de cena é feita em três leituras do arquivo Obj pela classe *ObjLoader*, como pode ser vista na fig. 3.12. A leitura inicial conta quantos vértices, vetores normais e texturas estão sendo declarados. Esta contagem é importante para criar o *array* de vetores para segunda leitura. Foram cogitadas algumas outras possibilidades, como a utilização de estruturas de dados como a classe *Vector* do J2ME, mas esta estrutura recria todo o seu array interno a cada 16 posições, ou seja, uma leitura muito grande torna-se inviável recriar arrays de

```
v -1 -2 -1
v 1 -2 -1
v 0 2 -1

vn 0 0 1
vn 0 0 1
vn 0 -1 0

g Pyramid
f 1 2 3
```

Quadro 3.1 – Exemplo de um arquivo Wavefront desenhando um triângulo 2D

16 em 16 elementos e copiar todos os valores para o *array* maior. Outras estruturas como listas encadeadas também foram cogitadas, mas mostraram-se inviáveis, pois a cada elemento lido é necessário um ponteiro de 4 *bytes* para o próximo elemento, além de que elas não armazenam tipos primitivos, somente objetos criando mais 4 *bytes* a cada elemento. Dependendo do método de leitura escolhido, as referências ocupariam mais memória do que os próprios dados. Por outro lado, a leitura de um arquivo não é tão lenta e não ocupa uma memória muito alta.

Após a primeira leitura estar completa, quatro *arrays* são criados pelo método *createArrays* e armazenados na classe *ObjLoader*: vértices, vetores normais, vetores de textura e faces. Sendo que os *arrays* de vértices, vetores normais e faces tem seu tamanho multiplicado por três, pois armazenarão as suas estruturas sequencialmente, ou seja, a posição 0, 1 e 2 do *array* de vértices representa os valores dos eixos X, Y e Z do primeiro vértice. Da mesma maneira, o tamanho do *array* de texturas é multiplicado por dois. O *array* de vetores normais é inicializado com a sequência de valores 0, 0, 1.

Na segunda leitura do arquivo, efetuada pelo método *loadVertexNormalTexture* da classe *ObjLoader*, são considerados todos os pontos sequenciais: vértices, vetores normais, vetores de textura e os dados dos arquivos MTL. Os valores são lidos como *float* sem nenhuma transformação e armazenados em seus respectivos *arrays*. A classe *ObjLoader* lança uma exceção chamada InconsistentFileException caso algum problema ocorra no processo. Os dados do arquivo Mtl são importados por uma instância de *MtlLoader* em uma única leitura, com

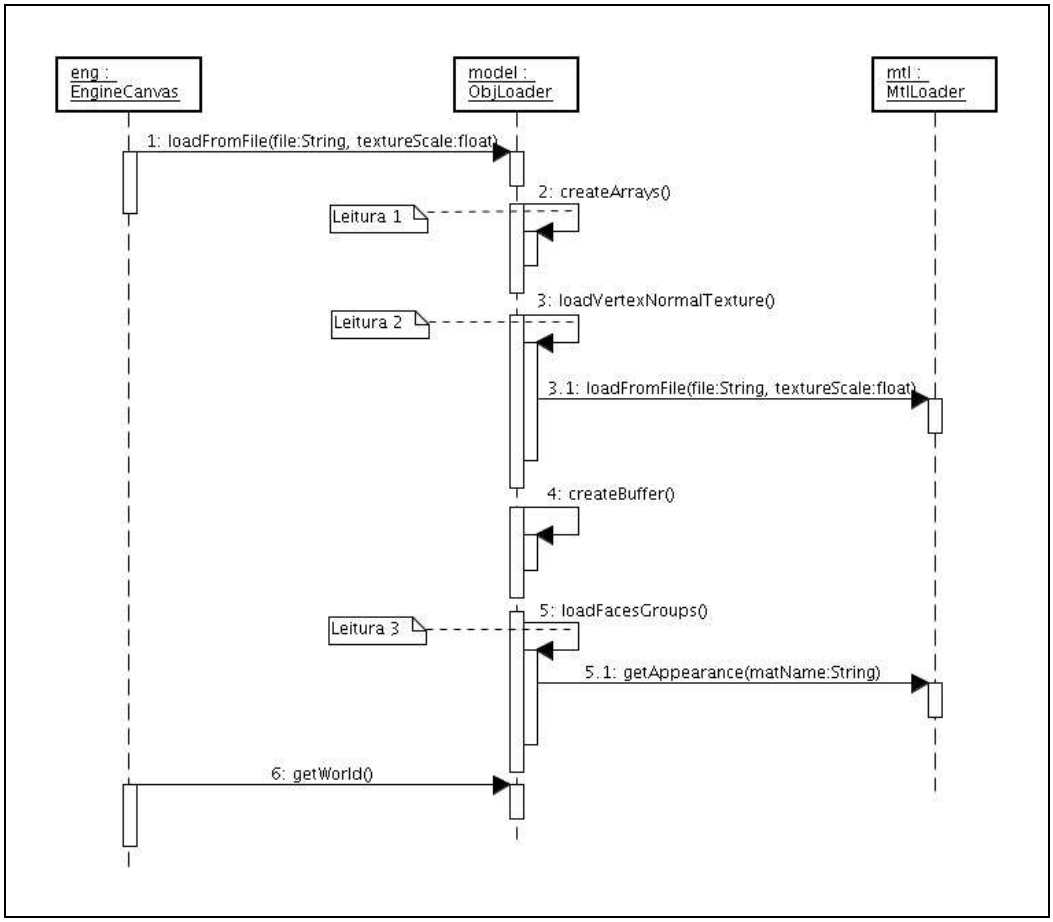


Figura 3.12 – Carregando arquivo Obj e Mtl

cuidado para converter as cores, que estão numa faixa entre 0 e 1, para 0 a 255 formando cores RGB. As informações de material e textura são armazenadas dentro de duas listas indexadas por seus nomes. As texturas são armazenadas com a opção "repetir em todos os lados"ativada.

Ao ler o arquivo Mtl, o valor de *Shininess* (instrução Ns) é limitado, pela própria M3G, a estar entre 0 e 128. A definição do arquivo Mtl apresenta esta informação como um campo *float*. Visto que alguns geradores de arquivos MTL informam nesta posição o valor fixo 80, o qual representa o mesmo valor 80 na M3G, não é efetuada nenhuma conversão na importação. Caso o valor seja menor que 0 ou maior que 128 ele é ignorado pela rotina de importação.

Após a segunda leitura se completar, a função *createBuffer()* é invocada. Esta função, primeiramente, configura valores padrão para os vetores de textura não informados. Um vetor de textura é composto por dois elementos X e Y e seus valores padrão são calculados a partir dos valores originais de seus vértices - 0.5 para X e -1.5 para o Y.

O principal objetivo da função *createBuffer()* é transformar todos os *arrays* de *float* lidos anteriormente em *arrays* de *bytes*. Isto é feito para diminuir o espaço alocado na memória e agilizar o processamento durante a execução do jogo. O processo é igual para os vetores normais e para os vértices, onde se busca os valores máximo e mínimo, declarados no arquivo, para cada *array* e converte-se o mínimo para -127 e o máximo para 128. Os demais são recalculados em uma regra de três composta baseado nos valores máximo e mínimo. Com os vetores de textura, o processo é semelhante, mas são utilizados *arrays* de tipo *short*, armazenando valores entre 0 e 255.

Ao ler e transformar os vetores de textura para valores entre 0 e 255, é necessário também, inverter o segundo float lido do arquivo. Isto ocorre porque no arquivo Obj, o segundo valor da instrução vt, valor do eixo Y, aumenta de baixo para cima como qualquer plano cartesiano, mas na M3G este valor aumenta de cima para baixo, como se o eixo Y estivesse invertido (DAVISON, 2004).

Depois de transformados, todos os vetores são passados para três instâncias de *VertexArray*. As três instâncias são unidas em um *VertexBuffer* criando uma única fonte e objeto para todo o modelo 3D.

Os valores convertidos em *byte* e *short* são novamente convertidos para uma faixa de valores entre -1 e 1 para os *arrays* de *byte* e 0 e 1 para os *arrays* de *short* durante a renderização das imagens pela M3G (DAVISON, 2004). Portanto, ao repassar as três instâncias de *VertexArray* para o *VertexBuffer* é indicado nos parâmetros, os valores necessários para que os *arrays* definidos anteriormente possam ser convertidos. Os parâmetros são chamados de *PBIAS* e *SCALE*.

Agora que todos os valores já estão otimizados, inicia-se a terceira leitura do arquivo. As faces, grupos e materiais são considerados e integrados aos *arrays* da leitura anterior. A cada grupo lido (instrução g) é criado uma instância de *Mesh*, e se houver um *Material* (instrução usemtl) este é consultado no objeto que leu o arquivo MTL e configurado para o *Mesh* atual. As faces são lidas em um *array* de tipo *short* sequencialmente, mas somente os índices das faces do objeto em questão são repassadas ao *Mesh*, criando vários *arrays* de tipo *short*, um para cada objeto 3D. Estes índices consultarão os *arrays* montados na segunda leitura para determinar os pontos e vetores. No quadro 3.2 os vértices, vetores normais e vetores textura estão lado a lado para exemplificar a leitura dos índices das faces.

Após a leitura de cada objeto 3D, antes de criar o objeto *Mesh*, calcula-se o ponto central pela média de seus extremos nos três eixos. Utiliza-se este ponto para determinar o local exato para o cálculo da posição de uma câmera quando relativa ao *Player*.

Quando o motor de jogos terminar de ler o arquivo Obj, o mundo, uma instância da classe *World*, está criado e devidamente montado. No exemplo criado para validar este trabalho, a carga deste arquivo demora 17 segundos em dispositivo real e 15 no simulador.

No quadro 3.2, está um arquivo *Wavefront* Obj gerado pelo editor gráfico AoI (EASTMEN, 2005), que desenha um Cubo exatamente no centro do sistema de coordenadas. O cubo está com uma textura chamada metal, e é carregada em um arquivo *Wavefront* Mtl que está no quadro 3.3.

A fig. 3.13 mostra a imagem de textura que existe no arquivo map51.jpg. Esta é a texura utilizada para renderizar o Cubo.

```
# Produced by Art of Illusion 2.0,
# Sat May 14 20:15:51 GMT-03:00 2005

mtllib map5.mtl
g Cubo
usemtl metal

v -1 0 1    | vt -1.5 -1.5 | vn -0.57735 -0.57735 0.57735
v 1 0 1     | vt 0.5 -1.5  | vn 0.57735 -0.57735 0.57735
v 1 0 -1    | vt 0.5 -1.5  | vn 0.57735 -0.57735 -0.57735
v -1 0 -1   | vt -1.5 -1.5 | vn -0.57735 -0.57735 -0.57735
v -1 2 1    | vt -1.5 0.5  | vn -0.57735 0.57735 0.57735
v 1 2 1     | vt 0.5 0.5   | vn 0.57735 0.57735 0.57735
v 1 2 -1    | vt 0.5 0.5   | vn 0.57735 0.57735 -0.57735
v -1 2 -1   | vt -1.5 0.5  | vn -0.57735 0.57735 -0.57735
v 0 1 1     | vt -0.5 -0.5 | vn 0 0 1
v 1 1 0     | vt 0.5 -0.5  | vn 1 0 0
v 0 1 -1    | vt -0.5 -0.5 | vn 0 0 -1
v -1 1 0    | vt -1.5 -0.5 | vn -1 0 0
v 0 0 0     | vt -0.5 -1.5 | vn 0 -1 0
v 0 2 0     | vt -0.5 0.5  | vn 0 1 0

f 2/2/2 1/1/1 13/13/13 | f 4/4/4 1/1/1 12/12/12
f 3/3/3 2/2/2 13/13/13 | f 1/1/1 5/5/5 12/12/12
f 4/4/4 3/3/3 13/13/13 | f 5/5/5 8/8/8 12/12/12
f 1/1/1 4/4/4 13/13/13 | f 8/8/8 4/4/4 12/12/12
f 2/2/2 3/3/3 10/10/10 | f 5/5/5 6/6/6 14/14/14
f 3/3/3 7/7/7 10/10/10 | f 6/6/6 7/7/7 14/14/14
f 7/7/7 6/6/6 10/10/10 | f 7/7/7 8/8/8 14/14/14
f 6/6/6 2/2/2 10/10/10 | f 8/8/8 5/5/5 14/14/14
f 1/1/1 2/2/2 9/9/9    | f 3/3/3 4/4/4 11/11/11
f 2/2/2 6/6/6 9/9/9    | f 4/4/4 8/8/8 11/11/11
f 6/6/6 5/5/5 9/9/9    | f 8/8/8 7/7/7 11/11/11
f 5/5/5 1/1/1 9/9/9    | f 7/7/7 3/3/3 11/11/11
```

Quadro 3.2 – Exemplo de um arquivo Wavefront Obj desenhando um Cubo

```
# Produced by Art of Illusion 2.0,
# Sat May 14 20:15:50 GMT-03:00 2005
newmtl metal
Kd 1 1 1
map_Kd map51.jpg
Ks 0 0 0
Ka 0 0 0
illum 1
```

Quadro 3.3 – Exemplo de um arquivo Wavefront Mtl com textura

Figura 3.13 – Textura map51.jpg

3.5 A ESTRUTURA DE UM JOGO

Ao contrário das outras aplicações, um jogo é feito com duas unidades de processamento distintas, como pode ser visto na fig. 3.14. Uma delas atua sobre a camada de modelo do padrão de projeto *Model-View-Controller* (MVC) (SUN MICROSYSTEMS, 2004c) enquanto a outra atua sobre as comadas de controle e visão.

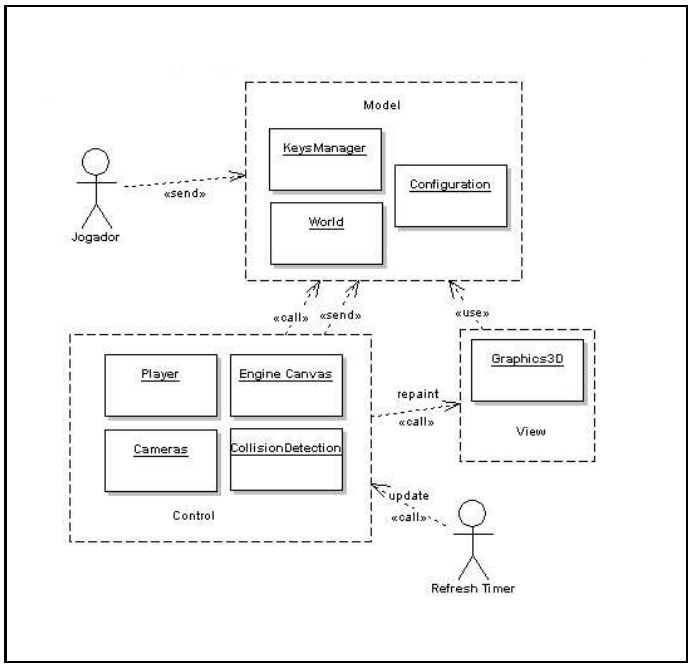


Figura 3.14 – Comparando M3GE com o padrão de projeto MVC

Na implmentação deste trabalho, o modelo é um atributo da classe *EngineCanvas*, a instância da classe *World*, e as classes *KeysManager* e *Configuration*. O controle são as instâncias da classe *Cameras*, da classe *Player*, da classe *EngineCanvas* e da classe *Collision-Detection*. Estas quatro instâncias atuam sobre o modelo em um processo (*Thread*) diferente

do processo disparado ao pressionar uma tecla. A visão é constituída da classe *Graphics3D*.

Em um jogo, os eventos de tecla pressionada e tecla liberada não invocam a atualização da tela. Quando ocorrem estes dois eventos, os valores das teclas são armazenados na instância de *KeysManager*, onde um único atributo pode controlar todas as teclas, e indicar, num determinado tempo, quais estão pressionadas e quais não estão. A *Thread* principal da aplicação só controla a montagem da mensagem ao *KeysManager* contando quais teclas foram pressionadas.

Ao criar toda a cena, a *EngineCanvas* cria também uma instância de *UpdateSceneTimerTask* e adiciona-a em uma agenda de execuções com o tempo entre as execuções determinado no arquivo de configurações. Esta classe será responsável por invocar as rotinas de atualização de modelo e repintura da tela em intervalos de tempo predefinidos criando um **ciclo de processamento**.

Como pode ser visto na fig. 3.15, quando a *UpdateSceneTimerTask* é ativada, ela chama o método updateScene da classe *EngineCanvas*, que, por sua vez, invoca o método updateGame na *KeysManager*. Este método analisa quais teclas estarão pressionadas e invoca a instância de *Player*, caso alguma tecla de movimentação esteja pressionada, e a instância de *Cameras*, caso a tecla de troca de câmeras esteja pressionada.

É neste ponto que todas as teclas pressionadas são computadas. Ou seja, só neste momento os modelos são alterados, o personagem anda, vira, pula, ou esbarra na parede. Isto evita que várias ações sejam disparadas ao mesmo tempo, o que ocorre, por exemplo, quando o jogador segura uma tecla pressionada. Evitar que várias ações sejam disparadas ao mesmo tempo é sinônimo de mais velocidade para o jogo. Uma taxa de atualização dos gráficos de 50 milisegundos, que corresponde a um limite de 20 frames por segundo, não influi em nada a jogabilidade e os gráficos do jogo.

O desenvolvedor de jogos pode usufruir das rotinas de atualização usando a classe *UpdateListener*. Com eventos ele pode aumentar a iteratividade do jogo e controlar a sua lógica de enredo, como pode ser visto no diagrama de classe mostrado na fig. 3.16. No diagrama, somente as classes *TestObjMidlet* e *GameControl* devem ser construídas pelo desenvolvedor de jogos. A *EngineCanvas*, caso possua uma instância válida de *UpdateListener*, invoca os

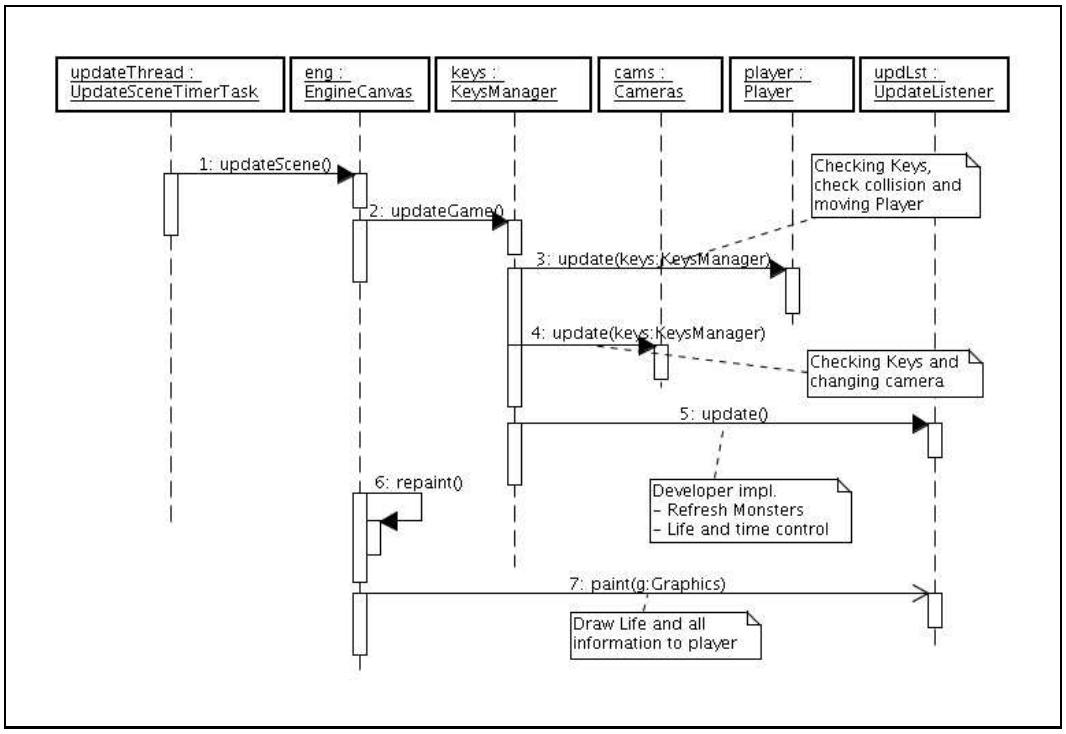


Figura 3.15 – Ciclo principal do jogo

métodos específicos de acordo com cada ação do personagem, permitindo assim que o *game designer* controle o seu enredo.

O método *update* é invocado antes de redesenhar o modelo na tela e é utilizado para fazer modificações no modelo, como animações, reações dos personagens e etc. O método *paint* é invocado após a M3G redesenhar a tela, permitindo que o *game designer* utilize rotinas 2D para levar informações ao jogador, como, por exemplo, a vida, o número de balas na sua arma, câmera utilizada e etc.

Os métodos *keyPressed* e *keyReleased* são acionados quando o jogador pressiona e solta cada tecla. O método *camUpdated* é invocado quando o jogador troca de câmera, muito útil para o *game designer* trocar o *layout* e as informações que disponibiliza para o jogador. O método *playerMoved* é chamado quando o personagem anda, sendo que, operações de rotação como virar à esquerda e virar à direita não são consideradas.

Por fim, o método *fire* é invocado quando o jogador pressiona a tecla de disparo de um tiro e acerta algum objeto 3D. Todas as informações pertinentes ao cálculo de colisão estão na instância de *RayIntersection*, classe da M3G, que está no parâmetro da função.

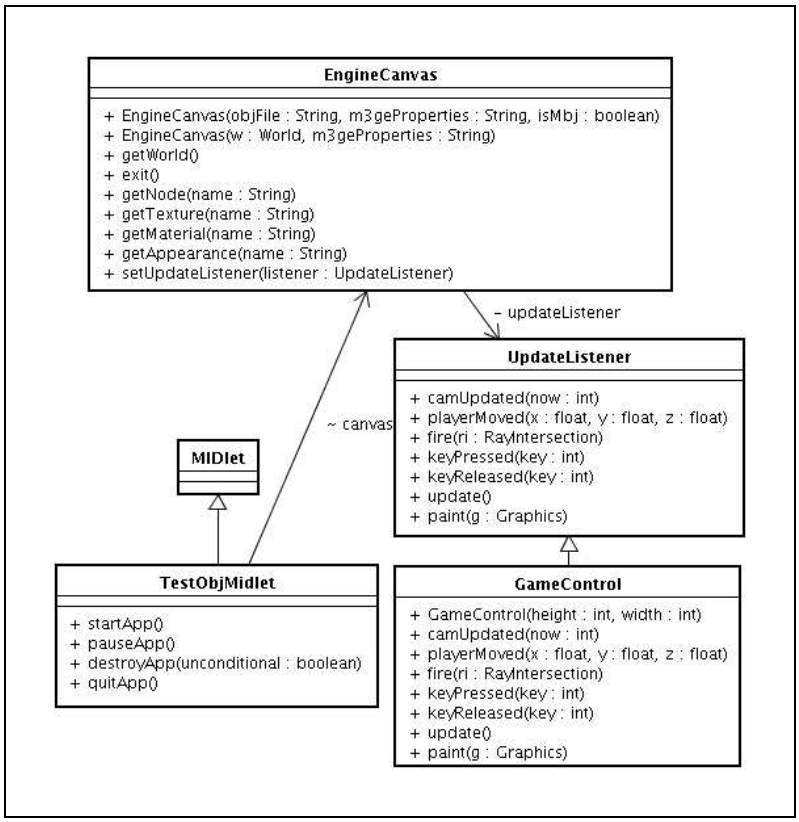


Figura 3.16 – Diagrama de classes de um jogo simples

3.6 COMO FUNCIONA O MOVIMENTO DE UM PERSONAGEM

Um personagem nada mais é do que um grupo de nós do grafo de cena. Este grupo possui, como seus filhos, uma ou mais câmeras, e o nó lido do arquivo *Wavefront* com o nome PLAYER. É primordial que este nó tenha o nome PLAYER em maiúsculo, caso não exista, não existirá nenhuma imagem para o personagem.

O movimento do personagem é totalmente identificado no arquivo de configurações. Sua velocidade e força de pulo são determinantes para um jogo com iteratividade. Estes valores variam de jogo para jogo e de acordo com o tamanho do mapa, pois são valores em pontos do modelo. Em outras palavras, ter o mundo do tamanho 1000 e andar a uma velocidade de 0.5 é bem diferente de ter o mundo em tamanho 10 e a velocidade em 0.5. No segundo caso, o personagem chega muito mais rápido ao seu destino.

O ângulo de rotação, utilizado nas teclas de movimentação da visão do personagem, é sempre 5º, sejam sobre o eixo X ou Y.

Toda a movimentação sobre o personagem não altera o modelo. Isto é possível devido a

classe *Transform* da M3G, que mantém, internamente, uma matriz de ordem 4, que é multiplicada por todos os pontos de um determinado grupo ou nó ao renderizar a sua estrutura. A classe *Transform* possui os métodos *postTranslate* e *postRotate* que são utilizados para movimentar o personagem.

Toda a câmera, mantida em um grafo de cena, inicia na posição (0,0,0) e virada para a posição (0,0,-1), como pode ser visto na fig. 3.17. Isso significa que para mover o personagem para frente, basta decrementar a velocidade sobre o valor de Z, e para mover para trás, basta incrementar o valor da velocidade ao valor do eixo Z. Para mover para esquerda e direita, decrementa-se e incrementa-se o valor da velocidade ao eixo X respectivamente. Mover para baixo e para cima não é diferente, basta diminuir e aumentar o valor de Y.

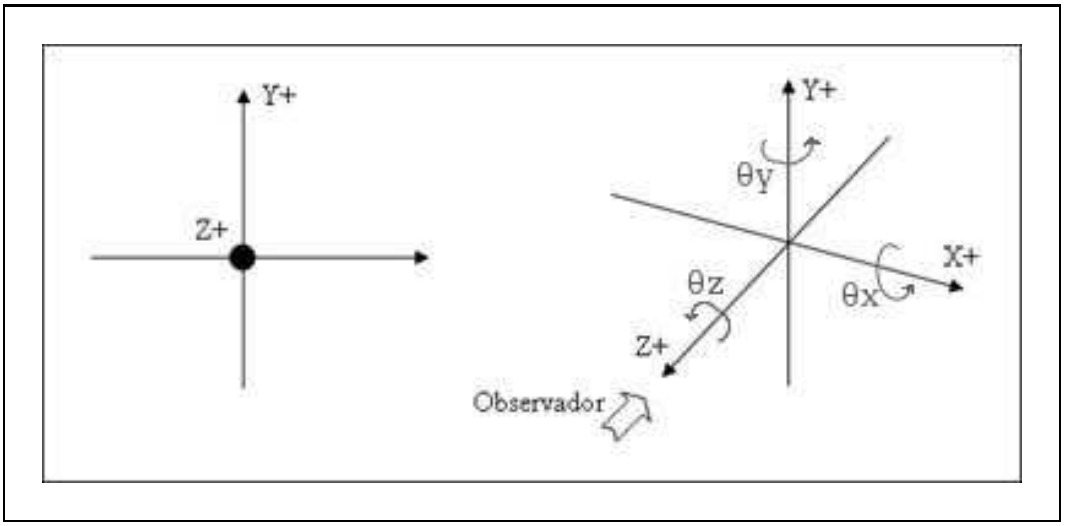


Figura 3.17 – Visão de uma câmera

Rotacionar para a esquerda significa girar o objeto 3D 5 graus a mais sobre o eixo Y, para a direita, 5 graus a menos. Para cima e para baixo são 5 graus a mais e a menos em relação ao eixo X. Na fig. 3.17 as rotações podem ser vistas pelos ângulos Θx, Θy e Θz.

3.7 O ARQUIVO DE CONFIGURAÇÕES

O arquivo de configurações é um arquivo que mantém todas as informações necessárias para que o motor de jogos possa estabelecer seu estágio inicial e o modo de iteração com o usuário. Possui o formato de um arquivo texto de propriedades, ou seja, uma única expressão $<chave>=<valor>$ por linha representa uma configuração. A classe *Configuration* carrega e

mantem consigo todas as informações.

As chaves estipuladas pelo motor, são *case-sensitive* e o *game designer* deve respeitar os tipos de dados requeridos. As chaves são divididas em oito grupos: de movimentação do personagem, de movimentação de visão do personagem, de configuração de câmeras no cenário, de detecção de colisão, de taxa de atualização, de velocidade e poder de pulo, de troca de câmera e ação e de tratamento de texturas. O caractere # indica uma linha de comentário, e, portanto, deve-se ignorar todas as informações lidas entre ele e uma quebra de linha ou o final do arquivo.

3.7.1 Grupo de movimentação do personagem

As chaves deste grupo indicam qual tecla deve ser utilizada para que o personagem se movimente no jogo. As chaves são: PLAYER_FRONT, PLAYER_BACK, PLAYER_LEFT, PLAYER_RIGHT, PLAYER_JUMP e PLAYER_DOWN que indicam para frente, para trás, para a esquerda, para a direita, pular e agachar respectivamente. Os valores para essas chaves devem ser inteiros e representa o código das teclas do dispositivo que o jogo irá executar ou será emulado.

3.7.2 Grupo de movimentação de visão do personagem

As chaves pertencentes a este grupo, movimentam o olho ou a cabeça do personagem. As chaves deste grupo são: VIEW_UP, VIEW_DOWN, VIEW_LEFT, VIEW_RIGHT, que representam olhar para cima, para baixo, para a esquerda e para a direita respectivamente. Assim como o item anterior, os valores para estas chaves também são inteiros e devem ser os códigos das teclas, que podem variar entre modelos de uma mesma marca.

Vale ressaltar a diferença entre movimentar a visão do personagem e movimentar o personagem que, ao invés de uma rotação, que ocorre na movimentação da visão, será utilizada uma translação.

3.7.3 Grupo de configuração de câmeras no cenário

Grupo composto por chaves que permitem indicar ao motor onde e como posicionar cada câmera do jogo. O número de câmeras é limitado somente pelas características físicas de cada

dispositivo e é determinado pelo números de câmeras declaradas no arquivo de configurações. É necessário haver pelo menos uma câmera para que o jogo possa executar.

Uma câmera existe quando qualquer uma das seguintes chaves está configurada: CAMERA<num>_X, CAMERA<num>_Y, CAMERA<num>_Z, CAMERA<num>_FOVY, CAMERA<num>_NEAR, CAMERA<num>_FAR e CAMERA<num>_RELATIVE_TO. Elas representam a posição no eixo X, Y e Z, o campo de visão em graus, o ponto mais perto e o ponto mais longe para visualizar os obstáculos e se a posição da câmera é relativa com o personagem ou se é relativa ao mundo. O valor de <num>é um número sequencial por câmera, sem quebras, determinando a ordem das câmeras para o motor.

A chave CAMERA<num>_RELATIVE_TO indica se a câmera será posicionada em relação ao personagem, e assim deve receber o valor de PLAYER, ou se deve ser posicionada em relação ao mundo e, portanto, receber o valor de WORLD. Caso seja posicionada em relação ao personagem, a câmera se moverá juntamente com ele, caso não, ela será estática e nunca vai mudar de posição.

3.7.4 Grupo de detecção de colisão

Grupo composto pelas chaves COLISION_DETECTION e COLISION_PLAYER_RAY. Identificam se as rotinas de detecção de colisão estão ativas ou não e qual o raio de colisão que deve ser considerado para o personagem respectivamente. Isto já permite que a M3GE implemente uma função básica de colisão.

O valor do raio é utilizado tanto para a distância entre a posição atual do personagem e ponto de colisão quanto para determinar e tamanho da distância para os testes laterais de colisão. Como será visto na seção 3.9.

Se a chave COLISION_DETECTION não for declarada igual a 1, a detecção estará desativada. O valor padrão para COLISION_PLAYER_RAY é um ponto flutuante de 0.5.

3.7.5 Grupo de velocidade e poder de pulo

No arquivo também podem ser configuradas a velocidade que o personagem anda e o poder de seu pulo. Como não foi implementado suporte a simulação física na M3GE, o poder de pulo fica sendo a taxa de movimentação para o eixo Y. As chaves VELOCITY e JUMP_POWER recebem valores de ponto flutuante, estarão indicando a velocidade e o poder de pulo respectivamente. Seus valores padrão são 0.05.

3.7.6 Grupo de taxa de atualização

Pertencente a este grupo só existe a palavra chave REFRESH_RATE. Esta, indica qual o intervalo de tempo, em milisegundos, que o motor deve recalcular e repintar a cena a partir da câmera atual. De acordo com o tamanho, a quantidade de objetos e a lógica adicionada pelo desenvolvedor de jogos, este valor pode ser maior ou menor, pois quanto maior o tempo de processamento menor será a taxa de atualização da tela. O valor desta chave deve ser um inteiro e o padrão é 50 milisegundos.

3.7.7 Grupo de troca de câmera e ação

Grupo das chaves especiais: CHANGE_CAMERA e FIRE. Elas representam os valores das teclas, em dados inteiros, para mudar de câmera do jogo e para o personagem atirar. Caso não sejam declaradas, estas funcionalidades estarão desabilitadas no jogo.

3.7.8 Grupo de tratamento de texturas

Este grupo foi reservado a uma série de configurações que podem ser feitas para as texturas, porém a única implementada é a de redimensionamento de textura. As demais tratariam de posicionamento em relação ao objeto 3D, tipo de visualização e algumas reações as luzes.

Algumas vezes o tamanho renderizado do mundo é muito maior que o tamanho da textura que está sendo utilizada, ou vice-versa. Para evitar que o desenvolvedor tenha que aumentar ou diminuir o tamanho da imagem, ocupando mais espaço nos pequenos discos dos dispositivos, a *engine* proporciona um meio para que a textura seja redimensionada na sua carga.

A chave TEXTURE_SCALE informa um valor inteiro que será usado para redimensionar a imagem em todas as direções. A imagem, para poder ser importada pela M3G, deve ser quadrada, portanto, a escala pode ter um único valor para redimensionar nos dois lados.

3.7.9 Exemplificando

Um arquivo M3GE de configuração ficaria semelhante ao existente no quadro 3.4 e o a visão do jogador seria como a fig. 3.18.

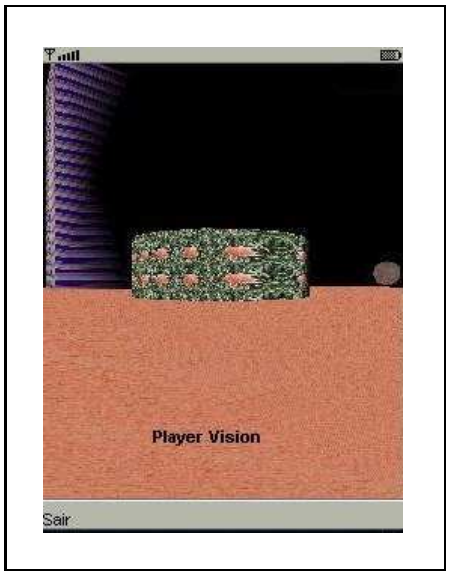


Figura 3.18 – Visualização do exemplo do arquivo de configurações

O exemplo determina as teclas de movimentação do personagem e da sua visão para o celular Siemens CX65. Uma única câmera foi adicionada ao mundo e está posicionada exatamente no centro do Objeto 3D denominado PLAYER. A detecção de colisão está habilitada atuando com um raio de 0.05 pontos, que é igual a velocidade e ao poder do pulo do personagem. Todas as texturas estão sendo redimensionadas em 40 pontos negativos, o que indica que as imagens estão maiores do que deveriam ser. A taxa de atualização é de 50 milisegundos ou 20 frames por segundo.

3.8 UMA ALTERNATIVA PARA O ARQUIVO WAVEFRONT

O arquivo *Wavefront* não foi especificado para ser utilizado por dispositivos móveis e, por consequência, não possui as otimizações necessárias para um bom funcionamento. Carregar um arquivo OBJ pequeno, com 2000 linhas, em um telefone Siemens CX65 demora 17

```
# up and down
PLAYER_FRONT= -59
PLAYER_BACK= -60
# Number 7 and 8
PLAYER_LEFT= 55
PLAYER_RIGHT= 56
# Left and right Keys
PLAYER_JUMP= 51
PLAYER_DOWN= 54
# NONE
VIEW_UP= 49
VIEW_DOWN= 52
# left
VIEW_LEFT= -61
# right
VIEW_RIGHT= -62
# center
FIRE=-26
```

```
REFRESH_RATE=50
TEXTURE_SCALE=-40

VELOCITY=0.05
JUMP_POWER=0.05
COLISION_DETECTION=1
COLISION_PLAYER_RAY=0.05

# Changing câmera key 9
CHANGE_CAMERA=57
CAMERA0_RELATIVE_TO=PLAYER
CAMERA0_X=0
CAMERA0_Y=0.00
CAMERA0_Z=0
CAMERA0_FOVY=70.0
CAMERA0_NEAR=0.01
CAMERA0_FAR=50.0
```

Quadro 3.4 – Exemplo de um arquivo de configuração M3GE

segundos, desde a carga até a apresentação. Os celulares Siemens possuem otimizações para re-leituras de arquivos, o que facilita muito a utilização de algoritmos como o implementado na seção 3.4, porém outras marcas podem não se comportar da mesma maneira, o que aumentaria significativamente o tempo de carga. Este tempo precisa ser melhorado, afinal, além das diferentes implementações, alguns jogos possuem muito mais dados do que este exemplo.

Para aumentar a velocidade com a carga do jogo, o presente trabalho propõe a especificação de um arquivo semelhante ao *Wavefront*, mas com algumas limitações. Nesta especificação, os tipos de dados já devem estar convertidos à faixa de valores em *byte*, como comentado na seção 3.4, com o cálculo do ponto central já realizado e com a informação sobre o tamanho dos *arrays* a serem montados. O arquivo recebeu a extensão MBJ, que significa, *Mobile Object File*, e é descrito abaixo.

O arquivo deve obrigatoriamente iniciar com 4 informações primordiais, o tamanho dos arrays a serem criados. As instruções nf, nv, nvt e nvn, determinam o tamanho necessário para os *arrays* de faces, de vértices, de vetores textura e vetores normais respectivamente. Os valores devem ser do tipo inteiro.

Após, deve vir a declaração para importar o arquivo Mtl de texturas e cores, juntamente com os vértices, vetores normais e vetores de textura em sequência. Como pode ser visto no quadro 3.5, o tipo de informação vértice, vetor de textura ou vetor normal é identificado uma única vez, e todos os pontos referentes aquele tipo devem ser listados em sequência. Isto evita o uso excessivo de comparadores de variáveis *String* para determinar que tipo de dado está sendo lido, e acrescentou um ganho de até dois segundos. Os valores dos vértices e vetores devem já estar em *byte* e na faixa especificada para cada tipo de informação.

Em sequência devem vir todas as declarações de grupos, usos de materiais e faces. Nesta definição de arquivo, todos os índices iniciam de 0 e não de 1 como ocorre no arquivo Obj. Só são permitidas faces triangulares, e o valor dos índices para vértices e vetores deve, obrigatoriamente, ser o mesmo. A declaração de grupo (g) mudou para armazenar o ponto central de cada objeto, evitando assim o cálculo na leitura. Para declarar um grupo, utiliza-se a sintaxe: g *float float float* nome, onde os três *floats* representam o ponto central em X, Y e Z respectivamente. Os vérticies e vetores normais possuem três pontos, os vetores de textura somente dois pontos.

O mesmo cubo apresentado no quadro 3.5 está no quadro 3.5 convertido com o utilitário de conversão de arquivo Obj para Mbj criado neste trabalho. Este programa foi construído utilizando as rotinas de leitura de arquivos Obj já existentes. A diferença é que, ao invés de retornar uma instância de *World* no final do processamento, é retornado um conjunto de *arrays*. É nestes *arrays* que estão os valores convertidos da leitura e, portanto, basta regravá-los em disco no formato especificado nesta seção.

Para ler um arquivo Mbj dentro do dispositivo, foi criada uma nova classe chamada *MbjLoader*, muito semelhante a *ObjLoader*, discutida na seção 3.4, mas sem fazer nenhum cálculo ou adaptação nos dados encontrados no arquivo. A leitura é mais rápida que a do formato anterior, principalmente, nos celulares que não implementam nativamente o *cache* de arquivos apresentado no início desta seção, evitando o acesso a memória ROM a partir da primeira leitura. Para carregar um modelo pelo arquivo *Wavefront* OBJ são necessárias três leituras, o que pode triplicar o tempo de carga do jogo se o celular não implementar este *cache*.

Mesmo com o *cache* de arquivos, a diferença entre carregar um Wavefront OBJ e um

```
nf 2184
nv 1140
nvt 760
nvn 1140
mtllib map5.mtl
```

v	vt	vn
-12 -3 5	119 136	-74 -74 73
5 -3 5	136 136	73 -74 73
5 -3 -12	136 136	73 -74 -74
-12 -3 -12	119 136	-74 -74 -74
-12 14 5	119 119	-74 73 73
5 14 5	136 119	73 73 73
5 14 -12	136 119	73 73 -74
-12 14 -12	119 119	-74 73 -74
-3 5 5	128 127	0 0 127
5 5 -3	136 127	127 0 0
-3 5 -12	128 127	0 0 -128
-12 5 -3	119 127	-128 0 0
-3 -3 -3	128 136	0 -128 0
-3 14 -3	128 119	0 127 0

```
g -0.025490198 0.04509804 -0.025490198 Cubo
usemtl metal
```

f 1 0 12	f 3 0 11
f 2 1 12	f 0 4 11
f 3 2 12	f 4 7 11
f 0 3 12	f 7 3 11
f 1 2 9	f 4 5 13
f 2 6 9	f 5 6 13
f 6 5 9	f 6 7 13
f 5 1 9	f 7 4 13
f 0 1 8	f 2 3 10
f 1 5 8	f 3 7 10
f 5 4 8	f 7 6 10
f 4 0 8	f 6 2 10

Quadro 3.5 – Exemplo de um arquivo Mbj mostrando um Cubo

arquivo MBJ fica em torno de 2 segundos. Um intervalo importante, visto que o exemplo construído para validar a implementação do motor é menor que um jogo profissional.

A classe *ObjLoader* foi deixada como opção para importação de arquivos.

3.9 DETECÇÃO DE COLISÃO

A detecção de colisão, efetuada pela classe *CollisionDetection*, foi implementada na forma mais simples possível, evitando perturbar a velocidade da renderização das imagens e prejudicar a jogabilidade. Inicialmente, a detecção acontecia por um único ponto de colisão e um único teste em cada movimento, como pode ser visto na fig. 3.19. Este algoritmo era rápido porém com erros inaceitáveis quando o ângulo de colisão não era de 90º e quando o ponto de teste, que era o ponto central do jogador, não conseguia detectar uma colisão, pois eram as suas laterais que iriam colidir. A fig. 3.19 representa os dois erros encontrados nesta implementação inicial: em ambos os casos, o personagem deveria colidir, mas não colide, ou colide atrasado cortando a visualização de um objeto 3D.

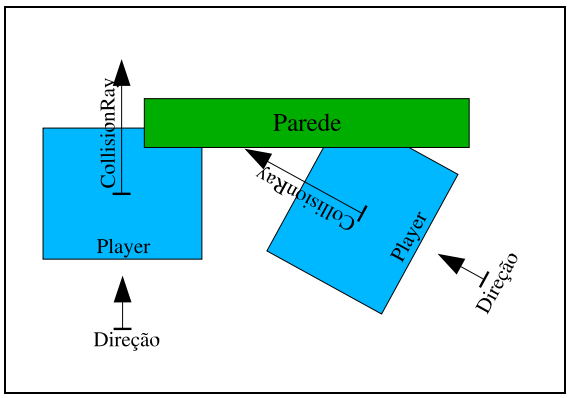


Figura 3.19 – 1ª versão da implementação de colisão

Visto a falha na implementação, evoluiu-se o algoritmo. Em vez de um único teste, três testes são realizados: um no centro e um em cada lateral conforme mostra a fig. 3.20. O raio de colisão, informado pelo desenvolvedor de jogos no arquivo de configurações, é utilizado para definir a distância de colisão e a distância, a partir do ponto central, para encontrar os pontos laterais.

O cálculo de colisão é feito pela função *pick* da classe *Group*. Esta função cria e preenche uma instância da classe *RayIntersection* e retorna verdadeiro caso alguma colisão for detectada.

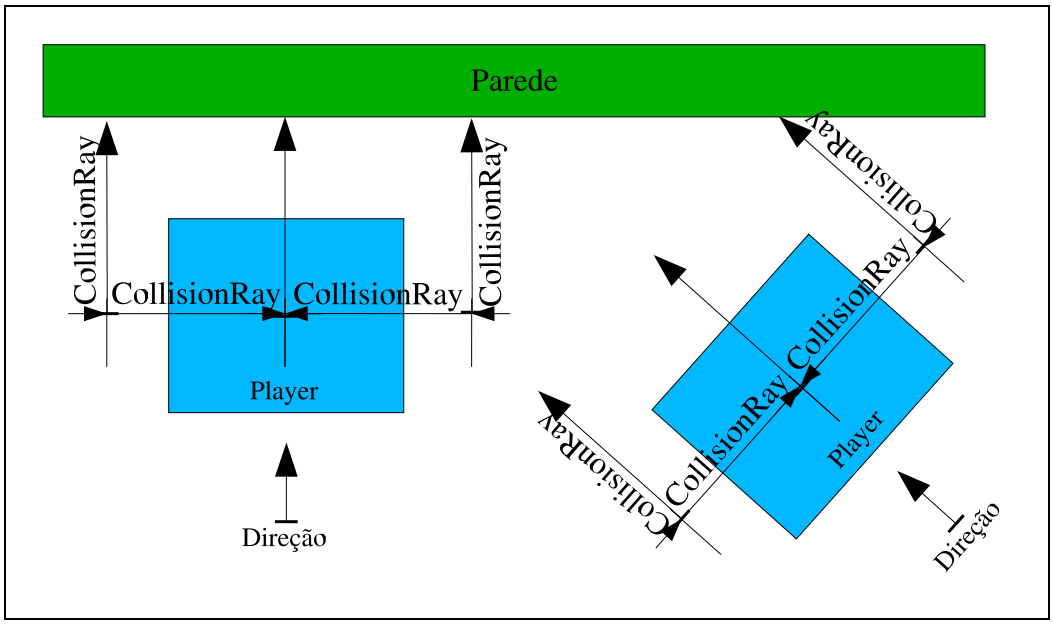


Figura 3.20 – Implementação de colisão

Os parâmetros desta função são a posição de origem, que é o ponto central do grupo *Player* e a posição de destino, que é calculada a partir dos dados do movimento do jogador em cima do ponto central.

O fato da função retornar verdadeiro não quer dizer que a colisão exista. É necessário verificar a que distância um determinado objeto se encontra a partir do ponto central do *Player*. Esta informação é calculada pelo método *pick* e preenchida no *RayIntersection*. A colisão só é determinada quando a distância entre o ponto central e a colisão é menor que o raio de colisão determinado no arquivo de propriedades.

Este cálculo foi separado em uma classe, pois utiliza muitas variáveis e para evitar que a máquina virtual declare e instancie estas variáveis a cada iteração, são utilizadas variáveis globais, ou melhor, atributos de instância.

3.10 RESULTADOS E DISCUSSÃO

Os testes feitos em simuladores foram satisfatórios na comparação com os celulares. Com exceção de um problema na limitação de memória para os aplicativos feitos em J2ME, na qual os emuladores limitavam memória, enquanto os dispositivos reais não. Fora este detalhe, eles conseguiram simular muito bem seus dispositivos.

No quesito velocidade, a leitura de um modelo 3D leva alguns segundos de diferença entre um emulador e um celular, e varia de aparelho para aparelho. A velocidade de jogo foi praticamente igual nas duas plataformas, exceto o tempo de compilação na primeira execução dos *bytecodes* Java. No telefone celular Siemens CX65 são necessários três segundos após o jogador tentar mover o personagem pela primeira vez. Este é o tempo em que a máquina virtual do celular otimiza os códigos deixando-os em cache para facilitar no ciclo de atualização das imagens na tela.

A diferença entre importar um arquivo *Wavefront* OBJ e um MBJ ficou em 2 segundos, sendo que o OBJ leva 17 segundos contra apenas 15 segundos para carregar o MBJ no celular Siemens. Estudos mais detalhados identificaram que os celulares Siemens demoram de 100 a 900 milisegundos para localizar e abrir um arquivo dependendo de seu tamanho. Como para este exemplo são utilizados 9 arquivos, entre eles 6 de imagens texturas, um Mbj ou Obj, um Mtl e um de configuração da engine, pode-se identificar que a abertura dos arquivos pela máquina virtual java consome grande parte do tempo de processamento.

Todos os testes foram feitos com texturas de alta qualidade, medindo 256 *pixels* de largura e altura. Ao reduzir o tamanho das seis imagens, utilizadas no exemplo, para o tamanho de 32 *pixels* de largura e altura, obteve-se um tempo de carga de 9 segundos. Ou seja, o fato de serem imagens grandes carrega a máquina virtual do celular, de maneira que o celular testado gasta 6 segundos só para abrir e descompactar as imagens jpeg.

A fig. 3.21 apresenta a demonstração criada neste trabalho para validar as implementações desenvolvidas no motor de jogos 3D. Estas imagens foram capturas do emulador. Na primeira está a visão do personagem, na segunda a visão de uma câmera acima do personagem em um ângulo de -10 graus em relação ao eixo X. A terceira imagem corresponde a uma câmera que está a 8 unidades acima do nível do mapa e aponta a sua lente para baixo. A quarta imagem retrata o que o personagem pode fazer com todas as teclas habilitadas.

Ao trocar de câmera o jogo mostra uma mensagem indicando o nome da câmera atual por 1.5 segundos. Ao fazer o personagem atirar, uma mensagem é mostrada indicando qual objeto 3D o tiro acertou. A velocidade do jogo varia bastante de aparelho para aparelho, mas

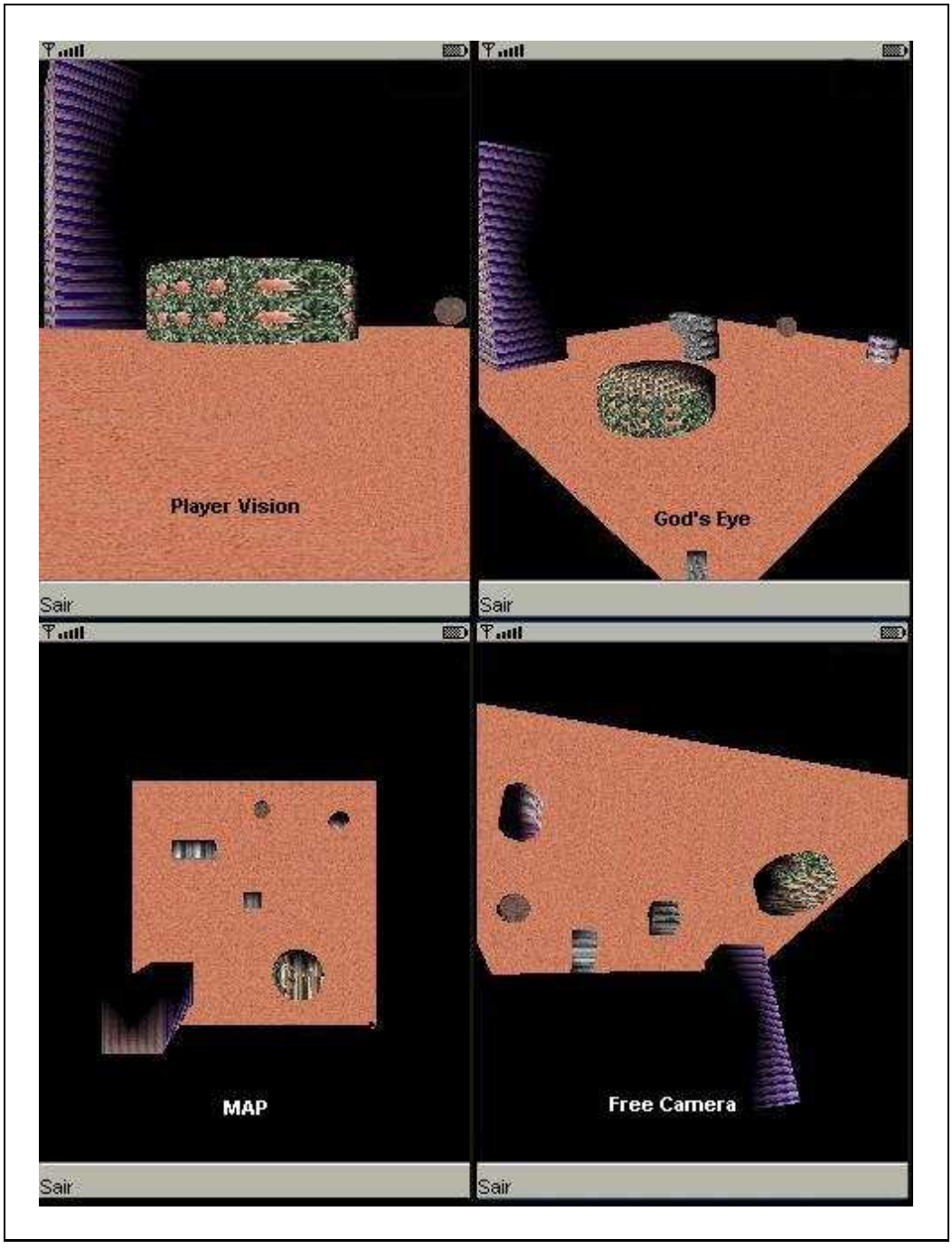


Figura 3.21 – Implementação demonstração de um jogo

fica em torno de 4 a 12 frames por segundo para movimentação e 15 a 20 frames por segundo para a rotação do personagem, onde não existe teste de colisão.

4 CONCLUSÃO

O presente trabalho apresentou a especificação e o desenvolvimento de um motor de jogos 3D em java para celulares com suporte a especificação Mobile 3D Graphics API for J2ME. Definiu e implementou um formato de arquivos especial para facilitar a importação de modelos 3D e uma aplicação utilitária para a conversão de arquivos Wavefront para o formato Mbj. A partir de resultados obtidos em testes realizados em dispositivos reais, pode-se concluir que a tecnologia java pode e deve ser utilizada para construir jogos em dispositivos limitados, embora a preocupação com algoritmos velozes seja sempre necessária.

Construir aplicações com poder de processamento e memória limitada é muito diferente de desenvolver aplicações normais, para micro-computadores. Enquando que nas aplicações *desktop* existe uma pressão por fazer um bom código, separando em componentes, utilizando todos os recursos da orientação a objetos, definindo responsabilidades para cada objeto, implementando *design patterns*, e tantas outras questões pertinentes ao desenvolvimento de um bom software, na data de publicação deste trabalho o ambiente móvel não permite a utilização destes recursos visto a sua limitação computacional. É provável que, com a evolução da tecnologia, estes recursos poderão ser utilizados, desde que não prejudiquem a qualidade gráfica e a jogabilidade da aplicação a ser criada.

O recurso de desenvolvimento em espiral facilitou, e muito, o trabalho para o autor. As especificações iniciais mostraram-se ineficazes no ambiente móvel e foram mudando com o passar do tempo. Ao final do trabalho, pouca especificação original havia restado, sendo que a grande maioria dela foi modifica ou totalmente reescrita.

É interessante ressaltar que todo o desenvolvimento do trabalho pôde ser feito com software livre, desde as IDEs de desenvolvimento e sistemas operacionais utilizados, até as ferramentas gráficas de modelagem 3D. Todas as ferramentas utilizadas agradaram e conseguiram realizar seu papel da maneira como descrita em suas documentações.

Todos os requisitos deste trabalho foram implementados com sucesso:

a) carregar e desenhar um ambiente virtual a partir de um arquivo de configurações;

b) troca de câmeras no cenário;

c) movimentação de personagens no cenário;

d) portabilidade;

e) velocidade.

E ainda mais três itens foram especificados e implementados:

a) detecção de colisão;

b) modelo de eventos;

c) aumento de velocidade reduzindo o arquivo Obj.

A implementação deste trabalho levantou uma série de perguntas sobre o futuro das tecnologias. Será que estamos vivendo hoje a mesma evolução já vista em tempos passados, quando muitos main-frames foram substituídos por computadores pessoais? Será que algum dia os micros pessoais cairão em desuso? Os celulares tem crescido a um ritmo impressionante, e estão a cada dia mais poderosos.

Um exemplo de evolução rápida da tecnologia pôde ser vista no decorrer do desenvolvimento deste trabalho. Enquanto que, no início dos estudos para a confecção da proposta, as fabricantes de celulares não faziam idéia de quando poderiam lançar um celular com suporte a especificação M3G, ao término deste trabalho a Nokia lançou um preview de seu novo aparelho, o Nokia 770 (NOKIA, 2005). Um dispositivo de 14,10 x 7,90 cm, com uma tela *touch screen* com resolução de 800x480 e 65 mil cores, acessando a internet por inúmeras maneiras, e com sistema operacional baseado em linux. Este aparelho provavelmente permitirá aplicações VOIP e TV digital, assim como as aplicações comuns que rodam em desktops: leitor de e-mails, navegador web, *media player*, visualizadores de PDF, entre outros.

4.1 TRABALHOS FUTUROS

Há muito o que pesquisar e o que implementar. Agilizar e/ou melhorar as rotinas de colisão e implementar simulação física de corpos rígidos, como a gravidade e o arremesso de objetos, são alguns dos objetivos mais próximos ao trabalho atual. Especificar e construir componentes para suporte à engine, como barras de progresso, menus iterativos, salvar e restaurar

o estado atual do jogo, carregar pequenos filmes, entre outras, são idéias que precisam ser bem especificadas e implementadas, pois não serão utilizadas em todos os jogos e, portanto, estariam lá apenas para ocupar espaço.

Criar linguagens de *script* para adicionar lógica ao jogo, incluindo inteligência artificial e plataforma para agentes distribuídos. Criar a possibilidade da configuração do jogo via linha de comando, controlar objetivos do jogo e níveis de dificuldade ou fases, são objetivos para trabalhos mais extensos.

Mas, antes de continuar este trabalho, deve-se estudar o estado da tecnologia atual. É possivel que, em alguns anos, jogos complexos feitos para ambientes *desktop* estejam rodando em dispositivos móveis, sem qualquer mudança na programação, tornando este trabalho obsoleto. A tecnologia vai evoluir, e vai tentar trazer todos os recursos dos micro-computadores para os celulares e PDAs, e, quem sabe, criar uma portabilidade, hoje inexistente, entre *desktop* e móvel.

REFERÊNCIAS BIBLIOGRÁFICAS

AARNIO, Tomi. **A new dimension for Java games**: mobile 3d graphics api. [P.O.Box], 2004. Disponível em: <http://www.nokia.com/nokia/0,,62395,00.html>. Acesso em: 12 set. 2004.

BATTAIOLA, André L. et al. Desenvolvimento de jogos em computadores e celulares. **Revista de Informática Teórica e Aplicada**, v. 8, n. 2, out 2001.

BLENDER FOUNDATION. **Blender3D.org**: home. [Amsterdam], 2005. Disponível em: <http://www.blender3d.com/cms/Home.2.0% -.html>. Acesso em: 15 abr. 2005.

CONITEC CORPORATION. **3D GameStudio / A6**. San Diego, 2004. Disponível em: <http://conitec.net/a4info.htm>. Acesso em: 02 out. 2004.

CONTROL CHAOS. **About scrum**. [Lexington], 2005. Disponível em: <http://www-.controlchaos.com/about/>. Acesso em: 21 maio 2005.

CRYSTAL SPACE. **Crystal Space 3D**. [S.l.], 2004. Disponível em: <http://crystal-.sourceforge.net>. Acesso em: 30 set. 2004.

DAVISON, Andrew. **Java games programming techniques**: Java graphics and gaming. [Hat Yai], 2004. Disponível em: <http://fivedots.coe.psu.ac.th/˜ad/jg>. Acesso em: 30 out. 2004.

DISCREET. **Autodesk 3ds max**. [Montreal], 2005. Disponível em: <http://www4.discreet-.com/3dsmax/>. Acesso em: 15 abr. 2005.

EASTMEN, Peter. **The art of illusion**. [S.l.], 2005. Disponível em: <http://www.artofillusion-.org/>. Acesso em: 26 maio 2005.

EBERLY, David H. **3D game engine design**: a practical approach to real-time computer graphics. São Francisco: Morgan Kaufmann, 2001.

ECLIPSE ENTERTAINMENT. **Welcome to the home of Genesis3D**. [RedMond], 2004. Disponível em: <http://www.genesis3d.com/>. Acesso em: 02 out. 2004.

ECLIPSE FOUNDATION. **Eclipse main page**. [Ottawa], 2004. Disponível em: <http://www-.eclipse.org/>. Acesso em: 19 set. 2004.

ECLIPSEME TEAM. **EclipseME home page**. [Scottsdale], 2005. Disponível em: <http://eclipseme.org/index.html>. Acesso em: 25 maio 2005.

EIWA SYSTEM MANAGEMENT, INC. **Jude**. [S.l.], 2005. Disponível em: <http://jude.esm-.jp/>. Acesso em: 16 set. 2004.

HI CORP. **Mascot capsule toolkit for M3G**. Meguro, 2005. Disponível em: <http:-//www.mascotcapsule.com/M3G/download% -/e_plugins.html>. Acesso em: 25 maio 2005.

IBGE. **Pesquisa industrial - produto**: combustíveis, automóveis e minério de ferro lideram as vendas na indústria. [São Paulo], 2004. Disponível em: <http://www.ibge.gov.br-/home/presidencia% -/noticias/noticia_impressao.php?id_noticia=164>. Acesso em: 12 set. 2004.

IBM. **Rational unified process**. [New York], 2005. Disponível em: <http://www-306.ibm-.com/software/awdtools/rup/>. Acesso em: 21 maio 2005.

JBENCHMARK. **JBenchmark result database**: Jbenchmark 3d results. [Budapest], 2005. Disponível em: <http://www.jbenchmark.com/result.jsp>. Acesso em: 15 maio 2005.

JCP. **The Java community process(SM) program**: JCP procedures - JCP 2 process document. [Palo Alto], 2004. Disponível em: <http://www.jcp.org/en/procedures/jcp2>. Acesso em: 30 set. 2004.

JCP. **The Java community process(SM) program**: JSRs - Java specification requests - JSR overview. [Palo Alto], 2004. Disponível em: <http://www.jcp.org/en/jsr/overview>. Acesso em: 28 out. 2004.

KHONOS GROUP. **OpenGL ES**: overview. [San Francisco], 2004. Disponível em: <http://www.opengles.org/opengles/index.html>.

KILE. **Kile**: an integrated latex environment. [S.l.], 2005. Disponível em: <http://kile-.sourceforge.net/>. Acesso em: 28 maio 2005.

LATEX PROJECT TEAM. **Project Latex**: Latex - a document preparation system. [Roedermark], 2005. Disponível em: <http://www.latex-project.org/>. Acesso em: 26 maio 2005.

M3G EXPORTER. **M3G exporter**. [S.l.], 2005. Disponível em: <http://www.m3gexporter-.com/>. Acesso em: 25 maio 2005.

MAHMOUD, Qusay H. **Getting started with the mobile 3D graphics API for J2ME**. [Palo Alto], 2004. Disponível em: <http://developers.sun.com/techtopics/mobility/apis/articles-/3dgraphics/>. Acesso em: 30 out. 2004.

MCNAMARA, Antoine. **MTL file specification**. [New Haven], 2000. Disponível em: <http://zoo.cs.yale.edu/classes/cs490/00-01a/mcnamara.antoine.amm43/mtl.html>. Acesso em: 10 maio 2005.

NOKIA. **JSR-184 mobile 3D API for J2ME**. [P.O.Box], 2003. Disponível em: <http://www-.forum.nokia.com/main/0,6566,040,00.html?fsrParam=1-3-&fileID=3960>. Acesso em: 11 set. 2004.

NOKIA. **RI binary for JSR-184 3D graphics API for J2ME**. [P.O.Box], 2004. Disponível em: <http://www.forum.nokia.com/main/1,6566,040,00.html?fsrParam=2-3-/main-.html&fileID=5194>. Acesso em: 30 set. 2004.

NOKIA. **Series 60 developer platform: 3-D game engine example**. [P.O.Box], 2004. Disponível em: <http://www.forum.nokia.com/main/1,,040,00.html?fsrParam=2-3-/main-.html&fileID=5197>. Acesso em: 30 set. 2004.

NOKIA. **Nokia 770**. [P.O.Box], 2005. Disponível em: <http://www.nokia.com/nokia-/0,1522,,00.html?orig=/770>. Acesso em: 28 maio 2005.

OBJECT MANAGEMENT GROUP. **UML**. [Needham], 2004. Disponível em: <http://www-.uml.org/>. Acesso em: 26 out. 2004.

O'REILLY & ASSOCIATES INC. **GFF format summary**: wavefront obj. [S.l.], 1996. Disponível em: <http://netghost.narod.ru/gff/graphics/summary/waveobj.htm>. Acesso em: 10 maio 2005.

OTTO, George H. **Graphics short course II**: fundamentals of 3-d computer graphics. [S.l.], 2002. Disponível em: <http://viz.aset.psu.edu/gho/sem_notes/3d_fundamentals/index.html>. Acesso em: 1 maio 2005.

PARALELO COMPUTAÇÃO. **Fly3D.com.br**. [Niterói], 2004. Disponível em: <http://www-.fly3d.com.br>. Acesso em: 2 out. 2004.

SIEMENS AG. **CX65 - Siemens - mobile phones portal**. Munich, 2005. Disponível em: <http://communications.siemens.com/cds/frontdoor/0,2241,hq_en_0_24605_rArNrNrNrN,00-.html>. Acesso em: 17 maio 2005.

SIEMENS AG. **Siemens communications group**. Munich, 2005. Disponível em: <https://communication-market.siemens.de/portal/main-.aspx?LangID=0&MainMenuID=10&ParentID=10&LeftID=30&pid=1&cid=0&tid=3000&xid=0>. Acesso em: 25 maio 2005.

SUN MICROSYSTEMS. **Java 2 platform, micro edition (J2ME): JSR 68 overview**. [Palo Alto], 2004. Disponível em: <http://java.sun.com/j2me/overview.html>. Acesso em: 10 set. 2004.

SUN MICROSYSTEMS. **Java 2 platform micro edition, wireless toolkit**. [Palo Alto], 2004. Disponível em: <http://java.sun.com/products/j2mewtoolkit% -/index.html>. Acesso em: 19 set. 2004.

SUN MICROSYSTEMS. **Java BluePrints: model-view-controller**. [Palo Alto], 2004. Disponível em: <http://java.sun.com/blueprints/patterns/MVC-detailed.html>. Acesso em: 20 set. 2004.

SUN MICROSYSTEMS. **Java technology fuels commerce, community and creativity at world's largest developer conference, javaone 2004**. [Palo Alto], 2004. Disponível em: <http://www.sun.com/smi/Press/sunflash/2004-06/sunflash.20040628.1.html>. Acesso em: 1 maio 2005.

SUN MICROSYSTEMS. **Mobile media API (MMAPI); JSR-135 overview**. [Palo Alto], 2004. Disponível em: <http://java.sun.com/products/mmapi/overview.html>. Acesso em: 02 out. 2004.

SUN MICROSYSTEMS. **Sun Microsystems**. [Palo Alto], 2005. Disponível em: <http:/-/www.sun.com>. Acesso em: 11 maio 2005.

WIKIPEDIA. **KISS principle**. [San Diego], 2005. Disponível em: <http://en.wikipedia.org-/wiki/KISS_principle>. Acesso em: 21 maio 2005.

WUESTEFELD, Klaus. **Xispê**. [Curitiba], 2001. Disponível em: <http://www.xispe.com.br-/index.html>. Acesso em: 21 maio 2005.