Desenvolvimento de Jogos em Computadores e Celulares

André Luiz Battaiola [1], Rodrigo de G. Domingues [1], Bruno Feijó [2], Dilza Scwarcman [2], Esteban Walter G. Clua [2], Lauro Eduardo Kosovitz [2], Marcelo Dreux [3], Carlos André Pessoa [4], Geber Ramalho [4]

Resumo: Jogos por computador têm apresentado um surpreendente grau de evolução tecnológica nos últimos anos. O grande interesse no desenvolvimento deste tipo de software se deve ao seu atrativo comercial, cujo mercado mundial movimenta dezenas de bilhões de dólares. Assim, empresas das áreas de computação e entretenimento estão fazendo pesados investimentos no desenvolvimento de técnicas computacionais sofisticadas, as quais podem ser empregadas em outros programas interativos, principalmente naqueles que envolvam interfaces gráficas complexas, o que significa o uso conjunto de várias mídias, animações com gráficos 2D e 3D, vídeos, som, etc, bem como ambientes multiusuário baseados em Internet. Como o interesse por conteúdo na Internet tem aumentado acentuadamente, é razoável se supor que também cresça o interesse por formas mais sofisticadas de apresentá-lo, o que tornam bastante atrativas as técnicas de implementação dos jogos por computador. O objetivo é apresentar os principais conceitos envolvidos na área de jogos por computador.

Palavras chaves: Jogos por Computador, Telefones celulares, Motores de jogos.

Abstract: In the last years, computer games have shown an impressive technological evolution. The great interest in the development of this kind of software is its commercial profit. Great companies in the computer and entertainment market are investing a lot of resources in the development of sophisticated computer techniques that can be applied in the development of other interactive programs, specially the programs that require complex graphics interfaces. This means the simultaneous use of many medias as 2D and 3D graphics animations, videos, sound, 3D sound, etc, and multiusers environments based in the Internet. As the interest for Internet content increases dramatically, it makes sense to consider that will increase the interest for sophisticated ways to show it. In this case, games implementation techniques will be very attractive. The goal of this paper is to emphasize the market, technological and scientific potential of the computer games area by a presentation of the main concepts involved in its implementation.

Keywords: Computer Games, Cellular phones, Game motors.

[1] Departamento de Computação - UFSCar, Via Washington Luiz, Km. 235, Cx. Postal 676, 13565-905 - São Carlos – SP. andre@dc.ufscar.br

[2] ICAD/IGames - Departamento de Informática – PUC-Rio, Rua Marquês de São Vicente 225, 22453-900, Rio de Janeiro – RJ {bruno;dilza;esteban;lauro}@inf.puc-rio.br

[3] ICAD/IGames - Departamento de Engenharia Mecânica – PUC-Rio dreux@mec.puc-rio.br

[4] Centro de Informática - UFPE, Cx. Postal 7851, 50732-970 - Recife – PE {cacp;glr}@cin.ufpe.br

1 Introdução

Este curso, organizado por três instituições diferentes, objetiva proporcionar uma visão geral sobre jogos em computadores e celulares. A primeira parte (item 2) apresenta um panorama da evolução dos jogos, as suas componentes principais e alguns conceitos básicos de animação gráfica, vídeo e som passíveis de utilização na implementação de um jogo. A segunda parte (itens 3, 4, 5 e 6) do curso enfoca duas importantes ferramentas atuais para o desenvolvimento de interfaces gráficas dos jogos, analisa o conceito de motores 3D, bem como tece considerações sobre jogos em rede. A última parte do curso (item 7) enfoca o desenvolvimento de um motor de jogo para celulares usando a plataforma J2ME (*Java 2 Platform – Micro Edition*).

2 Fundamentos de Jogos por Computador

Jogos por computador são atualmente um produto que dispensa apresentações. A maioria dos usuários de computadores já gastou várias horas se divertindo com algum jogo deste tipo. Se o produto é bem conhecido, o mesmo não se pode dizer de diversas características suas, tais como, componentes, técnicas de implementação, importância tecnológica e comercial, etc. O capítulo 2 enfoca alguns destes conceitos em concordância com os outros capítulos.

2.1 Tipos de jogos por computador

O tipo de um jogo define algumas de suas características de interface e de motor, com implicações claras nas técnicas de implementação. A descrição dos principais tipos é mencionada a seguir [1,2,3].

Para melhor definir os tipos de jogos é importante salientar que, dependendo do tipo, a interface define uma forma particular de atuação do usuário no universo do jogo. Uma possibilidade é o usuário atuar como uma *terceira pessoa* (o usuário se vê na cena), com uma visão descrita na literatura de *God's Eye* (olhar de Deus) ou onipotência, usada geralmente em narrações por ser normalmente uma visão superior de todo o sistema. Por exemplo, em jogos de lutas, o usuário se vê no jogo como um lutador. Outra possibilidade é o usuário atuar em primeira pessoa (a cena exibida representa o que os olhos do usuário vêem), como em alguns jogos de aventura em que o usuário caminha por túneis atirando em inimigos.

2.1.1 Tipos de jogos por computador em função da Interface

Quando a interface é em *terceira pessoa*, o personagem que representa o usuário na cena é denominado de ator. Quando é em primeira pessoa, o personagem é denominado de *avatar*. O ator é parte integrante da cena e deve ser bem caracterizado, o que exige um visual sofisticado e bem elaborado. O *avatar* simplesmente marca o usuário na cena e a sua sofisticação deve ir até o limite em que não prejudique a ilusão de imersão do usuário no jogo [4]. A interface também pode variar em termos de projeção gráfica. Quando o jogo é feito sobre um grande mapa, é usual se utilizar projeções planares ortográficas (planta, vista lateral, vista frontal e axonométrica isométrica) [5]. Estas projeções são aquelas que

compõem a descrição de um objeto em uma folha de desenho técnico tradicional. Projeções perspectivas são utilizadas quando a noção de profundidade é requerida, como, por exemplo, quando um personagem de um jogo entra atirando em um túnel.

2.1.2 Tipos de jogos por computador em função do Objetivo

Segundo o objetivo, os jogos podem ser de vários tipos, embora a tendência hoje seja combinar características de cada tipo para propor jogos diferentes. Os principais tipos são os seguintes:

- Estratégia: jogos idealizados com o objetivo do usuário gerencia recursos a fim de conquistar objetivos usando estratégias e táticas;
- Simuladores: jogos que requerem reflexo do usuário que está imerso em um ambiente que tenta retratar a realidade;
- Aventura: jogos que combinam ações baseadas em raciocínio e reflexo onde o jogador deve ultrapassar estágios que envolvam a solução de enigmas para chegar ao final;
- Infantil: jogos que enfocam quebra-cabeças educativos ou histórias simples com o objetivo de divertir a criança;
- Passatempo: jogos simples, como quebra-cabeças rápidos e sem nenhuma história relacionada, cujo objetivo essencial é atingir uma pontuação alta;
- RPG (Role Playing Game): versão computadorizada do RPG convencional, onde o jogador, assumindo uma personalidade, enfrenta situações para ganhar experiência;
- Esporte: jogos de habilidade em esportes populares, como os jogos de futebol, futebol americano, vôlei, basquete, boxe, basebol, etc.;
- Educação/Treinamento: Jogos com fins educacionais que podem envolver as características de qualquer um dos jogos anteriores.

2.1.3 Tipos de jogos por computador em função do Número de Usuários

Os jogos por computador, em sua maioria, são monousuários. Aqueles que suportam múltiplos jogadores, normalmente o fazem em pequeno número, por questões de limitações tecnológicas nas áreas de *software* e *hardware* (modelagem e sintetização de ambientes realistas em tempo-real) e de redes de comunicação (o estado dos objetos no ambiente do jogo muda no decorrer do jogo, e esta mudança tem que ser refletida, em tempo-real, no ambiente compartilhado por todos os jogadores, o que nem sempre é possível satisfatoriamente quando os jogadores estão separados geograficamente e interligados através de redes de comunicação de longa distância) [1,6]. No início dos anos 80, com a disponibilidade das redes de comunicação em escala crescente, uma variação dos jogos centrados em gráficos 2D que fizeram sucesso são os MUDs *(Multi-User Dungeons)*, ambientes imaginários cujos contextos variam desde a resolução de quebra-cabeças até roteiros complexos de aventura e violência. O maior fator de atração dos MUDs, ao contrário dos jogos tradicionais, é a interação entre vários jogadores [7].

Alguns jogos multiusuários em rede que contam com interfaces gráficas sofisticadas já estão sendo produzidos, como, por exemplo, o *Asheron's Call* da *Turbine Entertainment*

Software, cujo projeto custou milhões de dólares e é o primeiro jogo da geração dos *Massively Multiplayer (MMP) Games*. Considerações adicionais sobre jogos em rede são descritas no capítulo 6.

2.2 Definição de um jogo por computador (JC)

Um JC pode ser definido como um sistema composto de três partes básicas: **enredo**, **motor** e **interface interativa**. O sucesso de um jogo está associado à combinação perfeita destes componentes [1,3].

O enredo define o tema, a trama, o(s) objetivo(s) do jogo, o qual através de uma série de passos o usuário deve se esforçar para atingir. A definição da trama não envolve só criatividade e pesquisa sobre o assunto, mas também a interação com pedagogos, psicólogos e especialistas no assunto a ser enfocado pelo jogo.

A interface interativa controla a comunicação entre o motor e o usuário, reportando graficamente um novo estado do jogo. O desenvolvimento da interface envolve aspectos artísticos, cognitivos e técnicos. O valor artístico de uma interface está na capacidade que ela tem de valorizar a apresentação do jogo, atraindo usuários e aumentando a sua satisfação ao jogar. O aspecto cognitivo está relacionado à correta interpretação da informação gráfica pelo usuário. Note-se que em termos de jogos educacionais, a interface deverá obedecer a critérios pedagógicos.O aspecto técnico envolve performance, portatilidade e a complexidade dos elementos gráficos.

O motor do jogo é o seu sistema de controle, o mecanismo que controla a reação do jogo em função de uma ação do usuário. A implementação do motor envolve diversos aspectos computacionais, tais como, a escolha apropriada da linguagem de programação em função de sua facilidade de uso e portatilidade, o desenvolvimento de algoritmos específicos, o tipo de interface com o usuário, etc.

2.3 Técnicas de implementação da interface gráfica de um jogo por computador

A interface gráfica de um jogo é fundamental para aumentar o seu realismo e o nível de engajamento do usuário ao seu ambiente. Além disso, o atrativo visual desta interface, baseado na sofisticação artística e beleza das imagens, é um fator de grande importância para o sucesso de um jogo. Adicionalmente, a interface deve também considerar outros fatores importantes como a facilidade de interação, a rapidez de resposta, a incorporação ou não de vídeo, trilhas sonoras dependentes da cena, som 3D, etc. Em termos de implementação, deve se considerar também o ambiente gráfico a ser utilizado. Parte destas características serão abordadas nos itens subseqüentes.

2.3.1 Animação, sprites e renderização de polígonos

Jogos por computador têm como um dos seus principais atrativos a interatividade, o que requer a movimentação de personagens e objetos. O termo animação, no contexto da computação gráfica e dos jogos por computador, se refere a qualquer alteração em uma cena em função do tempo. As características passíveis de modificação, neste caso, são: posição,

forma, cor, iluminação, material, transparência, etc [1,3,8,9]. Entre as diversas técnicas de animação, três se destacam em ambiente computacional: quadro-a-quadro, *sprite* e renderização de polígonos.

A animação quadro-a-quadro segue o mesmo método da exibição cinematográfica onde a idéia de movimento é alcançada pela exibição seqüencial de quadros a uma determinada taxa, usualmente, 24 quadros por segundo. Esta animação é geralmente pré-definida e não-interativa, o que, mais a exibição em seqüência fixa dos quadros, limitam a utilização desta técnica em jogos por computador. Assim, as técnicas de *sprites* e de renderização de polígonos são as de maior interesse de jogos por computador.

Os implementadores dos primeiros jogos, em função da limitação dos recursos computacionais, tiveram que elaborar técnicas de animações interativas compatíveis com o nível destes recursos. Uma destas técnicas é a de *sprites*, um objeto gráfico bidimensional que se move sobre a tela, sem deixar marcas sobre a área que passa, como se fosse um espírito. Note-se que a palavra em inglês *sprite* é derivada da palavra em latim *spiritus*, espírito, e significa duende, fada, espírito, alma ou fantasma [9].

Um *sprite* é composto de uma seqüência de imagens (texturas), usualmente retangulares, que definem a movimentação de um determinado elemento. Em função desta movimentação, as imagens são substituídas na tela. Para permitir que não só elementos retangulares sejam exibidos através de um *sprite*, as imagens, normalmente, contém áreas que devem ficar transparentes durante a sua exibição. Esta operação pode ser implementada via cores transparentes ou aplicação de máscaras. No segundo caso, associa-se uma máscara ao *sprite*. Para se sobrepor à imagem sobre a tela, inicialmente, se faz um AND entre a máscara e a área do fundo onde ele será colocado, depois uma operação XOR entre esta área do fundo e o *sprite*. Para se recompor o fundo basta se armazenar em memória a área em questão e recuperá-la posteriormente. O uso de *sprites* sobrepostos ao mapa de bits da tela de fundo, usualmente denominado de *tile*, permite implementar uma animação rápida do objeto. Ao *sprite* se associa altura, largura, posição, estado de visibilidade, prioridade sobre outros *sprites*, transformações de escala, translação e rotação, controle de colisão com outros *sprites*, manipulação de tabela de cores, etc. Os *sprites* podem ser 2D (movimentação em x e y) ou 3D (movimentação em x, y e z).

Em termos de interface, os jogos podem ser 2D, 2.5D e 3D. Os jogos 2D e 2.5D fazem uso de *sprites* e/ou texturas e os jogos 3D renderizam em tempo real os personagens e cenários. Jogos 2D se caracterizam essencialmente por utilizar mapas de bits, através do uso preferencial de *sprites* e de técnicas relacionadas, como *double-buffering* e *scroll*. Jogos 2.5D, ou *sprite* 3D, fazem uso extensivo de técnicas computacionalmente simples para simular uma cena 3D sem renderizações custosas. Esta técnica se divide em 2 classes: *sprite*-planares e *geo-sprites*. A primeira faz uso de texturas aplicadas a objetos 3D simples, como planos. Para compreender o seu funcionamento pode-se tomar como exemplo um homem que caminha na direção do ponteiro das horas de um relógio. Se uma textura mapeada sobre um plano inidica o homem caminhando na direção das 2 horas, basta espelhar verticalmente o plano e a textura para se indicar o homem caminhando na direção das 10 horas. A técnica de *geo-sprites* é mais sofisticada. Neste caso, os elementos do jogo são descritos em termos

de polígonos, mas ao invés de serem renderizados, eles são preenchidos com texturas pré-definidas e armazenadas em memória. Para economizar espaço em memória, tenta-se definir personagens simétricos, para que se armazene somente metade das texturas associadas ao personagem [10,11,12].

Jogos 3D têm vários objetos constituídos de polígonos ou malhas de triângulos passíveis de renderização. Este tipo de técnica permite um aspecto visual e uma movimentação mais natural dos personagens e dos cenários, no entanto, é extremamente custosa em termos computacionais. A melhor solução para contornar este problema é aumentar o poder computacional do hardware.

2.3.2 Técnicas importantes de modelagem de objetos em computação gráfica

A definição de todas as técnicas de modelagem gráfica exigiria um artigo a parte. Assim, a idéia deste item é apenas fornecer uma noção de como esta modelagem é realizada em ferramentas gráficas do tipo 3D Studio Max da Kinetix [13,14], Maya da Alias|Wavefront, etc. Inicialmente, o primeiro conceito que se deve ter é o de projeções planares, dado que a tela do computador pode ser considerada o plano de projeção de um objeto 3D e, assim, o usual, é modelar este objeto visualizando-o de diferentes pontos de vista. Neste caso, as projeções mais importantes de serem conhecidas são as projeções ortográficas, os raios projetores partem do infinito e incidem de forma perpendicular ao plano, e as projeções perspectivas, os raios projetores partem de um ponto no finito. A vantagem das projeções ortográficas é que elas mantêm as proporções de comprimento, o que facilita alterações no objeto a ser modelado. As projeções perspectivas, por outro lado, possibilitam uma visão mais natural do objeto por manterem a noção de profundidade [15].

Uma forma simples de se modelar um objeto é a combinação de objetos 3D básicos. A maioria dos editores gráficos permite a geração de paralelogramos, esferas, pirâmides, cilindros, etc. Operações especiais podem ser aplicadas sobre estes sólidos, do tipo unir ou subtrair um do outro, ou então calcular a intersecção de ambos. Utilizando estas operações é possível se gerar um objeto 3D complexo. A operação de *loft* permite a geração de um objeto 3D através da definição de formas 2D que são associadas a um caminho. A extrusão funciona de forma similar ao *loft*, permitindo que após o desenho de uma forma 2D, um texto, por exemplo, ela seja replicada ao longo do eixo vertical, formando, por exemplo, um texto 3D. A revolução permite que uma forma 2D seja rotacionada em torno de um eixo para a geração de um sólido. Outros métodos de geração de sólidos são disponíveis, a escolha de um deles é influenciada por parâmetros do tipo: qualidade visual do objeto gerado, complexidade de uso da ferramenta, flexibilidade para permitir modificações e ajustes, etc [13,14].

Os objetos são gerados usualmente na representação de arame, que é uma forma com menor custo computacional e serve apenas como um guia para o processo de geração. Finalizada a geração de um ou mais objetos de uma cena, pode-se passar para o processo de renderização quando ao objeto se associam as características que vão torná-lo similar a um objeto real, tais como, cores, sombras, rugosidade, brilho, etc. O termo renderização vem da palavra inglesa *rendering*, que se origina na palavra francesa *rendre*, que por sua vez, se

presume originar da palavra *rendere* do Latim popular. Dentre os muitos significados de *rendering* em Inglês, um é o de "aplicação de uma camada de plástico ou cimento", operação esta normalmente utilizada em construção civil. Assim, *rendering* significa o processo de associação de paredes, usualmente pré-fabricadas no Estados Unidos, à estrutura levantada de uma casa, o que é similar ao processo, em computação gráfica, de preenchimento da estrutura de arame dos objetos de uma cena. Para que o processo de renderização se torne mais real, o usuário pode associar fontes de iluminação aos objetos de uma cena. A iluminação pode ser programada para usar um ou vários tipos de fontes de luz [13,14]. Note-se que as luzes de feixe cônico ou cilíndrico podem ser configuradas para ter seus feixes apontados para determinadas posições, as quais não se alteram mesmo que a fonte de luz se mova.

Implementadores de jogos fazem usualmente um balanço entre a qualidade visual do jogo e o seu desempenho. Técnicas de diminuição do número de polígonos de um objeto do jogo permitem que ele seja processado mais rapidamente pelas funções gráficas e, conseqüentemente, exibido mais rápido. Estas técnicas podem se basear em ações tomadas diretamente sobre o objeto pelo seu criador ou então pelo uso de programas especiais, que diminuim de forma automática [16,17]. As duas técnicas podem trazer bons resultados. O uso repetido de operações booleanas para a obtenção de um sólido complexo implica, normalmente, na geração de muitos polígonos adicionais. Em um processo conhecido como *decimation*, programas específicos eliminam estes polígonos através de cálculos matemáticos que se baseiam em algumas heurísticas. A principal delas é a de que uma região de pouca curvatura, formada por superfícies quase coplanares, precisa de menos polígonos para ser representada do que uma região com grande curvatura. Esta questão é crítica, pois no caso do personagem humano de um jogo, a região do seu cotovelo apresenta uma alta curvatura e, se houver eliminação de muitos polígonos nesta região, o movimento do braço do personagem ficará pouco natural. A renderização de um sólido se dá através da coloração dos polígonos que o compõem. Esta coloração pode ser feita de diferentes formas, seguindo métodos que usualmente tomam como base a normal ao polígono. Quanto maior o número de polígonos, maior a suavidade do objeto na renderização, o que relega ao modelador o balanceamento entre a qualidade visual e o número de polígonos. *Tesselation* é o nome do processo de se adicionar mais polígonos à representação gráfica de um objeto.

2.3.3 Técnicas gerais de animação

O processo de animação tradicional nos estúdios de produção de desenhos animados estabelecia que o animador mestre desenhava o personagem em posições chaves (quadros $\underline{n}$ e $n+i$) e os desenhistas auxiliares os quadros intermediários (quadros de $n+1$ ao $\underline{n+i-1}$). Este método é denominado de *keyframe*, ou quadro-chave. Atualmente, os aplicativos gráficos fazem o papel dos desenhistas intermediários, o usuário é o animador mestre, ele estabelece os quadros-chave e o aplicativo se encarrega de gerar os quadros intermediários. De um quadro-chave para outro, o usuário praticamente pode alterar qualquer parâmetro do desenho: posição, ângulo, cor, iluminação, material, etc.

Apesar da potencialidade desta técnica, há determinadas aplicações que requerem uma forma diferente de produção da animação. Por exemplo, animar uma bailarina dançando requer que o usuário seja capaz de conhecer os passos da dança ao longo do tempo. Se o usuário criar de forma pouco criteriosa os quadros-chave da bailarina, é bem provável que a animação associe movimentos artificiais à bailarina. Para se corrigir este problema, dois métodos são passíveis de uso: *rotoscoping* e *motion-capture*. A técnica de *rotoscoping*, no contexto da animação gráfica, utiliza os quadros de um vídeo como fundo, ou textura, para a produção de uma animação [18]. Assim, considerando-se novamente o caso da bailarina, pode-se filmá-la dançando, importar o vídeo e produzir a animação considerando os seus movimentos para cada intervalo de quadros determinado. Captura de movimento *(Motion Capture)* [19,20], é a técnica utilizada para capturar, através de equipamentos especiais, os movimentos de objetos reais e mapeá-los a objetos sintetizados no computador. Esta técnica é geralmente utilizada para capturar movimentos humanos, como, por exemplo, o de um lutador de artes marciais. Opta-se por esta técnica quando se exige maior naturalidade ou suavidade no movimento do personagem ou quando o movimento apresenta dificuldades para ser gerado por outras técnicas. Normalmente, os dispositivos para realizar esta captura são sensores eletromagnéticos sem fio que são acoplados a diferentes partes do corpo de um ator. Outros equipamentos de captura de movimentos incluem câmeras, *flashes* infravermelhos de alta luminosidade, sincronizadores, marcadores reflexivos passivos, etc.

O uso de uma técnica de animação mais sofisticada exige um tratamento mais apurado do personagem. Por exemplo, considerando-se que o movimento capturado vai ser associado ao esqueleto do personagem para lhe permitir uma movimentação suave e natural, então o seu corpo deve refletir estas características. Neste caso, por exemplo, o corpo do personagem deve ter as junções de seu corpo (joelho, cotovelo, etc) modeladas com uma quantidade maior de polígonos.

Outro cuidado que se deve ter com esta técnica se refere à coerência inicial das seqüências de movimento. Se um espadachim parte de uma posição inicial para começar uma luta, então todas as tomadas realizadas devem considerar sempre este ponto de partida. Isto pode dificultar as tomadas, porque é difícil de se exigir um comportamento tão preciso de um ator humano. Note-se que uma tomada para captura de movimentos difere das tomadas cinematográficas, pois esta última apresenta uma seqüência linear ditada por um roteiro.

2.3.4 Técnicas de animação de faces

Associar falas e emoções às faces dos personagens de um jogo é um problema de crucial importância porque dela também depende, em grande parte, a naturalidade dos personagens. Animar faces é um processo que segue usualmente os seguintes passos: a) a construção da face, b) a síntese da imagem facial e c) a animação da face [21].

Não há uma única maneira para construção de um rosto. O processo mais comum divide a construção em 3 partes: 1) criar a boca pelo método radial, 2) derivar outras superfícies a partir da boca e 3) criar e inserir os olhos e o nariz. O método radial deriva seu nome do primeiro passo de criação da boca, que é a geração de uma curva NURB 2D representando o lábio e a cavidade da boca. A curva é utilizada para, através do processo de revolução, gerar

uma malha de aspecto igual ao de um vaso, a qual é modelada para atingir a forma de uma boca. Após a criação da boca, cria-se a superfície do rosto, ou, mais especificamente, a superfície das bochechas. A boca é concatenada à superfície criada, o que gera uma única malha. O nariz e os olhos são modelados a parte e depois fundidos à malha da boca. Um tratamento especial deve ser dado a esta fusão, de forma a remover rugosidades na face. A malha da cabeça é gerada pelo mesmo processo de modelagem da face. Posteriormente, a malha da face é fundida à da cabeça. Note-se que pode ser necessário repetir a criação da cabeça para cada expressão facial utilizada na animação.

Gerar expressões faciais complexas e reais de uma face 3D através da modelagem de malhas é uma tarefa complexa. Um método que pode simplificar essa tarefa é a combinação de modelos de faces 3D com texturas de cada expressão facial, as quais são geralmente projetadas de forma cilíndrica sobre a face 3D. A associação de falas à face é usualmente um processo complexo, cujo problema principal é a dificuldade de sincronização do movimento da boca e da face com a voz. Outros problemas são: a) a preservação do realismo da fala e da expressão em modelos com número menor de polígonos, como muitos utilizados em jogos por computador e b) o ajuste da animação total a um determinado tempo. A associação de falas a face pode ser realizada por um dos seguintes métodos: 1) pesquisa, 2) combinação de fonemas, 3) captura facial 2D e 4) captura facial 3D. O método da pesquisa, baseado em análise de todos os movimentos dos músculos do rosto humano, requer um trabalho extenso de uma numerosa equipe de artistas. Este método foi utilizado pela Pacific Data Images para a animação do filme *Antz* da Dream Works.

Fonema é a menor unidade fonética significativa para uma língua, por exemplo, *r* e *rr* em português. O método da combinação de fonemas associa a cada fonema um padrão de malha de polígonos. Existem vários métodos e ferramentas para a realização deste processo, mas, em geral, todos seguem a mesma idéia. O processo envolve a criação de expressões para os fonemas e a associação de cada fonema a um canal controlado por uma barra deslizante. A posição de cada barra determina a percentagem de contribuição de seu fonema à expressão total. Cada canal atualiza apenas os vértices dos polígonos da face que são associados ao seu fonema, o que permite a combinação de um franzir de sobrancelhas com um bocejo. Este método possibilita uma boa interatividade, no entanto, pode apresentar problemas no controle geral do tempo da animação.

O método de captura facial 2D é amplamente utilizado. A idéia básica é capturar movimentos faciais do rosto de ator de um ponto de vista frontal. A captura dos movimentos pode ser feita por marcadores presos ao rosto do ator, os quais são rastreados por um dispositivo montado sobre a sua cabeça. Neste caso, há o problema da dificuldade da colocação dos marcadores em determinadas partes do rosto, como, os olhos, por exemplo. O método de captura facial 3D retorna a maioria das informações sobre o movimento da face, o que inclui movimentos de cabeça e quase todas as expressões faciais. A captura é realizada por equipamentos especiais que permitem o escaneamento 3D do ator, os quais retornam uma grande quantidade de dados que requerem a análise e a filtragem adequada. Um problema deste tipo de método é a dificuldade de se manipular a geometria da face após a

aplicação do movimento a ela. Para evitar este problema, pesquisadores trabalham em um método alternativo que combina este método com o de fonemas.

2.3.5 Inserção de vídeos em jogos

Vídeo é um dos elementos dos jogos por computador que tem apresentado cada vez maior uso. No entanto, muitas vezes, os implementadores de jogos não sabem como utiliza-lo da melhor forma. Note-se que quando bem usado, o vídeo pode aumentar o prazer e a compreensão da trama do jogo, reforçar uma narrativa, introduzir um personagem, etc. Quando mal usado, ele pode matar o jogo [22]. Os vídeos imersos em ambientes interativos foram denominados inicialmente de vídeos interativos. Atualmente, a nomenclatura adotada é FMV (*Full-Motion Video*), significando vídeo com movimentação total.

A história de uso de vídeos em jogos começa em 1983 com *Dragon's Lair*, o primeiro a usar vídeo analógico. *Sherlock Holmes: Consulting Detective*, liberado em 1992, foi o primeiro a usar vídeo digital. O primeiro jogo com vídeo de sucesso foi *The 7th Guest*, da Trilobyte, liberado em 1993. O mais popular de todos foi *Myst*, da Cyan, liberado para PC em 1994 e que utiliza adequadamente o vídeo, através de pequenas inserções que se casam com o contexto geral do ambiente do jogo. Na seqüência, vieram os bons jogos *Phantasmagoria*, 1996, e *Wing Commander III*. Uma regra importante para a utilização de vídeo em jogos é o da naturalidade da transição, ou seja, a mudança de vídeo para a animação gráfica e vice-versa deve estar totalmente de acordo com o contexto do jogo, possibilitando que, em alguns casos, o usuário nem a perceba. O uso de um vídeo longo e maçante no começo do jogo, ou então, de vídeos curtos e desconexos da trama no começo de cada nível não é um procedimento que vai de encontro a esta regra. Além do encaixe perfeito com a trama e a interface do jogo, a inserção de vídeo em jogos também deve obedecer algumas regras técnicas de produção. Estas regras mais aquelas necessárias para a utilização da técnica do cromaqui são citadas abaixo. O cromaqui é uma técnica que consiste em filmar uma cena em que o cenário de fundo é totalmente pintado de verde ou azul, o qual depois será removido e substituído por uma imagem, animação gráfica ou outro vídeo. Tradicionalmente, a técnica utilizava equipamentos ópticos especiais para a eliminação do fundo. Atualmente, placas gráficas permitem a captura do vídeo por um computador e a sua edição através de programas específicos que possibilitam também a realização do cromaqui.

A incorporação de vídeo aos jogos deve se acentuar rapidamente em função de três fatos: o aumento expressivo da capacidade de armazenamento das mídias (CD – 640 Mbytes, DVD – 9 Gbytes), sofisticação das técnicas de compressão (vide os esforços na definição do padrão MPEG-4) e a diminuição dos preços dos equipamentos para a produção de vídeos.

2.3.6 Formatos de arquivos de som

Sons são um fator extremamente importante para se aumentar a emoção ao jogar. Em jogos, o áudio é composto basicamente de dois elementos: a) música e b) sons diversos (buzina, vozes, ruídos, etc). A música é usualmente armazenada em arquivos MIDI e os sons diversos são armazenados como áudio digital em arquivos WAV. Os principais formatos de armazenamento de som de interesse dos desenvolvedores de jogos são citados abaixo.

WAV: O Microsoft *Windows Riff Wave* é atualmente a base do áudio digital, sendo largamente utilizado em efeitos sonoros. Várias ferramentas manipulam este formato e permitem a geração dos mais variados tipos de efeitos. Dependendo da qualidade utilizada para um arquivo Wav, ele pode requerer uma taxa de transmissão de 40 MBytes por minuto, o que o torna muito pesado para operar em computadores de pequeno porte.

MIDI: A interface MIDI (*Musical Instrument Digital Interface*) define uma linguagem de transmissão de dados digital entre sistemas computacionais, sintetizadores e instrumentos musicais. O MIDI trata informações musicais, tais como, o tipo da nota, a sua velocidade, timbre, etc. Ele não é áudio digital, portanto não permite a gravação de vozes ou efeitos sonoros. O MIDI permite que seus dados possam ser mudados facilmente de forma manual (direta) ou via programa. O MIDI ocupa pouco espaço em disco, por exemplo, 1 minuto de som com só uma trilha e sem nenhum evento a ele ligado requer 3Kb por minuto, o que comparado com o formato Wav e muitos outros é muito pequeno.

MP3: O formato MP3, *Moving Pictures Experts Group Audio Layer 3 compressed audio*, objetiva armazenar som compactado. A sua definição é baseada em um modelo psico-acústico, ou seja, o ouvido humano não é capaz de ouvir todas as freqüências, assim, um algoritmo elimina grande parte das freqüências que ele possa não escutar. Note-se que esta compactação é destrutiva. A qualidade do som MP3 é igual ao de CD e 12 vezes menor que a de um arquivo igual em formato Wav.

2.3.7 Áudio Interativo e 3D

Áudio interativo consiste na modificação em tempo real do som para que este pareça mais real em uma certa circunstância [23]. Como usualmente as aplicações apresentam limitações de espaço, de hardware e de software, e requerem flexibilidade, se opta pela utilização do formato MIDI. Neste caso, através de ferramentas especiais é feita a conversão de áudio digital em MIDI. O áudio digital é utilizado prioritariamente em aberturas ou cenas sem interatividade, dado que os formatos para este tipo de som são pesados e pouco flexíveis. No Wav, por exemplo, o processo de adicionar efeitos e modificar o som é demorado e complexo. Uma alternativa para tornar o áudio digital um pouco mais interativo é a gravação de música ou efeitos sonoros em pequenos trechos e das mais variadas formas, incluindo mudanças de tempo e velocidade, freqüência, modulações e efeitos como *reverb*, *wah-wah*, envelope, etc.

Em função destes problemas e com as otimizações no formato MIDI, ele se tornou a opção mais atrativa para o áudio interativo. Isto, tanto pelo tamanho dos arquivos, quanto pela facilidade de modificações e alterações. Com a utilização de ferramentas pode-se fazer modificações radicais em um MIDI sem muita demora e esforço. Por exemplo, tendo-se um MIDI de alguma música do Beethoven em piano, pode-se, dentre os vários efeitos, modificar o volume da trilha, realizar *pan* (translação) entre vários trechos, modificar o estilo do som pré-definido de piano para guitarra em algum trecho, etc. Devido à facilidade de manipulação do MIDI, foram criadas linguagens de programação simples para se alterá-lo. Neste caso, por exemplo, se em um jogo, o personagem está para entrar em uma situação de combate, antes o som ambiente estava calmo, no momento em que os outros personagens

aparecem o som muda, fica mais pesado, mais rápido. Isto é facilmente programável em termos de recursos MIDI. Muitos profissionais da área de jogos afirmam que o futuro dos jogos está no áudio interativo. Com melhorias em hardware e software, os processos disponíveis para o MIDI poderão estar disponíveis para o áudio digital, o que acentuará a qualidade deste tipo de som.

A idéia do áudio 3D acompanha as tendências 3D de jogos e outros aplicativos atuais. No entanto, diferentemente da imagem, o som 3D é mais complicado de ser manipulado e requer uma visão correta do que seja.

Áudio 3D não é a manipulação do som entre esquerda/direita ou frente/trás. Usualmente se confunde este conceito com o efeito *pan* no som estéreo. Áudio 3D é a possibilidade de se colocar o som em qualquer lugar em torno da cabeça do ouvinte. Para alcançar este efeito, o ideal é o ouvinte estar equipado com um fone de ouvido porque a posição do ouvido está bem definida em relação aos ouvidos. Outra possibilidade é se utilizar som quadrifônico ou *surround*, com as caixas em posições definidas [24]. Se o som vier de duas caixas à frente do ouvinte, fica difícil de gerar som 3D.

Aperfeiçoamentos têm ocorrido nesta área, mas ainda existem barreiras a serem transpostas, como, por exemplo, o áudio 3D depende da posição da cabeça do ouvinte em relação às caixas, logo para ouvintes diferentes podem ocorrer erros de posicionamento. O ideal neste caso seria o usuário utilizar um fone de ouvido. Para criar um ambiente com áudio 3D, deve-se observar as seguintes características da interação do som com o ambiente: reverberação, distância do ouvinte e da fonte do som, efeito Doppler, objetos no caminho entre o ouvinte e a fonte do som e a absorção do ar.

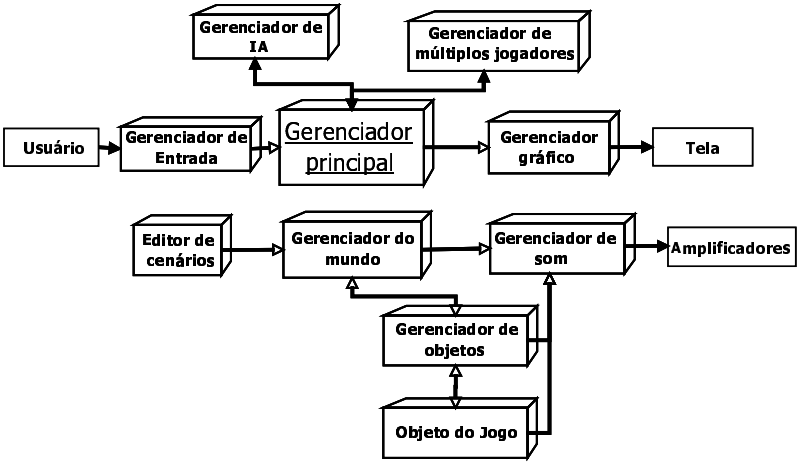

Figura 1: Arquitetura Genérica de um Jogo por Computador

2.4 O motor de um jogo

Apesar de haver uma concepção genérica da arquitetura do motor de um jogo (Figura 1) [25], a formalização de seus componentes ainda é um campo em aberto, como várias concepções na área de jogos [26]. Esta falta de formalização prejudica o transporte de jogos para diferentes ambientes, bem como o uso de módulos específicos. Esforços têm sido realizados no sentido de definir uma arquitetura padrão, bem como associar aos motores técnicas de engenharia de software, de forma a garantir níveis satisfatórios de performance, de robustez, de modularidade, de reusabilidade de código, de padronização, etc [2,27,28,29].

Módulos da arquitetura do motor de um jogo	
Gerenciador de Entrada	Encarregado de identificar eventos dos dispositivos de entrada e encaminhar o ocorrido para outro módulo que irá processar o dado e realizar uma ação.
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 se comuniquem entre si. Para tanto, deve gerenciar a conexão e a troca de informações entre os diversos computadores que estão 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.
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 a 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 a 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.

2.4.1 Ferramentas de autoria em jogos

Usualmente, as técnicas envolvidas na implementação de um motor de um jogo incluem: estruturação e classificação de dados, mecanismos de comunicação, métodos sofisticados de controle dos personagens e do mundo, de detecção de colisão [30], de sincronização, de animações gráficas, de imagens e som, etc. Como é usualmente complicado se lidar com todas estas técnicas, é usual se partir para a utilização de alguma ferramenta de autoria para o desenvolvimento do jogo. No contexto dos jogos, ferramentas de autoria são programas que fornecem o máximo de recursos para facilitar o trabalho de desenvolvimento de um jogo, possibilitando economia de esforço e tempo. Estas ferramentas são elementos essenciais de projeto.

Os implementadores do jogo *Hyperblade*, da Activision, consideraram que era necessário disponibilizar um ambiente de criação de jogos que reduzisse a quantidade de código a ser escrito e que fosse acessível para não-programadores. O sistema de desenvolvimento ADLIB (*Authoring and Design Language for Interactive Behavior*) foi criado com esta finalidade. O ADLIB é constituído de três partes: 1) uma linguagem para descrever o comportamento e a interação dos personagens, 2) um ambiente de criação e 3) um sistema de tempo de execução [31].

A necessidade de acesso a ferramentas gratuitas, para múltiplas plataformas, com código aberto e amplo, deu origem a algumas iniciativas para a definição e implementação de ferramentas de autoria para jogos. Uma destas iniciativas originou o *Allegro*, sigla recursiva para *Allegro Low Level Game Routines*, que é uma biblioteca que disponibiliza aos programadores C/C++ as rotinas de baixo nível que são normalmente utilizadas na implementação de jogos, tais como, as rotinas para o controle de entrada de dados, de tempo das animações, de som MIDI, e de geração de gráficos e de efeitos de som. A biblioteca começou a ser desenvolvida por Shawn Hargreaves e, atualmente, o trabalho conta com a ajuda de programadores de todo o mundo [32,33]. Alguns dos recursos da *Allegro* incluem: facilidade de uso, capacidade de ser extensível, portatilidade, ter código aberto e ser gratuita.

Uma das bibliotecas *Allegro* disponíveis é a *Allegro-GL*, que permite o uso das suas rotinas em conjunto com as da biblioteca gráfica *OpenGL*. A *jaw3d* é outra biblioteca disponível que permite uma programação 3D ampla e pode ser usada para criar aplicações 3D simples em praticamente qualquer sistema operacional.

Atualmente, devido à acirrada concorrência entre as indústrias de jogos, o que exige a geração de bons produtos no menor tempo possível, os implementadores começaram a utilizar ferramentas denominadas *frameworks*, uma estrutura pré-implementada de um motor de jogo. Várias destas ferramentas estão disponíveis atualmente no mercado, entre elas pode-se citar: Fly3D [34], o Genesis3D [35], o Crystal Space [36], o Golgotha [37], etc. Algumas destas ferramentas são caras, no entanto, a maioria é gratuita. Muitas destas ferramentas apresentam uma série de deficiências, tais como, baixa performance, falta de documentação, código mal escrito, etc, no entanto, esforços estão sendo feitos para incorporar técnicas de engenharia de software em seu desenvolvimento, de forma a torna-las atrativas para o implementador de jogos [44]. Este assunto é abordado mais extensamente no item 5.

Muitos dos motores de jogos 3D são construídos com base na biblioteca gráfica OpenGL [38], por ela ser poderosa, relativamente fácil de se programar, portátil, eficiente e muitas placas gráficas permitirem aceleração por hardware para as suas funções. Como a maioria dos jogos tem como alvo a plataforma Windows, a OpenGL é utilizada de forma integrada com os módulos do DirectX [39], o qual é analisado no capítulo 4. Neste caso, a OpenGL compete com o Direct3D [40], item 4.5, a biblioteca 3D do DirectX, em termos de uso. Note-se que jogos podem ser implementados utilizando Direct3D em modo *Immediate* (simplificado) ou *Retained* (completo) [41].

3 Java 3D

O uso de Java 3D como opção para desenvolvimento de jogos 3D sempre traz à tona a comparação de Java com C/C++. Java é uma linguagem de mais alto nível do que C/C++, visto que não usa ponteiros explícitos, tem *garbage collector* e é rigorosamente orientada a objetos. Java é, sem dúvida, uma linguagem mais moderna, pois já nasceu com a visão de rede, objetos distribuídos, ambientes de janelas e interação baseada em eventos. Além do mais, Java foi criada com o conceito "write once, run anywhere", ou seja: é multiplataforma e muito mais fácil de se programar do que outras linguagens sem esta característica. Este conceito multiplataforma também se estende ao JavaBeans que, em comparação com os componentes ActiveX, funciona em todas as plataformas e vem automaticamente com um gerente de segurança. Entretanto, quanto ao desempenho, Java ainda é uma linguagem lenta, apesar da possibilidade de uso dos compiladores *just-in-time*. A maior cautela que se deve ter com a opção por Java é quanto ao fato de ser um trabalho ainda em desenvolvimento, com muitas classes e métodos sendo freqüentemente criados e outros sendo descontinuados.

A API (Application Programming Interface) Java é dividida em Java Base API (linguagem núcleo, utility, I/O, rede, GUI e suporte a serviços) e Java Standard Extension API que estende a funcionalidade básica de Java (interfaces para comércio eletrônico, segurança, acesso a bancos de dados, telefonia, servidor Java e Java Media Framework). Um dos componentes do Java Media Framework é a API Java 3D que tanto pode ser usada para criar aplicações 3D isoladas como *applets* 3D para a Web [42].

A indústria estabeleceu APIs gráficas padrão que rodam nos principais hardwares gráficos. Entre as várias bibliotecas gráficas padrão, as mais populares são OpenGL e

Direct3D. A API Java 3D é uma camada de alto nível sobre OpenGL (Unix e Windows), Direct3D (Windows 98, ME e 2000) e Mesa (Linux), conforme ilustra a Figura 2.

O Java 3D permite um maior nível de abstração, podendo em certos casos aliviar o programador da preocupação com otimização de *rendering* ou da necessidade de criação de código para construção de polígonos e vértices. No caso de jogos, são especialmente importantes as classes para áudio (inclusive som 3D) e para construção de animação pré-fabricada (chamada *behavior*). Os aspectos mais importantes do Java 3D para jogos são o seu carater multiplataforma e as facilidades de programação e documentação. Estes aspectos têm papel decisivo no *time-to-market* da criação de um jogo.

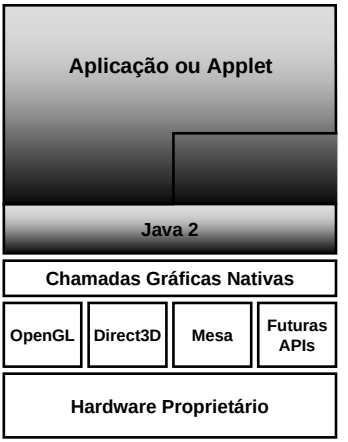

Figura 2 Interfaces com Java 3D

Ao se comparar Java 3D com outras APIs, observa-se que as APIs “antigas” (OpenGL, DirectX, ou engines de jogos 3D de mais baixo nível) requerem controle detalhado. Em contraste, as APIs “novas” são focadas em controle de alto nível, tais como Java 3D, VRML, SGI OpenInventor, Optimizer/Cosmo3D (sendo descontinuada) e SGI-Microsoft "Fahrenheit".

O Java 3D pode ser obtido gratuitamente em http://www.sun.com, como um pacote para Java 2 disponível para a maioria das plataformas. No caso de ambientes Windows pode-se escolher entre dois tipos de bindings (OpenGL ou DirectX). A compilação e a execução de um aplicativo utilizando a API Java 3D são feitas como nos programas Java usuais.

3.1 Modelo de Programação

O modelo de programação do Java 3D é o de Grafo de Cena (Scene Graph) fortemente inspirado no VRML (Figura 3). Um grafo de cena é uma estrutura hierárquica em forma de árvore que contém uma descrição completa da cena, incluindo modelos de dados, atributos e informação de visualização.

Um grafo de cena é criado usando-se instâncias de classes Java 3D. O primeiro procedimento do programador é a criação do Universo Virtual e a determinação de um sistema de coordenadas (uma instância da classe *Locale*). Ao objeto *Locale* são acopladas vários subgrafos cujas raízes definem grupos chamados de *Branch Groups* (**BG**). Existem duas categorias de grupo: visualização (*View*) e conteúdo (*Content*). O subgrafo de visualização (*View Branch Graph*) especifica os parâmetros de ambiente de visualização e os parâmetros físicos (p.ex. a posição do observador). O programador pode optar por uma versão simplificada e fixa de um universo que engloba o *Locale* e o *View Branch Graph*, chamada de Universo Simples (*Simple Universe*). O subgrafo de conteúdo (*Content Branch Graph*) contém os objetos visuais e de aúdio, tais como geometria, aparência, comportamento, som, luzes, … . Uma trajetória (*Scene Graph Path*) define completamente o estado de uma folha (*leaf*). O Java 3D faz o *rendering* das folhas na ordem que for mais eficiente (o programador não tem que se preocupar com o gerenciamento do processo de *rendering*).

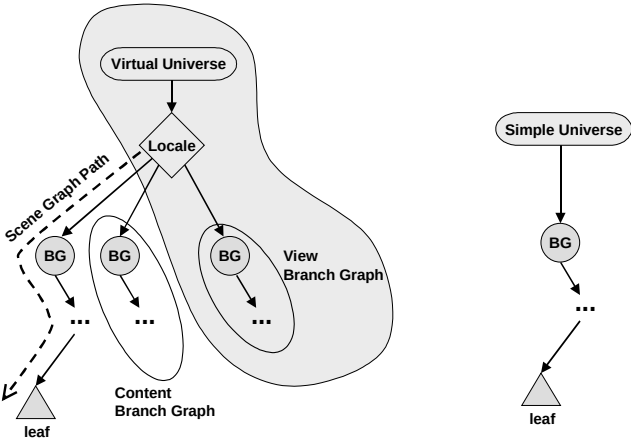

Figura 3 Grafo de Cena

Uma cena complexa é geralmente composta por vários subgrafos de conteúdo ligados ao mesmo *Locale* que podem ser trocados ou ativados de acordo com as ações do usuário. Por exemplo, os vários níveis de um jogo podem ser implementados como **BG**s que são desativados a medida que o jogador conquista novos níveis.

3.2 Exemplo

O exemplo da Figura 4 pode ser encontrado no site do Grupo de Jogos do ICAD (http://www.icad.puc-rio.br/igames). Este exemplo consiste numa hipotética viagem pelo espaço, visando utilizar uma boa parte da API Java 3D.

Neste exemplo, a animação dos objetos e câmera visa explorar os comportamentos Java (*Java behaviors*), um recurso de alto nível que permite animações pré-fabricadas

(rotação, translação ou trajetórias). Além das rotações dos objetos e translação automática do observador, é possível combinar os botões do mouse e a tecla ALT para reposicionar a câmera e experimentar novos ângulos.

Os corpos celestes permitem explorar o mapeamento de texturas e sua combinação (BLEND/REPLACE) com o modelo de iluminação empregado.

A falsa "sombra" do eclipse permite explorar a biblioteca de primitivas básicas (esferas e cilindros), bem como o uso de material transparente e fosco. A idéia da “sombra” também visa o uso de transformações locais para "soldar" o movimento da lua ao seu cone (na realidade, um cilindro) de sombra.

Figura 4 Exemplo Java 3D

A disposição das luzes permite a exploração de volumes de influência e tipos de luzes (iluminação solar e iluminação da terra pela passagem da nave espacial). O fundo de estrelas obviamente, estimula a exploração da API para carregamento de *background*.

O som de fundo e o som posicional (i.e. aquele que aumenta com a proximidade da câmera ao planeta terra) explora os recursos de áudio da API. Por fim, a nave espacial permite explorar o recurso de alto nível de carregamento de modelo em formato comercial (Alias/Wavefront).

4 DirectX

4.1 Introdução

O DirectX foi projetado para ser uma ferramenta completa para desenvolvimento de aplicações que exijam visualização em tempo real, tanto 3D como 2D, procurando principalmente beneficiar-se dos recursos oferecidos pelo hardware da máquina (placa gráfica, placa de som, etc). Uma das suas principais deficiências está no fato de que é voltado apenas para plataforma Microsoft Windows. Não existe, no entanto, nenhuma API tão

completa como esta. O OpenGL, por exemplo, é apenas uma API para 3D, não dando nenhum suporte a rede, som ou dispositivos de entrada.

Cada tipo de hardware possui sua própria "linguagem" de acesso aos seus recursos. Assim, programar uma aplicação que procure utilizar estes benefícios, seria uma tarefa árdua, pelo fato de que um programa deveria ser devidamente traduzido para cada tipo de placa. Este será outro benefício que se espera de uma API: criar uma camada que abstraia a linguagem específica de cada hardware, permitindo que um mesmo programa funcione corretamente, independente do hardware e da plataforma. No caso dos recursos gráficos, o DirectX disponibiliza para este fim do **HAL** - Hardware Abstraction Layer, permitindo desenvolver um jogo sem estar preocupado com o hardware que estará disponível ao usuário final.

Atualmente existem no mercado uma variedade grande de placas gráficas. As placas mais complexas dispõem de mais recursos de aceleração do que as mais baratas. Ao desenvolver um programa, não é possível prever qual será o conjunto de recursos que estarão presentes, já que é impossível saber qual será o hardware do usuário final. Desta forma o DirectX dispõe do **HEL** - Hardware Emulation Layer - que possui implementação por software de todos os possíveis recursos gráficos, de maneira que, se uma aplicação precisar de uma delas e esta não estiver disponível na placa gráfica, a API irá emular este recurso. Isto permitirá rodar uma aplicação que necessite de recursos gráficos até num usuário que não possui nenhum hardware dedicado, embora o desempenho possa ficar muito baixo.

O DirectX está baseado na tecnologia de **componentes**, que consiste numa extensão dos conceitos relacionados a programação por objetos: Um componente é responsável por uma série de recursos, que serão acessados pela aplicação via as **interfaces**. As interfaces serão como que um elo de ligação entre a aplicação e o componente. Uma vez criada a interface, dispõem-se de uma série de métodos que estão como que encapsulados por ela. Estes métodos somente poderão ser acessados quando a interface já foi inicializada. Para disponibilizar de métodos de um mesmo componente que estão em interfaces diferentes, deve-se realizar um empilhamento dos mesmos, o que será feito pela função QueryInterface.

Ao criar um componente, na verdade diz-se que se está criando uma **instância** de um componente. Em muitos casos, apenas se poderá ter uma única instância de um componente, que será o responsável por todos os recursos da aplicação que a está utilizando. Um componente ou uma interface possui um **GUID** (*Global Unique Identifiers*) único que o identifica. Um GUID para componente receberá o nome de **CLSID** e um GUID para interface será chamado de **IID**.

Todos os métodos retornam uma estrutura do tipo **HRESULT**, que indica se a operação foi bem sucedida ou não. O valor retornado não é apenas verdadeiro ou falso, mas também indica que tipo de falha houve. Existem várias palavras reservadas já definidas e associadas a diversos tipos de erros que podem ocorrer, facilitando com isto a tarefa de DEBUG. Para simplesmente testar se a chamada foi bem sucedida ou não, pode-se utilizar algumas das macros disponíveis: **SUCCEEDED** ( ) ou **FAILED** ( ). Estas irão desprezar

todas as características do erro ou do sucesso e será a forma mais comum para testar a validade de um método.

Assim sendo, é comum que a chamada de um método numa aplicação tenha a seguinte forma:

```
if (FAILED ( método )) {      /* Tratamento da falha */  }
else {      /* Método foi bem sucedido...*/  }
```

O DirectX é dividido em módulos diferentes, cada um destinado a finalidades específicas nas diversas etapas de desenvolvimento de uma aplicação: DirectPlay (para recursos de distribuição em rede), DirectInput (para ler entrada de dados de teclado, mouse, joystick e diversos dispositivos), DirectAudio (para acessar recursos da placa de som), DirectShow (para trabalhar com recursos de sequências de vídeo) e o DirectGraphics (para acessar ou emular recursos gráficos disponíveis na placa gráfica). A seguir se apresentará uma breve introdução a alguns destes módulos.

4.2 DirectShow

Este componente foi incluído no DirectX a partir da versão 8.0. Até então era chamado de DirectX Media e era um componente à parte. Disponibiliza recursos para manipular sequências de vídeo, nos mais diversos padrões atuais (AVI, MPEG, MOV, etc). Exemplos de aplicações que podem ser desenvolvidas utilizando o DirectShow são DVD Players, aplicações para edição de vídeo, conversores de padrões, aplicações para captura de vídeo, MP3 Players, etc. No desenvolvimento de um jogo será essencial para mostrar trechos de vídeo ou sequências já renderizadas.

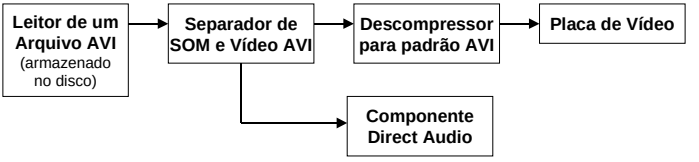

Figura 5 Diagrama de *Filters* para uma aplicação simples

Para se criar uma aplicação no DirectShow existe um tipo de componente fundamental, chamado *Filter*. Este componente possui sempre uma entrada e produz uma saída, sendo capaz de realizar uma operação sobre uma sequência de vídeo, tal como ler um arquivo, repassá-lo para a placa de vídeo ou para a placa de som, decodificar um padrão de compressão, tal como o MPEG, etc. Assim, uma aplicação consistirá numa sequência de vários *Filters*, onde a saída de um será conectada à entrada de outro. A Figura 5 ilustra uma aplicação capaz de mostrar um AVI e cujo código será descrito em seguida. O DirectShow disponibiliza um filtro especial, chamado *Filter Graph Manager*, capaz de gerenciar todos os filtros de uma aplicação, permitindo que o programador não precise manipular cada *filter* individualmente.

Para modo a implementar o exemplo da Figura 5, será necessário criar 3 interfaces: IGraphBuilder (para construir o *Filter Graph*), IMediaControl (controle dos vídeos no *Filter Graph*) e IMediaEvent (controle de eventos do *Filter Graph*). Todas estas interfaces estarão implementadas no *Filter Graph.*

Assim, o primeiro passo para a implementação consiste em criar um *Filter Graph* e seguir, cria-se uma instância do *Filter Graph Manager*, que será apontado por pGraph:

```
qIGraphBuilder *FilterGraph;
CoInitialize(NULL); // Apenas para inicializar a library dos componentes
CoCreateInstance (CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER, IID_IGraphBuilder, (void **)&pGraph);
```

Para acrescentar as outras duas interfaces que serão necessárias, deve-se realizar um empilhamento dos recursos (Query) utilizando o pGraph:

```
IMediaControl *pMediaControl;
IMediaEvent *pEvent;
pGraph -> QueryInterface (IID_IMediaControl, (void **)&pMediaControl);
pGraph -> QueryInterface (IID_IMediaEvent, (void **)&pEvent);
```

Por último, deve-se construir o *Filter Graph*, e abrir um vídeo e/ou som para ser tocado. Isto é feito pelo método RenderFile, tendo como parâmetro o nome do arquivo que será utilizado. O método Run irá colocar o *Filter Graph* em ação, tocando o arquivo recém aberto e o método WaitForCompletion ditará o tempo, em milisegundos, em que o *Filter Graph* irá estar funcionando, de forma a mostrar o vídeo. Colocando-se INFINITE como parâmetro, garante-se que nada interromperá o playback da sequência de vídeo:

```
pGraph->RenderFile ("C:\\Nome_do_Video.avi", NULL);
pMediaControl -> Run();
pEvent->WaitForCompletion(INFINITE, &evCode);
```

4.3 DirectAudio

O DirectAudio surgiu na versão 8.0 do DirectX, como resultado da união do DirectMusic e do DirectSound, que até a versão anterior eram componentes separados.

Esta API permite manipular sequências de áudio, podendo tirar as vantagens disponíveis dos recursos do hardware de som, colocar vários canais de som tocando simultaneamente, criar efeitos de som 3D, trabalhar com captura Midi e wav a partir de canais de entrada, criar efeitos com filtros, etc. Estes recursos, quando integrados numa aplicação 3D são fundamentais para um jogo rico em recursos multimídia.

As etapas para se criar uma aplicação com o DirectAudio podem ser divididas nas seguintes:

1. Inicialização do componente.

```
CoInitialize(NULL);
```

2. Criar e inicializar o **performance**, que é o objeto responsável por controlar o fluxo de dados do arquivo fonte de áudio para o **DLS (Douwnloadable Sound) Sinthesizer**. O DLS tem a função de converter todos os dados que ainda não estão no formato wav, uma vez que todos os áudios, antes de serem enviados para a placa de som, devem estar neste padrão. O performance se encarregará do timing, encaminhamento de mensagens, do controle de múltiplos canais e uma série de outras tarefas fundamentais. Raramente uma aplicação precisará de mais de um performance. Para acessar seus métodos, deve-se inicializar a interface IDirectMusicPerformance8. A inicialização será feita através do método InitAudio, que opcionalmente também poderá setar um path default para o áudio:

```
IDirectMusicPerformance8* g_pPerformance = NULL;

CoCreateInstance (CLSID_DirectMusicPerformance, NULL, CLSCTX_INPROC,
    IID_IDirectMusicPerformance8, (void**)&g_pPerformance );

// Para uma aplicação simples pode-se utilizar os valores Defaults

// dos parâmetros do método InitAudio.

G_pPerformance->InitAudio (NULL, NULL, NULL,
    DMUS_APATH_SHARED_STEREOPLUSREVERB, 64,
    DMUS_AUDIOF_ALL,  NULL );
```

3. Criar um objeto **segmento**, que são objetos capazes de encapsular sequências de áudio. Um simples midi ou wav pode ser considerado um segmento, que neste caso encapsula-se a si mesmo. Entretanto um segmento pode vir a conter várias trilhas diferentes e para isto deverá ser um arquivo do tipo sgt. Um segmento pode ser primário ou secundário. Apenas um segmento primário pode ser tocado por vez, sendo que este irá corresponder ao canal de áudio principal. Os segmentos secundários serão tocados sobre o primário e normalmente serão pequenos trechos de músicas ou efeitos sonoros a serem sobrepostos.

```
IDirectMusicSegment8* g_pSegment = NULL;
```

4. Criar o **loader**, que é um objeto capaz de carregar um arquivo e colocá-lo num segmento. Cada aplicação necessita apenas de um loader, uma vez que o mesmo pode ser utilizado para carregar diversos arquivos. Isto será feito pela inicialização da interface IDirectMusicLoader8.

```
IDirectMusicLoader8* g_pLoader = NULL;
```

```
CoCreateInstance (CLSID_DirectMusicLoader, NULL, CLSCTX_INPROC,
                  IID_IDirectMusicLoader8, (void**)&g_pLoader);
```

O método LoadObjectFromFile irá carregar um arquivo a ser tocado no segmento criado anteriormente:

```
g_pLoader->LoadObjectFromFile(CLSID_DirectMusicSegment,
IID_IDirectMusicSegment8, "C:\\audio.mid", (LPVOID*) &g_pSegment )));
```

5. Converter o segmento para tipo wav, enviado-o para o DLS Sinthesizer, pelo método Download( ).

```
g_pSegment -> Download( g_pPerformance );
```

6. Tocar o segmento, através do método PlaySegmentEx( ).

```
g_pPerformance->PlaySegmentEx ( g_pSegment, 0, 0, 0, 0, 0, 0, NULL, NULL );
MessageBox ( NULL, "Pressione OK para sair.", "Play Audio", MB_OK );
```

4.4 DirectGraphics

Até a versão 7.0 do DirectX existiam 2 módulos distintos e separados para o tratamento 2D e 3D de uma aplicação: o DirectDraw e o Direct3D, respectivamente. A partir da versão 8, ambos os módulos se fundiram num único, chamado DirectGraphics, com a proposta de tornar o ambiente de desenvolvimento mais integrado. Apesar desta fusão, grande parte das interfaces, de suas funcionalidades e de seus métodos continuam funcionando de maneira equivalente às já existentes em cada módulo. Sua função principal é fornecer acesso aos recursos de aceleração gráfica do hardware e permitir um acesso direto à memória da placa de vídeo.

Esta área de memória é chamada pelo DirectX de **surface** e pode ser de 2 tipos:

Front buffer: Corresponde ao que será mostrado no monitor: tudo o que for colocado nesta região será automaticamente exibido no monitor. Por razões lógicas, apenas pode haver uma região primária;

Off-screen ou **back buffer**: Serão regiões da memória que não serão mostradas no monitor. A princípio o DirectX procura criar estas regiões dentro da memória da placa de vídeo, mas também poderá utilizar a memória RAM, caso não haja mais espaço disponível. Esta superfície pode vir a ser nomeada, em algum momento da aplicação, como uma superfície primária. Realizar esta troca corresponde ao **Bliting** ou **Page Fliping.** Uma aplicação pode simultaneamente ir calculando e preenchendo o back buffer, enquanto mostra algo no front buffer.

4.5 Direct3D

De forma a expor o funcionamento deste componente, serão mostradas as diversas etapas que se devem seguir para criar um aplicativo básico. No caso, o programa será um visualizador de objetos 3D, capaz de ler arquivos no formato .x, que é o formato padrão do

DirectX para malhas poligonais, podendo ter opcionalmente materiais e texturas associadas. Este formato de arquivo é, em geral, suportado pela grande maioria dos programas de modelagem 3D, especialmente aqueles que se propõem a servir como ferramentas para modelagem de jogos 3D. Toda aplicação que irá utilizar os recursos do Direct3D deve começar pela inicialização do objeto Direct3D. Isto será feito mediante a função Direct3DCreate8, que irá retornar um ponteiro para a interface do Direct3D, que contém as diversas funcionalidades da API.

```
LPDIRECT3D8 Objeto_D3D = NULL;
Objeto_D3D = Direct3DCreate8 (D3D_SDK_VERSION);
```

Antes de inicializar-se uma aplicação é necessário criar um Device que seja adequado aos recursos gráficos que estarão disponíveis. Um **Direct3D Device** é o componente que tem a função de armazenar e processar as etapas do rendering do programa. Existem 3 tipos de devices diferentes:

HAL Device (Hardware Emulation Layer): Suporta aceleração por hardware para as funções de rasterização e aceleração por hardware e/ou software para o processamento de vértices. Este device deve ser escolhido sempre que o sistema possua uma placa aceleradora gráfica, de forma a obter a maior performance possível.

Reference Device: Implementa todos os recursos gráficos através de software, de forma a ter precisão nas operações. Este tipo é mais conveniente apenas para testar aplicações.

Pluggable Software Device: Implementa todas as funções do HAL via software, embora tente otimizar o processamento através de instruções disponíveis no processador.

Um device funciona como uma camada que está abaixo do componente do Direct3D e lhe oferece os recursos gráficos permitidos pela placa ou possíveis de serem emulados. A

Figura 6 mostra como é sua arquitetura. Para se criar um device, deve-se inicialmente preencher a estrutura descrita por D3DPRESENT_PARAMETERS, que indica o tipo de apresentação na tela do vídeo (full-screen, resolução, espaço para buffer secundário, etc):

```
LPDIRECT3DDEVICE8 D3D_Device = NULL;
D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory ( &d3dpp, sizeof(d3dpp) );
d3dpp.Windowed = TRUE;
d3dpp.SwapEffect = D3DSWAPEFFECT_COPY_VSYNC;
```

Feito isto, pode-se criar o device. No código abaixo inicializa-se um HAL Device. (Numa aplicação mais detalhada deveria-se realizar antes um teste para ver qual o device mais adequado):

```
Objeto_D3D ->CreateDevice ( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
hWnd,D3DCREATE_SOFTWARE_VERTEXPROCESSING, &d3dpp,&D3D_Device) )
```

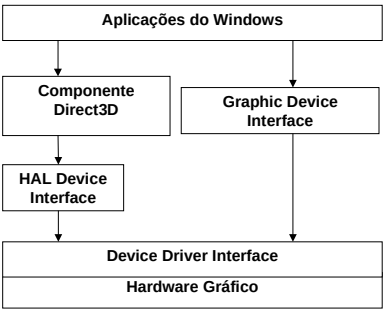

Figura 6 Devices do Direct3D

Uma vez ativado o Device, uma aplicação já pode utilizar os recursos gráficos. Na maioria dos casos, uma aplicação para Windows funciona baseada no sistema de monitoração de mensagens empilhadas ao sistema. A visualização pelo device ocorrerá quando for empilhada a mensagem WM_PAINT, que diz para uma aplicação redesenhar toda a janela ou parte dela. O tratamento das mensagens, tendo em vista o rendering de uma cena, será da seguinte forma:

```
LRESULT WINAPI MsgProc ( HWND hWnd, UINT msg, WPARAM wParam,
                                        LPARAM lParam ){
    switch( msg ){
        case WM_DESTROY:
            PostQuitMessage( 0 );
            return 0;
        case WM_PAINT:
            Render();
                ValidateRect( hWnd, NULL );
            return 0;
      }
    return DefWindowProc( hWnd, msg, wParam, lParam );
}
```

A função Render ( ) deve ser composta pelas seguintes etapas:

1. Limpar toda a janela da aplicação ou uma parte dela. Isto será feito pelo método Clear, pertencente à interface do Direct3D device:

```
g_pd3dDevice ->Clear ( 0, NULL, D3DCLEAR_TARGET,
    D3DCOLOR_XRGB(0,0,255),  1.0f, 0 );
```

A combinação do primeiro parâmetro valendo 0 e o segundo NULL fará com que seja limpada toda a tela. O quarto parâmetro pinta a área da janela de azul.

2. Iniciar a cena, através do método BeginScene( ). Todos os métodos de rendering devem estar entre a chamada deste e do método EndScene( ):

```
g_pd3dDevice  -> BeginScene();
```

3. Processar o rendering da cena.

4. Encerrar a cena, pelo método EndScene( ):

```
g_pd3dDevice -> EndScene();
```

5. Mostrar o resultado do rendering, pelo método Present( ):

```
g_pd3dDevice -> Present ( NULL, NULL, NULL, NULL );
```

Para se visualizar uma cena, deve-se ter alguma geometria sendo descrita. Esta pode ser definida no próprio programa, criando-se uma lista dos vértices, faces, normais, materiais e texturas ou pode ser obtida a partir de um arquivo contendo uma cena modelada por um outro programa. A leitura de um arquivo será feita pelo método D3DXLoadMeshFromX, no instante da inicialização da cena, colocando o resultado num buffer irá conter temporariamente todos os materiais e texturas do objeto a serem abertos no programa. Este número normalmente corresponde ao total de elementos que compõem o objeto 3D em questão:

```
LPD3DXBUFFER pD3DXMtrlBuffer;
D3DXLoadMeshFromX ( "sibgrapi01.x", D3DXMESH_SYSTEMMEM,
    D3D_Device, NULL, &pD3DXMtrlBuffer, &g_dwNumMateriais,&g_pMesh ) ) ;
```

Ainda nesta etapa deve-se extrair todas as propriedades do material e o nome dos arquivos das texturas. Isto será feito inicialmente pela aquisição do ponteiro para o buffer onde se encontram os materiais e as texturas:

```
D3DXMATERIAL* d3dxMateriais = (D3DXMATERIAL*)pD3DXMtrlBuffer->
        GetBufferPointer();
```

Posteriormente deve-se criar objetos que suportem os materiais e as texturas. Deve haver um destes objetos para cada um dos materiais e texturas, sendo utilizado para isto o valor retornado pelo parâmetro dwNumMaterials. Todos estes objetos de suporte serão criados na forma de um vetor:

```
g_pMeshMateriais = new D3DMATERIAL8[g_dwNumMateriais];
g_pMeshTexturas = new LPDIRECT3DTEXTURE8[g_dwNumMateriais];
```

Para cada um dos objetos deve-se copiar as propriedades do material e opcionalmente associar um valor para a componente ambiente do mesmo.

Também será necessário criar a textura para o material:

```
For (i = 0; i < g_dwNumMateriais]; i++)
```

```
{
        g_pMeshMateriais[i] = d3dxMateriais[i].MatD3D;
        g_pMeshMateriais[i].Ambient = COR_AMBIENTE;
        if (FAILED (D3DXCreateTextureFromFile ( g_pd3dDevice,
                d3dxMateriais[i].pTextureFilename, &g_pMeshTexturas[i] ) ) )
                g_pMeshTexturas[i] = NULL;
}
```

Criados os objetos para os diversos materiais, pode-se liberar o buffer temporário:

```
pD3DXMtrlBuffer->Release();
```

Finalmente, para renderizar o objeto, deve-se fazer um loop que irá percorrer cada sub conjunto de elementos da cena. Como o Direct3D funciona baseado em estados de rendering, em cada iteração do loop será ativado o estado do material correspondente ao elemento que será renderizado:

```
for( int i=0; i < g_dwNumMateriais; i++ )
  {  // Ativar o estado para o material e para a textura do sub-elemento i
     g_pd3dDevice->SetMaterial ( &g_pMeshMateriais[i]);
     g_pd3dDevice->SetTexture ( 0, g_pMeshTexturas[i]);
     //  Calcula a visualização do sub-elemento i
     g_pMesh->DrawSubset( i );
  }
```

Contudo até este ponto da aplicação não foi definido ainda o ponto de vista do observador. Isto será feito através da definição das matrizes de visão e de projeção, o que deve ocorrer na etapa de rendering, antes de que qualquer objeto seja desenhado. A função D3DXMatrixLookAtLH irá criar uma matriz de visão, sendo o segundo, terceiro e quarto parâmetros respectivamente a definição da localização do observador, o alvo onde está olhando e o vetor que indica qual a sua direção vertical.

```
D3DXMATRIX matriz_Visao;
D3DXMatrixLookAtLH ( &matriz_Visao, &D3DXVECTOR3( 1.0f, 1.0f, 1.0f ),
     &D3DXVECTOR3( 0.0f, 0.0f, 0.0f ), &D3DXVECTOR3( 0.0f, 1.0f, 0.0f ) );
```

A matriz de projeção irá dizer como a cena 3D será projetada numa tela 2D. Isto será feito criando-se uma matriz de projeção através da função D3DXMatrixPerspectiveFovLH, cujos parâmetros são a matriz que será devolvida com o resultado, o ângulo de visão, o aspect ratio, o Near Plane e o Far Plane.

```
D3DXMATRIX matriz_Projecao;
D3DXMatrixPerspectiveFovLH ( &matriz_Projecao, D3DX_PI/4, 1.0f, 1.0f, 100.0f );
g_pd3dDevice->SetTransform( D3DTS_PROJECTION, &matriz_Projecao );
```

5 Motores 3D

Apesar de haver uma concepção genérica da arquitetura do motor de um jogo (ver Figura 1), a formalização de seus componentes ainda é um campo em aberto, como várias concepções na área de jogos [43]. Esta falta de formalização prejudica o transporte de jogos para diferentes ambientes, bem como o uso de módulos específicos. Esforços têm sido realizados no sentido de definir uma arquitetura padrão, bem como associar aos motores técnicas de engenharia de software, de forma a garantir níveis satisfatórios de performance, de robustez, de modularidade, de reusabilidade de código, de padronização, etc [2,44,45,46].

5.1 Classificação de Motores 3D

As APIs gráficas disponibilizam uma série de recursos para que uma aplicação possa ter um alto desempenho em relação aos recursos gráficos. Entretanto, estas ferramentas disponibilizam recursos que não prevêm soluções para aplicações específicas, já que não se propõem a solucionar um problema particular. Assim, da mesma forma que seria impratícável desenvolver um programa diretamente em assembler para Windows, seria extremamente trabalhoso criar um jogo 3D sofisticado utilizando apenas uma API, tal como o OpenGL ou o DirectX. Neste contexto, existem *frameworks* voltados para o desenvolvimento de jogos 3D, chamados de Motores 3D (*3D Engines*).

Atualmente existe um número muito grande de *engines* - mais de 600 já catalogados - sendo que cada um possui suas particularidades. A grosso modo, pode-se separar os tipos de engines em relação ao tipo de jogo que se propõem criar: jogos 3D de primeira pessoa, jogos 2D de estratégia, jogos 3D com vista de *terceira pessoa*, simuladores, jogos de corrida, etc. Muitas vezes, determinados Motores 3D são especializados em apenas um tipo de ambiente: fechado (tipo Doom) ou aberto (i.e. com modelos de terreno). O tempo de desenvolvimento de um jogo é radicalmente encurtado quando se usa o Motor 3D adequado. Os jogos mais complexos geralmente requerem o desenvolvimento de novos componentes (muitas vezes chamados de *plug-ins*) a serem incorporados no *framework* do Motor 3D. Por esta última razão é importante a escolha do Motor 3D quanto à linguagem núcleo (em geral C/C++) e a plataforma (Windows e/ou Unix).

Pode-se classificar um Motor 3D de acordo com os recursos e facilidades que oferece para o desenvolvimento. A seguir estão alguns critérios que são importantes para desenvolver um jogo 3D. Quanto melhor forem cumpridos estes requisitos, melhor será a ferramenta (e em muitos casos maior será o seu valor econômico no mercado):

- Facilitar o desenvolvimento de um tipo de jogo específico, fazendo com que uma série de operações fundamentais sejam transparentes para o programador (cálculo de colisão, projeção das texturas, leitura de dispositivos de entrada, simulações físicas, etc);
- Possibilitar o cálculo de BSPs e portais da cena;
- Realizar o cálculo de mapas de luzes, para a iluminação estática da cena;

- Permitir que o programador possa escrever uma série de funções relacionadas a ações e eventos do jogo numa linguagem mais alto nível que o C++. Em geral isto será feito através de uma linguagem script própria do Motor 3D;
- Beneficiar-se dos recursos disponíveis no hardware gráfico, utilizando numa camada mais baixa uma ou mais APIs.
- Permitir o desenvolvimento de um jogo multi-usuário, utilizando algum protocolo padrão de comunicação.
- Possibilitar que se possam construir, com facilidade, bibliotecas de objetos, funções e mapas, de forma a poder reutilizá-las com facilidade ao programar outro jogo;
- Disponibilizar ferramentas para interagir com outros programas de modelagem, e ser capaz de ler/converter formatos padrões de descrição de cenas 3D.
- Facilitar que artistas possam elaborar níveis de jogos, sem estarem preocupados com programação.

A estrutura de um engine padrão está ilustrada na Figura 7.

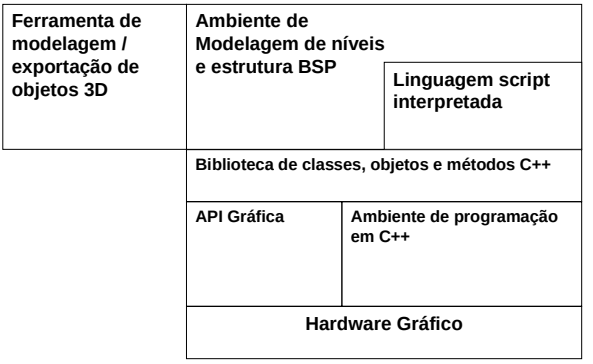

Figura 7 Estrutura de um Motor 3D padrão

O ambiente de modelagem de níveis consiste numa ferramenta que auxilia a concatenação de modelos 3D, a descrição do cenário, a definição da iluminação e a descrição geométrica e de animação dos personagens. Este ambiente pode ser desde um modelador complexo, com ferramentas de visualização e simulação de nível, até uma simples listagem do que irá compor o nível. A interação deste ambiente com o código do jogo pode ser feito através de um *script* interpretado, através de plug-ins que acessam as funções e os recursos presentes no núcleo do Motor 3D ou através de funções e métodos que facilitem os processos de cálculos padrão de jogo.

Uma linguagem *script* permite que sejam acessados elementos da cena e suas propriedades sem estar preso à sintaxe do C++ repleta de ponteiros e nomes de funções complicadas. Desta maneira, o *script* facilita a manipulação dos elementos da cena e a

criação de ações para esses elementos. Quanto maiores são as facilidades oferecidas por uma linguagem *script*, mais fácil é criar a inteligência do jogo.

Um Motor 3D deve permitir que o desenvolvedor possa programar em qualquer um destes níveis. Quanto mais ampla é esta abertura, maior é o escopo de tipo de jogos que são possíveis de serem desenvolvidos com esta ferramenta.

5.2 Alguns Motores 3D

Alguns dos motores interessantes de se experimentar são os seguintes: Crystal Space, Genesis3D, 3D Game Studio e Fly3D.

O Crystal Space é *open source*, o que reduz as consequências de possíveis desativações no futuro (um fenômeno relativamente comum no mercado dinâmico de jogos). Uma boa vantagem do Crystal Space é ser escrito em C++ multiplataforma (Windows, Linux, etc.), com uma boa modelagem de classes. As falhas deste motor podem ser sanadas por implementações dos próprios usuários.

O Genesys3D também é *open source*, mas só roda em Windows. Uma desvantagem marcante deste motor é ser escrito em C, isto é, não há uma biblioteca de classes.

O 3D Game Studio é um produto inteiramente comercial para plataforma Windows, cuja principal vantagem é a eficiência aliada a uma boa interface. Pode-se usar o Game Studio diretamente em C++ ou através de sua linguagem *script* compilada.

O Fly3D é um excelente produto nacional com vários graus de acesso público, comercial e acadêmico (a versão 1.0 é integralmente *open souce*), criado pelos autores do livro Games 3D [43]. O Fly3D é escrito em C++ e só roda em Windows.

6 Jogos em Rede

Em um jogo distribuído em rede, cada estação participante apresenta uma vista do espaço virtual. Assim, o estado do ambiente é compartilhado por todas as estações. Porém, para que os diversos jogadores tenham a ilusão de estar vivenciando o mesmo local, o ambiente virtual distribuído deve apresentar um bom nível de consistência, ou seja, as estações participantes devem exibir animações com o maior grau de semelhança possível. Naturalmente, para levar a efeito as mudanças de estado do ambiente nas diversas estações, é necessário que mensagens de atualização de estado sejam transmitidas na rede.

A consistência é total ou absoluta quando todas as estações participantes apresentam exatamente as mesmas seqüências de mudanças de estado. Seria inútil exigir que todas as estações apresentassem as mesmas cenas nos mesmos instantes pois os retardos de rede impossibilitam tal comportamento. É através das técnicas de gerenciamento de estado compartilhado que a consistência do ambiente distribuído é controlada. Técnicas diferentes alcançam graus diferentes de consistência. De um modo geral, técnicas que garantem maior consistência exigem um número maior de mensagens transmitidas na rede.

Além da consistência, os jogos distribuídos em rede – principalmente os distribuídos na Internet – devem apresentar escalabilidade, isto é, capacidade de incluir novos participantes sem degradar o sistema. Um dos fatores que determinam a escalabilidade é o número de mensagens adicionais transmitidas na rede por cada nova estação participante. Para favorecer a escalabilidade é necessário minimizar as mensagens na rede. Portanto, aumentar a consistência tende a prejudicar a escalabilidade.

Um jogo distribuído é visualmente correto [47] se cada um dos usuários não é capaz de distinguir, nas cenas apresentadas na tela, os componentes virtuais controlados pela estação local dos componentes controlados por estações remotas. Para que o usuário não seja capaz de diferenciar os componentes locais dos remotos, todos devem se apresentar com movimentos suaves e interagindo de forma precisa. Um exemplo de interação imprecisa seria um avatar colocar a mão em posição não compatível com a de um outro avatar na execução de um aperto de mãos. Freqüentemente, "pulos" e interações imprecisas ocorrem quando o estado de um componente remoto é atualizado. Existe então uma relação inversa entre consistência e corretude visual.

O projetista de um jogo distribuído em rede deve buscar um equilíbrio entre consistência, escalabilidade e corretude visual de forma a produzir um sistema de qualidade. A busca de tal equilíbrio parte da escolha da arquitetura de comunicação do jogo distribuído. No que se refere à arquitetura de comunicação, os dois principais tipos de jogos são: "client-server games" e "peer-to-peer games". Nos jogos do tipo "client-server", as estações participantes enviam mensagens umas às outras através de um ou mais servidores. Naturalmente, os servidores aumentam o tempo de latência e tendem a se tornar "bottlenecks". Porém, eles possibilitam um gerenciamento mais simples do sistema – base de dados que descreve o ambiente, jogadores, etc. – e ainda podem otimizar a transmissão de mensagens, comprimindo múltiplos pacotes em um só, eliminando mensagens redundantes e gerenciando a taxa de transmissão de acordo com a capacidade da estação cliente. Servidores também podem realizar tarefas administrativas como contabilizar o tempo de participação de cada jogador. Uma outra vantagem é que o *software* no cliente se torna mais leve. Atualmente a maioria dos jogos distribuídos na Internet são "client-server games" porque, além da facilidade de administração, a comunicação com estações que utilizam conexão via modem exige otimizações. Nos "peer-to-peer games", estações se comunicam diretamente umas com as outras, eliminando os "bottlenecks" e diminuindo o tempo de latência das mensagens transmitidas. Este tipo de arquitetura é o mais recomendado para jogos que necessitam de maior escalabilidade.

7 Jogos para Dispositivos Móveis

Tradicionalmente, dispositivos como celulares e PDAs vem de fábrica com alguns programas instalados que são desenvolvidos sob a coordenação do fabricante do dispositivo. Essa situação inviabiliza o desenvolvimento de novos programas pela comunidade interessada, uma vez que a arquitetura interna de software e hardware dos dispositivos não é de domínio público. Visando contornar essa situação, diversos fabricantes têm procurado formas de permitir que aplicações sejam desenvolvidas por qualquer um interessado e ao

mesmo tempo não revelar os seus segredos industriais. A solução para esse problema baseia-se na criação da linguagem de programação Java 2 Micro Edition (J2ME) [48,49] e da inclusão de uma máquina virtual Java nos dispositivos.

Dentre as aplicações para dispositivos móveis, jogos parecem ser os que têm maior potencial mercadológico, haja visto o que ocorre atualmente [50]. Dado este potencial, deve-se assistir nos próximos anos uma multiplicação de grupos interessados nesse nicho de aplicação baseando-se em J2ME. Entretanto, como discutido a seguir, para poder produzir jogos na qualidade e velocidade demandadas, é preciso superar algumas limitações de J2ME, assim como construir ferramentas de desenvolvimento como motores (framework) e editores de cenários.

7.1 Java 2 Micro Edition (J2ME)

Atualmente existem três edições diferentes da linguagem Java – Enterprise Edition (J2EE) [51], Standard Edition (J2SE) [52] e Micro Edition (J2ME) [53], cada uma sendo direcionada para diferentes categorias de dispositivos. A linguagem Java 2 Micro Edition (J2ME), com aproximadamente um ano de criação, é uma simplificação da linguagem Java 2 Standard Edition (J2SE) realizada através da remoção e modificação de partes fundamentais de J2SE com o objetivo de criar um ambiente de execução de programas para dispositivos com grandes restrições de memória e processamento.

7.1.1 Arquitetura de J2ME

Uma grande parte do trabalho de definição da arquitetura da nova linguagem foi dedicada a considerar que componentes da linguagem Java original deveriam ser cortados, mantidos ou modificados. Com a observação da variedade de dispositivos envolvidos, J2ME adotou uma arquitetura modular e escalável, consistindo de um conjunto de blocos que se complementam para representar as particularidades dos grupos de dispositivos existentes. J2ME define três camadas de software construídas sobre o sistema operacional presente em cada dispositivo, a saber:

- *Camada da Máquina Virtual Java (JVM),* que é uma implementação da máquina virtual Java baseada no sistema operacional presente em cada dispositivo;
- *Camada de definição de uma Configuração*, acima da anterior, define uma plataforma mínima para uma larga categoria de dispositivos;
- *Camada de definição de um Perfil (profile),* mais próxima dos usuários e desenvolvedores de aplicações, que define a API voltada para demandas específicas de um segmento de mercado, tais como, carros, celulares, televisores, etc.

Baseado nesse conceito de camadas, apenas as funcionalidades comuns a todos os dispositivos fariam parte das camadas mais centrais, enquanto que as particularidades ficariam restritas apenas aos perfis e configurações mais específicas. Naturalmente, um programa escrito para um dado perfil tem a portabilidade garantida para qualquer dispositivo que suporte tal perfil. A idéia é que isso ocorra da mesma maneira que aplicações para PCs são desenvolvidas para um determinado sistema operacional sem preocupação sobre o fabricante do computador que irá executá-lo. Atualmente existem duas configurações, uma

para dispositivos fixos com grande demanda de energia e conexão, Connected Device Configuration - CDC [54] e outra para dispositivos móveis e mais simples, Connected Limited Device Configuration - CLDC [55]. Um perfil que abrange celulares e *pagers*, Mobile Information Device Profile – MIDP [56] já está definido sobre o CLDC. Existem outros perfis em desenvolvimento.

7.1.2 Suporte ao Desenvolvimento de Jogos: Desafios

Em se tratando do uso de CLDC/MIDP para o desenvolvimento de jogos, uma observação rápida mostra a viabilidade de seu uso. A seguir seguem os pontos mais importantes que são suportados:

- *Suporte a programação orientação a objetos (OO)*, por manter os conceitos encontrados em J2SE como Classes, Interfaces, Classes Abstratas e Pacotes, uma modelagem OO pode ser considerada;
- *Suporte a manipulação da tela gráfica*, que é essencial para o desenvolvimento de jogos, já que é preciso acessar diretamente a tela para desenhar (além desse suporte, oferecido pela classe Canvas, é suportado o desenho de algumas formas geométricas e imagens no formato PNG);
- *Suporte a manipulação do teclado dos dispositivos*, sem o qual seria muito difícil desenvolver um jogo, que é uma aplicação inerentemente interativa;
- *Conectividade com outros aparelhos*, via protocolo HTTP, para desenvolvimento de jogos em rede .

Porém, existem certas limitações bastante significativas. A seguir estão as mais críticas, além das conseqüências dessas limitações:

- *Ausência de ponto flutuante*, o que obriga o programador a adaptar algoritmos já conhecidos para possibilitar o trabalho com aproximações inteiras;
- *Ausência de acesso à cor de um pixel da tela*. Tal acesso, e a alteração do conteúdo dos pixels da tela ou de uma imagem, é importante para a execução de certos algoritmos gráficos. Sem esse suporte, esses algoritmos tornam-se inaplicáveis;
- *Ausência de som*, tornando inviáveis alguns recursos e deixando os jogos monótonos;
- *Baixo poder de processamento*, em torno de 25 MHz, inviabilizando a execução de jogos mais complexos, principalmente aqueles que necessitam de processamento em tempo real ou de certos algoritmos de inteligência artificial;
- *Pouca memória*, em média 32 KB para execução de programas (memória RAM) e 128 KB para persistência de dados, prejudicando jogos que manipulam muitos objetos ou que precisam armazenar muitos dados em memória;
- *Ausência de polígonos*, dispondo apenas de primitivas gráficas básicas (elipses, retângulos e linhas).

Diante destas limitações, o desenvolvimento de jogos para dispositivos móveis impõe uma série de restrições e adaptações.

7.2 Motores

Assim como nos jogos para PCs, é necessário, para uma escala profissional, utilizar um framework (motor) de desenvolvimento de jogos assim como ferramentas de apoio. Como discutido nas seções anteriores, já existem vários motores que podem ser reutilizados no caso de jogos para PCS. No entanto, não existe ainda, à exceção do wGEM (ver seção 7.3), tal ferramenta para celulares. Em muitos casos então, será necessário desenvolver seu próprio motor, além de ferramentas associadas.

Dada a novidade desta tecnologia, não há consenso sobre o que seria um motor de jogos padrão para dispositivos móveis como os celulares. No entanto, estudando as soluções equivalentes existentes em PCs [57, 58], é possível adaptar a arquitetura proposta na Figura 1 para o caso de jogos 2D baseados em J2ME.

Na verdade, atualmente, só não é viável a implementação do gerenciador de som e do de múltiplos jogadores, por se basearem em funcionalidades ainda indisponíveis. Todos os outros componentes podem ser implementados, com as devidas simplificações devido às limitações de processamento, memória e interface dos dispositivos móveis atuais.

Portanto, além dos princípios gerais de desenvolvimento de motores, como modularidade e reusabilidade, certos cuidados devem ser observados no desenvolvimento de um motor em J2ME. Uma atenção especial deve ser dada à otimização do uso de CPU e de memória. Isto implica em dar um tratamento especial às estruturas de dados, devido ao acesso aos objetos do jogo ser uma ação de grande freqüência. Por exemplo, o uso de *hashtables* ou vetores, adequados para aplicações em PC, podem gerar um excesso de uso de memória e processamento.

Outro aspecto importante é desacoplar o máximo possível os componentes do motor, de forma que seu uso possa ser seletivo, implicando em menores programas e, consequentemente, menor espaço em memória.

Por fim, será igualmente necessário realizar as adaptações ao J2ME, em particular para a questão da ausência de aritmética de ponto flutuante e do desenho de polígonos.

7.2.1 Editores de Cenários

O uso de um motor por si só não é suficiente para garantir um rápido desenvolvimento. Diversas ferramentas de apoio geral são úteis, em particular um editor de cenários adaptado para o motor. Este editor teria o objetivo de permitir de uma forma visual e simples a definição de estágios de um jogo. Para isso, as seguintes funcionalidades devem ser fornecidas:

- *Definir visualmente o mapa do jogo*, incluindo altura, largura, textura, *tiles*, etc.*;*
- *Definir os objetos do jogo,* incluindo alguns atributos como velocidade, identidade visual, etc.;
- *Definir o cenário*, possivelmente utilizando recursos de *drag and drop,* permitindo que os objetos sejam facilmente adicionados, removidos e editados para modificação de suas propriedades;

- *Salvar e carregar cenários* desenvolvidos, para facilitar a continuidade e reutilização do trabalho;
- *Exportar os cenários* em formato compatível com o motor em uso.

7.3 wGEM

Como resultado de uma pesquisa que vem sendo realizada no Centro de Informática da UFPE desde setembro de 2000 [59], foi desenvolvido um motor em J2ME para o desenvolvimento de jogos, denominado wGEM *(Wireless Game Engine for Mobile Devices)*. O wGEM atende a todos os requisitos enumerados na seção 7.2, e conta igualmente com um editor de cenário específico.

7.3.1 Motor

As funcionalidades do motor são distribuídas entre diversos módulos e classes do motor, baseadas na arquitetura mostrada na Figura 1. O desenvolvedor de jogos basta então utilizar essas classes, seguindo a arquitetura do wGEM, e implementar a lógica particular de cada jogo, pois as tarefas mais comuns já são realizadas pelo wGEM.

Atualmente, o wGEM provê os recursos básicos para o desenvolvimento de jogos 2D. No entanto, alguns recursos, como suporte a som, rede e IA, ainda não estão presentes. O suporte a som ainda não é disponível no J2ME, já as funcionalidades de rede e de inteligência artificial podem ser realizadas, apesar de bem limitadas, e serão modeladas no futuro. Os testes realizados indicam um excelente desempenho e um uso de memória adequado.

7.3.2 Editor de Cenários

Foi desenvolvido um editor de cenário para o wGEM que dispõe de todas as funcionalidades enumeradas na seção anterior. Foi também adicionado ao wGEM um *parser* capaz de interpretar o arquivo exportado pelo editor e iniciar os objetos do jogo.

7.3.3 Jogos Desenvolvidos

Para validar o wGEM, tanto o motor quanto o editor de cenários, foram implementados dois jogos que podem ser executados no emulador J2ME produzido pela Sun [60]. O custo de execução do motor nos dois casos foi baixo. Nenhuma memória adicional é requerida durante o jogo, além daquela utilizada na inicialização dos cenários. Além disso, o desempenho foi muito satisfatório, atingindo taxas de 100 fps no emulador citado.

O primeiro jogo implementado foi o clássico BreakOut (ver Figura 8), onde o jogador tem o objetivo de destruir um conjunto de blocos utilizando uma bola que é controlada por uma raquete. O usuário deve então controlar a raquete para evitar que a bola saia do campo de jogo e ao mesmo tempo tentar dar uma direção à bola que provoque o choque com algum bloco.

Os seguintes gerenciadores de objetos foram modelados para o BreakOut:

- Gerenciador da bola. Responsável por identificar colisão da bola com a raquete, os tijolos e a parede, assim como tomar as ações coerentes, como refletir a trajetória da bola, "avisar" a um tijolo que houve colisão, etc.;
- Gerenciador da raquete que inclui a movimentação da raquete em função de comandos do usuário;
- Gerenciador dos tijolos, capaz de inicializar todos os tijolos com suas devidas características e removê-los em caso de colisões.

O segundo jogo implementado foi denominado Ship (ver Figura 8), um jogo de nave no estilo RiverRaid, onde o jogador tem o objetivo de controlar uma nave por um campo contendo algumas naves inimigas, destruir o máximo de inimigos possível utilizando as armas da nave e coletar alguns itens de energia e armas especiais pelo caminho.

Os seguintes gerenciadores de objetos foram modelados para o Ship:

- Gerenciador da nave, responsável por identificar colisões dela com os inimigos, a borda da tela e os itens a serem coletados, assim como responder aos comandos do jogador (movimentação e tiro);
- Gerenciador dos inimigos, que inicializa os inimigos no momento previsto e administra seu ciclo de vida;
- Gerenciador dos tiros da nave do jogador, que permite a visualização dos tiros e que detecta colisão entre eles e os inimigos, tomando as providências devidas;
- Gerenciador das balas dos inimigos, idem ao caso anterior.
-

Figura 8 - Jogos BreakOut e Ship, respectivamente, em execução no emulador

8 Implementação de Jogos no Brasil

No Brasil, a produção de jogos, ou mesmo a pesquisa tecnológica, é ainda bastante incipiente. No entanto, um significativo avanço tem sido registrado nos últimos dois anos tanto na academia quando no meio empresarial. As principais iniciativas são rapidamente enumeradas a seguir.

A *Continuum Entertainment* [61] do Paraná é uma das poucas empresas nacionais que trabalham com o desenvolvimento de jogos. *Othello* e *Outlive* são jogos sofisticados produzidos por esta empresa, que, até onde sabemos, foi a primeira a ter um *publisher* disposto a comercializar o software no exterior.

Em Recife várias iniciativas tem sido criadas em torno do Centro de Informática (CIn) da UFPE. A Empresa JoyStudios (www.joystudios.com), a pioneira na cidade (com o nome ArtVoodoo), começou em 1997 e hoje já conta com um investidor nacional. A JoyStudios tem trabalhado em jogos para Web e para PC, tendo como alguns de seus principais produtos a www.mesadejogos.com.br e o www.chessclubber.com/portugues. Outra empresa é a Jynx Playware (www.jynx.com.br), ainda em incubação mas já financiada por investidor brasileiro, que em breve estará lançando o jogo FutSim, um jogo de estratégia em futebol. O próprio CIn realiza vários esforços em jogos, indo desde a criação da primeira disciplina brasileira dedicada exclusivamente ao tema (www.cin.ufpe.br/~games), a três dissertações de mestrado concluídas mais uma série de outras em andamento. Ultimamente, o CIn tem se dedicado a formar também uma forte competência em jogos para dispositivos móveis.

No Departamento de Computação da Universidade Federal de São Carlos está sendo estruturado um trabalho multidisciplinar em Educação a Distância. Uma parte deste projeto utiliza técnicas de jogos por computador para sintetizar um ambiente lúdico de aprendizagem. Edugraph [62] é um programa de ensino de conceitos de computação gráfica que utiliza técnicas de jogos por computador. Também, em conjunto com o CIn/UFPE está sendo avaliado e desenvolvido um módulo 3D a ser acoplado ao motor FORGE V8 [44] desenvolvido neste centro.

A cidade do Rio de Janeiro abriga o maior programa de Pós-Graduação de Computação Gráfica do país através da parceria entre a PUC-Rio e o IMPA, que envolve cursos, empresas incubadas e grupos de pesquisa (ICAD, TeCGraf, NAE, Visgraf, ...). A linguagem Lua [63] , desenvolvida por Roberto Ierusalimschy (PUC-Rio), Luiz Henrique Figueiredo (PUC-Rio, IMPA) e Waldemar Celes (PUC-Rio), foi usada pela empresa Lucas Arts na implementação dos jogos *GreenFandango* e *Escape From The Monkey Island*. O TeCGraf desenvolveu um jogo de guerra naval de interesse internacional. Na PUC-Rio, o ICAD montou recentemente um grupo de pesquisa em jogos (IGames), em parceria com o NAE (Nucleo de Arte Eletronica do Dept. de Artes & Design). No mercado carioca, merece destaque o jogo Solaris, da empresa Solaris, com capital de banco privado.

Fábio Policarpo, um desenvolvedor de software e fundador da companhia Paralelo Computação instalada no Rio de Janeiro, escreveu em conjunto com o professor Alan Watt do Departamento de Computação da Universidade de Sheffield, Inglaterra, o livro *3D Games, Volume 1 : Real-time Rendering and Software Technology*, que tem sido bastante referenciado. O livro se reporta ao Fly3D [64], um motor de jogo escrito totalmente em C++, o qual usa a OpenGL 1.1 como API 3D, o DirectInput do DirectX como controlador de entrada via mouse e teclado, o DirectSound para a implementação de som 3D e o DirectPlay para controle de multi-usuários. O motor roda no Windows/MS 95 e 98, 2000 e NT4.

9 Considerações Finais

A idéia central deste artigo foi o de apresentar, de forma sucinta e introdutória, alguns dos principais conceitos envolvidos na implementação de jogos por computador. No entanto, outro objetivo importante foi o de chamar a atenção para a área, ressaltando a importância de se começar a desenvolver no país, em centros acadêmicos e empresariais, tanto a tecnologia como a cultura necessária para a implementação deste tipo de aplicativo.

Observe-se que, além de sua importância comercial em entretenimento e das possibilidades de aplicações educacionais, este tipo de software suscita desafios científicos e tecnológicos maiores, especialmente nas áreas de computação gráfica, inteligência artificial e engenharia de software. Mais que isto, esta área acentua o perfil da computação como área meio, pois o desenvolvimento de um jogo só é possível com a colaboração estreita, perene e intensa de outras áreas em particular as artes visuais, a música e o cinema.

Assim, é injustificável que a comunidade acadêmica e empresarial se coloque à margem deste processo. Neste contexto, os autores se consideraram satisfeitos se tiverem contribuído para a diminuição do preconceito em relação a pesquisas nesta área, bem como para o crescimento de trabalhos que explorem, de forma direta ou indireta, o tema.

Agradecimentos

Os autores gostariam de agradecer ao CNPq pelo suporte aos seus grupos de pesquisa, bem como a todos aqueles que colaboraram para a confecção deste texto. Agradecimentos especiais a Eduardo Poyart e Cesar Pozzer, do IGames da PUC-Rio, pela colaboração na redação do documento. O NAE da PUC-Rio, também colaborou com valiosas sugestões.

Referências

[1] R. B. de Araujo e A. L. Battaiola, "Jogos 3D Interativos Multiusuários na Internet: Estado Atual, Perspectivas e Desafios", Relatório Interno do DC/UFSCar, 1998.

[2] N. Falstein, "*Game Design: How Platform Choices Dictates Design*", *Game Developer*, Maio, 1997.

[3] A. LaMothe, J. Ratcliff, M. Seminatore, D. Tyler, "*Tricks of the Game Programming Gurus*", *Sams Publishing*, 1994.

[4] T. Gard, "*Building Character*", *Game Developer*, Maio, 2000.

[5] A. L. Battaiola, "Projeções e o seu Uso em Computação Gráfica", JAI98, XVIII Congresso Nacional da SBC, Belo Horizonte, MG, 1998.

[6] M. Morrison, "*Networking your Game using DirectPlay*", *Game Developer*, Julho, 1996.

[7] P. Curtis, "*Mudding: Social Phenomena in Text-Based Virtual Realities*", bin.ponton.de/~ole/literatur/mudding.html

[8] B. Hook, "*A Recent History of Interactive 3D Computer Graphics*", *Game Developer*, Julho, 1997.

[9] M. Pritchard, "*Supercharge Your Sprites*", *Game Developer*, Abril, 1995.

[10] J. White, "*Oldtimer's Guide to Better Textures*", *Game Developer*, Junho, 1998.

[11] D. Sicks, "*Different Perspectives on 3D Sprites*", *Game Developer*, Maio, 1997.

[12] D. Sicks, "*Dawn of the 3D Pixel Sprite*", *Game Developer*, Junho, 1997.

[13] S. Elliot, P. Miller et. al., "*Inside 3D Studio MAX 2*", volume 1 – *New Riders.*

[14] P. Kakert e D. J. Kalwick, "Aprenda em 14 dias 3D Studio MAX 2.5", Editora Campus.

[15] A. L. Battaiola, "Projeções e o seu Uso em Computação Gráfica", JAI98, XVIII Congresso Nacional da SBC, Belo Horizonte, MG, 1998.

[16] S. Melax, "*A Simple, Fast, and Effective Polygon Reduction Algorithm*", *Game Developer*, Novembro, 1998.

[17] P. Steed, "*The Art of Low Polygon Modeling*", *Game Developer*, Junho, 1998.

[18] M. Guymon, "*And Now for Something Completely Different*", *Game Developer*, Março, 1999.

[19] J. Lander, "*Working with Motion Capture File Formats*", *Game Developer*, Janeiro, 1998.

[20] D. Knight et al., "*When Motion Capture Beats Keyframing*", *Game Developer*, Set. 1997.

[21] J. Rodgers, "*Animating Facial Expressions*", *Game Developer*, Novembro, 1998.

[22] B. Waggoner e H. York, "*Video in Games: The State of Industry*", *Game Developer*, Março, 1999.

[23] M. S. Miller, "*Producing Interactive Audio: Thoughts, Tools, & Techniques*", *Game Developer*, Outubro, 1997.

[24] G. Graham, "*Exploiting Surround Sound Using DirectSound3D*", *Game Developer*, Janeiro, 1997.

[25] D.Filion,"*Structure of Games*", members.nbci.com/armagammon/articles/structure.htm.

[26] S. Corley, "*Architecting a 3D Animation Engine*", Game Developer, Abril, 1998.

[27] C. Madeira, G. Ramalho, C. Ferraz, "*FORGE V8*: Um framework para o desenvolvimento de jogos de computador e aplicações multimídia", Dissertação de Mestrado, Pernambuco, Julho, 2001.

[28] C. A. Siebra, G. Ramalho, A. Frery, "Uma Arquitetura para Suporte de Atores Sintéticos em Ambientes Virtuais- uma aplicação a jogos de estratégia", Dissertação de Mestrado, Pernambuco, Agosto de 2000.

[29] A. Rollings, D. Morris, "*Game Architecture and Design*", The Coriolis Group, Novembro, 1999.

[30] J. Lander, "*When Two Hearts Collide*", *Game Developer*, Fevereiro, 1999.

[31] S. Rosen e R. Duisberg, "*Bringing Life to HyperBlade*", *Game Developer*, Setembro, 1996.

[32] www.talula.demon.co.uk/allegro/index.html

[33] www.allegro.cc/

[34] A. Watt, F. Policarpo, "*3D Games – Real-time Rendering and Software Technology*", Addison-Wesley, 2001.

[35] http://www.genesis3d.com, 2001.

[36] http://crystal.linuxgames.com, 2001.

[37] http://www.planetquake.com/golgotha, 2001.

[38] R. S. Wright et al., "*OpenGL SuperBible*", Waite Group Press, 1996.

[39] J. Kolb, "*Win game developer's guide with DirectX 3*", Waite Group Press, 1997.

[40] R. Glidden, "*Graphics Programming with Direct3D*", Addison Wesley, 1997.

[41] N. Thompson, "*Building a Scene Using Retained Mode Direct3D*", *Game Developer*, Setembro, 1996.

[42] Sowizral, H.A. and Nadeau, D.R. Introduction to Programming with Java 3D. Tut. SIGGRAPH 99.

[43] S. Corley, "*Architecting a 3D Animation Engine*", Game Developer, Abril, 1998.

[44] C. Madeira, G. Ramalho, C. Ferraz, "*FORGE V8*: Um framework para o desenvolvimento de jogos de computador e aplicações multimídia", Dissertação de Mestrado, Pernambuco, Julho, 2001.

[45] C. A. Siebra, G. Ramalho, A. Frery, "Uma Arquitetura p/ Suporte de Atores Sintéticos em Ambientes Virtuais- uma aplic. a jogos de estratégia", Dis. de Mestrado, Pernambuco, Agosto de 2000.

[46] A. Rollings, D. Morris, "*Game Architecture and Design*", The Coriolis Group, Novembro, 1999.

[47] Szwarcman, D., Feijó, B. and Costa, M. A framework for networked reactive characters, *Proceedings of SIBGRAPI 2000*, Gramado, Brazil, p. 203-210, 2000.

[48] Giguère, E. *Java 2 Micro Edition*. Wiley Computer Publishing, 2000

[49] Feng, Y., Zhu, J. *Wireless Java Programming with J2ME*. Sams Publishing, 2001.

[50] Wireless Newsfactor. URL: http://www.wirelessnewsfactor.com/

[51] Java 2 Platform, Enterprise Edition (J2EE), URL: http://java.sun.com/j2ee/

[52] Java 2 Platform, Standard Edition (J2SE), URL: http://java.sun.com/j2se/

[53] Java2 Platform, Micro Edition (J2ME), URL: http://java.sun.com/j2me/

[54] CDC and the CVM Virtual Machine, URL: http://java.sun.com/products/cdc/

[55] CLDC and the K Virtual Machine (KVM), URL: http://java.sun.com/products/cldc/

[56] Mobile Information Device Profile (MIDP), URL: http://java.sun.com/products/midp/

[57] Fan, Joel. et al. *Black Art of Java Game Programming*. Waite Group Press, 1996

[58] Filion, Dominic. *Structure of Games*, members.nbci.com/armagammon/articles/structure

[59] Pessoa, Carlos & Ramalho, Geber. wGEM: Um Motor para Desenv. de Jogos para Dispositivos Móveis. Monografia de Trabalho de Graduação do Centro de Informática – UFPE, 2000.

[60] Java 2 Platform Micro Edition, Wireless Toolkit, java.sun.com/products/j2mewtoolkit/

[61] www.continuum.com.br

[62] A. L. Battaiola, C. Goyos e R. B. Araujo, "EduGraph: Sistema para aprendizado de conceitos de computação gráfica", WISE-99, Workshop Internacional sobre Educação Virtual, organizado pela Universidade Estadual do Ceará com o apoio da UNESCO e IFIP, 1999

[63] The Programming Language LUA - www.tecgraf.puc-rio.br/lua/

[64] A. Watt, F. Policarpo, "*3D Games – Real-time Rendering and Software Technology*", Addison-Wesley, 2001.