UNIVERSIDADE DE CAXIAS DO SUL
CENTRO DE COMPUTAÇÃO E TECNOLOGIA DA INFORMAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LUCIANO COPPI

Desenvolvimento de uma engine 3D voltada a dispositivos móveis

Prof. Carlos E. Nery
Orientador

Caxias do Sul, Dezembro de 2009

"A mind that has been stretched will never return to it's original dimension."

Albert Einstein

AGRADECIMENTOS

Agradeço ao meu orientador, Prof. Carlos Eduardo Nery, pela contribuição no desenvolvimento e correção do trabalho, aos meus amigos pelo apoio e, especialmente aos meus familiares que sempre estiveram presentes durante esta importante etapa da minha vida. Muito obrigado a todos!

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 6

LISTA DE FIGURAS 7

LISTA DE TABELAS 9

RESUMO 10

ABSTRACT 11

1 INTRODUÇÃO 12
1.1 Motivação 13
1.2 Objetivos 14
1.3 Estrutura do trabalho 14

2 FUNDAMENTAÇÃO TEÓRICA 16
2.1 A Evolução dos dispositivos portáteis 16
2.2 Histórico dos Jogos para Dispositivos Móveis 20
2.3 O Sistema Operacional Windows Mobile 22
2.4 API *Managed Direct3D Mobile* 25
2.4.1 Classes de Renderização e Iluminação 25
2.4.2 Classes de Transformação 28

3 ENGINE 3D 32
3.1 Engine 3D 32
3.2 Gerenciador de Entrada 34
3.3 Gerenciador de Objetos 35
3.3.1 O Formato *Wavefront* 35
3.3.2 O Formato MD3DM 38
3.4 Gerenciador Gráfico 39
3.5 Gerenciador de Mundo 40

4 IMPLEMENTAÇÃO . . . 41
4.1 Estrutura e organização do código fonte . . . 41
4.2 Metodologia e técnicas utilizadas . . . 41
4.3 Ferramentas utilizadas no desenvolvimento . . . 42
4.4 MDGE . . . 43
4.4.1 Loader . . . 44
4.4.2 Ligths . . . 47
4.4.3 Objects . . . 48
4.4.4 Transforms . . . 51
4.4.5 *World* . . . 52
4.4.6 *Form* . . . 59
4.5 Aplicações de Teste . . . 60
4.5.1 Primeira Aplicação de Teste . . . 61
4.5.2 Segunda Aplicação de Teste . . . 62

5 RESULTADOS . . . 66

6 CONSIDERAÇÕES FINAIS . . . 68
6.1 Trabalhos futuros . . . 69

REFERÊNCIAS . . . 70

LISTA DE ABREVIATURAS E SIGLAS

MD3DM	Managed Direc3D Mobile
GAPI	Game API
API	Application Programming Interface (Interface de programação de aplicativos)
PDA	Pesrsonal Digital Assistant (Assistente Pessoal Digital)
GDI	Graphics Device Interface (Interface gráfica do dispositivo)
CPU	Central Processing Unit (Unidade de processamento central)
FPS	Frames Per Second (Quadros por segundo)
CF	Compact Framework
FP	First Person (Primeira Pessoa)
3D	Tridimensional
2D	Bidimensional

LISTA DE FIGURAS

Figura 2.1: Exemplos de celulares, *smarphones* e PDAs 16
Figura 2.2: *Palm Pilot* 1000 17
Figura 2.3: Nokia Communicator 18
Figura 2.4: Interface gráfica do *Pocket PC* 2000 18
Figura 2.5: Nokia *N-Gage* 19
Figura 2.6: Apple iPhone 20
Figura 2.7: Interface do jogo *Snake* 20
Figura 2.8: Um jogo de corrida desenvolvido em Java 21
Figura 2.9: Um jogo de corrida para o iPhone 22
Figura 2.10: O jogo *Doom* para Windows Mobile 23
Figura 2.11: Interface dos emuladores, versão *Pocket PC* e *Smartphone* 24
Figura 2.12: Exemplo de uma malha de triângulos 27
Figura 2.13: *View Frustum* 29
Figura 2.14: *Field of View* 30

Figura 3.1: Arquitetura de uma *engine* 3D. 32
Figura 3.2: Dois tipos de teclado, virtual à direita e físico à esquerda. 35
Figura 3.3: Vetor normal em uma face 37
Figura 3.4: Coordenadas de textura 38
Figura 3.5: Exemplo de um *Mesh* no formato *.md3dm*. 39

Figura 4.1: Arquitetura da MDGE 44
Figura 4.2: Classe *ObjectLoader* 45
Figura 4.3: Classe *MaterialLoader* 46
Figura 4.4: Classes *Util* e *Md3dmLoader* 46
Figura 4.5: Arquivo XML de luzes 47
Figura 4.6: Classe *LigthsLoader* 48
Figura 4.7: Iluminação desativada à esquerda e ativada à direita 48
Figura 4.8: Classes *SphereMesh* e *PolygonMesh* 49
Figura 4.9: Classe GraphicMesh 50

Figura 4.10: Classe *BoundingVolume* 51
Figura 4.11: O *BoundingBox* à esquerda e o *BoundingSphere* à direita 51
Figura 4.12: Classe *Rotation* 52
Figura 4.13: Classes *Scale* e *Translate* 52
Figura 4.14: Classes *CameraLoader* e *CameraObject* 53
Figura 4.15: Arquivo XML contendo os valores das câmeras 54
Figura 4.16: Classe *Camera* 55
Figura 4.17: Relação entre pontos da tela e espaço 3D 55
Figura 4.18: Trecho de código com os cálculos dos ângulos X e Y 56
Figura 4.19: Método *AddToCameraPosition* 56
Figura 4.20: Classe *FligthCamera* 57
Figura 4.21: Trecho de código demonstrando o cálculo da rotação 57
Figura 4.22: Método *UpdateCamera* 58
Figura 4.23: Classes *FpsTimer* e *RenderStateConfig* 59
Figura 4.24: Modo *Wireframe* e modo Normal 59
Figura 4.25: Exemplo de um objeto renderizado na cena 61
Figura 4.26: Câmera Inicial 62
Figura 4.27: Câmera panorâmica 62
Figura 4.28: Câmera superior 62
Figura 4.29: Tela da aplicação de teste 2 63
Figura 4.30: Tela da aplicação de teste 2 64

LISTA DE TABELAS

Tabela 2.1: Classes básicas da API MD3DM 26
Tabela 2.2: Principais métodos da estrutura *Matrix* utilizadas neste trabalho 31
Tabela 2.3: Principais métodos da estrutura *Quaternion* utilizadas neste trabalho 31

Tabela 3.1: Componentes da arquitetura de uma *Engine* 33
Tabela 3.2: Exemplo de um arquivo *Obj* 37
Tabela 3.3: Exemplo de um arquivo *Mtl* 38

Tabela 4.1: Dispositivos utilizados no desenvolvimento e testes 43
Tabela 4.2: Configuração das teclas na aplicação de teste 1 64
Tabela 4.3: Configuração das teclas na aplicação de teste 2 65

RESUMO

A indústria jogos se desenvolveu muito nos últimos anos, batendo recordes de venda e faturando bilhões. Com a crescente popularidade de dispositivos móveis, como celulares, smartphones entre outros, um novo mercado para o desenvolvimento de jogos surgiu. Entretanto desenvolver aplicações tão complexas quanto um jogo, voltado para dispositivos com pouca memória e poder de processamento, é uma tarefa árdua que requer a utilização de ferramentas como uma *engine* 3D para facilitar este processo.

O objetivo deste trabalho é desenvolver um protótipo de uma *engine* 3D para dispositivos móveis, englobando algumas de suas funcionalidades básicas. Serão abordados alguns aspectos como a arquitetura de uma *engine*, o sistema operacional onde a aplicação será rodada e a *API* gráfica utilizada para renderização dos objetos. Para o desenvolvimento será utilizada a *API Mobile Direct3D* e a linguagem de programação C#.

Palavras-chave: Engines 3D, Aplicações Móveis, Jogos, Mobile Direct3D.

Development of a game engine for mobile devices

ABSTRACT

The game industryes had an incredible growth in the last years, expanding the sales and adding billions. The mobile devices popularity, like cellular, Smartphone, PDA, and others, created a new market for game development. However, to develop these complex applications, directed for devices with a set of restrictions, isn't easy. Is necessary to apply different resources like an 3D engine.

The main objective is develop a prototype of an 3D engine for mobile devices, that include some of it's basics functionality. It was approach some aspects like the architecture of an engine, the operational system that was used and the graphic API used to render the objects. The development was realized applying the Mobile Direct3D API and C# language.

Keywords: 3D Engines, Mobile Development,Games, Mobile Direct3d.

1 INTRODUÇÃO

Os jogos eletrônicos são uma opção de entretimento que sempre despertou o interesse das pessoas. Ao longo dos tempos, os jogos para consoles[1] e computadores foram se modernizando, passando da simplicidade dos gráficos em duas dimensões (2D) para um mundo muito mais realista, com complexos gráficos tridimensionais (3D) e efeitos sonoros capazes de imergir o jogador completamente na história.

Analisando o sucesso atingido pelos jogos eletrônicos, pode-se afirmar que a mesma evolução acontecerá com os jogos para dispositivos portáteis, como celulares e PDAs[2]. Atualmente existem algumas plataformas portáteis de jogos extremamente bem sucedidas comercialmente, como o Nintendo DS (NINTENDO, 2009) e o PlayStation Portable (SONY, 2009), capazes de suportar gráficos com uma qualidade equivalente ao dos jogos para computadores do final da década de 90. No entanto estas plataformas são específicas para jogos.

Novos recursos passaram a ser incorporados recentemente nos PDAs disponíveis no mercado como um aumento na capacidade de processamento, a incorporação de telas com alta resolução e sensíveis ao toque (*touchscreen*) e a utilização de sistemas operacionais juntamente com linguagens de programação para gerenciar os recursos do dispositivo. Esta evolução no hardware permite que estes dispositivos possam suportar gráficos 3D, originalmente utilizados nas interfaces gráficas para facilitar a interação com o usuário. Isto criou um novo mercado em potencial a ser explorado: o dos jogos eletrônicos para estes dispositivos.

Atualmente os jogos em 3D para dispositivos móveis ainda estão em um estado primitivo, apresentando gráficos com baixa resolução. Entretanto com o aumento da popularidade deste tipo de jogo, a tendência é que a qualidade gráfica melhore significativamente, acompanhando a evolução do hardware como já está acontecendo com os jogos para o iPhone (APPLE, 2009) que hoje já atingiram uma qualidade gráfica quase equivalente aos dos jogos para plataformas portáteis, embora este dispositivo não seja específico para jogos.

[1]Console é o termo utilizado para designar aparelhos de videogame
[2]*Personal Digital Assistans* é um computador de dimensões reduzidas, portátil

Desenvolver um jogo é uma tarefa extremamente complexa, pois envolve um conhecimento profundo em diversas áreas da computação. Essa complexidade se acentua ainda mais em ambientes com poder limitado de processamento e memória. Por isso é imprescindível utilizar ferramentas que facilitem o desenvolvimento de um jogo como os motores de jogos 3D ou, em inglês, 3D *game engines*.

Entretanto, uma *engine* 3D completa exerce inúmeras funções, cada uma quase tão complexa quanto um jogo em si, tornando o seu desenvolvimento igualmente complicado. Por isso, dentro deste trabalho será desenvolvido apenas um protótipo de *engine* 3D, englobando apenas algumas de suas funcionalidades.

A API[3] gráfica *Managed Direct3D Mobile* (MD3DM) (MICROSOFT, 2005a) será utilizada no desenvolvimento deste trabalho. Essa API está presente no *framework*[4] de desenvolvimento .NET *Compact Framework* 3.5 (.NET CF 3.5) (MICROSOFT, 2008a) que roda no sistema operacional Windows Mobile 6.1 (MICROSOFT, 2009a).

1.1 Motivação

A economia mundial movimenta bilhões de dólares por ano através do comércio de jogos eletrônicos. Nos últimos anos houve uma expansão das vendas de dispositivos móveis, impulsionados pelo iPhone que se tornou um verdadeiro fenômeno de vendas, aumentando consideravelmente a venda de jogos parta estes dispositivos.

Além disso, a popularização destes dispositivos faz com que grandes empresas comecem a enxergar nesta área uma excelente oportunidade de negócios. A Microsoft, por exemplo, lançou em 2005 uma versão móvel de sua API para desenvolvimento de aplicações gráficas 3D, o MD3DM. Além desta, lançou em 2006 um *framework* para o desenvolvimento de jogos para console e computador chamado de XNA (MICROSOFT, 2009b) que na suas versões mais recentes permite também o desenvolvimento para dispositivos portáteis como o Zune[5] .

No entanto, o desenvolvimento de jogos em 3D para dispositivos portáteis ainda é uma área muito recente, que requer muito estudo e desenvolvimento. Por este motivo não existem muitas ferramentas ou *frameworks* que facilitem o desenvolvimento de um jogo. Devido a esta necessidade é que surge a motivação para este trabalho, desenvolver um protótipo de uma *engine* 3D para dispositivos portáteis, com o intuito de não apenas facilitar o desenvolvimento de jogos, mas também de iniciar novas pesquisas nesta área que está em franca expansão nos últimos anos.

[3] *Application Programming Interface*, Interface de Programação de Aplicativos é um conjunto de rotinas e padrões estabelecidos por *software* para a utilização das suas funcionalidades por programas aplicativos.

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

[5] Zune é um reprodutor de mídia digital portátil fabricado pela Microsoft.

1.2 Objetivos

O objetivo principal deste trabalho é desenvolver um protótipo de uma *engine* 3D para jogos voltada a dispositivos portáteis (celulares e PDAs). Este protótipo deve contemplar os seguintes requisitos:

a) carregar e renderizar modelos tridimensionais nos formatos *Wavefront* e *.md3dm*;

b) permitir a navegação pelo cenário, utilizando o teclado e o *touchscreen* do dispositivo, permitindo também a criação de câmeras em diferentes pontos do cenário;

c) permitir a execução de operações como rotação, translação e escala sobre os modelos;

d) permitir a renderização de algumas formas geométricas básicas

e) permitir que o usuário defina e utilize quantas luzes precisar para iluminar a cena;

f) oferecer portabilidade, entre outros dispositivos móveis compatíveis com o .NET CF e Windows Mobile.

O trabalho também apresentará a implementação de duas aplicações de testes, para validar e avaliar o desempenho de todas as funcionalidades implementadas pela *engine*. O desempenho será avaliado através do cálculo dos *frames* por segundo(FPS), que serão exibidos em cada aplicação.

1.3 Estrutura do trabalho

Esta monografia divide-se da seguinte forma:

- No Capítulo 2 são apresentados alguns temas como a evolução dos dispositivos móveis e, a plataforma Windows Mobile e os jogos para dispositivos móveis. Além disso, também é feita uma descrição sobre as características da API MD3DM.

- No Capítulo 3, são apresentados os conceitos que envolvem a construção de uma *Engine*, descrevendo os principais componentes que serão abordados neste trabalho.

- No Capítulo 4 são descritas as técnicas utilizadas na implementação dos conceitos discutidos no Capítulo 3, bem como o desenvolvimento de aplicações de teste, para testar as funcionalidades implementadas.

- No Capítulo 5 são apresentados os testes e, os resultados obtidos.
- No Capítulo 6 são apresentadas a contribuição deste trabalho e, sugestões para trabalhos futuros nesta área.
- No Capítulo 7 são listadas as referências bibliográficas que foram utilizadas no desenvolvimento deste trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão tratados alguns conceitos relacionados a este trabalho. As duas primeiras seções apresentam uma breve descrição sobre evolução dos dispositivos portáteis e, o histórico dos jogos para dispositivos móveis. Logo em seguida são apresentados o sistema operacional Windows Mobile e, algumas das tecnologias de desenvolvimento disponíveis para ele. Na última seção serão apresentadas, algumas características da API gráfica que será utilizada no implementação deste trabalho.

2.1 A Evolução dos dispositivos portáteis

Segundo PAMPLONA (2005) a computação móvel é caracterizada por um dispositivo móvel, com capacidade de processamento em um ambiente sem-fio. Alguns exemplos destes dispositivos são os PDA's, os telefones celulares e os *smartphones*[1] que podem ser observados na Figura 2.1

Figura 2.1: Exemplos de celulares, *smarphones* e PDAs

A primeira geração de telefones celulares propriamente ditos surgiu em meados da década de 80. Eram aparelhos muito simples, que apenas efetuavam e recebiam ligações utilizando uma tecnologia de transmissão analógica. Alem disto, estes

[1] *Smartphone* é um telefone celular com funções avançadas, como a presença de um sistema operacional completo que serve como plataforma para o desenvolvimento de aplicativos

aparelhos eram muito grandes e suas baterias não duravam muito tempo, impossibilitando a sua utilização por um longo período de tempo reduzindo a mobilidade.

A partir do inicio da década de 90 um novo sistema de telecomunicação sem fio passou a ser utilizado, o sistema digital 2G. Devido à digitalização dos dados, novos serviços como o sistema de troca de mensagens de texto (SMS) e, posteriormente serviços de internet passaram a ser oferecidos. Aliado a esta evolução na tecnologia de transmissão de dados e, a miniaturização dos circuitos eletrônicos, o hardware destes aparelhos também evoluiu.

A diminuição do tamanho dos aparelhos e o desenvolvimento de baterias mais sofisticadas fez com que eles se tornassem extremamente populares. Paralelamente a isto, em 1996 os primeiros PDAs como o *Palm Pilot* 1000 (Figura 2.2) (PALM, 1996a), começaram a surgir no mercado. O *Pilot* era muito limitado em termos de memória, tendo apenas alguns *Megabytes* para armazenamento, entretanto apresentou um dos primeiros sistemas operacionais com uma interface gráfica baseada no *touchscreen*[2] para um dispositivo móvel, o *Palm OS* (PALM, 1996b) e, acabou vendendo cerca de 1 milhão de unidades em apenas 18 meses.

Este sistema operacional foi utilizado em um grande numero de dispositivos que se sucederam dando inicio a uma nova era na computação móvel. Nesta mesma época a Microsoft iniciou o desenvolvimento do Windows CE (MICROSOFT, 1996), que será apresentado posteriormente neste trabalho e, surgiram os primeiros *smartphones* como o Nokia *Communicator* (NOKIA, 1997a)(Figura 2.3).

Figura 2.2: *Palm Pilot* 1000

O Nokia *Communicator* revolucionou o mercado dos telefones celulares, pois foi uns dos primeiros aparelhos a reunir as funções de um PDA com as de um celular comum, criando o segmento dos *smartphones*. Estes aparelhos se tornaram muito mais populares do que os PDAs no final da década de 90 e, trouxeram uma nova

[2] *Touchscreen* é uma tela capaz de detectar e localizar um toque dentro de sua área

gama de funcionalidades como a presença de teclado do tipo QWERTY[3], câmera de vídeo embutida, conectividade com a internet através de uma rede sem fio, entre outras.

A Microsoft, empresa que sempre esteve na vanguarda do desenvolvimento de software somente entrou de vez no mercado dos dispositivos móveis em 2000 quando lançou o seu primeiro sistema operacional voltado exclusivamente para PDAs, o Microsoft *Pocket PC* 2000. Ele possuía uma interface gráfica muito similar ao do Windows 98, conforme pode ser observado na Figura 2.4, o que o tornava muito fácil de ser utilizado. Para os *smartphones* existia também uma versão desse sistema, chamada de *Pocket PC Phone Edition*, que possuía uma interface mais simplificada, sem suporte ao *touchscreen*.

Figura 2.3: Nokia Communicator

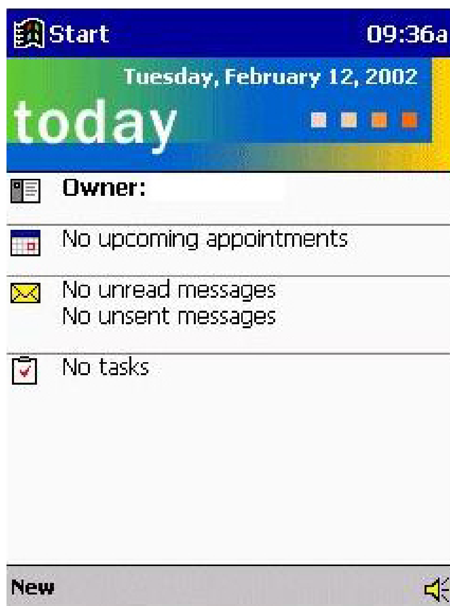


Figura 2.4: Interface gráfica do *Pocket PC* 2000

[3]Querty é o layout de teclado mais utilizado em computadores atualmente

Em 2003, a Microsoft substituiu a nomenclatura *Pocket PC* para Windows Mobile, no intuito de aproveitar a popularidade do Windows. Neste mesmo período a Nokia lançou o primeiro telefone celular voltado especificamente para jogos, o *N-Gage* (NOKIA, 2003) (vide Figura 2.5). Ele tinha um design que lembrava o *joystick* de um videogame, possuía um sistema operacional e utilizava o mesmo hardware de um *smartphone* comum o que o tornava bastante limitado para jogos um pouco mais complexos. Embora tenha sido um fracasso de vendas, o *N-Gage* foi muito importante, pois foi a primeira iniciativa de um grande fabricante em tentar reunir em um único dispositivo um videogame e um telefone celular. Além disso, a Nokia criou um dos primeiros sistemas de distribuição online de jogos e outros aplicativos para este dispositivo e mais tarde, para todos os outros modelos produzidos pela empresa.

Entretanto, em termos de inovação tecnológica, o dispositivo mais inovador lançado no mercado até hoje foi o iPhone (Figura 2.6). Introduzido em 2007 pela Apple, o iPhone foi o primeiro *smartphone* a apresentar uma unidade de processamento gráfico (GPU), permitindo a este aparelho suportar gráficos 3D somente vistos anteriormente em consoles ou computadores e, inaugurou um novo mercado para o desenvolvimento de aplicações gráficas 3D. Alem disso modificou completamente a forma de interação com um dispositivo ao apresentar novidades como a utilização do acelerômetro[4] que permite ao aparelho modificar a orientação da tela de acordo com a sua posição e também controlar os jogos e outros aplicativos reproduzindo exatamente o mesmo movimento que esta sendo feito pela mão do usuário.

Atualmente, os dispositivos que estão sendo produzidos no mercado seguem uma tendência, com interfaces gráficas complexas, presença de jogos em 3D, compatibilidade com diversos formatos de vídeo e uma capacidade de processamento maior para suportar todas essas aplicações.

Figura 2.5: Nokia *N-Gage*

[4]Acelerômetro é um sensor capaz de converter a aceleração da gravidade ou o movimento em sinais elétricos

Figura 2.6: Apple iPhone

2.2 Histórico dos Jogos para Dispositivos Móveis

Os jogos para dispositivos móveis começaram a ser desenvolvidos a partir da metade da década de 90, para os telefones celulares. Eram em 2D e extremamente limitados permitindo apenas uma interação básica com o usuário, porém a evolução do hardware dos celulares e outros dispositivos móveis ao longo dos anos proporcionou a criação de jogos cada vez mais complexos.

Os primeiros jogos que surgiram eram versões compactas dos jogos para console do final da década de 70. O primeiro jogo presente em um celular foi o *Snake* (NOKIA, 1997b) (Figura 2.7), introduzido pela Nokia em 1997 e esteve presente em mais de 350 milhões de telefones no mundo inteiro. Apesar de ser um jogo muito simples, pode-se dizer que ele introduziu o conceito de jogos para dispositivos móveis e serviu como base para outros jogos que seriam desenvolvidos posteriormente.

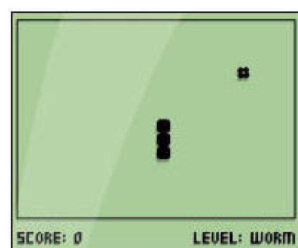


Figura 2.7: Interface do jogo *Snake*

Estes primeiros jogos eram pré-instalados no celular o que causava certa limitação, pois não era possível instalar novos jogos no celular e nem removê-los. Com o desenvolvimento da tecnologia WAP[5] e sua utilização em telefones celulares, no final da década de 90, os jogos e outros aplicativos passaram a ser distribuídos por *download*.

[5] *Wireless Application Protocol*, em português Protocolo para Aplicações sem Fio, é um padrão internacional para comunicação de dados sem fio, internet móvel

Esta nova forma de distribuição permitiu que os usuários pudessem escolher os jogos que quisessem e também, que os desenvolvedores lucrassem com a venda, já que os *downloads* eram pagos.

Praticamente todas as operadoras de telefonia móvel e desenvolvedoras de jogos passaram a explorar este mercado comercialmente e novas tecnologias de desenvolvimento de aplicações móveis como o *Java 2 Moblie Edition* (J2ME) (SUN, 2009), *BREW* (QUALCOMM, 2009), .NET CF 1.0, entre outras começaram a aparecer. Dentre estas tecnologias, no que diz respeito ao desenvolvimento de jogos, o Java se tornou a mais popular devido a sua portabilidade, já que era capaz de funcionar em qualquer sistema operacional, desde que ele possuísse a máquina virtual do Java instalada.

Os jogos em Java eram simples, com gráficos em 2D lembrando os dos videogames, e de fácil instalação e remoção. A grande vantagem desta tecnologia era o fato de funcionar em inúmeros sistemas operacionais diminuía os custos de desenvolvimento e facilitava a distribuição dos jogos em vários aparelhos de diferentes fabricantes, sua principal vantagem em relação às demais tecnologias. Além disso, por serem jogos simples, os arquivos eram pequenos, com um tamanho inferior a 1 MB, o que era perfeito visto que os celulares e outros dispositivos móveis possuíam pouca memória disponível para o armazenamento. A Figura 2.8, apresenta a tela de um jogo desenvolvido em Java.

Figura 2.8: Um jogo de corrida desenvolvido em Java

Em 2004, surgiram as primeiras APIs gráficas como a *Mobile 3D Graphics* (M3G)(NOKIA, 2004) para Java, *OpenGL ES* (KHRONOS, 2004), MD3DM, entre outras. Isto deu inicio ao desenvolvimento dos primeiros jogos em 3D dando origem a uma nova era de jogos para dispositivos móveis. Esta era foi marcada pela entrada de grandes desenvolvedoras de jogos neste mercado como a *Electronic Arts* e a *Ubisoft*, duas das maiores distribuidoras de jogos para console e computador no mundo, que passaram a investir milhões de dólares e licenciar seus principais jogos para os dispositivos móveis. Isto revolucionou o mercado aumentando a populari-

dade deste tipo de jogo, entretanto em termos de qualidade gráfica, estes jogos ainda eram muito ruins se comparados com as plataformas portáteis de jogos da época, especialmente por causa do hardware dos dispositivos que ainda era muito limitado.

Dentre estas APIs, a mais utilizada no desenvolvimento de jogos é o *OpenGL ES*, uma versão compacta da API *OpenGL* para *desktop*. Isto se deve ao fato de que esta API foi criada para funcionar em qualquer sistema operacional, sua principal vantagem em relação às demais. Esta portabilidade faz com que se possam desenvolver aplicações multiplataforma, sem a necessidade de fazer grandes alterações no código, levando-se em conta apenas a capacidade do hardware do aparelho.

A introdução do iPhone no mercado modificou completamente o mundo dos jogos para dispositivos móveis. Com uma qualidade gráfica muito superior ao de qualquer outro dispositivo podia rodar jogos complexos, com gráficos extremamente detalhados. Alem disso, as possibilidades de interação que ele oferecia, utilizando o *touchscreen* ou os sensores de movimento, o tornaram uma excelente plataforma para jogos. Foi um grande sucesso de vendas, impulsionando a venda e o desenvolvimento de novos jogos. Um exemplo de um jogo para o iPhone pode ser conferido na Figura 2.9.

Figura 2.9: Um jogo de corrida para o iPhone

Dentre os inúmeros sistemas operacionais disponíveis que suportam jogos em 3D, tem relevância a este trabalho apenas o Windows Mobile. Este sistema operacional ainda foi muito pouco explorado para utilização de jogos, devido a sua baixa popularidade, porém recentemente, muitos jogos que estão sendo lançados no mercado, possuem uma versão para este sistema operacional.

2.3 O Sistema Operacional Windows Mobile

Em 1996 a Microsoft iniciou o desenvolvimento de um sistema operacional que deveria funcionar em um dispositivo com menos de 1 *megabyte* de memória o Windows CE. Embora este sistema tenha sido projetado para computadores portáteis,

foi utilizado também em muitos outros equipamentos, como robôs, videogames, sistemas de navegação por satélite, PDAs, entre outros.

O Windows CE deu origem a todos os sistemas operacionais para dispositivos móveis lançados pela Microsoft posteriormente como o Windows Mobile lançado em 2003. Conjuntamente com este sistema surgiu o *framework* de desenvolvimento .NET CF 1.0, que permitia o desenvolvimento de aplicações de uma forma muito parecida com a dos computadores. Isto era possível, pois segundo BARNES (2003), o .NET CF é um subconjunto do .NET *Framework* para *desktop*, consistindo de algumas bibliotecas de classe básicas e algumas bibliotecas adicionais especificas para o desenvolvimento de aplicações móveis provendo um modelo de programação consistente e familiar. Esta similaridade no desenvolvimento de aplicações, fez com que muitos programas, incluindo alguns jogos fossem facilmente portados para os dispositivos móveis.

Neste período também surgiram as primeiras API gráficas para Windows Mobile como a *Game API* (GAPI) (MICROSOFT, 2005b) e a *Graphics Device Interface* (GDI) (MICROSOFT, 2009c). A API GAPI foi muito utilizada para o desenvolvimento de jogos como por exemplo, o jogo *Doom* (REVOLUTION STUDIOS, 2007) (Figura 2.10) para Windows Mobile e outras aplicações gráficas. Entretanto, ela estava disponível somente para os PDAs e possuía algumas limitações em termos de desempenho principalmente por ter apenas algumas funções básicas para renderização de imagens. A API GDI possuía algumas funções para desenhar linhas, curvas, texto e imagens e foi muito utilizada para o desenvolvimento de interfaces gráficas e jogos em 2D.

Figura 2.10: O jogo *Doom* para Windows Mobile

Estas APIs foram muito utilizadas até o surgimento do Windows Mobile 5.0 em 2005 e do .NET CF 2.0 que apresentava pela primeira vez o MD3DM, a versão para dispositivos móveis da API *Direct3D* para computadores, que acabou tornando-se o padrão para o desenvolvimento de aplicações gráficas, substituindo as APIs anteriores. O Windows Mobile 5.0 era distribuído em duas versões, *Smartphone* a versão mais limitada voltada a telefones celulares e, ao *Pocket PC*, a versão mais completa

desenvolvida para PDAs. No que diz respeito ao desenvolvimento de software para essa plataforma, a Microsoft lançou o Windows Mobile SDK[6] que funcionava em conjunto com o *Visual Studio* (MICROSOFT, 2009d).

Este SDK incluía um emulador, que simulava o hardware de um PDA ou de um *smartphone*, dependendo da plataforma alvo, exibindo inclusive a interface do dispositivo, conforme pode ser observado na Figura 2.11, o que permitia a realização de testes sem a necessidade do dispositivo físico embora, este processo não fosse muito eficiente devido às limitações do emulador. Também era possível compilar o código diretamente no dispositivo, através da sincronização do aparelho com o *Visual Studio* através de uma porta USB.

Figura 2.11: Interface dos emuladores, versão *Pocket PC* e *Smartphone*

Em 2007, o Windows Mobile 6 foi introduzido no mercado trazendo apenas algumas melhorias com relação à versão anterior. As maiores mudanças apresentadas foram na nomenclatura do programa que mudou para Windows Mobile 6 *Professional*, para PDAs com funções de telefone e Windows Mobile 6 *Standard*, para *smartphones* sem *touchscreen*. Em 2008 surgiu uma atualização desse sistema operacional, o Windows Mobile 6.1 que melhorou o desempenho de algumas funções do sistema e também modificou algumas partes da interface gráfica. O SDK também foi atualizado para a versão 6, oferecendo algumas funcionalidades novas para o desenvolvimento de aplicações. Nesse mesmo período, foi lançado o .NET CF 3.5, que trouxe uma série de novidades, principalmente no que diz respeito ao acesso as funções do hardware como reprodução de sons, melhorias na entrada e saída de dados, suporte ao GPS, suporte a compactação de arquivos entre outras.

Atualmente o Windows Mobile encontra-se na versão 6.5, entretanto em termos de desenvolvimento de aplicações gráficas, não houve nenhuma evolução na API MD3DM e nem o surgimento de outras APIs gráficas.

[6] *Software Development Kit*, é um conjunto de ferramentas que permite o desenvolvimento de aplicações para uma determinada plataforma

2.4 API *Managed Direct3D Mobile*

O MD3DM é uma API que provém suporte a aplicações com gráficos em 3D para ambientes Windows CE/Windows Mobile, a partir da versão 5.0. Ela é derivada da API *Managed Direct3D* encontrada no sistema operacional Windows para *desktop* porém, foi otimizada para ser utilizada em ambientes portáteis com menor poder de processamento. O termo *Managed* (Gerenciada) significa que esta API roda dentro do .NET CF 3.5 e pode ser acessada por qualquer linguagem deste *framework*, não precisando ser instalada no dispositivo para funcionar corretamente.

Em comparação com a versão para *desktop*, o suporte a várias funções 3D foi removido, devido às limitações de hardware dos aparelhos, deixando a API bastante simplificada. Toda a parte de desenho dos objetos na tela é feita por uma instância da classe *Device*. Esta classe representa um *display* gráfico que se comunica com o hardware do dispositivo para a exibição das imagens e, é a principal classe desta API. Segundo WIGLEY; MOTH; FOOT (2007) quando a instância da classe *Device* é criada, é passada a informação para descrever como utilizar os recursos do sistema e criar o *display*. Isto é feito através de uma instância da classe *PresentParameters*, que possui algumas opções, sendo que para uma aplicação móvel, apenas duas são utilizadas: *Windowed*, que define se a aplicação será executada em modo *fullscreen* ou não e, *SwapEffect* que está relacionado com a utilização de um *back buffer*[7], que geralmente é descartado (definindo o valor como *Discard*) pois, os dispositivos móveis não possuem um monitor como o dos computadores, podendo operar sem este buffer. Estas duas classes servem para a inicialização do MD3DM, porém a API possui outras classes importantes que podem ser agrupadas de acordo com suas funcionalidades da seguinte forma: classes de renderização e iluminação e, classes de transformação.

2.4.1 Classes de Renderização e Iluminação

A API MD3DM possui algumas classes que permitem a renderização de imagens. A descrição destas classes pode ser observada na Tabela 2.1.

Todos os objetos renderizados, são compostos por uma lista de vértices alocados na memória. Esta lista pode ser representada pela classe *VertexBuffer*, o que facilita a construção de objetos visto que esta classe permite total controle sobre os vértices, controlando desde o formato dos vértices até a forma como serão alocados na memória. Também é possível definir as primitivas de renderização, ou seja, a forma como os vértices serão desenhados. Ao todo, existem 6 tipos de primitivas disponíveis na API para isso:

[7] *Back buffer*, é uma área da memória onde as imagens são renderizadas antes de serem copiadas para o monitor do dispositivo.

- *PointList*: Renderiza os vértices como pontos, desenhados individualmente.
- *LineList*: Renderiza cada par de vértices como linhas individuais, sendo necessário uma quantidade par de vértices para isso.
- *LineStrip*: Renderiza os vértices com uma única linha contínua, onde a primeira linha é desenhada com os dois primeiros vértices e, cada segmento de linha subsequente é desenhado utilizando o ponto final da linha anterior como ponto inicial.
- *TriangleList*: Renderiza cada conjunto de três vértices como um único triângulo individual.
- *TriangleStrip*: É a forma mais utilizada para representação de objetos 3D, renderizando uma malha de triângulos onde cada triângulo forma uma face. Um exemplo desta primitiva pode ser observado na Figura 2.12.
- *TriangleFan*: É um formato muito parecido com a malha de triângulos porém, a única diferença é que cada triângulo compartilha o mesmo primeiro vértice, formando um leque.

Nome	**Descrição**
VertexBuffer	Classe que mantém na memória uma lista de vértices. Sua utilização consiste em uma seqüência de quatro operações: carga dos vértices, alocação do buffer, escrita das informações e liberação do buffer.
IndexBuffer	Representa um buffer que armazena os índices dos vértices. Cada índice representa um vértice e cada três vértices (um triângulo) representam uma face. As informações contidas no *IndexBuffer* descrevem como os vértices estarão interligados entre si para representar uma forma geométrica.
Mesh	Representa um objeto tridimensional. É composta por um *VertexBuffer* e um *IndexBuffer* que podem ser definidos manualmente, através da carga de um arquivo. Além desta opção, a classe *Mesh* possui algumas primitivas para renderização de objetos básicos como um cubo, uma esfera, um torus, um cilindro e um polígono.
Material	É uma estrutura (*struct*) que define as propriedades de um material para um determinado objeto como, a forma com que o objeto vai refletir a luz ambiente, a difusa e a especular.
Texture	Classe que representa a textura de um objeto e, geralmente, é carregada a partir de um arquivo de imagem através de outra classe chamada *TextureLoader*.

Tabela 2.1: Classes básicas da API MD3DM

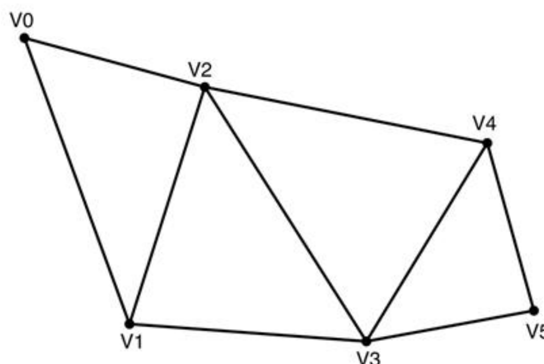


Figura 2.12: Exemplo de uma malha de triângulos

A alocação dos vértices na memória sem qualquer controle pode ser ineficiente e extremamente custosa já que em alguns casos, os vértices são repetidos várias vezes formando uma malha de triângulos. Para evitar que isso aconteça, é necessário utilizar um *IndexBuffer*. Esta estrutura, armazena os índices dos vértices alocados no *VertexBuffer* reduzindo a quantidade de vértices necessária para formar uma face, pois torna possível a repetição dos vértices sem a necessidade de realocá-los na memória.

A API MD3DM também possui uma classe para criação de luzes chamada *Ligth*. Essa classe possui vários parâmetros, como a posição, as cores difusa, ambiente e especular, a direção para onde irá irradiar, o tipo de luz, entre outros, que necessitam ser inicializados para que a luz seja criada. Apenas dois tipos de luzes são suportados: direcional que é uma luz que viaja em uma direção infinitamente e ponto ou posicional, que é uma luz que irradia em todas as direções igualmente, como uma lâmpada elétrica. Cada fonte de luz é associada a uma propriedade da classe *Device* chamada *Ligths*, que consiste de um array que armazena todas as luzes que serão utilizadas.

Todos os recursos utilizados pela API (luzes, meshes, etc) estão vinculados à classe *Device*, por isso qualquer alteração, como por exemplo, a mudança de tamanho da tela, exige que estes recursos sejam realocados na memória do dispositivo. Este processo pode ser feito automaticamente, sendo necessário apenas recarregar as informações no caso de um modelo tridimensional ou manualmente, cabendo ao desenvolvedor realocar a memória e recarregar todos os recursos.

Para iniciar a renderização de uma cena, é necessário preparar o dispositivo. Isto é feito em três etapas:

- Limpeza da tela: indicada pelo método *Clear* da classe *Device*, que efetua a atualização da tela, preenchendo o plano de fundo com a cor escolhida;

- Inicialização da cena: indicada pelo método *BeginScene* da classe *Device*, que

sinaliza que o dispositivo esta preparado para desenhar a cena;

- Finalização da cena: indicada pelo método *EndScene* da classe *Device*, que sinaliza o final da cena, ou seja, que nada mais vai ser desenhado liberando o dispositivo.

2.4.2 Classes de Transformação

A API MD3DM oferece alguns recursos para aplicar transformações nos objetos. Estes recursos são definidos por três estruturas básicas: *Matrix*, *Vector3*, *Quaternion*. Todas as transformações são aplicadas na propriedade *Transform* da classe *Device*, utilizando-se matrizes 4X4, representadas pela estrutura *Matrix* que possui vários métodos para a manipulação de matrizes e para aplicação transformações, conforme são descritos na Tabela 2.2. Esta propriedade possui vários membros, porém os mais importantes e utilizados são três membros, definidos como matrizes 4x4:

- *Projection*: segundo MILLER (2003), é uma transformação que define como a cena será projetada na tela, podendo ser representada como uma pirâmide com o topo cortado, com a área interna da pirâmide sendo a parte visível da cena, chamada de *view frustum*(volume de visualização) (Figura 2.13). É facilmente gerada pela função *PerspectiveFovLH* da estrutura *Matrix*, criando uma matriz de projeção perspectiva utilizando o *Field of View* (campo de visão), que é o ângulo de abertura da câmera em um sistema de coordenadas de mão-esquerda, conforme pode ser observado na Figura 2.14;

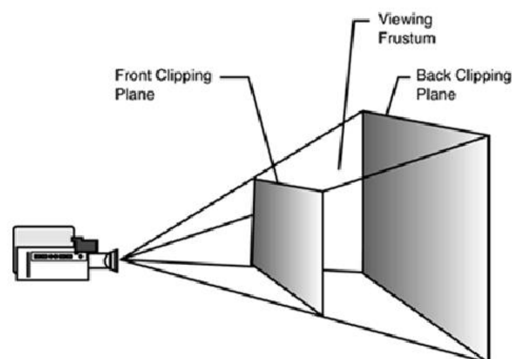
- *View*: contém as informações sobre a câmera, como a posição da câmera no espaço, o *target* (alvo) que é a posição para onde a câmera está apontada e, a direção que será considerada como *up* (para cima). Estas informações podem ser obtidas utilizando-se a função *LookAtLH* da estrutura *Matrix*;

- *World*: converte as coordenadas locais do modelo, onde todos os vértices são definidos relativos a origem local(centro do modelo), para as coordenadas de mundo, onde os vértices são definidos relativos a uma origem comum a todos os objetos na cena. Todas as transformações, que podem ser qualquer combinação de translações, rotações e escala, são aplicadas nessa matriz, individualmente para cada objeto. A classe *Matrix* provém muitas funções para criar estas transformações como *RotatioZ*, *Translation*, *Scaling*, entre outras.

No desenvolvimento de jogos, as matrizes são muito utilizadas para a aplicação de transformações como rotação, escala e translação. No entanto, no caso da rotação, a utilização de matrizes pode causar um problema chamado *Gimbal lock*, que ocorre

quando é aplicada rotações em eixos diferentes, onde uma rotação pode acabar anulando a outra, causando a perda da liberdade de movimentos em um dos eixos. Uma das alternativas que existem para evitar isso é a utilização de uma estrutura chamada *quaternion*. Segundo DUNN; PARBERRY (2002), esta estrutura é composta por um vetor de 3 dimensões que define o eixo de rotação dos objetos e um escalar, que representa o ângulo que o objeto esta rotacionado sobre este eixo. Ainda segundo DUNN; PARBERRY (2002), existem varias vantagens e desvantagens na utilização de *quaternions* como:

- **Interpolação suave.** Esta forma de representação de orientações possibilita a interpolação entre duas orientações, ao contrário das matrizes.
- **Rápida concatenação e inversão de deslocamentos angulares.** Essas operações podem ser feitas de forma muito mais eficiente utilizando-se *quaternions* ao invés de matrizes.
- **Representação Econômica.** Um *quaternion* requer muito menos memória para ser alocado do que uma matriz, porém ocupam mais espaço do que um vetor.
- **Podem se tornar inválidos.** Isto pode ocorre devido à entradas inválidas ou acumulação de imprecisão de ponto flutuante.
- **Dificuldade de trabalhar.**É a forma de representação que possui a utilização e visualização mais complexas.

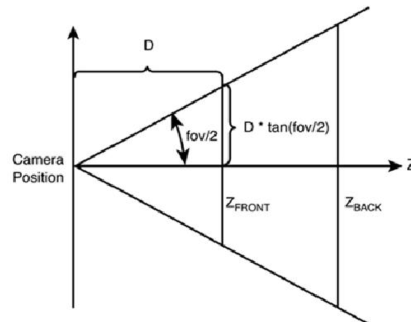


Fonte: MILLER (2003)

Figura 2.13: *View Frustum*

Na API MD3DM, os *quaternions* são representados por uma estrutura chamada *Quaternion*. Os principais métodos desta estrutura utilizados neste trabalho estão

descritos na Tabela 2.3. A estrutura *Vector3*, é a mais simples e fácil de ser utilizada. Representa um vetor de três dimensões(X,Y,Z), sendo utilizado basicamente para indicar posições. Também permite aplicar transformações, embora apenas uma seja utilizada neste trabalho, através do método *TransformCoordinate*, que recebe como parâmetro um vetor de três dimensões e uma Matriz de transformação, retornando o vetor transformado.



Fonte: MILLER (2003)

Figura 2.14: *Field of View*

Embora seja uma API, com muitos recursos e de fácil desenvolvimento, o MD3DM não é muito utilizado no desenvolvimento de jogos, devido a pouca popularidade dos dispositivos equipados com Windows Mobile. Sua versão nativa, utilizando C/C++, é um pouco mais completa e, por ter um desempenho superior ao da versão gerenciada, é a mais utilizada para o desenvolvimento de aplicações gráficas para este sistema operacional. Para o desenvolvimento de ferramentas como as *engines* 3D, a versão para *desktop* desta API é muito utilizada, entretanto, não há nenhuma disponível até o presente momento, que utilize o MD3DM ou sua versão nativa.

Nome	**Descrição**
Translation	Cria uma matriz de translação, a partir de um vetor de três dimensões.
Scale	Cria uma matriz de escala em torno de três eixos, representados por um vetor de três dimensões.
RotationX	Cria uma matriz de rotação em torno do eixo X. Através dos métodos *RotationY* e *RotationZ* é possível realizar essa mesma operação para os outros dois eixos.
RotationYawPitchRoll	Cria uma matriz de rotação, com uma rotação em torno do eixo Z(*Roll*), seguida por uma rotação em torno de X(*Pitch*)e, por uma rotação em Y(*Yaw*).
RotationQuaternion	Cria uma matriz de rotação a partir de um *quaternion*.
PerspectiveFovLH	Cria uma matriz de projeção perspectiva, no sistema de coordenadas mão-esquerda, baseado no campo de visão(*Field Of View*).
LookAtLH	Cria uma matriz de visualização no sistema de coordenadas mão-esquerda.
Identity	Representa uma matriz de identidade, com a diagonal igual a 1.

Tabela 2.2: Principais métodos da estrutura *Matrix* utilizadas neste trabalho

Nome	**Descrição**
RotationAxis	Cria um *quaternion* rotacionado em torno de um eixo indicado por um vetor de três dimensões e, um ângulo de rotação
RotationMatrix	Cria um *quaternion* a partir de uma Matriz de rotação.
Slerp	Cria um *quaternion* a partir de uma interpolação linear esférica entre dois quaternions.
Conjugate	Retorna o conjugado de um *quaternion*, ou seja, com a parte vetorial invertida.
Identity	Representa um *quaternion* identidade, que indica que não contém nenhuma rotação.

Tabela 2.3: Principais métodos da estrutura *Quaternion* utilizadas neste trabalho

3 ENGINE 3D

Neste capítulo serão descritos a arquitetura de uma *engine* 3D e, os seus principais componentes, que serão implementados neste trabalho.

3.1 Engine 3D

Uma *engine* é uma biblioteca de desenvolvimento responsável pelo gerenciamento de um jogo, tratando toda a parte de baixo nível como cálculos matemáticos, comunicação com hardware, renderização de imagens entre outros, servindo como base para o desenvolvimento do jogo. Sua utilização é tão importante que praticamente todos os jogos disponíveis atualmente utilizam uma ou várias *engines* específicas no processo de desenvolvimento. Segundo EBERLY (2001), "A *engine* deve tratar as questões de gerenciamento do grafo de cena através de uma interface que provê de forma eficiente a entrada da camada exibidora, podendo esta estar baseada em hardware ou software".

A arquitetura de uma *engine* 3D completa, conforme o proposto por PESSOA (2002) pode ser observada na Figura 3.1 e, os seus respectivos componentes são descritos na Tabela 3.1.

Fonte: PESSOA (2002)

Figura 3.1: Arquitetura de uma *engine* 3D.

Nome	Descrição
Gerenciador de Entrada	Encarregado de identificar eventos ocorridos no teclado e de encaminhá-los para o gerenciador principal.
Gerenciador Gráfico	Encarrega-se de realizar todo o processamento necessário para transformar a cena criada pelos objetos do jogo em dados que sejam suportados pelas rotinas do sistema para desenho na tela. Geralmente é implementado por uma biblioteca gráfica.
Gerenciador de Som	Responsável pela execução de sons a partir de eventos.
Gerenciador de IA	Gerencia o comportamento de objetos controlados pela máquina. Geralmente é desenvolvido em linguagens lógicas como Prolog.
Gerenciador de Múltiplos Jogadores	Tem como objetivo permitir que vários jogadores do mesmo jogo comuniquem-se entre si, deve lidar com o gerenciamento e a troca de informações entre os diversos computadores conectados via Internet.
Gerenciador de Objetos	Realiza o gerenciamento de um grupo de objetos do jogo. Isso envolve a carga dos objetos, o armazenamento em alguma estrutura de dados e o controle do ciclo de vida dos mesmos. Em geral, um jogo possui vários gerenciadores de objetos que além de suas funções normais, ainda precisam se comunicar.
Objeto do Jogo	Representa uma entidade que faz parte do jogo (tais como, carro, bola, etc) e todas as informações necessárias para que seja gerenciada. Esses dados envolvem, por exemplo, posição, velocidade, dimensão, detecção de colisão, o próprio desenho, entre outros.
Gerenciador do Mundo	Responsável por armazenar o estado atual do jogo e para isso utiliza o gerenciador de objetos. Em geral é associado com um editor de cenários que descreve o estado inicial do mundo em cada nível do jogo
Editor de Cenários	Ferramenta para a descrição de estados do jogo de forma visual sem a necessidade de programação que posteriormente serão carregados pelo gerenciador do mundo .
Gerenciador Principal	Indica qual objeto vai processar qual informação, representa a “fachada”de todo o projeto, no sentido em que é um local de acesso único ao sistema.

Tabela 3.1: Componentes da arquitetura de uma *Engine*

A principal vantagem da utilização de uma *engine* é o tempo poupado pela sua reutilização. Uma vez feita, ela pode ser utilizada para todos os jogos que serão desenvolvidos posteriormente. Além disso, novas funcionalidades podem ser adicionadas à *engine*, à medida que são necessárias. Normalmente uma *engine* 3D completa é composta por varias *sub-engines*, responsáveis por funções específicas como a detecção de colisão, renderização, inteligência artificial, animação dos objetos, entre outras.

Atualmente, é possível encontrar algumas *engines* 3D proprietárias para dis-

positivos móveis como *Unit3D* (UNITY TECHNOLOGIES, 2009) para o iPhone, *EDGELIB* (ELEMENTS INTERACTIVE, 2009) multiplataforma, a *Galatea Engine* (MINDPOOL CORPORATION, 2009) para Windows Mobile e algumas *engines open source*, a maioria projetos acadêmicos ou de pesquisa. Embora a utilização de uma *engine* seja fundamental para a criação de um jogo, o seu desenvolvimento é quase sempre mais complicado do que o do próprio jogo, devido a complexidade envolvendo cada uma de suas funcionalidades. Por esse motivo, neste trabalho serão implementadas apenas algumas funções básicas da *engine* como os gerenciadores de entrada, gráfico, de objetos e de mundo utilizando para isso o *framework* .Net CF.

3.2 Gerenciador de Entrada

O gerenciador de entrada é o responsável por tratar os eventos gerados por um teclado, mouse ou qualquer outro dispositivo de entrada que o aparelho possua. Isto é feito utilizando-se *event handlers*[1], que no .Net CF são representados pelas *delegates*[2] *KeyboardEventHandler* e*MouseEventHandler*.

Cada uma dessas *delegates* pode ser associada a um controle especifico, permitindo o tratamento do evento. Existem inúmeros tipos de controles disponíveis para o tratamento de eventos, no entanto os mais utilizados são aqueles responsáveis por detectar quando uma tecla é pressionada e o controle do *touchscreen*, que é feito em três etapas:

- *MouseDown*: detecta o toque na tela e registrando a posição da tela onde isto ocorreu;
- *MouseMove*: detecta a movimentação na tela, capturada após o toque inicial, atualizando a posição em x,y do toque instantaneamente;
- *MouseUp*: detecta o final do movimento, quando o toque é encerrado, registrando a posição final na tela em que isto ocorreu.

Os eventos responsáveis por identificar quando uma tecla é pressionada chamam-se: *KeyDown*, que detecta quando a tecla é pressionada identificando o valor desta tecla e, *KeyUp* que detecta quando a tecla deixa de ser pressionada. Alguns dispositivos que possuem sensores de movimento como o acelerômetro, oferecem bibliotecas para a detecção e tratamento dos eventos gerados pelo sensor. Entretanto, devido a uma falta de padronização no mercado, principalmente com relação ao teclado já

[1] *Event Handler*, é uma sub-rotina que manipula eventos como, a entrada de dados proveniente de um teclado ou mouse ou, gerados pelo próprio programa

[2] *Delegate*, é uma estrutura de dados que representa uma função que pode ser passada como parâmetro para outra função

que alguns aparelhos possuem um teclado físico do tipo QWERTY, outros apenas um teclado numérico, alguns apenas o teclado virtual, é necessário desenvolver um gerenciador de entrada, considerando as diferenças que existem entre os modelos disponíveis. Na Figura 3.2, é possível observar alguns tipos de teclados disponíveis.

Figura 3.2: Dois tipos de teclado, virtual à direita e físico à esquerda.

3.3 Gerenciador de Objetos

O gerenciador de objetos é responsável pela carga dos arquivos, através da leitura de um arquivo de texto ou binário que contém toda a informação sobre como desenhar o objeto. Atua também no gerenciamento do ciclo de vida destes objetos carregados, realizando a alocação ou a desalocação dos mesmos na memória. Segundo (PAMPLONA, 2005), em uma *engine* simples, o gerenciador invoca métodos que devem ser implementados pelo usuário. Cada um destes métodos indicará ações ou eventos acontecidos no jogo como, por exemplo: tiros, colisões entre outros. Uma *engine* costuma oferecer funções para carregar objetos em inúmeros formatos , facilitando a tarefa de desenvolvimento de um jogo. Dentre os formatos de arquivos mais utilizados para a representação de modelos 3D, dois serão utilizados neste trabalho: o *Wavefront*(BOURKE, 1996) e o MD3DM.

3.3.1 O Formato *Wavefront*

O *Wavefront* é um formato padrão para armazenar modelos 3D. É largamente utilizado por ser uma especificação muito simples, totalmente aberta e reconhecida no mundo inteiro. Os arquivos *Wavefront* são arquivos de texto puro, com a extensão *.obj* podendo ser facilmente editados em qualquer editor de textos comum. Cada

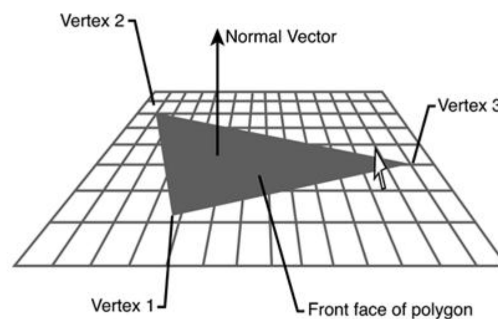
linha do arquivo representa um item, como coordenadas de textura, normais, vértices e faces que compõem um polígono. Um exemplo de um arquivo .obj pode ser observado na Tabela 3.2. Este arquivo possui os seguintes elementos:

- #: indica um comentário, podendo ser completamente ignorado;
- v <x> <y> <z>: representa um vértice tridimensional com as coordenadas x,y,z. Cada vértice é enumerado sequencialmente a partir de 1 e desde o início do arquivo, servindo como índice para a face.
- vn <x> <y> <z>: representa um vetor normal com as coordenadas x,y,z que são utilizadas para definir como será aplicada à iluminação no vértice. Cada vetor normal é associado a um único vértice sendo enumerado da mesma forma. Na Figura 3.3, é possível observar um vetor normal, em conjunto com uma face.
- vt <u> <v>: representa as coordenadas de textura (Figura 3.4) definindo como ela será aplicada à face. Da mesma forma que os elementos anteriores, também é enumerado sequencialmente a partir de 1 e, o valor dos parâmetros varia de 0 até 1.
- g <nome>: determina um grupo, ou seja, tudo o que é lido após o g pertencerá a um determinado grupo, como as texturas e o conjunto de faces. Com a utilização de um grupo, é possível desenhar o modelo em partes;
- f: representa uma face, que normalmente segue uma estrutura triangular, ou seja, é composta por três vértices, conjuntamente com seus respectivos vetores normais e coordenadas de textura. É muito flexível, podendo assumir diversos formatos, sedo que o mais completo deles é <v1>/<vt1>/<vn1> <v2>/<vt2>/<vn2> <v3>/<vt3>/<vn3>;
- mtlib <arquivo>: indica que será importado um outro arquivo, contendo o material para o modelo. O arquivo importado tem a extensão *mtl* que segue o formato *Wavefront MTL*(RAMEY; ROSE; TYERMAN, 1995), que será descrito posteriormente;
- usemtl <nome>: indica que o grupo atual utilizará um material com o nome indicado no parâmetro.

O arquivo no formato *Wavefront MTL*, segue uma especificação muito parecida com a do anterior(Tabela 3.3), tendo a extensão *.mtl*. O item *newmtl* indica que um novo material será criado e, todos os itens que vierem posteriormente a ele e antes do próximo farão parte do mesmo material. Os demais elementos podem ser descritos como:

```
#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>
```

Tabela 3.2: Exemplo de um arquivo *Obj*



Fonte: MILLER (2003)

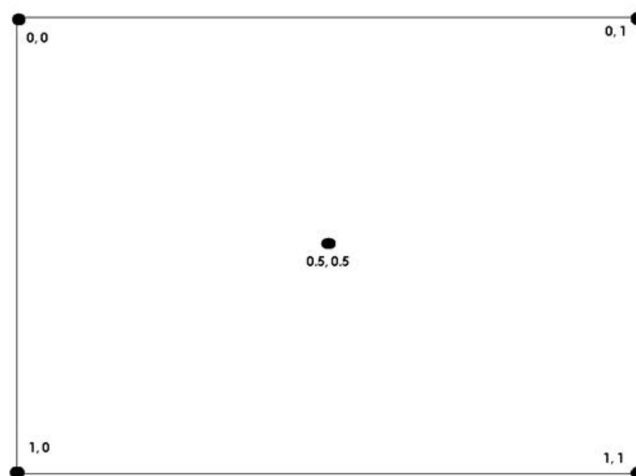
Figura 3.3: Vetor normal em uma face

- `Ka <r> <g> <b>`: define a cor ambiente do material em RGB;
- `Kd <r> <g> <b>`: define a cor difusa do material em RGB;
- `Ks <r> <g> <b>`: define a cor especular do material em RGB;
- `d <alpha>`: indica o nível de transparência, valor alpha, do material;
- `map_Ka <arquivo>`: importa um arquivo de imagem que representa a textura afetada pela cor ambiente do material;
- `map_Kd <arquivo>`: importa um arquivo de imagem que representa a textura afetada pela cor difusa do material;
- `map_Ks <arquivo>`: importa um arquivo de imagem que representa a textura afetada pela cor especular do material;

Com esta especificação detalhada do formato *Wavefront*, é muito fácil desenvolver um algoritmo para carregar um arquivo *.obj*, passá-lo para um *Mesh* e, visualizar o modelo em 3D.

```
#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>
```

Tabela 3.3: Exemplo de um arquivo *Mtl*



Fonte: MILLER (2003)

Figura 3.4: Coordenadas de textura

3.3.2 O Formato MD3DM

O formato MD3DM, é o padrão do *Direct3d Mobile* para modelos 3D. É um subconjunto do formato *.x file* (BOURKE, 1999), o padrão do *Direct3D* para *desktop*, com a extensão *.md3dm* e, foi desenvolvido pois o formato *.x* era complexo demais para ser utilizado em dispositivos móveis, sendo uma versão simplificada do mesmo.

Diferentemente de sua versão para *desktop*, que possui um método para carregar um modelo a partir de um arquivo, o MD3DM não possui nenhuma função para isto. Este problema foi resolvido com o lançamento de uma série de exemplos (MICROSOFT, 2007), disponibilizados juntamente com o CF pela Microsoft, onde constava um algoritmo que efetuava a conversão de um arquivo *.x* para o formato *.md3dm* e também, realizava a carga deste arquivo para um *Mesh*. Como este algoritmo realiza algumas simplificações os modelos não possuem normais, o que impede que sejam aplicados efeitos de iluminação e, perdem algumas texturas.

A normais tem que ser computadas manualmente, já que o MD3DM não possui nenhuma primitiva que faça essa operação.

No entanto, este formato foi projetado especificamente para dispositivos móveis, sendo um arquivo muito mais leve do que o formato *Wavefront*. Em termos de performance, isto acaba fazendo a diferença, pois como o modelo é simplificado, ele possui menos polígonos a serem desenhados. Além disso como praticamente todas as ferramentas de modelagem existentes exportam arquivos no formato *.x* o que, com a possibilidade de conversão, torna possível encontrar e modelar um grande número de objetos diferentes. Um exemplo de um modelo carregado neste formato pode ser observado na Figura 3.5.

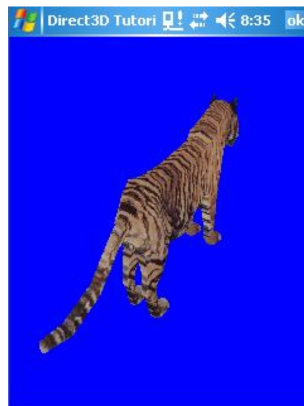


Figura 3.5: Exemplo de um *Mesh* no formato *.md3dm*.

3.4 Gerenciador Gráfico

O gerenciador gráfico é o responsável por todo o processamento necessário para transformar a cena criada pelos objetos do jogo em dados que sejam suportados pelas rotinas do sistema para desenho na tela. Esta tarefa geralmente é desempenhada por uma API gráfica como no caso deste trabalho onde será utilizado o MD3DM. Esta API segue um *pipeline gráfico*[3](MICROSOFT, 2008b) composto por 4 etapas: transformação, iluminação, rasterização e operações por pixel.

A primeira etapa do *pipeline* é a transformação. Este processo pega as coordenadas em um sistema cartesiano homogêneo e produzir coordenadas que são apropriadas para sua utilização. Conforme visto anteriormente, o MD3DM possui várias primitivas para a realização deste processo. Nesta fase também é realizada operações como o corte de visão (*culling*), que remove as arestas do objeto fora do campo de

[3] *Pipeline gráfico*, descreve o fluxo de funcionamento do processamento gráfico, a transformação de uma cena em uma imagem

visão da câmera, e o calculo das normais. Após a aplicação da transformação, inicia a segunda etapa, a iluminação. Geralmente os dois processos atuam em conjunto já que a iluminação é necessária para a visualização da cena.

O MD3DM simula efeitos de iluminação baseados em vértices. Quando a iluminação esta habilitada, parâmetros de iluminação são utilizados para computar a cor difusa e especular de cada vértice. A iluminação depende do tipo de material e de iluminação utilizados na cena.

A transformação e a iluminação são passadas para o rasterizador, a terceira etapa. A rasterização é o processo que analisa os vértices que compõem a primitiva, para gerar pixels que serão cobertos por ela. Para pixels que se encontram entre os vértices, os valores dos vértices são interpolados ao redor deles, gerando uma cor para cada um. Estes valores são passados à unidade de processamento de pixels.

Com a rasterização, também é possível adicionar efeitos como a neblina (*Fog*), tipicamente utilizada para obscurecer objetos próximos ao plano mais distante do espaço visível (*frustum*), para que eles não apareçam de repente no meio da cena, quando cruzarem a fronteira daquele plano. Quando o rasterizador tiver computado o valor de saída de cada pixel, este valor passa por uma série de operações que realizam o processamento da cor exata para cada um individualmente, calculada a partir dos níveis de iluminação computados durante a primeira fase e os valores atribuídos calculados durante a rasterização.

Esta é a última fase do *pipeline* e, ao final do processamento, os pixels são escritos nas coordenadas apropriadas dentro de um buffer ou, como no caso dos dispositivos móveis diretamente na tela. O *pipeline* é executado automaticamente pela API, bastando ao desenvolvedor definir quais os tipos de técnicas e transformações que serão utilizadas durante o processo.

3.5 Gerenciador de Mundo

O gerenciador de mundo é o responsável por armazenar o estado atual do jogo e, conta com vários gerenciadores para executar tal tarefa. Normalmente ele esta associado a um editor de cenários, que não será abordado neste trabalho. Este gerenciador é o responsável por delegar tarefas para o gerenciador de objetos, a fim de carregar os objetos de acordo com as requisições.

As funções exercidas por este gerenciador podem ser distribuídas entre vários objetos. Tendo uma classe responsável pela visualização dos objetos, outra pela carga dos objetos e uma terceira responsável por identificar os eventos, repassá-los aos seus respectivos destinos e inicializar a cena, é possível realizar tal distribuição. Isto será descrito posteriormente neste trabalho no capítulo da implementação.

4 IMPLEMENTAÇÃO

Este capítulo mostrará o processo de implementação de uma *engine* 3D, e o desenvolvimento de algumas aplicações de teste.

4.1 Estrutura e organização do código fonte

Todos os nomes dos métodos e variáveis que foram implementados, estão em inglês, para manter um padrão de código próximo ao do .NET CF e da API MD3DM. Para facilitar o entendimento, todos os métodos e algumas váriaveis receberam comentários em português, explicando o que cada um faz. Existem duas formas de se fazer isto em C#:

- // indica um comentário de linha, é utilizado para explicar determinados trechos do código e algumas variáveis;
- /// indica um comentário de um método, sendo inserido antes dele, indica uma breve descrição de como o método funciona e, o significado de cada parâmetro, se ele possuir algum.

Além dos comentários, também existe a possibilidade de delimitar um determinado trecho de código ou um agrupamento de métodos, utilizando-se uma função chamada *region*. Esta função omite o trecho que esta dentro dela substituindo por uma legenda, o que torna o código muito mais limpo e organizado. Além disso, algumas subrotinas também serão utilizadas para reduzir o tamanho de cada função implementada, distribuindo o processamento entre várias funções diferentes.

4.2 Metodologia e técnicas utilizadas

O desenvolvimento deste trabalho foi feito seguindo dois princípios: *Keep It Simple, Stupid*(KISS)(ALMQVIST, 2001) e *You Aren't Going to Need It*(YAGNI)(FOWLER, 2004). Estes princípios pregam que alguma coisa só deve ser realmente feita se for necessária e que nunca se deve implementar algo que se ache que irá precisar futuramente. Como este trabalho é voltado para dispositivos móveis, que são equipamentos

limitados em termos de memória, a simplicidade é fundamental para se ter um bom desempenho, o que torna válida a aplicação destes princípios.

Esta simplicidade também se refere à utilização de algumas técnicas de programação como a redução do numero de heranças e métodos sobrescritos e, a ausência de padrões de projeto, desnecessários, pois tornam o desenvolvimento mais complexo atrapalhando o desempenho. Isto dá liberdade ao desenvolvedor para adicionar e remover funcionalidades da aplicação, sem a necessidade de seguir um padrão específico. Dentre as técnicas de programação utilizadas neste trabalho, visando à eficiência e o menor tempo de processamento, destacam-se:

- Utilização do tipo *float* para representação de números em ponto flutuante o que, apesar de ter uma precisão menor do que um tipo de precisão dupla ou maior, é mais eficiente devido a quantidade menor de memória que ele utiliza. Além disso, este tipo é capaz de suprir todas as necessidades deste trabalho, não havendo a necessidade de se trabalhar com precisões maiores;

- Passagem de parâmetros por referência, utilizada principalmente na carga dos modelos tridimensionais, por ser mais eficiente do que a passagem por cópia, já que não há perda de tempo na duplicação dos dados, diminuindo a quantidade de memória requerida;

- Utilização da programação dinâmica, ou seja, o armazenamento de valores que são constantemente utilizados e, que possuem uma baixa frequência de modificação evitando cálculos desnecessários.

- Controle de acesso dos atributos, através dos métodos *get* e *set*, permitindo uma maior segurança já que os atributos não podem ser acessados diretamente fora da classe onde foram criados, apenas através desses métodos.

Nas próximas seções serão listadas as ferramentas utilizadas na composição deste trabalho, os requisitos que a aplicação deverá contemplar e, os detalhes referentes aos processos de desenvolvimento.

4.3 Ferramentas utilizadas no desenvolvimento

O trabalho foi desenvolvido utilizando o *Visual Studio* 2008. Foi utilizado o SDK para o desenvolvimento de aplicações voltadas à dispositivos equipados com o Windows Mobile 6.0 ou superior. Este SDK contêm bibliotecas de componentes visuais, classes e emuladores. Além disso, foram utilizados o .NET CF 3.5 e a API MD3DM, descritos anteriormente. Todas essas tecnologias são integradas o que facilitou o desenvolvimento. A linguagem de programação utilizada neste trabalho foi o C#.

Os testes realizados foram feitos em um *notebook*, através de um emulador e, em um *smartphone*. Os detalhes referentes à configuração destes equipamentos realizados pode ser conferidos na Tabela 4.1. Para a modelagem das classes apresentadas neste trabalho, foi utilizado o próprio *Visual Studio*, que permite que todo este processo seja feito diretamente facilitando a geração dos diagramas. Os outros diagramas apresentados, foram feitos com o Microsoft Visio 2007(MICROSOFT, 2009e).

Dispositivo	**Sistema Operacional**	**Memória RAM**	**Processador**
Powernote SR90	Windows Vista 64bit	2 GB	Intel Core 2 Duo 2.20 GHz
HTC Tytn II	Windows Mobile 6.1	128 MB	Qualcomm 7200 400 MHz

Tabela 4.1: Dispositivos utilizados no desenvolvimento e testes

Para a modelagem 3D foram utilizadas duas ferramentas, *Art Of Illusion*(EASTMAN, 2009) para os arquivos no formato *Wavefront* e o 3D *Studio Max*(AUTODESK, 2009). Ambas possuem diversos recursos como a definição formas geométricas pré-definidas, aplicação de materiais e texturas, a definição de câmeras, entre outros e são de fácil utilização. Esta monografia foi redigida com o auxílio do editor de textos TeXnicCenter(TEXNICCENTER, 2008) e do sistema de preparação de documentos Latex(LATEX PROJECT TEAM, 2009).

4.4 MDGE

O presente trabalho trata do desenvolvimento de uma *engine* 3D para dispositivos móveis. Este projeto, chamado *Mobile Device Game Engine* (MDGE) faz acesso direto à API gráfica MD3DM e ao .NET CF dando uma certa flexibilidade ao desenvolvedor, permitindo o desenvolvimento de algumas aplicações gráficas básicas. O acesso as funções do sistema operacional é efetuado através do CF. Foram implementadas algumas funções básicas para carregar modelos 3D, visualizar o ambiente, permitindo também a movimentação pelo cenário, aplicar transformações nos objetos e, renderizar algumas formas geométricas pré-definidas.

Apesar destas funções básicas, a MDGE também permite a incorporação de novas funcionalidades, adicionadas as já existentes ou, até as substituindo. Isto é indispensável, pois cada aplicação pode ter necessidades específicas que ainda não foram incorporadas a *engine*. Além disso, por se tratar de um protótipo com algumas limitações, a implementação novas funcionalidades é fundamental para que a *engine* seja aperfeiçoada.

A arquitetura da MDGE, descrita na Figura 4.1, está dividida em 5 grandes

componentes: *Loader*, *Ligths*, *Objects*, *Transforms* e *World*.

Fazendo uma analogia com a “arquitetura de uma *engine* 3D”, apresentada no Capítulo 3, os componentes *Loader*, *Objects* e *Transforms* executam as funções do gerenciador de objetos, o *Form*, que é a interface da aplicação, atua como gerenciador principal identificando os eventos de entrada, distribuindo para a respectiva classe, além de realizar a atualização da tela. Os componentes *World* e *Ligths* fazem o papel de gerenciador de mundo, pois permitem a visualização da cena. A API MD3DM atua como gerenciador gráfico, em conjunto com o .NET CF.

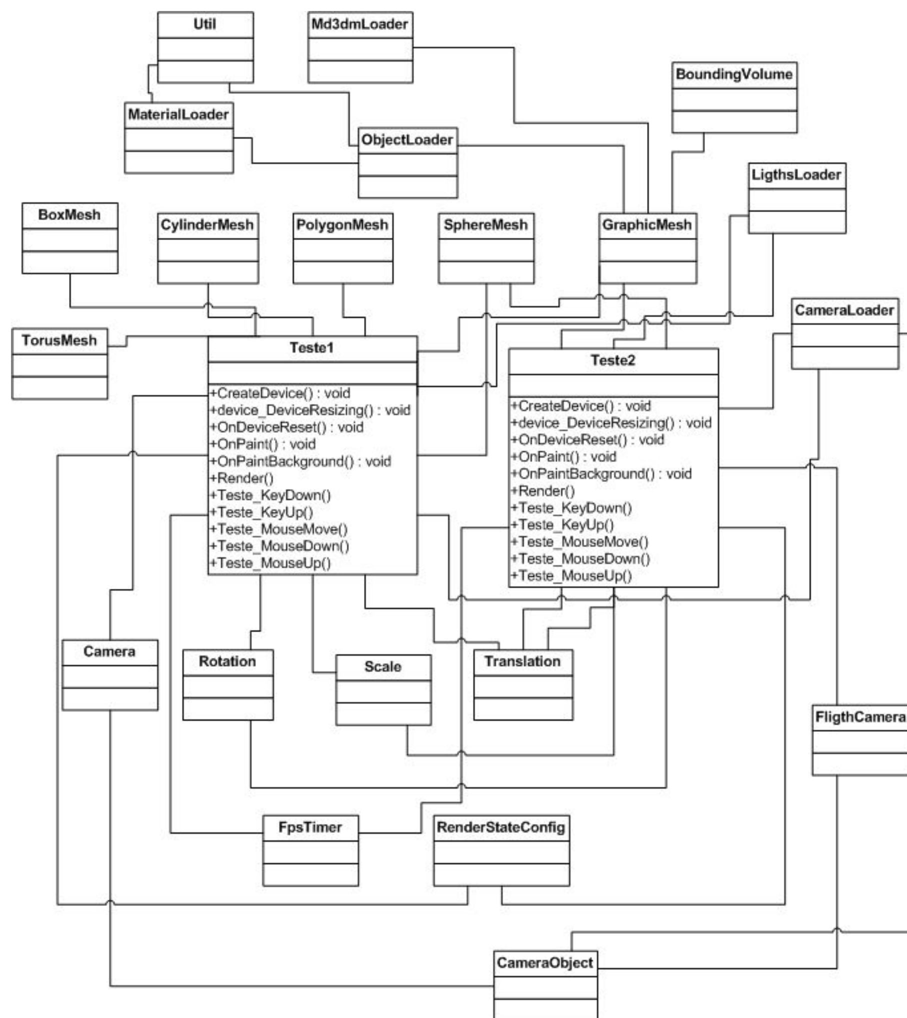


Figura 4.1: Arquitetura da MDGE

4.4.1 Loader

O componente *Loader* encapsula as funções necessárias para carga de modelos 3D nos formatos *Wavefront* e *.md3dm*. Conforme visto anteriormente, o primeiro formato é composto por dois arquivos de texto com a extensão *.obj* e *.mtl*, que serão tratados separadamente, em classes diferentes. Já o segundo formato, será carregado conforme a especificação fornecida pela Microsoft.

O processo de carga de um arquivo .obj é realizado pela classe *ObjectLoader* descrita na Figura 4.2, através do método *CreateObjMesh* onde o arquivo é lido e carregado na memória. Este processo é realizado em três etapas representadas pelos métodos *LoadVertices*, *AddFacesToBuffer* e *LoadMaterials*.

Na primeira delas, o arquivo é lido linha por linha, identificando os vértices, os vértices normais, as coordenadas de textura e faces, armazenando cada um em uma lista especifica. Para facilitar a representação de cada componente, foram utilizadas três estruturas: *Vertex* que representa os vértices e os vértices normais, *VertexTexture* que representa as coordenadas de textura e *Face* que representa uma face. Embora não siga o padrão da orientação de objetos a utilização de estruturas para a representação de objetos pequenos ao invés de classes é mais eficiente, pois menos memória tem que ser alocada para cada objeto. Dentro deste método também é realizada a inicialização dos *arrays* que representam o *VertexBuffer* e o *IndexBuffer* utilizados para desenhar as faces e, a chamada dos métodos que representam as próximas etapas.

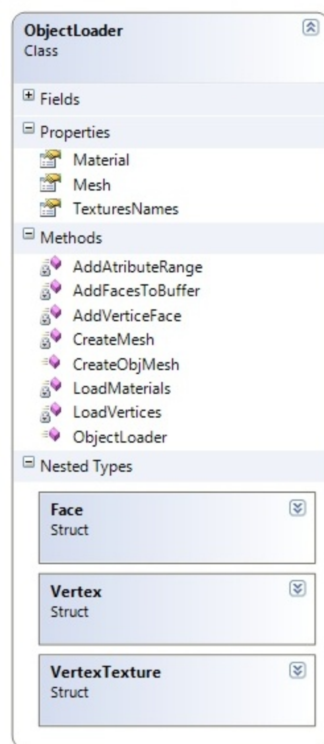


Figura 4.2: Classe *ObjectLoader*

A segunda etapa é representada pelo método *AddFacesToBuffer*. Nesta etapa, os vértices são adicionados ao *VertexBuffer* de acordo com a ordem em que se encontram na face e o *IndexBufer* é atualizado. Cada vértice está armazenado em sua respectiva lista, assim como cada face. É possível definir o formato dos vértices que serão adicionados no *buffer* através da enumeração *VertexFormats* presente na

API MD3DM. Isto permite que se trabalhe com um formato flexível, como no caso desta aplicação onde o formato do vértice segue o padrão posição/normais/textura.

Na última etapa, os materiais e texturas são carregados dentro do método *LoadMaterials*, através da utilização de uma instância da classe *MaterialLoader*, descrita na Figura 4.3 criada sempre que o elemento *mtlib* for encontrado e, no final deste processo um objeto *Mesh* é criado com todas as informações carregadas. Dentro desta classe o arquivo *.mtl* é carregado da mesma forma que o anterior, lendo o arquivo linha por linha, criando os materiais e carregando os arquivos de textura.

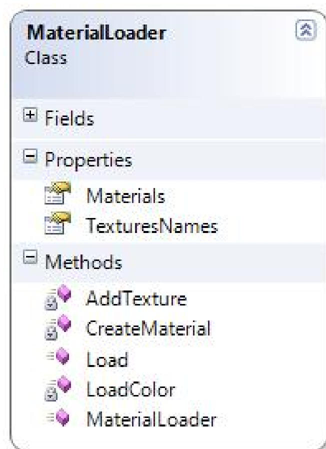


Figura 4.3: Classe *MaterialLoader*

A classe que realiza a carga dos arquivos *.md3dm* se chama *Md3dmLoader*, descrita conjuntamente com a classe *Util* na Figura 4.4. Ela atua de uma forma muito parecida com as classes anteriores, sendo que a única diferença esta na leitura do arquivo, que possui uma especificação diferente e, está em um formato binário, ao invés de arquivo texto. O método *LoadMesh* realiza esta tarefa, gerando um *Mesh* composto pelas informações lidas. A carga dos materiais e texturas também é efetuada neste método não requerendo nenhuma classe extra para isso.

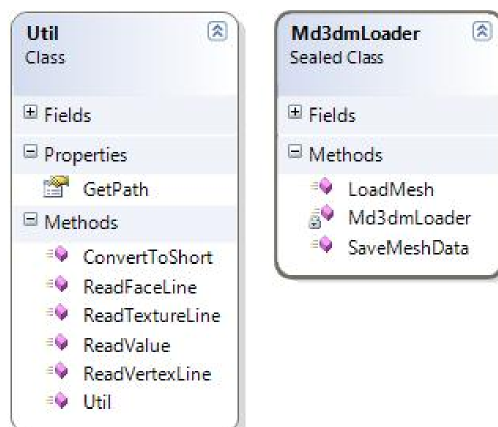


Figura 4.4: Classes *Util* e *Md3dmLoader*

A classe *Util*, encapsula os métodos de conversão de uma linha em formato *string*, para um formato específico como um tipo *Vertex* ou *Face*. Todos os métodos dessa classe recebem uma *string* como parâmetro e retornam um tipo específico, por referência para tornar o processo mais eficiente.

4.4.2 Ligths

O componente *Ligths* realiza a carga das luzes que serão utilizadas na cena. Como a API MD3DM possui uma classe que representa uma luz, chamada de *Ligth*, não foi necessário implementar nenhuma classe ou estrutura extra pra isso. Os principais atributos desta classe são lidos a partir de um arquivo XML[1], o que facilita o processo de leitura, já que os dados estão organizados em *tags* onde cada uma, representa um atributo. Um exemplo de um arquivo XML utilizado neste trabalho pode ser observado na Figura 4.5.

```
<mdge>
  <ligths>
    <add type="directional" position="0,10,0" difuse="255,255,255" attenuation="0.2" range="1000.0"
         ambient="255,255,255" direction="0,-10,0"/>
    <add type="point" position="0,8,-2" difuse="255,255,0" attenuation="0.4" range="1000.0"
         ambient="255,255,0" direction="0,-8,0"/>
  </ligths>
</mdge>
```

Figura 4.5: Arquivo XML de luzes

O processo de carga dos atributos é realizado dentro do método *ReadFile* da classe *LigthsLoader*, demonstrada na Figura 4.6. Os atributos são convertidos para o formato desejado com o auxílio dos métodos *ReadVector* e *ReadColor*. Logo após, é criada uma lista de objetos do tipo *Ligth*, passando os atributos carregados para ela, no método *AddLigth*. Em seguida, o objeto é adicionado a uma lista que será utilizada ao término da leitura para inicializar as luzes do dispositivo. Caso haja algum problema na leitura do arquivo de luzes ou, caso o nome do arquivo de origem não seja passado, é criada uma luz padrão na cor branca para iluminar a cena. A cores estão definidas no formato RGB[2], que permite a representação de praticamente todas as cores existentes.

Para que a luz seja corretamente adicionada a cena, é necessário recarregar as luzes do dispositivo antes de cada processo de desenho da cena. Isto é feito através do método *SetupLigths*, que recebe como parâmetro o próprio dispositivo e, efetua a atualização das luzes, utilizando os valores armazenados na lista de luzes. Também é possível desativar as luzes, bastando para isso definir a variável *Enabled* da classe

---

[1] *Extensible Markup Language*, ou linguagem de marcação extensível, é um formato para a criação de dados organizados de forma hierárquica, como em documentos de texto formatados, imagens vetoriais ou bancos de dados.

[2] RGB é um sistema de cores aditivas formada por Veremlho(R), Verde(G) e Azul(B) que variam numa escala de 0 a 255

Ligth como falso embora, isso não seja aconselhável, pois é necessário ao menos uma luz para renderizar a cena ou, caso contrário, a cena ficará totalmente escura. Isto pode ser melhor observado na Figura 4.7

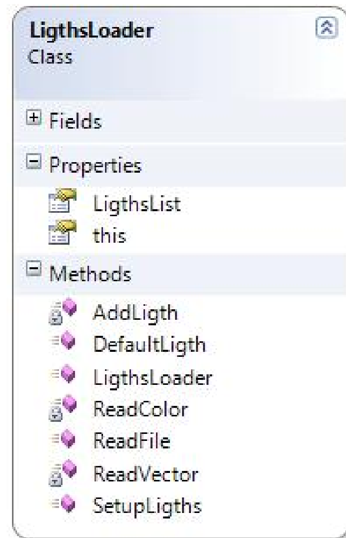


Figura 4.6: Classe *LigthsLoader*

Figura 4.7: Iluminação desativada à esquerda e ativada à direita

4.4.3 Objects

O componente *Objects*, encapsula algumas funções básicas da API MD3DM para criação de objetos 3D. Dentro deste componente também estão as classes que realizam a renderização dos objetos carregados e, a que efetua o cálculo do volume dos objetos.

A classe *Mesh*, da própria API, possui métodos para a criação de formas geométricas em 3D. Entretanto, estas funções são complicadas de serem utilizadas e, por isso, foram encapsuladas dentro de classes específicas. Existem 5 formas básicas disponíveis:

cilindro, esfera, cubo, torus e polígono. Cada uma é representada por uma classe com o mesmo nome e, o processo de criação destes objetos é muito parecido. Primeiro define-se alguns parâmetros básicos, como raio, comprimento, quantidade de fatias(vértices), entre outros e depois, a cor com a qual o objeto será renderizado. Caso não seja sinalizada nenhuma cor, os objetos serão renderizados na cor padrão vermelha. A cor, neste caso, é representada pela classe *Color* do CF.

Após a criação da forma escolhida, é possível renderiza-la na posição escolhida ou, sem definir a posição, com o objeto sendo desenhado na origem (coordenadas igual a zero), ambas realizadas pelo método *Render* existente dentro da respectiva classe. A primeira opção é mais utilizada para renderizar um objeto estático já a segunda, dá liberdade para a escolha da posição sendo mais utilizada para objetos com movimento. As 5 classes que representam as formas geométricas se chamam: *SphereMesh, PolygonMesh, BoxMesh, CylinderMesh,* e *TorusMesh*. A descrição de duas dessas classes pode ser conferida na Figura 4.8, sendo que as demais seguem basicamente o mesmo modelo, com diferenças apenas na forma de criação do objeto.

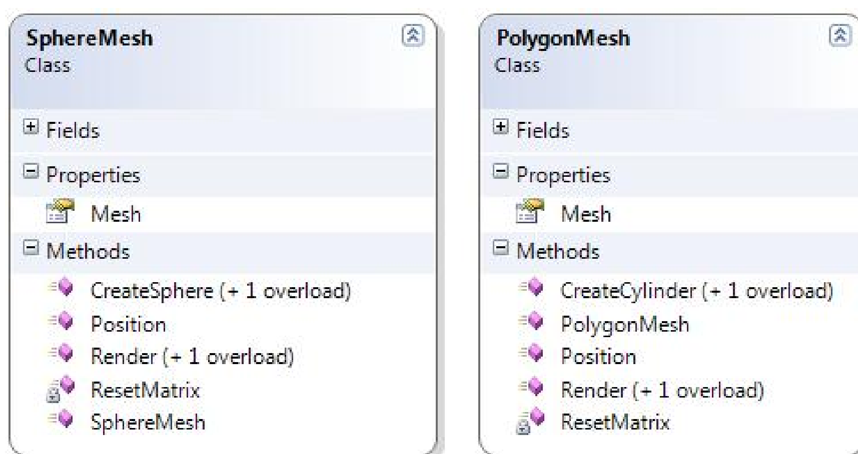


Figura 4.8: Classes *SphereMesh* e *PolygonMesh*

A classe *GraphicMesh*, demonstrada na Figura 4.9 , encapsula os métodos para carga de arquivos, do componente *Loader*. Como os arquivos à serem lidos podem estar em formatos diferentes e, possuem várias informações como texturas, materiais e um *Mesh*, seria muito complicado trabalhar com eles diretamente. Quando uma instância dessa classe é criada, é passado por parâmetro o nome do arquivo que será lido. Como este arquivo pode estar em dois formatos diferentes, a extensão do arquivo é testada. Se for *.md3dm*, o método interno que realizará a leitura será o *LoadMesh*, caso contrário, sendo *.obj* o método será o *LoadObjMesh*. Como estes métodos são internos, ou seja, somente podem ser acessados de dentro da própria classe, há um método externo que indica que o arquivo deve ser carregado chamado *LoadFile*. Dentro desse método também é realizado o cálculo do volume do objeto.

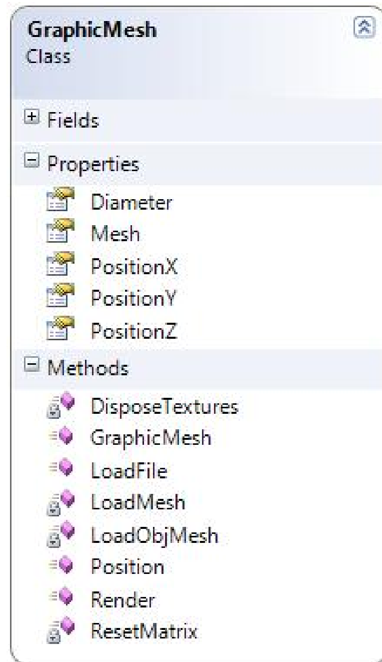


Figura 4.9: Classe GraphicMesh

As informações lidas são armazenadas em variáveis globais, dentro da própria classe. Apenas o *Mesh* e o diâmetro do objeto são acessíveis de fora. Isto é feito para proteger as informações, visto que não é possível alterará-las fora da classe. A carga dos arquivos deve ser efetuada sempre que o dispositivo for reinicializado. O método para a renderização do objeto se chama *Render* e, desenha o *Mesh*, em partes, chamadas *SubSets*, cada uma com uma textura e um material diferente. Ao final deste processo, um objeto 3D texturizado será exibido na tela. Como a maioria dos modelos carregados são muito “grandes”, para serem desenhados na janela do dispositivo, todas as operações de escala, translação e rotação devem ser feitas fora desta classe.

A última classe que integra este componente é a classe *BoundingVolume*, descrita na Figura 4.10. Esta classe possui dois métodos para efetuar o cálculo do volume que cerca um objeto 3D. O primeiro método se chama *ComputeBoundingBox* e, efetua o cálculo de um cubo, gerado ao redor do objeto, retornando os cantos inferior e superior do objeto. O segundo se chama *ComputeBoundingSphere* e, efetua o cálculo da esfera que rodeia o objeto, retornando o centro da esfera, que representa o centro do próprio objeto e, o raio. Esses cáculos são efetuados, com o auxílio da classe *Geometry*, da API MD3DM, recebendo como parâmetro um *Mesh* e, retornando as informações como o valor do centro, do raio, do canto superior e inferior. A classe *Geometry* também possui algumas funções para determinar a intersecção do volume com um raio, que não foram utilizadas neste trabalho. Na Figura 4.11 é possível observar esses dois volumes.

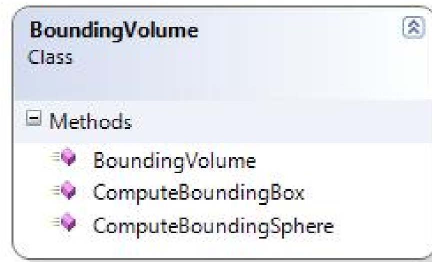


Figura 4.10: Classe *BoundingVolume*

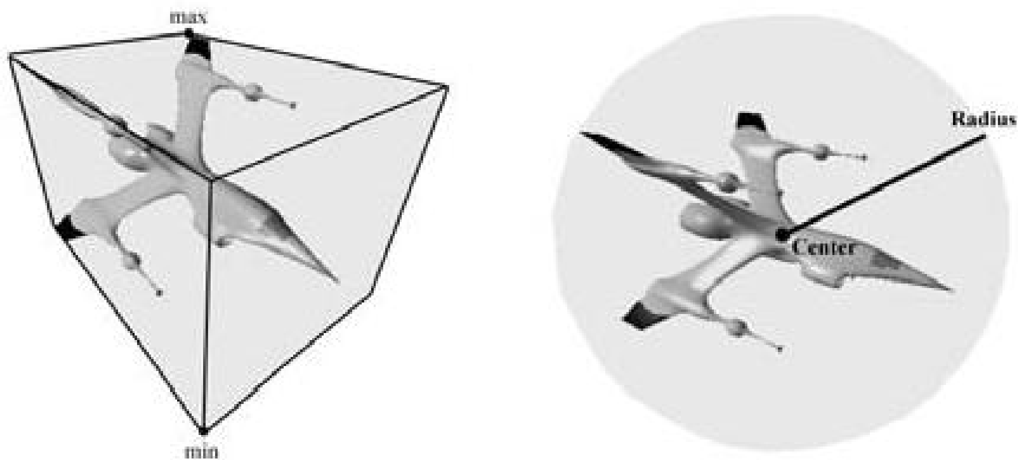


Figura 4.11: O *BoundingBox* à esquerda e o *BoundingSphere* à direita

O cálculo do volume é muito importante para a detecção de colisões pois, sabendo a posição do objeto no espaço e o volume que ele ocupa, é possível determinar se houve a colisão com algum outro objeto ou com um plano.

4.4.4 Transforms

Esta componente contém três classes que realizam transformações: *Rotation*, *Translation*, *Scale*. Todas essas classes possuem métodos que retornam uma matriz de transformação, ou seja, um objeto do tipo *Matrix* com uma transformação aplicada. A principal vantagem de se utilizar esta abordagem é permitir que várias transformações sejam aplicadas ao mesmo tempo, através da multiplicação de matrizes.

A primeira classe, descrita na Figura 4.12, realiza operações de rotação, que podem ser aplicadas em qualquer eixo (x,y,z) individualmente através dos métodos *ApplyInX, ApplyInY, ApplyInZ*, que recebem como parâmetro o ângulo de rotação ou, nos três eixos ao mesmo tempo através do método *ApplyInAll*, que recebe como parâmetro três ângulos de rotação. Também é possível aplicar uma rotação a partir de um *quaternion*, passado como parâmetro.

As outras duas classes, descritas na Figura 4.13, funcionam de forma muito parecida com a anterior, sendo que a única diferença é a transformação que será aplicada. Todas as classes implementam o método *ResetTransform*, que recebe por parâmetro o *Device* e, reinicializa a matriz *World*, passando uma matriz identidade. Isto deve ser feito após o final das transformações aplicadas à um objeto, para permitir que outras transformações sejam aplicadas em outros objetos na cena simultaneamente e, também para que a própria transformação seja atualizada. Se não houver esta reinicialização, a transformação não acontecerá e o objeto ficará "travado".

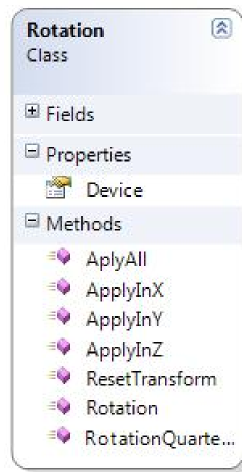


Figura 4.12: Classe *Rotation*

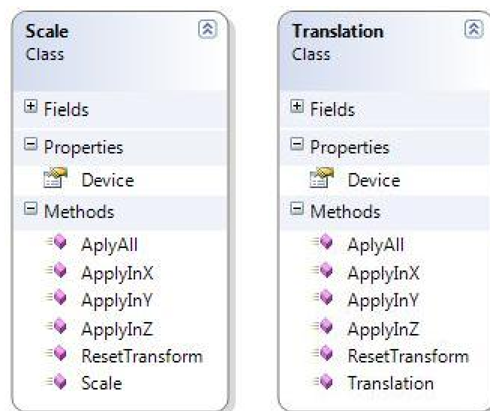


Figura 4.13: Classes *Scale* e *Translate*

4.4.5 *World*

O componente *World*, contém todas as classes utilizadas para a visualização da cena, inicialização do *Device* e, cálculo dos frames por segundo (FPS). Neste componente também é realizada a carga do arquivo de câmeras a partir de um arquivo XML, dentro da classe chamada *CameraLoader*. Este processo é exatamente

igual ao descrito no componente *Ligths*, mudando apenas os valores lidos e os tipos de dados. Todos os valores lidos são passados para um objeto representado da classe *CameraObject*, descrita na Figura 4.14 juntamente com a classe anterior, que contém todos os parâmetros necessários para se construir uma câmera.

Após isso, o objeto é armazenado em uma lista, o que permite a troca dos valores das câmeras. O último valor lido do arquivo, é o que será utilizado para inicializar a câmera, visto que a leitura é feita do início para o fim. A variável *currentCameraIndex* armazena o índice da câmera atual. Um arquivo que exemplifica os valores utilizados como parâmetro pode ser conferido na Figura 4.15.

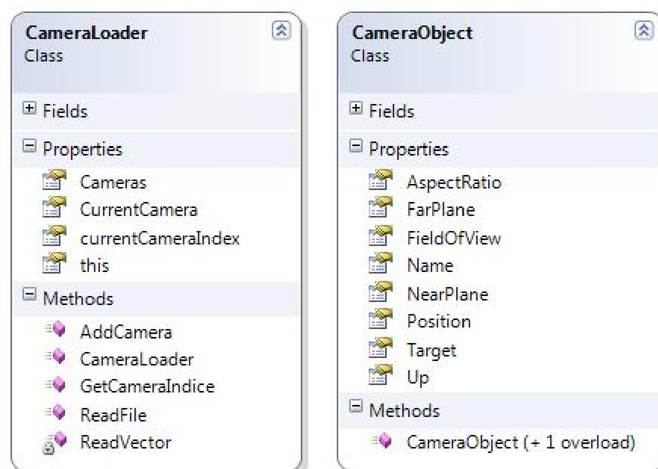


Figura 4.14: Classes *CameraLoader* e *CameraObject*

Para que a câmera seja criada, é necessário ao menos uma entrada no arquivo XML, caso contrário, a câmera não será criada e uma *Exception*[3] será lançada, encerrando a aplicação. Os parâmetros utilizados para a construção de uma câmera são:

- *AspectRatio*: define a proporção da tela. Este valor é calculado dividindo-se a largura da tela pela altura;
- *FieldOfView*: define o campo de visão na direção do eixo y;
- *FarPlane*: define a profundidade máxima do plano de visão. Tudo que será desenhado na cena devera estar dentro deste plano, caso contrário não sera exibido;
- *NearPlane*: define a profundidade mínima do plano de visão, ou seja, onde o plano inicia;
- *Position*: define a posição da câmera no espaço 3D;

[3] *Exception* é um mecanismo utilizado para tratar algumas condições especiais que alteram o fluxo normal do programa.

- *Target*: define a posição para onde a câmera estará apontada;

- Up: define a direção do vetor up, indicando qual dos eixos está voltado para cima. Normalmente utiliza-se y como up, desta forma [0,1,0].

```
<mdge>
  <cameras>
    <add name="cam1" position="-1.0,20.5,2.5" target="0,-16.5,-1.0" up="0,1,0"
         fieldOfView="4.0" nearPlane="1.0" farPlane="1000.0"/>
    <add name="cam2" position="-2.0,0.5,-2.0" target="0,0,-1.0" up="0,1,0"
         fieldOfView="4.0" nearPlane="1.0" farPlane="1000.0"/>
    <add name="cam3" position="-2.0,6.5,16,5" target="0,-0.5,-1.0" up="0,1,0"
         fieldOfView="4.0" nearPlane="1.0" farPlane="1000.0"/>
    <add name="cam4" position="-2.0,0.5,-6.0" target="0,0,-1.0" up="0,1,0"
         fieldOfView="4.0" nearPlane="1.0" farPlane="1000.0"/>
  </cameras>
</mdge>
```

Figura 4.15: Arquivo XML contendo os valores das câmeras

Existem dois tipos de câmeras disponíveis neste trabalho, representadas pelas classes *Camera* e *FligthCamera*, descritas respectivamente nas Figuras 4.16 e 4.20. Ambas necessitam ser inicializadas com a leitura do arquivo, através do método *SetupCamera* que passa os valores armazenados na lista para a própria câmera. A primeira classe representa uma *First Person*(FP) câmera, que é um dos tipos mais utilizados em jogos. O comportamento desta câmera simula o ponto de vista do personagem, com a mesma perspectiva que seria observada a partir dos olhos. Isto oferece uma liberdade total de movimentos em todas as direções. Para tratar este movimento, normalmente utiliza-se uma combinação de *mouse* e teclado, porém como os dispositivos móveis possuem algumas limitações físicas, as funções do *mouse* podem ser substituídas pelo *touchscreen*.

Entretanto, isto traz uma série de dificuldades, pois é necessário mapear as coordenadas 2D da tela para o espaço 3D. Na Figura 4.17 é possível visualizar o relacionamento entre as coordenadas da tela do dispositivo e o espaço 3D. Este mapeamento é feito em três etapas básicas dentro da classe *Camera*:

- Captura da posição inicial do mouse, efetuada dentro do método *MouseDown*, que armazena dentro de uma estrutura chamada *Point*, disponível dentro da própria classe, as coordenadas do primeiro toque e, dentro do método *MouseUp*, que armazena as coordenada do último toque na tela.

- Atualização do movimento, calculada dentro do método *MouseMove* através da diferença entre a posição atual do toque e a posição final. Cada coordenada é computada individualmente e, a diferença é multiplicada por um fator chamado *rotationSpeed*, cujo valor é 0.005, definindo o quão rápido a câmera

vai rotacionar. Estes valores são decrementados nas variáveis *leftrigthRot*, para a coordenada X e *updownRot* para a coordenada Y, conforme o trecho de código exibido na Figura 4.18;

- Após os cálculos anteriores, a câmera é atualizada dentro do método *Update*, onde a matriz de rotação da câmera é criada, utilizando uma rotação no eixo X, com o ângulo *updonRot* multiplicada por uma rotação em Y, com o ângulo *leftrigthRot*. Como na multiplicação de matrizes, a ordem interfere no resultado final, é necessário ter atenção para não se inverter a ordem das transformações. A matriz de rotação resultante será utilizada para transformar os vetores do *Target* e *Up*, criando uma movimentação em 360 graus.

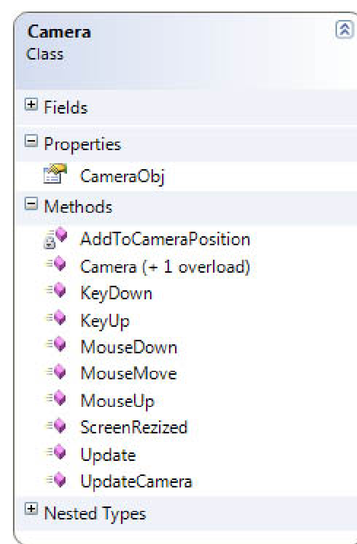


Figura 4.16: Classe *Camera*

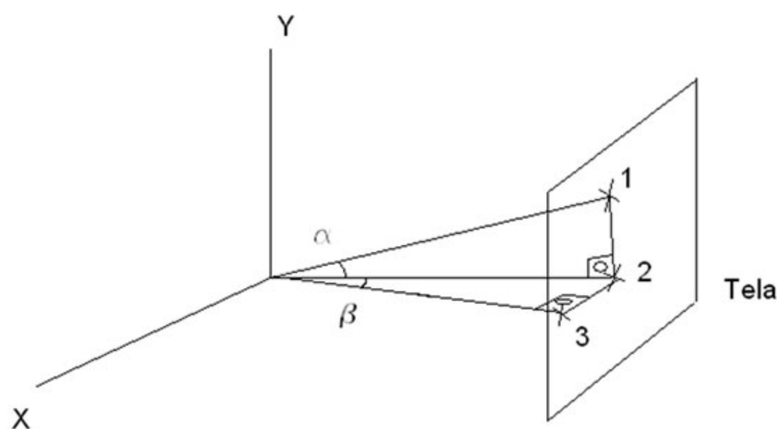


Figura 4.17: Relação entre pontos da tela e espaço 3D

A movimentação pelo cenário ocorre dentro do método *KeyDown*. Quando alguma tecla do dispositivo é pressionada, ela é tratada por essa função. Para mover

a câmera para frente, é preciso adicionar o vetor (0,0,-1) à posição da câmera, para trás (0,0,1), para a esquerda (-1,0,0) e para direita (1,0,0). Também é possível alterar a altura da câmera, incrementando y na posição da câmera da mesma forma que as anteriores adicionando os vetores (0,-1,0) e (0,1,0). Como a câmera pode estar rotacionada em qualquer direção, é necessário aplicar essa transformação ao vetor que será adicionado com a posição. Isto é feito através do método *AddToCameraPosition*, descrito na Figura 4.19. este método deve ser chamado a cada vez que for realizada uma movimentação em qualquer direção, passando aqueles vetores como parâmetro.

```
currentMousePos.X = e.X;
currentMousePos.Y = e.Y;

float xDifference = currentMousePos.X - oldMousePos.X;
float yDifference = currentMousePos.Y - oldMousePos.Y;
leftrightRot -= rotationSpeed * xDifference;
updownRot -= rotationSpeed * yDifference;
```

Figura 4.18: Trecho de código com os cálculos dos ângulos X e Y

```
private void AddToCameraPosition(Vector3 vectorToAdd)
{
    float moveSpeed = 0.5f;
    Matrix cameraRotation = Matrix.RotationX(updownRot) * Matrix.RotationY(leftrightRot);
    Vector3 rotatedVector = Vector3.TransformCoordinate(vectorToAdd, cameraRotation);
    Position += moveSpeed * rotatedVector;

}
```

Figura 4.19: Método *AddToCameraPosition*

O segundo tipo de câmera disponível simula o comportamento de uma nave espacial. Isto significa que ela pode se movimentar em qualquer direção com liberdade de movimentos e, sem a necessidade de utilizar o *touchscreen* para isso. A câmera também pode acompanhar o deslocamento de um objeto pela cena, sendo posicionada ligeiramente atrás, movendo-se a uma velocidade constante. A posição da câmera é calculada baseando-se na posição do objeto e, as transformações que forem aplicadas no objeto, são aplicadas também a câmera, criando a ilusão de que a câmera está voando, logo atrás do objeto. Para que situações como o *Gimbal lock*, descrito anteriormente, aconteçam *quaternions* são utilizados ao invés de matrizes. Como a câmera se move a uma velocidade constante, não há a necessidade de incrementar a posição, da mesma forma que era feito na classe anterior. Para a movimentação, são utilizadas duas variáveis locais, dentro do método *KeyDown* que armazenam um ângulo de rotação, *upDownRot* e *leftRigthRot*. Uma terceira variável chamada *roll* controla a rotação em torno do eixo Z. Estas variáveis são utilizadas

para calcular a rotação do objeto e, consequentemente da câmera, da forma exibida no trecho de código descrito na Figura 4.21.

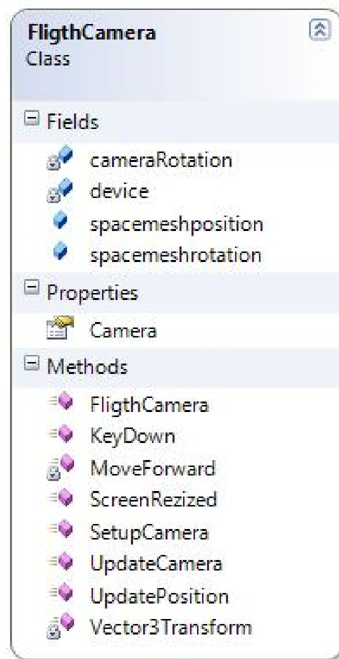


Figura 4.20: Classe *FligthCamera*

```
Quaternion additionalRot = Quaternion.RotationAxis(new Vector3(-1, 0, 0), upDownRot) *
    Quaternion.RotationAxis(new Vector3(0, -1, 0), leftRightRot)
    * Quaternion.RotationAxis(new Vector3(0, 0, 1), roll);

spacemeshrotation = spacemeshrotation * additionalRot;
```

Figura 4.21: Trecho de código demonstrando o cálculo da rotação

A rotação resultante é armazenada na variável global *spacemeshrotation*. Dentro do método *MoveFoward*, essa rotação é utilizada para transformar um vetor que indica a direção do movimento, cujo valor é igual à (0,0,-1) significando um movimento para frente que, posteriormente é multiplicado pela velocidade, no caso 0.2, e adicionado com a posição do objeto. Isto faz com que o objeto se mova para qualquer direção, já que a rotação esta sendo aplicada no vetor que indica a direção do movimento. A posição da câmera é calculada dentro do método *UpdateCamera*, descrito parcialmente na Figura 4.22.

Para criar uma leve diferença entre a rotação do objeto e a da câmera, foi utilizada a função *Quaternion.Slerp* que, gera uma diferença de cerca de 10% entre as duas, tornando o movimento um pouco mais dinâmico. As outras duas classes presentes neste componente se chamam *RenderStateConfig* e *FpsTimer*, descritas na Figura 4.23. A primeira classe é a responsável pela inicialização e configuração do

Device. Esta operação é efetuada dentro do método construtor da classe, seguindo uma sequência muito simples:

- Configuração dos parâmetros de apresentação, através da classe *PresentParameters* da API MD3DM;
- Inicialização do *Device* passando os parâmetros de apresentação e, a tela de controle, que é o *Windows.Form* onde a aplicação está rodando;
- Configuração do *RenderState* ou, modo de renderização da cena;

```
public void UpdateCamera(Device device)
{

    cameraRotation = Quaternion.Slerp(cameraRotation, spacemeshrotation, 0.1f);

    Vector3 cameraOriginalPosition = Camera.Position;
    Vector3 cameraRotatedPosition = Vector3.TransformCoordinate(cameraOriginalPosition,
        Matrix.RotationQuaternion(cameraRotation));
    Vector3 cameraFinalPosition = spacemeshposition + cameraRotatedPosition;

    Vector3 camOriginalUp = Camera.Up;
    Vector3 camFinalUp = Vector3.TransformCoordinate(camOriginalUp,
        Matrix.RotationQuaternion(cameraRotation));
```

Figura 4.22: Método *UpdateCamera*

Ao final desta operação o *Device* é retornado por referência. Dentro desta classe também há um método *KeysDown*, que trata eventos de entrada, permitindo a troca do modo de renderização da cena do modo normal, para o *Wireframe*[4]. Este modo é definido dentro da propriedade *RenderState* da classe *Device* e, por ser relativamente mais simples e rápido de ser calculado do que o normal, acaba aumentando o desempenho da aplicação. Um exemplo do modo *wireframe* pode ser conferido na Figura 4.24.

A classe *FpsTimer* realiza os cálculos dos FPS. Isto é feito em três etapas, representadas pelos métodos *StartFrame* colocado no início do processo de renderização, *Render*, colocado durante o processo de renderização e *StopFrame* colocado no final. O primeiro método inicializa o contador de tempo, o segundo exibe a média dos FPS calculados no terceiro método, na tela. Este cálculo é efetuado com a diferença entre o tempo inicializado em *StartFrame*, com o tempo atual, que é armazenada em uma lista. Como a tela é atualizada várias vezes, é possível contar aproximadamente quanto tempo cada frame levou para ser renderizado, obtendo a média de FPS. Esta média também é armazenada e recalculada constantemente, pois podem haver picos de FPS, dependendo do que estiver sendo renderizado.

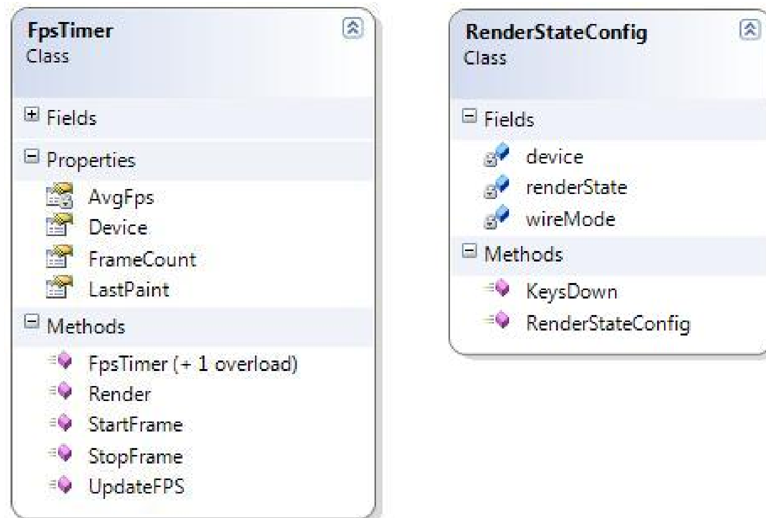


Figura 4.23: Classes *FpsTimer* e *RenderStateConfig*

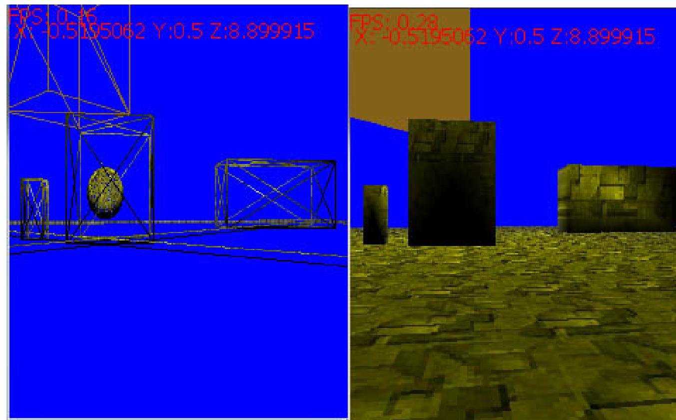


Figura 4.24: Modo *Wireframe* e modo Normal

A utilização de FPS para medir o desempenho de um jogo é um dos meios mais simples e eficientes que existem para descobrir se o dispositivo consegue suportar bem a aplicação. Se os números do FPS forem muito baixos, significa que a aplicação é pesada demais, caso contrário, se o FPS for alto significa que o dispositivo consegue suportar muito bem a aplicação.

4.4.6 *Form*

Todas as aplicações desenvolvidas utilizando o CF necessitam de uma classe que implemente a interface Windows *Form*. Esta classe é a base para a construção da interface de usuário de uma aplicação, contendo todos os métodos para desenho da tela e, gerenciamento de entrada. Estes métodos tem que ser sobrescritos para funcionarem corretamente. Os ambientes de teste que serão descritos posteriormente,

[4] *Wireframe* é um modo de renderização que desenha os objetos como linhas, desenhadas em cada borda

foram implementados utilizando o *Form* como classe principal, mantendo todas as instâncias de outras classes da MDGE. Como todos os eventos de entrada são identificados por esta classe, ela atua como um intermediário, encaminhando o evento ao seu respectivo destino.

Outra função importante exercida por esta classe é a de desenho da tela. Isto é feito através do método *OnPaint*, chamado constantemente durante o funcionamento da aplicação. Para que as imagens sejam desenhadas corretamente, é necessário desativar o método *OnPaintBackground*, caso contrário o plano de fundo sempre irá sobrepor a tela, impedindo a sua atualização. Por algumas limitações do próprio CF, não é possível utilizar menus, botões ou qualquer outro componente em conjunto com a API MD3DM, apenas um *MessageBox*[5] pode ser utilizado para a exibição de mensagens. O Form será a classe principal da MDGE, e cada aplicação que será desenvolvida, utilizará um Form diferente.

4.5 Aplicações de Teste

Paralelamente à implementação das funcionalidades da *engine*, foram desenvolvidas duas aplicações de testes. Seu principal objetivo é verificar o desempenho da MDGE, tanto em um dispositivo físico quanto em um emulador, possibilitando uma análise mais precisa dos resultados.

A MDGE encapsula as funções de criação de câmeras, para a visualização da cena, de carga e renderização de modelos 3D, e da inicialização do dispositivo. Com apenas estas funções básicas, já é possível desenvolver algumas aplicações mais simples, conforme será descrito nas próximas seções.

Para que uma cena seja criada, é necessário antes de tudo, criar uma classe *Form*, que atuará como a classe principal e inicializar o dispositivo, o que é feito através da classe *RenderStateConfig*. Logo em seguida, a câmera, os modelos e a iluminação devem ser carregados, utilizando cada uma das classes descritas anteriormente. As classes que representam as câmeras, presentes na *engine*, possuem um comportamento pré-definido, ou seja, não é necessário implementar a movimentação da mesma. Como a MDGE pode ser alterada, se o desenvolvedor desejar, pode criar um novo tipo de câmera, seguindo o mesmo modelo das duas existentes.

A carga da iluminação e dos modelos é feita a partir de vários arquivos, que devem ser passados respectivamente como parâmetro logo na criação de cada um desses objetos. Essa carga, deve ocorrer apenas uma vez, logo na inicialização do dispositivo, devendo ser refeita caso o mesmo seja reinicializado. Com relação as câmeras, é necessário utilizar a classe *CameraLoader*, que também recebe como parâmetro o nome de um arquivo, para inicializar a câmera principal da aplicação,

[5] *MessageBox*, classe do CF que permite a exibição de caixas de mensagem na tela.

já que esta classe armazena uma lista de câmeras. A troca das câmeras deve ser implementada pelo desenvolvedor, da forma como preferir.

Todos os métodos que tratam eventos de entrada, das classes *Camera* e *FligthCamera* devem ser colocados dentro dos *EventHandlers* da classe principal. Os modelos podem ser renderizados em varias posições diferentes dentro do cenário.

Entretanto, a movimentação desses objetos deve ser implementada pelo próprio desenvolvedor. Novas funcionalidades, também podem ser adicionadas, conforme a necessidade da aplicação. Isto pode ser feito pela inclusão de novos métodos nas classe já existentes, ou então, através da criação de novas classes.

4.5.1 Primeira Aplicação de Teste

A primeira aplicação teste, chamada de Teste1 atua como um visualizador de modelos 3D, englobando todas as funcionalidades implementadas neste trabalho. O modelo que será utilizado como cenário está no formato *Wavefront*. Os outros objetos renderizados na cena são formas geométricas básicas da *engine* e, estão sendo utilizadas para demonstração da iluminação. Isto pode ser observado na Figura 4.25.

A movimentação pelo cenário será feita através da utilização da câmera em primeira pessoa, descrita anteriormente, o que dará uma liberdade total de movimento pelo cenário. Além disso também haverá a possibilidade de trocar os pontos de visualização, ou seja, a troca de câmeras, que serão lidas do arquivo XML chamado Camera1.xml. Nas Figuras 4.26, 4.27 e 4.28 é possível visualizar algumas das câmeras utilizadas nesta aplicação. Todas as luzes utilizadas na cena são carregadas do arquivo Ligths.xml

Para que todos os arquivos sejam lidos, eles têm que estar dentro da pasta *Content* da MDGE. Como os arquivos são compilados e enviados para o dispositivo juntamente com o executável, é necessário indicar na propriedade de cada arquivo que eles serão passados como *Embedded Resource*, caso contrário, não será possível carregá-los na memória.

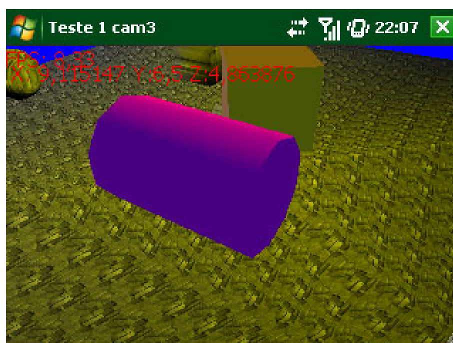


Figura 4.25: Exemplo de um objeto renderizado na cena

Para facilitar a visualização das informações referentes ao desempenho, os FPS

são exibidos na parte de cima da tela, juntamente com as coordenadas da posição da câmera. A configuração das teclas nesta aplicação se encontra na Tabela 4.2, localizada no final deste capítulo.

Figura 4.26: Câmera Inicial

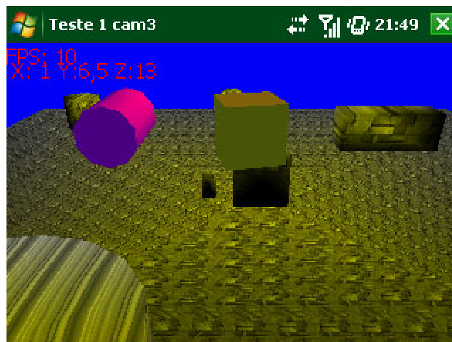


Figura 4.27: Câmera panorâmica

Figura 4.28: Câmera superior

4.5.2 Segunda Aplicação de Teste

A segunda aplicação, chamada Teste2 é a mais complexa, pois simula voo de uma nave espacial. Esta aplicação pode ser considerada o primeiro jogo desenvolvido utilizando a MDGE, embora seja bastante simples se comparada a um jogo completo. Seu principal objetivo é comprovar que realmente é possível desenvolver um jogo

simples utilizando as funcionalidades implementadas neste trabalho obtendo um desempenho satisfatório. A câmera utilizada nesta aplicação é a *FligthCamera* e, embora ela seja inicializada a partir de um arquivo XML, como a anterior, não será possível trocar a posição da câmera, limitando a visualização à apenas uma posição.

Este jogo consiste em uma nave voando por um cenário, com alguns alvos que devem ser destruídos, distribuídos aleatoriamente pelo espaço. A nave pode disparar projéteis para isso, porém se ela colidir com um alvo ou com o terreno, o jogo será reiniciado com a nave retornando a posição original. O modelo do terreno utilizado nesta aplicação está no formato *.md3dm* e, o da nave espacial no formato *Wavefront*. Os alvos e os projéteis foram renderizados como esferas, nas cores vermelha e amarela respectivamente. Nas Figuras 4.29 e 4.30 é possível observar algumas telas do jogo.

Apesar de a detecção de colisão não fazer parte dos requisitos principais contemplados pela MDGE, foi implementado, apenas nesta aplicação, um método simples para realizar esta tarefa chamado de *CheckCollision*. Este método recebe como parâmetro a posição de um objeto, o diâmetro e o fator de escala e, retorna um número inteiro, indicando o tipo de colisão. Dentro dele, três coisas são calculadas:

- O raio do *BoundingSphere*, que é o diâmetro dividido por 2. Se o objeto estiver escalonado, é necessário multiplicar o fator de escala pelo raio, para obter o tamanho correto;

- Se o objeto ultrapassou os limites do cenário, ou seja, se o objeto chegou próximo demais do terreno ou subiu alto demais, ou ainda, se afastou demais do cenário. Neste caso é retornando 1,2 ou 3.

- Se o objeto colidiu com um alvo. Neste caso é retornado o valor 4 e, o alvo é removido da tela e da memória. Se o objeto testado for um projétil, ele também será removido, caso seja a nave ela retornará a posição inicial.

Figura 4.29: Tela da aplicação de teste 2

Os testes de colisão são aplicados apenas na nave e nos projéteis. O *Bounding-Sphere* da nave, que contém o diâmetro, é calculado no momento em que a nave é carregada utilizando a classe *BoundingVolume*, descrita anteriormente. Como os projéteis e os alvos são esferas, o diâmetro já é conhecido e por isso não é calculado. Este teste é utilizado apenas nesta aplicação, por isso não será descrito mais detalhadamente.

Figura 4.30: Tela da aplicação de teste 2

Algumas limitações físicas do dispositivo como os controles de entrada, que não permitem que duas teclas sejam pressionadas ao mesmo tempo e, do próprio sistema operacional, que trava caso uma quantidade muito grande de memória seja usada prejudicaram o desempenho desta aplicação. A configuração das teclas para esta aplicação pode ser conferida na Tabela 4.3.

Tecla	**Função**
Seta para cima	Movimenta a câmera para a frente.
W	Movimenta a câmera para a frente.
Seta para baixo	Movimenta a câmera para a trás.
S	Movimenta a câmera para a trás.
Seta para esquerda	Movimenta a câmera para a esquerda.
A	Movimenta a câmera para a esquerda.
Seta para direita	Movimenta a câmera para a direita.
D	Movimenta a câmera para a direita.
SoftKey esquerda	Movimenta a câmera para cima.
SoftKey direita	Movimenta a câmera para baixo.
V	Modifica a câmera.
M	Habilita/desabilita o modo *wireframe*.
Touchscreen	Movimenta o alvo da câmera em qualquer direção.

Tabela 4.2: Configuração das teclas na aplicação de teste 1

Foram realizados alguns testes de carga em ambas aplicações, onde uma grande

quantidade de objetos 3D foi renderizada na tela, para descobrir qual seria a quantidade máxima de objetos suportada em cada aplicação e, o quanto isso afetaria o desempenho. Os resultados dos testes realizados, serão apresentados no próximo capítulo.

Tecla	**Função**
Seta para cima	Guinada da nave para baixo.
W	Guinada da nave para baixo.
Seta para baixo	Guinada da nave para cima.
S	Guinada da nave para cima.
Seta para esquerda	Guinada da nave para a esquerda.
A	Guinada da nave para a esquerda.
Seta para direita	Guinada da nave para a direita.
D	Guinada da nave para a direita.
K	Rotação da nave para esquerda.
L	Rotação da nave para a direita.
Enter	Dispara um projétil.
M	Habilita/desabilita o modo *wireframe*.

Tabela 4.3: Configuração das teclas na aplicação de teste 2

5 RESULTADOS

Todos os testes realizados neste trabalho foram feitos tanto em um dispositivo real, quanto em um emulador. Isso permitiu a criação de parâmetros de comparação reais, haja visto que não existem muitas aplicações gráficas, além dos exemplos do próprio CF, que utilizem a mesma tecnologia que este trabalho. Em relação ao desempenho, os resultados obtidos foram satisfatórios, embora os testes tenham sido feitos em apenas um modelo de dispositivo físico. A utilização de dispositivos diferentes proporcionaria dados mais concretos.

Durante os testes, foi detectada uma enorme diferença entre o dispositivo e o emulador do *Visual Studio*. Enquanto as aplicações de teste rodavam no aparelho à uma média de 10-12 FPS, o emulador não conseguia mais do que 0,5 FPS. Normalmente, os emuladores têm um desempenho inferior ao de um dispositivo, pois toda a emulação é feita por software porém, essa diferença de desempenho foi grande demais, dada a as configurações do computador em que os testes foram realizados.

Isto mostra que, em termos de desenvolvimento e desempenho, os emuladores ainda estão em um estágio muito primitivo e que ainda não é possível utiliza-los para testes. Por este motivo todos os resultados que foram exibidos neste trabalho são referentes apenas ao dispositivo físico.

A velocidade de carga dos modelos 3D na primeira aplicação de teste foi de 13 segundos em média, enquanto na segunda o tempo médio foi de 11 segundos. O tamanho do arquivo que foi carregado na primeira aplicação era de 52 KB, enquanto na segunda aplicação, o modelo da nave tinha 37 KB e o terreno 52 KB. Todas as texturas utilizadas nessas aplicações tinham cerca de 4 KB e, os demais objetos desenhados em cena, eram formas geométricas básicas que, eram desenhadas automaticamente, sem a necessidade da leitura de um arquivo.

Com relação à velocidade da aplicação, as médias registradas foram de 10 FPS na primeira aplicação, com picos mínimos de 8 FPS e máximos de 20 FPS, e de 12 FPS na segunda, com picos mínimos de 7 FPS e máximos de 16 FPS. Estes picos acontecem por causa do *culling*, explicado anteriormente, que ocorre quando há poucos objetos dentro do campo de visão da câmera. Quando esses objetos não

estão focalizados, eles não são renderizados aumentando o desempenho da aplicação. Entretanto, quando a câmera focaliza muitos objetos, muitas coisas tem que ser renderizadas na tela, diminuindo o desempenho da aplicação.

Além dos testes normais, também foram executados alguns testes para verificar a quantidade máxima de objetos, além do cenário, que cada aplicação seria capaz de suportar com um desempenho satisfatório. Na primeira aplicação, além dos dois objetos originalmente na cena, foi possível adicionar mais quinze esferas 3D, gerando uma queda na média de *frames* para 7 FPS, o que pode ser considerado um bom resultado. Acima desse valor houve uma queda brusca nos FPS impossibilitando um bom desempenho.

Na segunda aplicação, além do terreno, da nave, dos vinte alvos originais e dos projéteis, foi possível adicionar mais 10 alvos, ocasionando uma queda da média de *frames* para 8 FPS. Acima disso, o desempenho também cai bruscamente. Os tempos de carregamento aumentaram em média cerca de meio segundo em cada aplicação.

Com base nos resultados obtidos nos testes pode-se concluir que a MDGE consegue satisfazer as necessidades para o desenvolvimento de aplicações 3D básicas em dispositivos móveis. Entretanto, para o desenvolvimento de aplicações mais complexas, será necessário utilizar um dispositivo com maior poder de processamento e, talvez uma API gráfica mais poderosa do que o MD3DM.

6 CONSIDERAÇÕES FINAIS

O presente trabalho apresentou a implementação de um protótipo de uma *engine* 3D para dispositivos móveis compatíveis com o sistema operacional Windows Mobile e com o .NET CF 3.5. Além disso, também foram desenvolvidas duas aplicações, com o objetivo de testar o desempenho das funcionalidades implementadas neste trabalho, tanto em um dispositivo físico como em um emulador.

O desenvolvimento de aplicações para dispositivos móveis utilizando o CF é feito de uma forma muito similar ao desenvolvimento de aplicações normais para computador. É possível utilizar praticamente as mesmas funções que nos computadores, porém as limitações de hardware do aparelho, exigem que novas técnicas e metodologias sejam aplicadas, pois qualquer coisa a mais que seja acrescentada pode influenciar drasticamente o desempenho.

Por este motivo, algumas práticas de comuns de programação, como a utilização de *design patterns*, devem ser minimizadas, visto que elas podem acrescentar uma complexidade maior a aplicação, exigindo mais memória e poder de processamento do dispositivo. Com a evolução desses dispositivos, um dia será possível desenvolver aplicações multiplataforma, que funcionem da mesma forma tanto num computador quanto em um dispositivo móvel, diferenças mínimas de desempenho.

Durante o desenvolvimento do projeto a maior dificuldade encontrada foi a falta de uma documentação mais completa sobre a API MD3DM. Apesar de existirem muitos materiais sobre o Direct3D para *desktop* e sobre XNA, que acabaram servindo como auxílio, nenhum deles é específico para dispositivos móveis. Além disso, nenhuma das ferramentas de modelagem 3D utilizadas criou modelos leves o suficiente para serem utilizados em ambientes móveis. Foi preciso reduzir manualmente o número de polígonos e a qualidade das texturas, para que sua utilização fosse possível.

Do ponto de vista tecnológico, com a constante evolução que ocorre com os dispositivos móveis, fica evidente que em breve, estes aparelhos irão suportar gráficos com uma qualidade quase equivalente ao dos computadores. Essa mesma evolução também acontecerá com as APIs gráficas, proporcionando mais recursos para o

desenvolvimento de aplicações gráficas 3D e consequentemente, melhorando a qualidade dessas aplicações. Atualmente, embora existam dispositivos como o iPhone, que é capaz de suportar gráficos com uma qualidade muito boa, ainda há um caminho muito longo a ser percorrido, para que um dia os dispositivos móveis consigam atingir o mesmo nível dos consoles.

O resultado final deste trabalho é a base para criação de uma *engine* 3D completa para dispositivos móveis. Foi desenvolvida uma estrutura básica e funcionalidades suficientes para desenvolver algumas aplicações gráficas mais simples. Além disso, pela forma com a MDGE foi construída, dividida em componentes, novas funcionalidades podem ser facilmente incorporadas, gerando outros trabalhos que podem ser desenvolvidos com base neste como: a utilização de métodos mais eficientes para melhorar o desempenho e a implementação de outros módulos como o de inteligência artificial ou de física, visando complementar a *engine*.

6.1 Trabalhos futuros

Durante o desenvolvimento deste trabalho surgiram diversas sugestões para trabalhos futuros. Ainda existem muitas coisas que podem implementadas, com o objetivo de tornar a MDGE uma *engine* completa, tomando este trabalho como base. As sugestões são as seguintes:

- Toda *engine* 3D não deve ficar restrita apenas a dois formatos de arquivos. Portanto, uma possível extensão seria a criação de novas classe para renderizar e carregar arquivos em diversos formatos ou, até mesmo criar um novo formato otimizado. Também seria possível efetuar a conversão dos arquivos nos formatos suportados pela *engine*;

- Desenvolver um módulo de inteligência artificial para adicionar lógica aos jogos;

- Utilizar outras tecnologias disponíveis nos dispositivos móveis como o acelerômetro, para controlar algumas funções como a movimentação da câmera ou a de um objeto ;

- Criar um editor de cenários simplificado, para facilitar o desenvolvimento de futuros jogos;

- Desenvolver um sistema *multiplayer*, ou seja, que permita que dois ou mais dispositivos possam jogar entre si, conectados através de uma rede. Como existem vários tipos de tecnologias para transmissão de dados sem fio, poderia ser feito um estudo sobre qual seria a mais adequada para este tipo de aplicação.

REFERÊNCIAS

ALMQVIST, P. **Keep it Simple, Stupid!**, 2001. Disponível em: http://www.digital-web.com/articles/keep_it_simple_stupid/. Acesso em: 26 out. 2009.

APPLE. **iPhone 3G**, 2009. Disponível em: http://www.apple.com/br/iPhone/. Acesso em: 24 maio 2009.

AUTODESK. **3D Studio Max**, 2009. Disponível em: http://usa.autodesk.com/adsk/servlet/pc/index?id=13567410siteID=123112. Acesso em: 29 out. 2009.

BARNES, D. **Fundamentals of Microsoft .NET Compact Framework Development for the Microsoft .NET Framework Developer**, 2003. Disponível em: http://msdn.microsoft.com/enus/library/aa446549.aspxnet_vs_netcf_topic14. Acesso em: 14 set. 2009.

BOURKE, P. **Wavefront .obj file format**, 1996. Disponível em: http://local.wasp.uwa.edu.au/ pbourke/dataformats/obj/. Acesso em: 12 out. 2009.

BOURKE, P. **Direct-X File Format**, 1999. Disponível em: http://local.wasp.uwa.edu.au/ pbourke/dataformats/directx/. acesso em 12 out. 2009.

DUNN, F.; PARBERRY, I. **3D Math Primer for Graphics and Game Development**. [S.l.]: Wordware Publishing, Inc., 2002.

EASTMAN, P. **Art Of Ilusion**, 2009. Disponível em: http://www.artofillusion.org. Acesso em: 29 out. 2009.

EBERLY, D. H. **3D Game Engine Design** : a Practical Approach to Real-Time Computer Graphics. São Francisco: Morgan Kaufmann, 2001.

ELEMENTS INTERACTIVE. **EDGELIB**, 2009. Disponível em: http://www.edgelib.com/. Acesso em: 12 out. 2009.

FOWLER, M. **O Design Está Morto?**, 2004. Disponível em: http://www.infoq.com/br/articles/is-design-dead. acesso em 29 out. 2009.

KHRONOS. **OpenGL ES Overview**, 2004. Disponível em: http://www.khronos.org/opengles/. Acesso em: 29 set. 2009.

LATEX PROJECT TEAM. **LATEX - A document preparation system**, 2009. Disponível em: http://www.latex-project.org/. Acesso em: 29 out. 2009.

MICROSOFT. **Windows CE**, 1996. Disponível em: http://msdn.microsoft.com/en-ph/windowsembedded/default(en-us).aspx. Acesso em: 06 ago. 2009.

MICROSOFT. **Windows Mobile Directx and Direct3d**, 2005. Disponível em: http://msdn.microsoft.com/en-us/library/ms1725074.aspx. Acesso em: 24 maio 2009.

MICROSOFT. **Game API (GAPI)**, 2005. Disponível em: http://msdn.microsoft.com/en-us/library/ms831615.aspx. Acesso em: 03 set. 2009.

MICROSOFT. **Step by Step Managed Casual Game Development**, 2007. Disponível em: http://msdn.microsoft.com/en-us/bb278108.aspx. Acesso em 12 out. 2009.

MICROSOFT. **.NET Compact Framework 3.5**, 2008. Disponível em: http://msdn.microsoft.com/en-us/netframework/aa497273.aspx. Acesso em: 24 maio 2009.

MICROSOFT. **Direct3d Mobile Rendering Pieline**, 2008. Disponível em: http://msdn.microsofft.com/en-us/library/aa916433.aspx. Acesso em: 13 out. 2009.

MICROSOFT. **Windows Mobile 6.1**, 2009. Disponível em: http://www.microsoft.com/windowsmobile/6-1/default.mspx. Acesso em: 24 maio 2009.

MICROSOFT. **Xna**, 2009. Disponível em: http://www.xna.com. Acesso em: 24 maio 2009.

MICROSOFT. **Windows Graphics Device Interface (GDI)**, 2009. Disponível em: http://msdn.microsoft.com/en-us/library/dd145203(VS.85).aspx. Acesso em: 03 set. 2009.

MICROSOFT. **Visual Studio**, 2009. Disponível em: http://msdn.microsoft.com/pt-br/vstudio/default.aspx. Acesso em: 14 set. 2009.

MICROSOFT. **Office Visio 2007**, 2009. Disponível em: http://office.microsoft.com/pt-br/visio/FX100487861046.aspx. Acesso em: 20 nov. 2009.

MILLER, T. **Managed Directx 9**: Graphics and Game Programming. [S.l.]: Sams Publishing, 2003.

MINDPOOL CORPORATION. **Galatea Engine Technology**, 2009. Disponível em: http://www.mindpol.com/Technology/GT/. Acesso em: 12 out. 2009.

NINTENDO. **Nintendo DS**, 2009. Disponível em: http://www.nintendo.com/ds. Acesso em: 24 maio 2009.

NOKIA. **Nokia Communicator**, 1997. Disponível em: http://europe.nokia.com/get-support-and-software/product-support/nokia-9110. Acesso em: 6 ago. 2009.

NOKIA. **Snake**, 1997. Disponível em: http://www.nokia.com/about-nokia/company/story-of-nokia/mobile-revolution/snake-game. Acesso em: 22 ago. 2009.

NOKIA. **N-Gage**, 2003. Disponível em: http://www.n-gage.com/ngi/ngage/web/br/en/home.html. Acesso em: 7 ago. 2009.

NOKIA. **Mobile 3d Graphics API**, 2004. Disponível em: http://www.jcp.org/en/jsr/detail?id=297. Acesso em: 22 ago. 2009.

PALM. **Palm Pilot 1000**, 1996. Disponível em: http://www.palminfocenter.com/news/8493/pilot-1000-retrospective. Acesso em: 6 ago. 2009.

PALM. **Palm OS 1.0**, 1996. Disponível em: http://en.wikipedia.org/wiki/Palm_OS. Acesso em: 6 ago. 2009.

PAMPLONA, V. F. **Um protótipo de motor de jogos 3D para dispositivos móveis com suporte a especificação mobile 3D graphics API for J2ME.** 2005. Monografia (Trabalho de Conclusão em Ciência da Computação) — Centro de Informática, Universidade Regional de Blumenau, Blumenau.

PESSOA, C. A. C. **wGEM** : um Framework de Desenvolvimento de Jogos para Dispositivos Móveis. 2002. Dissertação (Mestrado em Ciência da Computação) — Centro de Informática, Universidade Federal de Pernambuco, Recife.

QUALCOMM. **BREW (Binary Runtime Environment for Wireless)**, 2009. Disponível em: http://brew.qualcomm.com/brew/en. Acesso em: 30 ago. 2009.

RAMEY, D.; ROSE, L.; TYERMAN, L. **MTL material format (Lightwave, OBJ)**, 1995. Disponível em: http://local.wasp.uwa.edu.au/ pbourke/dataformats/mtl/. Acesso em: 12 out. 2009.

REVOLUTION STUDIOS. **Doom**, 2007. Disponível em: http://revolution.cx/DoomCE.htm. Acesso em: 14 set. 2009.

SONY. **Playstation Portable**, 2009. Disponível em: www.us.playstation.com/PSP. Acesso em: 24 maio 2009.

SUN. **Java 2 Plataform, Micro Edition**, 2009. Disponível em: http://java.sun.com/javame/technology. Acesso em: 22 ago. 2009.

TEXNICCENTER. **TeXnicCenter**, 2008. Disponível em: http://www.texniccenter.org/. Acesso em: 20 nov. 2009.

UNITY TECHNOLOGIES. **Unity 3D**, 2009. Disponível em: http://unity3d.com/unity/. Acesso em: 12 out. 2009.

WIGLEY, A.; MOTH, D.; FOOT, P. **Microsoft Mobile Development Handbook**. [S.l.]: Microsoft Press, 2007.