UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

GITA: UM FRAMEWORK PARA MANIPULAÇÃO DE MODELOS 3D EM DISPOSITIVOS MÓVEIS UTILIZANDO A PLATAFORMA .NET CF 2.0

MARCOS DELL' ANTONIO DE SOUZA

BLUMENAU
2007

2007/1-28

MARCOS DELL' ANTONIO DE SOUZA

GITA: UM FRAMEWORK PARA MANIPULAÇÃO DE MODELOS 3D EM DISPOSITIVOS MÓVEIS UTILIZANDO A PLATAFORMA .NET CF 2.0

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 Ciências da Computação — Bacharelado.

Prof. Paulo César Rodacki Gomes, Dr. - Orientador

BLUMENAU
2007

2007/1-28

GITA: UM FRAMEWORK PARA MANIPULAÇÃO DE MODELOS 3D EM DISPOSITIVOS MÓVEIS UTILIZANDO A PLATAFORMA .NET CF 2.0

Por

MARCOS DELL' ANTONIO DE SOUZA

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

Presidente:	Prof. Paulo César Rodacki Gomes, Dr. – Orientador, FURB
Membro:	Prof. Maurício Capobianco Lopes, Msc. – FURB
Membro:	Prof. Dalton Solano dos Reis, Msc .– FURB

Blumenau, 05 de julho de 2007

Dedico este trabalho às únicas pessoas existentes capazes de dedicar amor incondicional a mim: meus pais.

AGRADECIMENTOS

Aos meus pais, pela oportunidade de cursar uma Universidade.

Aos verdadeiros amigos, que sempre estiveram presentes incentivando a conclusão do trabalho.

À minha namorada, pela compreensão e paciência durante o desenvolvimento do trabalho.

À empresa Bremen Sistemas e Serviços LTDA, pela liberação durante o horário de trabalho para a realização deste projeto.

Ao meu orientador, Paulo César Rodacki Gomes, pela confiança e dedicação.

Ao acadêmico Vitor Fernando Pamplona, por todas as dicas e dúvidas respondidas ao longo do trabalho.

Ao colega Evgeniy Pankov, da Bielo-Rússia, por todo o tempo dedicado às minhas dúvidas.

Ao colega Enrique Sangenis, do México, por ter executado o projeto em um dispositivo móvel e enviado todas as informações necessárias para descrever a experiência neste documento.

Até quando você vai ficar usando rédia? Até quando você vai ficando mudo? Muda, que o medo é um modo de fazer censura.

Gabriel, O pensador

RESUMO

Este trabalho descreve a construção de um *framework* para manipulação de modelos 3D em dispositivos móveis compatíveis com o .NET Compact Framework 2.0 e a biblioteca Managed Direct3D Mobile. Ele permite que modelos 3D sejam carregados e renderizados em um equipamento rodando o sistema operacional Windows Mobile 5.0 ou 6.0. Além do *framework*, também descreve a implementação de um software demonstrativo para validar o trabalho e as tecnologias empregadas.

Palavras-chave: *Framework*. Computação gráfica. Managed Direct3D Mobile. Modelos 3D.

ABSTRACT

This work describes a framework to manipulate 3D models in mobile devices compatible with the .NET Compact Framework 2.0 and the Managed Direct3D Mobile library. It allows 3D models to be loaded and rendered in a device running the Windows Mobile operational system version 5.0 or 6.0. Moreover, it also describes a single software implementation to validate the work and the technologies used.

Key-words: Framework. Graphic computer. Managed Direct3D Mobile. 3D models.

LISTA DE ILUSTRAÇÕES

Figura 1 - Jogo DOOM desenvolvido com a API GAPI para Windows Mobile 15

Figura 2 – Jogo Space Invaders desenvolvido com a API GDI 15

Figura 3 – Exemplos de dispositivos móveis 16

Figura 4 – Teclado *Qwerty* 17

Figura 5 – DELL Axim x51v 17

Figura 6 – Versões do Windows Mobile 18

Figura 7 – Arquitetura do .NET CF 2.0 19

Figura 8 - Exemplos de objetos modelados utilizando malha de polígonos 20

Figura 9 – *View Frustum* 24

Figura 10 – *Field of View* 24

Quadro 1 – Formato do arquivo *OBJ* 26

Quadro 2 – Exemplo de uso do comando usemtl 27

Figura 11 – Vetor normal 27

Figura 12 – Mapeamento da textura 27

Quadro 3 – Formato do arquivo *MTL* 28

Quadro 4 – Exemplo de um cubo: arquivo *OBJ* 29

Quadro 5 – Exemplo de um cubo: arquivo *MTL* 29

Figura 13 – Exemplo apresentado nos quadros 4 e 5 sendo renderizado sem modificações ... 29

Figura 14 – Exemplo dos quadros 4 e 5 sendo renderizado após algumas rotações, escalas e a adição de uma luz 30

Figura 15 – Resultados obtidos com a mOGE 31

Figura 16 – Visão geral da M3GE 32

Figura 17 – Resultado obtido com a M3GE 33

Figura 18 – Emuladores do Windows Mobile 5.0 e 6.0, respectivamente 35

Figura 19 – Relacionamento entre os componentes do projeto 36

Figura 20 – Componentes da biblioteca Gita 37

Figura 21 – Classes que compõem o `ObjLoader` 38

Figura 22 – Propriedades e métodos da classe `ObjLoader` 39

Figura 23 – Propriedades e métodos da classe `Vertex` 40

Figura 24 – Propriedades e métodos da classe `VertexNormal` 40

Figura 25 – Propriedades e métodos da classe `VertexTexture` 40

Figura 26 – Propriedades e métodos da classe `Face`..........41
Quadro 6– Definição do FVF..........41
Figura 27 – Propriedades e métodos da classe `MbjLoader`..........42
Figura 28 – Componente `Transformations`..........43
Figura 29 – Interface `ITransformation`..........43
Figura 30 – Classe `Params`..........44
Figura 31 – Enumeração `TransformKind`..........44
Figura 32 – Classe `TransformationFactory`..........44
Quadro 7 – Formato do arquivo de especificação das câmeras..........45
Figura 33 – Classe `Câmera`..........46
Figura 34 – Classe `CameraList`..........46
Figura 35 – Classe `BoudingBox`..........47
Figura 36 – Funções do 3D Viewer..........48
Figura 37 – Janela de pesquisa dos arquivos *OBJ*..........48
Figura 38 – Transformações disponíveis no menu *Transform*..........49
Figura 39 – Escolha da câmera para navegação..........49
Figura 40 – Classes que compõem o visualizador e o relacionamento com a Gita e MD3DM50
Figura 41 – Diagrama de seqüência para carregar um arquivo e renderizá-lo..........51
Figura 42 – Diagrama de seqüência do método `Render`..........52
Figura 43 – Diagrama de seqüência do método `DrawMesh`..........52
Quadro 8 – Implementação do evento `DeviceReset`..........53
Quadro 9 – Evento `OnPaintBackground` sobrescrito para anular o seu comportamento..........53
Figura 44 – Modelo 3D renderizado no emulador..........54
Figura 45 – Modelo 3D renderizado em um dispositivo HTC S620..........54
Figura 46 – Textura utilizada nos modelos das figuras 44 e 45..........55

LISTA DE SIGLAS

.NET CF - *.NET Compact Framework*

2D – Bidimensional

3D – Tridimensional

API – *Application Programming Interface*

CLR – *Commom Language Runtime*

FVF – *Flexible Vertex Format*

GAPI – *Game Application Programming Interface*

GPU – *Graphics Processing Unit*

HTC – *High Tech Computer Corporation*

KISS – *Keep It Single,* Stupid

MD3DM – *Managed Direct3D Mobile*

ONU – Organização das Nações Unidas

PDA – *Personal Digital Assistant*

RAM – *Random Access Memory*

SDK – *Software Development Kit*

UIT – União Internacional de Telecomunicação

UML – *Unified Modeling Language*

XML – *eXtensible Markup Language*

XP – *eXtreme Programming*

YAGNI – *You Ain't Gonna Need It*

SUMÁRIO

1 INTRODUÇÃO **12**
1.1 OBJETIVOS DO TRABALHO 13
1.2 ESTRUTURA DO TRABALHO 13
2 FUNDAMENTAÇÃO TEÓRICA **14**
2.1 HISTÓRIA DO DESENVOLVIMENTO DE APLICAÇÕES 3D MÓVEIS UTILIZANDO AS TECNOLOGIAS DA MICROSOFT 14
2.2 DISPOSITIVOS MÓVEIS 16
2.3 WINDOWS MOBILE E A PLATAFORMA .NET CF 2.0 18
2.4 MODELOS TRIDIMENSIONAIS 19
2.5 MANAGED DIRECT3D MOBILE 21
2.6 O FORMATO DE ARQUIVOS *WAVEFRONT* 25
2.7 TRABALHOS CORRELATOS 30
2.7.1 mOGE 30
2.7.2 M3GE 31
3 DESENVOLVIMENTO DO TRABALHO **34**
3.1 REQUISITOS PRINCIPAIS 34
3.2 FERRAMENTAS E TECNOLOGIAS UTILIZADAS 35
3.3 GITA E 3DV 36
3.3.1 GITA 37
3.3.1.1 ObjLoader 37
3.3.1.2 Transformations 42
3.3.1.3 Camera 45
3.3.1.4 Common 46
3.3.2 3D Viewer 47
3.4 RESULTADOS E DISCUSSÃO 53
4 CONCLUSÕES **56**
4.1 EXTENSÕES 56
REFERÊNCIAS BIBLIOGRÁFICAS **58**

1 INTRODUÇÃO

A representação digital de modelos tridimensionais, definidos por Santee (2005, p. 150) como corpos formados por um conjunto de vértices, torna-se mais comum à medida que aumenta a capacidade de processamento dos computadores. Segundo Moore (2005), esta capacidade tende a dobrar a cada dezoito meses.

Essa evolução aplica-se também a dispositivos móveis, tais como celulares e PDAs. Segundo Pamplona (2005, p. 16), outras mudanças acontecem paralelamente, tais como a incorporação de uma tela sensível ao toque (*touch screen*) com resolução maior e colorida e o uso de sistemas operacionais juntamente com linguagens de programação para gerenciar e acessar os recursos dos dispositivos.

Quanto ao desenvolvimento, a necessidade de padronização e simplicidade molda um futuro onde toda e qualquer espécie de código será gerenciada por um *framework*[1]. Os benefícios que acompanham essa tendência são vários: portabilidade, segurança, padronização, reusabilidade, etc.

O sistema operacional para dispositivos móveis *Windows Mobile* (MICROSOFT, 2005c), por exemplo, suporta o *.NET Compact Framework 2.0*[2] (MICROSOFT, 2005a). Levando em consideração que alguns celulares e PDAs rodam este sistema operacional, uma aplicação escrita para um modelo de dispositivo rodará, teoricamente, no outro.

Tendo em vista esses recursos oferecidos pelos dispositivos móveis em questão, surgiu a idéia de explorá-los através de uma aplicação para a manipulação de modelos 3D. Em outras palavras, o desafio proposto neste trabalho é criar um *framework* compatível com o .NET CF 2.0 e a biblioteca gráfica MD3DM capaz de manipular modelos 3D em dispositivos móveis.

A API MD3DM, que é a biblioteca gráfica da Microsoft para dispositivos móveis, não tem suporte para a carga de modelos 3D a partir de arquivos, o que a deixa muito limitada, pois não é possível ler e renderizar qualquer informação do mundo externo. Portanto, este trabalho é pioneiro no estudo desta biblioteca para renderização de modelos 3D a partir de um formato aberto, como é o *Wavefront* (O'REILLY & ASSOCIATES INC, 1996).

---

[1] *Framework* é um conjunto de classes/rotinas pré-definidas que são reutilizáveis em vários projetos.

[2] O *.NET Compact Framework* é a versão compacta para dispositivos móveis do *Framework .NET* da Microsoft.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um *framework* para manipulação de modelos 3D para dispositivos móveis que permita ao usuário navegar pelo ambiente visualizado.

Os objetivos específicos do trabalho são:

a) ler, carregar e renderizar um modelo 3D no formato *Wavefront*;
b) permitir que o usuário navegue pelo modelo carregado através do posicionamento da câmera;
c) validar o uso da biblioteca gráfica MD3DM, através de uma aplicação exemplo, no que diz respeito ao desempenho utilizando a métrica de *frames* por segundo.

1.2 ESTRUTURA DO TRABALHO

Esta monografia divide-se em três partes: fundamentação teórica, desenvolvimento do trabalho e conclusões.

O capítulo sobre fundamentação teórica mostra ao leitor todas as informações necessárias para entender o problema e a solução proposta. Já no capítulo sobre o desenvolvimento do trabalho, são apresentados os detalhes do problema, as tecnologias e ferramentas utilizadas durante o desenvolvimento e a solução apresentada. Por fim, as conclusões refletem a opinião do autor sobre todos os recursos e tecnologias utilizados durante o trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados alguns conceitos teóricos relacionados ao trabalho. A primeira seção apresenta uma breve história sobre o desenvolvimento de aplicações 3D em dispositivos móveis e de que forma as tecnologias da Microsoft evoluíram neste sentido. Logo em seguida são apresentados os conceitos e tecnologias utilizados no desenvolvimento do protótipo e, por fim, os trabalhos correlatos.

2.1 HISTÓRIA DO DESENVOLVIMENTO DE APLICAÇÕES 3D MÓVEIS UTILIZANDO AS TECNOLOGIAS DA MICROSOFT

O desenvolvimento de aplicações 3D para dispositivos móveis torna-se mais comum com o passar dos anos, pois a capacidade de processamento e os recursos destes hardwares estão cada vez melhores (PAMPLONA, 2005, p. 16). No entanto, não são só os equipamentos que evoluem. Os sistemas operacionais, por exemplo, já incorporam APIs de alto nível (no mesmo estilo dos *desktops*) para o desenvolvimento de aplicações 3D.

Um exemplo muito claro da evolução no que diz respeito aos softwares, foi o lançamento do Windows Mobile 5. Junto com ele veio o .NET CF 2.0 que traz incorporado à sua arquitetura o MD3DM. Enquanto estas tecnologias não estavam disponíveis, os desenvolvedores de aplicações gráficas utilizavam as seguintes APIs:

a) GAPI: disponível somente para Pocket PCs e de forma nativa, provê apenas doze funções básicas. Seu ponto forte é o acesso direto e exclusivo ao *buffer* do vídeo, o que proporciona um desempenho excelente na renderização e manipulação das informações. Foi muito utilizada para a criação de jogos e aplicações gráficas até o lançamento do MD3DM. Um exemplo de jogo desenvolvido com esta API pode ser visto na figura 1;
b) GDI: possui funções básicas para desenhar linhas, curvas, textos e imagens. Utiliza as mensagens do sistema operacional para determinar quando a aplicação deve renderizar as informações. Diferentemente da GAPI, a GDI não oferece acesso direto ao *buffer* do vídeo. Apesar de ser uma API antiga, ainda hoje é usada em aplicações gráficas. A figura 2 apresenta um exemplo de um jogo desenvolvido

com esta API.

Fonte: Revolution Software (2007).

Figura 1 - Jogo DOOM desenvolvido com a API GAPI para Windows Mobile

Fonte: Gold (2003).

Figura 2 – Jogo Space Invaders desenvolvido com a API GDI

O MD3DM representa uma evolução no desenvolvimento de aplicações gráficas para dispositivos móveis, pois trouxe ao desenvolvedor muitas das facilidades já existentes no desenvolvimento para *desktop*. Na seção 2.5 esta API é descrita em detalhes.

2.2 DISPOSITIVOS MÓVEIS

Segundo Pamplona (2005 p. 17), a computação móvel é caracterizada por um dispositivo móvel com capacidade de processamento em um ambiente sem fio. Exemplos destes equipamentos são os PDAs, Pocket PCs, Smartphones e celulares, tais como os apresentados na figura 3.

Fonte: Yoshimura (2006).
Figura 3 – Exemplos de dispositivos móveis

A capacidade de processamento e a disponibilidade de memória destes equipamentos são muito baixas, o que torna o desenvolvimento de aplicações muito mais difícil em relação ao uso limitado de tais recursos do que em ambientes *desktop*. Porém, a sua adoção no mercado é incontestável e inevitável, pois eles dão mais mobilidade às pessoas e aumentam a produtividade no dia-a-dia.

No que diz respeito ao desenvolvimento de aplicações com estes dispositivos, existem algumas limitações:

a) entrada de dados: o teclado numérico, padrão para os celulares, está longe de ser a melhor opção para alimentar o dispositivo com informações do mundo exterior. Apesar de existirem alguns celulares com teclados *Qwerty* (vide figura 4), que possui todas as teclas alfanuméricas, eles ainda não são um padrão de mercado, e por isso normalmente não são considerados como tal na hora do desenvolvimento;
b) *hardware*: além da baixa capacidade de processamento e falta de memória, a compatibilidade entre os dispositivos também é um empecilho, pois uma aplicação desenvolvida para um equipamento pode não funcionar no outro (este é o principal problema que os *frameworks* tentam resolver, mas as soluções disponíveis hoje em dia ainda não resolvem o problema por completo);
c) *software*: apesar das evoluções na área de sistemas operacionais, ainda há uma enorme necessidade de integração e portabilidade entre os softwares desenvolvidos para dispositivos móveis e *desktops*. Normalmente a arquitetura básica de uma aplicação é portável entre vários ambientes. No entanto, as características inerentes

de cada equipamento fazem com que a aplicação acabe se tornando específica para ele.

Fonte: eXpansys (2007).

Figura 4 – Teclado *Qwerty*

Não há no mercado, até a data do presente trabalho, um telefone celular compatível com o Windows Mobile e com hardware gráfico específico (GPU). Porém, já existe um PDA que executa satisfatoriamente aplicações 3D: o DELL Axim x51v (vide figura 5). Ele roda o sistema operacional Windows Mobile 5.0 e possui o *chipset* gráfico da Intel modelo 2700g, que comporta 16MB de memória para processamento gráfico (SOLOMATIN, 2006).

Apesar de todas estas limitações, a adoção destes aparelhos não pára de crescer. Segundo dados da UIT, órgão ligado à ONU, o número de usuários de telefonia celular no mundo cresceu 137% nos últimos cinco anos e chegou a 1,74 bilhão de usuários (FOLHA ONLINE, 2005).

Fonte: Smart Device Central (2005).

Figura 5 – DELL Axim x51v

2.3 WINDOWS MOBILE E A PLATAFORMA .NET CF 2.0

O sistema operacional da Microsoft específico para dispositivos móveis (PDAs e celulares) é o Windows Mobile. A primeira versão significativa no contexto deste trabalho, ou seja, com recursos mais avançados para o desenvolvimento de aplicações gráficas, foi lançada em 2003 juntamente com o .NET CF 1.0. A partir daí a tecnologia evoluiu, passando pelas versões 2003se, 5 até a atual 6 (vide figura 6).

O lançamento da versão 5.0 trouxe recursos que dividiram a história do desenvolvimento de aplicações 3D para dispositivos móveis. Antes desta versão eram utilizadas duas APIs simples (descritas no item 2.1), após o lançamento dela os desenvolvedores passaram a ter à disposição uma biblioteca muito mais completa: o MD3DM, descrito no item 2.5.

A versão 6, que possui o codinome *Crossbow*, manteve as mesmas características da anterior no que diz respeito ao desenvolvimento de aplicações 3D. Ela sofreu somente alterações de acessibilidade e integração, mas a API MD3DM e o .NET CF 2.0 continuaram os mesmos (MOBILE REVIEW, 2006).

Figura 6 – Versões do Windows Mobile

Sobre o .NET CF 2.0, Barnes (2003) o descreve como uma plataforma de desenvolvimento para dispositivos móveis que traz ao desenvolvedor a possibilidade de criar aplicações gerenciadas e aproveitar um extenso conjunto de funcionalidades já implementadas e suficientemente testadas. Além disso, a experiência adquirida no desenvolvimento para computadores é reaproveitada para os dispositivos móveis.

A arquitetura do .NET CF 2.0 (figura 7), segundo a Microsoft (2005b), possui três componentes básicos: o *framework* em si, que é formado por uma biblioteca de classes reutilizáveis; a CLR, que é responsável por executar as aplicações; e o sistema operacional

Microsoft Windows para dispositivos móveis.

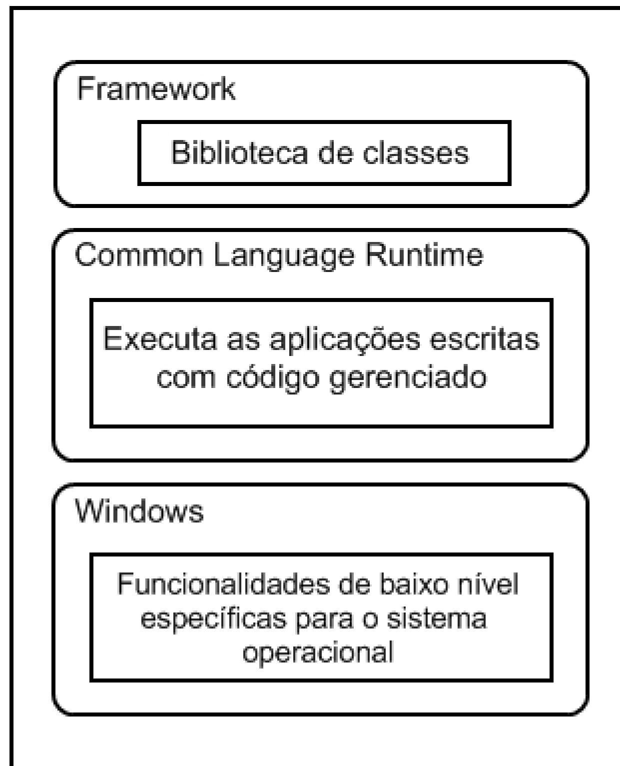


Figura 7 – Arquitetura do .NET CF 2.0

Apesar de seguir a mesma arquitetura básica para PDAs e celulares, o .NET CF 2.0 possui algumas diferenças entre estes dois modelos de *hardware*. Uma delas é a inexistência da classe OpenFileDialog na versão disponível para os celulares. Esta, por sua vez, é muito útil quando a aplicação precisa que o usuário selecione algum arquivo externo. Uma forma de suprir esta necessidade é utilizando uma implementação de terceiros, tal como a OpenNETCF (OPENNETCF, 2005), que é um *framework* que estende o .NET CF 2.0 em diversas direções.

2.4 MODELOS TRIDIMENSIONAIS

Segundo Santee (2005, p. 150), um corpo formado por um conjunto de vértices, como uma bola ou o corpo humano, é chamado de modelo 3D. Em aplicações e jogos profissionais, estes modelos são criados utilizando editores gráficos, tais como: 3D Studio Max (AUTODESK, 2006a), Maya (AUTODESK, 2006b), LightWave (NEWTEK, 2006) e Blender (BLENDER FOUNDATION, 2005). No que diz respeito ao Blender, também há uma versão portátil dele para Pocket PCs, chamada de BlenderPocket (RUSSO, 2007).

Gomes e Velho (2006) utilizam quatro universos para descrever os fundamentos da

modelagem tridimensional. Desta forma, pode-se analisá-la em quatro níveis. São eles:

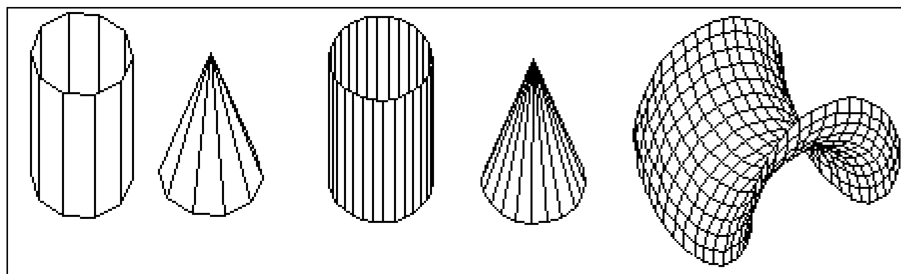
a) físico: situam-se os objetos a serem caracterizados;
b) matemático: define-se modelos para a geometria desses objetos;
c) representação: os esquemas e parâmetros associados aos objetos são criados;
d) implementação: são estabelecidas as estruturas de dados e procedimentos para representar os objetos no computador.

Com base nos fundamentos acima, são descritos alguns esquemas de representação dos modelos 3D. Os mais comuns são:

a) família de primitivos: deve ser definido um conjunto de funções para descrever uma classe de objetos. Além disto, transformações geométricas também podem ser incluídas nestas definições;
b) esquemas construtivos: as primitivas geométricas e operações de combinação de conjunto de pontos são utilizados nesta representação. Desta forma, é possível construir objetos compostos complexos a partir de objetos simples;
c) esquemas de decomposição: nesta situação é adotada uma estratégia contrária a dos esquemas construtivos. Ao invés de partir de formas simples para construir formas mais complexas, o processo é exatamente o reverso (utilizam-se formas complexas e a partir delas são obtidas formas mais simples).

Tenorio (2003), por sua vez, sintetiza as idéias acima afirmando que a representação por malha de polígonos, além de ser a mais utilizada, é definida como sendo uma coleção de polígonos que juntos formam um objeto. Se for preciso modelar uma superfície plana, as malhas adaptam-se perfeitamente ao modelo. Se o objeto possuir superfícies curvas, elas devem ser aproximadas por um conjunto de polígonos.

A figura 8 ilustra alguns objetos modelados utilizando malha de polígonos.



Fonte: Tenorio (2003).

Figura 8 - Exemplos de objetos modelados utilizando malha de polígonos

2.5 MANAGED DIRECT3D MOBILE

A MD3DM é a biblioteca gráfica da Microsoft para dispositivos móveis. Ela implementa várias funcionalidades da API Direct3D para computadores e é suportada por equipamentos que rodam os sistemas operacionais Windows Mobile a partir da versão 5.0. Além disso, já vem incorporada ao .NET CF 2.0, dispensando qualquer trabalho de instalação ou configuração para tê-la funcionando.

As classes e estruturas (*structs*) básicas desta API são:

a) `Device`: representa o dispositivo em si em uma aplicação. Este objeto é o pai de qualquer outro. É ele quem gerencia todos os recursos do equipamento e possui os métodos necessários para renderizar o modelo. É também através deste objeto que alguns eventos (`OnPaint`, `OnDeviceReset`, etc) podem ser capturados e tratados de acordo com a situação;

b) `PresentParams`: controla o comportamento do objeto da classe `Device` na sua criação e durante a execução. Possui uma série de opções, sendo a `Windowed` e `SwapEffect` as mais usadas. A primeira delas define se a aplicação rodará em *full-screen* ou não. A segunda, por sua vez, está relacionada ao *back buffer*. Este *buffer* funciona da seguinte maneira: ao fazer chamadas às rotinas de renderização, normalmente as informações não são desenhadas diretamente na tela. Ao invés disso, são renderizadas em um local separado, chamado de *back buffer*. Quando todo o processo de desenho estiver pronto, o *back buffer* será copiado para a tela. Portanto, o `SwapEffect` controla o que deverá ser feito com este *buffer* após sua cópia (normalmente é descartado, ou seja, usa-se o valor `Discard` para este parâmetro);

c) `VertexBuffer`: mantém em memória um conjunto de dados que representa uma lista de vértices. A semântica das informações contidas no *buffer* varia conforme o FVF. Ele pode armazenar desde uma simples coordenada 2D ou 3D até coordenadas de texturas e vetores normais. Tendo em vista que não é possível acessar diretamente o *hardware* gráfico, este *buffer* possui um recurso chamado *lock*, que permite sua alocação para escrever as informações de forma rápida e simples. Seu uso consiste em uma seqüência de quatro operações: carregamento dos vértices, alocação do *buffer*, escrita das informações e liberação do *buffer*;

d) `IndexBuffer`: define de que forma os vértices do `VertexBuffer` estarão

interligados para formarem os objetos geométricos. Normalmente este *buffer* é especificado em um *array* de inteiros ou *short*, sendo que cada três elementos (na prática, um triângulo) representam uma face do objeto a ser desenhado. O mecanismo de *lock* descrito no item anterior também se aplica ao `IndexBuffer`.

e) `Mesh`: representa um modelo com todas as suas informações (vértices, faces, objetos, etc). Dentro do objeto `Mesh` ficam o `VertexBuffer` e o `IndexBuffer` de todo o modelo. Um arquivo OBJ (descrito na próxima seção), por exemplo, pode ser carregado inteiramente dentro de um único objeto `Mesh`. Cada grupo ou objeto processado torna-se um *SubSet*, que é representado internamente por um *array* de objetos do tipo `AttributeRange`;

f) `AttributeRange`: representa um subconjunto (*SubSet*) de objetos dentro de um `Mesh`. Cada `AttributeRange` pode ter suas próprias texturas e materiais. As principais propriedades de um *SubSet* são: `AttributeId`, que é um identificador único daquele subconjunto; `FaceCount`, que define quantas faces do `IndexBuffer` serão utilizadas para este objeto; `FaceStart`, que representa a primeira face do `IndexBuffer` utilizada no objeto; `VertexCount`, que define a quantidade de vértices do `VertexBuffer` que será usada pelo objeto; e, por fim, `VertexStart`, que representa qual é o primeiro vértice usado pelo objeto.

Com os elementos descritos acima é possível carregar na memória do dispositivo um modelo gráfico básico, com coordenadas de vértices, normais e texturas. No entanto, a representação dos materiais e dos arquivos de textura é feita utilizando os seguintes componentes:

a) `Material`: é uma estrutura (*struct*) que define as propriedades do material para um determinado objeto. Dentre estas propriedades as mais comuns são: `AmbientColor`, que representa a cor do ambiente; `DiffuseColor`, que representa a cor difusa; e `SpecularColor`, que representa a cor especular. Para cada objeto do modelo normalmente existe um material. A forma mais comum para que esse material influencie no processo de renderização, é torná-lo o material padrão do objeto `Device` para que o próximo objeto desenhado assuma as suas propriedades. Esta atribuição é facilmente obtida através da propriedade `Material` do objeto que representa o `Device`;

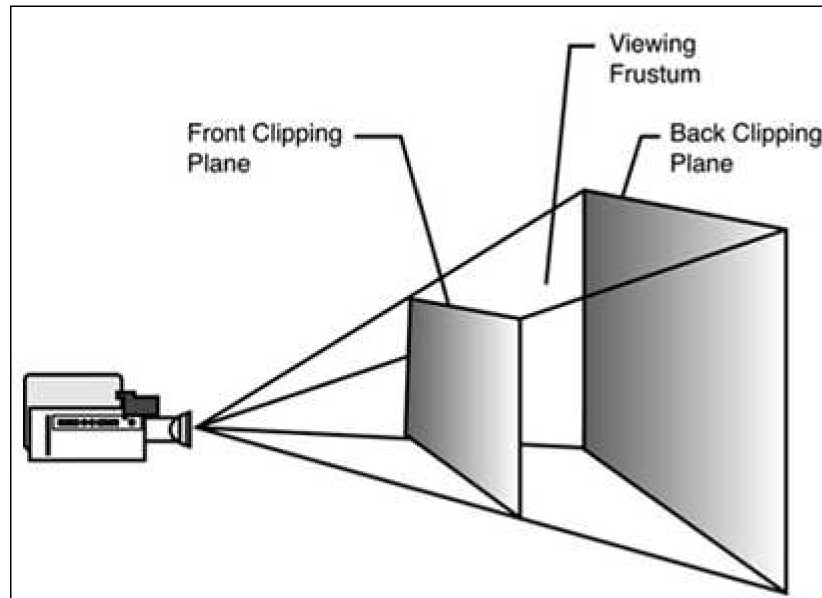
b) `Texture`: é a classe responsável por manipular a textura de um objeto. Normalmente os objetos deste tipo não são manipulados diretamente. Para isto,

existe a classe TextureLoader, que é responsável por carregar uma textura a partir de uma Stream (método FromStream) ou de um arquivo qualquer (método FromFile). Para que uma determinada textura seja aplicada a um objeto durante a renderização, usa-se o método SetTexture do Device. A partir da sua chamada, o funcionamento é exatamente o mesmo descrito anteriormente para a estrutura Material, ou seja, os objetos desenhados posteriormente serão afetados conforme a textura definida.

Além de todos estes artifícios para manipular o modelo, a API MD3DM também oferece vários recursos para a configuração de câmera e transformações no ambiente. Todos esses ajustes devem ser feitos na propriedade Transform do objeto Device através da definição de algumas matrizes 4x4. A classe Matrix, neste caso, é muito usada, pois possui uma série de métodos (tais como os apresentados nos itens abaixo) que geram essas matrizes de uma forma bem simples através de alguns parâmetros.

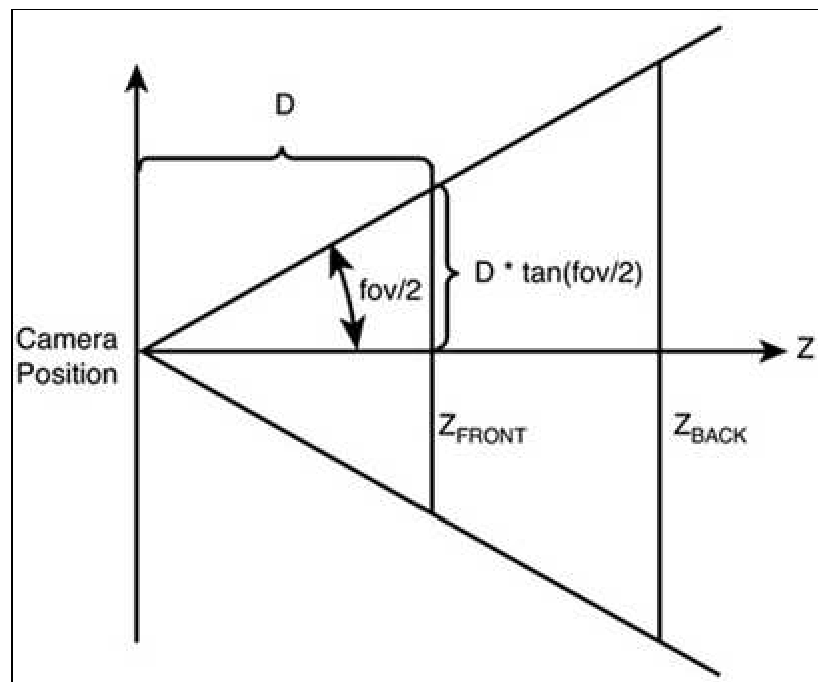
A propriedade Transform possui os seguintes membros (todos eles matrizes de tamanho 4x4):

a) View: armazena as informações do posicionamento e alvo da câmera. Seu valor é facilmente obtido através do método LookatLH da classe Matrix;
b) Projection: define o *View Frustum* da câmera. Este, por sua vez, é uma espécie de pirâmide na projeção em perspectiva que define o que será enxergado na cena (vide na figura 9). Esta matriz também é facilmente obtida através do método PerspectiveFovLH da classe Matrix. Outra definição muito importante que este método gera é o *Field of View* (campo de visão), que, em outras palavras, é o ângulo de abertura da câmera, conforme demonstrado na figura 10;
c) World: através desta matriz é possível executar uma série de transformações nos objetos do modelo. Qualquer operação de rotação, translação ou escala, por exemplo, deverá ser definida utilizando esta propriedade. A classe *Matrix* provê uma série de métodos para esta situação, tais como: RotationX, RotationY e RotationZ, que executam operações de rotação; Scaling, que define uma operação de escala; e Translation, que define uma operação de translação. Se for preciso executar uma série de transformações cumulativas sobre o modelo, basta multiplicar o valor atual da propriedade View pela nova matriz.



Fonte: Miller (2003).

Figura 9 – *View Frustum*



Fonte: Miller (2003).

Figura 10 – *Field of View*

Outro aspecto interessante da MD3DM é a forma com que ela trata as informações que estão na memória do dispositivo. Basicamente, todos os recursos utilizados por ela (`VertexBuffers`, `IndexBuffers`, matrizes de câmeras, etc) estão vinculados ao objeto `Device`, que por sua vez está diretamente ligado com a tela do dispositivo (isto pode ser observado logo na criação dele, pois recebe uma referência da tela que irá manipular). Logo, sempre que uma alteração é feita nesta tela (mudança de tamanho, por exemplo) os recursos alocados pelo MD3DM devem ser ajustados também. Porém, este ajuste depende da forma

com que os recursos foram criados. As duas principais são:

a) Default: os recursos devem ser realocados e também recarregados;

b) Managed: todos os recursos criados utilizando esta forma de armazenamento têm suas informações copiadas para a memória RAM do dispositivo. Com isso, eles não precisam ser realocados, somente recarregados.

A principal vantagem de se utilizar o método Managed é que o próprio MD3DM consegue realocar os recursos e, portanto, resta ao usuário recarregá-los.

Um dos pontos fracos desta API é que ela não possui nenhuma implementação nativa para carregar um modelo 3D a partir de arquivos. Em outras palavras, não há a possibilidade de ler e renderizar nativamente qualquer formato de arquivo com esta API. Este problema, por sua vez, é o tema principal deste trabalho.

2.6 O FORMATO DE ARQUIVOS *WAVEFRONT*

A especificação de um modelo 3D pode ser feita de várias formas, tais como: arquivos binários, arquivos XML e arquivos com texto puro. Cada uma destas possui vantagens e desvantagens, pois as características de armazenamento e a forma de carregamento do modelo variam de uma para a outra.

O formato *Wavefront* é especificado através de arquivos com texto puro, onde cada linha representa um item. Um modelo 3D definido neste padrão pode ser alterado em qualquer editor de textos simples.

Este formato é composto por dois arquivos. O primeiro deles é o *OBJ* (vide quadro 1) e possui basicamente os seguintes elementos:

a) #: representa um comentário de linha. Nenhuma informação até o final dela será considerada. Se for preciso utilizar mais linhas, cada uma delas deve possuir este caractere no início;

b) v <x> <y> <z>: define um vértice tridimensional com as coordenadas x, y e z. Cada vértice é enumerado sequencialmente a partir do 1. Esta enumeração será utilizada para formar as faces do modelo;

c) vn <x> <y> <z>: é um vetor tridimensional normal (vide figura 11) com as coordenadas x, y e z que identifica como será aplicada a iluminação sobre o

vértice. Da mesma forma que o item descrito anteriormente, o `vn` também é seqüencial e inicia a partir do número 1. Sua enumeração será utilizada para relacionar cada vértice com sua respectiva normal a fim de gerar as faces do modelo;

d) `vt <u> <v>`: também enumerado sequencialmente e a partir do número 1, as coordenadas de textura definem como ela será aplicada à face. O valor dos parâmetros normalmente é escalar, ou seja, varia de 0 até 1, conforme a figura 12;

e) `g <nome>`: representa um grupo dentro do modelo. Em um cubo, por exemplo, cada lado pode ser representado como um grupo separado, pois possui vértices e outras características independentes dos outros. O nome do grupo é um identificador único e muito útil para desenhar o modelo em partes, ou seja, pelos nomes dos grupos;

f) `f`: representa uma face do modelo. Este item é muito flexível, podendo assumir diversos formatos, dentre eles: `<v1> <v2> <v3>`, `<v1>/<vt1> <v2>/<vt2> <v3>/<vt3>` e, o mais completo de todos, `<v1>/<vt1>/<vn1> <v2>/<vt2>/<vn2> <v3>/<vt3>/<vn3>`. As faces normalmente são especificadas utilizando a técnica de triangulação, ou seja, através de três vértices, formando um triângulo. É neste item que os vértices, normais e texturas são relacionados;

g) `mtllib <arquivo>`: importa um determinado material para o modelo. O arquivo importado deve ter a extensão *mtl* e estar no formato *Wavefront MTL* (MCNAMARA, 2000), descrito posteriormente;

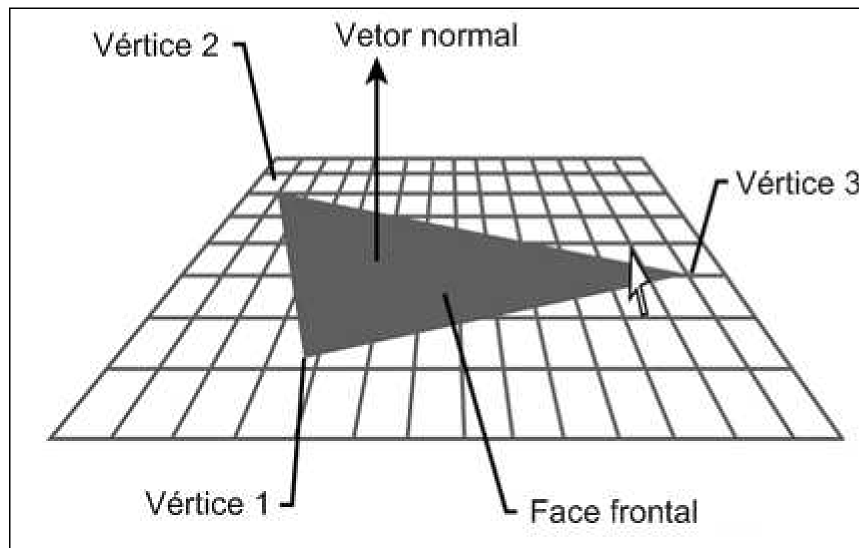
h) `usemtl <nome>`: depois que o material foi importado com o comando `mtllib`, para que ele faça algum efeito sobre o modelo em questão, o comando `usemtl` deve ser usado. Este, por sua vez, define que a partir da sua chamada todos os itens do modelo, até o fim do grupo atual, sofrem alterações de luz, textura e outros, conforme o material selecionado. Normalmente o `usemtl` é usado após a definição de um grupo (quadro 2).

```
#comentário
v  <x> <y> <z>
vn <x> <y> <z>
vt <u> <v>
g <nome>
mtllib <arquivo>
usemtl <material>
f <v1>/<vt1>/<vn1> <v2>/<vt2>/<vn2>
<v3>/<vt3>/<vn3>
```

Quadro 1 – Formato do arquivo *OBJ*

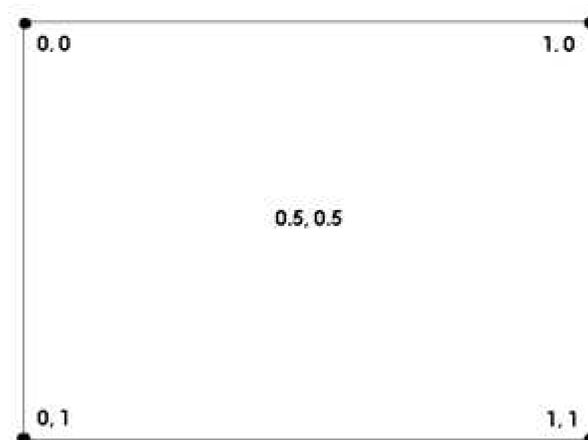
```
mtllib teste.mtl
v  0.000000 2.000000 0.000000
v  0.000000 0.000000 0.000000
v  2.000000 0.000000 0.000000
v  2.000000 2.000000 0.000000
vt 0.000000 1.000000 0.000000
vt 0.000000 0.000000 0.000000
vt 1.000000 0.000000 0.000000
vt 1.000000 1.000000 0.000000
g grupo1
usemtl material1
f 1/1 2/2 3/3 4/4
```

Quadro 2 – Exemplo de uso do comando usemtl



Fonte: Miller (2003).

Figura 11 – Vetor normal



Fonte: Miller (2003).

Figura 12 – Mapeamento da textura

O outro arquivo que compõe o formato *Wavefront* é o *MTL* (vide quadro 3). Seus principais elementos são:

a) `#`: representa um comentário de linha, conforme já descrito;
b) `newmtl <nome>`: define um novo material. Todos os itens que vierem depois deste comando e antes do próximo farão parte de um único material;

c) `Ka`: define a cor ambiente;

d) `Kd`: define a cor difusa;

e) `Ks`: define a cor especular;

f) `d`: indica o valor *alpha* do material, ou seja, o seu nível de transparência;

g) `map_Ka`: importa uma imagem que representa a textura afetada pela cor ambiente do material;

h) `map_Kd`: importa uma imagem que representa a textura afetada pela cor difusa do material;

i) `map_Ks`: importa uma imagem que representa a textura afetada pela cor especular do material.

```
#comentário
newmtl <nome>
Ka <r> <g> <b>
Kd <r> <g> <b>
Ks <r> <g> <b>
d <alpha>
map_Ka
<arquivo>
map_Kd
<arquivo>
map_Ks
<arquivo>
```

Quadro 3 – Formato do arquivo *MTL*

Nos quadros 4 e 5 é apresentado um exemplo de um cubo especificado usando o padrão *Wavefront*. A figura 13 apresenta o modelo renderizado sem modificação alguma. Já a 14 demonstra o mesmo modelo desenhado após algumas rotações, escalas e a adição de uma luz.

```
mtllib material.mtl
g cubo
v   1.000000  1.000000 -1.000000
v   1.000000 -1.000000 -1.000000
v  -1.000000  1.000000 -1.000000
v  -1.000000  1.000000 -1.000000
v   1.000000  1.000000  1.000000
v  -1.000000 -1.000000  1.000000
v  -1.000000  1.000000  1.000000
vn  0.000000  1.000000  0.000000
vn -1.000000  0.000000  0.000000
vn  0.000000 -1.000000  0.000000
vn  1.000000  0.000000  0.000000
vn  1.000000  0.000000  0.000000
vn  0.000000  0.000000  0.000000
vn  0.000000  0.000000  0.000000
usemtl Material
f 5//1 1//1 4//1
f 5//1 4//1 8//1
f 3//2 7//2 8//2
f 3//2 8//2 4//2
f 2//3 6//3 3//3
f 6//3 7//3 3//3
f 1//4 5//4 2//4
f 5//5 6//5 2//5
f 5//6 8//6 6//6
f 8//6 7//6 6//6
f 1//7 2//7 3//7
f 1//7 3//7 4//7
```

Quadro 4 – Exemplo de um cubo: arquivo *OBJ*

```
newmtl Material
Ka 0.565000 0.875000
0.141000
d 1.000000
```

Quadro 5 – Exemplo de um cubo: arquivo *MTL*

Figura 13 – Exemplo apresentado nos quadros 4 e 5 sendo renderizado sem modificações

Figura 14 – Exemplo dos quadros 4 e 5 sendo renderizado após algumas rotações, escalas e a adição de uma luz

2.7 TRABALHOS CORRELATOS

Existem vários trabalhos que estão relacionados com o desenvolvimento desta aplicação. Entretanto, somente dois foram estudados. Ambos são considerados motores de jogos para celulares, ou seja, ferramentas que tornam o desenvolvimento de jogos um processo automatizado, padronizado, rápido e permitem a exibição de modelos 3D. São eles: Mobile Graphics Engine (mOGE) (MACEDO JÚNIOR, 2005) e Mobile 3D Game Engine (M3GE) (PAMPLONA, 2005).

2.7.1 mOGE

Segundo Macedo Júnior (2005, p. 4), mOGE é um motor gráfico 3D voltado ao desenvolvimento de jogos para dispositivos móveis. Ele foi inspirado em outros motores gráficos existentes para computadores e celulares, portanto implementa diversas funcionalidades conhecidas, tais como:

a) gerenciador de entrada: identifica os eventos de entrada do dispositivo e encaminha para outro módulo executar o processamento;
b) gerenciador de inteligência artificial: gerencia o comportamento de objetos

controlados pela máquina;

c) gerenciador de objetos: armazena em alguma estrutura de dados e controla o ciclo de vida dos objetos presentes no jogo;

d) gerenciador de mundo: armazena o estado atual do jogo.

Este motor utiliza a biblioteca OpenGL ES e o sistema operacional Symbian. A figura 15 apresenta alguns resultados obtidos neste projeto.

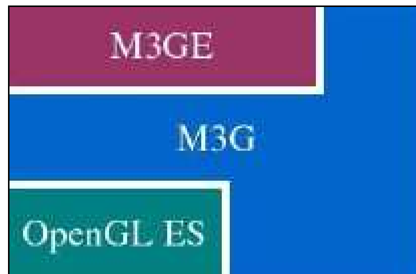
Fonte: Macedo Júnior (2005, p. 42).

Figura 15 – Resultados obtidos com a mOGE

2.7.2 M3GE

De acordo com Pamplona (2005, p. 37), M3GE é um motor de jogos escrito em Java e baseado na *Mobile 3D Graphics API* (M3G) (NOKIA, 2003). Dentre outras implementações, esta biblioteca suporta detecção de colisão, controla a entrada e saída de informações e permite a definição de até quatro câmeras para navegar pelo modelo carregado. Além disso, segue uma especificação definida pela *Java Community Process* (JCP) (JCP, 2004a), a *Java Specification Request* (JSR) 184 (JCP, 2004b). Esta e outras especificações da JCP determinam padrões de desenvolvimento, novas implementações e revisões sobre diversos assuntos e tecnologias. Por isto, são utilizadas como referência por diversos desenvolvedores.

A M3GE foi projetada para ser utilizada em conjunto com a M3G (figura 16). Ou seja, as duas bibliotecas interagem entre si. Isso proporciona ao desenvolvedor do jogo flexibilidade e velocidade quando for preciso (PAMPLONA, 2005, p. 39).



Fonte: Pamplona (2005, p. 39).
Figura 16 – Visão geral da M3GE

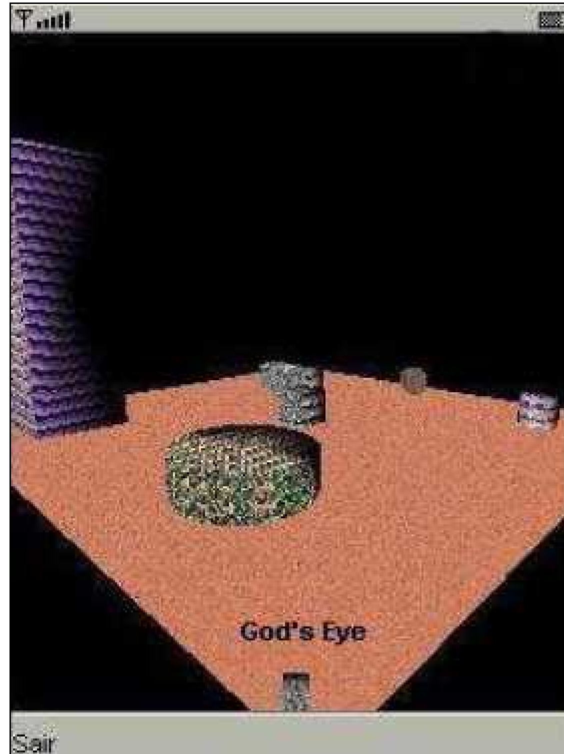
Ainda segundo Pamplona (2005, p. 39), este projeto está dividido em dois grandes componentes: o responsável pela leitura dos arquivos (modelos 3D) e o motor de jogos em si (*core*). É sobre o *core* que as implementações dos desenvolvedores devem ficar, ou seja, o enredo e a lógica do jogo são implementados sobre ele.

Até o final do presente trabalho, a M3GE suportava os seguintes formatos de modelos 3D:

a) *Wavefront*: é o mesmo formato suportado pela Gita e que está descrito na fundamentação teórica deste trabalho;
b) *Mobile Object File* (MBJ): tendo em vista que o formato *Wavefront* não é otimizado para dispositivos limitados, o autor da M3GE criou um formato específico para o seu trabalho. Algumas das alterações definidas por ele são: os tipos de dados já devem estar convertidos à faixa de valores em *byte,* o cálculo do ponto central já deve estar realizado para cada grupo, e, por fim, os tamanhos dos *arrays* de vértices, normais, texturas e faces já devem ser informados logo no início do arquivo.

Foram realizados testes com um telefone celular Siemens CX65 (SIEMENS AG, 2005a) e com os emuladores do *Siemens Mobility Toolkit* (SIEMENS AG, 2005b). Os resultados obtidos foram praticamente os mesmos. O tempo de leitura de um modelo 3D, por exemplo, foi o mesmo nas duas situações e a velocidade durante o jogo ficou em torno de 4 a 12 *frames* por segundo para movimentação e 15 a 20 *frames* por segundo para rotação do personagem, onde não existe teste de colisão.

A figura 17 ilustra o resultado obtido com a M3GE. Nela pode-se observar a utilização de uma câmera que possibilita a visualização completa do modelo. Além disso, todos os objetos desenhados possuem textura e há uma definição de luzes para gerar a sombra.



Fonte: Pamplona (2005, p. 75).

Figura 17 – Resultado obtido com a M3GE

3 DESENVOLVIMENTO DO TRABALHO

O presente trabalho, por ser focado em equipamentos que possuem recursos limitados, foi desenvolvido seguindo duas máximas:

a) YAGNI (FOWLER, 2000): é uma prática muito difundida entre os seguidores da metodologia XP. Ela diz o seguinte: sempre implemente as coisas que você precisa atualmente, nunca as que você acha que vai precisar.
b) KISS (ALMGVIST, 2001): a filosofia por trás desta sigla é complementar a anterior: faça de forma simples o que você precisa.

Colocar em prática estes dois conceitos nem sempre é uma tarefa fácil. Aliando eles às limitações dos dispositivos em questão, vários artifícios pertinentes às boas práticas de programação precisam ser ignorados.

A programação orientada a objetos, por exemplo, provê diversos recursos para o desenvolvimento de aplicações. Porém, uma série deles foi descartada durante o desenvolvimento, tais como: alguns padrões de projeto mais complexos, heranças demasiadas, sobrecarga excessiva de métodos, etc. Todos estes itens fazem com que o desempenho seja pior e, não menos importante, o tamanho da aplicação cresça além do aceitável para tais dispositivos.

Nas próximas seções são listados os detalhes do problema, as ferramentas e tecnologias usadas no processo de desenvolvimento e, por fim, a solução proposta.

3.1 REQUISITOS PRINCIPAIS

A aplicação deverá contemplar os seguintes requisitos:

a) carregar um modelo tridimensional no formato *Wavefront*;
b) permitir a navegação em primeira pessoa pelo cenário a partir do teclado do dispositivo;
c) permitir que o usuário execute operações de translação, rotação e escala sobre o modelo;
d) ser portável entre os dispositivos móveis, característica oferecida pelo .NET CF;

3.2 FERRAMENTAS E TECNOLOGIAS UTILIZADAS

O trabalho foi desenvolvido utilizando a IDE Visual Studio 2005 (MICROSOFT, 2004) rodando em um *notebook* com processador AMD Turion 1.6GHz e 1GB de memória RAM. Foram utilizados os SDKs para desenvolvimento de aplicações para dispositivos móveis que suportam o Windows Mobile 5.0 ou 6.0. Estes *kits* de desenvolvimento fornecem uma extensa biblioteca de componentes visuais, classes e alguns emuladores, tais como os apresentados na figura 18.

Figura 18 – Emuladores do Windows Mobile 5.0 e 6.0, respectivamente

Os testes em um dispositivo real foram feitos utilizando o HTC S620 (HTC, 2007). Ele possui um processador de 200MHz e 64MB de memória RAM.

Além disso, o .NET CF versão 2.0 e a biblioteca MD3DM, já descritos anteriormente, também foram usados durante o desenvolvimento do trabalho. Todas estas tecnologias são integradas, ou seja, não foi preciso desenvolver código extra para utilizá-las em conjunto.

No que diz respeito às ferramentas de modelagem, foram utilizadas duas: Enterprise Architect e o próprio Visual Studio. Esta última, além de possibilitar o desenvolvimento de todo o projeto, facilitou a geração dos diagramas de classes que são apresentados neste documento de tal forma que todo o trabalho de modelagem pôde ser feito diretamente a partir da IDE de desenvolvimento. Infelizmente ela só suporta este tipo de diagrama, por isso o Enterprise Architect foi usado para suprir as necessidades que restavam (diagrama de seqüência e componentes).

Para a modelagem 3D foram utilizadas as ferramentas Art of Illusion (EASTMEN, 2005) e uma versão *trial* do 3D Studio Max (AUTODESK, 2006a). Ambas suportam o formado *Wavefront* e possuem diversos recursos explorados por este trabalho, tais como: formas geométricas pré-definidas, possibilidade de definir diversas câmeras para visualizar a mesma cena, aplicação rápida e simples de materiais e texturas, etc.

3.3 GITA E 3DV

O presente trabalho trata da criação de um *framework* para a manipulação de modelos 3D em dispositivos móveis. Este projeto, chamado Gita, é responsável por carregar os modelos e possui suporte ao posicionamento de câmeras e transformações.

Além disso, este trabalho também apresenta um protótipo chamado 3D Viewer para validar o *framework*. Estes dois projetos juntamente com o .NET CF 2.0, a API MD3DM e o Windows Mobile estão relacionados conforme a figura 19 e foram desenvolvidos utilizando a linguagem C#.

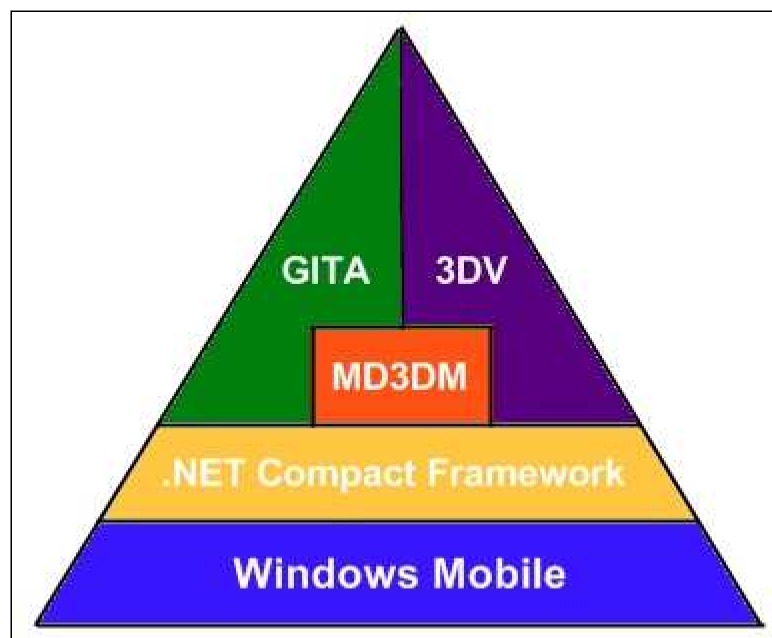


Figura 19 – Relacionamento entre os componentes do projeto

A Gita é utilizada pelo 3DV e ambos fazem acesso direto à biblioteca MD3DM e ao .NET CF. Isso permite uma certa flexibilidade ao desenvolvedor, pois se for preciso criar alguma rotina não suportada pela biblioteca ou visualizador, todas as classes necessárias estão acessíveis. As rotinas do sistema operacional somente serão acessadas através do .NET CF.

3.3.1 GITA

Este projeto, ou *Class Library* (biblioteca de classes), como é definido pelo Visual Studio, implementa todos os recursos necessários para carregar o modelo 3D no formato *Wavefront*, navegar através dele e executar alterações utilizando as transformações disponíveis.

Apesar de prover algumas funcionalidades pré-definidas, nada impede que o desenvolvedor opte por acessar diretamente a API MD3DM para criar suas próprias rotinas. Isto é indispensável, pois a Gita é limitada, tendo em vista que é um estudo pioneiro com a MD3DM.

A arquitetura da Gita está dividida em quatro grandes componentes (vide figura 20). Cada um deles possui responsabilidades específicas e é independente dos outros, exceto o `Common`, pois ele disponibiliza uma série de funcionalidades básicas utilizadas pelos demais.

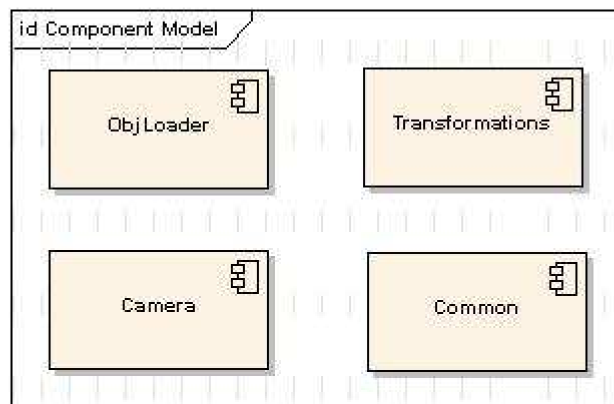


Figura 20 – Componentes da biblioteca Gita

3.3.1.1 ObjLoader

O componente `ObjLoader` encapsula todas as funcionalidades necessárias para carregar o modelo no formato *Wavefront*. Como descrito anteriormente, este modelo é representado por dois arquivos: o *OBJ* e o *MTL*. Logo, a API também trata estes dois arquivos separadamente, ou seja, existem classes responsáveis por carregar o conteúdo do arquivo *OBJ* e outras, totalmente independentes, que carregam o *MTL*. A figura 21 apresenta as classes do `ObjLoader`.

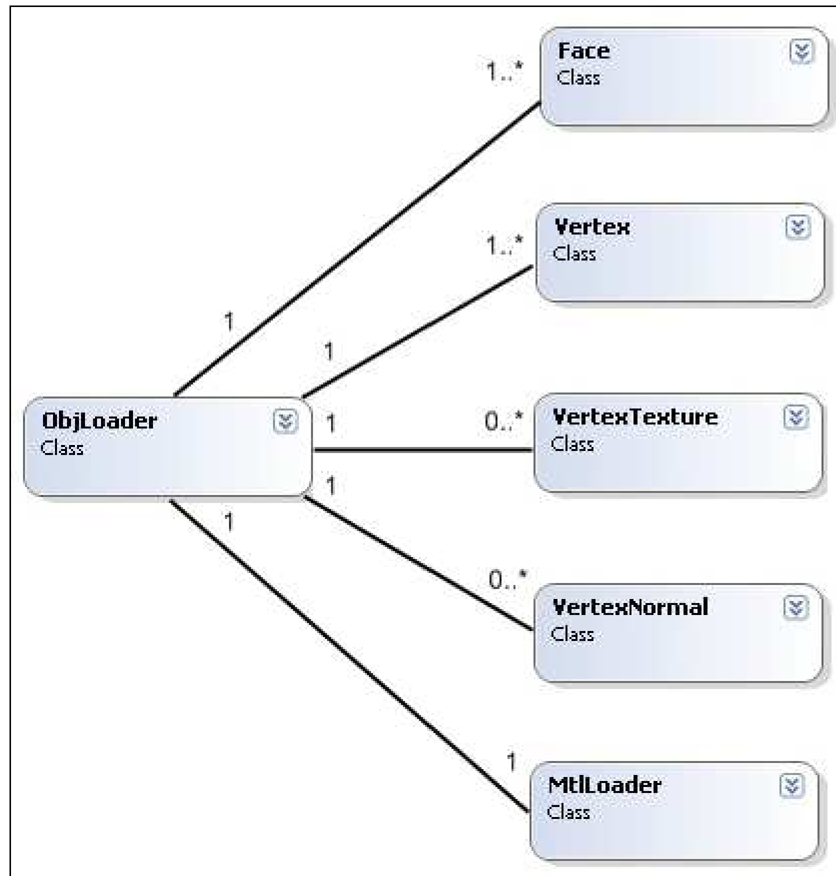


Figura 21 – Classes que compõem o `ObjLoader`

O arquivo *OBJ* é processado pelas classes `ObjLoader`, `Vertex`, `VertexNormal`, `VertexTexture` e `Face`, conforme descrito abaixo:

a) `ObjLoader`: esta classe gerencia todo o carregamento do modelo 3D. Através do método `LoadMesh` o arquivo no formato *Wavefront* é lido e carregado na memória do dispositivo. Suas propriedades e métodos são apresentados na figura 22. A carga do modelo é feita em duas etapas representadas pelos métodos `FirstStep` e `SecondStep`. Na primeira delas as seguintes operações são realizadas: leitura das coordenadas de vértices, normais e texturas usando as classes `Vertex`, `VertexNormal` e `VertexTexture`, respectivamente; cálculo do *bouding box*[3] para servir como base no posicionamento da câmera; verificação do número de vértices e faces, sendo que deve haver ao menos três vértices e uma face para que o modelo seja considerado válido; e, por fim, são alocados os *arrays* de vértices e faces. O `SecondStep`, por sua vez, carrega as faces, grupos, materiais e texturas. Além disso, também define os valores dos *arrays* de vértices e faces que foram criados anteriormente. No fim do processo, cria e carrega um objeto `Mesh` com todas as

---

[3] *Bouding box* é o paralelepípedo mínimo que envolve o modelo.

informações lidas;

b) `Vertex`: classe que lê as informações (coordenadas `x`, `y` e `z`) de um determinado vértice (elemento `v` do arquivo lido). Sua definição é apresentada na figura 23;

c) `VertexNormal`: classe que lê as coordenadas `x`, `y` e `z` referentes a um elemento `vn` (vetor normal) do arquivo em questão. Está definida na figura 24;

d) `VertexTexture`: responsável pela leitura das coordenadas `u` e `v` de um elemento `vt` (textura). A figura 25 apresenta a sua definição;

e) `Face`: da mesma forma que as três classes acima lêem informações referentes aos elementos `v`, `vt` e `vn`, esta, por sua vez, é responsável pela leitura de tudo o que diz respeito ao elemento `f` (face). As faces, como descritas anteriormente, podem assumir diversas formas. A implementação feita neste trabalho suporta todas elas, no entanto a partir do momento que um padrão é utilizado, os demais elementos devem seguir o mesmo formato, caso contrário o modelo não será carregado. A definição desta classe é apresentada na figura 26.

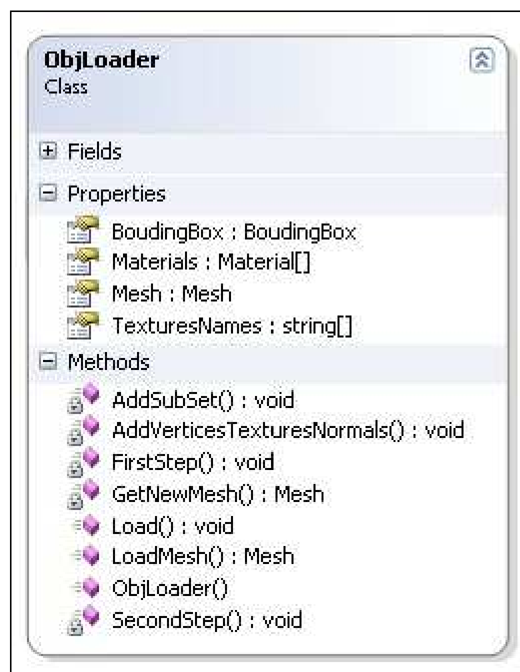


Figura 22 – Propriedades e métodos da classe `ObjLoader`

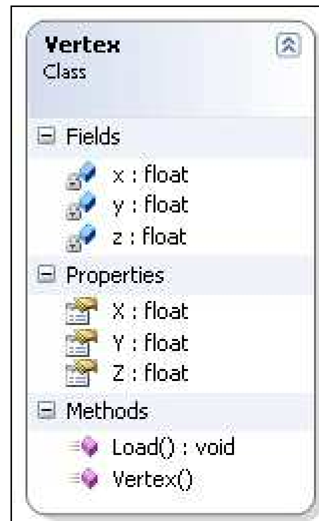


Figura 23 – Propriedades e métodos da classe `Vertex`

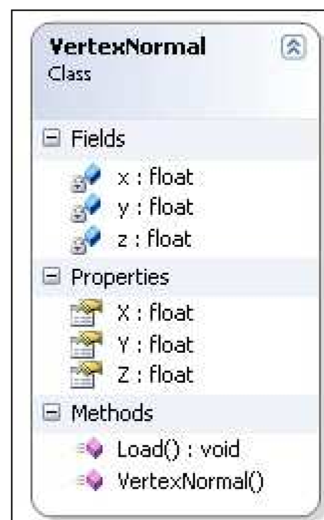


Figura 24 – Propriedades e métodos da classe `VertexNormal`

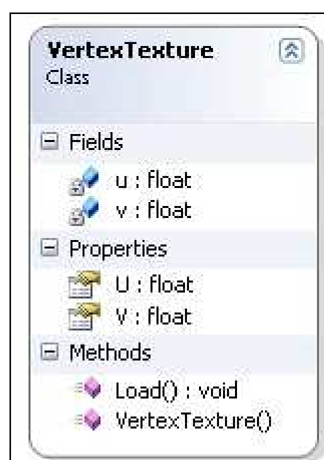


Figura 25 – Propriedades e métodos da classe `VertexTexture`

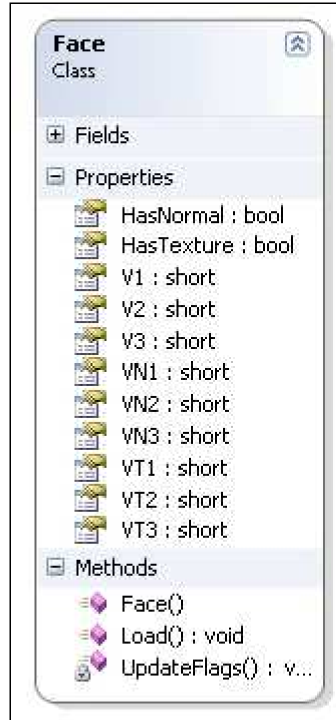


Figura 26 – Propriedades e métodos da classe `Face`

A lista de vértices do `VertexBuffer` (descrito no item 2.5) é carregada conforme os objetos `Vertex`, `VertexTexture` e `VertexNormal` definidos pelo `ObjLoader`. Cada item desta lista é composto por uma coordenada de vértice, normal e textura. Este formato (chamado de FVF e já descrito no item 2.5) é definido conforme o quadro 6.

```
private Mesh GetNewMesh()
{
    VertexFormats fvf = VertexFormats.Position |
                        VertexFormats.Normal |
                        VertexFormats.Texture1;

    Mesh mesh = new Mesh(indexBuffer.Length / 3,
                         vertexBuffer.Length / 8,
                         0,
                         fvf,
                         device);

    // ...
}
```

Quadro 6– Definição do FVF

O arquivo *MTL*, que armazena os materiais e texturas utilizados para alterar a aparência do modelo 3D, é carregado pela classe `MtlLoader`, definida na figura 27.

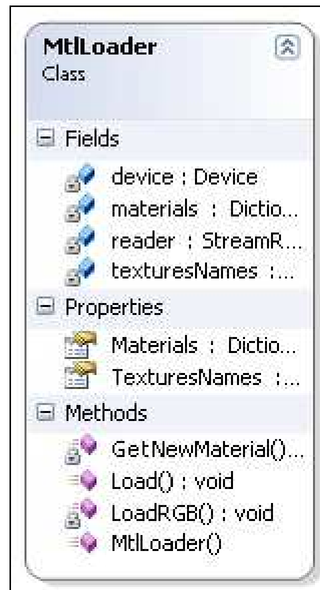


Figura 27 – Propriedades e métodos da classe `MbjLoader`

Uma instância dessa classe é criada sempre que o elemento *mtllib* é encontrado no arquivo *OBJ*. A partir deste momento, o objeto `MtlLoader` faz a leitura de todas as definições que estão no arquivo *MTL*. Posteriormente, quando o `ObjLoader` encontrar um elemento *usemtl*, este objeto é consultado para retornar o determinado material e textura que devem ser usados.

3.3.1.2 Transformations

Este componente é responsável por realizar as transformações de rotação, translação e escala nos objetos do modelo. Ele é composto pelas classes `Params`, `TransformationFactory`, `Rotation`, `Translation` e `Scale`. Além disso, também possui a enumeração `TransformKind` e a interface `ITransformation`. Todos estes itens estão representados na figura 28.

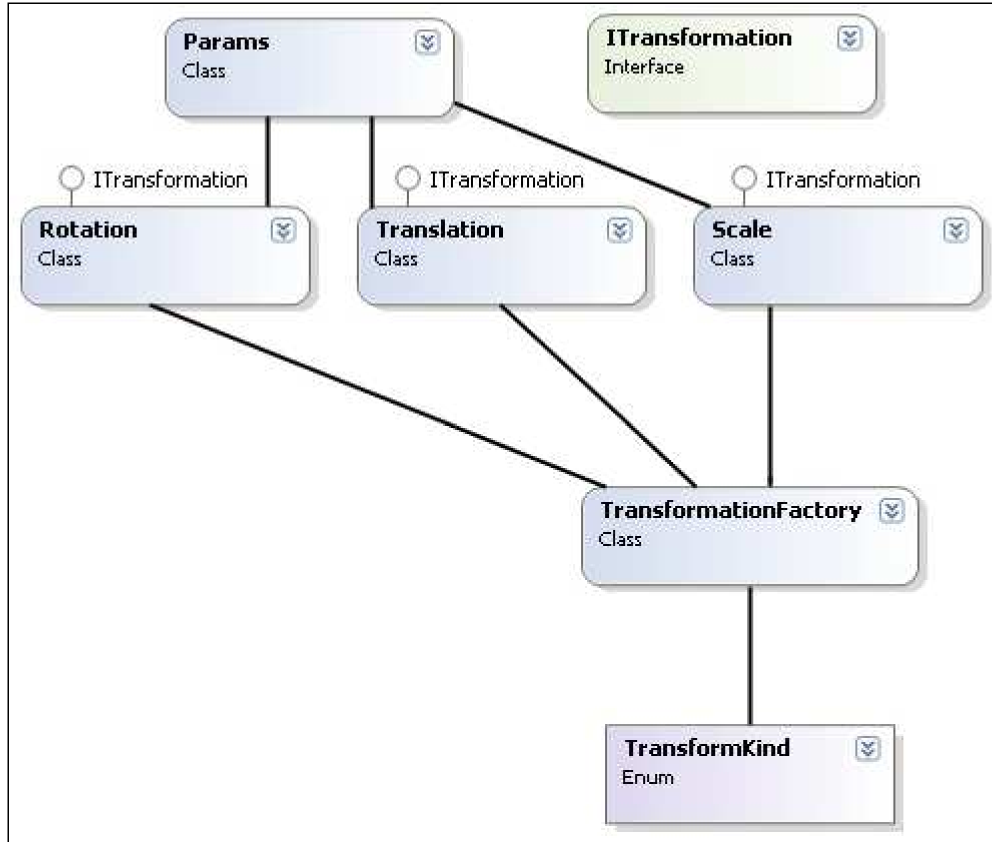


Figura 28 – Componente `Transformations`

A interface `ITransformation` especifica os métodos que deverão ser implementados pelas três classes de transformação. Seus membros estão descritos na figura 29.

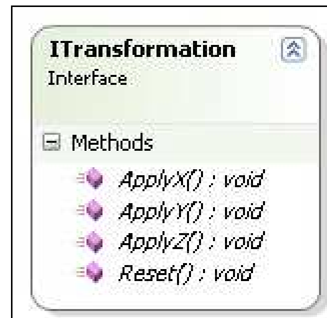


Figura 29 – Interface `ITransformation`

Cada um dos métodos `Apply` define um novo valor à matriz de transformação para a sua respectiva coordenada (x, y ou z). Esta definição só é possível porque os objetos de transformação possuem uma referência ao `Device` atual. Desta forma, eles podem alterar diretamente a matriz `World`, que, conforme visto anteriormente, é responsável pelas transformações no modelo.

Os valores aplicados por cada um dos métodos acima estão definidos na classe `Params`, conforme a figura 30.

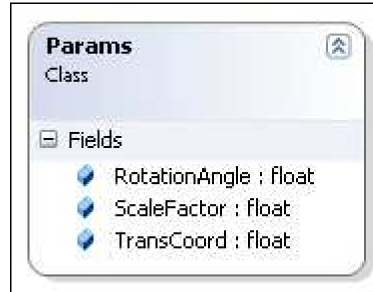


Figura 30 – Classe `Params`

Em outras palavras, `ApplyX` da classe `Rotation` aplicará uma rotação de `RotationAngle` sobre o eixo x. Já o método `ApplyY` da classe `Translation`, fará a translação de `TransCoord` sobre o eixo y. E por fim, o método `ApplyZ` da classe `Scale`, aplicará o fator de escala definido na variável `ScaleFactor` sobre o eixo z.

Tendo em vista que as operações descritas acima são usadas com muita freqüência, foi feita uma implementação que garante que um, e somente um objeto de cada classe estará disponível durante todo o ciclo de vida da aplicação. Para isto, o padrão de projeto *Singleton* foi utilizado. Segundo Gamma et al (1995, p. 127), este padrão garante que somente uma instância de determinada classe existirá no contexto da aplicação. Todas as três classes (`Rotation`, `Scale` e `Translation`) implementam este padrão no método `GetInstance` e definindo todos os construtores da classe como privados. Desta forma, é impossível obter uma instância destes objetos sem chamar o método `GetInstance`.

Junto com o *Singleton*, outro padrão também foi utilizado para facilitar a criação destes objetos: o *Factory Method*. Este, por sua vez, define um método capaz de criar objetos, que implementam os mesmos comportamentos, conforme o contexto (GAMMA et al., 1995 p. 107). O contexto, neste caso, é definido pela enumeração (*enum*) `TransformKind`, descrita na figura 31, e a classe que implementa *o Factory Method* está na figura 32.

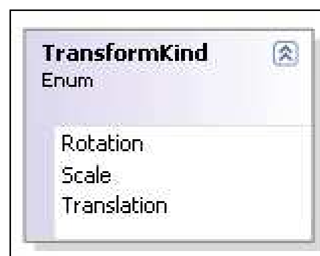


Figura 31 – Enumeração `TransformKind`

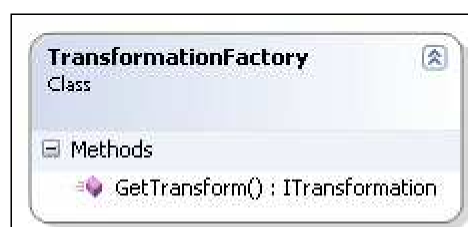


Figura 32 – Classe `TransformationFactory`

Estes padrões foram cuidadosamente escolhidos para solucionarem os problemas em questão. Ambos não demandam de muitos recursos de processamento e das técnicas de orientação a objetos. Logo, podem ser usados sem prejudicar o desempenho da aplicação.

3.3.1.3 Camera

Dentro deste componente ficam todas as classes necessárias para definir as câmeras que a aplicação utilizará. Cada arquivo *OBJ* pode estar associado a outro que define uma série de câmeras que deverão ser carregadas ao ler o modelo. Este outro arquivo é especificado utilizando a linguagem de marcação XML e deve ter exatamente o mesmo nome do arquivo *OBJ*, porém com a extensão XML. Sua definição é apresentada no quadro 7.

```
<gita>
  <cameras>
      <add name="cam1" position="x,y,z" target="x,y,z″ up="x,y,z" />
      <add name="cam2" position="x,y,z" target="x,y,z″ up="x,y,z" />
      <add name="camN" position="x,y,z" target="x,y,z″ up="x,y,z" />
  </cameras>
</gita>
```

Quadro 7 – Formato do arquivo de especificação das câmeras

Se este arquivo não existir, então a classe responsável pela carga das câmeras irá criar uma padrão e sua configuração será definida através do *bouding box* do modelo carregado.

As classes responsáveis pela representação e leitura das informações do arquivo XML são as seguintes:

a) `Camera`: classe que armazena na memória do dispositivo todos os parâmetros de uma determinada câmera. É com base nas informações deste objeto que as matrizes `View` e `Projection`, descritas anteriormente, serão definidas quando for preciso renderizar os objetos. Suas propriedades são apresentadas na figura 33;
b) `CameraList`: esta classe gerencia todas as câmeras utilizadas para visualizar a cena. É através dela que o arquivo XML que está vinculado ao *OBJ* é lido. A figura 34 apresenta a sua definição. A propriedade `Cameras` armazena todas as câmeras lidas través do método `ReadCameras`. Este, por sua vez, lê o arquivo e cria vários objetos `Camera`. Após a leitura, a última câmera será definida como a atual, que é representada através da propriedade `Current`. Porém, como mencionado anteriormente, este arquivo não é obrigatório, e caso não exista, uma câmera padrão com base no *bouding box* do modelo será criada.

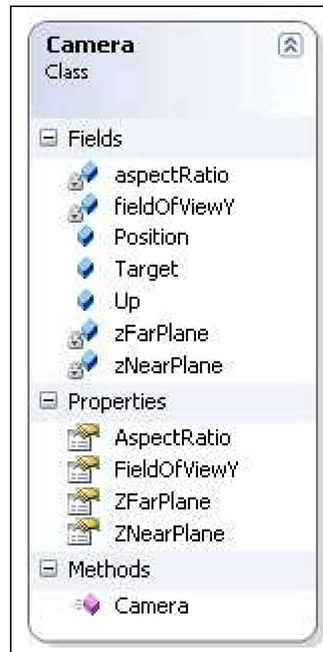


Figura 33 – Classe `Câmera`

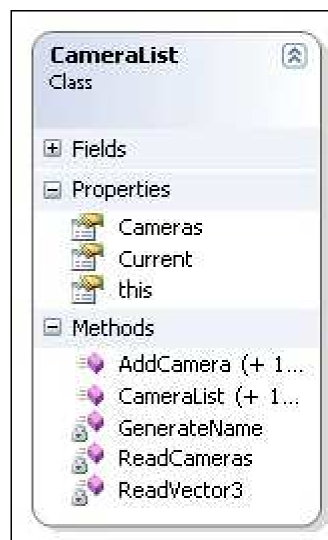


Figura 34 – Classe `CameraList`

3.3.1.4 Common

Este componente encapsula funcionalidades básicas com o intuito de facilitar o desenvolvimento dos outros três e também fornecer ao 3DV algumas rotinas auxiliares para a visualização do modelo. A classe `Utils`, por exemplo, define alguns métodos que são utilizados em diversos locais da biblioteca. Um deles, chamado `ReadValue`, é utilizado para ler os parâmetros dos comandos *v*, *vt* e *vn* do arquivo *OBJ*.

A classe `BoudingBox`, demonstrada na figura 35, serve para representar o *bouding box*

de um modelo ou objeto.

Figura 35 – Classe `BoudingBox`

Como descrito no item anterior, as classes que fazem parte do componente `Camera` utilizam esta implementação para posicionar uma câmera padrão caso nenhuma seja definida nos arquivos de configuração.

3.3.2 3D Viewer

O 3DV é o protótipo desenvolvido com a finalidade de validar a biblioteca Gita. Ele utiliza todos os recursos disponíveis nela para renderizar e manipular alguns modelos 3D. Suas funções, apresentadas na figura 36 e acessíveis através da tecla *Enter* do dispositivo, são as seguintes:

a) *Load file*: abre uma janela para o usuário selecionar o arquivo no formato *OBJ*. Como descrito anteriormente, não existe uma implementação padrão no .NET CF para abrir esta janela. Portanto, foi utilizada a classe `OpenFileDialog` (figura 37), que está disponível no *framework* OpenNETCF. Foi definido um diretório padrão para localizar os modelos (*\Program Files\threedv\Models*), mas ele pode ser alterado através da opção *Change Folder* que está dentro do *Menu*. Depois que o arquivo foi selecionado a aplicação já tenta carregá-lo. Em caso de sucesso, será renderizado na tela, caso contrário o usuário será informado sobre o problema e poderá selecionar outro modelo;

b) *Transform*: disponibiliza ao usuário as três transformações básicas: translação,

rotação e escala (vide figura 38). Assim que uma delas é selecionada, o visualizador envia o usuário de volta à tela inicial, onde o modelo está sendo renderizado, e então ele poderá aplicar a transformação selecionada utilizando as teclas *Left*, *Right*, *Up* e *Down* do dispositivo;

c) *Navigate*: nesta opção o usuário poderá definir qual das câmeras ele utilizará para visualizar o modelo (vide figura 39). Fazendo isto, automaticamente estará habilitando a função de navegação, que permite que ele navegue pela cena carregada na tela.

d) *Reset*: esta opção serve para redefinir todas as opções conforme estavam logo após a carga do modelo. Na prática, é como se o usuário selecionasse *Load file* e buscasse o modelo atual novamente. No entanto, esta opção chama métodos da Gita que são responsáveis por reiniciar as transformações e também o posicionamento das câmeras sem efetuar a recarga do modelo.

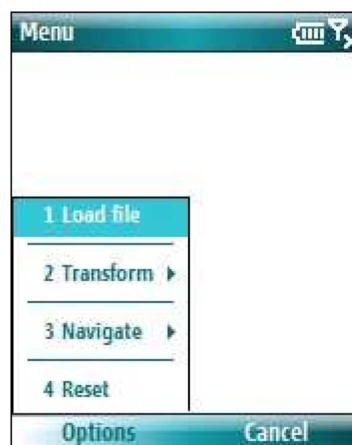


Figura 36 – Funções do 3D Viewer

Figura 37 – Janela de pesquisa dos arquivos *OBJ*

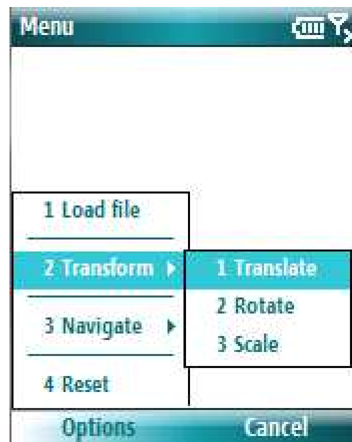


Figura 38 – Transformações disponíveis no menu *Transform*

Figura 39 – Escolha da câmera para navegação

Todas as funcionalidades apresentadas acima estão disponíveis na classe `Menu`. Esta, por sua vez, é coordenada pela `Viewer`, que é a principal classe do sistema e está representada na figura 40, juntamente com as outras classes que compõem o visualizador. Nela é que o modelo 3D será renderizado.

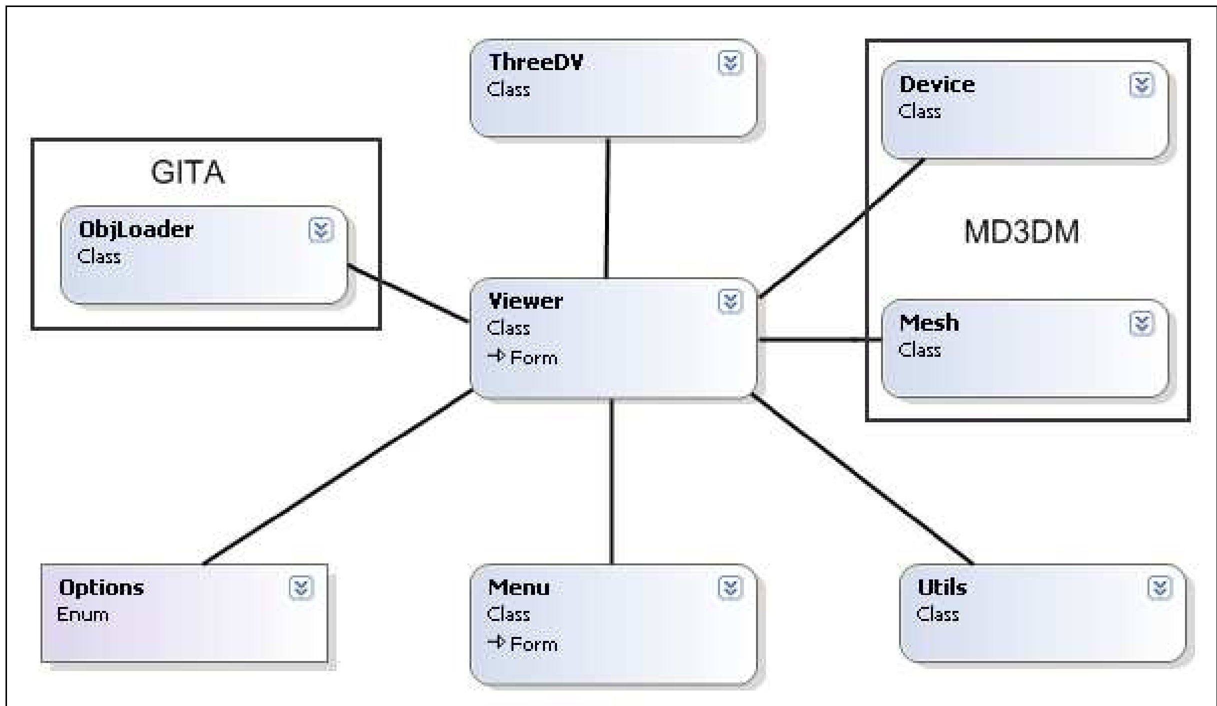


Figura 40 – Classes que compõem o visualizador e o relacionamento com a Gita e MD3DM

Infelizmente não há muita flexibilidade na tela que renderizará o modelo, havendo uma série de restrições neste ponto, tais como:

a) não é possível utilizar um `Panel` para desenhar o modelo 3D;

b) somente a partir da tela criada com o método `Run` da classe `Application` é que o objeto `Device` pode ser inicializado;

c) não é possível utilizar um menu na mesma tela que será usada para desenhar o modelo, pois ao criar o `Device` ele é que terá o controle sobre esta tela. Por isso é que o menu está acessível a partir da tecla *enter* e uma tela totalmente separada.

Apesar destas limitações, é perfeitamente possível renderizar os modelos carregados. Este processo é descrito pelo diagrama da figura 41.

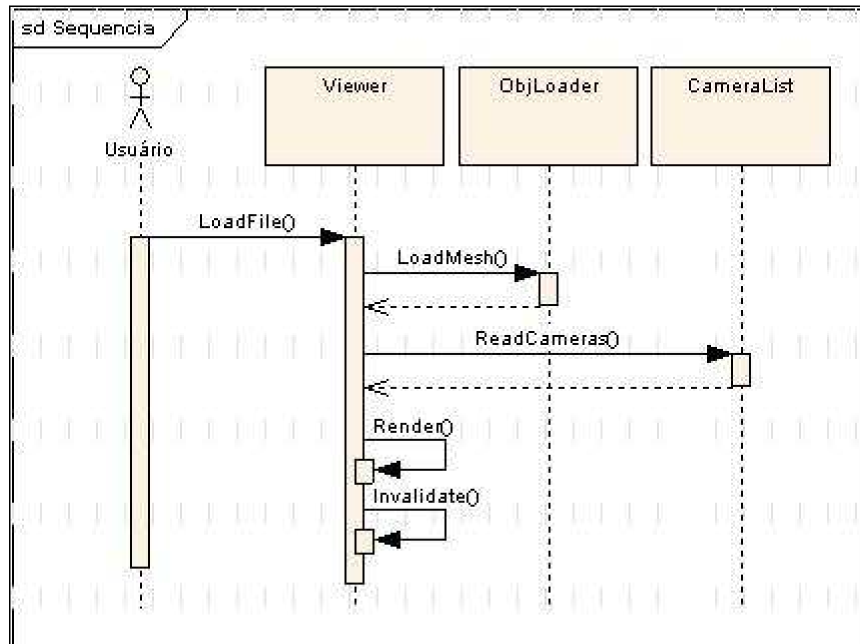


Figura 41 – Diagrama de seqüência para carregar um arquivo e renderizá-lo

Indiferente de outras aplicações, o usuário é que inicia todo o processo. O primeiro passo é carregar o arquivo utilizando o método `LoadFile`. Este, por sua vez, fará uma chamada ao `LoadMesh` da biblioteca Gita. Assim que a API terminar de carregar o modelo, ele é retornado para o `Viewer` que solicita a carga das câmeras. Quando terminar este último passo, armazena o conjunto de câmeras localmente também. A partir deste ponto, o modelo está pronto para ser renderizado através do método `Render` (conforme a figura 42). Após renderizar o modelo, o método `Invalidate` é chamado para fazer com que a tela seja redesenhada futuramente.

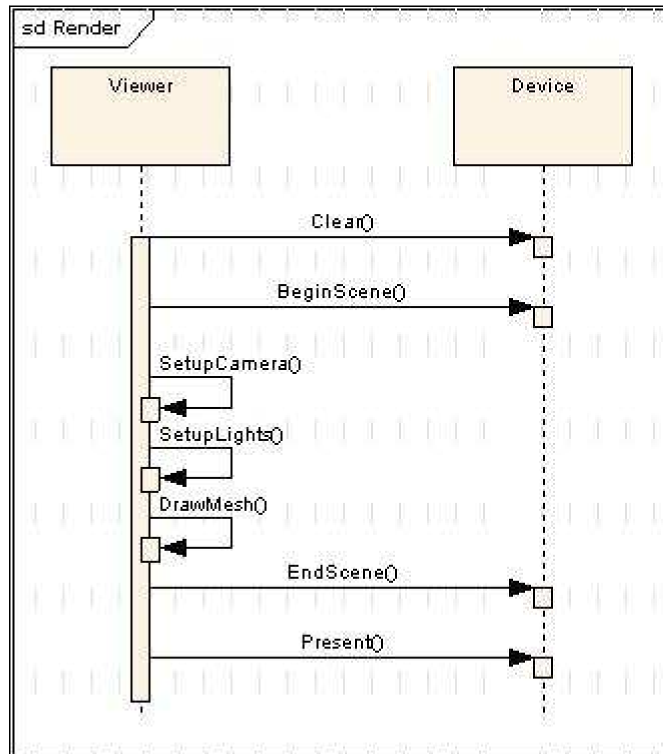


Figura 42 – Diagrama de seqüência do método `Render`

O método `Render` é chamado sempre que for preciso renderizar o modelo carregado. Através dele, o `Viewer` invoca o `Device` e faz com que ele limpe a tela (`Clear`) e inicie a cena (`BeginScene`). Novamente o `Viewer` assume o controle do processo e desta vez configura no `Device` as câmeras carregadas e algumas luzes. Após isso, faz o desenho do modelo, finaliza a cena e apresenta ela.

O desenho do modelo é feito conforme o diagrama da figura 43.

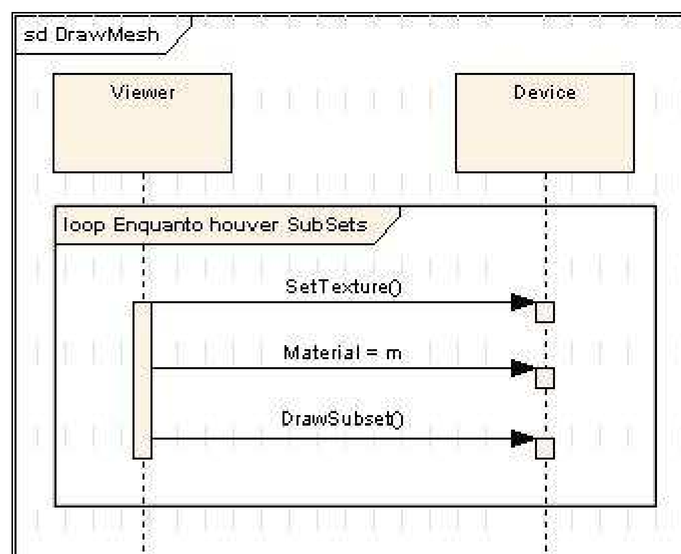


Figura 43 – Diagrama de seqüência do método `DrawMesh`

As operações apresentadas acima são executadas para cada *SubSet* do modelo

carregado, portanto eles podem assumir diferentes texturas e materiais, conforme descrito anteriormente.

Toda esta arquitetura definida não funcionaria se alguns detalhes fossem ignorados, tais como:

a) a qualquer momento o dispositivo pode sofrer um *reset* e, neste caso, tudo o que havia sido carregado é apagado da memória. Para resolver este problema, o evento `DeviceReset` da classe `Device` deve ser interceptado e forçar a recarga dos objetos, conforme o quadro 8;
b) o evento `OnPaintBackground` deve ser sobrescrito para que sua implementação padrão, que é pintar o fundo da tela, seja desabilitada (vide quadro 9). Caso contrário o modelo carregado pelo MD3DM nunca aparecerá, pois logo após o seu desenho, o .NET CF pintará novamente a tela, e isso sobrescreverá o que havia sido desenhado anteriormente.

```
protected void OnDeviceReset(object sender, EventArgs e)
{
    // Recarrega o último arquivo selecionado
    LoadFile();
}
```

Quadro 8 – Implementação do evento `DeviceReset`

```
protected override void OnPaintBackground(PaintEventArgs e)
{
    // Desabilita o evento…
}
```

Quadro 9 – Evento `OnPaintBackground` sobrescrito para anular o seu comportamento

3.4 RESULTADOS E DISCUSSÃO

Os testes feitos com o emulador do Visual Studio 2005 apresentaram uma performance muito abaixo da obtida com o dispositivo HTC S620[4]. Enquanto a taxa de *frames* por segundo ao renderizar os modelos utilizando o emulador não passou de 1 fps (vide figura 44), o dispositivo conseguiu realizar a mesma tarefa com um desempenho muito superior: entre 10 e 35 *frames* por segundo, tal como apresentado na figura 45. A figura 46 demonstra somente a textura utilizada no modelo.

---

[4] O autor do trabalho não possui um dispositivo compatível com a aplicação em questão. No entanto, Enrique Sangenis, do México, colaborou enviando algumas imagens da aplicação sendo executada no HTC S620.

Figura 44 – Modelo 3D renderizado no emulador

Figura 45 – Modelo 3D renderizado em um dispositivo HTC S620

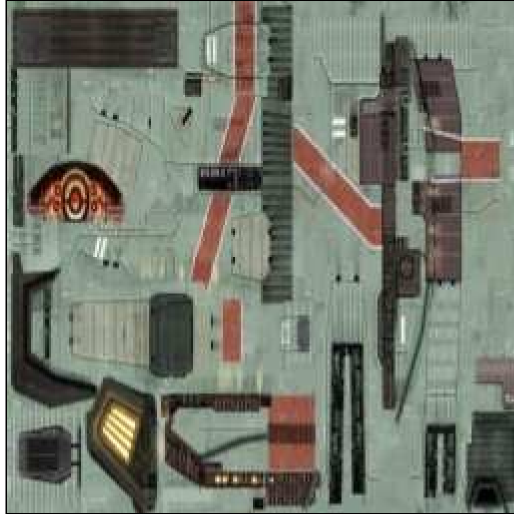


Figura 46 – Textura utilizada nos modelos das figuras 44 e 45

O modelo 3D renderizado nas figuras 44 e 45 possui aproximadamente 450 faces e 2100 linhas com coordenadas de vértices, normais e texturas. Além disso, o arquivo usado para texturização possui 256 *pixels* de largura e altura. O tempo de carregamento no emulador não ultrapassou os 20 segundos. No dispositivo real foi inferior aos 5 segundos, ou seja, quatro vezes menos.

Utilizando um modelo mais simples com aproximadamente 15 definições de vértices e normais, sem textura e 12 faces, o tempo de carregamento no emulador foi de 4 segundos. No HTC S620 foi instantâneo, não atingindo 1 segundo. O desempenho em *frames* por segundos no dispositivo real foi de 30 fps, enquanto no emulador permaneceu em 1 fps. Esta diferença de tempo de carregamento e desempenho com o mesmo modelo existe porque o emulador utilizado é genérico, ou seja, não é específico para o dispositivo real em questão.

Os resultados obtidos no presente trabalho são semelhantes aos da M3GE. Nesta, com um modelo tão complexo quanto o apresentado nas figuras 44 e 45, o tempo de carregamento foi de 17 segundos, enquanto a renderização ficou entre 15 e 20 *frames* por segundo quando não há teste de colisão, tal como é na Gita.

No que diz respeito ao desempenho, a única diferença identificada entre as duas bibliotecas foi nos emuladores. Enquanto com a M3GE os resultados foram praticamente os mesmos obtidos no dispositivo real, na Gita os valores foram bem distintos.

4 CONCLUSÕES

O presente trabalho apresentou a implementação de um *framework* para manipulação de modelos 3D no formato *Wavefront* para dispositivos móveis que suportam o .NET CF 2.0. Além disso, também foi criada uma aplicação para validar o *framework*. Até a data da sua concepção, não havia uma biblioteca para o .NET CF 2.0 com a capacidade de ler e renderizar um modelo 3D especificado abertamente.

Com base nos resultados obtidos nos testes com os dispositivos reais, pode-se concluir que a plataforma .NET CF satisfaz as necessidades para o desenvolvimento de aplicações 3D em dispositivos limitados. No entanto, a emulação deste tipo de aplicação demonstrou-se pouco eficiente.

Todos os objetivos previamente formulados foram alcançados. Além destes, foram implementadas rotinas de transformação (rotação, translação e escala) no modelo, que são essenciais para, futuramente (vide item 4.1), tornar a API uma biblioteca voltada para o desenvolvimento de jogos.

Duas dificuldades ficaram evidentes durante o desenvolvimento do projeto: a falta de emuladores (existem só dois, um para Windows Mobile 5.0 e outro para a versão 6.0) e a documentação extremamente precária sobre a biblioteca MD3DM. No que diz respeito a esta última, existem muitos materiais sobre o Direct3D para *desktop* que auxiliaram durante o desenvolvimento do trabalho, mas nada específico para os dispositivos móveis. Além disso, esta biblioteca demonstrou-se muito flexível e completa para o desenvolvimento do *framework* Gita.

Assim, torna-se cada vez mais claro que o desenvolvimento de aplicações 3D para dispositivos móveis está alcançando a sua maturidade. A questão já não é mais “como” e sim “quando” os recursos estarão disponíveis de tal forma que os desenvolvedores possam criar jogos e outras aplicações 3D da mesma forma como fazem nos *desktops*.

4.1 EXTENSÕES

Há uma série de extensões a serem pesquisadas e desenvolvidas. As mais interessantes giram em torno de um único objetivo: transformar a Gita em um motor de jogos 3D.

Todo motor de jogos, antes de qualquer coisa, não deve ficar restrito a um único formato de arquivo. Portanto, uma possível extensão é a criação de novas classes para carregar e renderizar modelos nos mais diversos formatos (X, MD2 ou até mesmo um novo e mais otimizado).

Outra questão importante é a detecção de colisão. Existem diversas técnicas para aplicar este conceito, mas deve-se sempre levar em consideração as limitações dos dispositivos. Um trabalho futuro comparando todas estas técnicas seria de extrema importância, pois serviria como base para a escolha do melhor algoritmo para adotar em um motor de jogos voltado a dispositivos limitados.

A criação de um novo componente voltado à iluminação também é uma possível extensão. Na implementação atual já existe a possibilidade de definir um arquivo XML com algumas informações a mais para cada modelo carregado. Este arquivo poderia ser utilizado também para definir todas as luzes padrão da aplicação ou jogo.

REFERÊNCIAS BIBLIOGRÁFICAS

ALMGVIST, P. **Keep it single, stupid!** [S.l.], 2001. Disponível em: < http://www.digital-web.com/articles/keep_it_simple_stupid>. Acesso em: 27 abr. 2007.

AUTODESK. **Autodesk 3ds max**. [Montreal], 2006a. Disponível em: <http://www.autodesk.com/3dsmax>. Acesso em: 28 abr. 2007.

______. **Autodesk Maya**. [Montreal], 2006b. Disponível em: <http://www.autodesk.com/maya>. Acesso em: 28 abr. 2007.

BARNES, D. **Fundamentals of Microsoft .NET compact framework development for the Microsoft .NET framework developer**. [Redmond], 2003. Disponível em: <http://msdn.microsoft.com/netframework/programming/netcf/gettingstarted/default.aspx?pull=/library/en-us/dnnetcomp/html/net_vs_netcf.asp>. Acesso em: 26 abr. 2007.

BLENDER FOUNDATION. **Blender3D.org:** home. [Amsterdam], 2005. Disponível em: <http://www.blender.org>. Acesso em: 12 jul. 2007.

EASTMEN, Peter. **Art of illusion**. [S.l.], 2005. Disponível em: <http://www.artofillusion.org>. Acesso em: 12 jul. 2007.

EXPANSYS. **HP P4350 Pocket PC**. [S.l.], 2007. Disponível em: <http://www.expansys.com/zoompic.aspx?type=item&i=141502>. Acesso em: 25 abr. 2007.

FOLHA ONLINE. **Número de usuários de celular cresce 137% em cinco anos**. [São Paulo], 2005. Disponível em: <http://www1.folha.uol.com.br/folha/informatica/ult124u19266.shtml>. Acesso em: 25 abr. 2007.

FOWLER, M. **Is design dead?** [S.l.], 2000. Disponível em: <http://www.martinfowler.com/articles/designDead.html >. Acesso em: 27 abr. 2007.

GAMMA, E. et al. **Design patterns**: elements of reusable object-oriented software. Massachusetts: Addison-Wesley Professional, 1995.

GOLD, M. **Considerations in porting and deploying a winforms GDI+ game to the pocket PC**. [S.l.], 2003. Disponível em: <http://www.vbdotnetheaven.com/UploadFile/mgold/PocketPCDeploy02012007235321PM/PocketPCDeploy.aspx>. Acesso em: 27 abr. 2007.

GOMES, J. VELHO, L; **Sistemas gráficos 3D**. Rio de Janeiro: IMPA, 2006.

HTC. **HTC S620** [S.l.], 2007. Disponível em: <http://www.europe.htc.com/products/htcs620.html>. Acesso em: 12 jul. 2007.

JCP. **The Java community process (SM) program**: JCP procedures – JCP 2 process document. [Palo Alto], 2004a. Disponível em: <http://www.jcp.org/en/procedures/jcp2>. Acesso em: 10 mar. 2007.

______. **The Java community process (SM) program**: JSRs – Java specification requests – JSR overview. [Palo Alto], 2004b. Disponível em: <http://www.jcp.org/en/jsr/detail?id=184>. Acesso em: 17 mar. 2007.

MACEDO JÚNIOR, I. J. A. **mOGE – mobile graphics engine**: o projeto de um motor gráfico 3D para a criação de jogos em dispositivos móveis. 2005. 75 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro de Informática, Universidade Federal de Pernambuco, Recife. Disponível em: < http://www.cin.ufpe.br/~tg/2004-2/ijamj.pdf>. Acesso em: 17 abr. 2007.

MCNAMARA, A. **MTL file specification**. [New Haven], 2000. Disponível em: <http://zoo.cs.yale.edu/classes/cs490/00-01a/mcnamara.antoine.amm43/mtl.html>. Acesso em: 18 maio 2007

MICROSOFT. **.NET compact framework**. [Redmond], 2005a. Disponível em: <http://msdn.microsoft.com/mobility/netcf/>. Acesso em: 10 set. 2007.

______. **.NET compact framework architecture**. [Redmond], 2005b. Disponível em: <http://msdn2.microsoft.com/en-us/library/9s7k7ce5.aspx>. Acesso em: 18 set. 2007.

______. **Visual Studio 2005**. [Redmond], 2004. Disponível em: <http://www.microsoft.com/windowsmobile/5/default.mspx >. Acesso em: 30 abr. 2007.

______. **Windows mobile 5**. [Redmond], 2005c. Disponível em: <http://www.microsoft.com/windowsmobile/5/default.mspx >. Acesso em: 10 set. 2007.

MILLER, T. **Managed DirectX 9 kick start**: graphics and game programming. Califórnia: Sams Publishing, 2003.

MOBILE-REVIEW. **Review of Windows Mobile 6.0 for PPC**. [S.l.], 2006. Disponível em: < http://www.mobile-review.com/pda/articles/wm-crossbow-en.shtml >. Acesso em: 25 abr. 2007.

MOORE, G. **Moore’s law 40th anniversary**. [S.l.], 2005. Disponível em: <http://www.intel.com/pressroom/kits/events/moores_law_40th/index.htm>. Acesso em: 10 set. 2007.

NEWTEK. **Lightwave 3D**. [Texas], 2006. Disponível em: <http://www.newtek.com/lightwave/>. Acesso em: 10 maio 2007.

NOKIA. **JSR-184 mobile 3D API for J2ME**. [P.O.Box], 2003. Disponível em: <http://forum.nokia.com/info/sw.nokia.com/id/9e6bdb61-b739-44d7-a7ae-42a4f56ee119/jsr184-specification-1.0_Installer.zip.html>. Acesso em: 10 mar. 2007.

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: 21 maio 2007.

OPENNETCF. **Smart device framework**. [S.l.], 2005. Disponível em: <http://www.opennetcf.org>. Acesso em: 25 abr. 2007.

PAMPLONA, V. F. **Um protótipo de motor de jogos 3D para dispositivos móveis com suporte a especificação mobile 3D graphics API for J2ME**. 2005. 83 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

REVOLUTION SOFTWARE. **Doom**. [S.l.], 2007. Disponível em: <http://www.revolution.cx/DoomCE.htm>. Acesso em: 25 abr. 2007.

RUSSO, S. **BlenderPocket**. [S.l.], 2007. Disponível em: <http://russose.free.fr/BlenderPocket>. Acesso em: 15 jul. 2007.

SANTEE, A. **Programação de jogos com C++ e DirectX**. São Paulo: Novatec, 2005.

SIEMENS AG. **Siemens CX65**. [S.l], 2005. Disponível em: <http://www.gsmarena.com/phone657.php>. Acesso em: 10 maio 2007.

SIEMENS AG. **Siemens Mobility Toolkit**. [S.l.], 2005. Disponível em: <http://www.mobilefish.com/emulators/siemens_smtk_m50/siemens_smtk_m50.html>. Acesso em: 25 abr. 2007.

SMART DEVICE CENTRAL. **Dell Axim x51v**. [S.l.], 2005. Disponível em: <http://www.smartdevicecentral.com/print_article/Dell+Axim+X51v/160439.aspx >. Acesso em: 25 abr. 2007.

SOLOMATIN, S. **Intel 2700G chipset – 3D accelerator in your pocket**. [S.l.], 2006. Disponível em: <http://www.digit-life.com/articles2/pda/intel-2700g.html>. Acesso em: 25 abr. 2007.

TENORIO, G. S. **Modelos 3D**: produção, visualização e desdobramentos. Brasília, 2003. Disponível em: <http://www.unb.br/fau/disciplinas/cg2/texto1.html>. Acesso em: 18 mar. 2007.

YOSHIMURA, B. H. **O que é smartphone?**. [S.l.], 2006. Disponível em: <http://www.linkgratis.com.br/materia/O_Que_e_Smartphone/>. Acesso em: 25 abr. 2007.