Universidade Federal de Pernambuco
Graduação em Ciência da Computação
Centro de Informática

Trabalho de Graduação

mOGE – mObile Graphics Engine

O projeto de um motor gráfico 3D para a criação de jogos em dispositivos móveis

Ives José de Albuquerque Macêdo Júnior

Orientador: Sílvio de Barros Melo
Co-orientador: Geber Lisboa Ramalho

Recife
Março de 2005

"The Road goes ever on and on
Down from the door where it began.
Now far ahead the Road has gone,
And I must follow, if I can,
Pursuing it with eager feet,
Until it joins some larger way
Where many paths and errands meet.
And whither then? I cannot say."

—JOHN RONALD REUEL TOLKIEN (1892-1973)

A meus pais e irmãs

AGRADECIMENTOS

"Gratitude is not only the greatest of virtues,
but the parent of all the others."

—WINSTON CHURCHILL

Aos meus pais e irmãs, por todo o carinho e suporte que me foram dados durante toda a minha vida.

Ao meu mestre e orientador *Sílvio Melo*, no qual encontrei um excepcional professor e um grande amigo desde o primeiro período de minha graduação.

Ao meu amigo e co-orientador *Geber Ramalho*, por todo o apoio me dado e pela confiança que depositou em mim tanto no desenvolvimento deste trabalho quanto durante minha iniciação científica.

A Paulo Borba, meu amigo e primeiro orientador na graduação, por despertar meu interesse em pesquisa e desenvolvimento.

Aos demais professores do CIn que, embora não tenham sido meus orientadores oficialmente, em muito influenciaram minha formação acadêmica. Dentre os quais, *Sérgio Cavalcante*, *Carlos Ferraz*, *André Santos* e *Aluízio Araújo*, cronologicamente, se destacam pelo impacto que tiveram na minha vida acadêmica e pela admiração e amizade que tenho por eles.

Ao meu amigo *Marco Tulio*, que em muito me auxiliou no desenvolvimento deste trabalho e me influenciou o gosto pela computação gráfica aplicada aos jogos de computador.

Aos meus amigos *Aércio*, *Afonso*, *Leonardo (Gênio)*, *Vicente* e *Vilmar* por terem sido tão bons amigos e companheiros nessa graduação e, especialmente, nessas "viradas de noite" no CIn que antecederam a entrega deste relatório[1].

A todos os meus outros amigos e colegas que, devido à quantidade, não pude citar nesta pequena página.

Ao professor *Luiz Velho* do *Laboratório VisGraf*, por me ceder, tão gentilmente, a infra-estrutura necessária para o desenvolvimento de parte deste

[1] *Aércio* e *Vilmar* merecem um certo destaque. Se não fossem as caronas deles, eu teria assistido a bem menos aulas na minha graduação e estaria ainda mais preocupado com o tempo disponível para o desenvolvimento deste trabalho pois, tragicamente, o *Rio Doce / CDU* não passa à 1h da madrugada...

trabalho durante a minha estada no *Instituto Nacional de Matemática Pura e Aplicada* nesses primeiros meses do ano.

E, finalmente, ao *Centro de Informática* (CIn) da *Universidade Federal de Pernambuco*, pelo fabuloso ambiente de trabalho, no qual passei quase tanto tempo quanto em minha própria casa, fornecido durante toda a minha graduação.

RESUMO

Desenvolver um jogo é uma tarefa árdua e não-trivial, pois sua criação envolve um conhecimento profundo de diversas áreas da computação. As dificuldades se acentuam ainda mais em ambientes de processamento e memória restritos (como *celulares* e *PDA*'s). Por isso, a utilização de *frameworks* e *padrões de projeto*, encapsulando essa complexidade, mostra-se crucial para a sobrevivência de qualquer projeto profissional.

Esse trabalho tem por objetivo prover um *survey* de técnicas avançadas da *Computação Gráfica 3D* cuja aplicabilidade na criação de jogos para dispositivos móveis tem viabilidade potencial[2]. Além disso, projetamos um protótipo de um *motor gráfico 3D* (conhecido como *mOGE*) voltado ao desenvolvimento de jogos para esses dispositivos, componente crítico de qualquer *framework* 3D para jogos. Esta é uma iniciativa pioneira no *Centro de Informática*, posto que apenas frameworks 2D foram criados. Dessa forma, visamos atuar como um ponto de partida para a criação de um *framework* completo e introduzir uma cultura de jogos 3D nessa instituição.

Palavras-chave: Computação Gráfica 3D, Entretenimento Digital, Engenharia de Software, Jogos Portáteis, Dispositivos Móveis.

[2]Isso porque nem todas essas técnicas já foram utilizadas em jogos para celulares.

ABSTRACT

Game development is an arduous and non-trivial task, because its creation involves deep knowledge of diverse areas of computer science. The difficulties are even harder in environments with restrict processing and memory (such as *mobile phones* and *PDA*'s). Therefore, the use of *frameworks* and *design patterns*, encapsulating all complexity, is crucial for the survival of any professional project.

This work's main goal is providing a *survey* on *Computer Graphics 3D* advanced techniques, whose applicability on game development for mobile devices has potential viability. Besides, we projected a *3D Graphics Engine* archetype (known as *mOGE*), targeted to mobile device game development, crucial when developing a game. This is a pioneer effort in *Centro de Informática* (UFPE – Pernambuco – Brazil), until today only 2D *frameworks* have been studied and developed. This way, our intent is to begin the creation of a complete *framework* and introduce a culture of 3D games development in this institution.

Keywords: 3D Computer Graphics, Digital Entertainment, Software Engineering, Portable Games, Mobile Devices.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

CAPÍTULO 1

INTRODUÇÃO

"We do not stop playing because we grow old.
We grow old because we stop playing."
—AUTOR DESCONHECIDO

Poucos setores da economia mundial têm crescido tanto quanto o mercado de *entretenimento digital*. A *DFC Intelligence*[48] (um grupo especializado em análises do mercado mundial de jogos eletrônicos) estima que esse setor crescerá dos recentes US$ 23.2 bilhões para US$ 31.6 bilhões em 2009. Ainda levando em consideração dados da *DFC Intelligence*, com o lançamento do *PlayStation Portable*[78], da *Sony*, e do *GameBoy DS*, da *Nintendo*, (competidores do modelo *N-Gage*[74], da *Nokia*), o crescimento do setor de jogos para dispositivos móveis se dará dos US$ 3.9 bilhões faturados em 2003 para US$ 11.1 bilhões em 2007.

Tais números têm atraído uma série de gigantes da computação, dentre elas a *Sony*[46], a *Nintendo*[80] e a *Nokia*[81] que vêm travando uma guerra em um nicho bastante promissor: o dos *consoles de jogos portáteis*. Além dessa disputa esse crescimento do mercado tem incentivado o aparecimento de diversas empresas especializadas tanto na produção de jogos portáteis quanto no desenvolvimento de ferramentas que auxiliem nessa criação.

Essas últimas empresas realizam tarefas que vão desde implementações de especificações de bibliotecas (e.g. o *Gerbera*[38] da *Hybrid*[39], implementação em software da especificação *OpenGL ES*[30] do grupo *Khronos*[41]) até *frameworks* completos para a criação de jogos 3D (e.g. os motores *Swerve Client*[16], da *Superscape*[40], *Mascot Capsule*[13], da *Hi*[17], *Xforge 2*[101], da *Fathammer*[60] e *Mophun*[72], da *Synergenix*[49]).

Existem várias plataformas voltadas ao desenvolvimento de aplicações em dispositivos móveis. Abaixo, segue uma breve descrição das mais proeminentes:

- *J2ME – Java 2 Platform, Micro Edition*[50]: desenvolvida pela *Sun Microsystems*[68], possui aceitação muito boa por parte dos desenvolvedores (de jogos 2D) pois *Java* é uma linguagem de fácil programação e possui uma portabilidade satisfatória[73], embora alguns desenvolvedores reclamem da performance e das limitações de sua *API*.

- *Qualcomm BREW – Binary Runtime Environment for Wireless*[11]: desenvolvido pela *Qualcomm*[47], tem grande aceitação e abrangência no território norte-americano, é uma plataforma consolidada que permite a execução de aplicações nativas desenvolvidas em C/C++. Sua maior desvantagem é a dificuldade de se desenvolver para ela, por ser proprietária e possuir um modelo de negócios rígido (mas muito bem visto pelas operadoras).

- *Symbian OS*[83]: *Symbian* é um consórcio de empresas de telefonia para o desenvolvimento de um sistema operacional para celulares. O *Symbian OS* possui o conceito de *"Open Standard Operating System"* o que permite, pela primeira vez, que se desenvolvam aplicações para celulares da forma que são desenvolvidas para PC's.

- *Windows Mobile*[71]: desenvolvido pela *Microsoft Corporation*, ainda tem adoção muito baixa no mercado de celulares, embora traga a vantagem de um modelo de programação bastante semelhante ao existente para PC's, ao qual a maior parte dos desenvolvedores é habituada.

Assim como é grande a diversidade de plataformas disponíveis para desenvolver aplicações para dispositivos móveis, são várias as tecnologias voltadas à programação de gráficos nesses dispositivos (essenciais no desenvolvimento de jogos 3D). Abaixo, segue uma breve descrição das tecnologias mais promissoras:

- *OpenGL ES*[30]: é uma *API* multi-plataforma de baixo nível para criação de sistemas gráficos, tanto 2D quanto 3D, para sistemas embarcados. É na realidade um sub-conjunto bem-definido da já bem conhecida e muito utilizada *OpenGL*. *OpenGL ES* é dividida em três partes. Dessas, duas são especificações que endereçam diferenças entre configurações de dispositivos (poder de processamento, capacidade de memória etc.): a *Common*, que utiliza a aritmética de ponto-flutuante, e a *Common Lite*, a qual utiliza um sistema de ponto fixo. Já a terceira, a *EGL*, é uma especificação que busca abstrair o sistema de janelas nativo do dispositivo. Atualmente a *OpenGL ES* é considerada a mais promissora plataforma para o desenvolvimento de jogos 3D para dispositivos móveis.

- *Mobile 3D Graphics / Java Specification Request #184*[32]: é uma proposta de especificação *Java* de alto nível incluída como pacote opcional no *J2ME* e *MIDP*. Tem o objetivo de ser simples o bastante para viabilizar uma criação eficiente de sistemas gráficos 3D. Fundamenta-se no

Java3D da *J2SE* que por sua vez segue um paradigma de programação baseado em grafos de cenas, um conceito bem comum em computação gráfica 3D. Normalmente é implementado em software utilizando o suporte a gráficos nativo do dispositivo, às vezes utilizando *OpenGL ES* para realizar as operações gráficas de baixo nível.

- *Direct3D Mobile*[70]: é a opção da *Microsoft* para a criação de gráficos tridimensionais. É um sub-conjunto da *API* do *Direct3D* para *Desktop*, sendo suportado por plataformas baseadas em *Windows CE / Windows Mobile*. Tem uma arquitetura que permite a implementação de sua *API* tanto em *hardware* quanto em *software*, ou uma abordagem híbrida. Foi projetada para vários tipos de arquiteturas implementando tanto aritmética de ponto-flutuante quanto de ponto-fixo. Vale salientar que o ponto-flutuante pode ser emulado dependendo da plataforma tornando-se, conseqüentemente, menos eficiente.

Devido a essa grande variedade de plataformas e tecnologias, fez-se necessário um cuidadoso estudo comparativo que avaliou a viabilidade de cada uma delas para a criação de jogos 3D para dispositivos móveis e de *frameworks* eficientes e eficazes voltados para seu desenvolvimento.[1] Essa última aplicação dessas tecnologias é a principal proposta deste trabalho: *um motor gráfico 3D para a criação de jogos 3D para dispositivos móveis*, o *mOGE*.[2]

A seguir, são apresentadas as principais motivações que resultaram no desenvolvimento deste trabalho, os objetivos mais importantes a serem atingidos, os principais desafios em sua realização e uma visão geral da estrutura desta monografia.

1.1 MOTIVAÇÃO

Poucas são as áreas da computação que têm a multidisciplinaridade encontrada na área de jogos eletrônicos. Desenvolver um jogo é uma tarefa árdua e não-trivial, pois *jogos são complexos sistemas de tempo-real multimídia e interativos*. Sua criação envolve um conhecimento profundo de diversas áreas da *computação* e *ciências humanas*. As dificuldades se acentuam ainda mais em ambientes de *processamento* e *memória* restritos (como telefones celulares e *PDA*s). Por isso, a utilização de *frameworks* e *padrões de*

[1] Tal estudo foi realizado neste trabalho com o objetivo de selecionar as tecnologias mais adequadas e promissoras para a implementação da primeira versão do protótipo de nosso motor gráfico. Os critérios utilizados nessas análises encontram-se descritos no Apêndice A (página 47).

[2] O nome *mOGE* origina-se de uma abreviação para *mO*bile *G*raphics *E*ngine.

projeto para esse tipo específico de *software* mostra-se crucial para a sobrevivência de qualquer projeto profissional de um jogo [34, 6, 59], fazendo com que eles encapsulem grande parte dessa complexidade em bibliotecas mais simples de utilizar ou modelos de arquitetura a serem adotados[28, 29, 36, 25].

Tendo em vista a importância do problema de desenvolver *frameworks* que simplifiquem o desenvolvimento de jogos, o *Centro de Informática* da *Universidade Federal de Pernambuco* têm buscado aprofundar estudos nessa área por meio de uma série de trabalhos em pesquisa e desenvolvimento de "motores de jogos" (*game engines*) voltados tanto a plataformas como *PC*s e *consoles*[62] quanto a dispositivos móveis, como telefones celulares e *PDA*s[86, 58, 5, 75, 22].

Entretanto, por falta de pessoal com um conhecimento mais profundo em *Computação Gráfica 3D*, essas pesquisas tinham se restringido a *frameworks* para jogos 2D e 2.5D. Tal fato acabou por refletir no mercado de jogos em todo estado (devido à importância deste centro), pois até mesmo as empresas locais de maior sucesso têm sido ainda bastante tímidas quanto à criação de jogos 3D, os quais são a grande atração do momento (responsáveis por grande parte dos atuais, e previstos, bilhões de dólares movimentados pela indústria de *entretenimento digital interativo*).

1.2 OBJETIVOS

Como dito anteriormente, o desenvolvimento profissional de jogos demanda a utilização de diversas ferramentas de apoio, em especial a utilização de um *framework*.

Dos componentes de um "motor de jogos", aquele que mais atrai a atenção dos jogadores e encapsula boa parte da complexidade do desenvolvimento de um jogo é o módulo gráfico (comumente conhecido como "motor gráfico").

Embora nosso *Centro de Informática* tenha adquirido bastante experiência no desenvolvimento de "motores de jogos", é uma deficiência grave a falta de trabalhos que tratem da criação de jogos 3D. Isso porque, sendo Recife um pólo nacional da indústria de jogos, é necessário um domínio desse assunto de forma a possibilitar uma absorção das novas tecnologias dessa área para não perder a competitividade.

Assim, esse trabalho tem dois objetivos básicos: apresentar ao leitor, nos moldes de um *survey*, os principais tópicos de *Computação Gráfica 3D* (dos mais básicos aos do estado da arte) utilizados no desenvolvimento de jogos 3D orientados a dispositivos móveis, buscando assim documentar um longo e profundo estudo do estado atual das pesquisas dessa área, posto que, até onde sabemos, não existe um tal *survey*. Bem como objetivamos projetar um "motor gráfico 3D", de forma a estudar como as restrições de processamento e

memória dos dispositivos móveis influenciariam na criação do módulo gráfico de um "motor de jogos 3D".

Tudo isso visa a orientar tanto futuras pesquisas nessa área quanto interessados em adquirir conhecimento necessário ao desenvolvimento de *frameworks* para a criação de jogos 3D para dispositivos móveis. [3]

1.3 DESAFIOS

A construção de um motor gráfico 3D para jogos em dispositivos móveis impõe uma série de desafios devido à simplicidade e restrições desses dispositivos. Em pleno auge das placas aceleradoras gráficas para *PC*s, bibliotecas multimídia, computadores com *DVD*s e abundância de memória e processamento, o desenvolvimento de aplicações para dispositivos móveis pode ser avaliada como uma retomada aos quesitos de restrições de memória, processamento, recursos gráficos, e vários outros, que existiam nos computadores de 10 anos atrás. Em se tratando de jogos, essa situação é similar àquela da era dos primeiros videogames, onde uma série de otimizações de código e artifícios de programação eram necessários para que os jogos se tornassem viáveis para a tecnologia disponível.

Assim, uma série de adaptações são necessárias aos algoritmos utilizados hoje em dia na implementação de *frameworks*, várias delas podendo ser inspiradas em soluções encontradas nos bons jogos de 10 anos atrás.

Tal situação é agravada ainda mais porque os usuários de hoje estão muito mais exigentes com relação à qualidade das imagens geradas, principalmente, por conta dos jogos para *PC*s e consoles atuais, os quais têm uma qualidade tão superior a dos jogos para celulares de hoje que chegam a se aproximar do foto-realismo conseguido por efeitos especiais utilizados no cinema.

A situação fica muito pior ao tentar lidar tanto com as limitações dos dispositivos-alvo, quanto pela complexidade da matemática e dos algoritmos necessários para gerar gráficos 3D com uma qualidade aceitável.

1.4 VISÃO GERAL

O restante deste trabalho está estruturado da seguinte maneira: no próximo capítulo trataremos das diferenças entre o desenvolvimento de jogos para *PC*s e consoles e daqueles voltados a dispositivos móveis (analisando tanto a criação de jogos 2D/2.5D quanto a de jogos 3D).

A primeira contribuição deste trabalho será introduzida no capítulo 3,

[3] Neste ponto, se faz necessário um pequeno esclarecimento. Os tipos de dispositivos móveis em que focaremos nossas atenções neste trabalho serão os *telefones celulares*, devido ao *expertise* tanto do autor quanto da comunidade de jogos móveis do Recife.

onde relacionamos (na forma de um "*survey*") o estado da arte dos principais tópicos da *Computação Gráfica 3D* cujo conhecimento é necessário no desenvolvimento de um motor gráfico para jogos tridimensionais em dispositivos móveis.

A segunda contribuição aparece no capítulo 4, o qual é dedicado ao *mObile Graphics Engine*. Nele enfatizamos a concepção e as decisões tomadas durante o desenvolvimento do *mOGE*, sendo de grande importância para que o leitor possa compreender a arquitetura e as soluções aplicadas durante sua concepção. Como também os escopo e objetivos da primeira versão de seu protótipo.

Finalmente, no capítulo 5, serão apresentadas nossas conclusões tiradas no desenvolvimento deste trabalho, bem como suas contribuições e possibilidades de trabalhos futuros relacionados.

CAPÍTULO 2

DESENVOLVIMENTO DE JOGOS

"Find a job you like and you add five days to every week."
—H. JACKSON BROWN, JR.

Fazendo um retrospecto sobre o desenvolvimento dos jogos de computadores, desde os anos 70 até hoje, houve uma grande mudança na metodologia de desenvolvimento. Enquanto durante um bom tempo a tendência era escrever cada jogo inteiramente desde o início, com praticamente nenhuma reusabilidade de código construído anteriormente, especialmente de terceiros, hoje em dia essa indústria tem dado bastante importância aos benefícios que a *Engenharia de Software* pode proporcionar a um projeto. Além disso, com o advento de novas linguagens de programação adotando técnicas de *Orientação a Objetos*, tais como C++ e Java, passou-se a dar considerável importância à *metodologia de desenvolvimento*, *reusabilidade*, *modularidade*, *abstração* e vários outros critérios de *qualidade de software*.

Neste capítulo, com o objetivo de situar o leitor, faremos uma *breve* revisão da forma como têm sido desenvolvidos jogos para *PC*s e *telefones celulares*.

2.1 DESENVOLVIMENTO DE JOGOS PARA COMPUTADORES PESSOAIS E CONSOLES

"Um jogo de computador pode ser definido como um sistema interativo que permite ao usuário experimentar, sem riscos, uma situação de conflito. Quase sempre, como pano de fundo dessa situação existe um enredo, que define do que se trata o jogo, as regras que o jogador deve seguir e os objetivos que ele deve se esforçar para atingir."[91]

Uma peça chave no desenvolvimento de jogos é o motor do jogo. O motor tem como principal objetivo servir de base ao desenvolvimento de jogos, principalmente por se supor que ele irá executar uma grande parte das tarefas de baixo nível necessárias em todos os jogos, permitindo que o desenvolvedor foque seus esforços nos aspectos de alto nível (mais relacionados com o enredo e regras específicas do jogos em desenvolvimento).

Entretanto, o problema da construção de sistemas complexos como *frame-*

works e motores[1] de jogos é o *controle dos riscos e dos problemas de desenvolvimento*. Neste caso, existem diversos aspectos que devem ser considerados e não é raro que o projeto, por falta de uma boa estrutura e planejamento, fracasse como vítima de sua própria complexidade.

A confirmação disso pode ser obtida pelo infeliz resultado do jogo *Golgotha*, desenvolvido pela *Crack dot Com*[24], que era um projeto comercial para o desenvolvimento de um jogo com motor proprietário 3D. Neste projeto, doze pessoas estiveram envolvidas durante dois anos de trabalho, e quinhentos mil dólares foram gastos, mas o projeto foi encerrado incompleto devido a sua não conclusão após o período preestabelecido, sendo então o código fonte liberado para o público. Isso exemplifica o enorme esforço que deve ser empregado para tal tarefa, mesmo que sejam utilizados extensamente os princípios de *Engenharia de Software*. Atualmente, um grupo de desenvolvedores de jogos está tentando retomar o projeto, com o *Golgotha Forever*[33].

Portanto, é importante, ao se projetar um framework ou motor de jogos, que se tenha em mente seus objetivos e propósitos, para evitar estes tipos de prejuízos.

Após uma série de estudos envolvendo a análise de motores existentes, [86] propôs uma compilação de módulos e arquitetura necessários a motores de jogos, dos quais foi possível extrair uma adaptação aplicável ao desenvolvimento de motores para dispositivos móveis. Um esquema de sua arquitetura e descrições da comunicação entre seus módulos componentes seguem, respectivamente, na Figura 2.1 e nas Tabelas 2.1 e 2.2.

O presente trabalho se deterá no detalhamento e projeto do módulo gráfico dessa arquitetura[2]. Diferentemente de [86], nosso módulo gráfico objetiva gerar imagens de mundos tridimensionais aplicando tanto tópicos tradicionais quanto do estado atual da arte da *Computação Gráfica 3D*.

2.2 DESENVOLVIMENTO DE JOGOS PARA DISPOSITIVOS MÓVEIS

O desenvolvimento de jogos sempre esteve na vanguarda do uso das novas tecnologias e sempre empurrando a produção da indústria para avanços tecnológicos que permitissem acesso a mais memória e mais recursos gráficos. Entretanto, no ambiente dos dispositivos móveis, os desenvolvedores vêem-se

[1] Os motores diferem dos *frameworks* pois já são aplicações quase completas, para um domínio bem mais específico do que o de um framework. Os motores de jogos já rodam sozinhos, bastando ao desenvolvedor acrescentar as regras do jogo.

[2] *Não* é o foco deste trabalho o estudo dos demais módulos de tal arquitetura, posto que já foram bem estudados em trabalhos anteriores desenvolvidos neste mesmo centro[86, 58, 5, 75, 22].

Figura 2.1. Comunicação entre os módulos da arquitetura[86]

novamente com restrições encontradas nos primeiros jogos produzidos há 10 anos atrás, isto significa, nada mais de placas aceleradoras para gráficos 3D[3], som estéreo, joysticks, etc...

Além das restrições impostas pela própria arquitetura dos dispositivos móveis, outra questão encontrada pelos desenvolvedores é a linguagem/plataforma para ser utilizada no desenvolvimento. Esta escolha, em geral, baseia-se nos recursos oferecidos por cada uma delas e, também, pelo número e modelos de dispositivos que suportam estas plataformas.[4]

Os jogos desenvolvidos para aparelhos celulares são limitados pelas características dos próprios dispositivos. Comparando com o desenvolvimento para *PC*s e *consoles*, os dispositivos móveis representam um grande desafio para os programadores devido às grandes restrições existentes quando comparadas com o desenvolvimento tradicional de jogos. Outro aspecto complicador é a variedade de modelos com telas e recursos diferentes e as diferenças de implementação das plataformas de cada um deles. Estas diferenças e restrições

[3]Embora já estejam surgindo empresas como a *Bitboys*[7], especializada no desenvolvimento de placas aceleradoras 3D para dispositivos móveis.

[4] No caso de jogos 3D, há ainda a preocupação com a escolha da biblioteca gráfica a ser utilizada no desenvolvimento, onde uma escolha errada pode ser determinante na performance do jogo (e, conseqüentemente, em suas vendas).

geram a necessidade da adoção de soluções criativas e otimizadas.

Além das restrições de recursos, outra grande característica que limita os jogos desenvolvidos para dispositivos móveis[5] é a forma de interação com o jogador. Os aparelhos não foram feitos para jogar e por isso o tamanho de tela, as cores apresentadas, a falta de suporte a som dificultam o estabelecimento de uma boa relação entre o jogo e o jogador. A entrada de dados também é bastante prejudicada por causa do teclado reduzido e da falta de suporte para o tratamento de mais de uma tecla pressionada ao mesmo tempo. Bem como, devido à ausência de processadores de ponto-flutuante em alguns dispositivos (como aparelhos celulares), é necessário trabalhar com *aritmética de ponto-fixo*, onde é pré-determinada a quantidade de *bits* alocados tanto para o valor inteiro quanto para a mantissa do número representado, o que abre possibilidades para mais erros numéricos em cálculos utilizados pelos algoritmos que geram gráficos 3D (exigindo, ainda mais, o conhecimento de "truques" específicos que diminuam esses erros e não afetem, *muito*, a performance do jogo).

São essas e outras restrições que tornam o desenvolvimento de jogos para dispositivos móveis tão complexo e tão necessária a utilização de *frameworks* e motores que simplifiquem essa tarefa.

2.3 CONCLUSÕES

Neste capítulo, procuramos apresentar ao leitor os problemas relacionados ao desenvolvimento de jogos tanto para plataformas tradicionais, como *PC*s e consoles, quanto para os novos dispositivos móveis, como telefones celulares.

Vimos que há uma série de preocupações adicionais ao se decidir criar jogos para *hardwares* mais restritos e que, se em plataformas mais flexíveis a utilização de *frameworks* e motores específicos para a criação profissional de jogos é bastante útil e produtiva, seu uso no desenvolvimento de jogos para dispositivos móveis é praticamente via de regra, daí a importância deste estudo e de outros já realizados em trabalhos anteriores de alunos e pesquisadores do Centro de Informática[86, 58, 5, 75, 22].

Na Tabela 2.3, relacionamos as principais diferenças existentes entre as plataformas tradicionais e as móveis. É bastante útil mantê-las em mente ao se avaliar a aplicabilidade de determinadas técnicas e algoritmos comumente utilizados em jogos tradicionais no desenvolvimento de jogos para aparelhos celulares.

[5] Em especial, aqueles para telefones celulares.

Tabela 2.1. Módulos da arquitetura de jogos para computadores[86]

Gerenciador de Entrada	Encarregado de identificar eventos dos dispositivos de entrada e encaminhar o ocorrido para outro módulo que realizará algo.
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.
Gerenciador de Som	Responsável pela execução de sons a partir de eventos. Para isso deve ser capaz de converter e processar amostras de som, pois em geral jogos profissionais trabalham com formatos de arquivos proprietários, diferentes daqueles conhecidos pelo sistema operacional ou API utilizada.
Gerenciador de Inteligência Artificial	Gerencia o comportamento de objetos controlados pela máquina. Para isso, realiza certas ações de acordo com o estado atual do jogo e algumas regras. Sua complexidade varia de acordo com o tipo de jogo ou nível.
Gerenciador de Múltiplos Jogadores	Tem o objetivo de permitir que jogadores do mesmo jogo comuniquem-se entre si. Para esse fim, deve lidar com o gerenciamento da conexão e troca de informações entre os diversos computadores conectados via *Internet* ou *Intranet*.
Gerenciador de Objetos	Uma das partes mais fundamentais da arquitetura. Deve ser capaz de gerenciar um grupo de objetos do jogo. Isso envolve armazená-los em alguma estrutura de dados e controlar o ciclo de vida dos mesmos. Em geral, um jogo contém vários gerenciadores de objetos, que além de gerenciarem parte do jogo também trocam informações entre si.

Tabela 2.2. (*Cont.*) Módulos da arquitetura de jogos para computadores[86]

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, etc. Além disso, deve possuir métodos necessários para a execução do jogo, tais como, detectar colisão com outros objetos e desenhar-se.
Gerenciador do Mundo	É o responsável por armazenar o estado atual do jogo. Conta com os diversos gerenciadores de objetos para realizar tal tarefa. Em geral, é associado com um editor de cenários que descreve o estado inicial do mundo em cada nível do jogo. Tal descrição resultará em delegações de tarefas para os gerenciadores de objetos a fim de que eles iniciem seus objetos de acordo com a especificação.
Editor de Cenários	Ferramenta para descrição de estados do jogo de forma visual e sem necessidade de programação. Em geral, é utilizada para gerar as diversas instâncias do jogo (níveis). Ao final, deve gerar um arquivo a ser processado pelo gerenciador do mundo para iniciar cada nível.
Gerenciador Principal	Ponte para troca de informações entre algumas outras partes do jogo. Representa a "fachada"de todo o projeto, no sentido que é um local de acesso único ao sistema.

Tabela 2.3. Diferenças entre plataformas de desenvolvimento de jogos

***PC*s e Consoles**	**Telefones Celulares**
Grande poder de processamento	Pequeno poder de processamento
Megabytes de memória disponíveis em *heap*	*Kilobytes* disponíveis em *heap*
Gigabytes de memória disponíveis para *armazenamento*	*Megabytes* de memória disponíveis para *armazenamento*
Multi-tarefa	Mono-tarefa
Escalonamento Preemptivo	Escalonamento Cooperativo
Disponibilidade de processamento de números em *ponto-flutuante*	Representação de números em *ponto-flutuante* substituída pela baseada em *ponto-fixo*

CAPÍTULO 3

DESENVOLVIMENTO DE MOTORES GRÁFICOS 3D

"It is wonderful that so simple a figure as the triangle is so inexhaustible."

—LEOPOLD CRELLE

Um motor gráfico 3D não é nada mais que um sistema gráfico 3D[51, 26, 1, 28, 29] que sintetiza imagens em tempo-real respondendo a chamadas feitas pela aplicação (no caso, o jogo). Para o desenvolvimento de tais sistemas, é necessário o conhecimento de uma ampla variedade de técnicas, algoritmos, estruturas de dados e conhecimentos matemáticos. É esse tipo de conhecimento que é abstraído pelo usuário de um motor 3D, por isso a produtividade do desenvolvimento de jogos é maior com a utilização de um motor pré-existente.

Neste capítulo, apresentamos ao leitor a primeira de nossas principais contribuições: um *survey* dos principais tópicos da *Computação Gráfica 3D* que julgamos de aplicabilidade interessante no desenvolvimento de motores gráficos 3D *para dispositivos móveis*. Aqui teremos uma visão geral de cada tópico referenciando livros e artigos relevantes que aprofundam o tema em estudo[1], objetivando dessa maneira fornecer aos interessados na criação de motores 3D as principais fontes para um aprendizado sólido de cada um dos assuntos abordados[2]

[1] É importante explicar que visões mais detalhadas desses tópicos foram omitidas devido à complexidade e extensão de cada um desses tópicos, muitos dos quais são assuntos tratados por livros dedicados especificamente a eles.

[2] Como esse capítulo se trata de um estudo do estado da arte da computação gráfica aplicada a jogos, é necessário que o leitor tenha conhecimentos básicos da terminologia e conceitos dessa área e assuntos correlatos (e.g. cálculo, geometria analítica, álgebra linear, algoritmos, estruturas de dados, ótica, etc...). Caso o leitor ainda não possua tais conhecimentos, duas fontes de muito boa qualidade contendo tanto conceitos básicos quanto avançados dessas áreas são [27, 51].

3.1 PIPELINE GRÁFICO

É devido à complexidade e variedade dos cálculos necessários para a geração de imagens pelo computador, que o processo de síntese de imagens é subdividido em vários estágios. O resultado da decomposição desse processo é comumente conhecido como "*pipeline gráfico*"[3].

O *pipeline gráfico* descreve todo o fluxo de processamento pelo qual a descrição de uma cena (ou mundo) tridimensional é transformada em uma imagem. Assim, um sólido entendimento desse processo permite o desenvolvimento de motores que se adequem ao *pipeline* de forma a minimizar o tempo necessário para gerar a imagem a partir da descrição de sua cena. Posto que, como lidamos com sistemas interativos e *hardware* bastante restritivo, nossa meta é conseguir gerar imagens tão rapidamente e a um menor custo de qualidade quanto for possível. De forma a permitir liberar tempo de processamento às outras tarefas do jogo (e.g. *IA*, som, regras específicas do jogo etc.)

De maneira geral, o processamento de uma sistema gráfico 3D de tempo-real pode ser dividido em três estágios principais[10, 1, 28, 29], os quais podem ser refinados em estágios ainda menores: o estágio de *aplicação*, o de *geometria* e o de *rasterização*[4].

3.1.1 Estágio de aplicação

No *estágio de aplicação*, o desenvolvedor do motor tem completo controle sobre o que acontece no sistema, visto que esse estágio é implementado em *software*. Assim, é possível alterar sua implementação de forma que se obtenham ganhos em performance. Entretanto, nos outros estágios esse controle não é total, posto que várias partes desses estágios são implementadas em *hardware* ou não existe a possibilidade de modificar sua implementação. Embora seja ainda possível afetar, indiretamente, o tempo de processamento nesses estágios, por exemplo, diminuindo (já nesse estágio) o número de faces a serem renderizadas (e.g. utilizando alguma técnica de *occlusion culling*, onde polígonos que não são visíveis pela câmera são descartados do *pipeline*). Ao fim desse estágio, toda a geometria (pontos, linhas e triângulos) a ser renderizada é passada ao estágio seguinte.

[3] Devido à analogia com a decomposição do fluxo de processamento em uma *CPU*, denominada *pipeline*.

[4] É importante observar que há um responsável por cada um desses estágios. Para o *estágio de aplicação*, esse responsável é o desenvolvedor do motor. Tanto a *API* quanto o *hardware* gráficos são responsáveis pelo *estágio de geometria*, enquanto que o *estágio de renderização* é quase sempre implementado completamente em *hardware*.

3.1.2 Estágio de geometria

O *estágio de geometria* é responsável por grande parte das operações sobre polígonos e vértices, sendo comumente subdividido nos seguintes *estágios funcionais*: *transformações de modelo e vista*, *iluminação e sombreamento*, *projeção*, *recorte* e *mapeamento em tela*.

3.1.2.1 Transformações de modelo e vista Todo objeto da cena é transformado em e para vários *espaços* (ou *sistemas*) de coordenadas. Inicialmente, o objeto encontra-se em seu próprio *espaço de modelo*, onde seus vértices e normais são descritos em *coordenadas de modelo* (ou seja, ele ainda não foi transformado de nenhuma maneira). A cada objeto é associada uma *transformação de modelo* de forma a orientar e posicionar o objeto no mundo em que está inserido, possibilitando que várias cópias (chamadas *instâncias*) do mesmo objeto tenham diferentes localizações e orientações na cena sem replicação da geometria básica (proporcionando uma economia em memória).

Após a transformação de modelo, os objetos possuem posicionamentos e orientações únicas na cena coexistindo em um único *espaço mundial*, onde seus vértices e normais são descritos em *coordenadas de mundo*.

A câmera possue posicionamento e orientação próprias descritos em coordenadas mundiais. Entretanto, para simplificar os processos de projeção e recorte, os componentes da cena (incluindo as fontes de luz) são submetidos a uma única transformação que alinha a posição da câmera com a origem e sua orientação de forma que a atenção da câmera é alinhadas com o eixo Z[5], o eixo Y é orientado para cima e o X para a direita. Essa transformação é conhecida como *transformação de vista* e os vértices transformados estão em *coordenadas de vista* do *espaço de vista*.

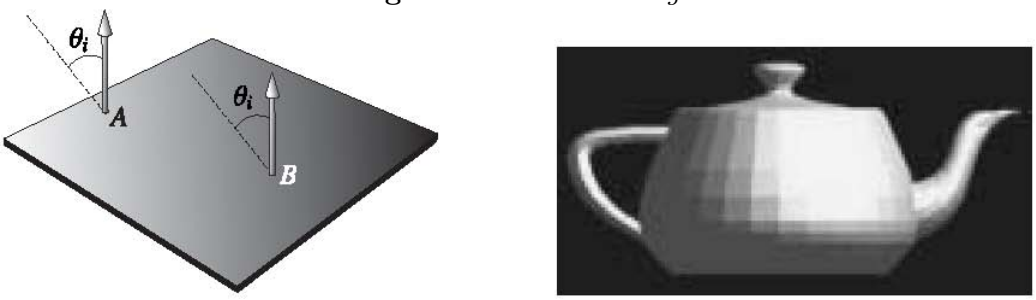
3.1.2.2 Iluminação e sombreamento Em geral, a cena possue uma ou mais fontes de luz, as quais podem (ou não) afetar a aparência da geometria. No caso em que as luzes afetam a aparência dos objetos, é necessária a aplicação de um *modelo de iluminação* (ou sombreamento), utilizado para calcular a cor de um dado vértice com respeito ao posicionamento da câmera e das fontes de luz.

São vários os esquemas de sombreamento propostos na literatura de computação gráfica[98]. Entretanto, os mais utilizados em aplicações como jogos são os modelos locais: *flat* e de *Gouraud*.

[5] Alguns textos preferem alinhar a orientação da câmera com o eixo Z negativo. Entretanto, a diferença é basicamente semântica, visto que a transformação entre um e outro é muito simples.

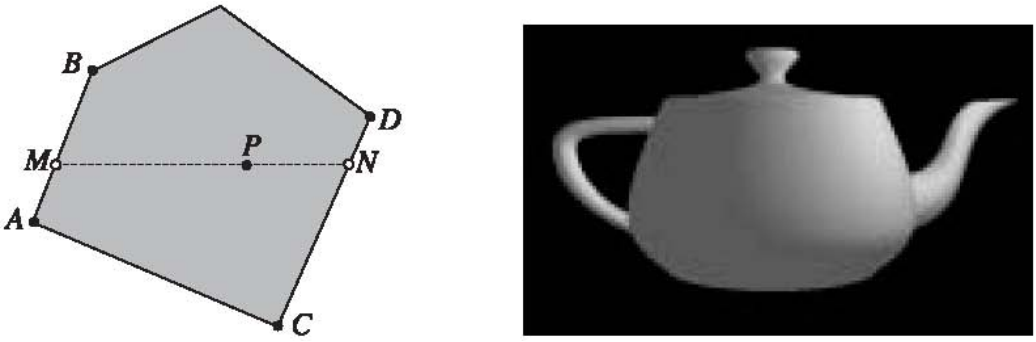
- *Flat shading*: baseia-se na aplicação de uma equação de iluminação em um dos pontos do polígono, a cor resultante é atribuída a todos os outros pontos desse polígono[97]. (*Ver figura 3.1.*)

Figura 3.1. *Flat shading*



- *Gouraud shading*: uma equação de iluminação é aplicada em cada um dos vértices do polígono e as cores resultantes são interpoladas de forma a encontrar as cores dos demais pontos desse polígono[44]. (*Ver figura 3.2.*)

Figura 3.2. *Gouraud shading*

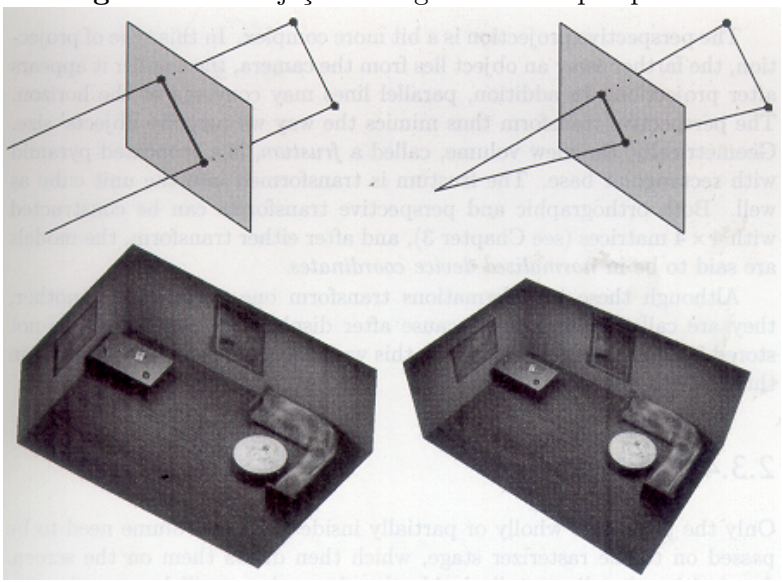


Como os dispositivos móveis são bastante restritos quanto ao poder de processamento, é aconselhável utilizar o modelo de *flat shading* o máximo possível, visto que seus custos computacionais são significativamente menores que aqueles necessários ao modelo de *Gouraud*.

3.1.2.3 Projeção Após serem iluminados, os pontos são projetados no plano da imagem (vista como um quadrado unitário). As principais transformações de projeção utilizadas em jogos (e aplicáveis a motores para dispositivos móveis) são:

- *Projeção ortográfica (ou paralela)*: onde se simula uma câmera com ponto focal localizado no infinito (muito utilizada em desenhos técnicos).
- *Projeção em perspectiva*: tenta simular o comportamento geométrico da luz sobre uma câmera ideal, onde a abertura do ponto em que entra a luz (diafragma) tende a zero (modelo largamente mais utilizado em jogos). (*Ver figura 3.3.*)

Figura 3.3. Projeções ortográfica e em perspectiva



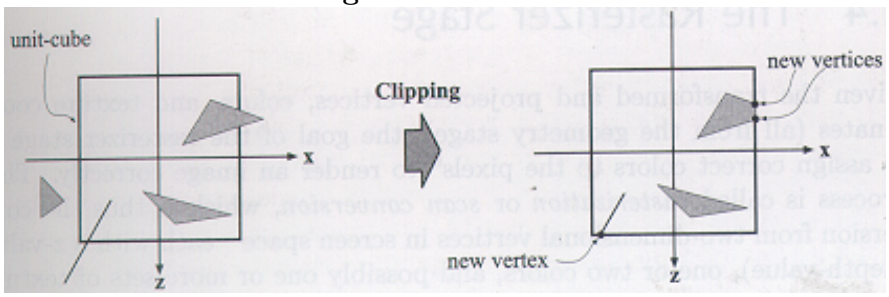
Após a projeção, diz-se que os pontos encontram-se em *coordenadas de dispositivo normalizadas* (ou nas *NDC*[6]).

3.1.2.4 Recorte Sabendo-se que apenas os polígonos completa ou parcialmente dentro do campo de visão da câmera precisam ser processados, todos os objetos completamente fora desse campo são descartado neste estágio. Assim, aqueles que estão completamente dentro do campo de visão são repassados para seu mapeamento em tela. Entretanto, os polígonos que estão apenas parcialmente dentro do volume de visão devem ser recortados para que

[6]Do inglês, *Normalized Device Coordinates*.

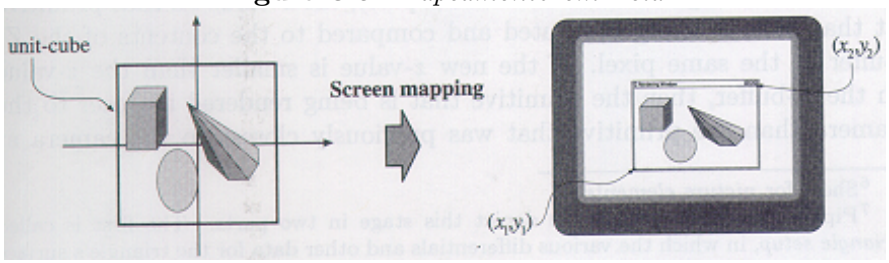
somente polígonos completamente visíveis sejam repassados para o estágio de mapeamento em tela, simplificando esse outro estágio. (*Ver figura 3.4.*)

Figura 3.4. *Recorte*



3.1.2.5 Mapeamento em tela Apenas primitivas completamente dentro do volume de visão são passadas ao estágio de mapeamento em tela, e as coordenadas (*NDC*) são ainda tridimensionais, relativas ao mapeamento do molume de visão em um cubo unitário (realizado durante o estágio de projeção), quando entram neste estágio. As coordenadas x e y de cada primitiva são transformadas para *coordenadas de tela* (uma simples transformação afim). Tais coordenadas junto com as coordenadas z da *NDC* são também chamadas de *coordenadas de janela*. Esse processo está exemplificado na figura 3.5.

Figura 3.5. *Mapeamento em Tela*



3.1.3 Estágio de rasterização

Dados todos os triângulos com os vértices (devidamente transformados e projetados), cores e coordenadas de texturas (tudo vindo do estágio de geometria), é responsabilidade do *estágio de rasterização* atribuir as cores corretas aos *pixels* para renderizar uma imagem corretamente. Tal processo

é comumente conhecido como *rasterização*[7] (ou *scan conversion*, pois é a conversão de vértices bidimensionais do espaço de tela – cada qual com um valor de profundidade z, uma ou duas cores, e possivelmente um ou mais conjuntos de coordenadas de texturas associadas a cada vértice – em *pixels* na tela[8]).

Nas figuras 3.6 e 3.7, são apresentados os *pipelines* das bibliotecas gráficas *Direct3D* e *OpenGL*. Note que eles detalham apenas os estágios de geometria e rasterização, posto que o estágio de aplicação é parte do *software* que utiliza tais bibliotecas.

Figura 3.6. Diagrama do *Pipeline* gráfico da biblioteca gráfica *Direct3D*[18]

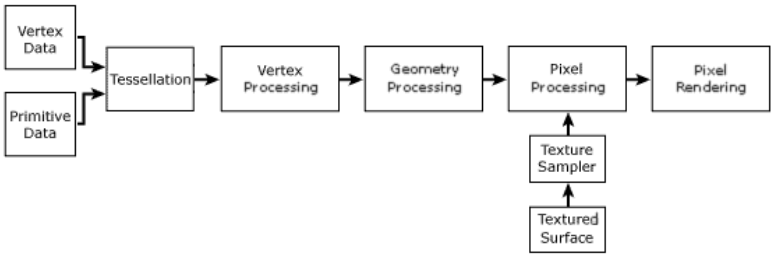
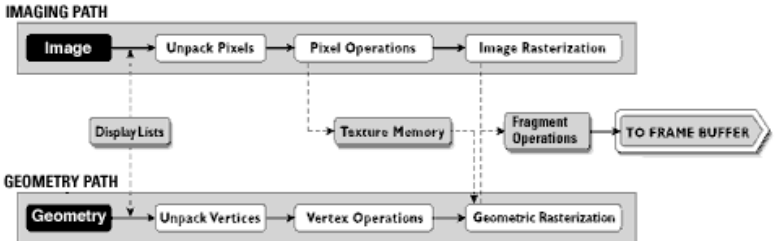


Figura 3.7. Diagrama do *Pipeline* gráfico da biblioteca gráfica *OpenGL*[82]



3.2 REPRESENTAÇÃO DE OBJETOS

As restrições de memória e processamento dos dipositivos móveis influenciam fortemente a escolha das estruturas de dados com utilização viável em um motor gráfico. Devido a essa influência, as representações dos objetos

[7] Do inglês, *rasterization*.

[8] Alguns diagramas de *pipelines* subdividem esse estágio em duas partes. A primeira é chamada *triangle setup*, onde uma série de dados de pré-processamento são coletados sobre o triângulo a ser rasterizado, e a segunda parte é a rasterização em si, na qual os *pixels* são checados e preenchidos.

devem admitir uma implementação tão *compacta* e de *simples* manipulação quanto possível.

Outro fator importante que restringe nossa escolha é o conjunto de primitivas geométricas disponibilizado pelas bibliotecas gráficas 3D para dispositivos móveis (e as atuais propostas de *hardware* gráfico para esses dispositivos). Com o objetivo de simplificar sua implementação, tais bibliotecas e placas oferecem, comumente, apenas três primitivas básicas (*ponto*, *linha* e *triângulo*) que são passadas ao *pipeline gráfico* como *arrays* de vértices ou *arrays* de *índices* para um *array* de vértices, ambos interpretados pelo *pipeline* como conjuntos desconexos de primitivas (pontos, linhas e triângulos isolados) ou como seqüências conectadas (*loops*, *strips* ou *fans*). Nas figura 3.8, 3.9, 3.10, 3.11, 3.12 e 3.13, exemplificamos algumas maneiras de representar objetos gráficos disponibilizadas por tais bibliotecas.

Figura 3.8. *Point lists*

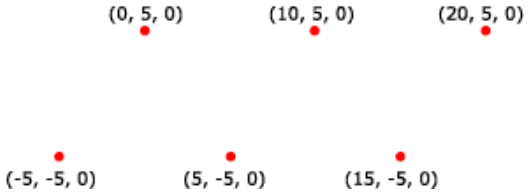


Figura 3.9. *Line lists*

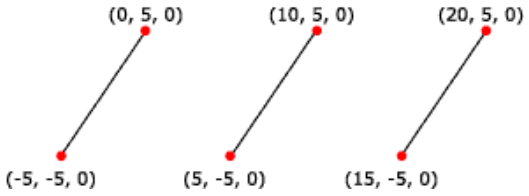


Figura 3.10. *Line strips*

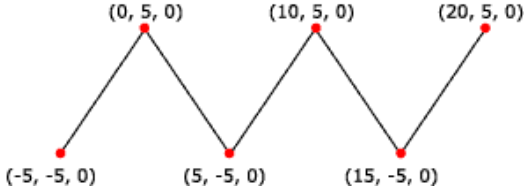


Figura 3.11. *Triangle lists*

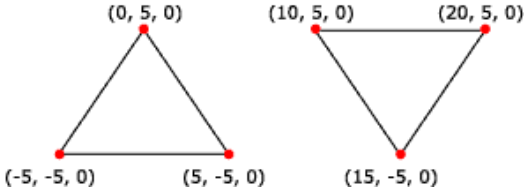
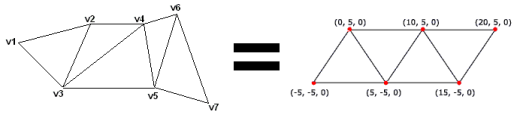


Figura 3.12. *Triangle strips*



A literatura apresenta uma grande variedade quanto às maneiras de representar objetos gráficos de uma cena 3D. Embora exista tão grande quantidade de representações possíveis, as restrições apresentadas indicam uma representação baseada em *malhas simpliciais*.

Como estamos principalmente interessados em representar superfícies, o tipo de malha que nos mais interessa é a *malha de triângulos* (tipicamente resultante de *triangulações* de conjuntos de pontos, modeladores 3D e *triangularizações* de superfícies implícitas ou de alta-ordem).

Isso nos remete ao estudo de algoritmos e estruturas de dados que permitam (de forma simples e compacta) manipular e representar *malhas de triângulos*.

3.2.1 Técnicas poligonais

As técnicas de manipulação de malhas triangulares mais relevantes à implementação de *frameworks* 3D para dispositivos móveis englobam algoritmos de *triangularização*, *stripificação* e *simplificação*.

3.2.1.1 Triangularização É considerada o processo de *subdivisão* de superfícies em um conjunto de triângulos. Nas literaturas de jogos, computação gráfica e métodos de elementos finitos são abundantes os algoritmos de triangularização, dentre eles podemos citar [87, 64, 57, 23] como fontes bastante relevantes para o estudo de diferentes técnicas de triangularização e para a implementação de ferramentas e motores gráficos 3D para celulares.

Na figura 3.14, exemplificamos esse conceito com diferentes triangularizações de um heptágono.

Figura 3.13. *Triangle fans*

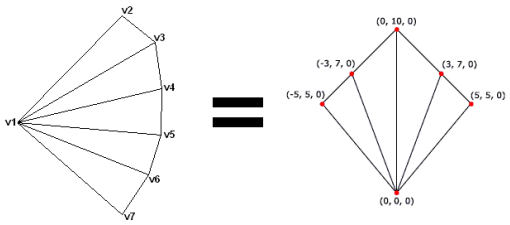
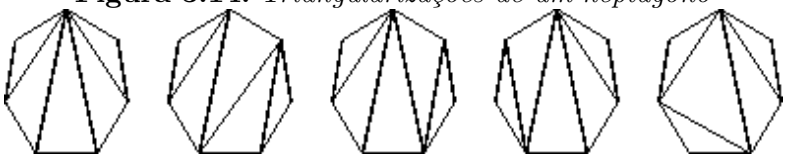
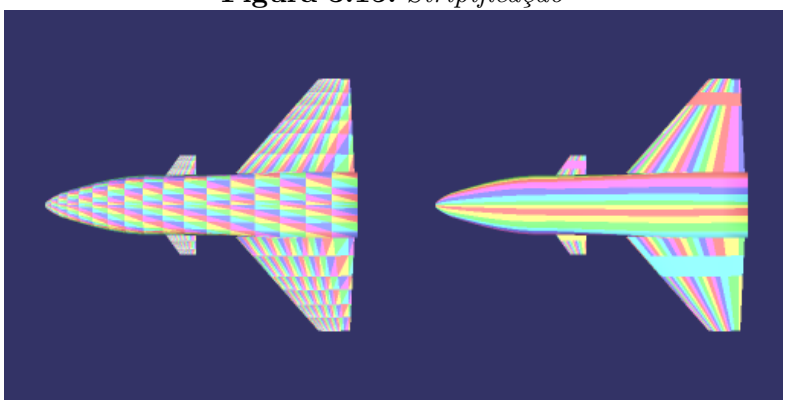


Figura 3.14. *Triangularizações de um heptágono*



3.2.1.2 Stripificação É o processo de geração de *strips* de triângulos a partir de uma malha triangular. A obtenção de *triangle strips* ótimas é problema *NP-Completo*[31], assim, é necessário a utilização de métodos heurísticos para uma geração com um bom aproveitamento da coerência entre os triângulos. [69, 92, 35, 45] são boas referências para o estudo e implementação desses métodos. (*Ver figura 3.15.*)

Figura 3.15. *Stripificação*

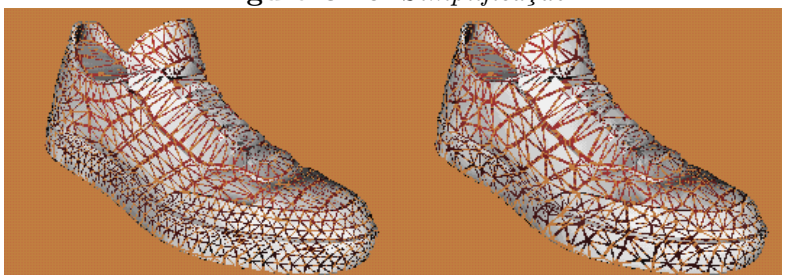


A *stripificação* é de grande utilidade em nosso objetivo de manipular os objetos de forma simples e com baixo consumo de memória. Isso porque os *triangle strips* são primitivas nativas das bibliotecas e placas gráficas 3D (aumentando a performance de renderização) e oferecem um bom grau de

compactação dos objetos em simples listas de vértices (diminuindo os custos de memória e simplificando sua manipulação).

3.2.1.3 Simplificação Também conhecida como *redução de dados* e *decimação*, é o processo de tomar uma malha triangular[9] e diminuir seu número de faces com a mínima perda de qualidade possível. (*Ver figura 3.16.*)

Figura 3.16. *Simplificação*



Uma classificação das técnicas de simplificação e um vasto *survey* são fornecidos em [21, 20], enquanto [61] é uma ótima referência ao atual estado da arte dessa ativa área de pesquisa.

Acreditamos que a utlização dessas técnicas durante a criação de conteúdo para o jogo (por meio de ferramentas que as implementem) traria bastantes benefícios aos requisitos de memória e processamento[10].

3.3 GRAFOS DE CENA

O modelo de programação baseado em *grafos de cena* permite uma abstração de *como* os objetos da cena 3D são efetivamente *enviados ao* e *processados pelo pipeline gráfico* implementado pelo motor. Esse modelo faz uso de um paradigma declarativo para descrever os objetos e suas localização e orientação na cena 3D. Dessa maneira, simplificando o trabalho do desenvolvedor e possibilitando tanto uma independência da biblioteca gráfica quanto alterações na implementação das funções do *grafo de cena* sem que isso afete a aplicação (obviamente, desde que as interfaces sejam mantidas)[93, 28, 29].

Um *grafo de cena* consiste de objetos (chamados *nós*) organizados de forma hierárquica em uma árvore. Sendo assim, não é possível a ocorrência de

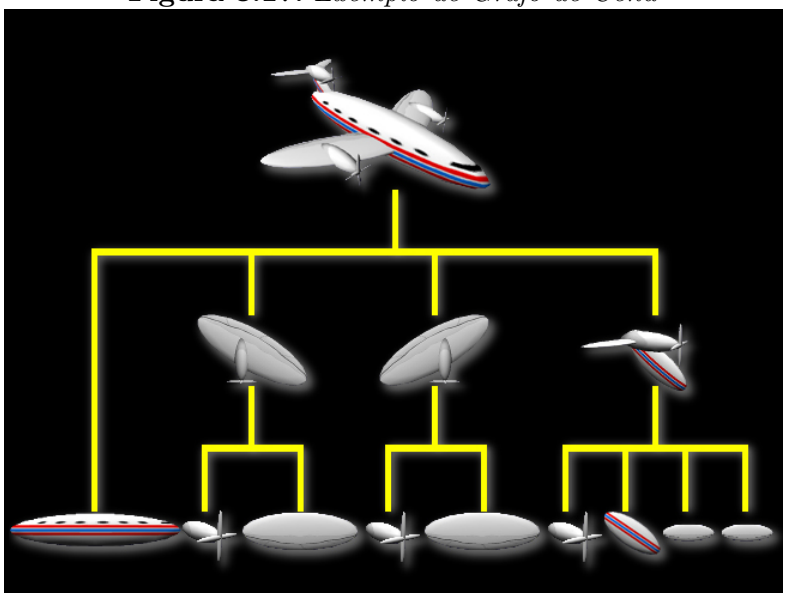
[9] Na verdade, é uma técnica aplicada para malhas *poligonais* gerais.

[10] Note que sua utilização no motor seria redundante e apenas aumentaria o tamanho do código-fonte. Assim, julgamos que suas aplicações trazem mais benefícios como um pré-processamento para a geometria da cena 3D.

ciclos nos relacionamentos entre os nós do *grafo de cena*[11]. Assim, podemos simplificar ainda mais a utilização do *grafo de cena* liberando o desenvolvedor de preocupações como a (des)alocação da memória ocupada pelo *grafo de cena* por meio da utilização do padrão de projeto *Smart Pointer*[25]. Dessa forma, oferecendo um modelo de programação mais *simples* e *robusto*.

Na figura 3.17, exemplificamos a estrutura hierárquica de um modelo de avião que pode ser diretamente mapeada em um *grafo de cena*[12].

Figura 3.17. *Exemplo de Grafo de Cena*



3.4 ANIMAÇÃO

Uma representação hierárquica da cena 3D fornece, além das vantagens apresentadas anteriormente, uma maneira concisa de animar os objetos do mundo. Com essa representação, é possível, por exemplo, movimentar um avião para frente sem ter que, explicitamente, transladar cada um dos seus vértices, apenas precisando ordenar à subárvore gerada pelo nó principal do avião que todos os seus filhos sejam movimentados.

[11] Embora, internamente, seja possível reaproveitar objetos já alocados e manter apenas as diferenças entre as instâncias dos objetos, tornando a estrutura do *grafo de cena* uma *DAG*, diminuindo a memória total utilizada. Tal estratégia é semelhante ao padrão de projeto *Flyweight*[36].

[12] Note que é possível reaproveitar as instâncias de partes mais simples do avião (como hélices e asas). Assim, não é necessário manter mais de uma cópia da geometria dessas partes, diminuindo a utilização total da memória.

São várias as técnicas de animação existentes e aplicadas na criação de conteúdo para jogos[98, 89, 90]. Abaixo, relacionamos e descrevemos brevemente aquelas que julgamos mais relevantes ao desenvolvedor de um motor gráfico 3D para dispositivos móveis.

3.4.1 Animação de corpos rígidos

Essa técnica fundamenta-se em *transformações de corpos rígidos* (translações e rotações) que variam com o tempo. Assim, dependedo do instante de tempo e das interações entre corpos, os objetos da cena mudam suas localizações e orientações de alguma maneira.

3.4.2 Animação de deformações e skinning

Para possibilitar uma maior realismo na animação de personagens e objetos deformáveis, se utilizam técnicas que modificam a estrutura local dos corpos de forma a simular superfícies que contraem e esticam como resposta a forças externas (esse tipo de animação é bastante relacionada com técnicas de modelagem e animação físicas[4] e constuma ser denominada *skinning*). [98, 28, 2] fornecem uma boa introdução a esse tópico. (*Ver figura 3.18.*)

Figura 3.18. *Exemplo de Skinning*

3.4.3 Animação por keyframes

O processo de *animação por keyframes* fundamenta-se na definição de uma série de posições intermediárias enquanto o motor interpola os *frames* restantes. A figura 3.19 exibe quatro de uma série de *keyframes* que compõem uma animação.

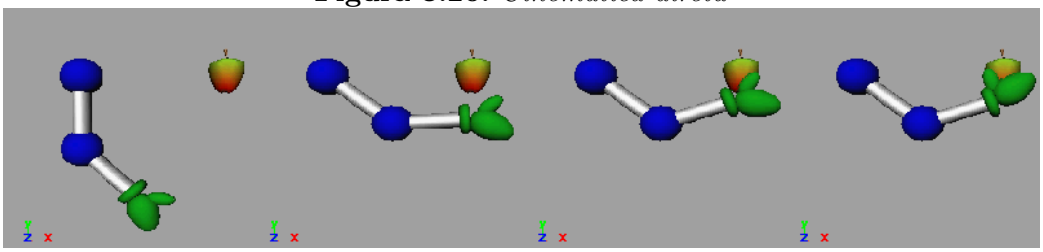
Esse processo possue uma série de variantes, mas as principais são duas: a *cinemática direta* e a *cinemática inversa*.

3.4.3.1 Cinemática direta É um sistema de animação *top-down*, onde os movimentos dos objetos filhos são relativos àqueles do objeto pai. Por exemplo, para poder segurar uma simples maçã, um robô rotaciona seu ombro,

Figura 3.19. *Animação por keyframes*

então seu ante-braço, seguido de seu braço e, finalmente, sua mão. Note que esse movimento é muito mecânico e nada natural, ou seja, nem um pouco indicado para a animação de personagens humanos ou animais[98, 28, 29]. (*Ver figura 3.20.*)

Figura 3.20. *Cinemática direta*



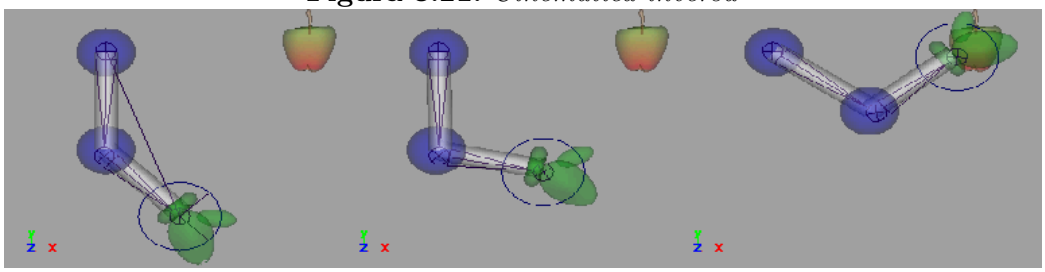
3.4.3.2 Cinemática inversa É um sistema de animação *bottom-up*, cujo princípio é completamente oposto ao da cinemática direta. Tem como vantagem um resultado mais natural nas interpolações de movimento entre os *keyframes*[98, 28, 29]. (*Ver figura 3.21.*)

3.5 TEXTURIZAÇÃO

Texturização é um processo que altera a aparência de uma superfície em cada um de seus pontos utilizando como base informações de alguma imagem, função ou outra fonte de dados[1].

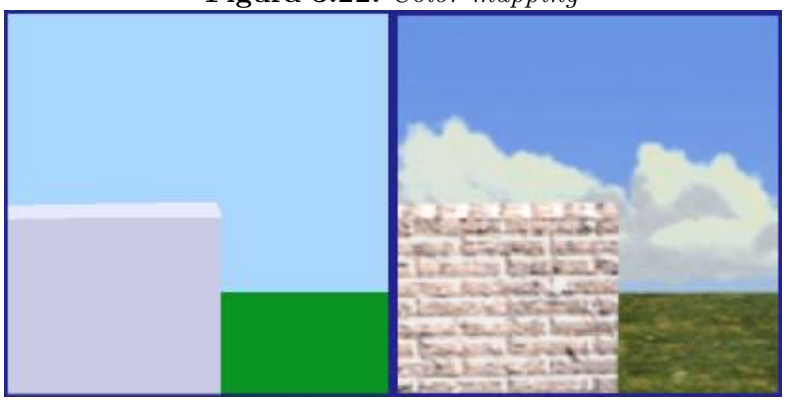
Essa definição do processo de *texturização* engloba uma série de diferentes técnicas que simplificam o trabalho do artista e produzem uma série efeitos bastantes realistas[54, 96, 84]. Dentre essas, podemos citar como

Figura 3.21. *Cinemática inversa*



sendo de aplicação interessante em um motor para dispositivos móveis: *color mapping*[14], *bump mapping*[9], *environment mapping*[8, 77] e *light mapping*[67, 12, 76]. (*Ver figuras 3.22, 3.23, 3.24 e 3.25.*)

Figura 3.22. *Color mapping*



Uma grande fonte de referências sobre técnicas de mapeamento de texturas é [85].

3.6 SELEÇÃO DE OBJETOS: PICKING

O termo *picking* se refere ao processo de selecionar um objeto 3D a partir de sua projeção 2D na tela com o auxílio de um apontador (seja ele um *mouse* ou um marcador controlado pelo teclado). Para um modelo de câmera com projeção em perspectiva, esse processo equivale a traçar um raio do foco da câmera passando pelo *pixel* da tela selecionado até o mundo 3D.

O suporte à seleção de objetos em um *grafo de cena* resulta em uma busca recursiva, através dos nós desse grafo, pelo objeto cujo ponto intersectado pelo raio é o mais próximo do foco da câmera dentre todos os outros objetos da cena.

Figura 3.23. *Bump mapping*

O problema de intersectar raios com objetos geométricos ainda é um dos problemas bastante estudados em computação gráfica e várias formas de otimização desse processo já foram propostas[42, 43].

A inclusão de um módulo de *picking* em um motor gráfico 3D (para aparelhos celulares) é discutível, visto que a seleção de objetos foi primeiramente projetada para sua utilização baseada em *mouses* ou canetas de *touch-pad*s e *touch-screen*s (esse tipo de caneta é bastante utilizado em dispositivos *PDA*s, o que justifica a inclusão desse tópico neste trabalho).

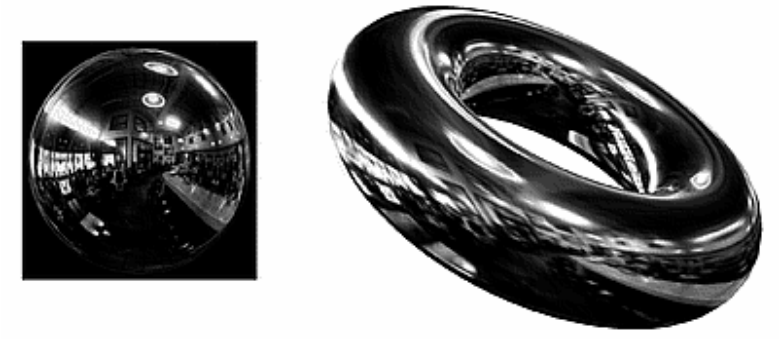
3.7 DETECÇÃO DE COLISÕES

A *detecção de colisões* é um problema fundamental em várias aplicações de *computação gráfica*, *realidade virtual*, *CAD/CAM*, *animação*, *modelagem física*, *robótica*, dentre muitas outras. Devido à essa grande variedade de aplicações, o campo de pesquisa em técnicas de *detecção de colisões* é, ainda hoje, bastante ativo.

Assim, é enorme a quantidade de algoritmos e estruturas de dados já projetados com o objetivo de oferecer soluções eficientes a esse problema. [88, 1, 28, 29] são ótimas referências (bem como ótimos *surveys*) tanto para tópicos básicos quanto avançados dessa intensa área de pesquisas[13].

[13]Note que o próprio problema de *picking*, tratado na seção anterior, é um caso particular

Figura 3.24. *Environment mapping*



3.8 EFEITOS ESPECIAIS BASEADOS EM IMAGENS

Uma série de efeitos visuais bastante realísticos pode ser conseguida com a utilização de imagens como primitivas. Desde complexos efeitos ópticos originados da interação entre as luzes e as lentes da câmera até a simulação de fumaça e fogo podem ser simulados com pequenos "truques" de manipulação de imagens a um custo relativamente pequeno tanto de processamento quanto de consumo da memória. Nós consideramos essas técnicas como um ferramental de grande valor para os artistas e que pode ser agregado a um motor a um custo de desenvolvimento relativamente pequeno.

Aqui citamos e exemplificamos alguns dos mais utilizados efeitos especiais baseados em imagens: *sprites*[37, 65], *billboards*[102, 3, 66], *lens flare*[63, 103, 15], *sistemas de partículas*[53, 66, 99, 100, 56, 98] e *skyboxes*[52]. *Ver figuras 3.26, 3.27, 3.28, 3.29 e 3.30*[14].)

3.9 CONCLUSÕES

Neste capítulo, abrangemos uma grande variedade de tópicos básicos e avançados de *Computação Gráfica 3D* cuja aplicabilidade no desenvolvimento de motores e jogos 3D para dispositivos móveis é bastante viável.

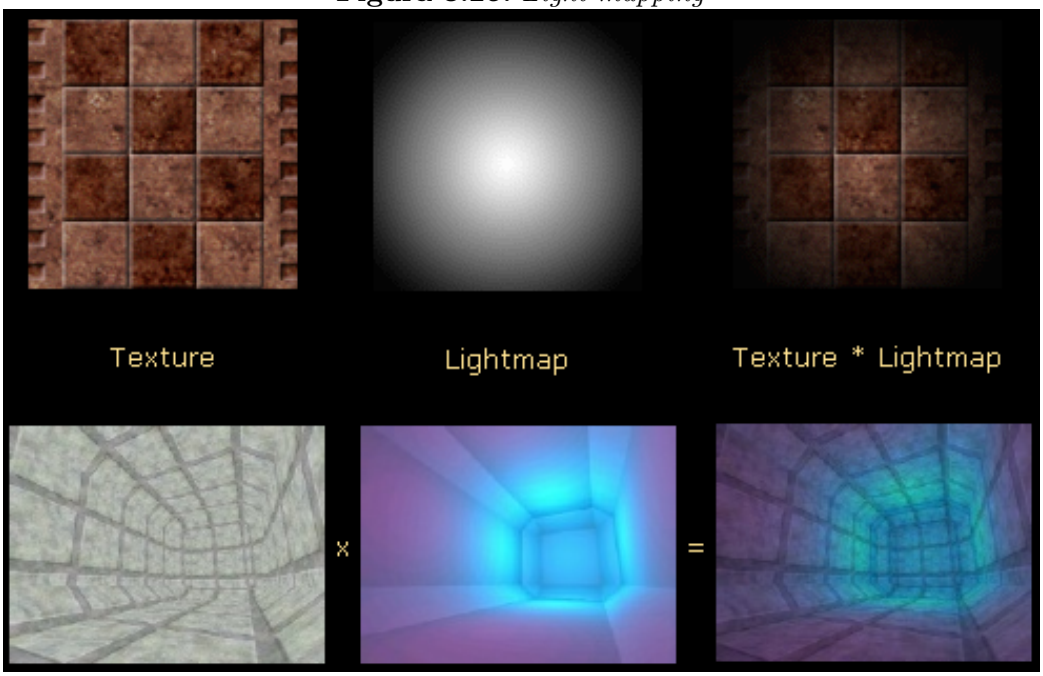
Buscamos apresentar as principais técnicas utilizadas hoje no desenvolvimento de jogos 3D que podem ser adaptadas ou mesmo trazidas sem adaptações para o mundo dos dispositivos móveis.

Além disso, foram apresentadas várias referências tradicionais e do estado da arte para aqueles realmente interessados em estudar os detalhes dos

no problema geral de detecção de colisão (visto que não passa de uma determinação da colisão entre um raio e um objeto).

[14] *Screenshot* do jogo "Seven Seas", desenvolvido na disciplina "Projeto e Implementação de Jogos" durante este período letivo.

Figura 3.25. *Light mapping*



algoritmos e técnicas comentados, tarefa essencial àqueles que pretendem desenvolver um motor gráfico de qualidade.

Figura 3.26. *Sprites*

Figura 3.27. *Billboards*

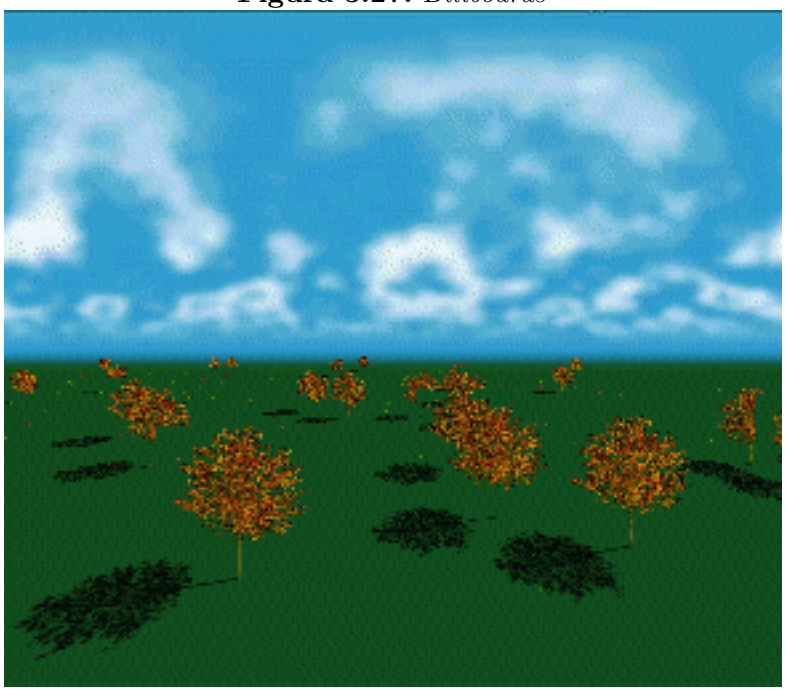


Figura 3.28. *Lens flare*

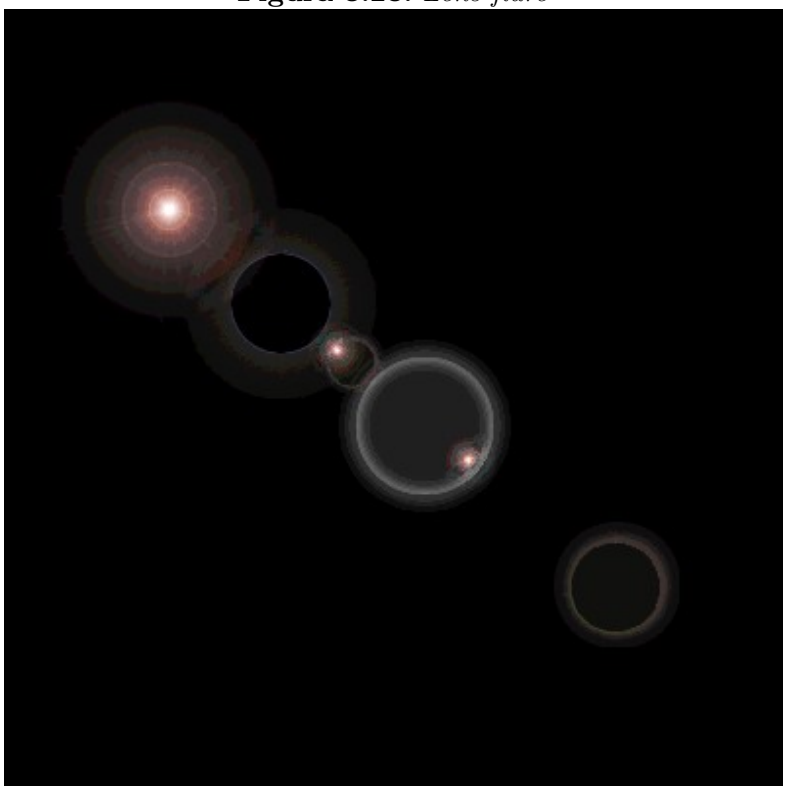


Figura 3.29. *Sistema de partículas*

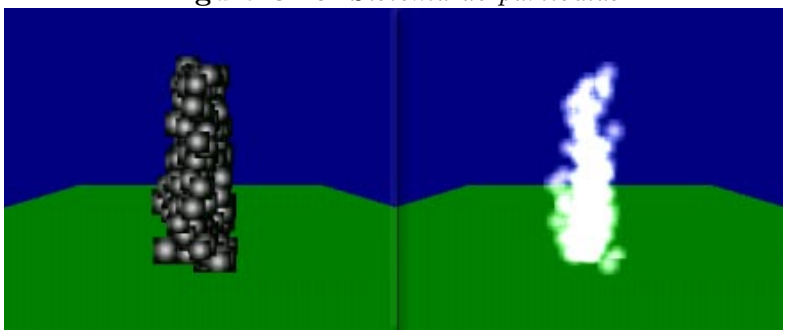


Figura 3.30. *Skyboxes*

CAPÍTULO 4

MOBILE GRAPHICS ENGINE

". . . , building software will always be hard.
There is inherently no silver bullet."
—FREDERICK P. BROOKS, JR.

Neste capítulo, apresentamos ao leitor como se deu o processo de elicitação dos requisitos do *mOGE*, bem como o projeto de sua arquitetura.

Além disso, relatamos como se deu a implementação do protótipo e os resultados atingidos.

4.1 ELICITAÇÃO DOS REQUISITOS

A elicitação dos requisitos do *mOGE* se deu com base no estudo dos módulos gráficos de outros motores 3D existentes tanto para *PC*s[94, 95] quanto para *aparelhos celulares*[79]. Além disso, nos baseamos em nossa experiência no desenvolvimento de sistemas gráficos 3D e aplicações para aparelhos celulares. Assim, foram coletadas as informações referentes aos tópicos da *Computação Gráfica 3D* apresentadas no capítulo anterior.

Dessa maneira, os requisitos coletados de nosso sistema são aquelas técnicas do capítulo 3 diretamente aplicáveis no *mOGE*.

4.2 PROJETO DA ARQUITETURA

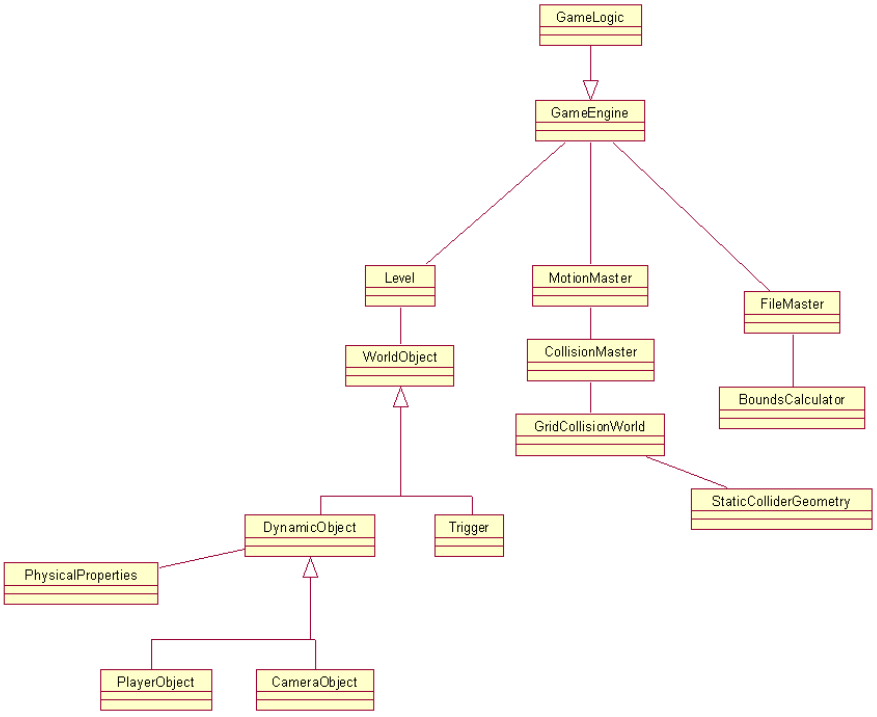
A arquitetura do *mOGE* foi largamente influenciada por outras arquiteturas desenvolvidas especificamente para jogos em dispositivos móveis[86, 79]. Ela busca manter uma estrutura *modular* com tarefas bem definidas para cada um de seus componentes.

Abaixo, detalhamos a arquitetura adotada para o motor e as tarefas atribuídas a cada um de seus componentes.

4.2.1 Arquitetura

Os módulos da arquitetura do *mOGE* encontram-se ilustrados na figura 4.1 e têm suas responsabilidades descritas na tabela 4.1.

Figura 4.1. Arquitetura



4.3 IMPLEMENTAÇÃO DO PROTÓTIPO

Com o objetivo de iniciar o processo de validação da arquitetura, foi implementado um protótipo com algumas funcionalidades básicas dentre as descritas no capítulo anterior.

4.3.1 Escopo do protótipo

Para validar a utilidade e produtividade conseguidas com o projeto do motor gráfico, seria necessário código implementado da maneira mais tradicional (fazendo chamadas diretas à biblioteca gráfica de baixo nível, em nosso caso a *OpenGL ES*). Entretanto, não dispunhamos de tempo suficiente para implementar o protótipo e os exemplos (que deveriam ser implementados duas vezes, uma em *OpenGL ES* e outra com o *mOGE*). Sendo assim, decidimos reaproveitar o código distribuído com o *SDK* do *Symbian OS* e apenas reimplementá-los com os módulos do *mOGE*, do qual só seriam implementados aqueles necessários para replicar os exemplos vindos na distribuição.

Na seção seguinte, avaliamos os resultados obtidos.

4.3.2 Resultados

Durante a implementação dos casos de validação, pudemos notar uma grande diferença quanto à simplicidade do código escrito em comparação ao código distribuído com o *SDK*. Além disso, notamos uma significativa diminuição do código-fonte, posto que boa parte das rotinas gráficas já estava implementada no *mOGE*. Na figura 4.2, mostramos os *screenshots* de algumas das pequenas aplicações que implementamos[1].

4.4 CONCLUSÕES

Neste capítulo, avaliamos o projeto da arquitetura do *mObile Graphics Engine* bem como validamos seu projeto com a implementação de um pequeno protótipo, no qual já foi possível notar melhoras na tarefa de desenvolver pequenas aplicações que lidam com gráficos tridimensionais em tempo-real.

[1]Em ordem, temos exemplos da utilização de malhas triangulares coloridas, fontes de luz pontuais e *Gouraud shading*, *environment mapping*, *billboarding* e dois exemplos que incorporam tanto *billboarding* quanto *sistemas de partículas*.

Tabela 4.1. Módulos da arquitetura do *mOGE*

Game Engine	A classe central do motor. Cria o *KeyboardListener*, *MotionMaster* e configura a janela da aplicação. Também cria o mundo e adiciona a ele o conteúdo de *Level*. O "loop" principal da aplicação está localizado nesta classe.
GameLogic	É uma subclasse de *GameEngine* (mas *não* faz parte do motor). A intenção é que essa classe defina o jogo, suas regras específicas e comportamento.
MotionMaster	Manipula a movimentação e orientação dos objetos do mundo. A cada ciclo do jogo, seu método *move* é chamado com o tempo decorrido e essa classe propaga esse tempo a todos os objetos dinâmicos. Os objetos têm a possibilidade de serem movidos obedecendo algumas regras físicas simples, chamando um outro método de *MotionMaster* (*doPhysics*). Em cooperação com o *CollisionMaster* é também uma das classes principais do sistema de *detecção de colisão*.
CollisionMaster	É o responsável por checar as colisões entre os objetos da cena.
GridCollisionWorld	Representa uma grade 3D no mundo de forma que cada cubo dessa grade guarda referências aos objetos estáticos da cena. A implementação foi baseada em uma *octree* com o objetivo de não desperdiçar memória com as áreas da grade que não contém nenhum objeto.

Tabela 4.2. (*Cont.*) Módulos da arquitetura do *mOGE*

BoundsCalculator	Nela, existem métodos utilitários específicos para o cálculo de esferas e caixas limitantes para objetos.
FileMaster	Oferece uma interface para a implementação de *parsers* para diferentes formatos de *level*.
WorldObject	O tipo principal do jogo, todos os objetos (definidos pelo usuário do motor) que podem ser renderizados devem herdar dessa classe. Nela são armazenadas informações como transformações de modelo e posicionamento.
DynamicObject	Disponibiliza uma interface para a aplicação de certas transformações sobre o posicionamento e a orientação de um objeto da cena. Todos os objetos dinâmicos do mundo devem herdar dessa classe.
Level	Representa uma coleção de objetos e atributos necessários para descrever uma "parte isolada do mundo", como objetos (estáticos e dinâmicos), luzes, etc.
StaticColliderGeometry	Todos os objetos estáticos do mundo devem herdar dessa classe, pois apenas os objetos desse tipo são armazenados no *GridCollisionWorld*.

Figura 4.2. Resultados

CAPÍTULO 5

CONCLUSÕES

"I like the dreams of the future better than the history of the past."

—THOMAS JEFFERSON

Durante a realização deste trabalho, ficou clara a necessidade da utilização de *frameworks* específicos para um desenvolvimento profissional de jogos. De acordo com a proposta, foi elaborado um documento (*survey*) contendo muito do conhecimento adquirido nesse trabalho e um primeiro passo para o crescimento de uma comunidade de desenvolvimento de jogos 3D e criado o projeto de um motor gráfico 3D para dispositivos móveis, bem como um protótipo desse motor (o que é mais uma fonte para *trabalhos futuros*).

5.1 CONTRIBUIÇÕES

As principais contribuições desse trabalho se referem ao estudo, pioneiro neste Centro, de técnicas e tecnologias da *Computação Gráfica 3D* e suas aplicabilidades no desenvolvimento de jogos para dispositivos móveis.

Com o objetivo de disseminar o conhecimento adquirido durante todo o trabalho realizado e guiar novos pesquisadores nessa área, foi elaborado um *survey* com os principais tópicos da *Computação Gráfica 3D* apresentando referências relevantes tanto para aqueles que estão iniciando seus estudos quanto para os que pretendem aprofundar seus conhecimento com o estado da arte do desenvolvimento de jogos 3D (relevante à criação de motores gráficos para dispositivos móveis).

5.2 DIFICULDADES ENCONTRADAS

Por ser um projeto pioneiro, foram encontradas uma série de dificuldades durante o desenvolvimento desse trabalho. Dentre elas podemos citar:

- O desenvolvimento de aplicações para dispositivos móveis envolve preocupações com uma série de restrições, impondo um modelo de programação bastante diferente daquele comumente adotado no desenvolvimento de aplicações para *PC*s.

- A complexidade do conhecimento necessário para a criação de um motor 3D é um grande desafio, necessitando de profundo entendimento da computação gráfica e da matemática envolvida.

- A associação das duas dificuldades acima torna o problema ainda maior, pois é necessário prover ao usuário do motor uma abstração geral de boa parte desse conhecimento algorítmico e matemático, mas, ao mesmo tempo, desenvolver uma solução otimizada associada às restrições de programação para os dispositivos móveis.

- A falta de dispositivos nos quais testar as aplicações foi sempre uma preocupação. Isso porque nem sempre que uma aplicação executa sem problemas no emulador da plataforma, quer dizer que ela também irá executar bem no dispositivo. Como estamos trabalhando com uma tecnologia extremamente nova, é escassa a quantidade de dispositivos que suportam as tecnologias com que trabalhamos (e esse é um fato em nível mundial, tornando nosso trabalho ainda mais desafiador.

- Pouco tempo após o início desse projeto como um *trabalho de graduação*, foi *necessário* que o autor reduzisse drasticamente sua atenção dedicada a este trabalho durante um período de 2 meses, devido à necessidade de sua participação no *Programa de Verão* do *Instituto Nacional de Matemática Pura e Aplicada* (o que era condição necessária à sua admissão no *Programa de Mestrado* nessa mesma instituição).

- Durante o pouco tempo restante para o desenvolvimento deste trabalho, várias falhas na infra-estrutura do Centro de Informática (e.g. instabilidade da rede, cortes de energia elétrica não informados, baixa performance das máquinas, etc.) prejudicaram uma conclusão mais imediata do trabalho.

Todos esses fatores contribuíram para que o trabalho realizado tenha ficado mais simples que o esperado pelo próprio autor. Embora, essas mesmas dificuldades tenham, indiretamente, aberto novas possibilidades para trabalhos futuros.

5.3 TRABALHOS FUTUROS

Devido ao pioneirismo deste trabalho, são várias as possibilidades de trabalhos futuros. Dentre elas, podemos citar:

- Finalizar a implementação do protótipo da arquitetura proposta.

- Incorporar aos motores já existentes[86, 5, 75] um módulo gráfico 3D.
- Estudar as estruturas de dados que possibilitem melhor *trade-off* de processamento e uso da memória em diferentes arquiteturas de dispositivos móveis.
- Implementar técnicas mais avançadas para a aceleração dos algoritmos utilizados no motor.
- Validar a arquitetura com o desenvolvimento de um número significativo de jogos 3D.
- Avaliar os benefícios da utilização do motor desenvolvendo *outras* aplicações multimídias 3D (e.g. screensavers).

5.4 CONSIDERAÇÕES FINAIS

Neste capítulo apresentamos as nossas principais contribuições e possíveis trabalhos relacionados a este. É importante destacar que o principal propósito deste trabalho é prover um passo inicial ao estudo e desenvolvimento de técnicas da *Computação Gráfica 3D* com aplicabilidade no desenvolvimento de jogos (iniciativa pioneira neste Centro).

A originalidade deste trabalho é destacada na primeira tentativa (até onde sabemos) da elaboração de um *survey* com técnicas avançadas da *Computação Gráfica 3D* cuja utilização na criação de *frameworks* e motores de jogos para dispositivos móveis tem viabilidade potencial.

O uso de técnicas de *Engenharia de Software* e *Padrões de Projeto*[36, 25] auxiliou bastante o projeto do motor, principalmente, reforçando propriedades de *modularidade*, *portabilidade*, *robustez* e *performance*. Comprovando a relevância da prática de processos e padrões da *Engenharia de Software* no desenvolvimento tanto de jogos quanto das ferramentas envolvidas em sua criação.

APÊNDICE A

SELEÇÃO DE TECNOLOGIAS PARA O PROTÓTIPO

"We choose our joys and sorrows long before we experience them."

—KAHLIL GIBRAN

[1] Como em qualquer outro projeto de *software*, uma atenção especial deve ser dada à escolha das tecnologias utilizadas no desenvolvimento de um motor gráfico 3D. Essa preocupação deve ser levada ainda mais a sério no desenvolvimento para celulares, devido a todas as restrições de processamento e memória que são impostas por esses dispositivos.

Dessa maneira, antes de dar início à implementação do protótipo, foi realizado um estudo das principais e mais promissoras tecnologias para o desenvolvimento de aplicações 3D para telefones celulares. Estudo esse que foi dividido em duas partes: *Bibliotecas gráficas* e *Ambientes de desenvolvimento (e execução)*[2].

A.1 BIBLIOTECAS GRÁFICAS

A maior preocupação na escolha das tecnologias utilizadas no desenvolvimento de um motor gráfico 3D reside na definição da biblioteca gráfica utilizada em sua implementação.

Ao escolher uma biblioteca gráfica, vários fatores devem ser levados em consideração, dentre eles os principais seriam:

- *Padronização e estabilidade*: Softwares padronizados possuem uma API semanticamente mais segura, dando ao programador a segurança de que as chamadas a mesma terão comportamento nas mais diversas plataformas. A padronização, normalmente, é feita por empresas de

[1] Sendo o *projeto* de um motor gráfico 3D (independemente da plataforma utilizada) o propósito deste trabalho, a escolha das tecnologias para a implementação do protótipo são especificidades que não se encaixam nos objetivos deste texto. Assim, essa etapa de nosso estudo foi acrescentada ao documento na forma de um apêndice.

[2] Propósitos e características dessas bibliotecas e ambientes foram descritos no capítulo de introdução.

alto porte e conhecidas no meio o que de certa maneira assegura uma vida útil maior da API.

- *Suporte e adoção da indústria*: quanto maior o suporte industrial a uma biblioteca, maior a disponibilidade de dispositivos compatíveis (o que implica em uma diminuição de seus preços). Bem como maior o comprometimento com a qualidade de suas implementações e mais pessoas com acesso à tal tecnologia (mais clientes em potencial).

- *Consumo de processamento e armazenamento*: Levando em conta o dispositivo atacado pelo motor, o consumo com processamento e armazenamento é de extrema importância. Deve se salientar que esse tópico é de julgamento complicado, pois diferentes implementações do mesmo framework podem ter peformances bem diferentes.

- *Aproveitamento do* hardware: É necessário que o framework aproveite o que o hardware oferece, evitando perdas de performance desnecessárias.

- *Flexibilidade e extensibilidade*: Os sistemas de software atuais é um complexo conjunto com os mais variados componentes interagindo das mais diversas formas. Incluir novas funcionalidades em tais sistemas é um dos principais problemas da engenharia de software. Um framework flexível e extensível deve levar o programador a códigos, ao mesmo tempo, mais robustos e entendíveis.

- *Confiabilidade e portabilidade*: Softwares para dispositivos móveis precisam de cuidados dobrados nesse item. A maioria das empresas reguladoras, como o caso da "QUALCOMM"com o BREW, obrigam que os softwares lançados oficialmente para suas plataforma, sejam submetidos a rigidos e caros testes. Além disso, o hardware é muito diferente entre os diversos dispositivos. Normalmente, o mesmo programa sofre pequenas modificações para cada um dos aparelhos. Uma API portável diminuirá o custo e o tempo necessário para tais procedimentos.

- *Documentação disponível*: Documentação é importante pra qualquer API de alto porte. Conhecer a ferramenta utilizada é a maior arma que um engenheiro pode ter. A falta de documentação é fator responsavel em aumentar a curva de aprendizado, criando empecilhos a sua utilização e adoção.

- *Curva de aprendizagem*: A curva de aprendizagem determinada a aceitação da tecnologia como padrão na comunidade de desenvolvedores. Sem a

aceitação dos desenvolvedores, mesmo com apoio de grandes empresas, o framework não deve apresentar uma vida útil longa.

Levando em conta esses fatores, foram analisadas três diferentes bibliotecas: *OpenGL ES*, *M3G* e *Direct3D Mobile*.

A.2 AMBIENTES DE DESENVOLVIMENTO (E EXECUÇÃO)

Além das bibliotecas gráficas, foram analisados vários *frameworks* orientados ao desenvolvimento e execução de aplicações em telefones celulares. Buscou-se realizar um estudo com os mais proeminentes e promissores objetivando definir o ambiente no qual seria desenvolvido o protótipo do motor.

Para escolher qual seria utilizado nessa implementação, foram levados em consideração os seguintes fatores:

- *Expectativas de crescimento*: As tecnologias para dispositivos móveis são pouco consolidadas. Cada uma delas tem uma parte relevante na fatia de mercado. Sendo assim, é bastante complicado a escolha de uma tecnologia especifica. Uma escolha errada agora pode acarretar problemas a médio e longo prazo.

- *Atual fatia de mercado*: A situação presente de um tecnologia fornece bases para justificar o desenvolvimento de um software. Uma boa influência no mercado é um forte fator para que o software possa ser adotado por mais usuários.

- *Consumo de processamento e armazenamento*: Levando em conta o dispositivo atacado pelo motor, o consumo com processamento e armazenamento é de extrema importância. Deve se salientar que esse tópico é de julgamento complicado, pois diferentes implementações do mesmo framework podem ter peformances bem diferentes.

- *Padronização e estabilidade*: Softwares padronizados possuem uma API semanticamente mais segura, dando ao programador a segurança de que as chamadas a mesma terão comportamento nas mais diversas plataformas. A padronização, normalmente, é feita por empresas de alto porte e conhecidas no meio o que de certa maneira assegura uma vida útil maior da API.

- *Documentação disponível*: Documentação é importante pra qualquer API de alto porte. Conhecer a ferramenta utilizada é a maior arma que um engenheiro pode ter. A falta de documentação é fator responsavel em aumentar a curva de aprendizado, criando empecilhos a sua utilização e adoção.

- *Curva de aprendizagem*: A curva de aprendizagem determinada a aceitação da tecnologia como padrão na comunidade de desenvolvedores. Sem a aceitação dos desenvolvedores, mesmo com apoio de grandes empresas, o framework não deve apresentar uma vida útil longa.

Dessa maneira, foram estudados quatro diferentes *frameworks*: *Symbian*, *BREW*, *J2ME* e *Windows Mobile*.

A.3 CONCLUSÕES

Após estudar todas essas tecnologias e analisá-las com relação a todos os fatores apresentados, ficou decidido a adoção da biblioteca gráfica *OpenGL ES* (por seu bom desempenho em *todos* os fatores apresentados) e do ambiente de desenvolvimento *Symbian* na implementação do protótipo do motor (por seu *ambiente nativo*, *bem documentado*, com boas *atuação no mercado* e *expectativa de crescimento* e similaridades com o desenvolvimento tradicional para *PC*'s, implicando em uma boa *curva de aprendizado*).

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering, 2nd Edition*. A.K. Peters, 2002.

[2] Alex Ferrier. *Real-time Soft-object Animation using Free-form Deformation*. *Gamasutra*, 1999.

[3] Alexandre Meyer, Fabrice Neyret and Pierre Poulin. *Interactive Rendering of Trees with Shading and Shadows*. *12th Eurographics Workshop on Rendering*, pages 182–195, 2001.

[4] Andrew Witkin and David Baraff. *Physically Based Modeling*. *Course notes at SIGGRAPH 2001*, 2001.

[5] Tiago Guedes F. Barros. *SymbG(r)aF - Symbian Games Framework*. Trabalho de graduação, Universidade Federal de Pernambuco, 2003.

[6] Erik Bethke. *Game Development and Production*. Wordware Publishing, Inc., 2003.

[7] Bitboys. http://www.bitboys.com/, Março 2005.

[8] Blinn and Newell. *Texture and reflection in computer generated images*. *Communications of the ACM*, 19(10):542–547, 1976.

[9] James Blinn. *Simulation of wrinkled surfaces*. *Computer Graphics (SIGGRAPH'77 Proceedings)*, C-20:286–292, 1977.

[10] James Blinn. *Jim Blinn's Corner: A Trip Down The Graphics Pipeline*. Morgan Kaufmann, 1996.

[11] Qualcomm BREW. http://brew.qualcomm.com/, Março 2005.

[12] Brian Hook. *Multipass Rendering and the Magic of Alpha Blending*. *Game Developer*, 4(5):12–19, 1997.

[13] Mascot Capsule. http://www.mascotcapsule.com/, Março 2005.

[14] Edwin Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Tese de Doutorado, University of Utah, 1974.

[15] Chris Maughan. *Texture Masking for Faster Lens Flare. Game Programming Gems 2*, 2001.

[16] Swerve Client. http://www.superscape.com/products/swerve_client/, Março 2005.

[17] Hi Corporation. http://www.hicorp.co.jp/, Março 2005.

[18] Microsoft Corporation. *DirectX Graphics' Programming Guide*, Março 2005.

[19] Microsoft Corporation. http://www.microsoft.com/, Março 2005.

[20] David Luebke. *A Developer's Survey of Polygonal Simplification Algorithms. IEEE Computer Graphics & Applications*, 21(3):24–35, 2001.

[21] David Luebke. *Advanced Issues in Level of Detail. Course 41 notes at SIGGRAPH 2001*, 2001.

[22] Wabber Miranda de Arruda Filho. *Um Framework para Editores de Jogos desenvolvido com J2ME*. Trabalho de graduação, Universidade Federal de Pernambuco, 2002.

[23] de Berg, van Kreveld, Overmars and Schwarzkopf. *Computational Geometry–Algorithms and Applications*. Springer-Verlag, 2000.

[24] Crack dot Com. http://www.crack.com/, Março 2005.

[25] Bruce Powel Douglass. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley Professional, 2002.

[26] Luiz Velho e Jonas Gomes. *Sistemas Gráficos 3D*. Instituto Nacional de Matemática Pura e Aplicada, 2001.

[27] Jonas Gomes e Luiz Velho. *Fundamentos da Computação Gráfica*. Instituto Nacional de Matemática Pura e Aplicada, 2003.

[28] David H. Eberly. *3D Game Engine Design: A Pratical Approach to Real-Time Computer Graphics*. Morgan Kaufmann, 2000.

[29] David H. Eberly. *3D Game Engine Architecture: Engineering Real-Time Applications with Wild Magic*. Morgan Kaufmann, 2004.

[30] OpenGL ES. http://www.khronos.org/opengles/, Março 2005.

[31] Esther Arkin, Martin Held, Joseph Mitchell and Steven Skiena. *Hamiltonian Triangulations for Fast Rendering. IEEE Computer Graphics & Applications*, 21(3):24–35, 2001.

[32] Java Specification Request 184: Mobile 3D Graphics API for J2ME. http://jcp.org/jsr/detail/184.jsp, Março 2005.

[33] Golgotha Forever. http://golgotha.sourceforge.net/, Março 2005.

[34] Frederick P. Brooks, Jr. *No Silver Bullet: Essence and Accidents of Software Engineering. Computer Magazine*, 20(4):10–19, 1987.

[35] Gabriel Taubin and Jarek Rossignac. *Geometric Compression through Topological Surgery. ACM Transactions on Graphics*, 17(2), 1998.

[36] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, 1995.

[37] George Geczy. *2D Programming in a 3D World: Developing a 2D Game Engine Using DirectX 8 Direct3D. Gamasutra*, 2001.

[38] Gerbera. http://www.hybrid.fi/main/esframework/, Março 2005.

[39] Hybrid Graphics. http://www.hybrid.fi/, Março 2005.

[40] Superscape Group. http://www.superscape.com/, Março 2005.

[41] The Khronos Group. http://www.khronos.org/, Março 2005.

[42] Hanan Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing and GIS*. Addison-Wesley, 1989.

[43] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.

[44] Henri Gouraud. *Continuous shading of curved surfaces. IEEE Transactions on Computers*, C-20:623–629, 1971.

[45] Hughes Hoppe. *Optimization of Mesh Locality for Transparent Vertex Caching. Computer Graphics (SIGGRAPH 99 Proceedings)*, pages 269–276, 1999.

[46] Sony Computer Entertainment Inc. http://www.sony.com/, Março 2005.

[47] Qualcomm Incorporated. http://www.qualcomm.com/, Março 2005.

[48] DFC Intelligence. http://www.dfcint.com/news/prsep222004.html, Março 2005.

[49] Synergenix Interactive. http://www.synergenix.se/, Março 2005.

[50] Java 2 Platform, Micro Edition (J2ME). http://java.sun.com/j2me/, Março 2005.

[51] Steven K. Feiner James D. Foley, Andries van Dam and John F. Hughes. *Computer Graphics: Principles and Practice in C (2nd Edition)*. Addison-Wesley Professional, 1995.

[52] Jason Shankel. *Rendering Distant Scenery with Skyboxes. Game Programming Gems 2*, 2001.

[53] Jeff Lander. *The Ocean Spray in Your Face. Game Developer Magazine*, 5(7):13–19, 1998.

[54] Jeff Lander. *That's Wrap: Texture Mapping Methods. Game Developer Magazine*, 7(10):21–26, 2000.

[55] Jerry Edsall. *Animation Blending: Achieving Inverse Kinematics and More. Gamasutra*, 2003.

[56] John van der Burg. *Building an Advanced Particle System. Gamasutra*, 2000.

[57] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1998.

[58] Börje Felipe Fernandes Karlsson. *Incorporando Comportamentos de Movimentação e Aspectos de Modelagem Física ao wGEM*. Trabalho de graduação, Universidade Federal de Pernambuco, 2001.

[59] François Dominic Laramée, editor. *Secrets of The Game Business*. Charles River Media, 2003.

[60] Fathammer Ltd. http://www.fathammer.com/, Março 2005.

[61] Luebke, Reddy, Cohen, Varshney, Watson and Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann, 2002.

[62] Charles Andryê Galvão Madeira. *FORGE V8: Um framework para o desenvolvimento de jogos de computador e aplicações multimídia*. Dissertação de mestrado, Universidade Federal de Pernambuco, 2001.

[63] Mark Kilgard. *Fast OpenGL-rendering of Lens Flare*, Março 2005.

[64] Martin Held. *FIST: Fast Industrial-Strenght Triangulation of Polygons. Algorithmica*, 30(4):563–596, 2001.

[65] Masib McCuskey. *Using 3D Hardware for 2D Sprite Effects. Game Programming Gems*, 2000.

[66] McReynolds, Blythe, Grantham, Nelson. *Advanced Graphics Programming Techniques Using OpenGL. Course notes at SIGGRAPH 1999*, 1999.

[67] Michael Abrash. *Michael Abrash's Graphics Programming Black Book. Doctor Dobb's Journal*, 1997.

[68] Sun Microsystems. http://www.sun.com/, Março 2005.

[69] Mike Chow. *Using Strips for Higher Game Performance. Apresentação em* Meltdown X99.

[70] Direct3D Mobile. http://msdn.microsoft.com/library/en-us/wcemultimedia5/html/wce50oridirect3dmobile.asp, Março 2005.

[71] Windows Mobile. http://msdn.microsoft.com/mobility/windowsmobile/, Março 2005.

[72] Mophun. http://www.mophun.com/, Março 2005.

[73] John W. Muchow. *Core J2ME – Tecnologia & MIDP*. Pearson Makron Books, 2004.

[74] Nokia N-Gage. http://www.n-gage.com/, Março 2005.

[75] Ivo Frazão Nascimento. *Desenvolvimento de um Framework para Jogos sobre a Plataforma BREW*. Trabalho de graduação, Universidade Federal de Pernambuco, 2003.

[76] Naty Hoffman and Kenny Mitchell. *Photorealistic Terrain Lighting in Real Time. Game Developer*, 8(7):32–41, 2001.

[77] Ned Green. *Environment Mapping and Other Applications of World Projections. IEEE Computer Graphics & Applications*, 6(11):21–29, 1986.

[78] PlayStation Portable Developer Network. http://www.psp-pro.com/, Março 2005.

[79] Jonas Nilsson and Fredrik Nygren. Evaluation of Java 3D for mobile platforms with M3G. Dissertação de mestrado, Lund University, 2004.

[80] Nintendo. http://www.nintendo.com/, Março 2005.

[81] Nokia. http://www.nokia.com/, Março 2005.

[82] OpenGL.org. *OpenGL Overview*, Março 2005.

[83] Symbian OS. http://www.symbian.com/, Março 2005.

[84] Paul Haeberli and Mark Segal. *Texture Mapping as a Fundamental Drawing Primitive. Grafica Obscura*, 1993.

[85] Paul Heckbert. *Survey of Texture Mapping. IEEE Computer Graphics & Applications*, pages 56–67, 1986.

[86] Carlos André Cavalcante Pessoa. *wGEM: um Framework de Desenvolvimento de Jogos para Dispositivos Móveis*. Dissertação de mestrado, Universidade Federal de Pernambuco, 2002.

[87] Philip Schneider and David Eberly. *Geometry Tools for Computer Graphics*. Morgan-Kaufmann Press, 2002.

[88] Richard Parent. *Colision Detection*. Morgan Kaufmann, 2001.

[89] Richard Parent. *Computer Animation: Algorithms and Techniques*. Morgan Kaufmann, 2001.

[90] Richard Parent. Computer Animation: Algorithms and Techniques – *http://www.cse.ohio-state.edu/~parent/book/outline.html*. Março 2005.

[91] Sociedade Brasileira de Computação - SBC, 20., 2000, Curitiba. *Jogos por Computador - Histórico, Relevância Tecnológica e Mercadológica, Tendências e Técnicas de Implementação. Anais da XIX Jornada de Atualização em Informática*, 2:83, 2000.

[92] Stefan Gumhold and Wolfgang Straßer. *Real Time Compression of Triangle Mesh Connectivity. Computer Graphics (SIGGRAPH 98 Proceedings)*, pages 133–140, 1998.

[93] Sun Microsystems, Inc. *The Java 3D API Specification*.

[94] The Crystal Space 3D Team. Crystal Space 3D.

[95] The OGRE Team. OGRE 3D – Open source GRaphics Engine.

[96] Tito Pagán. *Efficient UV Mapping of Complex Models. Game Developer*, 8(8):28–34, 2001.

[97] W. Jack Bouknight. *A Procedure for Generation of Three-dimensional Half-toned Computer Graphics Presentations. Communications of the ACM*, 13:527–536, 1970.

[98] Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques*. Addison-Wesley Professional, 1992.

[99] William Reeves. *Particle Systems–A Technique for Modeling a Class of Fuzzy Objects. ACM Transactions on Graphics*, 2(2):91–108, 1983.

[100] William Reeves and Ricki Blau. *Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems. Computer Graphics (SIGGRAPH'85 Proceedings)*, pages 313–322, 1985.

[101] X-Forge2. http://www.fathammer.com/id/technology/, Março 2005.

[102] Yoshinori Dobashi, Kazufumi Kaneda, Hideo Yamashita, Tsuyoshi Okita and Tomoyuki Nishita. *A Simple, Efficient Method for Realistic Animation of Clouds. Computer Graphics (SIGGRAPH 2000 Proceedings)*, pages 19–28, 2000.

[103] Yossarian King. *2D Lens Flare. Game Programming Gems*, 2000.