M3GE: um motor de jogos 3D para dispositivos móveis com suporte a Mobile 3D Graphics API

PAULO CÉSAR RODACKI GOMES
VITOR FERNANDO PAMPLONA

FURB - Universidade de Regional de Blumenau- Departamento de Sistemas e Computação
rodacki@inf.furb.br, vitor@babaxp.org

Resumo

Este artigo apresentas os primeiros resultados obtidos no processo de construção de um motor de jogos 3D para dispositivos móveis chamado Móbile 3D Game Engine (M3GE). O motor utiliza a especificação Mobile 3D Graphics API for J2ME (M3G) e possui recursos de tratamento de colisão, mapeamento de texturas, entre outros. É também apresentada a implementação de um protótipo simples de jogo 3D, utilizando o motor desenvolvido.

Palavras Chave: *Motores de jogos, Dispositivos móveis, celulares, M3G, OpenGL ES*

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, e tende a ultrapassar o volume de produção dos jogos atuais. Esta é a área de jogos para dispositivos móveis. Um mercado muito rico, amplo e diversificado [1].

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 jogos eletrônicos tridimensionais executados em micro-computadores estarão rodando em celulares, *palm tops*, *pocket pcs* e outros [2]. Espera-se que esta evolução crie novos mercados tais como o de desenvolvimento de arquiteturas e ferramentas para facilitar o desenvolvimento desses jogos. Um exemplo deste tipo de ferramenta são os motores de jogos. A plataforma *Java 2 Micro Edition* (J2ME) [3] é uma versão simplificada da linguagem das APIs da linguagem Java e da sua máquina virtual. 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 inicialmente descartado por ser mais lento em comparação a aplicações em C++. A adoção para este tipo de desenvolvimento foi maior em linguagens nativas dos dispositivos por serem mais rápidas e com mais recursos gráficos.

Além do forte e recente investimento da Sony, até o presente momento surgiram poucas iniciativas para o desenvolvimento de jogos 3D em dispositivos móveis em Java. Uma das mais recentes é a especificação *Mobile 3D Graphics API for J2ME* (M3G), proposta pela Nokia [4]. Este artigo propõe e descreve o uso do J2ME e da M3G para implementação de um motor de jogos 3D para dispositivos móveis. Os autores desconhecem propostas semelhantes na literatura e acreditam que esta ausência de referência se deve ao fato da especificação M3G ser ainda muito recente. O motor proposto já está sendo utilizado no desenvolvimento de um jogo comercial [5].

Na próxima seção é feita uma breve discussão sobre motores de jogos 3D. Os componentes da arquitetura proposta são apresentados na seção 3. A seção 4 apresenta a implementação de um protótipo que visa ilustrar a viabilidade da presente proposta. Por fim, a seção 5 apresenta conclusões e trabalhos futuros. Como complemento deste artigo, a página web do projeto (*https://m3ge.dev.java.net*) disponibiliza os arquivos fonte.

2 Motores de Jogos

Os motores são bibliotecas de desenvolvimento responsáveis pelo gerenciamento do jogo, das imagens, do processamento de entrada de dados e outras funções. A idéia é que os motores implementem funcionalidades e recursos comuns à maioria dos jogos de determinado tipo, permitindo que esses recursos sejam reutilizados a cada novo jogo criado [6]. Os motores são tão importantes que estão em praticamente todos os jogos para micro-computadores, controlando a estrutura do jogo e seu ciclo de vida. A figura 1 exibe a arquitetura típica de um motor de jogos 3D [1], [7].

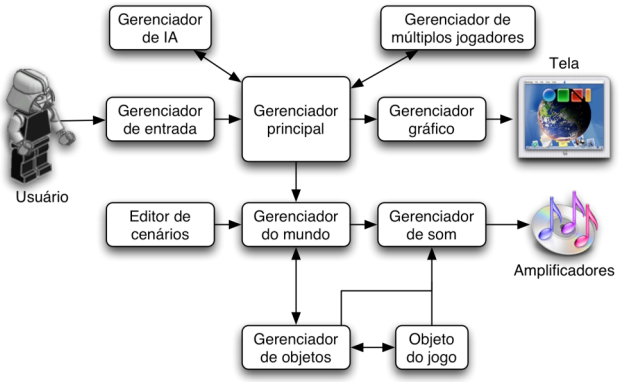

Figura 1: arquitetura de um motor de jogos

O gerenciador de entrada recebe e identifica os eventos de entrada e os envia para o gerenciador principal. O gerenciador gráfico transforma o modelo que define o estado atual do jogo em uma visualização para o usuário. O gerenciador de inteligência artificial gerencia o comportamento dos objetos desenvolvidos pelo *designer* do jogo. O gerenciador de múltiplos jogadores trata da comunicação dos jogadores, independentemente do meio físico em que se encontram. O 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. O objeto do jogo possui dados relevantes para uma entidade que faça parte do jogo (como "avião", "monstro", etc). Esta parte do motor controla a posição, velocidade, dimensão, detecção de colisão, entre outros. O gerenciador do mundo armazena o estado atual do jogo e para isso utiliza os gerenciadores de objetos. Em geral, uma ferramenta externa, o editor de cenários, descreve um estado inicial do jogo para cada um de seus níveis. Por fim, o gerenciador principal é responsável pela coordenação entre os demais componentes.

Atualmente existem vários motores de jogos 3D disponíveis. A lista é extensa, e podemos citar alguns exemplos, tais como os projetos *open source Crystal Space* [8] e *Ogre3D* [9], além do motor proprietário *Fly3D* [10]. Para jogos em dispositivos móveis, os motores mais conhecidos para desenvolvimento utilizando linguagem nativa são o *ExEn* [11] e o *Mophun*, implementada em linguagem C [12]. As engines que utilizam a máquina virtual Java são mais escassas, temos como exemplos o wGem, um dos primeiros motores de jogos para dispositivos móveis do Brasil [5].

Em 1998, com a criação do *Java Community Process* (JCP) [13] a tecnologia Java deixa de ser propriedade da *Sun Microsystems* e passa a ser propriedade de um grupo de especificação, do qual qualquer empresa poderia pertencer. O JCP criou as *Java Specification Request* (JSR) [14], 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 difundida.

Em 2002, o fabricante de telefones celulares Nokia lançou a JSR-184 com o objetivo de criar rotinas gráficas velozes e práticas para substituir as implementações das linguagens nativas, a

Mobile 3D Graphics API for J2ME (M3G). A diferença entre a maioria dos motores de jogos para dispositivos móveis citados acima e o motor proposto neste trabalho está no uso da M3G. A M3G "define rotinas de baixo e alto nível para tornar eficiente e criar interatividade de gráficos 3D para dispositivos com pouca memória e poder de processamento, sem suporte de hardware ou para operações com pontos flutuantes" [15]. Embora esta definição faça menção a dispositivos sem suporte de hardware para 3D, praticamente, apenas os dispositivos que implementam alguma função nativa da *Application Programming Interface* (API) 3D conseguem uma velocidade de renderização aceitável. Alguns celulares Nokia, por exemplo, utilizam uma implementação nativa do OpenGL ES [16], uma versão simplificada do OpenGL. Outros fabricantes de dispositivos estão implementando estruturas similares. A M3G foi especificada para as versões *Mobile Information Device Profile* (MIDP) 2.0 e *Connected Limited Device Configuration* (CLDC) 1.1 [3]. O CLDC é uma configuração que define os recursos da máquina virtual e as bibliotecas principais para J2ME, e o MIDP consiste em um perfil para dispositivos portáteis definindo APIs como a de interface com o usuário, redes e conectividade, armazenamento, entre outros. A figura 2 ilustra a arquitetura básica da M3G:

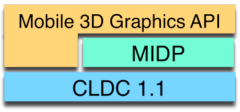

Figura 2: arquitetura básica da M3G

A JSR 184 (M3G), definiu o seguinte conjunto de capacidades que a API deve suportar: (i) 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; (ii) a API deve importar malhas de polígonos 3D, texturas e grafos de cena; (iii) os dados devem estar em formato binário para diminuir o tamanho do armazenamento e a transmissão; (iv) deve ser possível implementar a API sobre a OpenGL ES, sem recursos de ponto flutuante de hardware; (v) a API deve implementar valores com ponto flutuante; (vi) ROM e RAM ocupadas devem ser mínimas. A API deve ocupar menos de 150 KB; (vii) a API deve implementar algum mecanismo de *garbage collection*; e (viii) a API deve ser inter-operável com outras APIs Java, especialmente o MIDP.

A API está definida para ser implementada dentro do pacote *javax.microedition.m3g* contendo 30 classes divididas em 6 grupos: classes básicas, classes para nós de grafo de cena, classes para carga de arquivos e funcionalidades de baixo nível, classes para atributos visuais, classes modificadoras, e, por fim, classes para animação e tratamento de colisão. A seguir é apresentada a proposta do motor de jogos utilizando a M3G.

3 O Motor de jogos M3GE

O motor de jogos proposto no presente artigo é chamado *Mobile 3D Game Engine* (M3GE).Para sua implementação, foram levantados os seguintes requisitos principais: carregar e desenhar um ambiente 3D a partir de um arquivo de configuração; criação de um número indefinido de câmeras, que podem ser trocadas dinamicamente durante o jogo; tratamento de eventos do usuário; movimentação de personagens no cenário, com seis graus de liberdade; portabilidade, oferecida pela linguagem Java; desempenho para renderização em tempo real, com mapeamento de texturas; e tratamento de colisão.

A fig. 3 ilustra a arquitetura proposta pelos autores. Como pode ser visto, a M3GE foi projetada como uma API anexa à M3G. Ou seja, mesmo usando a M3GE, é possível utilizar a M3G diretamente. As duas bibliotecas interagem entre si, proporcionando ao desenvolvedor do jogo uma maior flexibilidade, pois pode-se acessar o OpenGL ES diretamente quando necessário.

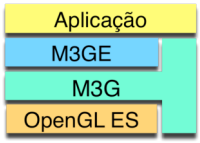

Figura 3: arquitetura básica da M3GE

Em comparação com a arquitetura típica de motores de jogos 3D apresentada na fig. 1, a M3GE implementa elementos escurecidos da fig. 4:

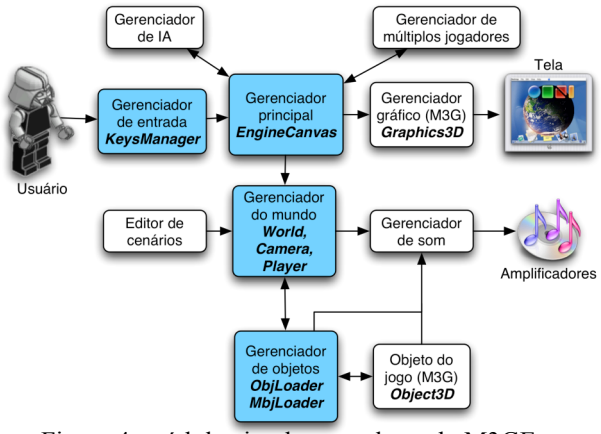

Figura 4: módulos implementados pela M3GE

O projeto de implementação da M3GE foi dividido em dois grandes componentes: o responsável pela leitura dos arquivos de entrada e o responsável pelo motor de jogos propriamente dito, ou *core*, como pode ser visto na fig. 5.

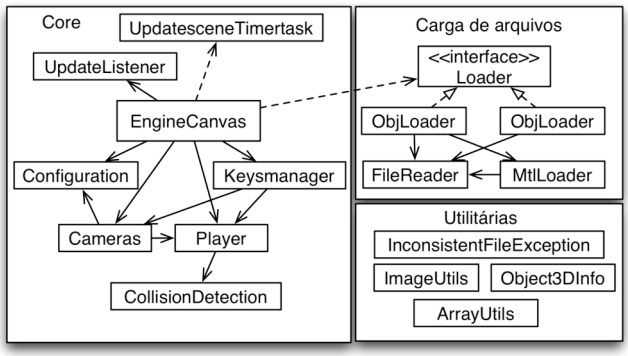

Figura 5: as classes da M3GE

A construção do cenário do jogo é feita pela leitura de um arquivo de configurações gerais do ambiente, um arquivo com as malhas de polígonos e uma série de arquivos de texturas Conforme será visto na seção 3.2. A seguir, são detalhadas as classes do motor de jogos proposto.

3.1 Classes do *core*

A classe *KeysManager* atua como gerenciador de entrada, recebendo eventos do usuário quando ele digita alguma tecla do dispositivo. Esta classe verifica se existem métodos atribuídos a cada tecla pressionada e, quando existir, chama os objetos responsáveis pela ação de acordo com cada tecla. Para que isto aconteça, um evento é lançado pela classe *EngineCanvas*, que é o gerenciador principal do motor de jogos 3D. Ela é responsável pelo gerenciamento do ciclo de vida de todos os objetos, pela chamada do *KeysManager* a cada tecla pressionada, pela renderização da cena utilizando a M3G, pela carga dos arquivos de configuração, acionada pelo construtor da classe, e pela criação de câmeras. Esta classe trabalha com instâncias das classes *Player*, *Cameras* e *World*, que foram construídas separando as responsabilidades de um único gerenciador de mundo, mas atuando em conjunto.

A classe *Player* implementa o personagem principal do jogo, controlado pelo usuário, ela mantém posição, ângulos e tamanho, assim como a geometria 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. Quando necessário, o *Player* também é responsável por chamar as rotinas de teste de colisão. O método *update(KeysManager keys)* movimenta o personagem de acordo com as teclas pressionadas, testando colisão entre o personagem e os demais objetos e a colisão de tiro com algum objeto no modelo.

A classe *UpdateListener* deve ser implementada pelo desenvolvedor do jogo o caso haja necessidade de tratar alguma ação nas rotinas de renderização, ou alguma lógica de jogo a ser adicionada em termos de eventos. Isto permite, por exemplo, que o desenvolvedor de jogos detalhe informações para os jogadores, escrevendo-as diretamente na tela. Os principais eventos disponíveis são o evento *camUpdate*, gerado quando o jogador aciona a troca de câmera, o evento *fire* gerado quando o jogador atira e acerta algum objeto 3D, *keyPressed* e *keyReleased*, gerados quando o usuário pressiona e solta alguma tecla do dispositivo, *update*, gerado antes de ser feito o redesenho da cena e *paint* gerado após o redesenho da cena. Este evento permite, por exemplo, que seja desenhado algum elemento 2D no dispositivo gráfico, após o desenho da cena 3D.

O gerenciador gráfico já é implementado pela M3G, e é representado pela classe *Graphics3D*. Sua responsabilidade é desenhar o

modelo 3D em um dispositivo gráfico 2D. As classes *Configuration* e *CollisionDetection* são responsáveis por carregar os arquivos de configuração do motor e fazer o tratamento de colisões. A detecção de colisão, efetuada pela classe *CollisionDetection*, foi implementada na forma mais simples possível, procurando não perturbar a velocidade da renderização das cenas e não prejudicar a jogabilidade. São feitos testes de colisão em três pontos, um no centro e a frente do jogador e os outros dois nas laterias conforme ilustrado no modelo de detecção de colisão ilustrado na fig. 6, onde R representa o parâmetro *CollisionRay*, que é informado no arquivo de configurações do ambiente do jogo. Esta classe também implementa o cálculo de colisão de tiro disparado pelo personagem contra objetos 3D do ambiente.

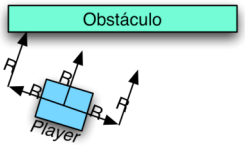

Figura 6: modelo de detecção de colisão

A classe *Cameras* é responsável por manter toda a estrutura de câmeras do jogo, carregando as configurações de arquivo, gerenciando o posicionamento de cada câmera no mundo e identificando qual a câmera atualmente utilizada para renderizar as imagens. A classe mantém um vetor com os parâmetros de cada uma das câmeras do jogo, e um atributo *listener* para lançar eventos de troca de câmeras. A classe *Object3DInfo* armazena as coordenadas do ponto central de cada objeto tridimensional e o seu nome. Por fim, a classe *UpdateSceneTimerTask* implemeta um mecanismo de chamada às rotinas de desenho do motor em um determinado intervalo de tempo.

3.2 Classes de carga de arquivo

As classes *ObjLoader* e *MbjLoader* atuam como gerenciadores de objetos, criando instâncias da classe *Object3D* e dos nós de grafo de cena, e gerenciando o seu ciclo de vida. A M3GE especifica o formato *Wavefront* como padrão para arquivos de entrada de dados para criação do cenário e configuração do ambiente [16]. Este formato foi criado para ser utilizado com o *Wavefront Advanced Visualizer* da Viewpoint DataLabs. O modelo 3D é separado em dois arquivos ASCII. O arquivo *obj* descreve a geometria, podendo conter uma série de tipos de primitivas gráficas. No presente trabalho, apenas as malhas de polígonos 3D são consideradas e lidas pela M3GE. Este arquivo também declara grupos, que são coleções de polígonos 3D formando objetos 3D na cena. O segundo arquivo, de extensão *mtl*, descreve cores, propriedades de materiais e texturas do ambiente ou de um grupo de objetos 3D. As texturas são armazenadas em arquivos de imagem separados. De acordo com a especificação da M3G as imagens devem ser quadradas, com largura e altura medindo potências de 2 *pixels*. Para que a importação de um arquivo *obj* seja viável considerando as limitações de memória e processamento dos telefones celulares atuais, os autores propõem que a sua definição seja extendida adicionando algumas características: os vértices devem ser obrigatoriamente 3D e as faces devem ser triangulares, todos os dados de textura e cores devem ser colocados num único arquivo *mtl*, deve haver a mesma quantidade de vértices, vetores normais, e vetores de textura e eles devem ser igualmente seqüenciados no arquivo.

A carga completa do grafo de cena é feita em três leituras do arquivo *obj* pela classe *ObjLoader*. A leitura inicial conta a quantidade de vértices, vetores normais e texturas 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 o que torna inviável sua utilização na M3GE. O uso de outras estruturas, tais como listas encadeadas também é inviável, devido ao *overhead* de memória necessário. Na segunda leitura do arquivo são obtidas as coordenadas dos vértices, dos vetores normais e dos vetores de textura, além de referências aos dados dos arquivos *mtl* para cada grupo (objeto 3D) da cena, que são importados em uma única leitura do arquivo *mtl*. As coordenadas no arquivo estão em tipos de dados *float* (número com ponto flutuante e precisão simples) e são convertidas internamente para tipo *byte.* Após isso, é feita uma terceira

leitura do arquivo *obj* para a criação dos objetos 3D no jogo, a partir dos grupos declarados no arquivo. Neste momento, é calculado o ponto central de cada objeto 3D.

O arquivo *Wavefront* não foi especificado para ser utilizado por dispositivos móveis e, por conseqüência, este processo de leituras repetidas acaba acarretando demora na montagem do cenário do jogo. Em testes feitos pelos autores, a carga de um arquivo *obj* relativamente pequeno, com 2000 linhas de código, em um telefone celular Siemens CX65 demorou em média 17 segundos até sua apresentação na tela do aparelho. Os celulares Siemens possuem otimizações para re-leituras de arquivos tais como *cache* de arquivos, porém dispositivos de outros fabricantes podem não se comportar da mesma maneira, o que aumentaria significativamente o tempo de carga. Para minimizar este problema, o presente trabalho propõe a especificação de um arquivo semelhante ao *Wavefront*, mas com algumas limitações. Nesta especificação, os dados devem ser do tipo *byte* ao invés do tipo *float* com as coordenadas dos pontos centrais pré-calculadas e com informações sobre o tamanho dos *arrays* a serem montados. O arquivo recebeu a extensão *mbj*, que significa, *Mobile Object File* e sua estrutura é mostrada no quadro 1.

```
nf <int>
nv <int>
nvt <int>
nvn <int>
mtlib <arquivo.mtl>
v vt vn <byte> <byte><byte><byte><byte><byte><byte><byte>
g <float>  <float>  <float> <nome>
usemtl <nome de um material>
f <int[/int[/int]]>[<int[/int[/int]]>[<int[/int[/int]]>]] ...
```

Quadro 1: o formato de arquivos *mbj*

Os atributos nf, nv, nvt e nvn indicam respectivamente as quantidades de faces, vértices, vetores de textura e vetores normais. A seguir, são listadas as coordenadas x, y e z de cada vértice, as coordenadas u e v das texturas e as coordenadas dos vetores normais. Por fim, vêm as declarações de grupos, materiais e faces.

Para ler um arquivo *mbj* dentro do dispositivo, é utilizada a classe *MbjLoader*, muito semelhante a *ObjLoader*, mas sem fazer nenhum cálculo ou conversão de tipos de dados encontrados no arquivo. A leitura é mais rápida que a do formato anterior, principalmente, nos celulares que não implementam nativamente algum tipo de *cache* de arquivos, 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 a carga do mesmo modelo de testes nos dois formatos de arquivo ficou em torno de 2 segundos.

4 Implementação de um jogo simples

Neste trabalho um protótipo de jogo simples é implementado para testar e demonstrar o funcionamento do motor M3GE desenvolvido. Ao contrário das outras aplicações, um jogo é feito com duas unidades de processamento distintas, como pode ser visto na fig. 7. Uma delas atua sobre a camada de modelo do padrão de projeto *Model-View-Controller* (MVC) [3] e é enquanto a outra atua sobre as camadas de controle e visão.

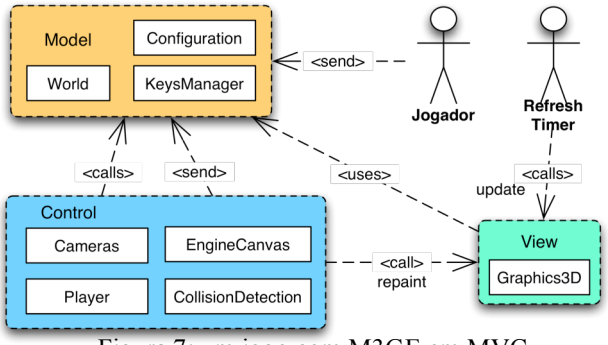

Figura 7: um jogo com M3GE em MVC

Para carregar um cenário e permitir que um personagem ande sobre ele, deve-se criar uma instância da classe *EngineCanvas* passando referências para os arquivos de entrada de dados. O protótipo implementado possui 7 objetos 3D (incluindo um personagem), 4 câmeras e 6 texturas de 256x256 *pixels* cada. Foi utilizado tratamento de colisão do personagem e implementado o disparo de tiros. A fig. 8 exibe a visualização do protótipo num emulador de aparelhos celulares. Na tela da esquerda está a visão em 1ª pessoa do personagem, a direita, tem-se a visão em 3ª pessoa.

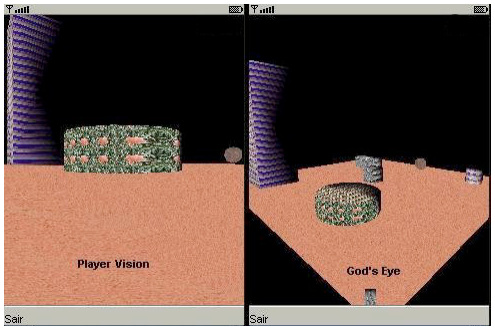

Figura 8: visualização das diferentes câmeras do protótipo

Tanto a M3GE quanto o protótipo foram implementados no Eclipse com o *plugin* EclipseME e Sun Wireless Toolkit (WTK) [18]. Os testes foram realizados com emuladores do *Siemens Wireles Toolkit* [19] e a implementação de referência da M3G da Nokia que já vem com emuladores próprios [4]. Além dos emuladores, foram feitos testes no aparelho celular Siemens CX65. Alguns emuladores limitavam a memória utilizada, enquanto que os aparelhos reais não. 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 cenas na tela. Além disso, o aparelho gastou aproximadamente 6 segundos para ler e descompactar as texturas em formato *jpeg*. A carga de arquivos *obj* demorou 17 segundos, enquanto que o arquivo *mbj* foi carregado em 15 segundos. Considerando a totalidade dos arquivos, estudos mais detalhados identificaram que os celulares Siemens demoram de 100 a 900 milesegundos para localizar e abrir um arquivo dependendo de seu tamanho, assim concluímos que a abertura dos arquivos pela máquina virtual Java consome grande parte do tempo de processamento. A taxa de renderização ficou em torno de 8 *frames* por segundo para movimentação do personagem e 17 *frames* por segundo para a rotação do personagem, onde não existe teste de colisão.

Um primeiro jogo comercial utilizando a M3GE está sendo desenvolvido pela empresa norte americana Autonomous Productions [5]. O jogo, cujo provável nome comercial é *Tranquility*, deverá ser lançado no mercado nos próximos meses, e consiste em um simulador de vôos 3D. A fig. 9 apresenta uma tela do jogo no emulador do WTK [19].

Figura 8: simulador de vôo em desenvolvimento

5 Conclusões

O presente trabalho apresentou a especificação e o desenvolvimento de um motor de jogos 3D em java para celulares com suporte à especificação *Mobile 3D Graphics API for J2ME*. O trabalho também definiu e implementou um formato de arquivos especial para facilitar a importação de modelos 3D e uma aplicação 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 ser utilizada para construir jogos 3D em dispositivos limitados, embora a preocupação com algoritmos velozes seja sempre necessária pois construir aplicações com poder de processamento e memória muito limitados é muito diferente de desenvolver aplicações normais, para micro-computadores.

A viabilidade da proposta é apresentada através da modelagem e implementação de um

protótipo de jogo simples. Não se trata de um trabalho completo, mas sim de uma primeira experiência, visto que a M3GE ainda não implementa uma série de funcionalidades desejáveis tais como força gravitacional, dinâmica de corpos rígidos e outras. Mesmo assim, um primeiro jogo comercial já está sendo desenvolvido com a primeira versão da M3GE, disponibilizada sob licença GPL. Como proposta de continuação da presente pesquisa, sugere-se o desenvolvimento de um *framework* com uma biblioteca de classes mais completa. Ainda como sugestão de trabalhos futuros, aponta-se a necessidade de testes de desempenho com diferentes modelos de aparelhos celulares.

6 Agradecimentos

Os autores agradecem ao Sr. Marcelo Eduardo M. de Oliveira, do Instituto Nokia de Tecnologia e ao Dr. Andrew Davison pela troca de informações e disponibilização de material de consulta.

7 Referências

[1] Battaiola, A. L. et al. *Desenvolvimento de jogos em computadores e celulares*. Revista de Informática Teórica e Aplicada, v. 8, n. 2, out 2001.

[2] Aarnio, T. *A new dimension for Java games: mobile 3d graphics api*. http://www.nokia.com/nokia/0,,62395,00.html (12/09/2004).

[3] Sun Microsystems. *Java 2 platform, micro edition (J2ME): JSR 68 overview*, http://java.sun.com/j2me/overview.html (10/09/2004)

[4] Nokia. *JSR-184 mobile 3D API for J2ME*, http://www.forum.nokia.com/main/0,6566,040,00.html?fsrParam=1-3-&fileID=3960 (11/092004).

[5] Guiliano, S. *Autonomous Productions*, http://www.autonomousproductions.com/website/index.html, (14/09/2005).

[6] Pessoa, C. *wGem: um motor de desenvolvimento de jogos para dispositivos móveis*, Dissertação de Mestrado, UFPE, 2001.

[7] Eberly, D. H. *3D game engine design: a practical approach to real-time computer graphics*, Morgan Kaufmann, San Francisco, 2001.

[8] Crystal Space. *Crystal Space 3D*, http://crystal.sourceforge.net (30/09/2004).

[9] Ogre3D. *OGRE 3D: open source graphics engine*, http://www.ogre3d.org, (15/07/2005).

[10] Paralelo Computação. *Fly3D*, http://www.fly3d.com.br, (02/10/2004).

[11] Ramos, O. R. et al. *A Mobile Device Game Development Initiative in Academia: Challenges and Preliminary Results*, In: proceedings of WJogos 2003, Porto Alegre, 2003.

[12] Amaro, P. *The Clash of Mobile Platforms: J2ME, ExEn, Mophun and WGE*. http://www.gamedev.net/reference/articles/article1944.asp. (15/08/2004).

[13] JCP. *The Java community process(SM) program: JCP procedures - JCP 2 process document*, http://www.jcp.org/en/procedures/jcp2. (30/09/2004).

[14] JCP. *The Java community process(SM) program: JSRs - Java specification requests - JSR overview*, http://www.jcp.org/en/jsr/overview, (28/10/2004).

[15] Mahmoud, Q. H. *Getting started with the mobile 3D graphics API for J2ME*, http://developers.sun.com/techtopics/mobility/apis/articles/, (30/10/2004).

[16] Khonos Group. *OpenGL ES: overview*, http://www.opengles.org/opengles/index.html, (04/08/2005).

[17] O'Reilly & Associates Inc. *GFF format summary: wavefront obj*, http://netghost.narod.ru/gff/graphics/summary/waveobj.htm, (10/05/2005).

[18] Sun Microsystems. *Java 2 platform micro edition, wireless toolkit*, http://java.sun.com/products/j2mewtoolkit%-/index.html, (19/09/2004).

[19] Siemens AG. *Siemens communications group*, https://communication-market.siemens.de/portal/, (25/05/2005).