Juliano Soares dos Santos

NUVENS VIRTUAIS COMO EXEMPLO DE TÉCNICAS DE JOGOS PARA GRÁFICOS TRIDIMENSIONAIS EM TEMPO REAL

Dissertação apresentada ao
Programa de Pós-Graduação em
Engenharia de Produção da
Universidade Federal de Santa Catarina
como requisito parcial para obtenção do grau
de Meste em Engenharia de Produção.

Orientador: Ricardo Miranda Barcia, Ph.D.

Florianópolis
2004

S237n Santos, Juliano Soares dos
Nuvens virtuais como exemplo de técnicas de jogos para gráficos tridimensionais em tempo real / Juliano Soares dos Santos; orientador Ricardo Miranda Barcia. – Florianópolis, 2004.
125f. : il.

Dissertação – (Mestrado) Universidade Federal de Santa Catarina, Programa de Pós-Graduação em Engenharia de Produção, 2004.

Inclui bibliografia

1. Realidade virtual. 2. Computação gráfica. 3. Jogos. 4. Nuvens virtuais. I. Barcia, Ricardo Miranda. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia de Produção. III. Título.

CDU:681.31

Catalogação na fonte por: Onélia Silva Guimarães CRB-14/071

Juliano Soares dos Santos

NUVENS VIRTUAIS COMO EXEMPLO DE TÉCNICAS DE JOGOS PARA GRÁFICOS TRIDIMENSIONAIS EM TEMPO REAL

Esta dissertação foi julgada e aprovada para obtenção do grau de Mestre **em Engenharia de Produção** no **Programa de Pós-Graduação em Engenharia de Produção** da Universidade Federal de Santa Catarina.

Florianópolis, 22 de dezembro de 2004.

Prof. Edson Pacheco Paladini, Dr.
Coordenador do Programa

BANCA EXAMINADORA:

Prof. Ricardo Miranda Barcia, Ph.D.
(Orientador)

Prof. Rodolfo Pinto da Luz, Dr.

Profa. Irla Bocianoski Rebelo, Dra.

Profa. Silvana Pezzi, Dra

Dedico este trabalho a duas pessoas em especial:

Meu pai, mestre, guia, ideal.
Aquele que sempre me apoiou; aquele que sempre acreditou.
Professor Neri, minha admiração só é superada pela minha gratidão.

Minha esposa, a pessoa mais importante em minha vida.
Seus olhos e sorriso me davam motivo para seguir adiante.
Vanessa, eu sempre te amarei.

Agradecimentos

Ao professor Ricardo Miranda Barcia, que visionava realidade virtual enquanto outros começavam a pensar em videoconferência.

A todo LRV, em especial os amigos Onivaldo, Rodolfo, Irla e Fabiano, que foram o meu norte nesta confusa jornada.

A minha mãe,
meu conforto psicológico nos momentos mais críticos.

Ao meu irmão,
muito mais que um revisor, o cara mais fantástico deste planeta.

A todos que contribuíram para a realização desta pesquisa e/ou acreditaram na concretização deste trabalho.

Resumo

SANTOS, Juliano Soares dos. **Nuvens Virtuais Como Exemplo de Técnicas de Jogos para Gráficos Tridimensionais em Tempo Real.** 2004. 125f. Dissertação (Mestrado em Engenharia de Produção) – Programa de Pós-Graduação em Engenharia de Produção, UFSC, Florianópolis, 2004.

Através da exploração de uma série de técnicas para geração de nuvens virtuais, este trabalho procura estabelecer a viabilidade do estudo de jogos computacionais modernos como um meio para incrementar a qualidade visual de aplicativos de realidade virtual, em especial os de baixo custo. O texto é fundamentado em três grandes campos relacionados: realidade virtual, gráficos computacionais e jogos. Deste último, o funcionamento e a estrutura contemporânea são apresentados. As técnicas descritas são acompanhadas de implementações em C/C++, para referência e para que suas construções sejam mais concretas, embora elas possam ser aplicadas em diversos projetos pois não são dependentes de nenhuma linguagem de programação específica.

Palavras-chave: Realidade Virtual, Computação Gráfica, Jogos, Nuvens Virtuais.

Abstract

SANTOS, Juliano Soares dos. **Nuvens Virtuais Como Exemplo de Técnicas de Jogos para Gráficos Tridimensionais em Tempo Real.** 2004. 125f. Dissertação (Mestrado em Engenharia de Produção) – Programa de Pós-Graduação em Engenharia de Produção, UFSC, Florianópolis, 2004.

Throughout the exploration of a number of techniques to generate virtual clouds, this work tries to establish the viability of the study of modern computer games as a mean to improve the visual quality of virtual reality softwares, specially at low cost. The text is based in three big related fields: virtual reality, computer graphics and games. About the former, it is shown how it works and its contemporary structure. The described techniques come with C/C++ implementation for reference and for their constructions to be more concrete, although they can be applied in a variety of projects, since they are not dependent of any specific program language.

Key-words: Virtual Reality, Computer Graphics, Games, Virtual Clouds.

Sumário

Lista de Figuras

Lista de Quadros

1 INTRODUÇÃO

1.1 Apresentação

> *A pessoa deve olhar a tela (do computador) como sendo uma janela através da qual se vê o mundo virtual. O desafio da computação gráfica é fazer a imagem na janela parecer real, soar real e os objetos agirem de forma real.*
>
> *(SUTHERLAND, 1965)*

Nos últimos 50 anos vem ocorrendo uma espantosa evolução dos computadores, com processadores cada vez mais rápidos e preços cada vez mais acessíveis. Há muito que não se trata de uma máquina de escrever avançada; hoje quase todos tocam música de alta fidelidade e filme de alta resolução; praticamente todos têm acesso à computação gráfica e muitos à computação gráfica tridimensional em tempo real.

Entretanto uma máquina poderosa não garante “uma janela que pareça real”, apenas um software que faça uso adequado deste hardware pode garantir tal feito. E muitas aplicações de realidade virtual não o fazem, seja pelo foco do estudo, pelo uso indevido das ferramentas ou pela limitação de recursos. O certo é que a experiência do usuário fica limitada.

É verdade que a experiência do usuário é limitada por definição, pois o meio digital ainda não permite explorar todos os sentidos (GERVAUTZ, 1999). Equipamentos para tato são caros e não práticos, enquanto os para olfato e paladar estão no estado inicial de desenvolvimento. Mas a análise da contribuição de cada sentido estabelece outra perspectiva (HEILIG, 1992):

- Visão: 70%
- Audição: 20%
- Olfato: 5%
- Tato: 4%
- Paladar: 1%

Assim, ainda que se trabalhe somente com visão e audição é possível capturar a maior parte da atenção do cérebro, permitindo um considerável nível de imersão. E, por essa ótica, uma atenção especial deve ser dada à visão.

No nível comercial, os aplicativos que têm explorado o limite da computação gráfica têm sido os jogos (HADWIGER, 2001). Um passeio de automóvel, uma

guerra no meio da floresta, uma final de copa do mundo. O realismo é tal que muitas vezes é difícil definir se são imagens verídicas ou sintéticas.

Figura 1 - Imagem verídica X sintética.
Fonte: *Konami Studio*

É interessante que esse realismo possa ocorrer não somente em aplicações de entretenimento. E é neste contexto que se apresenta este trabalho.

1.2 Justificativa

Desde a década de 70, a idéia de através da tecnologia mergulhar em outros lugares, mundos ou universos tem invadido a imaginação das pessoas, visto a quantidade de romances literários e cinematográficos sobre o assunto[1]. Porém a realidade virtual já não pertence ao campo da ficção: é fato, é cotidiano. Basta observar como são projetados carros na *BMW*, como são apresentadas novas aeronaves na *Embraer*, como são realizados tratamentos para vários tipos de fobias[2].

Passou a ser algo tão banal que o próprio termo parece desgastado, às vezes soa até ultrapassado. Acontece que não se trata mais de uma aplicação específica, todas as áreas do conhecimento passaram a usufruir um pouco da realidade virtual.

O desafio desvia-se da concretização dessa tecnologia e passa para sua popularização. Neste sentido, o caminho ainda parece longo, pois os aplicativos de baixo custo ficam distante do imaginário popular, sendo visualmente muito pouco convincentes.

---

[1] Por exemplo, na literatura: *True Names* (Vernor Vinge, 1978), *Neuromancer* (William Gibson, 1984), *Snow Crash* (Neal Stephenson, 1992). No cinema: *Tron* (1982), *The Lawnmower Man* (1992), Johnny *Mnemonic* (1995), *The 13th Floor* (1999), *eXistenZ* (1999), *The Matrix* (1999).

[2] Ver seção 2.1.3, página 22.

A exceção fica por conta da classe de aplicativos voltada para o entretenimento: os jogos. O diferencial está no fato destes explorarem um mercado de 18 bilhões de dólares anuais (FULLERTON, 2004) com dezenas de novos títulos mensais, mas destinados a funcionarem em máquinas de baixo custo. Assim, vários recursos e procedimentos foram desenvolvidos e aplicados, de forma que um novo título se sobressaísse em relação aos outros e fazendo com que cada nova geração fosse mais foto-realística que as anteriores.

Nos jogos, os efeitos gráficos são usados, em geral, simplesmente para apelo visual. No fogo não há preocupação com o escoamento laminar ou turbulento da chama, com a velocidade da queima do combustível sólido ou líquido, nem com a quantidade de energia luminosa resultante da reação. As nuvens não têm relação com a umidade, nem sofrem influência das massas de ar quente ou frio. Não se trata de uma simulação, o evento não precisa seguir um modelo físico. Por outro lado, o resultado é bastante persuasivo.

Se outras aplicações fizerem uso destas mesmas técnicas, é possível que a realidade virtual apresente requisitos mais simples e preços mais competitivos, saindo do âmbito das universidades ou grandes empresas e alcançando a grande massa.

1.3 Objetivos

Para delinear o projeto apresentado são definidos dois níveis de objetivos a serem alcançados na execução deste trabalho: um geral e outro específico, que podem ser entendidos, respectivamente, como metas e submetas, divididos e descritos conforme segue.

1.3.1 Objetivo geral

O objetivo geral desta dissertação é explorar uma série de técnicas utilizadas nos jogos computacionais modernos para atingir determinado efeito visual, servindo como referência para o desenvolvimento de aplicativos de realidade virtual. Por esta razão, as técnicas são apresentadas de forma direta e prática, muitas vezes no estilo “tutorial”.

Nesse percurso pretende-se levar o leitor a conhecer um pouco mais sobre realidade virtual, computação gráfica e jogos.

1.3.2 Objetivos específicos

Podemos elencar três objetivos específicos a serem alcançados neste trabalho:

- Apresentar múltiplas técnicas para obtenção de nuvens;
- Desenvolver uma implementação de referência para cada técnica;
- Categorizar as técnicas segundo os critérios de hardware necessário, recursos computacionais consumidos e nível de especialização.

1.4 Questão a investigar

Diante do exposto anteriormente, pode-se determinar como questão a ser investigada a viabilidade do uso das técnicas de jogos, em especial as de efeitos visuais, por aplicativos ou projetos de realidade virtual de uma forma geral.

As produções contemporâneas de jogos trabalham com enormes orçamentos, cronogramas extensos e grandes equipes (FULLERTON, 2004), admitindo processos que projetos mais modestos não podem aplicar, como no caso da construção dos modelos tridimensionais e texturas. Além disso, existe o fato deste tipo peculiar de software permitir certos procedimentos que podem não ser cabíveis em outros aplicativos, isto é, o uso de técnicas de jogos pode acarretar em perdas no realismo científico em aplicações industriais ou mesmo educacionais, inviabilizando a utilização.

Por isso a importância de algum tipo de avaliação ou categorização das técnicas.

1.5 Delimitação do estudo

Quatro tipos de efeitos visuais obtidos em jogos seriam desejáveis de ser estudados: sombras e iluminação; geração de terrenos; oceanos e líquidos; fenômenos naturais não-sólidos como nuvens, fogo, raios. A fim de tornar este trabalho realizável foi explorado apenas nuvens, efeito escolhido conforme o capítulo 4.

Para o efeito escolhido, foi pesquisado em vários jogos – com reconhecimento técnico ou comercial – como se implementou o mesmo. Entretanto, a natureza fechada do setor impede uma mostra tão diversificada quanto ao número de títulos disponíveis.

A principal fonte de recursos são os congressos onde os desenvolvedores trocam informações e experiências, como por exemplo, SIGGRAPH[3], EUROGRAPHICS[4], *Game Developer Conference*[5].

É importante salientar o contraste das gerações de técnicas presentes nesse trabalho, misturando títulos ultrapassados, contemporâneos e ainda não liberados. A idéia não é indicar a melhor técnica para o efeito, e sim a melhor técnica para o projeto, que pode ser limitado em virtude da plataforma destinada ou pelo nível de especialização do pessoal disponível.

1.6 Estrutura do trabalho

O presente trabalho está dividido em nove diferentes capítulos, abordando os conteúdos descritos a seguir.

O capítulo 1 introduz o trabalho, dissertando sobre o problema abordado, justificando-o e limitando-o. São descritos também os objetivos geral e específicos, além da questão a ser investigada.

No capítulo 2 são apresentados os três grandes assuntos estudados: realidade virtual, computação gráfica e jogos. É exposto um breve histórico, os principais conceitos e como se relacionam.

Já o capítulo 3 trata especificamente sobre jogos, como funcionam e como são estruturados. No final fala-se sobre *OpenGL*, principal biblioteca gráfica da atualidade, utilizada aqui nas implementações de referência.

O capítulo 4 traz os motivos da escolha do efeito e como cada técnica para realizá-lo é categorizada.

---

[3] A ACM SIGGRAPH (*Association of Computing Machinery's* / *Special Interest Group on Computer Graphics*) organiza anualmente uma conferência nos Estados Unidos reunindo as maiores inovações na teoria e prática da computação gráfica e técnicas de interatividade.

[4] Na Europa, a EUROGRAPHICS (*European Association for Computer Graphics*) faz papel semelhante, mostrando o estado da arte na computação gráfica, multimídia, visualização científica e interfaces homem-computador.

[5] Realizado pela editora *CMP Media LLC* e com apoio da IGDA (*International Game Developers Association*), a *Game Developer Conference* é o grande evento sobre desenvolvimento de jogos.

No capítulo 5 são apresentadas e explicadas as técnicas investigadas para geração de nuvens.

O capítulo 6 apresenta a conclusão do trabalho relatando os objetivos alcançados e sugerindo trabalhos que podem vir a ser realizados tomando por base o ora apresentado.

No capítulo 7 são listadas as obras citadas e consultadas para fundamentação teórica do presente documento ou que forneceram algum subsídio para sua elaboração.

O capítulo 8 contém as implementações completas das técnicas para serem utilizadas como referência.

Por fim, o capítulo 9 trata-se do glossário, listando os vários termos, expressões e siglas presentes no trabalho.

2 REALIDADE VIRTUAL, GRÁFICOS COMPUTACIONAIS E JOGOS

2.1 Realidade virtual

Mitificada pela mídia no começo da década de 90, a realidade virtual ainda carrega dúvidas e desconfianças. O que é realidade virtual? Quando passou a existir? O que é feito e que benefício isso trás?

Nesta área de estudo integram-se diversas tecnologias e pesquisam-se principalmente interfaces homem-máquina, técnicas de interação, dispositivos e métodos que agucem os sentidos humanos. Mas para melhor entendê-la é importante conhecer seus conceitos, origens e aplicações.

2.1.1 Conceitos

Realidade virtual, por se tratar de uma tecnologia recente e baseada em diversas áreas de conhecimento, ainda não possui um conceito único, mas definições distintas e muitas vezes conflitantes.

Para alguns é a união de dispositivos tecnológicos como capacete, luvas, geradores de áudio e vídeo tridimensional. Para outros é algo mais amplo, incluindo livros convencionais, filmes ou até sonhos. Mas em geral existe sempre uma relação com computadores, interatividade e resposta em tempo real.

Pimentel (1995) define realidade virtual como o uso da alta tecnologia para convencer o usuário de que ele está em outra realidade: um novo meio de “estar” e “tocar” em informações: “realidade virtual é o local onde os humanos e computadores fazem contato.”

Aukstakalnis (1992) segue uma linha semelhante: “realidade virtual é uma maneira de humanos visualizarem, manipularem e interagirem com computadores e dados extremamente complexos.”

Latta (1994) cita realidade virtual como uma avançada interface homem-máquina que simula um ambiente realístico e permite que participantes interajam com ele: “realidade virtual envolve a criação e experimentação de ambientes.”

Burdea (1994) estabelece que para uma aplicação seja realidade virtual esta deve proporcionar imersão, interação e ainda estimular a imaginação.

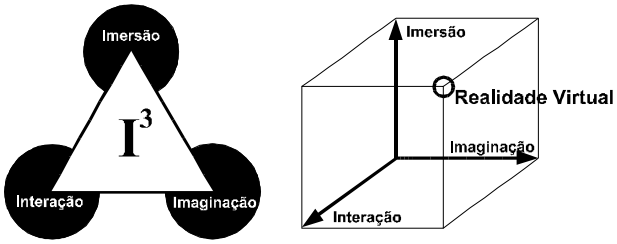

Figura 2 - Triângulo/trilogia da realidade virtual

Aqui estaria o grande debate do conceito, pois a obtenção dessa trilogia não traria a realidade virtual, mas a realidade em si: se houver consciência de que se está no virtual, não há real imersão; se não houver, qual a distinção? Seria algo como o mito da caverna de Platão[6] ou a cilada do demônio de Descates[7].

Pode-se ir além. Segundo Johnston (2000) "(a consciência humana) evoluiu para impor uma interpretação específica das energias e matérias que estão à nossa volta", ou seja, não há objetos vermelhos ou verdes, mas objetos que emitem ondas eletromagnéticas de determinadas freqüências que são captadas pelos olhos e interpretadas de modo a facilitar a identificação. Dessa forma, a própria consciência humana pode ser considerada uma realidade virtual, totalmente imersa num ambiente completamente interativo e constantemente modificado/atualizado pela imaginação (ROSA Jr, 2003).

Por isso, o termo é muitas vezes rejeitado, sendo usado em seu lugar "ambiente virtual", "realidade artificial" e "realidade sintética". Visto dessa maneira, não considerando de forma tão rígida a acepção das palavras, pode-se usar a definição de Luz (1997):

> *(...) realidade virtual é a utilização de artifícios para a reprodução da realidade, sendo que atualmente o meio mais utilizado é o digital, através do uso dos computadores. Para se criar à realidade virtual, é necessário aguçar o maior número de sentidos do usuário, sejam eles visual, auditivo, tátil, dentre outros. Assim, o usuário sente-se inserido, ou seja, imerso em um ambiente no qual pode interagir com objetos e outras pessoas.*

---

[6] O filósofo grego Platão escreveu, no século IV a.c., um conto sobre uma prisão subterrânea na qual as pessoas ficavam amarradas no mesmo lugar desde a infância e onde tudo o que conseguiam ver eram as sombras das pessoas e objetos que estavam fora. O cárcere era tão eficiente que elas não percebiam que estavam presas e pensavam que o mundo era mesmo aquele monte de sombras.

[7] René Descartes, filósofo francês do século 18, imaginou a possibilidade de um terrível demônio estar constantemente lhe dando a ilusão de que todas as suas certezas eram corretas, quando na verdade elas não fariam qualquer sentido. Ele concluiu que, como não se pode provar se esse demônio existe ou não, nenhuma de suas opiniões era segura.

É comum haver confusão com outros conceitos, como realidade aumentada, telepresença e cyberespaço.

Realidade aumentada (*augmented reality*) é uma tecnologia que “apresenta um mundo virtual que enriquece, ao invés de substituir o mundo real” (BRYSON, 1993). Isto é obtido pela projeção de objetos artificiais sobre os reais.

Telepresença é a capacidade de agir à distância tendo a sensação de estar naquele ambiente (GIBILISCO, 1994). Através de robôs, há a extensão dos sentidos dos seres humanos a locais remotos (VIEIRA, 1999).

Vários acessórios são comuns ou semelhantes nas três tecnologias – realidade virtual, realidade aumentada e telepresença – como, por exemplo, capacetes, rastreadores de movimento, dispositivos de retorno de força. A distinção fundamental da realidade virtual está em trabalhar num ambiente totalmente sintético.

Cyberespaço, apesar de relacionado com realidade virtual, é mais abstrato, só existindo na mente dos participantes. Tem mais afinidade com (a idéia) Internet. William Gibson criou o termo no seu livro *Neuromancer*:

> *O cyberespaço. Uma alucinação consensual vivida diariamente por bilhões de operadores autorizados, em todas as nações, por crianças aprendendo altos conceitos matemáticos... Uma representação gráfica de dados abstraídos dos bancos de todos os computadores do sistema humano. Uma complexidade impensável. Linhas de luz abrangendo o não-espaço da mente; nebulosas e constelações infidáveis de dados. Como marés de luzes de cidade...*

Também relacionados estão os termos mundo virtual, avatar e tempo real.

Mundo virtual é ora empregado simplesmente como sinônimo de realidade virtual (GERVAUTZ, 1999), outras vezes, de forma menos concreta, como um espaço que existe na mente do indivíduo. Em geral, é considerado como sendo os modelos computacionais com os quais os usuários interagem nas simulações (LUZ, 2004).

Avatar é a representação do usuário (personagem, boneco, objeto) dentro do mundo virtual. A origem do termo se encontra em uma palavra que indica a incorporação terrestre de uma entidade hindu (DAMER, 1998).

Por fim, tempo real é a necessidade de um sistema apresentar resposta correta dentro de um prazo imposto pelas limitações do ambiente. Por exemplo, uma interação em tempo real é atingida quando o usuário não percebe o atraso entre sua ação num dado ambiente virtual e a resposta sensorial deste ambiente – algo inferior

a 100 milissegundos (BARON, 2004). Para computação gráfica, o termo será mais bem detalhado na seção 2.2.2 (página 28).

2.1.2 Histórico

Sendo a integração de diversas tecnologias, não existe uma data precisa para o nascimento da realidade virtual. Pode-se dizer que em termos de área de estudo seu início se deu com os primeiros simuladores de vôo desenvolvidos no final da década de 20, que possuíam simples sistemas mecânicos e reproduziam os instrumentos de navegação e vôo (STUART, 1996).

É essencial lembrar dos primeiros sistemas multimodais, como o *Sensorama*, desenvolvido por Morton Heilig na década de 50. Esse sistema proporcionava as sensações de andar como caroneiro em uma motocicleta nas redondezas do Brooklyn em Nova York, Estado Unidos. Embora não fosse gerado computacionalmente nem em tempo real, estimulava os sentidos visual, auditivo, tátil e até mesmo o olfativo. Várias das técnicas foram posteriormente utilizadas nos sistemas modernos (HEILIG, 2001).

Mas o ponto de partida da realidade virtual contemporânea é considerado o artigo *The Ultimate Display* de Ivan Sutherland (1965), no qual é imaginada uma interface visual em que o computador controla a existência da matéria. Em 1968 Sutherland apresenta um novo artigo – *A head-mounted three dimensional display* – que descreve o desenvolvimento de seu capacete de visualização tridimensional, usando dois televisores em miniatura montados na frente dos olhos e sensores mecânico e ultra-sônico para rastreamento de movimento da cabeça. Os exemplos mostrados foram uma representação da molécula ciclo-hexana e um quarto cúbico, todos formados por simples arestas.

Dispositivos para captura de gestos foram criados em 1977, e para som espacial em 1987. Mas somente em 1989, Jaron Lanier cunhou o termo “realidade virtual” (STUART, 1996), mesmo ano em que o primeiro sistema comercial passou a ser disponível pela empresa *VPL Research*. A partir daí, o campo ganhou a forma atual.

Em 1990, a *W-Industries* lançou um sistema para grande público – o *Virtuality* – um jogo de *arcade* [8] que incluía capacete de visão estereoscópica com rastreamento de posição, controle de posicionamento e plataforma em forma de anel.

Momentos importantes incluem ainda 1992, com o lançamento da CAVE, um novo método de imersão sem capacetes, usando projeção de imagens (CRUZ-NEIRA, 1993); 1993, com o surgimento dos dispositivos de retorno de força (STUART, 1996); e 1998 quando a Disney abriu as portas do *DisneyQuest*, parque temático que incluía várias atrações em realidade virtual (MINE, 2003).

Até metade da década de 90, aplicações de baixo custo eram praticamente inexistentes, pois a computação gráfica tridimensional em tempo real estava restrita às estações de trabalho de alto desempenho, como as oferecidas pelas empresas *Silicon Graphics*, *Hewlett-Packard* e *Sun Microsystems*. Com as aceleradoras gráficas ganhando desempenho e preços acessíveis, a realidade virtual passou a ser usada nas mais variadas áreas (LUZ, 2004).

Novos equipamentos de apresentação, tais como projetores estereoscópicos e telas de fumaça, entraram no mercado, sendo a evolução da realidade virtual nos últimos 10 anos impulsionada principalmente pelo componente visual (BURDEA, 1996).

2.1.3 Aplicações

Como exposto anteriormente, o campo da realidade virtual amadureceu substancialmente, expandindo-se das pesquisas militares e visualização científica para diversas áreas, como medicina, psicologia, educação, engenharia, arquitetura, arte, marketing e entretenimento.

Suas principais aplicações são: visualização, interação, simulação, exploração, treinamento, educação, avaliação, tratamento e diversão (LUZ, 2004).

Por exemplo, a *Embraer* – Empresa Brasileira de Aeronáutica SA – possui um dos mais modernos e bem equipados centro de realidade virtual[9] do Brasil, com

[8] *Arcades*, no Brasil conhecidos como fliperamas, são máquinas de jogos que funcionam a base de moedas ou fichas (uma partida, uma ficha) e podem ser encontradas em bares, parques de diversão e casas especializadas.

[9] Centro de realidade virtual é o nome comercial dado para sistemas avançados que incluem grandes telas com projetores estereoscópicos, servidores gráficos e numéricos de alto desempenho, vários acessórios específicos do ramo de atuação. Geralmente possui uma equipe técnica multidisciplinar e serve a diversos setores da empresa.

recursos de visualização e simulação de variados graus de imersão e variadas formas de interação com o ambiente virtual (LEAL, 2002). É usado para redução do ciclo de desenvolvimento das aeronaves, através da eliminação ou redução de maquetes físicas, utilizadas na avaliação de posicionamento espacial de sistemas e estruturas aeronáuticas ou nas análises de ergonomia, relacionadas aos aspectos antropométricos de pilotos, passageiros e mecânicos. Além disso, a tecnologia é utilizada como uma ferramenta na campanha de marketing, na qual aeronaves são visualizadas em configurações específicas para potenciais clientes.

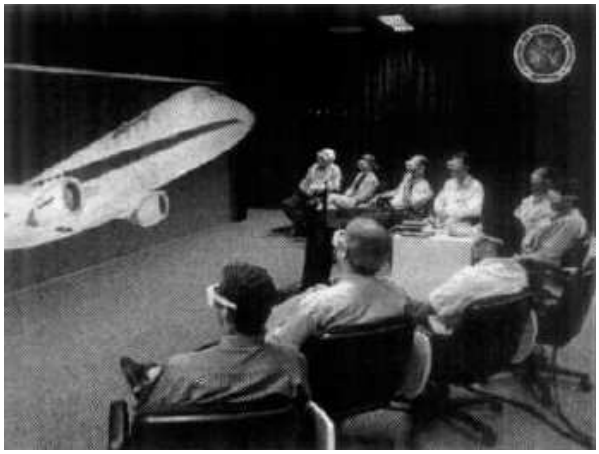

Figura 3 - Centro de realidade virtual da *Embraer*.
Fonte: *Embraer SA*

Na montadora *BMW*, seu centro de realidade virtual é ainda mais essencial, sendo usado para simular e estudar todo o processo construtivo do novo veículo, do acabamento interno ao sistema logístico adotado (RÖTHLEIN, 2003).

Figura 4 - Centro de realidade virtual da *BMW*.
Fonte: *Silicon Graphics Inc.*

Antes de um modelo ser de fato construído, ele, virtualmente, sofre impactos das mais diversas maneiras dentro do centro, buscando encontrar possíveis defeitos ou melhoras no projeto. Só na etapa final são realizados protótipos e *crash tests* verdadeiros, por uma questão legislativa e para realimentar o sistema, validando os dados.

Como o veículo é formado por cerca de 20.000 peças, a análise acústica, quando feita da maneira convencional, exige meses de trabalho, enquanto usando realidade virtual leva apenas alguns dias.

O centro ainda é empregado para auxílio no *design* interno do automóvel, análise ergonômica, avaliação dos materiais, estudo da distribuição de calor e visualização da futura linha de montagem (SGI, 2004).

Outro exemplo representativo é o uso nos Estados Unidos da realidade virtual no tratamento contra dor e fobias.

Num projeto conjunto da universidade de Washington (em específico o *HITLab*, laboratório de tecnologias em interface humana) e do *Harborview Burn Center*, de Seattle, a realidade virtual é usada para distrair pacientes com queimaduras severas (HOFFMAN, 2001). Estes pacientes, além do trauma inicial, precisam entrar numa longa jornada de recuperação, sendo necessária a remoção diária de pele morta para prevenir infecções – um processo tão agressivo que nem mesmo o uso de analgésicos a base de morfina consegue controlar a dor.

Como a dor tem um forte componente psicológico, a introdução de alguma distração para o paciente em tratamento, fazendo-o ouvir uma música, por exemplo, tem sido há muito usado; o benefício da realidade virtual está no fato dela buscar maior imersão e usar o maior número de sentidos. Com um software chamado *SnowWorld*, crianças em tratamento têm a ilusão de estar voando por um vale glacial, distraindo-se “atirando bolas de neve” enquanto médicos e enfermeiras executam o procedimento necessário.

Figura 5 - *SnowWorld* / tratamento de queimaduras.
Fonte: *Scienfic American*

Os pacientes têm relatado uma queda substantiva do sofrimento; fato comprovado pela análise da atividade cerebral em resposta à dor, usando ressonância magnética (HOFFMAN, 2003).

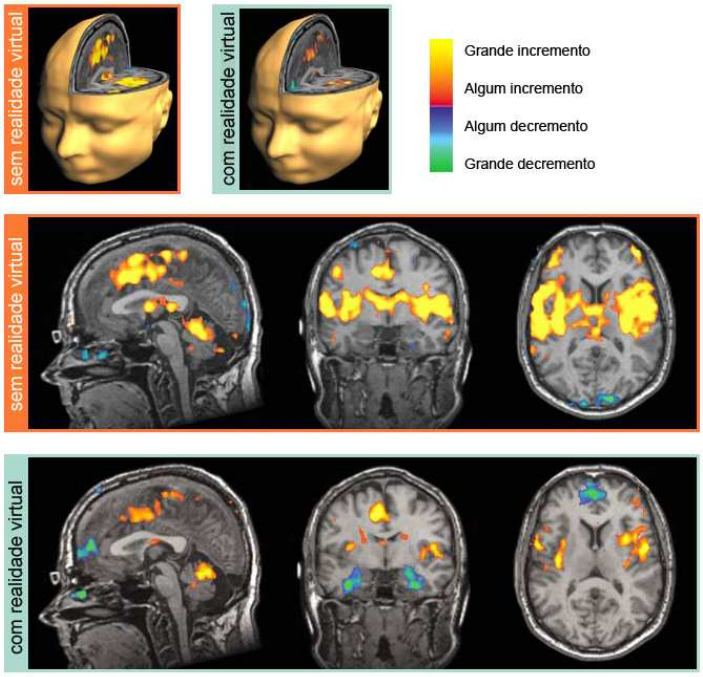

Figura 6 - Atividade cerebral em resposta à dor.
Fonte: Scienfic American

Casos estimulantes como esse e ainda, como o tratamento de estresse pós-traumático (PTSD) pelo hospital presbiteriano de Nova York e pelo *Weill Cornell Medical College* (DIFEDE, 2002), além do combate a fobias pelo *Virtual Reality Medical Center* (VRMC, 2004), indicam que a realidade virtual está se firmando como grande instrumento de redução de sofrimento e ajuda psicológica.

2.2 Gráficos computacionais

Computação gráfica é o ramo da computação que trabalha a geração de imagens sintéticas e a integração ou alteração das informações visuais e espaciais adquiridas do mundo real (imagens digitalizadas).

É um campo amplo e complexo, exigindo forte pesquisa em matemática, física, algoritmos e computação em geral, apenas para citar alguns tópicos.

2.2.1 Histórico

Os computadores já eram utilizados para mostrar pontos representando linhas e formas desde o advento dos tubos de raios catódicos (CRT). Por exemplo, em meados da década de 50, a força aérea americana desenvolveu o sistema SAGE (*Semi-Automatic Ground Environment*) que convertia informações de radar em imagem.

Mas os gráficos computacionais só tiveram real avanço quando Ivan Sutherland, em 1963, desenvolveu o software *Sketchpad*. Revolucionário, como entrada utilizava a recém inventada *light pen*; tinha conceito espacial (eixo x e y), organizava os dados geométricos como objetos e instâncias (apontando para a futura programação orientada a objetos) e foi o primeiro programa da história a ter a interface totalmente gráfica. Influenciou, inclusive, o trabalho de Douglas Engelbart[10], diretamente o NLS e consequentemente os trabalhos do *Xerox* PARC e os futuros *Apple Macintosh* e *Microsoft Windows*.

Em 1964, William Fetter, da empresa *Boeing*, demonstrou a primeira figura humana modelada computacionalmente, usando-a para estudo de *design* de *cockpits*. No *Bell Laboratories*, Ed Zajak, com um satélite animado, mostrou como animações poderiam ser utilizadas para apresentar resultados de cálculos científicos.

Em 1966, Dave Evans fundou na universidade de Utah o primeiro departamento de computação gráfica, o qual na década seguinte, viria a ser o maior pólo da área, com inovações, sobretudo, em gráficos tridimensionais. Três anos mais tarde, em parceria com seu amigo e colega de departamento Ivan Sutherland, fundou a *Evans & Sutherland*, empresa voltada ao desenvolvimento de equipamentos para gráficos computacionais, que tinha entre os empregados John Warnock (fundador da *Adobe*, empresa líder em programas para gráficos bidimensionais) e Jim Clark (fundador da *Silicon Graphics*, veterana em estações de trabalho para computação gráfica).

Novas técnicas e algoritmos surgiram com força na década de 70. Pierre Bézier, trabalhando para *Renault* e procurando formas de desenhar veículos, desenvolveu

[10] Douglas Engelbart inventou no mesmo ano o *mouse*. Durante a década de 60 ele foi o líder de um grupo de pesquisadores no *Stanford Research Institute*, onde desenvolveram o *oNLine System* (NLS) que empregava diversas idéias pioneiras na interação homem-computador. Posteriormente vários pesquisadores deixaram o grupo, terminando no *Xerox Palo Alto Research Center* (PARC), mas levando consigo vários conceitos como interface gráfica controlada pelo *mouse*, aplicações representadas por janelas e hipertexto.

uma maneira de representar curvas suaves com gráficos computacionais em 1970, e, em 1971, Henri Gouraud introduziu seu método de colorização. Em 1974, Sutherland e Gary Hodgman desenvolveram algoritmo para corte de polígonos. No mesmo ano, Ed Camull apresentou seu trabalho com *z-buffer* e mapeamento de textura em superfícies curvas na universidade de Utah. Nesta mesma universidade, Phong Bui-Tuong desenvolveu o modelo de iluminação especular e colorização com interpolação de normais, no ano de 1975. Jim Blinn introduziu o mapeamento ambiental em 1976. Em 1977, Frank Crow desenvolveu uma solução para o problema de interferência (*aliasing*), e, em 1978, Jim Blinn, o mapeamento de rugosidade (*bump mapping*). Turner Whitted mostrou em 1979 o método de traçado de raios (*raytracing*), paradigma que incorpora reflexão, refração, *antialiasing* e sombras.

Grande parte do avanço dessa década se deve ao aparecimento, em 1971, do microprocessador – miniaturizando os computadores e expandindo seu potencial – e da indústria da informática tornar fato a lei de Moore[11].

Também durante essa década a computação gráfica passou a ser ferramenta para a televisão. Escaneando material existente era possível manipulá-lo à vontade, esticando, deformando, girando e fazendo-o "voar" pela tela. Gráficos tridimensionais passaram a ser mais observados pela mídia, a partir de 1977, com o lançamento do filme *Star Wars*, que continha uma seqüência de 40 segundos de computação mostrando os planos de destruição da *Estrela da Morte*[12]. Nas décadas seguintes o cinema seria a referência para a evolução da tecnologia.

TRON, filme de 1982, continha 15 minutos e 235 cenas geradas por computador. Nesse ano Donald Meagher apresentou o *octree*, mecanismo para modelagem geométrica. Sistema de partículas foi introduzido por William Reeves em 1983. Em 1984 You-Dong Liang e Brian Barsky desenvolveram um eficiente algoritmo para corte de regiões retangulares, e Cindy Goral, Kenneth Torrance, Donald Greenberg e Bennett Battaile apresentaram radiosidade. Nesse mesmo ano o filme *The Last Star Fighter* usou computação gráfica ao invés de modelos físicos para representar

[11] Observação feita em 1965 por Gordon Moore, co-fundador da *Intel*, na qual o número de transistores por polegada quadrada, no circuito integrado, dobraria a cada ano desde a invenção deste (MOORE, 1965). Nos anos seguintes a tendência desacelerou, mas a densidade tem dobrado a cada 18 meses, e esta é atual definição da lei. A maioria dos especialistas – incluindo Moore – prevê que ela permaneça por pelo menos mais duas décadas.

[12] No filme *Guerra nas Estrelas* (*Star Wars Episode IV – A new hope*), a *Estrela da Morte* é uma imensa arma, do tamanho de uma lua, capaz de destruir um planeta inteiro.

as espaçonaves. Em 1986, Steve Jobs, fundador da *Apple*, comprou da *Lucasfilm* (produtora do filme *Star Wars*) a *Pixar*, empresa especializada em animação por computador. Três anos mais tarde ela recebeu o Oscar de melhor curta de animação com o filme *Tin Toy*.

O estado da arte da década de 90 é observado em películas como *Terminator 2*, *Jurassic Park*, *Independence Day*, *Star Wars – Episode I* e *The Matrix*. Durante esse período as interfaces gráficas passaram a ser dominantes com o *Microsoft Windows*. Os anos seguintes se caracterizaram pela difusão e acesso à tecnologia.

2.2.2 Conceitos

A computação gráfica possui dois grandes ramos de estudos, relacionados, porém independentes: gráficos bidimensionais (2D) e tridimensionais (3D). Enquanto os gráficos 3D trabalham com modelos volumétricos, os 2D estão vinculados a figuras planas, texto e fotos digitais. A relação principal está no fato dos dados serem apresentados numa saída bidimensional (tela do monitor) e assim em última instância, toda cena 3D passa a ser 2D. Esta operação é conhecida como *renderização*[13].

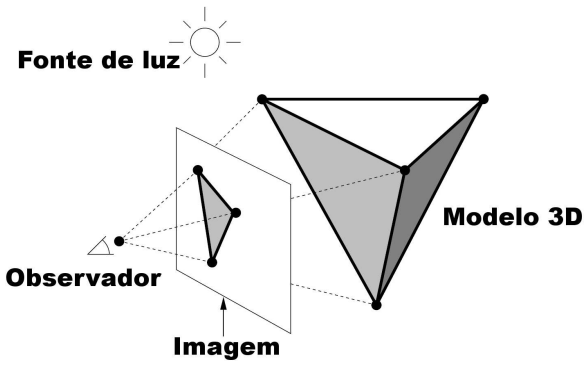

Figura 7 - Renderização

O processo se inicia pela produção do modelo matemático do objeto desejado, que representa não somente a forma, mas também sua cor e acabamento superficial (brilhante, fosca, transparente, irregular, lisa, áspera). Embora esta representação possa ser feita de diversas maneiras (Bézier, *B-splines*, etc.), a maioria dos sistemas gráficos faz algum tipo de conversão, trabalhando somente com polígonos. Curva

[13] Neologismo do autor. *To render*, na computação é a interpretação de algum código.

poligonal é uma seqüência finita de segmentos de linha unidos ponta a ponta. Estes segmentos são chamados arestas, e os pontos finais da linha são chamados vértices (facilmente representados por um conjunto de coordenadas indicando sua posição no espaço). Quando a curva é fechada – termina onde começa – tem-se um polígono e todas as formas são criadas a partir dele, em geral empregando-se triângulos.

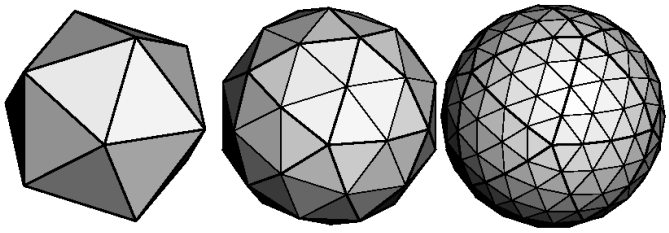

Figura 8 - Esferas poligonais

Esse objeto deve estar inserido numa cena que contém informação sobre a localização e características das fontes de luz (cor, brilho), e a natureza da atmosfera do ambiente na qual a luz viaja (nebuloso ou limpo). Ademais é necessário saber a localização do observador, aquele que segura a câmera virtual através da qual a imagem é fotografada e as características desta. Baseado em todos esses dados, vários passos são necessários para produzir a imagem desejada como, por exemplo, projeção, remoção das superfícies não visíveis, texturização, colorização e rasterização.

Na projeção, pelas características da câmera virtual, projeta-se a cena do espaço tridimensional na imagem plana bidimensional.

Como os elementos mais próximos da câmera escondem os mais distantes, é necessário determinar quais superfícies são visíveis e quais não.

À superfície do objeto pode ser aplicada uma textura[14], uma imagem “colada” como se fosse um adesivo, com objetivo de dar uma aparência mais realística. Essa imagem deverá ser deformada, esticada ou reduzida, de forma a se adaptar ao objeto (mapeamento de textura).

A colorização (*shading*) determina a cor de cada ponto da imagem, que é função da cor do objeto, da sua textura, da posição em relação às fontes luminosas, e nos métodos mais complexos da reflexão indireta da luz de outras superfícies da cena.

[14] Embora muitas vezes se use uma textura para se ter a aparência de um determinado acabamento superficial, *textura* e *acabamento superficial* não são sinônimos.

O passo final é a rasterização, o mapeamento desses pontos no dispositivo de saída gráfico, obtendo-se a imagem rasterizada[15]. Imagens rasterizadas são o que a maioria vê como imagem gerada por computador, matrizes retangulares de pontos chamados *pixels* (*picture element*). O número de *pixels* na matriz é chamado "resolução de imagem"; as possibilidades de cores que o *pixel* pode assumir são chamadas "profundidade de cor".

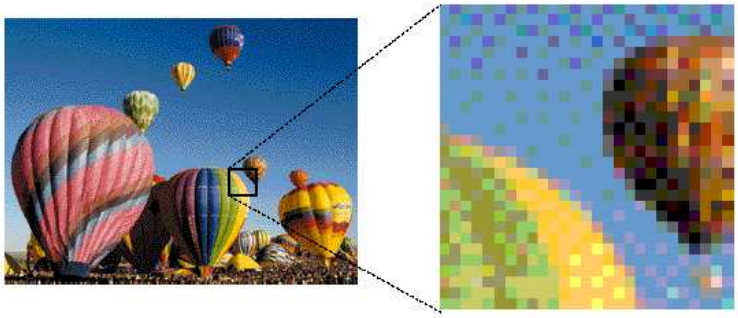

Figura 9 - Imagem rasterizada.

Como a rasterização funciona por amostragem, pois a representação vetorial é um domínio contínuo enquanto a representação matricial é um domínio discreto, problemas de interferência resultando em traços "serrilhados" (*aliasing*) são esperados. Técnicas de *antialiasing* podem ser empregadas, mas como cada primitiva pode gerar um grande número de *pixels*, haverá um custo em termos de desempenho.

Diz-se gráfico tridimensional em tempo real quando não há percepção do processo, ou de maneira mais objetiva, quando a renderização ocorre em menos de 66ms, atingindo um mínimo de 15 imagens (quadros) por segundo e criando uma animação com sensação de movimento natural. O número de imagens por segundo é chamado "taxa de quadros".

Apesar de todo trabalho de renderização poder ser baseado em software, sendo efetuado pelo processador central, nos últimos anos tem havido a disseminação de hardwares especializados: as aceleradoras gráficas. Nas primeiras gerações elas processavam apenas a etapa de rasterização, mas, na atual, há aceleração para todo processo geométrico, com suporte para operações tridimensionais como

[15] Imagens rasterizadas também são conhecidas como imagens matriciais, *pixelmaps* (mapas de *pixels*) ou ainda *bitmaps*. A rigor *bitmaps* são compostos apenas por *pixels* pretos ou brancos (um bit), mas é comumente usado como sinônimo de *pixelmap*, devido, principalmente, ao formato de imagem com dados brutos do *Microsoft Windows* se chamar "bitmap" (.bmp). Neste trabalho não será feito distinção.

transformação, recorte e projeção, além de possibilitar que a etapa de colorização seja programável.

Os recursos das aceleradoras podem ser acessados por APIs[16] como *Direct3D* e *OpenGL*. O assunto será melhor abordado no capítulo 3 (página 53).

A qualidade visual da cena é dependente da técnica de colorização utilizada. São três os principais tipos:

- Constante (*flat*) – cada ponto é colorizado baseado no ângulo entre a normal[17] do polígono que o contém e a direção da fonte de luz (ou seja, será resultante da cor da superfície e da intensidade da iluminação). Na figura 8 foi utilizado este método;
- Gouraud – A normal é computada para os vértices do polígono, determinando a intensidade da iluminação desses e sendo a cor de cada ponto determinada pela interpolação linear dessas intensidades;
- Phong – São calculadas as normais dos vértices do polígono. A partir dessas interpola-se a normal para cada *pixel*, e, baseado nesta, computa-se a intensidade da iluminação.

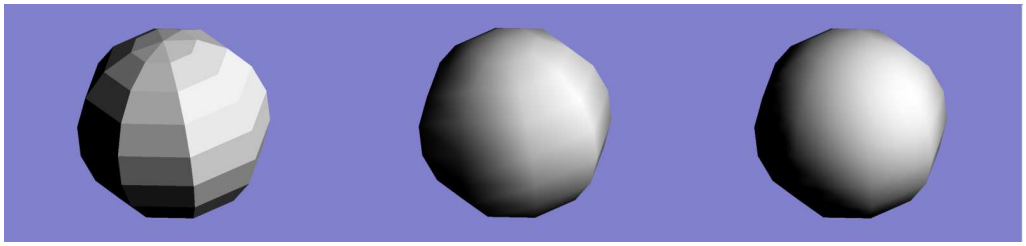

Figura 10 - Esfera com colorização constante, Gouraud e Phong

Tradicionalmente, aplicações que implementam todo sistema gráfico em software só utilizam colorização do tipo constante. Gouraud é o melhor tipo possível quando utilizado o auxílio de hardware acelerador gráfico com linha de processamento fixa. Com a disponibilidade de hardwares programáveis, tornou-se possível a colorização de Phong em tempo real (VALIENT, 2003).

---

[16] Interface de programação da aplicação (*application programming interface*) é um conjunto de definições que permitem a um software se comunicar com outro. É um método de se obter abstração, usualmente (mas não necessariamente) entre software de baixo nível e alto nível.
Uma das finalidades de uma API é prover um conjunto de funções comumente usadas, evitando aos programadores o trabalho de reconstruí-las.
API em si é abstrata: quando o programa provê uma API ele conhecido como *implementação* dela.

[17] O vetor normal indica a orientação angular de um plano ou superfície.

2.3 Jogos

Segundo o dicionário Houaiss (2001), a palavra jogo pode ser entendida como "designação genérica de certas atividades cuja natureza ou finalidade é recreativa; diversão, entretenimento" ou ainda:

> *"atividade, submetida a regras que estabelecem quem vence e quem perde; competição física ou mental sujeita a uma regra, com participantes que disputam entre si por uma premiação ou por simples prazer"*

Aqui não se está interessado em todas essas atividades, mas somente naquelas que ocorrem através do meio digital e eletrônico, possível em várias plataformas computacionais. Estas incluem computadores pessoais, *consoles* dedicados anexados ao aparelho de televisão (*Sony Playstation*, *Microsoft XBox*), *arcades* e específicos aparelhos de parques de diversões, acessórios portáteis especializados (*Nintendo DS*, *Sony PSP*) ou genéricos como assistentes pessoais digitais e telefones celulares.

Assim, não se está fazendo distinção entre as palavras e expressões *jogos*, jogos *computacionais*, *videogames* e *games*.

2.3.1 Evolução dos jogos 3D

Nesta seção pretende-se ilustrar a evolução dos jogos computacionais em três dimensões observando títulos tecnologicamente representativos.

Antes de 1992, os jogos ou não eram tridimensionais ou eram renderizados como simples estruturas aramadas, havendo poucas exceções usando colorização plana. Em alguns casos, eram utilizadas imagens previamente renderizadas de maneira a dar impressão tridimensional, embora o jogo em si não possuísse nenhum tipo de código para cálculo 3D.

***Ultima Underworld* (*Looking Glass Technologies*, 1992)**

Em *Ultima Underworld*, um RPG estabelecido no universo do famoso jogo *Ultima* da *Origin Systems*, os jogadores podiam pela primeira vez caminhar por um mundo tridimensional totalmente texturizado. Anteriormente, como nos primeiros *Ultima*, os RPGs tinham visão superior ou perspectiva isométrica, com gráficos bidimensionais

baseados em células (*tiles*). Não era possível caminhar pelo mundo em perspectiva de primeira pessoa, tornando a imersão muito mais limitada.

Na década anterior, tal limitação era desviada com uma técnica combinando células com imagens feitas a mão, como no jogo *Dungeon Master* da *FTL*. Os jogadores ficavam restritos a uma grade predefinida e só podiam mudar a visão com passos de 90^{o} de rotação, olhando sempre para frente, para esquerda, para direita ou para trás. Não se tratava de um mundo 3D de fato, pois não havia qualquer cálculo tridimensional em tempo de execução.

Por outro lado, o mundo de *Ultima Underworld* praticamente não continha restrição tecnológica quanto à visão e era até certo ponto totalmente 3D. Os jogadores não estavam mais restritos às grades pré-definidas, podiam de maneira fluida andar pelos pisos poligonais texturizados. Rotação sobre os principais eixos era permitida, inclusive olhar para esquerda ou direita com qualquer ângulo, e para cima e para baixo com a correta transformação dos polígonos (fato notável, visto que somente nas próximas gerações haveria coisa do gênero).

Entretanto, a flexibilidade permitida pelo motor[18] 3D de *Ultima Underworld* tinha seu preço em termos de desempenho. Uma vez que todo ambiente era mapeado por textura e os polígonos podiam ser visualizados em qualquer ângulo oblíquo, não era possível renderizar a tela inteira em tempo real; pelo menos não nos computadores disponíveis em 1992. Por esta razão, o jogo usava uma pequena parte para renderização da visão tridimensional do mundo e a área restante para elementos da interface do usuário, como ícones, inventório do personagem, saída de texto e correlatos.

Figura 11 - *Ultima Underworld* (*Looking Glass*)

[18] Motores de jogos serão melhor detalhados no capítulo 3 (página 48).

Personagens e objetos não eram 3D mas simples *sprites*[19] (*billboards*), isto é, uma série de *bitmaps* que eram escalonados de acordo com a distância do usuário e animados como nas animações 2D tradicionais, trocando-se as imagens.

***Wolfenstein 3D* (*id Software*, 1992)**

Logo após *Ultima Underworld*, uma pequena empresa chamada *id Software* lançou um título que viria a se tornar um novo gênero de jogo: tiro em primeira pessoa. Este jogo, *Wolfenstein 3D*, enfatizava acima de tudo rápida ação, com o jogador andando livremente em perspectiva de primeira pessoa com alta taxa de quadros, tendo como objetivo atirar em tudo que se movesse.

Figura 12 - *Wolfenstein 3D* (*id Software*)

Em comparação a *Ultima Underworld*, *Wolfenstein* era extremamente restrito. O jogador só tinha três graus de liberdade (DOF, do inglês *degrees of freedom*), dois para translação e um para rotação. Isto é, o movimento estava limitado a um único plano, e o ponto de vista só podia ser girado sobre o eixo perpendicular a ele. Então, era possível olhar para esquerda e para direita em qualquer ângulo, e nada mais.

Para que o desempenho fosse razoável, apenas as paredes eram mapeadas com textura; o chão e o teto eram simplesmente preenchidos com cor sólida. Além disso, o posicionamento das paredes era tal que todas ficavam a 90^{o} das outras e todas tinham a mesma altura. Combinado com a restrição de movimento isto permitia um mapeamento de textura extremamente simplificado. Paredes podiam ser renderizadas como uma série de colunas de *pixel*, onde a coordenada de

[19] *Sprites* na computação gráfica, e particularmente nos jogos bidimensionais, são uma categoria de imagens desenhadas na tela. Usualmente pequenos (em relação às dimensões e resolução da tela) e parcialmente transparentes (ganhando formas não retangulares) são usados para personagens e objetos móveis; por exemplo, em *Pac-Man* todos os bonecos são *sprites*.

profundidade era constante para cada coluna. Desta forma a correção da perspectiva era necessária apenas uma vez por coluna.

Assim como em *Ultima Underwolrd*, todos os personagens e objetos eram *sprites* bidimensionais.

A maior contribuição de *Wolfenstein 3D* foi se tornar o protótipo para jogos de tiro em primeira pessoa. Nos anos seguintes, seu estilo estabeleceu a evolução nas técnicas para gráficos tridimensionais em tempo real. Por esta razão, mesmo sendo tecnicamente menos evoluído, *Wolfenstein* é historicamente muito mais relevante que *Ultima Underworld*.

***Doom* (*id Software*, 1993)**

Com a premissa básica idêntica à de *Wolfenstein* – atirar em tudo que se move – mas com ambientes totalmente texturizados, som estéreo e ação frenética, *Doom* permitiu um nível de imersão nunca experimentado num jogo antes.

Em comparação ao título anterior, este apresentava vários avanços gráficos. As superfícies eram totalmente texturizadas (paredes, chão, teto), as paredes não precisavam ser perpendiculares umas as outras nem ter a mesma altura e havia variação do nível de iluminação no ambiente.

Figura 13 - *Doom* (*id Software*)

Isto só era possível em uma alta taxa de quadros e com os equipamentos da época porque algumas limitações eram inteligentemente impostas. Internamente o mundo era representado em duas dimensões, num plano, sendo a diferença de altura aplicada posteriormente (este tipo de combinação de processamento 2D com geometria 3D através de extrusão é conhecido como 2.5D). Uma das restrições decorrentes era a impossibilidade de colocar um ambiente em cima de outro. Por outro lado, isso mantém adequados os três graus de liberdade do *Wolfenstein*.

Como todas as paredes sofriam extrusão ortogonalmente ao chão a partir da representação plana do mundo em tempo de execução, todos os polígonos visíveis eram ou perpendiculares ou paralelos ao plano do chão. Combinado com o fato da rotação do ponto de vista ser possível apenas no eixo vertical, levava a coordenada de profundidade a ser sempre constante para toda distribuição vertical e horizontal dos polígonos (mapeamento de textura com z constante).

Como *Ultima Underworld* e *Wolfenstein 3D*, *Doom* usava simples *sprites* para os personagens e objetos.

***Quake*/*GLQuake* (*id Software*, 1996)**

Quake foi o primeiro jogo de tiro em primeira pessoa verdadeiramente tridimensional, utilizando, inclusive, ambientes espaciais (e não mapas planos que posteriormente sofriam extrusão) e geometrias para objetos e personagens.

Apesar de que isso possa parecer uma simples modificação do mundo 2.5D de *Doom*, na realidade os ambientes 3D são muito mais desafiadores de serem renderizados em tempo real. Por exemplo, todos os polígonos podem ser orientados arbitrariamente, demandando cálculos mais complexos para o mapeamento das texturas. *Quake* possibilitava os cálculos porque era feita a necessária divisão de perspectiva apenas a cada 16 *pixels* e interpolando linearmente dentro do intervalo (a diferença da aproximação e da verdadeira hipérbole da perspectiva era aceitável).

Anteriormente poucos jogos que requisitassem altas taxas de quadros usavam aritmética de ponto flutuante, simplesmente porque a unidade dos processadores não era suficientemente rápida. Porém, em 1996, este tipo de operação se tornou viável e *Quake* fazia uso pesado dela, inclusive no nível das subdivisões do mapeamento de textura. A interpolação linear e aritmética com *pixels* era feita tanto com inteiros quanto com ponto flutuante. Praticamente todas operações de alto nível como transformação, recorte, projeção, eram feitas inteiramente com ponto flutuante.

É importante lembrar dois pontos a respeito do assunto e da plataforma à qual *Quake* se destinava: o processador *Intel Pentium*. Primeiro, a conversão de valor ponto flutuante para inteiro era uma operação inerentemente lenta. Desta forma, aritmética com *pixels* era sempre mais rápida com inteiros, mesmo com o desempenho da unidade de ponto flutuante do *Pentium* ser comparável ou mais rápida às operações inteiras, pois os valores finais são inteiros. Segundo, havia uma

razão específica para este exato ponto de transição entre aritmética de ponto flutuante e inteiro. O *Pentium* era capaz de intercambiar os dois tipos de operações – dado que as unidades de processamento eram separadas – e capaz de operá-las em paralelo. Desta forma, o mapeamento de textura ocorria intercalando a divisão da perspectiva com a interpolação e renderização do *pixel* em processo para cada 16 *pixels* das subdivisões, obtendo a complexa divisão praticamente de graça.

Quake introduziu uma abordagem completamente nova para iluminação, utilizando os chamados mapas de iluminação. As informações das luzes estáticas de todos os polígonos, de um certo ambiente, eram pré-computadas numa textura especial que era utilizada para determinar a intensidade dos *pixels* das texturas de fato.

Figura 14 - *Quake* (*id Software*)

Enquanto inicialmente a renderização fosse baseada em software, logo após o lançamento, a id liberou um executável que permitia *Quake* tirar vantagem da aceleração por hardware, em especial das placas com processadores *VoodooGraphics*. Esta versão ganhou o nome de *GLQuake*, pois usava a API *OpenGL*, anteriormente usada apenas em estações gráficas de alto desempenho.

O momento foi perfeito. Anteriormente todas as tentativas de massificar hardwares especializados para gráficos 3D falharam, tanto pelo desempenho medíocre quanto pela falta de aplicativos que realmente tirassem proveito dos mesmos. A combinação *Voodoo*-*GLQuake* finalmente deixava para trás estes problemas, tornando-se naquele ano o grande sonho de consumo dos usuários de computadores e jogadores em geral.

O jogo, somente em software, só tinha performance adequada com resolução de imagem máxima de 320x240, devido ao nível mais baixo do processamento gráfico, ou seja, a rasterização dos polígonos em *pixels* e o mapeamento das texturas. A aceleração da *VoodooGraphics* era perfeita para estes trabalhos, além de livrar a CPU da realização da divisão da perspectiva para cada *pixel* (ou cada n-ésimo *pixel*), permitindo a utilização da resolução 640x480.

Além disso, a aceleração por hardware era capaz de aplicar filtro bilinear, ao invés de pontual nas texturas, levando a uma qualidade visual muito superior. Mapeamento de texturas com diferentes resoluções (*MIP-mapping*) podia ser usado em tempo real para cada *pixel* individual, em vez de procurar um mapa MIP aproximado para o polígono inteiro, como no caso do renderizado em software.

Em contraste com o *Quake* renderizado em software, que combinava o mapa de iluminação com a textura base antes da aplicação (usando um sofisticado sistema de *cache* de superfícies), *GLQuake* aplicava os mapas diretamente como uma segunda textura usando *alpha-blending* (transparência).

***Quake 3 Arena* (*id Software*, 1999)**

Embora tivesse funcionamento semelhante aos títulos anteriores, *Quake 3* era totalmente voltado a ser jogado via Internet. Havia a possibilidade de jogar sozinho, mas seria como no modo multiusuário sendo os oponentes controlados pelo computador. Por isso, grande parte da tecnologia estava na arquitetura de rede.

Em termos gráficos, *Quake 3* foi o primeiro grande título a exigir aceleração 3D para ser jogado, pois renderização por software não era nem mesmo opcional. Assim, ele podia explorar de forma efetiva os recursos das aceleradoras.

Pela primeira vez num jogo, eram usadas superfícies curvas adicionalmente aos blocos sólidos poligonais. *Quake 3* empregava curvas de Bézier quadráticas com 3x3 pontos de controle, que eram convertidas em triângulos antes de serem enviadas para a aceleradora.

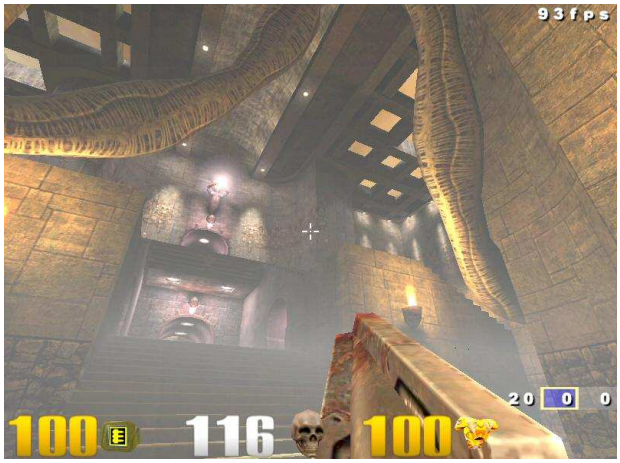

Figura 15 - *Quake 3 Arena* (*id Software*)

Quake 3 também mostrava a tendência de não se usar apenas mais polígonos para obter superfícies mais suaves e mais detalhadas; havia o forte uso de uma renderização de alta qualidade, através do uso de múltiplos passes na mesma.

A aparência da superfície não era mais descrita por uma única textura, um mapa de luz ou alguns outros atributos, mas definida por uma colorização flexível e geral chamada *shader*. Em uma das verdadeiras linguagens para colorização, o *shader* (uma função escrita a partir dessas linguagens) é usado para definir cada *pixel*; em *Quake 3* tratava-se de uma simples linguagem voltada para renderização em tempo real com as limitações da época, definindo toda superfície.

Os *shaders* em *Quake 3* eram, acima de tudo, uma maneira flexível de controlar como uma superfície deveria ser renderizada usando múltiplos passes. Por exemplo, um passe para o mapa de luz, um passe para uma textura de base animada combinando quatro texturas em tempo de execução, um passe para uma imagem de reflexão do ambiente em volta e um passe para a névoa volumétrica. A linguagem de colorização também podia ser usada para todo tipo de animação, não apenas de texturas, a partir do qual várias imagens podiam ser transformadas e compostas em tempo real, mas também animação de cores, animação de nível de transparência e até animação dos vértices.

Em princípio, esses *shaders* não ofereciam nada que não pudesse ser feito no código do jogo em si. Entretanto, representavam muito mais liberdade para os artistas e designers dos ambientes. Sem os *shaders* era sempre necessário um programador para implementar um determinado efeito especial. Dessa forma, os *shaders* permitiram maior possibilidade de experimentação para renderização de superfícies de alta qualidade.

***Doom 3* (*id Software*, 2004)**

Em desenvolvimento por quatro anos, *Doom 3* é uma releitura do título de 1992 buscando utilizar os mais modernos hardwares e técnicas disponíveis. Seu motor pode ser considerado o estado da arte em tecnologia para renderização tridimensional, tanto nos jogos computacionais, quanto nas aplicações de baixo custo em geral.

O conceito principal que o distingue é o sistema unificado de iluminação e sombreamento. Nos jogos anteriores (como a série *Quake*) eram usados vários modelos de iluminação simultâneos dependendo do contexto. Os cenários usavam mapas de iluminação previamente gerados; personagens e objetos projetavam suas sombras computadas em tempo de execução; certas luzes se utilizavam de uma colorização específica ou de mapeamento de iluminação extra. *Doom 3* usa um modelo único para todo ambiente, gerando sombras de forma uniforme em tempo de execução. Isto quer dizer que qualquer luz vai afetar a cena inteira, fazendo inclusive o modelo sombrear a si mesmo (um personagem em baixo de uma luminária tem a sombra de seu queixo no pescoço, independente da maneira como o artista o criou).

Figura 16 - *Doom 3* (*id Software*)

Para conseguir tal feito, *Doom 3* ignora o modelo de iluminação da API *OpenGL* – sobre a qual o jogo é escrito – e usa seu próprio sistema baseado numa técnica

chamada volumes de sombra, implementada sobre *stencil buffer*[20] em hardware, possível graças a capacidade das modernas aceleradoras gráficas.

A técnica divide o mundo virtual em dois: volumes que estão na sombra e volumes que não estão. Projeta-se uma linha da luz em direção a cada vértice do objeto que pode gerar sombra até o infinito. Todas projeções juntas formam um volume; qualquer ponto dentro é sombreado; qualquer ponto fora é iluminado.

De maneira a testar se um dado *pixel* é renderizado como sombreado ou não, o próprio volume de sombra é renderizado, embora não na imagem final, mas apenas no *stencil buffer*. Cada face voltada para o volume aumenta o valor do *stencil buffer*; cada face de costas para o volume reduz. Quando todos volumes estiverem renderizados no *stencil buffer*, qualquer *pixel* com valor diferente de zero está na sombra.

O jogo faz forte uso de luz especular, graças a esse sistema de iluminação por *pixel*, e mapeamento de rugosidade (*bump mapping*[21]). Possui também detalhadas superfícies representando interfaces gráficas do usuário para o personagem, inclusive com ponteiro de mouse e até janelas, quando previamente se interagia usando apenas botões.

Entretanto, não existe uma hegemonia tecnológica da *id Software* como aconteceu nos títulos anteriores. A inteligência artificial é pobre, a interatividade com o ambiente é muito pequena (existem jogos onde é possível destruir qualquer parede, por exemplo) e a física é apenas razoável. Vários outros títulos já faziam uso de iluminação por *pixel* e mapeamento de rugosidade. Ainda assim, pela maneira como está implementado, *Doom 3* detém vanguarda em termos gráficos e no aspecto técnico relacionado.

---

[20] Estêncil: material que se perfura com um desenho ou um texto, e que é usado para imprimir (o desenho ou o texto) sobre uma superfície, fazendo-o rolar ou premindo tinta através dos orifícios que compõem o desenho ou as letras do texto (Houaiss, 2002).
Stencil buffer é a seção da memória gráfica que guarda os dados do "estêncil", usado para restringir o desenho de certas partes da tela.

[21] Desenvolvido por James Blinn em 1978, *bump mapping* é uma técnica que através de uma textura extra, o mapa de rugosidade, perturba-se a normal à superfície no ponto onde se está calculando a intensidade luminosa. Isto equivale (visualmente) a ter-se uma superfície ondulada, embora não haja o aumento do número de polígonos.
Algumas vezes a aplicação da técnica ganha o nome de mapa de rugosidade quando muda a magnitude do vetor da normal (monocromática), e sendo chamada de mapa de normal quando muda a magnitude e direção (colorida).

2.3.2 Evolução dos hardwares dedicados

Mesmo existindo produtos sendo oferecidos anteriormente, 1996 foi o ano um para os hardwares de baixo custo dedicados à aceleração gráfica tridimensional em tempo real. A tecnologia era certa, o preço aceitável e os softwares (jogos) passaram a fazer real uso deles.

Esse sucesso impulsionou um novo nicho de mercado: das placas dedicadas à aceleração gráfica tridimensional para jogos. Em menos de uma década, a contagem passou de dezenas de milhares para centenas de milhões de polígonos. A evolução vem sendo superior à lei de Moore (VENKATASUBRAMANIAN, 2003).

A seguir é mostrada essa evolução a partir dos hardwares dedicados mais representativos.

***VoodooGraphics* (*3Dfx Interactive*, 1996)**

Como dito na seção anterior, a aceleradora *VoodooGraphics* foi o primeiro hardware para 3D realmente usado por vários jogos e a experimentar verdadeiro sucesso comercial (HADWIGER, 2000). Uma das propriedades que viabilizaram esta aceleradora em 1996 foi o fato do chip *VoodooGraphics* ser oferecido sem nenhum suporte para gráficos 2D. Era exclusivo para 3D, sendo necessário utilizá-lo em conjunto com uma placa de vídeo convencional. Ou o sinal de vídeo da placa 2D era retransmitido para o monitor ou não era utilizado, sendo enviado o sinal da placa 3D no lugar. Obviamente com esta configuração era impossível misturar imagens 2D e 3D, ou seja, era impossível renderizar numa janela gráfica; mas como os jogos são normalmente jogados em tela cheia, isto não implicava em um problema maior. Entretanto inviabilizava seu uso para outro tipo de aplicação como modelagem 3D e visualização científica.

A *VoodooGraphics* possuía memória dedicada para *framebuffer*[22] e para textura, em contraste com a arquitetura de memória unificada usada nas aceleradoras modernas, onde se compartilha a mesma RAM. Assim era impossível aumentar a memória de textura quando a memória de *framebuffer* não era totalmente usada. Na época isto não era uma restrição, pois as placas com o *chip VoodooGraphics* só suportavam duas resoluções, 512x384 e 640x480, e estas placas tinham dois

[22] *Framebuffer* é a memória dedicada ao processador gráfico e usada para guardar os *pixels* renderizados antes de serem mostrados na tela.

megabytes para *framebuffer* e dois *megabytes* para textura. Como as memórias não eram compartilhadas, também não era possível renderizar um quadro diretamente como textura (para criação dinâmica de mapas de ambientes, por exemplo), mas como os jogos não faziam uso da técnica isto não se apresentava como um problema.

O *chip* oferecia uma precisão de 16 bits tanto para o *buffer* de cor quanto para o de profundidade. Em 1996 isto era um avanço tremendo, pois praticamente todos os jogos utilizavam cores indexadas em 8 bits em conjunto com uma determinada paleta de 256 cores. Para aplicações 3D, não há preocupação apenas com o número de cores possíveis na tela mas também com os mapas de textura e usualmente o número máximo do buffer de cor e das texturas é o mesmo. No caso, os jogos com cores 8 bits estabeleciam a mesma paleta para as texturas. A *VoodooGraphics* suportava vários formatos de mapas de textura e também aceleração especial para uso de texturas paletizadas. A mais alta resolução de cor suportada era de 16 bits por *texel*, nas configurações 565 ou 1555 respectivamente. Isto com a eliminação da paleta global compartilhada por toda tela levava a um incrível aumento da qualidade visual. Porém os primeiros jogos que exploravam a aceleração do *chip* mantinham quase todas as restrições provenientes do uso de uma paleta global da renderização por software, simplesmente porque isto era um ponto central do motor gráfico e porque o trabalho artístico (texturas) não podia ser facilmente mudado. Assim, levou algum tempo para os jogos realmente explorarem as capacidades da nova placa.

Texturas eram suportas com resolução até 256x256, porém com algumas restrições. O aspecto não podia exceder 1:8 ou 8:1, não sendo possível usar uma textura de 256x16, por exemplo. A largura e a altura da textura tinham que ser sempre potência de dois. Isto só se tornou um problema nos anos seguintes, quando os jogos sem renderização por software (exclusivos para aceleração 3D) começaram a surgir. Anteriormente estas restrições já eram intrínsecas aos jogos, não evidenciando as limitações do *chip*.

Outra razão para o sucesso do *chip* estaria em suas habilidades poderem de fato ser usadas. Nas aceleradoras anteriores quando algo era utilizado havia forte impacto na taxa de quadros. Não era o caso da *VoodooGraphics*, permitindo o uso de filtro bilinear nas texturas, *buffer* de profundidade e *alpha blending*.

***Riva TNT* (*Nvidia Corporation*, 1998)**

Em 1998, a *Nvidia* foi o primeiro competidor da *3Dfx* a apresentar um produto equiparável e até superior a *Voodoo*, tanto em termos de desempenho quanto em termos de habilidades. A aceleradora *Riva TNT* oferecia a baixo custo renderização *OpenGL* de alto desempenho e alta qualidade.

A TNT, que quer dizer *twin-texel*, era uma arquitetura para multitextura em passe único, isto é, cada *pixel* podia receber duas texturas ao mesmo tempo. Isto era usado principalmente para mapas de iluminação, técnica popularizada com o jogo *GLQuake*. O *chip* oferecia *buffer* de cor de 32 bits, *buffer* de profundidade com 24 bits, e até mesmo um *stencil buffer* de 8 bits. Usava memória unificada e como a maior parte das placas com este *chip* tinha 16MB, eram possíveis resoluções até 1280x1024. Ao contrário das *Voodoos*, estas placas combinavam 2D e 3D (já fazendo o papel da placa de vídeo convencional), que somado à uma sólida implementação *OpenGL* 1.1, trazia aceleração por hardware por baixo custo a outras aplicações diferentes de jogos (pois também acelerava em modo janela). A TNT também permitia texturas com tamanhos até 2048x2048.

***GeForce 256* (*Nvidia Corporation*, 1999)**

A introdução da *GeForce 256* foi de grande impacto, pois permitia total aceleração de geometria dentro de uma placa de baixo custo. Anteriormente a aceleração era quase toda no nível da rasterização, de uma maneira mais 2D que 3D, sendo a maior parte do trabalho 3D feito pelo *driver* (software). Entretanto, com aceleração de geometria as primitivas 3D e as matrizes de transformação são enviadas para o hardware que toma conta das transformações, recortes, projeções, assim como rasterização das primitivas.

Outras habilidades incluíam mapeamento de rugosidade por hardware e suporte a mapeamento ambiental cúbico.

A disponibilidade da *GeForce 256* foi especialmente importante para os profissionais que trabalhavam com computação gráfica, uma vez que marca o momento em que as placas de baixo custo se igualavam a estações de ponta de poucos anos anteriores.

***GeForce 3* (*Nvidia Corporation*, 2001)**

Diferentemente da maioria dos produtos anteriores da *Nvidia*, a *GeForce 3* era um produto voltado para alto desempenho, ou seja, mais caro. O mercado, porém, já possuía uma fatia disposta gastar para ter o melhor em sistema para jogos. Vários títulos prometiam utilizar os recursos da aceleradora, em especial o então recém anunciado *Doom 3* (que só seria lançado três anos mais tarde).

Além do aumento no desempenho e várias melhorias como uma arquitetura para evitar o sobredesenho e a mudança do *antialiasing* de *super-sampling* para *multi-sampling*, mais eficiente, a principal característica era a presença de uma unidade especializada para *shaders* de vértices e de *pixels*, atendendo os requisitos da API *DirectX* 8.

Tratava-se de algo muito mais profundo e poderoso do que aqueles usados em *Quake 3*. Através de uma linguagem tipo *assembly*, passou a ser possível programar as etapas de renderização efetuadas pelo hardware. Os *shaders* de vértices permitiam controle das posições, cores e coordenadas de texturas dos vértices enquanto os *shaders* de *pixels* permitiam controle na maneira como cada *pixel* individual seria renderizado na tela.

***GeForce FX* (*Nvidia Corporation*, 2003) / *Radeon 9800* (*ATI Technologies*)**

Tanto a série NV30 (*GeForce FX 5x00*) quanto a R300 (*Radeon 9x00*) são compatíveis com as APIs *DirectX* 9 e *OpenGL* 2.0.

A principal inovação é o suporte de linguagem de alto nível para colorização (HLSL para *DirectX* e GLSL para *OpenGL*). Assim os *shaders* podem ser programados numa linguagem semelhante a C em contraste a tipo assembly da geração anterior.

2.4 Relação entre realidade virtual, gráficos computacionais e jogos

Com o exposto é possível responder algumas questões. Computação gráfica é realidade virtual? Jogos são aplicações de realidade virtual? Qual relação entre as três?

Realidade virtual em geral são aplicações visuais (mas não necessariamente, para uma pessoa cega, outros sentidos são mais importantes), sendo a qualidade e evolução intimamente ligada à computação gráfica. Por exemplo, Ivan Sutherland é

considerado o "pai" da realidade virtual contemporânea, da mesma maneira introduziu conceitos fundamentais para computação gráfica. Em seu artigo *The Ultimate Display* ele não usa o termo "realidade virtual" – que só viria a ser cunhado 24 anos mais tarde – mas computação gráfica, mostrando que originalmente os fundamentos das duas se confundiam. Inicialmente muito da computação gráfica foi impulsionado pelas pesquisas em realidade virtual, uma vez que a primeira era o grande gargalo para concretização da última; hoje muito da realidade virtual é impulsionado pelas pesquisas em computação gráfica, já que o vertiginoso desenvolvimento da última acaba automaticamente aperfeiçoando a primeira.

Da mesma forma, excluindo casos particulares como MUDs[23], jogos são aplicações visuais, praticamente servindo como retrato do desenvolvimento da computação gráfica: *sprites*, animações, a necessidade de maior resolução e profundidade de cor, o uso de imagens digitalizadas, vídeo com movimento (*full motion video*), computação gráfica tridimensional. Desde 1994 com o lançamento do *Playstation* da *Sony*, os jogos têm sido o motor propulsor do 3D em tempo real[24], o mercado necessário para existência de pesquisa no setor.

Por outro lado, gráficos computacionais não se resumem à realidade virtual e jogos. Quando o computador é ligado e entra-se no *Microsoft Windows* já se está numa aplicação gráfica, que não é nem realidade virtual nem jogo. A computação gráfica é fortemente usada para televisão e cinema, que mesmo havendo grande margem para discussão, na definição mais restrita usada neste trabalho não se trata de realidade virtual.

A pergunta final é se um jogo é realidade virtual. Observando Luz (1994) novamente:

> *Para se criar à realidade virtual, é necessário aguçar o maior número de sentidos do usuário, sejam eles visual, auditivo, tátil, dentre outros. Assim, o usuário sente-se inserido, ou seja, imerso em um ambiente no qual pode interagir com objetos e outras pessoas.*

Poucas aplicações fazem mais uso dos sentidos visual, auditivo, tátil[25] que os jogos. Jogos de tiro em primeira pessoa e simuladores proporcionam alto grau de

---

[23] MUD (multi-user dungeon) é um jogo RPG para computador jogado via Internet. O jogador assume o papel de um personagem e vê descrições textuais dos quartos, objetos e outros personagens do mundo virtual.

[24] Ver apêndice C.

[25] Os controles (*joypad*) nos videogames atuais vibram para estimular o sentido tátil. Outros acessórios como tapetes, câmeras de vídeo e microfones são usados para determinadas interações.

imersão, possibilitando inclusive interação com outras pessoas através de ambientes multiusuário. A regra não é geral, nem todos os jogos são imersivos, mas muitos já podem ser considerados realidade virtual.

Outra relação é que como uma aplicação de realidade virtual e um jogo são softwares para gráficos em tempo real, os desafios e a estrutura dos dois podem ser semelhantes, e assim, o estudo de um pode ajudar o outro.

Na seqüência será visto o funcionamento – em termos de programação – dos jogos computacionais, servindo como base para o estudo das técnicas gráficas.

3 JOGOS COMPUTACIONAIS

3.1 A estrutura de um jogo

Segundo LaMothe (2003):

> *Videogames são pedaços de software extremamente complexos. Na verdade, eles são sem nenhuma dúvida os programas mais difíceis de escrever. Claro, escrever algo como MS Word é mais difícil que escrever um jogo asteroids, mas escrever algo como Unreal, Quake Arena ou Halo é mais difícil do que qualquer outro programa que eu possa pensar – incluindo software para controle de armas militares!*

Um jogo é basicamente um *loop* contínuo que processa a lógica (o que pode, o que não pode, o que deve acontecer naquele tempo) e desenha uma imagem na tela. Um esquema simplificado é mostrado e descrito abaixo.

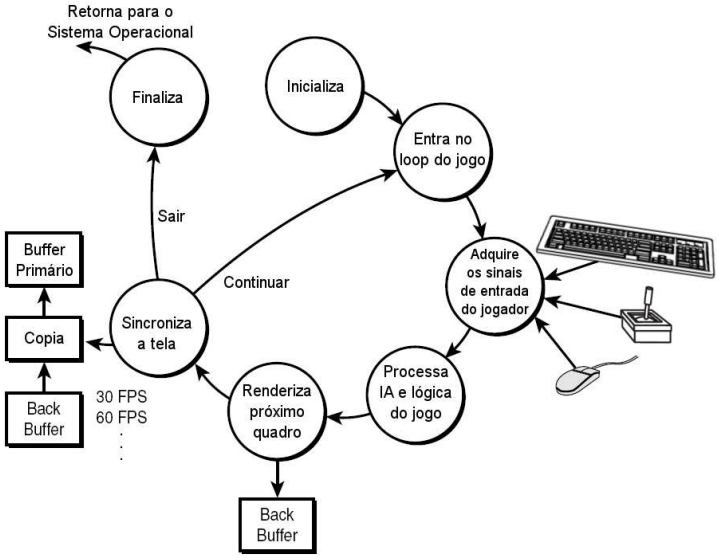

Figura 17 - *Loop* de jogo simplificado

- **Inicialização:** Realiza as operações padrões como alocação de memória, aquisição dos recursos, carregamento dos dados e assim por diante.
- **Entra no *loop* do jogo:** A execução do código entra no *loop* principal do jogo. Aqui é onde a ação começa e continua até o usuário sair do programa.
- **Adquire os sinais de entrada do jogador:** Nesta parte processam-se as entradas e as guardam para uso posterior pela lógica do jogo.
- **Processa Inteligência Artificial e lógica do jogo:** Aqui fica a maior parte do software. Inteligência artificial, física e lógica geral do jogo são executadas e o resultado é usado para processar o próximo quadro na tela.

- **Renderiza próximo quadro:** Com o resultado das entradas e processamento da lógica, pode-se gerar o próximo quadro da animação. Esta imagem é geralmente desenhada fora da tela (*back buffer*) de maneira que não se vê a renderização ocorrendo. Em seguida a imagem é rapidamente copiada para a tela, criando a ilusão de animação.
- **Sincroniza a tela:** Como o computador pode ter o resultado acelerado ou desacelerado pelo nível de complexidade exibido (por exemplo, 1000 objetos na tela têm maior impacto no processamento que do que apenas 10), a taxa de quadro vai variar, sendo necessários ajustes para que a saída se mantenha coerente.
- ***Loop*:** Retorna para o início do *loop* e realiza todo processo novamente.
- **Finalização:** É o fim do jogo, quando se retorna para o sistema operacional. Entretanto, antes disso é necessário liberar todos os recursos e "limpar" o sistema.

A listagem a seguir, embora não funcional, dá uma idéia de como uma verdadeira implementação se pareceria.

Quadro 1 - Implementação genérica do *loop* de um jogo

```
// Definição dos estados do loop do jogo
enum {INICIALIZANDO, MENU, INICIANDO, RODANDO, REINICIANDO, SAINDO};

// Variáveis globais
int estadoDoJogo = JOGO_INICIALIZANDO; // começa neste estado
int erro = 0; // usado para informar erros ao sistema operacional

// Função principal (main) começa aqui
int main() {
  // Implementação do loop principal do jogo
  while (estadoDoJogo != JOGO_SAINDO) {
    // Em que estado se encontra o loop do jogo
    switch (estadoDoJogo) {
      case JOGO_INICIALIZANDO: // O jogo está inicializando
        inicializa();
        estadoDoJogo = JOGO_MENU;
        break;
      case JOGO_MENU: // O jogo está no modo menu
        // Chama a função principal do menu e deixa-a trocar os estados
        estadoDoJogo = menu();
        // Poderia ser forçado um estado aqui
        break;
      case JOGO_INICIANDO: // o jogo está prestes a rodar
        // Este estado é opcional, mas é geralmente usado
        // para fazer os ajustes antes do jogo rodar de fato
        ajustaParaRodar();
        estadoDoJogo = JOGO_RODANDO;
        break;
      case JOGO_RODANDO: // o jogo está rodando
        // Esta seção contém todo lógica do jogo
```

```
        limparTela();
        pegarEntradas();
        fazerLogica();
        renderizarQuadro();
        esperar(); // Sincronização
        // O estado só muda por algum comando do usuário
        // (pela função pegarEntradas) ou perdendo...
        break;
      case JOGO_REINICIANDO: // o jogo irá reiniciar
        // Esta seção limpa os estados e variáveis para que o programa
        // rode novamente
        conserta();
        estadoDoJogo = JOGO_MENU;
        break;
      case JOGO_SAINDO: // o jogo está saindo
        liberaLimpaTudo();
        erro = 0;
        break;
      default: break;
    }
  }
  return(erro);
}
```

O código deixa claro que o *loop* do jogo se comporta como uma máquina de estado finito, com transição seqüencial entre os estados.

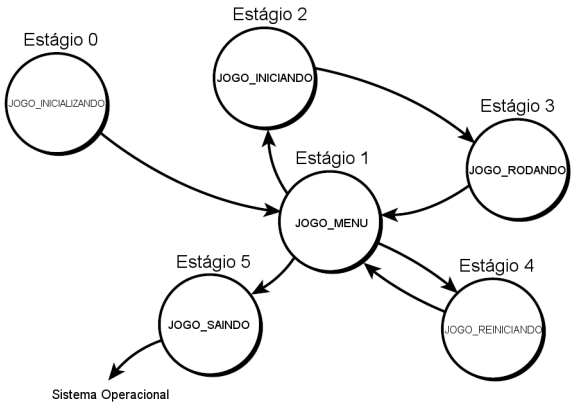

Figura 18 - Diagrama de transição de estado em um jogo

Desde *Ms Pac-Man* usa-se a estrutura de um jogo para fazer outro, porém foi com *Doom* que esta tendência se acentuou (DALMAU, 2004), sendo os jogos desenvolvidos com funcionalidades específicas para servirem de base para outros. Este tipo de estrutura é conhecida como o motor do jogo (*game engine*). Zerbst (2004) coloca:

> *Você pode definir um motor como uma máquina que propulsiona um veículo. Na programação de jogos, o motor é a parte de seu projeto que propulsiona certas funcionalidades do seu programa. É esperado que se você colocar a chave na ignição, o motor ligará e o carro dará a partida. Então você põe*

> *o pé no acelerador para fazer o carro mover. Dentro, o motor transfere energia cinética para o eixo. O motorista não precisa saber exatamente o que está acontecendo, nem mesmo se importa se ele apenas quer dirigir o carro.*
>
> *O mesmo conceito se aplica para motores 3D. Você chama a função Colocar_Chave_na_Ignicao() para dar a partida. Então o motor 3D deixa seu adaptador gráfico pronto para funcionar. Quando você pressionar o acelerador virtual (neste caso, mandando modelos 3D para o adaptador gráfico), o motor 3D faz os modelos 3D aparecerem na tela. O trabalho do motor é realizar as coisas desagradáveis de baixo nível, como se comunicar com o adaptador gráfico, ajustar seus estados de renderização, transformar os modelos, e tratar com a confusa matemática.*

Em sua opinião um motor deve:

- Gerenciar todos os dados de sua área de responsabilidade;
- Computar todos os dados de sua área de tarefa;
- Encaminhar todos os dados para sua instância seguinte, se necessário;
- Aceitar dados de instâncias precedentes para gerenciá-los e computá-los.

Assim, o motor de um jogo é na verdade um conjunto de motores integrados, cada um responsável por uma determinada área. Um jogo com gráficos tridimensionais certamente terá motor para som, motor para processamento de entrada, motor de física, motor para inteligência artificial, além do motor 3D.

O motor 3D em específico é o software que processa os dados do mundo tridimensional, incluindo luzes, ações, estados em geral e renderiza segundo o ponto de vista do jogador ou câmera. Não se trata apenas de enviar os dados para o adaptador gráfico, é necessário organizá-los de maneira que se obtenha velocidade, fazendo o chamado gerenciamento de objeto e cena. Embora a determinação de visibilidade seja atualmente realizada em hardware, através de algoritmo tipo *z-buffer*, nem mesmo a mais rápida das aceleradoras pode cuidar de uma cena contendo centenas de milhões de polígonos numa alta taxa de quadros. Assim, enviar todos os polígonos para serem gerenciados pelo hardware não é uma opção.

Dois passos são necessários para que uma cena seja reduzida de modo a ter complexidade aceitável. Primeiro, cortar tudo que estiver fora do campo de observação da câmera virtual (*frustum*). A remoção deve ser em grandes pedaços para que não seja necessário testar cada pequena parte da cena, demandando alto grau de processamento. Deste modo, deve haver algum esquema para agrupar

objetos, de preferência de maneira hierárquica. Os mais usados são *octrees*, *K-D trees* e principalmente *BSP trees*.

O segundo passo é remover tudo que for ocluso por outro objeto dentro do campo de observação. Os algoritmos mais utilizados são *potentially visible sets* (PVS) e portais.

3.2 Níveis de programação

Serão imaginados três níveis de programação de jogos: baixo, médio e alto.

No nível "mais baixo", trata-se com os hardwares e chamadas ao sistema operacional. Na verdade, exceto na programação para consoles de videogames, a diversidade de possíveis configurações obriga a trabalhar com um hardware abstrato, acessado por APIs. Por exemplo, acessa-se a aceleradora gráfica através das bibliotecas *OpenGL* e *Direct3D*, acessa-se o processador de som através das bibliotecas *OpenAL* e *DirectSound*, e assim por diante.

O nível intermediário é o motor do jogo em si. Ele oculta o nível mais baixo e cria o nível mais alto. No início, o código e o conteúdo dos jogos eram praticamente uma coisa só, com um dentro do outro. Posteriormente os jogos de computadores começaram a apresentar alguma organização interna, como por exemplo, o sistema SCUMM (*Script Creation Utility for Maniac Mansion*) da *LucasArts*, que possibilitava definir a estrutura do jogo (em termos de enredo), diálogos, locais e outros dados necessários para jogos de aventura (este sistema foi usado em *Maniac Mansion*, *Loom* e vários outros títulos até *Curse of Monkey Island*). Mas foi *Doom* que realmente expôs essa divisão, permitindo inclusive que os jogadores fizessem modificações no conteúdo[26], criando assim um novo jogo.

Existem soluções como *Java3D* e *OpenInventor*, que trabalham tanto no nível mais baixo quanto no intermediário, isto é, em alguns momentos se comportam como abstração do hardware, mas definem estruturas de alto nível.

Finalmente, no alto nível trabalha-se o conteúdo. Isto inclui todas as geometrias (mapas, objetos e personagens), texturas, sons, músicas e a programação ligada diretamente à estrutura do jogo.

---

[26] Os dados (conteúdo) do motor de *Doom* ficavam encapsulados num grande arquivo com extensão .wad e por isso conhecidos como arquivos WAD.
A modificação mais bem sucedida é o *Counter Strike*, feita a partir do jogo *Half-Life*, que por sua vez é baseado no motor de *Quake 2*.

Essa separação permite que hoje os times de desenvolvimento de jogos sejam comumente formados por artistas e programadores na proporção 80/20.

3.2.1 APIs

Nos últimos anos o foco do desenvolvimento gráfico na área dos jogos mudou exclusivamente da renderização via software para renderização através hardwares especializados para 3D. Inicialmente acessavam-se estes através de bibliotecas proprietárias que representavam um hardware específico, como por exemplo, o chip *VoodooGraphics*, que era exprimido pela API *Glide*. Com a diversificação dos modelos existentes, o uso passou a ser feito por padrões suportados por diferentes fabricantes, sendo hoje apenas duas bibliotecas realmente relevantes: *Direct3D* e *OpenGL*.

A escolha de uma API vai definir, além do conjunto de habilidades disponíveis, em que plataforma (hardware e sistema operacional) o software vai funcionar e como vai funcionar.

Quadro 2 - Resumo comparativo entre OpenGL e Direct3D

OpenGL	• Multiplataforma • Uso em diversos tipos de aplicações • Estabilidade da interface de programação
Direct3D	• Amplo suporte, incluindo *hardwares* de baixíssimo custo • Desenvolvido para sanar as necessidades dos jogos • Evolução mais dinâmica

OpenGL

No início da década de 90, a computação gráfica tridimensional – apesar de restrita às estações de trabalho de alto desempenho – já era utilizada, tanto para visualização quanto para efeitos especiais de cinema e televisão. Entretanto, cada fabricante oferecia sua biblioteca proprietária para produção de programas 3D, dificultando o aprendizado e limitando o uso da tecnologia.

Vendo o problema, a *Silicon Graphics* desenvolveu a biblioteca *OpenGL* (GL de *graphic library*, biblioteca gráfica aberta). Esta era baseada na sua biblioteca proprietária *IrisGL*, mas passava a ser governada por uma banca formada por diversos desenvolvedores, de modo a estar disponível (ter implementação) por

vários fabricantes. Desta forma passou a ser muito mais simples converter uma aplicação de uma plataforma para outra, pois os hardwares passaram a ser acessados por uma mesma interface de programação.

O principal objetivo da biblioteca é prover independência e ao mesmo tempo possibilitar acesso completo às funcionalidades do hardware. Ou seja, a API provê acesso às operações gráficas no nível mais baixo possível, mantendo autonomia. Como resultado, a *OpenGL* não provê meios para descrever ou modelar objetos geométricos complexos, ou em outras palavras, provê mecanismos para descrever como objetos geométricos complexos serão renderizados ao invés de mecanismos para descrever o objeto complexo em si.

Outra característica é o fato de somente prover acesso às operações de renderização. Por exemplo, não existem facilidades para obter entrada do usuário, pois é esperado que o sistema o faça.

O padrão de 1992 teve cinco revisões (até 1.5) e uma reestruturação mais profunda em 2004, chegando à versão 2.0. A compatibilidade com as versões anteriores é 100% mantida, sendo a principal inovação a capacidade de implementar seu próprio algoritmo de renderização usando uma linguagem de alto nível (GLSL).

Direct3D

Depois do lançamento do *Windows 95*, os jogos eram os únicos aplicativos que continuavam funcionando exclusivamente em DOS, simplesmente porque as capacidades em tempo real de áudio e vídeo do novo sistema eram terríveis. Para sanar parte do problema, a *Microsoft* criou o *Win-G*, que não se firmou como solução. Assim a empresa começou a desenvolver um novo conjunto de sistemas para acesso direto aos dispositivos de entrada, vídeo, áudio e rede: nasceu o *DirectX*. Ainda que as primeiras iterações não tenham convencido os programadores, a versão 3.0 oferecia desempenho superior às soluções programadas em *DOS32* e a versão 5.0 conquistou integralmente o mercado. Atualmente tem-se disponível a versão 9.

Direct3D é a parte do conjunto usada para renderizar gráficos tridimensionais. Mesmo que voltada para jogos, todos os tipos de aplicações que necessitam de 3D fazem uso da mesma.

Como a API é propriedade exclusiva da *Microsoft*, ela evoluiu – no sentido de adotar novos conceitos e apresentar novas habilidades – muito mais rapidamente

que a *OpenGL*, embora as mudanças nem sempre fossem as mais adequadas tecnicamente.

Há uma implementação padrão em software, de forma que quando o *driver* de um determinado hardware não é completo, o aplicativo busca socorro na primeira.

Assim como *OpenGL*, a última versão suporta linguagem de alto nível para colorização (neste caso, a linguagem HLSL).

A principal desvantagem é funcionar exclusivamente em *Windows* e na arquitetura *Intel x86*.

3.2.2 Middleware

Várias empresas oferecem motores de jogos como principal produto. Alguns nasceram a partir de títulos de sucesso enquanto outros foram desenvolvidos exclusivamente para terceirização. Conhecidos como *middlewares*, podem existir como soluções completas ou soluções específicas (destinadas a apenas um trabalho: gerar árvores, gerar terrenos, etc.)

Os *middleware*, além do motor, oferecem ferramentas adicionais para tornar o desenvolvimento mais direto. Jogos não podem ler arquivos do *3DStudio Max*, *Maya* ou *Softimage*, pois os mesmos não são otimizados para tempo real e para o próprio motor 3D. Assim esses pacotes oferecem programas para exportar as geometrias num formato adequado. Oferecem ainda ferramentas para análise e posterior otimização das geometrias e texturas exportadas.

Serão expostos brevemente apenas três pacotes como exemplos: *Renderware*, *NetImmerse* e *Unreal*.

***Renderware* (*Criterion Software*)**

A principal característica do *Renderware* é sua longevidade. Sendo usada por mais de 50 títulos, é uma tecnologia "de confiança" tendo em vista já ter sido colocada em prática de diversas maneiras. As várias versões cobrem praticamente todos os sistemas, com a atual suportando além do PC, os principais videogames.

A *Criterion* oferece o *Renderware Studio*, um ambiente integrado para criação de jogos multiplataforma. Os designers usam esta ferramenta para colocar modelos, animações, sons e códigos personalizados juntos no mesmo ambiente. O software

pode mostrar o ambiente criado nas quatro plataformas suportadas simultaneamente e ajustá-lo individualmente para cada uma delas.

- Linguagem: C.
- Arquitetura: Setores do mundo definem o conjunto de polígonos para geometrias estáticas; setores são organizados numa árvore BSP.
- Habilidades de renderização: Otimizado para geometrias *tri-stipping* e Bézier, mapas de iluminação, habilidades específicas do hardware como multitextura, *shaders* de *pixel* e sistema de partículas.
- Exportadores: *3DStudio Max* e *Maya* são totalmente suportados, incluindo janela com previsão no exportador.
- Ferramentas especiais: Mapeamento de iluminação, edição de setores do mundo para PVS, otimização de conteúdo para plataforma individual, animação hierárquica e não-hierárquica.
- Plataformas: PC (*MS Windows*), *Playstation 2*, *XBox*, *GameCube*.
- Exemplos de jogos: *Grand Theft Auto III*, *Tony Hawks Pro Skater 3*, *Sonic Heroes*.

***NetImmerse* (NDL)**

Desenvolvido exclusivamente para terceirização. A tecnologia apresentada pelos jogos que a utilizam tem impressionado tanto os jogadores quanto os especialistas na área (MCSHAFFRY, 2003).

- Linguagem: C++.
- Arquitetura: Grafo de cena hierárquico que pode receber subclasse, suportando instâncias.
- Habilidades de renderização: Clássico recorte baseado no campo de observação da câmera virtual, portais, multitextura, mapeamento de rugosidade, sistema de partículas, personagens podem "trocar de pele", detecção de colisão.
- Exportadores: *3DStudio Max* e *Maya* são totalmente suportados, incluindo janela com previsão no exportador.
- Ferramentas especiais: Otimização de grafos de cena, animação de personagens.
- Plataformas: PC (*Windows* e *Linux*), *Playstation 2*, *XBox*, *GameCube*.
- Exemplos de jogos: *Munch's Odyssey*, *Dark Age of Camelot*.

***Unreal* (*Epic*)**

Baseado na aclamada série de jogos *Unreal*, este motor 3D é provavelmente um dos sistemas de renderização mais avançados que existem (MCSHAFFRY, 2003). É um produto em contínuo desenvolvimento, tendo três grandes gerações e duas intermediárias: a primeira, usada no *Unreal* original; a 1.5, usada no *Unreal Tournament*, que além dos aprimoramentos adicionou suporte aos *videogames Playstation 2* e *Dreamcast*; a segunda geração, usada pela maioria dos títulos com esse motor e suporte adicional aos consoles *XBox* e *GameCube*; a 2.5, otimizada para *XBox*; e a terceira geração, em desenvolvimento com previsão para 2006 (com tecnologia semelhante, porém mais evoluída, a de *Doom 3*).

- Linguagem: C e C++ misturados.
- Arquitetura: Lista de cena (não um grafo) que pode receber subclasse, suportando instâncias.
- Habilidades de renderização: Clássico recorte baseado no campo de observação da câmera virtual, portais, multitextura, mapeamento de rugosidade, sistema de partículas, personagens podem "trocar de pele", detecção de colisão.
- Exportadores: *3DStudio Max* e *Maya* são totalmente suportados, incluindo janela com previsão no exportador.
- Ferramentas especiais: Animação de personagens.
- Plataformas: PC (*Windows*, *Linux*), *Playstation 2*, *XBox*, *GameCube*.
- Exemplos de jogos: A série *Unreal*, *Splinter Cell*, *Harry Potter and the Prisoner of Azkarban*.

A flexibilidade deste motor é comprovada pelo seu uso no VRND (reconstrução virtual em tempo real da Catedral de Notre Dame) (ROSA Jr., 2003).

3.2.3 Conteúdo

O contínuo refinamento dos motores permitiu a separação entre renderização e conteúdo (*scripts*, modelos 3D, texturas, mapas e ambientes).

É importante observar que não se trata apenas da "arte", boa parte da programação do jogo se define aqui, por meio de *scripts*[27]. A abrangência destes vai depender do motor. No *Unreal*, através do *UnrealScript*, que é baseado na linguagem de programação Java, é possível definir grande parte da lógica e inteligência artificial do jogo, componentes e funcionamento da interface gráfica do usuário, funcionalidades dos objetos no mundo virtual e o próprio comportamento desse mundo.

Em termos de artes, existem ferramentas específicas para cada tipo. Os mundos ou mapas, por estarem ligados a algum esquema de agrupamento (árvores BSP, por exemplo), são criados em ferramentas baseadas diretamente no motor. Novamente tomando como modelo o *Unreal*, este oferece um aplicativo chamado *UnrealEd*, que além de facilitar a criação de ambientes, garante melhor otimização do mapa para o jogo.

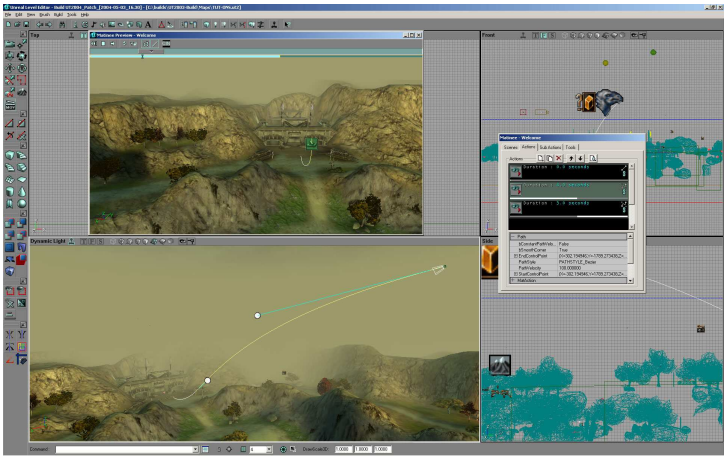

Figura 19 - *UnrealEd* 3.0 (*Epic Games*)

Para modelos 3D – construção de objetos e personagens – existem vários pacotes como solução. Os mais adotados e suportados, que possuem exportador ou

[27] Linguagens de programação baseadas em *scripts* conectam diversos componentes pré-existentes para cumprir determinada tarefa. Essas linguagens geralmente possuem como características: favorecimento do rápido desenvolvimento ao invés da eficiência de execução; são interpretadas ao invés de compiladas; têm forte conexão com componentes escritos em outras linguagens.

algum tipo de pré-visualização do resultado, são *3DStudio Max*, *Maya* (como visto na seção anterior) e *Softimage*.

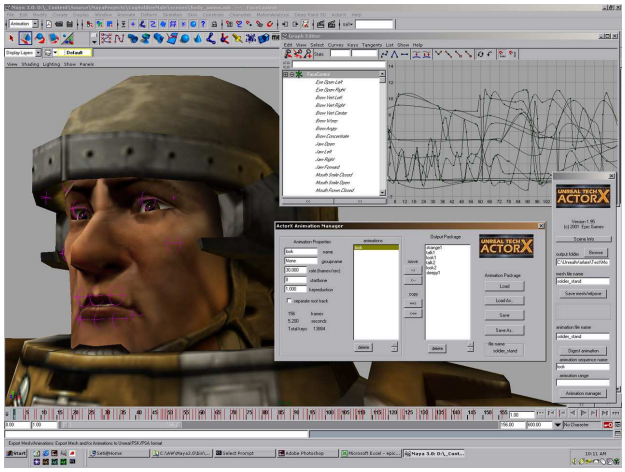

Figura 20 - *Maya* 3 (*Alias*)

Como as texturas são independentes do motor, a criação fica ao gosto do artista, desde que sejam respeitadas as limitações impostas por aquele (resolução, profundidade de cor, etc.). É interessante a disseminação do uso de texturas procedurais e linguagens para colorização de alto nível. Para criação desses *shaders*, a ferramenta mais utilizada é o *RenderMonkey*, que suporta tanto GLSL quanto HLSL.

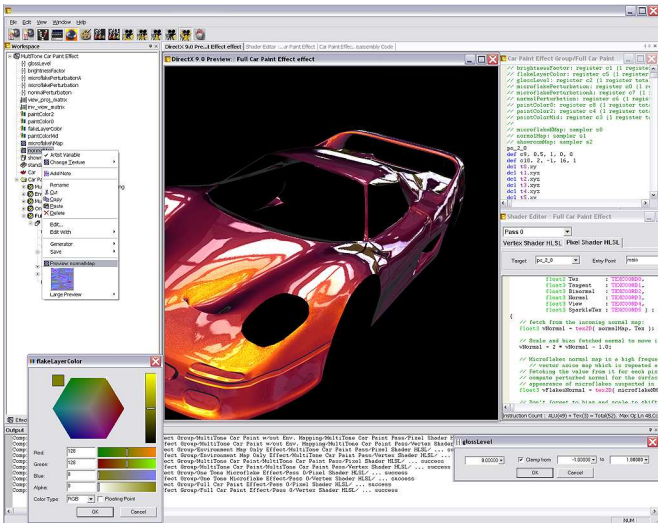

Figura 21 - *RenderMonkey* (*ATI Technologies*)

3.3 OpenGL

Durante a apresentação das técnicas, nas implementações de referência, será utilizado *OpenGL*. Existem três motivos:

1. **Mobilidade:** o fato da biblioteca ser multiplataforma, possibilitando que o código funcione em *Windows*, *Linux* (em diversas arquiteturas), *MacOS X* e estações gráficas da *Silicon Graphics* e *Sun*;
2. **Longevidade:** a API já tem mais de uma década de vida e os princípios permanecem basicamente os mesmos, indicando que os códigos não sofrerão obsolescência tão bruscamente como ocorrem com os escritos para *Direct3D*;
3. **Experiência do autor:** preferiu-se trabalhar com este padrão, pois os resultados se mostraram mais apropriados.

Será aproveitado este espaço para dar uma breve introdução a esta interface de programação, de maneira que os códigos apresentados nas técnicas não sejam exóticos em demasia. Uma exposição relativamente adequada iria requerer centenas de páginas e fugiria muito do escopo deste trabalho. Para correta explicação é recomendada a leitura de Davis (2003), Wright (1996) e Rost (2004).

OpenGL é uma linguagem gráfica procedural ao invés de descritiva. Em vez de descrever a cena e como ela deve parecer, o programador descreve os passos necessários para atingir essa cena e aparência. Estes "passos" envolvem chamadas a dezenas de comandos e funções que desenharão primitivas gráficas como pontos, linhas e polígonos em três dimensões (todas primitivas são descritas em termos de seus vértices). Adicionalmente, a *OpenGL* suporta iluminação e colorização, mapeamento de textura, animação e outros efeitos especiais.

Como dito anteriormente, a *OpenGL* não incluiu qualquer função para obter entrada do usuário (e assim interação), ou mesmo gerenciamento de janela ou entrada e saída de arquivo. Cada ambiente (*Windows*, *Linux*, etc.) tem suas funções para isto e as mesmas devem ser utilizadas.

Um quadrado seria descrito da seguinte forma:

Quadro 3 - Quadrado em OpenGL

```
{
  glColor3f (0.0, 0.0, 0.0);
  glBegin(GL_QUADS);
    glVertex3f(-1.0, 1.0, 0.0);
    glVertex3f(1.0, 1.0, 0.0);
```

```
    glVertex3f(1.0, -1.0, 0.0);
    glVertex3f(-1.0, -1.0, 0.0);
  glEnd();
}
```

OpenGL é uma máquina de estado, que uma vez colocada em determinados modos, permanece neles até que sejam explicitamente alterados. Na listagem anterior, definiu-se a cor do vértice sendo preta pelo comando glColor3f (0.0, 0.0, 0.0), e como não foi alterado, afetou todos os vértices do quadrado. A cor é apenas uma de muitas variáveis de estado mantidas. Outras incluem tipo de vista e transformação da projeção, padrão das conexões das linhas e polígonos, modo de desenho dos polígonos, convenção do empacotamento dos *pixels*, posição e características das luzes, propriedades do material do objeto sendo desenhado. Várias variáveis de estado são habilitadas ou desativadas pelos comandos *glEnable()* ou *glDisable()*.

Para conseguir um gráfico animado, desenha-se a imagem na tela, limpa-se a tela, desenha-se uma nova imagem levemente modificada, e assim por diante.

Quadro 4 - *Loop* para animação

```
while (estado != SAINDO) {
  limparTela(); // Apaga a imagem anterior
  pegarEntradas(); // Verifica se o usuário acionou algum comando
  fazerLogica(); // Atualiza os dados animação
  renderizarQuadro(); // Gera a nova imagem
}
```

A listagem completa comentada é exposta a seguir. Para gerenciamento da aplicação e obter sinais dos dispositivos de entrada é utilizada a biblioteca auxiliar SDL (SDL, 2004).

Quadro 5 - Quadrado animado

```
#include <SDL/SDL.h>
#include <gl/gl.h>

// Definição dos estados do loop.
enum {INICIALIZANDO, RODANDO, SAINDO};
// Variáveis globais
int estado = INICIALIZANDO; // Começa neste estado.
int erro = 0; // Usado para informar erros ao sistema operacional.
SDL_Event evento; // Usado para guardar as entradas do usuário.
float rotacao = 0.0; // Ângulo de rotação do quadrado; começa em 0 graus.

void inicializa() {
  // Esta função faz todo procedimento de inicialização.
  SDL_Init(SDL_INIT_VIDEO);
  SDL_SetVideoMode(640, 480, 0, SDL_OPENGL | SDL_HWSURFACE | SDL_NOFRAME);
```

```
  glViewport(0, 0, 640, 480); // Define um vista de 640x480.
  glClearColor(1.0, 1.0, 1.0, 1.0); // A tela será branca quando limpa.
  glMatrixMode(GL_PROJECTION); // Modificações na matrix de projeção.
  glLoadIdentity(); // Substitui a matriz atual pela matriz identidade.
  glOrtho(-4.0,4.0,-3.0,3.0,1.0,-1.0); // Multiplica a matriz por esta.
  glMatrixMode(GL_MODELVIEW); // Modificações na matrix de modelos.
  glColor3f (0.0, 0.0, 0.0);  // A cor do vértice será preta.
}

void limparTela() {
  glClear(GL_COLOR_BUFFER_BIT); // Limpa o buffer.
}

void pegarEntradas() {
  // Aqui faz-se processamento das entradas do usuário.
  // No caso, apenas a tecla "Esc" tem efeito (sair).
  SDL_PollEvent(&evento);
  if(evento.key.keysym.sym == SDLK_ESCAPE) { estado = SAINDO; }
}

void fazerLogica() {
  // A cada ciclo, o quadrado será girado em 0.1 graus.
  rotacao += 0.1;
  if (rotacao == 360.0) { rotacao = 0.0; }
}

void renderizarQuadro() {
  glLoadIdentity(); // Substitui a matriz atual pela matriz identidade.
  glRotatef(rotacao, 0.0, 0.0, 1.0); // Todos os vértices serão girados
                                     // em relação ao plano da tela.

  glBegin(GL_QUADS); // Define o quadrado.
    glVertex3f(-1.0, 1.0, 0.0);
    glVertex3f(1.0, 1.0, 0.0);
    glVertex3f(1.0, -1.0, 0.0);
    glVertex3f(-1.0, -1.0, 0.0);
  glEnd();

  SDL_GL_SwapBuffers(); // Copia a imagem do back buffer para a tela.
}

void liberaLimpaTudo() {
  SDL_Quit();
}

// Função principal (main) começa aqui
int main(int argc, char *argv[]) {
  // Implementação do loop principal
  while (estado != SAINDO) {
    // Em que estado se encontra o loop do jogo
    switch (estado) {
      case INICIALIZANDO:
        inicializa();
        estado = RODANDO;
        break;
      case RODANDO:
        limparTela();
        pegarEntradas();
        fazerLogica();
        renderizarQuadro();
        break;
```

```
      case SAINDO:
        liberaLimpaTudo();
        erro = 0;
        break;
      default: break;
    }
  }
  return(erro);
}
```

Como colocado, não se pretende “ensinar” OpenGL, apenas mostrar a aparência de um programa utilizando esta API. Porém, como os códigos estão expostos de forma estruturada e linear, espera-se que não haja grandes dificuldades no acompanhamento dos mesmos.

A partir dessa rápida introdução, passa-se a apresentar as técnicas investigadas para geração de nuvens virtuais.

4 NUVENS

4.1 Introdução

Nuvens são partes indissociáveis do céu. Sem elas, uma cena externa tem aparência extremamente falsa. Não é estranho então, que a grande maioria dos jogos tenha algum tipo de nuvem. Muitos vão além da estética; usam nuvens e mudança de clima para estabelecer o ambiente, o humor e o funcionamento da partida. Em *Metal Gear Solid 2* (*Konami Studio*), por exemplo, parte da ação se dá sobre forte chuva, criando um sentimento mais pesado e deprimente. Em simuladores de vôo, nuvens são necessárias para que haja um mínimo de realismo.

À distância, se observadas por algum tempo, percebe-se que as nuvens:

- Embora diversificadas, exibem grande similaridade nas suas estruturas;
- Atravessam o céu à medida que o vento as empurra;
- Mudam de forma e evoluem, algumas “nascem” e outras “morrem”, sendo que as pequenas sofrem transformações mais visíveis;
- Variam em quantidade, podendo o céu estar completamente limpo e azul, com algumas nuvens isoladas, ou totalmente coberto;
- Têm a aparência modificada pela quantidade, transitando de branco brilhante (céu azul) até cinza escuro (nublado);
- São iluminadas pelo sol de um lado e sombreadas do outro, variando muito com a alvorada e o poente;
- Flutuam em altitudes uniformes, formando uma camada. Às vezes, múltiplas camadas.

Quanto mais desses pontos forem simulados, menos recursos sobrarão para o resto do jogo. Desta forma, várias técnicas são empregadas para criar nuvens. Do modo mais simples: colocando uma imagem contendo o céu no fundo da tela; ao mais complexo: implementando um sistema de nuvens volumétricas com propriedades como densidade e temperatura, de maneira a fazer o jogador sentir a turbulência quando passa com seu avião.

4.2 Escolha dos efeitos gráficos

Ao longo da pesquisa realizada optou-se em estudar de maneira mais apropriada um único efeito. As nuvens são elementos importantes para o enriquecimento do ambiente. Em alguns casos – como simuladores de vôo – são o ponto central, mas mesmo em outras situações passam a sensação de integralidade ao mundo virtual, elemento essencial para imersão do usuário.

Existe uma diversidade de técnicas disponíveis para obtenção de nuvens, havendo soluções para hardware ou simplesmente software, para complexas simulações ou simplesmente como elemento decorativo.

Ademais, com pequenas alterações nas técnicas é possível produzir outros efeitos como fogo, fumaça, raios (através de texturas procedurais, *billboards*, partículas), oceanos (utilizando texturas procedurais para criar efeito cáustico), florestas (*billboards*), e mesmo sombras (com múltiplas texturas, muito semelhante a mapas de iluminação).

4.3 Categorização das técnicas

Uma comparação numérica não teria sentido; levaria facilmente a falsas conclusões, pois o desempenho das técnicas depende essencialmente das suas implementações e otimizações. Poderia ter alguma utilidade se fossem avaliadas diferentes implementações de uma mesma técnica; o que não é o caso. A implementação de referência serve apenas para isso – referência. Não é imposto forma, linguagem de programação, biblioteca auxiliar ou hardware, somente caminhos para atingir um determinado efeito.

Assim, a avaliação das técnicas é feita qualitativamente:

- Que tipo de equipamento é necessário? Pode funcionar de forma razoável em software (usando apenas a CPU) ou requer aceleradora gráfica? Com que características?
- De que maneira a técnica deve impactar na aplicação como um todo? A técnica necessita ser o ponto central da aplicação?
- A implementação da técnica exige conhecimento mais profundo sobre o efeito simulado? Em quais níveis de programação ela pode ser realizada?

O ponto final a ser levado em consideração é a obtenção do efeito em si e a possível validade científica deste efeito.

4.4 Skybox

O meio mais básico de colocar nuvens numa cena é estabelecer uma imagem de fundo fixa representando o céu. Alguns jogos de corrida valiam-se de uma única imagem que era "arrastada" para cima, para baixo, para esquerda e para direita conforme a posição do jogador na pista, pois a vista era limitada à direção do veículo, e o trajeto era pré-determinado.

Com a vista sem restrição para manter a sensação de um mundo realmente tridimensional, a imagem de fundo teve que se tornar ambiental, rodeando o observador. A técnica para implantar essa imagem é chamada *skybox*, também conhecida como mapeamento ambiental [28].

Skybox, literalmente "caixa do céu", é um cubo ou esfera com as normais voltadas para dentro (faces internas) e texturas cobrindo todos os polígonos que representam o céu.

No caso do cubo, seis imagens rodeiam o usuário que está no meio da caixa, e assim, não importa para onde olhe, sempre vê uma parte do cubo. A caixa é pintada sem qualquer tipo de iluminação, impossibilitando ver a junção das imagens e gerando um céu único; o jogador, assim, permanece inconsciente de que está dentro desse grande cubo.

Figura 22 - Montagem de uma *Skybox*

Uma *skybox* é desenhada como se fosse infinitamente grande: não importa o quão longe o usuário ande, ele continua cercado pela caixa. Dessa forma, apenas

[28] O nome mais correto é "mapeamento ambiental", mas os desenvolvedores costumam utilizar "*skybox*". O termo se refere a toda geometria que engloba o mundo virtual, desempenhando o papel de céu ou atmosfera. Existem, porém, termos específicos para determinadas geometrias, como *skydome* (domo, hemisfério) e *skyplane* (plano).

objetos que o jogador nunca poderá tocar parecem corretos, como nuvens ou montanhas distantes. É claro, se o movimento for limitado apenas a olhar ao redor, então a caixa pode ser usada até para mostrar objetos próximos.

O *loop* de criação de uma *skybox* simples, semelhante à figura anterior, tem o seguinte formato:

- Limpar o *buffer* de profundidade;
- Aplicar a transformação da câmera na matriz de visualização de modelo;
- Desabilitar o teste de profundidade e escrita no *buffer* de profundidade;
- Ajustar para posição central na matriz de visualização de modelo;
- Desenhar a *skybox*;
- Habilitar o teste de profundidade e escrita no *buffer* de profundidade;
- Desenhar o resto do mundo.

O teste de profundidade deve ser desativado, pois o céu é infinitamente distante e assim não afeta a visibilidade de nenhum outro objeto. Não é necessário limpar o *buffer* de cor.

Quadro 6 - Geração de uma *skybox*

```
// A skybox é criada a partir da função geraSkybox.
// Ela recebe 3 parâmetros: a largura, altura e espessura da caixa.
// Em geral é um cubo (a 3 iguais), mas vai depender de como foram
// produzidas as imagens.
void geraSkybox(float largura, float altura, float espessura) {
  // As variáveis x, y, z definem o centro da caixa.
  float x -= largura / 2;
  float y -= altura / 2;
  float z -= espessura / 2;

  // Aqui é gerada a caixa. Basicamente o trabalho é pegar a textura
  // específica da face e mapeá-la na mesma.

  // Face traseira.
  glBindTexture(GL_TEXTURE_2D, texturaTraseira);
  glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(x, y, z);
    glTexCoord2f(0.0, 1.0); glVertex3f(x, y+altura, z);
    glTexCoord2f(1.0, 1.0); glVertex3f(x+largura, y+altura, z);
    glTexCoord2f(1.0, 0.0); glVertex3f(x+largura, y, z);
  glEnd();

  // Face frontal.
  glBindTexture(GL_TEXTURE_2D, texturaFrente);
  glBegin(GL_QUADS);
    glTexCoord2f(1.0, 0.0); glVertex3f(x, y, z+espessura);
    glTexCoord2f(1.0, 1.0); glVertex3f(x, y+altura, z+espessura);
    glTexCoord2f(0.0, 1.0); glVertex3f(x+largura, y+altura, z+espessura);
    glTexCoord2f(0.0, 0.0); glVertex3f(x+largura, y, z+espessura);
  glEnd();

  // Face de baixo.
```

```
  glBindTexture(GL_TEXTURE_2D, texturaDeBaixo);
  glBegin(GL_QUADS);
    glTexCoord2f(1.0, 0.0); glVertex3f(x, y, z);
    glTexCoord2f(1.0, 1.0); glVertex3f(x, y, z+espessura);
    glTexCoord2f(0.0, 1.0); glVertex3f(x+largura, y, z+espessura);
    glTexCoord2f(0.0, 0.0); glVertex3f(x+largura, y, z);
  glEnd();

  // Face de cima
  glBindTexture(GL_TEXTURE_2D, texturaDeCima);
  glBegin(GL_QUADS);
    glTexCoord2f(1.0, 1.0); glVertex3f(x, y+altura, z);
    glTexCoord2f(1.0, 0.0); glVertex3f(x, y+altura, z+espessura);
    glTexCoord2f(0.0, 0.0); glVertex3f(x+largura, y+altura, z+espessura);
    glTexCoord2f(0.0, 1.0); glVertex3f(x+largura, y+altura, z);
  glEnd();

  // Face da esquerda
  glBindTexture(GL_TEXTURE_2D, texturaEsquerda);
  glBegin(GL_QUADS);
    glTexCoord2f(1.0, 0.0); glVertex3f(x, y, z);
    glTexCoord2f(0.0, 0.0); glVertex3f(x, y, z+espessura);
    glTexCoord2f(0.0, 1.0); glVertex3f(x, y+altura, z+espessura);
    glTexCoord2f(1.0, 1.0); glVertex3f(x, y+altura, z);
  glEnd();

  // Face da direita
  glBindTexture(GL_TEXTURE_2D, texturaDireita);
  glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(x+largura, y, z);
    glTexCoord2f(1.0, 0.0); glVertex3f(x+largura, y, z+espessura);
    glTexCoord2f(1.0, 1.0); glVertex3f(x+largura, y+altura, z+espessura);
    glTexCoord2f(0.0, 1.0); glVertex3f(x+largura, y+altura, z);
  glEnd();
}

void renderizarQuadro() {
  glLoadIdentity();
  gluLookAt(camera.posicao.x, camera.posicao.y, camera.posicao.z,
            camera.direcao.x, camera.direcao.y, camera.direcao.z,
            camera.topo.x, camera.topo.y, camera.topo.z);
  glDisable(GL_DEPTH_TEST);
  glPushMatrix();
    geraSkybox(400, 400, 400);
  glPopMatrix();
  glEnable(GL_DEPTH_TEST);
  SDL_GL_SwapBuffers();
}

void limparTela() {
  glClear(GL_DEPTH_BUFFER_BIT);
}
```

O quadro anterior deixa claro que a geração de uma *skybox* é simples e direta, com a real dificuldade presente apenas na criação de imagens que se encaixem perfeitamente.

Como dito no início do capítulo, *skybox* também é chamada de mapeamento ambiental (cúbico, no caso da caixa). A técnica foi desenvolvida em 1976 por Jim Blinn como um rápido método para calcular reflexões quando objetos são renderizados.

Os programas para modelagem 3D – *3DStudio Max*, *Maya*, *Softimage* – podem perfeitamente ser aplicados à produção de mapas ambientais, e assim, para formação de uma *skybox*. Como é de se esperar, serão necessárias seis imagens da cena a partir de um único ponto no espaço. A câmera deverá estar ajustada para originar imagens quadradas com um campo de visão (*field-of-view*) de 90º, tanto na horizontal quanto na vertical. Então, serão renderizadas as seis vistas, cada uma a 90º das seguintes: frente, esquerda, traseira, direita, em cima e em baixo. A disposição das imagens vai depender do motor do jogo; *Quake 2* utiliza um arranjo semelhante ao mapa ambiental cúbico do *3DStudio R4*:

Figura 23 - *Skybox* em *Quake 2*

São muito empregados também os programas para geração procedural de cenários, como *Terragen* – usado na série *Battlefield 1942* (*Digital Illusions CE*), nos dois *Serious Sam* (*Croteam*), no *Formula 1 2000* (*EA Sports*) – e *Bryce*.

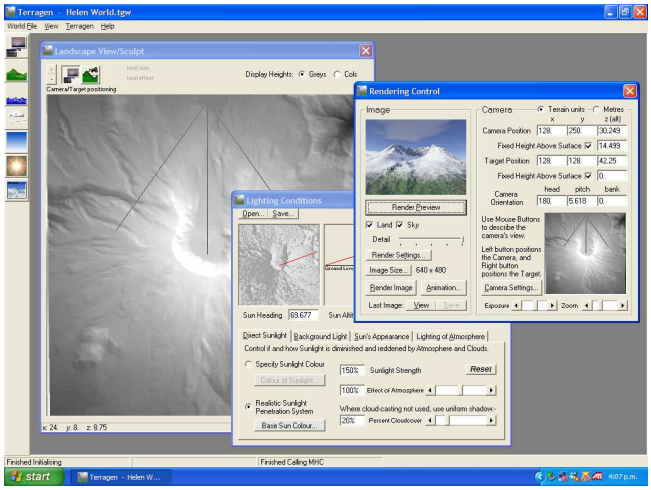

Figura 24 - *Terragen*

4.4.1 Múltiplas camadas

O jogo *Unreal*, de 1998, tinha proposta semelhante à de *Quake 2*. Diferentemente do último, o primeiro ocorria boa parte do tempo em ambientes abertos, enfrentando problemas distintos e demandando soluções diferenciadas. A *skybox* era uma delas; em *Quake 2*, o exterior era visto apenas pelas janelas dos corredores; em *Unreal*, o jogador, quando em ambiente aberto, estava todo instante observando o céu. Um efeito aplicado para que o meio parecesse mais real eram nuvens se movendo.

Para isso, são usadas múltiplas camadas, através de várias texturas ou faces no fundo da tela, numa técnica semelhante ao mapeamento de iluminação. A geometria cúbica de uma *skybox* não é recomendada, pois a animação torna óbvia a presença da caixa; assim, emprega-se um *skydome* (céu hemisférico) ou *skyplane* (plano curvado).

Tanto o hemisfério quanto o plano curvado só podem ser usados quando o terreno for até o horizonte, pois eles não cobrem 360º como no caso da *skybox*. Eles também consomem mais recursos, pois suas geometrias são muito mais complexas que no cubo. A vantagem, porém, é o grau de realismo potencialmente muito superior, permitindo o uso de neblina atingindo parte do cenário, pois na caixa a face inteira é apagada; mudança de cor individual dos vértices, possibilitando a simulação da iluminação do sol em diferentes horários do dia; e finalmente, o uso de múltiplas camadas, facilitando a construção de nuvens animadas.

Quadro 7 – Formação de um *skyplane*

```
// Função que gera o skyplane. As opções são:
// - O número de divisões do plano: quanto maior, mais detalhada e pesada;
// - O raio indicando o ponto mais alto da curvatura;
// - O raio indicando o quão esticado está o plano;
// - Mapeamento da textura na horizontal;
// - Mapeamento da textura na vertical.
void criaSkyplane(int divisoes, float raio1, float raio2,
                  float hTile, float vTile) {
  // Certifica que vetores estão limpos.
  if (vertices) {  // vertices é variável global.
    delete vertices;
    vertices = NULL;
  }
  if (indices) {  // indices é variável global.
    delete indices;
    indices = NULL;
  }
```

```
  // A curvatura é salva para centrar o céu.
  raio = raio1;  // raio é variável global.

  // Inicializa os vetores que conterão os vértices e índices.
  // numeroDeVertices e numeroDeIndices são variáveis globais.
  numeroDeVertices = (int)((divisoes + 1) * (divisoes + 1));
  numeroDeIndices  = (int)(divisoes * divisoes) * 2 * 3;
  vertices = new Vertice[numeroDeVertices];
  indices = new int[numeroDeIndices];

  // Calcula os vértices do skyplane
  float tamanhoDoPlano = 2.0*(float)sqrt((raio2 * raio2)-(raio1 * raio1));
  float delta = tamanhoDoPlano / (float)divisoes;
  float deltaTextura = 2.0f / (float)divisoes;
  float distanciaX   = 0.0f;
  float distanciaZ   = 0.0f;
  float alturaX = 0.0f;
  float alturaZ = 0.0f;
  float altura = 0.0f;
  Vertice verticeTemporario;
  for (int i=0; i <= divisoes; i++) {
    for (int j=0; j <= divisoes; j++) {
      distanciaX = (-0.5f * tamanhoDoPlano) + ((float)j * delta);
      distanciaZ = (-0.5f * tamanhoDoPlano) + ((float)i * delta);
      alturaX = (distanciaX * distanciaX) / raio2;
      alturaZ = (distanciaZ * distanciaZ) / raio2;
      altura = alturaX + alturaZ;
      verticeTemporario.x = distanciaX;
      verticeTemporario.y = 0.0f - altura;
      verticeTemporario.z = distanciaZ;
      verticeTemporario.u = hTile * ((float)j * deltaTextura * 0.5f);
      verticeTemporario.v = vTile * (1.0f - (float)i*deltaTextura*0.5f);
      vertices[i * (divisoes + 1) + j] = verticeTemporario;
    }
  }
  int indice = 0;
  for (int i = 0; i < divisoes; i++) {
    for (int j = 0; j < divisoes; j++) {
      int verticeInicial = (i * (divisoes + 1) + j);
      indices[indice++] = verticeInicial;
      indices[indice++] = verticeInicial + 1;
      indices[indice++] = verticeInicial + divisoes + 1;
      indices[indice++] = verticeInicial + 1;
      indices[indice++] = verticeInicial + divisoes + 2;
      indices[indice++] = verticeInicial + divisoes + 1;
    }
  }
}

// Desenha o skyplane a partir do vetor de vértices.
void desenhaSkyplane() {
  static float wrap = 0; // Variável usada para animar as nuvens.
  glActiveTextureARB(GL_TEXTURE0_ARB); // Ativa a textura 0 (céu).
  glEnable(GL_TEXTURE_2D);
  glBindTexture(GL_TEXTURE_2D,  textura[0]);
  glActiveTextureARB(GL_TEXTURE1_ARB); // Ativa a textura 1 (nuvens).
  glEnable(GL_TEXTURE_2D);
  glBindTexture(GL_TEXTURE_2D,  textura[1]);
    glPushMatrix();
      glTranslatef(0.0, raio, 0.0); // Centraliza o plano curvo.
```

```
      glBegin(GL_TRIANGLES);
        // Desenha todos os vértices.
        for (int i=0; i < numeroDeIndices; i++) {
          // Mapea a primeira textura.
          glMultiTexCoord2fARB(GL_TEXTURE0_ARB, vertices[indices[i]].u,
                               vertices[indices[i]].v);
          // Mapea a segunda textura.
          glMultiTexCoord2fARB(GL_TEXTURE1_ARB, vertices[indices[i]].u,
                               vertices[indices[i]].v - wrap);
          glVertex3f(vertices[indices[i]].x, vertices[indices[i]].y,
                     vertices[indices[i]].z);
        }
      glEnd();
    glPopMatrix();
  wrap += 0.0015f; // "Move" a textura das nuvens.
}

// Libera a memória utilizada pelos vetores.
// Função chamada a partir de liberaLimpaTudo.
void apagaSkyplane() {
  if (vertices) {
    delete vertices;
    vertices = NULL;
  }
  if (indices) {
    delete indices;
    indices = NULL;
  }
}


void renderizarQuadro() {
  glLoadIdentity();
  gluLookAt(camera.posicao.x, camera.posicao.y, camera.posicao.z,
            camera.direcao.x, camera.direcao.y, camera.direcao.z,
            camera.topo.x, camera.topo.y, camera.topo.z);
  glDisable(GL_DEPTH_TEST);
  desenhaSkyplane();
  glEnable(GL_DEPTH_TEST);
  SDL_GL_SwapBuffers();
}
```

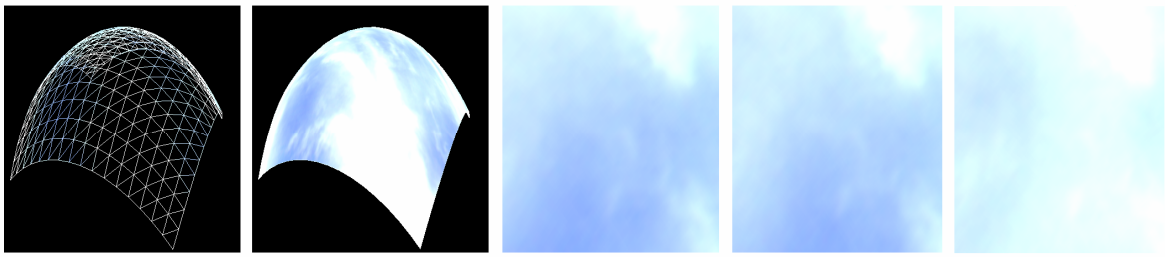

Figura 25 - *Skyplane* com múltiplas camadas e nuvens animadas

4.4.2 Nuvens procedurais

O uso de múltiplas camadas, ainda que satisfatório à primeira vista, rapidamente mostra sua natureza repetitiva quando observado por algum tempo. Uma solução é proposta por Pallister (2001), aplicando a geração procedural da textura das nuvens.

A idéia é baseada nas texturas procedurais de Perlin (1999), que imitam vários fenômenos e materiais naturais somando ruídos (números randômicos) de diferentes freqüências e amplitudes. A adaptação de várias etapas para serem realizadas pela aceleradora gráfica – suavização dos ruídos, interpolação das atualizações, combinação das oitavas – permitiu o uso da técnica em jogos.

Seis passos são necessários para se conseguir o efeito desejado:

1. Gerar os ruídos para as dadas oitavas[29] naquele ponto do tempo;
2. Suavizar os ruídos;
3. Interpolar os ruídos suavizados com a atualização anterior, para formar o quadro atual;
4. Compor todas as oitavas interpoladas numa única função de ruído turbulento;
5. Processar o ruído composto para que se pareça mais com nuvens;
6. Mapear a textura na geometria adequada.

O resultado final vai depender de como cada passo é realizado. O primeiro, a geração dos ruídos, é o mais importante, pois vai definir a qualidade das nuvens, a taxa de repetição, além de, em princípio, ser a etapa mais onerosa computacionalmente. Basicamente ela se resume numa função semeada pseudo-randômica[30] que gera o ruído. É necessário produzir ruídos em diferentes freqüências (oitavas), que serão representadas por texturas de diferentes resoluções (permitindo que sejam processados pela aceleradora). Cada oitava será responsável por certa parte da nuvem: as de freqüências mais baixas produzirão as partes maiores; as de freqüências mais altas construirão as partes menores.

Quadro 8 - Geração de ruídos

```
// Função geradora de ruído tridimensional
// (posição horizontal, vertical e tempo).
float ruido3d(int x, int y, int t) {
  // Baseado no trabalho de Hugo Elias:
  // http://freespace.virgin.net/hugo.elias/models/m_perlin.htm
```

[29] Oitava é um múltiplo inteiro de uma freqüência base. O nome é alusão às oitavas musicais, que possuem propriedade semelhante.

[30] Função semeada pseudo-randômica porque deve produzir o mesmo resultado para uma dada semente, de forma que seja possível reproduzir esse resultado.

```
  int n = x + y * 57 + t * 131;
  n = (n<<13) ^ n;
  return (1.0 - ((n * (n * n * 15731 + 789221) + 1376312589) & 0x7fffffff)
          * 0.000000000931322574615478515625);
}

// Desenha ruídos com a resolução da oitava desejada.
// O tempo é amplificado pela resolução para diminuir as semelhanças
// de dois desenhos consecutivos.
void desenhaRuido(int resolucao) {
  glDisable(GL_TEXTURE_2D);
  glBegin(GL_POINTS);
  for (int x = 0; x < resolucao; x++) {
    for (int y = 0; y < resolucao; y++) {
      // A cor é definida pela função ruído e é somada a 1 e divida por 2
      // para garantir valores positivos.
      float cor = ruido3d(x, y, tempo * resolucao);
      cor = (cor + 1) * 0.5;
      glColor3f(cor, cor, cor);
      glVertex2i(x, y);
    }
  }
  glEnd();
  glEnable(GL_TEXTURE_2D);
}
```

De maneira a tornar as texturas mais semelhantes com nuvens, é necessário suavizar os ruídos. Para minimizar o uso de recursos, basta aplicar um filtro bilinear sobre a textura, mapeando-a numa resolução maior que a sua. O processo pode ser repetido para obter suavização adicional, mas um único passe já oferece resultados satisfatórios.

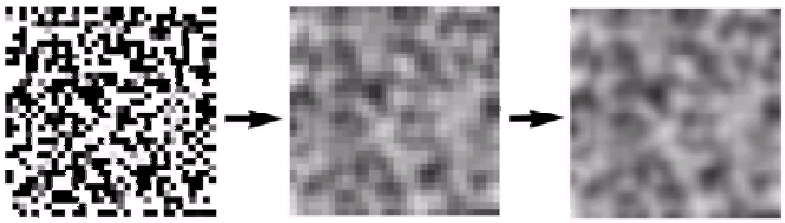

Figura 26 – Textura com oitava 32x32 suavizada e redimensionada para 256x256

Como se deseja movimento, cada textura é atualizada periodicamente. Quanto maior for a freqüência (neste exemplo, resolução), mais curto será o período de atualização, pois as nuvens menores sofrem transformações mais visíveis ao longo do tempo. Porém, a atualização deve ser suave, havendo uma transição entre a oitava anterior e a seguinte. Na implementação de referência usou-se uma simples interpolação linear:

- variação = tempo desde a última atualização / período de atualização

- cor atual = cor anterior * (1 – variação) + cor seguinte * variação

Como mencionado, as oitavas de freqüências mais baixas produzirão as partes maiores, e assim, as mais significantes das nuvens. Isso deve ser levado em consideração na combinação das oitavas. Por exemplo, uma geração baseada em quatro oitavas poderia ter a seguinte fórmula:

- cor = ½ oivata 1 + ¼ oitava 2 + 1/8 oitava 3 + 1/16 oitava 4

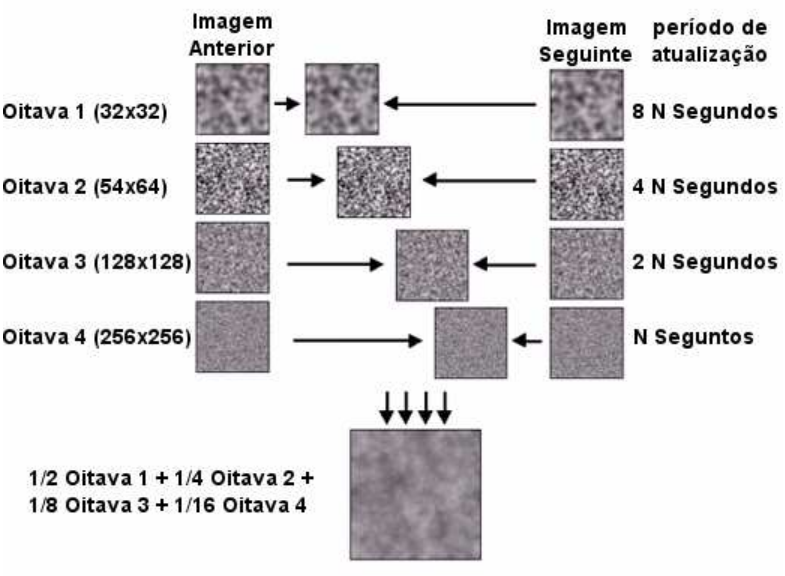

Figura 27 - Interpolação e composição das oitavas

Tendo as diferentes oitavas de um determinado tempo prontas e compostas, o passo final é tornar a imagem resultante mais parecida com nuvens (a imagem anterior mostra como o resultado se parece com fumaça). Como as nuvens são vapor d'água, abaixo de uma certa densidade elas não são visíveis; na textura resultante, todas as "densidades" são visíveis, sendo necessário remover as mais baixas. Porém, essa densidade depende de vários fatores como temperatura e pressão atmosférica, tornando a simulação, pelos menos em hardware, complicada. Para jogos, usa-se simplesmente um fator de corte.

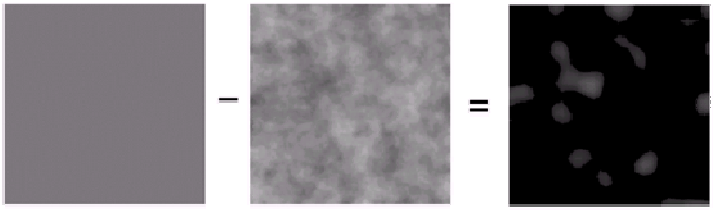

Figura 28 - Corte de ruídos para obtenção de nuvens isoladas

4.5 Face com textura

O uso exclusivo de *skybox* ou *skyplane* só é adequando quando o observador está sempre perto do chão, ou pelo menos, sempre longe das nuvens.

Em simuladores de vôo, por exemplo, deve se valer de uma solução adicional. Uma técnica bastante empregada, como no caso do *Flight Simulator 2002* e do *Combat Flight Simulator 3* (ambos da *Microsoft*), é desenhar a nuvem num *billboard*, de modo que cada nuvem possa se mover e reagir independentemente.

Billboard é uma face (polígono) texturizada sempre orientada segundo uma determinada vista. À medida que a vista muda, a orientação da face muda. Tem emprego em diversas situações:

- Gerar *sprites* em ambientes tridimensionais;
- Substituir geometrias por imagens, quando estas estão distantes do observador, reduzindo o número de polígonos a serem processados;
- Como elemento fundamental de sistemas de partículas, possibilitando de maneira simples a representação de vários fenômenos sem superfície sólida, como fumaça, fogo, explosões e auréola luminosa.

Fundamentalmente existem três tipos: alinhados à tela, orientados ao observador e axiais. No caso das nuvens, o *billboard* é orientado ao observador, distorcendo de maneira semelhante à geometria real.

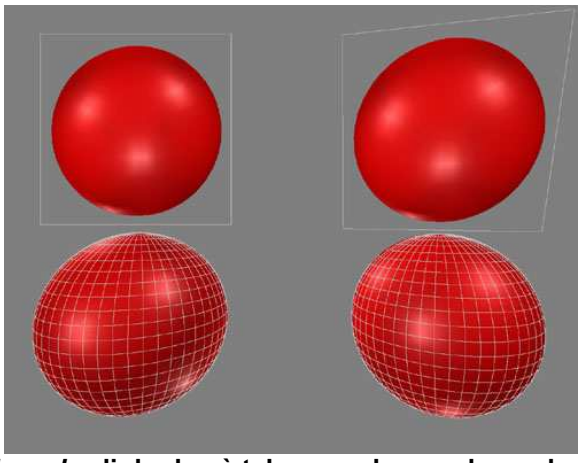

Figura 29 – *Billboards* alinhados à tela e ao observador, e duas esferas reais.

A maior parte do trabalho se resume em orientar a face de modo correto. Um determinado *billboard* deverá estar perpendicular à linha que une seu centro com a posição do observador. Por exemplo, considerando o eixo *x* indo da esquerda para direita da tela, o eixo *y* de baixo para cima, e o eixo *z* para fora, o *billboard* precisa

ser ajustado nos planos *xz* e *yz* (no plano *xy* as nuvens não giram com a câmera, pois o vetor que aponta o topo está sempre voltado para o "céu"):

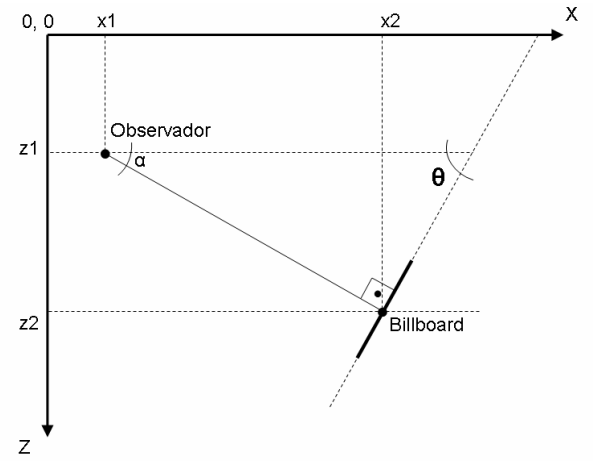

Figura 30 – Rotação do *billboard* na vista de topo (plano *xz*).

O ângulo θ representa a rotação necessária. Como a soma dos ângulos internos de um triângulo é 180º, θ + α + 90º = 180º ou θ = 90º - α, com $\alpha = \arctan\left(\frac{z2 - z1}{x2 - x1}\right)$.

Quadro 9 - Billboard alinhado ao observador

```
// A função desenha um billboard na posição (x, y, z)
// com a largura, a altura e a textura especificada.
void billboard(float x, float y, float z,
               float largura, float altura, int texturaID) {
  // No plano xz a rotação do billboard é determinada pela intersecção
  // da direção do polígono neste plano com a paralela ao eixo x
  // na posição z do observador.
  // O objeto observador é uma instância da classe Camera, detentora dos
  // membros posicao, direcao e topo; cada um possui os membros x, y, z.
  double catetoAdjacente = observador.posicao.x - x;
  double catetoOposto = observador.posicao.z - z;
  double tangente = catetoOposto / catetoAdjacente;
  // atan retorna o valor em radianos, necessitando conversão para graus.
  double anguloComplementar = atan(tangente) * 180.0 / M_PI;
  double anguloXZ = 90.0 - anguloComplementar;
  // O ângulo deve estar entre -90 e 90 graus,
  // impedindo que a face fique "ao contrário".
  if (anguloXZ > 90.0) { anguloXZ -= 180.0; }
  if (anguloXZ < -90.0) { anguloXZ += 180.0; }

  // No plano yz a rotação do billboard é determinada pela intersecção
  // da direção do polígono neste plano com a paralela ao eixo y
  // na posição z do observador.
  catetoAdjacente = y - observador.posicao.y;
  catetoOposto = z - observador.posicao.z;
  tangente = catetoOposto / catetoAdjacente;
  anguloComplementar = atan(tangente) * 180 / M_PI;
  double anguloYZ = 90 - anguloComplementar;
  if (anguloYZ > 90.0) { anguloYZ -= 180.0; }
  if (anguloYZ < -90.0) { anguloYZ += 180.0; }
```

```
  glEnable(GL_BLEND);
  glBlendFunc(GL_ONE, GL_ONE);
  glColor4f(1.0f,1.0f,1.0f,0.5f);
  glPushMatrix();
    glTranslatef(x, y, z);
    glRotatef(anguloXZ, 0, 1, 0);
    glRotatef(anguloYZ, 1, 0, 0);
    glBindTexture(GL_TEXTURE_2D, textura[texturaID]);
    glBegin(GL_QUADS);
      glVertex3f(-largura/2,  altura/2, 0.0);
      glVertex3f(-largura/2, -altura/2, 0.0);
      glVertex3f( largura/2, -altura/2, 0.0);
      glVertex3f( largura/2,  altura/2, 0.0);
    glEnd();
  glPopMatrix();
}
```

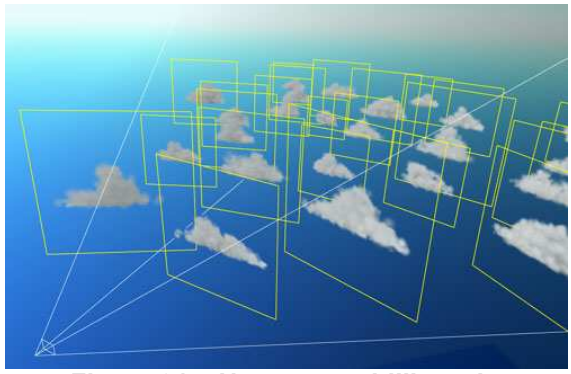

Figura 31 – Nuvens em *billboards*.

A principal desvantagem é o fato de não se poder atravessar a nuvem. Isto eliminaria a ilusão de volume, pois o usuário perceberia que se trata apenas de um plano.

4.6 Partículas e impostores

Quando o software precisa de uma representação mais realística, nuvens realmente tridimensionais devem ser empregadas. Entretanto, esse tipo deve ser cuidadosamente usado, uma vez que consome parte significante do processamento, reduzindo drasticamente o desempenho do aplicativo.

Duas áreas são estudadas para produção de nuvem: modelagem e renderização.

A modelagem de nuvens trata dos dados utilizados para representá-las no computador e como esses dados são originados e organizados. Destacam-se *voxels*, sistemas de partículas (REEVES, 1983), ruídos procedurais (PERLIN, 1985) e *metaballs* (DOBASHI, 2000).

A renderização é complicada em virtude da colorização, que, buscando aparência realística, requer a integração de efeitos das propriedades óticas ao longo do volume da nuvem e, ao mesmo tempo, incorporando a complexa refração da luz no meio. A maioria das técnicas foi desenvolvida para cenas não-interativas. Dobashi (2000) apresentou um método adequado para tempo real aplicando aproximação de uma única difração isotrópica. Harris (2002) o estendeu para múltiplas difrações seqüenciais e difração anisotrópica de primeira ordem.

Nos jogos ainda não há simulação da difração da luz, deixando boa parte da aparência por conta do artista. Entretanto, o trabalho de Harris é atualmente a maior referência para criação de nuvens em tempo real. Emprega uma mistura de sistema de partículas e impostores.

Sistema de partículas é um formalismo matemático usado para descrever fenômenos que são dinâmicos (temporais), compostos por pequenos componentes individuais e complexos. Por exemplo, nuvens, chuva, fogo, explosões. Consiste essencialmente num conjunto de simples primitivas (as partículas), cada uma com propriedades como posição, velocidade, cor e transparência, que mudam dinamicamente de acordo com alguma rotina estabelecida.

As partículas são representadas por pontos ou *billboards* alinhados à tela. Pode-se imaginar a nuvem do item anterior, todavia, formada por vários *billboards*, distribuídos não apenas no plano, mas nas três dimensões do espaço. Quanto menores e em maior número eles forem, mais detalhado será o objeto final, porém, resultando rapidamente em milhares de partículas para serem gerenciadas. Um meio de reduzir esse número é através de impostores.

Impostores são *billboards* que ao invés de usar como texturas imagens feitas por artistas, usam imagens geradas em tempo de execução, pelo sistema em si. A certa distância do observador, a nuvem formada por partículas é substituída por uma foto dela do quadro anterior. Dessa forma, dezenas de *billboards* serão substituídas por apenas um.

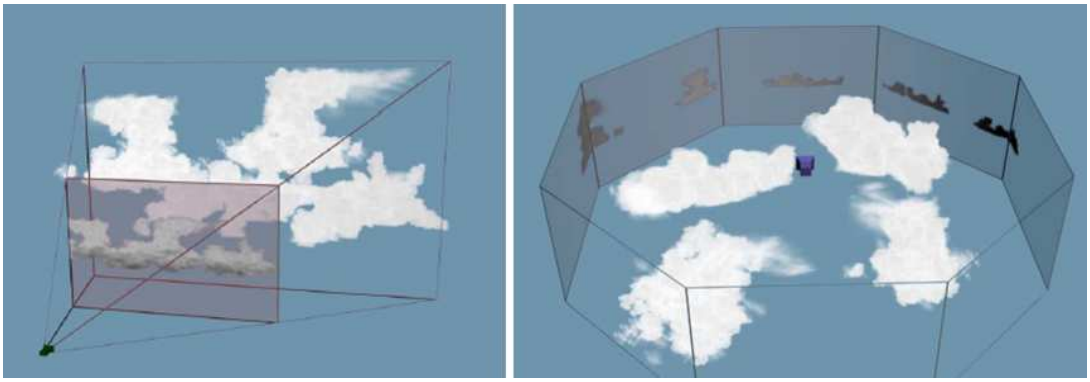

Figura 32 – Formação de impostores e uso a certa distância da câmera.

IL-2 Sturmovik (*Ubisoft*) usa partículas, mas não impostores. *Flight Simulator 2004* (*Microsoft*) utiliza partículas e impostores, porém com uma série de truques. Por exemplo, o posicionamento inicial das partículas é feito pelos artistas numa extensão (*plug-in*) para o *3DStudio Max*.

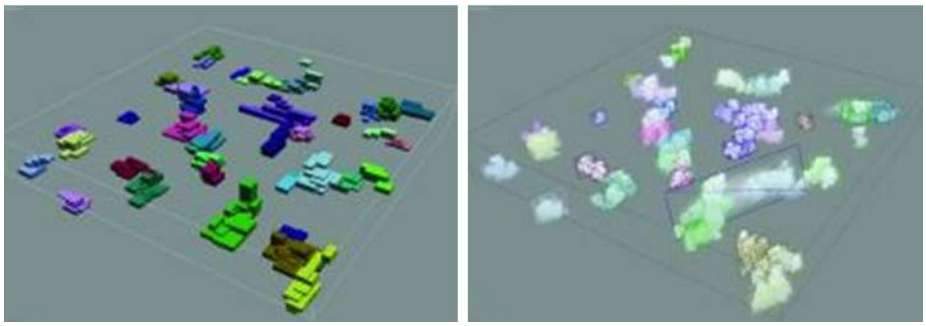

Figura 33 – *Flight Simulator 2004*: artistas posicionam blocos para criar nuvens.

4.7 Avaliação

Neste capítulo foi dito que a escolha do efeito se deu principalmente pela diversidade de técnicas disponíveis para obtenção do mesmo, o que foi corroborado, aqui, com os vários meios de se obter nuvens. No entanto, as técnicas apresentadas têm uma característica interessante: elas não são exclusivas, mas aditivas, possibilitando dosar realismo e desempenho. O uso de partículas para simular a massa gasosa não exclui o emprego de faces com texturas, pois impostores são um excelente meio para liberar recursos do sistema. Além disso, uma camada superior ou inferior de nuvens mais distantes pode ser representada por um *skyplane* com textura animada, e mais além, um *skybox* pode conter o inacessível céu e horizonte.

Como era de se esperar, grande parte do realismo gráfico não está na técnica ou implementação, mas nas mãos do artista que irá gerar o conteúdo (*skybox* e *billboards* convincentes dependem fundamentalmente das texturas).

Finalmente, é importante observar que não se trata apenas do aspecto visual, as técnicas são perfeitamente adaptáveis para simulações mais rigorosas. Nas texturas procedurais a geração dos ruídos e o processamento final da imagem podem ser baseados numa modelagem matemática e física mais adequada; o deslocamento e alteração dos *billboards* podem seguir uma mecânica fluída; o gerenciamento do sistema de partículas pode ser tal que simule perfeitamente nuvens de médio porte (*cumulus*).

4.7.1 Hardware necessário e recursos computacionais consumidos

Skybox e *skyplane* com múltiplas texturas funcionam adequadamente sem o auxílio de aceleração gráfica por hardware (como visto em *Quake 2* e *Unreal*). Com aceleração, o único cuidado é o consumo da memória de vídeo pelas texturas, visto que o uso dos recursos é praticamente irrisório. Uma aceleradora que aplica duas texturas num mesmo passe de renderização dispensa a duplicação da geometria do *skyplane*, tornando o processo ainda mais “leve”.

Se forem usadas texturas procedurais, hardware 3D passa a ser obrigatório. Além de múltiplas texturas em passe único, a construção dos ruídos vai exigir da aceleradora renderização fora da tela. Equipamentos mais modernos – com a linha de processamento gráfico programável – possibilitam que as etapas sejam mais diretas e rápidas, além de cálculos e resultados mais elaborados.

Billboards podem ser usados tanto em software quanto em hardware, embora a aceleração permita mais nuvens na cena e obtenha imagens muito mais convincentes, graças aos filtros nas texturas e transparências (*alpha blending*).

Infelizmente, o uso de partículas, mesmo valendo-se de impostores, ainda consome grande parte dos recursos, necessitando dos melhores equipamentos disponíveis. Hardwares modernos e a utilização de *shaders* melhoram o cenário, mas não permitem este tipo de nuvens em qualquer situação.

Quadro 10 – Resumo com as características das técnicas abordadas

Skybox	• Adequado para equipamentos antigos e modestos • Adequado para nuvens distantes e sem evidência • Consumo mínimo de recursos computacionais • Construção extremamente simples
Skyplane com múltiplas camadas	• Adequado para nuvens distantes • Baixo consumo de recursos computacionais • Construção simples
Skyplane com texturas procedurais	• Adequado para nuvens distantes e em destaque • *Hardware* com aceleração 3D obrigatório • *Hardware* preferencialmente com suporte a *shaders* • Necessita conhecimentos específicos do programador
Billboards	• Adequado para nuvens próximas, mas que não podem ser atravessadas • Baixo consumo de recursos computacionais • Construção relativamente simples
Partículas	• Adequado para nuvens próximas, inclusive quando atravessadas • Consumo intenso dos recursos computacionais • Necessita conhecimentos específicos do programador

4.7.2 Nível de especialização

Skybox, *skyplane* e *billboards* podem ser construídos no nível mais alto de programação, pelos *scripts*. Partículas, impostores e texturas procedurais necessitam trabalhar no nível mais baixo para obter desempenho adequado.

Partículas e texturas procedurais também vão requerer maior conhecimento da pessoa que irá implementá-los, tanto em termos de programação (por ser no nível mais baixo) quanto na modelagem física das nuvens (como elas se formam, como elas se dissipam, etc).

5 CONCLUSÕES E RECOMENDAÇÕES PARA TRABALHOS FUTUROS

5.1 Conclusões

Sobre este trabalho, destacam-se duas características. A primeira é a aparente ausência de público alvo: em muitos momentos fica demasiadamente superficial para conhecedores do assunto; em outros, demasiadamente técnico para leigos (em especial, as implementações). O resultado não é inesperado; foi um caminho conscientemente tomado, de modo a permitir sua utilização pelos variados profissionais que trabalham com mídia e conhecimento, possibilitando aos iniciantes um primeiro contato com o assunto – ainda que seja necessária uma segunda leitura – e aos mais familiarizados uma lista de referências nos tópicos abordados.

Com certo pesar do autor, a segunda característica é que o trabalho se mostrou muito curto, haveria muito para explorar. O desejo seria estudar outros tipos de efeitos (oceanos, geração de terrenos, sombras), ou mesmo, pesquisar ainda mais a fundo as técnicas apresentadas. Porém, o texto seria cada vez mais específico, afastando-se da área de pesquisa, sem necessariamente ajudar na questão investigada:

– É possível utilizar técnicas de jogos para efeitos visuais em aplicações de realidade virtual? Há ganhos? De que tipos? Em quais níveis?

Pelo menos no caso das nuvens, as técnicas são perfeitamente utilizáveis, podendo ser adaptadas para simulações mais rigorosas quando for o caso. Ou seja, independente do tipo e do grau de simulação, é interessante estudar a construção do efeito nos jogos.

O ganho real, entretanto, é justamente quando se pode aplicar as mesmas simplificações feitas nos jogos, possibilitando a aglutinação de vários efeitos visuais e resultando numa cena mais completa. Por exemplo, nuvens volumétricas, se considerada toda mecânica fluída e a difração da luz, esgotam os recursos disponíveis para o software; a aplicação é a nuvem e ponto. Um *skyplane* com múltiplas texturas tem uso praticamente irrisório dos recursos, permitindo colocar o sol, halos luminosos – quando ele é visualizado diretamente –, terrenos mais complexos, grama, árvores, etc.

A possível desvantagem do uso das técnicas de jogos é que elas são fortemente voltadas para os artistas. Neste sentido, a qualidade do resultado final fica diretamente atrelada à qualidade do conteúdo (do modelo geométrico, da textura, do *shader*) e muitos projetos não têm profissionais qualificados na área.

A validação da utilização de técnicas ratifica o uso de motores de jogos em aplicativos de realidade virtual, o que vários projetos já exercem. A parte mais crítica da simulação científica é adaptada dentro do motor, quando o código deste é aberto, ou é implementada através de *scripts*, no nível mais alto da aplicação.

Isso reforça a idéia que a fronteira de separação entre jogos e realidade virtual será cada vez mais tênue; os jogos como um caso especial de aplicativos de realidade virtual, ou talvez, a realidade virtual como um caso especial de jogo.

5.2 Recomendações para trabalhos futuros

A dissertação deixa aberta várias questões e possibilidades de exploração.

A pesquisa mais óbvia é o estudo de outros tipos de efeitos – sombras e iluminação, geração de terrenos, líquidos – nos mesmos moldes deste trabalho.

Tanto essas novas pesquisas quanto a deste documento devem ser um processo continuado de estudo, sendo periodicamente atualizadas, pois o campo do desenvolvimento dos jogos está em constante transformação. É perfeitamente plausível que se refaça este mesmo trabalho daqui cinco anos, e obtenham-se resultados inéditos.

Outro campo que já está sendo bem explorado, mas que ainda permite estudo, é o gerenciamento de cena em jogos e a construção de motores em geral para as diversas categorias (simuladores, tiro em primeira pessoa, cenários fechados, ambientes externos).

Por conseguinte, um trabalho interessante seria a construção de um motor de jogo genérico voltado para aplicações de realidade virtual que, de mesma forma como o *Renderware*, poderia ser usado para diversas situações.

Enfim, um trabalho importantíssimo é o estabelecimento de um projeto conjunto entre várias entidades (universidades, centros de pesquisa, pequenas produtoras, comunidade de software livre) para criação de repositório de conteúdo, em especial, modelos de objetos e personagens, além de texturas de alta qualidade, para que os

softwares de realidade virtual possam, de fato, apresentar resultados gráficos equivalentes aos dos atuais jogos.

6 REFERÊNCIAS

AUKSTAKALNIS, S.; BLATNER, D.; ROTH, S. F. **Silicon mirage: the art and science of virtual reality**. Berkeley, CA: Peachpit Press, 1992.

BARON, L.; ROBERT, J.M.; THÉRIAULT, L. **Virtual reality interfaces for virtual environments**. In *Proceeding of IEEE VRIC 2004*, ISTIA Innovaton, p. 109-117, Laval, France, 2004.

BRYSON, S. **Knowledge-Based Augmented Reality**. *Communications of the ACM*, Ed. 7, Vol. 26, p. 56-62, 1993.

BURDEA, G. C. **Force and touch feedback for virtual reality**. New York, USA: John Wiley and Sons, 1996. 339p.

BURDEA, G.; COIFFET, P. **Virtual reality technology**. New York, NY: John Wiley & Sons, 1994.

CRUZ-NEIRA, C.; SANDIN, D. J.; DEFANTI, T. A. **Surround-screen projection-based virtual reality: the design and implementation of the CAVE**. In *Proceeding of SIGGRAPH 93 Computer Graphics Conference*, ACM SIGGRAPH, p. 125-142, Anahein, CA, 1993.

DALMAU, D. S. C. **Core Techniques and Algorithms in Game Programming**. Indianapolis, IN, USA: New Riders Publishing, 2003. 888p.

DAMER, B. **Avatar! : exploring and building virtual worlds on the Internet**. Berkeley, CA: Peachpit Press. 1998.

DAVIS, T.; NEIDER, J.; SHREINER, D. **OpenGL programming guide: the official guide to learning OpenGL, version 1.4**. Boston, MA, USA: Addison Wesley, 2003. 800p.

DIFEDE, J.; HOFFMAN, H. G. **Virtual reality exposure therapy for World Trade Center post-traumatic stress disorder: a case report**. In CyberPsychology & Behavior, Vol. 5, N. 6, 2002. p. 529-535.

DOBASHI, Y.; KANEDA, K.; YAMASHITA, H.; OKITA, T.; NISHITA, T. **A simple, efficient method for realistic animation of clouds**. In *Proceeding of SIGGRAPH 2000 Computer Graphics Conference*, ACM SIGGRAPH, p. 19-28, New Orleans, Louisiana, 2000.

FULLERTON, T.; SWAIN, C.; HOFFMAN, S. **Game design workshop: designing, prototyping, and playtesting games**. San Francisco, CA, USA: CMP Books, 2004. 480p.

GERVAUTZ, M.; MAZURYK, T. **Virtual reality: history, applications, technology and future**. In *Technical Report TR-186-2-96-06, Vienna University of Technology,*

Institute of Computer Graphics and Algorithms, 1996. Disponível em <http://www.cg.tuwien.ac.at/research/TR/96/TR-186-2-96-06Abstract.html>. Acesso em 30 nov. 2004.

GIBILISCO, S. **The McGraw-Hill Illustrated Encyclopedia of Robotics & Artificial Intelligence**. New York, NY, USA: TAB Books, 1994. 420p.

GIBSON, W. **Neuromancer**. São Paulo: Aleph, 2003. 304p.

HADWIGER, M. **Design and architecture of a portable and extensible multiplayer 3D game engine**. 2000. 150f. Dissertação – *Institute of Computer Graphics, Vienna University of Technology*, Vienna, 2000.

HARRIS, M. J.; LASTRA, A. **Real-time cloud rendering**. In *Proceeding of EUROGRAPHICS 2001*, Vol. 20, N. 3, p. 76-84, 2001.

HEILIG, M. **Enter the experimental revolution**. In *Proceeding of Cyberarts Conference*, Pasadena, 1992, p. 292-305.

HEILIG, M. **The cinema of the future**. In *Multimedia: From Wagner to Virtual Reality*. New York, NY, EUA: W.W. Norton & Company, 2001. cap 22, p. 239-251.

HOFFMAN, H. G.; PATTERSON, D. R.; CARROUGHER, G. J.; SHARAR, S. R. **The effectiveness of virtual reality based pain control with multiple treatments**. *Clinical Journal of Pain*, n 17, p 229-235. 2001.

HOFFMAN, H. G.; RICHARDS, T.; CODA, B.; RICHARDS, A.; SHARAR, S. R. **The illusion of presence in immersive virtual reality during an fMRI brain scan**. In CyberPsychology & Behavior, Vol. 6, N. 2, 2003. p. 127-131.

HOUAISS, **Dicionário Houaiss da língua portuguesa**. Rio de Janeiro: Objetiva, 2001. 3008p.

JOHNSTON, V. S. **Why we feel: the science of human emotions**. New York, NY: Perseus Books Group, 2000. 210p.

LAMOTHE, A. **Tricks of the 3D game programming gurus**. Indianapolis, Indiana: Sams Publishing, 2003. 1728p.

LATTA, J. N.; OBERG, D. J. **A conceptual virtual reality model**. IEEE Computer Graphics & Applications, 14(1): p. 23-29, Jan. 1994.

LEAL, C.; TOLEDO, F. F. de, **Realidade virtual na indústria aeronáutica: caso Embraer**. In *SBC 4th Symposium on Virtual Reality - SVR 2001*, Florianópolis, SC, 2001.

LUZ, R. P. da. **Desenvolvimento de sistemas de realidade virtual**. In *XII Escola Regional de Informática ERI 2004*, CCT/UDESC, p. 93-123, Joinville, SC, 2004.

LUZ, R. P. da. **Proposta de especificação de uma plataforma de desenvolvimento de ambientes virtuais de baixo custo**. 1997. 108f. Dissertação (Mestrado em Engenharia de Produção , Inteligência Aplicada) – Programa de Pós-Graduação em Engenharia de Produção, Universidade Federal de Santa Catarina, Florianópolis, 1997.

MCSHAFFRY, M. **Game coding complete**. Scottsdale, Arizona: Paraglyph Press, 2003. 600p.

MINE, M. **Towards virtual reality for the masses: 10 years of research at disney's VR studio**. In *IPT / EGVE 2003: Seventh Immersive Projection Technology Workshop and Ninth Eurographics Workshop on Virtual Environments*. p. 11-18, Zurich, Switzerland, 2003.

PALLISTER, K. **Generating procedural clouds using 3D hardware**. In *Game programming gems 2*, Hingham, Massachusetts: Charles River Media, 2001. p. 463-473.

PERLIN, K. **An image synthesizer**. In *Proceeding of SIGGRAPH 1985 Computer Graphics Conference*, ACM SIGGRAPH, p. 287-296, San Francisco, California, 1985.

PIMENTEL, K.; TEIXEIRA, K. **Virtual reality – through the new looking glass**. 2. Ed. New York: MacGraw-Hill, 1995.

REEVES, W.; BLAU, R. **Approximate and probabilistic algorithms for shading and rendering structured particle systems**. In *Proceeding of SIGGRAPH 1985 Computer Graphics Conference*, ACM SIGGRAPH, p. 312-322, San Francisco, California, 1985.

ROSA Jr., O. **LRVChat3D, desenvolvimento de um ambiente virtual tridimensional multiusuário para Internet**. 2003. 110f. Dissertação (Mestrado em Engenharia de Produção) – Programa de Pós-Graduação em Engenharia de Produção, Universidade Federal de Santa Catarina, Florianópolis, 2003.

ROST, R. J. **OpenGL shading language**. Boston, MA, USA: Addison Wesley, 2004. 608p.

RÖTHLEIN, B. **Crashes which don't leave any dents**. BMWMagazine, 2003, p. 84-87.

SDL, **Simple DirectMedia Layer**. Disponível em <http://www.libsdl.org>.
Acesso em 30 nov. 2004.

SGI, **BMW and Reality Centers**. Disponível em <http://mfile.akamai.com/7772/rm/sgi.download.akamai.com/7772//Industries/Manufacturing/BMW/BMW_300.rpm>.
Acesso em 30 nov. 2004.

STUART, R. **The Design of Virtual Environments**. New York, NY, USA: McGraw Hill, 1996. 274p.

SUTHERLAND, I. **A head-mounted thee dimensional display**. In *Fall Join Computer Conference 1968*, Thompson Books, Washington, DC, p. 757-764, 1968.

SUTHERLAND, I. **The ultimate display**. NY: IFIP, p. 506-508, 1965.
VRMC. **Virtual Reality Medical Center**. Disponível em <http://www.vrphobia.com> Acesso em 30 nov. 2004.

VALIENT, M. **Accelerated real-time rendering**. 2003. 82f. Dissertação – *department of computer graphics, faculty of mathematics, physics and informatics, Comenius University*, Bratislava, 2003.

VENKATASUBRAMANIAN, S. **The graphics card as a stream computer**. In *SIGMOD-DIMACS Workshop on Management and Processing of Data Streams*, 2003. Disponível em < http://www.research.att.com/~suresh/papers/mpds/mpds.pdf> Acesso em 30 nov 2004.

VIEIRA, R. de S. **Protótipo de um sistema de monitoramento remoto inteligente**. 1999. 147f. Dissertação (Mestrado em Engenharia de Produção, Inteligência Aplicada) – Programa de Pós-Graduação em Engenharia de Produção, Universidade Federal de Santa Catarina, Florianópolis, 1999.

WRIGHT, R. S.; SWEET, M. **Opengl superbible: the complete guide to OpenGL programming for Windows NT and Windows 95**. Waite Group Press, 1996. 750p.
ZERBST, S. **3D game engine programming**. Boston, MA, USA: Course Technology Crisp, 2004. 896p.

7 APÊNDICES

7.1 Apêndice A – Implementações de referência

7.1.1 skybox.cpp

```
#include <SDL/SDL.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include "camera.h"
#include "textura.h"

// Definição dos estados do loop do jogo.
enum {INICIALIZANDO, RODANDO, SAINDO};
// Numeração das imagens da skybox.
enum {FRENTE, ESQUERDA, TRASEIRA, DIREITA, EMCIMA, EMBAIXO};

// Variáveis globais
const int LARGURA_DA_TELA = 800;
const int ALTURA_DA_TELA = 600;
const int COR_DA_TELA = 32;
int estado = INICIALIZANDO; // Começa neste estado.
int erro = 0; // Usado para informar erros ao sistema operacional.
Camera camera(0, 0, 0,  0, 0, -6,  0, 1, 0,
              LARGURA_DA_TELA, ALTURA_DA_TELA);
unsigned int textura[6] = {0};
SDL_Event evento;

// A skybox é criada a partir da função geraSkybox.
// Ela recebe 3 parâmetros: a largura, altura e espessura da caixa.
// Em geral é um cubo (a 3 iguais), mas vai depender de como foram
// produzidas as imagens.
void geraSkybox(float largura, float altura, float espessura) {
  // As variáveis x, y, z definem o centro da caixa.
  float x = -largura  / 2.0;
  float y = -altura / 2.0;
  float z = -espessura / 2.0;

  // Aqui é gerada a caixa. Basicamente o trabalho é pegar a textura
  // específica da face e mapeá-la na mesma.
  // Face frontal.
  glBindTexture(GL_TEXTURE_2D, textura[FRENTE]);
  glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(x, y, z);
    glTexCoord2f(1.0, 0.0); glVertex3f(x+largura, y, z);
    glTexCoord2f(1.0, 1.0); glVertex3f(x+largura, y+altura, z);
    glTexCoord2f(0.0, 1.0); glVertex3f(x, y+altura, z);
  glEnd();

  // Face da direita
  glBindTexture(GL_TEXTURE_2D, textura[DIREITA]);
  glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(x+largura, y, z);
    glTexCoord2f(1.0, 0.0); glVertex3f(x+largura, y, z+espessura);
    glTexCoord2f(1.0, 1.0); glVertex3f(x+largura, y+altura, z+espessura);
    glTexCoord2f(0.0, 1.0); glVertex3f(x+largura, y+altura, z);
  glEnd();
```

```
  // Face traseira.
  glBindTexture(GL_TEXTURE_2D, textura[TRASEIRA]);
  glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(x+largura, y, z+espessura);
    glTexCoord2f(1.0, 0.0); glVertex3f(x, y, z+espessura);
    glTexCoord2f(1.0, 1.0); glVertex3f(x, y+altura, z+espessura);
    glTexCoord2f(0.0, 1.0); glVertex3f(x+largura, y+altura, z+espessura);
  glEnd();

  // Face da esquerda
  glBindTexture(GL_TEXTURE_2D, textura[ESQUERDA]);
  glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(x, y, z+espessura);
    glTexCoord2f(1.0, 0.0); glVertex3f(x, y, z);
    glTexCoord2f(1.0, 1.0); glVertex3f(x, y+altura, z);
    glTexCoord2f(0.0, 1.0); glVertex3f(x, y+altura, z+espessura);
  glEnd();

  // Face de cima
  glBindTexture(GL_TEXTURE_2D, textura[EMCIMA]);
  glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(x, y+altura, z);
    glTexCoord2f(1.0, 0.0); glVertex3f(x+largura, y+altura, z);
    glTexCoord2f(1.0, 1.0); glVertex3f(x+largura, y+altura, z+espessura);
    glTexCoord2f(0.0, 1.0); glVertex3f(x, y+altura, z+espessura);
  glEnd();

  // Face de baixo.
  glBindTexture(GL_TEXTURE_2D, textura[EMBAIXO]);
  glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(x, y, z+espessura);
    glTexCoord2f(1.0, 0.0); glVertex3f(x+largura, y, z+espessura);
    glTexCoord2f(1.0, 1.0); glVertex3f(x+largura, y, z);
    glTexCoord2f(0.0, 1.0); glVertex3f(x, y, z);
  glEnd();
}


void liberaLimpaTudo() {
  glDeleteTextures(6, textura);
  SDL_Quit();
}

void renderizarQuadro() {
  glLoadIdentity();
  gluLookAt(camera.posicao.x, camera.posicao.y, camera.posicao.z,
            camera.direcao.x, camera.direcao.y, camera.direcao.z,
            camera.topo.x, camera.topo.y, camera.topo.z);
  glDisable(GL_DEPTH_TEST);
  geraSkybox(400, 400, 400);
  glEnable(GL_DEPTH_TEST);
  // Todo resto do mundo virtual seria desenhado aqui.
  SDL_GL_SwapBuffers();
}

// Aqui faz-se processamento das entradas do usuário.
// "Esc" sai; mouse rotaciona a câmera.
void pegarEntradas() {
  while(SDL_PollEvent(&evento)) {
    switch (evento.type) {
```

```
      case SDL_QUIT:
        estado = SAINDO;
        break;
      case SDL_KEYDOWN:
        if(evento.key.keysym.sym == SDLK_ESCAPE) { estado = SAINDO; }
        break;
      case SDL_MOUSEMOTION:
        camera.ajustaDirecaoComMouse();
        break;
      default: break;
    }
  }
}

void limparTela() {
  glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
}

// Esta função faz todo procedimento de inicialização.
void inicializa() {
  // Inicialização da SDL, responsável pela interface entre o sistema e a
OpenGL.
  SDL_Init(SDL_INIT_VIDEO);
  SDL_GL_SetAttribute( SDL_GL_DOUBLEBUFFER, 1 );
  SDL_GL_SetAttribute( SDL_GL_DEPTH_SIZE, COR_DA_TELA);
  SDL_GL_SetAttribute( SDL_GL_STENCIL_SIZE, 0);
  SDL_GL_SetAttribute( SDL_GL_ACCUM_RED_SIZE, 0);
  SDL_GL_SetAttribute( SDL_GL_ACCUM_GREEN_SIZE, 0);
  SDL_GL_SetAttribute( SDL_GL_ACCUM_BLUE_SIZE, 0);
  SDL_GL_SetAttribute( SDL_GL_ACCUM_ALPHA_SIZE, 0);
  SDL_WM_SetCaption("Skybox", "Skybox");
  SDL_SetVideoMode(LARGURA_DA_TELA, ALTURA_DA_TELA, COR_DA_TELA,
                   SDL_OPENGL | SDL_HWSURFACE | SDL_HWACCEL);
  SDL_EnableKeyRepeat(100, SDL_DEFAULT_REPEAT_INTERVAL);
  SDL_WM_GrabInput(SDL_GRAB_ON);
  SDL_ShowCursor(SDL_DISABLE);

  // Ajustes iniciais da OpenGL.
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_TEXTURE_2D);
  glEnable(GL_CULL_FACE);
  glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);

  glViewport(0, 0, LARGURA_DA_TELA, ALTURA_DA_TELA);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0f,(GLfloat)LARGURA_DA_TELA/(GLfloat)ALTURA_DA_TELA,
                 .5f ,5000.0f);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();


  // Carregamento das texturas da skybox, realizada através de textura.h.
  // Se não for possível abrir a imagem, erro será igual a 1.
  erro = criaTextura(textura, "traseira.tga", TRASEIRA);
  // Para não aparecer emendas nas imagens do cubo, as texturas devem estar
  // configuradas para não repitir (GL_CLAMP_TO_EDGE).
  texturaNaoRepete();
  erro = criaTextura(textura, "frente.tga", FRENTE);
  texturaNaoRepete();
  erro = criaTextura(textura, "embaixo.tga", EMBAIXO);
```

```
  texturaNaoRepete();
  erro = criaTextura(textura, "emcima.tga", EMCIMA);
  texturaNaoRepete();
  erro = criaTextura(textura, "esquerda.tga", ESQUERDA);
  texturaNaoRepete();
  erro = criaTextura(textura, "direita.tga", DIREITA);
  texturaNaoRepete();
}

// Função principal (main) começa aqui
int main(int argc, char *argv[]) {
  // Implementação genérica do loop principal de jogo
  while (estado != SAINDO) {
    switch (estado) {
      case INICIALIZANDO:
        inicializa();
        estado = RODANDO;
        break;
      case RODANDO:
        limparTela();
        pegarEntradas();
        renderizarQuadro();
        break;
      case SAINDO:
        liberaLimpaTudo();
        break;
      default: break;
    }
  }
  return erro;
}
```

7.1.2 skyplane.cpp

```
#include <SDL/SDL.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glext.h>
#include <cmath>
#include "camera.h"
#include "textura.h"

// Definição dos estados do loop do jogo.
enum {INICIALIZANDO, RODANDO, SAINDO};

struct Vertice {
      float x,y,z;
      unsigned int color;
      float u, v;
};

// Variáveis globais
PFNGLMULTITEXCOORD2FARBPROC glMultiTexCoord2fARB = NULL;
PFNGLACTIVETEXTUREARBPROC glActiveTextureARB = NULL;
const int LARGURA_DA_TELA = 800;
const int ALTURA_DA_TELA = 600;
const int COR_DA_TELA = 32;
int estado = INICIALIZANDO; // Começa neste estado.
int erro = 0; // Usado para informar erros ao sistema operacional.
```

```
Camera camera(0, 0, 0,  0, 0, -6,  0, 1, 0,
              LARGURA_DA_TELA, ALTURA_DA_TELA);
unsigned int textura[2] = {0};
SDL_Event evento;
Vertice* vertices;
int numeroDeVertices;
int* indices;
int numeroDeIndices;
float raio;

// Retorna o quadrado do número
float quadrado(float x) {
  float resultado = x * x;
  return resultado;
}

// Função que gera o skyplane. As opções são:
// - O número de divisões do plano: quanto maior, mais detalhada e pesada;
// - O raio indicando o ponto mais alto da curvatura;
// - O raio indicando o quão esticado está o plano;
// - Mapeamento da textura na horizontal;
// - Mapeamento da textura na vertical.
void criaSkyplane(int divisoes, float raio1, float raio2,
                  float hTile, float vTile) {
  // Certifica que vetores estão limpos.
  if (vertices) {
    delete vertices;
    vertices = NULL;
  }
  if (indices) {
    delete indices;
    indices = NULL;
  }

  // A curvatura é salva para centrar o céu.
  raio = raio1;

  // Inicializa os vetores que conterão os vértices e indices.
  numeroDeVertices = (int)quadrado(divisoes + 1);
  numeroDeIndices  = (int)quadrado(divisoes) * 2 * 3;
  vertices = new Vertice[numeroDeVertices];
  indices = new int[numeroDeIndices];

  // Calcula os vértices do skyplane
  float tamanhoDoPlano = 2.0f * (float)sqrt((quadrado(raio2) -
                                             quadrado(raio1)));
  float delta = tamanhoDoPlano / (float)divisoes;
  float deltaTextura = 2.0f / (float)divisoes;
  float distanciaX   = 0.0f;
  float distanciaZ   = 0.0f;
  float alturaX = 0.0f;
  float alturaZ = 0.0f;
  float altura = 0.0f;
  Vertice verticeTemporario;
  for (int i=0; i <= divisoes; i++) {
    for (int j=0; j <= divisoes; j++) {
      distanciaX = (-0.5f * tamanhoDoPlano) + ((float)j * delta);
      distanciaZ = (-0.5f * tamanhoDoPlano) + ((float)i * delta);
      alturaX = (distanciaX * distanciaX) / raio2;
      alturaZ = (distanciaZ * distanciaZ) / raio2;
      altura = alturaX + alturaZ;
```

```
      verticeTemporario.x = distanciaX;
      verticeTemporario.y = 0.0f - altura;
      verticeTemporario.z = distanciaZ;
      verticeTemporario.u = hTile * ((float)j * deltaTextura * 0.5f);
      verticeTemporario.v = vTile * (1.0f-(float)I * deltaTextura * 0.5f);
      vertices[i * (divisoes + 1) + j] = verticeTemporario;
    }
  }
  int indice = 0;
  for (int i = 0; i < divisoes; i++) {
    for (int j = 0; j < divisoes; j++) {
      int verticeInicial = (i * (divisoes + 1) + j);
      indices[indice++] = verticeInicial;
      indices[indice++] = verticeInicial + 1;
      indices[indice++] = verticeInicial + divisoes + 1;
      indices[indice++] = verticeInicial + 1;
      indices[indice++] = verticeInicial + divisoes + 2;
      indices[indice++] = verticeInicial + divisoes + 1;
    }
  }
}

// Desenha o skyplane a partir do vetor de vértices.
void desenhaSkyplane() {
  static float wrap = 0; // Variável usada para animar a textura da nuvens.
  // Ativa a textura 0, que contendo o céu.
  glActiveTextureARB(GL_TEXTURE0_ARB);
  glEnable(GL_TEXTURE_2D);
  glBindTexture(GL_TEXTURE_2D,  textura[0]);
  // Ativa a textura 1, contendo as nuvens.
  glActiveTextureARB(GL_TEXTURE1_ARB);
  glEnable(GL_TEXTURE_2D);
  glBindTexture(GL_TEXTURE_2D,  textura[1]);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
  glPushMatrix();
    glTranslatef(0.0, raio, 0.0); // Centraliza o plano.
    glBegin(GL_TRIANGLES);
      // Desenha todos os vértices.
      for (int i=0; i < numeroDeIndices; i++) {
        // Mapea a primeira textura.
        glMultiTexCoord2fARB(GL_TEXTURE0_ARB, vertices[indices[i]].u,
                             vertices[indices[i]].v);
        // Mapea a segunda textura.
        glMultiTexCoord2fARB(GL_TEXTURE1_ARB, vertices[indices[i]].u-wrap,
                             vertices[indices[i]].v);
        glVertex3f(vertices[indices[i]].x, vertices[indices[i]].y,
                   vertices[indices[i]].z);
      }
    glEnd();
  glPopMatrix();
  wrap += 0.0005f; // Move a textura das nuvens.
}

// Libera a memória utiliza pelos vetores.
// Função chamada a partir de liberaLimpaTudo.
void apagaSkyplane() {
  if (vertices) {
    delete vertices;
    vertices = NULL;
  }
  if (indices) {
```

```
    delete indices;
    indices = NULL;
  }
}

void liberaLimpaTudo() {
  apagaSkyplane();
  glDeleteTextures(2, textura);
  SDL_Quit();
}

void renderizarQuadro() {
  glLoadIdentity();
  gluLookAt(camera.posicao.x, camera.posicao.y, camera.posicao.z,
            camera.direcao.x, camera.direcao.y, camera.direcao.z,
            camera.topo.x, camera.topo.y, camera.topo.z);
  desenhaSkyplane();
  // Todo resto do mundo virtual seria desenhado aqui.
  SDL_GL_SwapBuffers();
}

// Aqui faz-se processamento das entradas do usuário.
// "Esc" sai; mouse rotaciona a câmera.
void pegarEntradas() {
  while(SDL_PollEvent(&evento)) {
    switch (evento.type) {
      case SDL_QUIT:
        estado = SAINDO;
        break;
      case SDL_KEYDOWN:
        if(evento.key.keysym.sym == SDLK_ESCAPE) { estado = SAINDO; }
        break;
      case SDL_MOUSEMOTION:
        camera.ajustaDirecaoComMouse();
        break;
      default: break;
    }
  }
}

void limparTela() {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}

// Esta função faz todo procedimento de inicialização.
void inicializa() {
  // Inicialização da SDL, responsável pela interface entre
  // o sistema e a OpenGL.
  SDL_Init(SDL_INIT_VIDEO);
  SDL_GL_SetAttribute( SDL_GL_DOUBLEBUFFER, 1 );
  SDL_GL_SetAttribute( SDL_GL_DEPTH_SIZE, COR_DA_TELA);
  SDL_GL_SetAttribute( SDL_GL_STENCIL_SIZE, 0);
  SDL_GL_SetAttribute( SDL_GL_ACCUM_RED_SIZE, 0);
  SDL_GL_SetAttribute( SDL_GL_ACCUM_GREEN_SIZE, 0);
  SDL_GL_SetAttribute( SDL_GL_ACCUM_BLUE_SIZE, 0);
  SDL_GL_SetAttribute( SDL_GL_ACCUM_ALPHA_SIZE, 0);
  SDL_WM_SetCaption("Skybox", "Skybox");
  SDL_SetVideoMode(LARGURA_DA_TELA, ALTURA_DA_TELA, COR_DA_TELA,
                   SDL_OPENGL | SDL_HWSURFACE | SDL_HWACCEL);
  SDL_EnableKeyRepeat(100, SDL_DEFAULT_REPEAT_INTERVAL);
  SDL_WM_GrabInput(SDL_GRAB_ON);
```

```
  SDL_ShowCursor(SDL_DISABLE);

  // Ajustes iniciais da OpenGL.
  glActiveTextureARB = (PFNGLACTIVETEXTUREARBPROC)
                       SDL_GL_GetProcAddress("glActiveTextureARB");
  glMultiTexCoord2fARB = (PFNGLMULTITEXCOORD2FARBPROC)
                         SDL_GL_GetProcAddress("glMultiTexCoord2fARB");
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_TEXTURE_2D);
  glEnable(GL_CULL_FACE);
  glViewport(0, 0, LARGURA_DA_TELA, ALTURA_DA_TELA);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0f,(GLfloat)LARGURA_DA_TELA/(GLfloat)ALTURA_DA_TELA,
                 .5f ,500.0f);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

  // Carregamento das texturas da skybox, realizada através de textura.h.
  // Se não for possível abrir a imagem, erro será igual a 1.
  erro = criaTextura(textura, "fundo.tga", 0);
  erro = criaTextura(textura, "nuvens.tga", 1);
  criaSkyplane(16, 100.0f, 900.0f, 1.0f, 1.0f);
}

// Função principal (main) começa aqui
int main(int argc, char *argv[]) {
  // Implementação genérica do loop principal de jogo
  while (estado != SAINDO) {
    switch (estado) {
      case INICIALIZANDO:
        inicializa();
        estado = RODANDO;
        break;
      case RODANDO:
        limparTela();
        pegarEntradas();
        renderizarQuadro();
        break;
      case SAINDO:
        liberaLimpaTudo();
        break;
      default: break;
    }
  }
  return erro;
}
```

7.1.3 nuvem_procedural.cpp

```
#include <SDL/SDL.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glext.h>
#include <cmath>
#include <iostream>
#include "camera.h"
#include "textura.h"
```

```
// Definição dos estados do loop do jogo.
enum {INICIALIZANDO, RODANDO, SAINDO};

struct Vertice {
      float x,y,z;
      unsigned int color;
      float u, v;
};

// Variáveis globais
PFNGLMULTITEXCOORD2FARBPROC glMultiTexCoord2fARB = NULL;
PFNGLACTIVETEXTUREARBPROC glActiveTextureARB = NULL;
const int LARGURA_DA_TELA = 800;
const int ALTURA_DA_TELA = 600;
const int COR_DA_TELA = 32;
int estado = INICIALIZANDO; // Começa neste estado.
int erro = 0; // Usado para informar erros ao sistema operacional.
Camera camera(0, 0, 0,  0, 0, -6,  0, 1, 0,
              LARGURA_DA_TELA, ALTURA_DA_TELA);
unsigned int textura[12] = {0};
SDL_Event evento;
Vertice* vertices;
int numeroDeVertices;
int* indices;
int numeroDeIndices;
float raio;
float taxa1 = 0.0, taxa2 = 0.0, taxa3 = 0.0;
float t1, t2, t3;

void criaTexturaVazia(int tamanho, int canais, int tipo, int identificacao)
{
  unsigned int* texturaTemporaria = NULL;
  texturaTemporaria = new unsigned int [tamanho * tamanho * canais];
  memset(texturaTemporaria, 0,
         tamanho * tamanho * canais * sizeof(unsigned int));
  glGenTextures(1, &textura[identificacao]);
  glBindTexture(GL_TEXTURE_2D, textura[identificacao]);
  glTexImage2D(GL_TEXTURE_2D, 0, canais, tamanho, tamanho,
               0, tipo, GL_UNSIGNED_INT, texturaTemporaria);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
  delete[] texturaTemporaria;
}

// Copia uma textura em outra. Solução extremamente lenta,
// mas independente de extensão OpenGL.
void copiaTextura(int resolucao, int texturaDeOrigem, int texturaDeDestino)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glViewport(0, 0, resolucao, resolucao);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(0, resolucao, 0, resolucao, 0, 1);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glBindTexture(GL_TEXTURE_2D, textura[texturaDeOrigem]);
  glBegin(GL_QUADS);
    glTexCoord2f(0.0f, 0.0f);    glVertex2f(0.0, 0.0);
    glTexCoord2f(1.0f, 0.0f);    glVertex2f(resolucao, 0.0);
    glTexCoord2f(1.0f, 1.0f);    glVertex2f(resolucao, resolucao);
    glTexCoord2f(0.0f, 1.0f);    glVertex2f(0.0, resolucao);
```

```
  glEnd();
  glBindTexture(GL_TEXTURE_2D,textura[texturaDeDestino]);
  glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 0, 0,
                   resolucao, resolucao, 0);
  glViewport(0, 0, LARGURA_DA_TELA, ALTURA_DA_TELA);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0f,(GLfloat)LARGURA_DA_TELA/(GLfloat)ALTURA_DA_TELA,
                 0.5f ,5000.0f);
  glMatrixMode(GL_MODELVIEW);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}

// Função geradora de ruído.
// Baseado no trabalho de Hugo Elias:
// http://freespace.virgin.net/hugo.elias/models/m_perlin.htm
float ruido3d(int x, int y, int t) {
  int n = x + y * 57 + t * 131;
  n = (n<<13) ^ n;
  return (1.0f - ((n * (n * n * 15731 + 789221) + 1376312589) & 0x7fffffff)
          * 0.000000000931322574615478515625f);
}

// Desenha os ruídos com a resolução da oitava desejada.
// O tempo é amplificado pela resolucao para diminuir as semelhanças
// de dois desenhos consecutivos.
void desenhaRuido(int resolucao) {
      glDisable(GL_TEXTURE_2D);
  glBegin(GL_POINTS);
  for (int x = 0; x < resolucao; x++) {
    for (int y = 0; y < resolucao; y++) {
      float cor = ruido3d(x, y, (int)(SDL_GetTicks() * 10 * resolucao));
      cor = (cor + 1) * 0.5;
      glColor3f(cor, cor, cor);
      glVertex2i(x, y);
    }
  }
      glEnd();
      glEnable(GL_TEXTURE_2D);
}

// Criação de uma determinada oitava. Indica-se a resolução desejada,
// e em que imagem ela será copiada.
void geraOitava(int resolucao, int numeroDaTextura) {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glViewport(0, 0, resolucao, resolucao);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(0, resolucao, 0, resolucao, 0, 1);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  desenhaRuido(resolucao);
  glBindTexture(GL_TEXTURE_2D,textura[numeroDaTextura]);
  glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 0, 0,
                   resolucao, resolucao, 0);
  glViewport(0, 0, LARGURA_DA_TELA, ALTURA_DA_TELA);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0f,(GLfloat)LARGURA_DA_TELA/(GLfloat)ALTURA_DA_TELA,
                 0.5f ,5000.0f);
  glMatrixMode(GL_MODELVIEW);
```

```
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}

void desenhaQuadrado(float l, int texturaID) {
  glBindTexture(GL_TEXTURE_2D, textura[texturaID]);
  glBegin(GL_QUADS);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(0, 0, 0);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(l, 0, 0);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(l, l, 0);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(0, l, 0);
  glEnd();
}

void limparTela() {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}

// Interpola linearmente duas texturas, copiando a imagem final
// em "resultado".
void interpolaOitavas(int oitavaInicial, int oitavaFinal,
                      float taxa, int resultado) {
  limparTela();
  glPushMatrix();
    glColor4f(1.0f,1.0f,1.0f,1.0f);
    desenhaQuadrado(256.f, oitavaInicial);
    glTranslatef(0, 0, -1.00);
    glColor4f(1.0f,1.0f,1.0f, taxa);
    desenhaQuadrado(256.f, oitavaFinal);
  glPopMatrix();
  glBindTexture(GL_TEXTURE_2D,textura[resultado]);
  glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 0, 0, 256, 256, 0);
}

// Mistura todas as oitavas, gerando a textura das nuvens.
void compoeOitavas() {
  glLoadIdentity();
  glViewport(0, 0, 256, 256);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(0, 256, 0, 256, 0, 1);
  glMatrixMode(GL_MODELVIEW);
  glDisable(GL_DEPTH_TEST);
  glEnable(GL_BLEND);
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  interpolaOitavas(1, 2, taxa1, 7);
  interpolaOitavas(3, 4, taxa2, 8);
  interpolaOitavas(5, 6, taxa3, 9);

  glViewport(0, 0, 512, 512);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(0, 512, 0, 512, 0, 1);
  glMatrixMode(GL_MODELVIEW);
  limparTela();
  glPushMatrix();
    glColor4f(1.0f,1.0f,1.0f,0.5f);
    desenhaQuadrado(512.0f, 7);
    glTranslatef(0, 0, -1.0);
    glColor4f(1.0f,1.0f,1.0f, 0.25f);
    desenhaQuadrado(512.0f, 8);
    glTranslatef(0, 0, -1.0);
```

```
    glColor4f(1.0f,1.0f,1.0f, 0.125f);
    desenhaQuadrado(512.0f, 9);
  glPopMatrix();
  glBindTexture(GL_TEXTURE_2D,textura[10]);
  glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 0, 0, 512, 512, 0);

  limparTela();
  glPushMatrix();
    glBlendFunc(GL_SRC_COLOR, GL_DST_COLOR);
    glColor4f(1.0f,1.0f,1.0f,1.0f);
    desenhaQuadrado(512.0f, 0);
    glTranslatef(0, 0, -1.0);
    desenhaQuadrado(512.0f, 10);
  glPopMatrix();
  glBindTexture(GL_TEXTURE_2D,textura[10]);
  glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 0, 0, 512, 512, 0);

  glDisable(GL_BLEND);
  glEnable(GL_DEPTH_TEST);
  glViewport(0, 0, LARGURA_DA_TELA, ALTURA_DA_TELA);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0f,(GLfloat)LARGURA_DA_TELA/(GLfloat)ALTURA_DA_TELA,
                 0.5f ,5000.0f);
  glMatrixMode(GL_MODELVIEW);
  limparTela();
}

// Retorna o quadrado do número
float quadrado(float x) {
  float resultado = x * x;
  return resultado;
}

// Função que gera o skyplane. As opções são:
// - O número de divisões do plano: quanto maior, mais detalhada e pesada;
// - O raio indicando o ponto mais alto da curvatura;
// - O raio indicando o quão esticado está o plano;
// - Mapeamento da textura na horizontal;
// - Mapeamento da textura na vertical.
void criaSkyplane(int divisoes, float raio1, float raio2,
                  float hTile, float vTile) {
      // Certifica que vetores estão limpos.
      if (vertices) {
            delete vertices;
            vertices = NULL;
      }
      if (indices) {
            delete indices;
            indices = NULL;
      }

      // A curvatura é salva para centrar o céu.
      raio = raio1;

      // Inicializa os vetores que conterão os vértices e indices.
      numeroDeVertices = (int)quadrado(divisoes + 1);
      numeroDeIndices  = (int)quadrado(divisoes) * 2 * 3;
      vertices = new Vertice[numeroDeVertices];
      indices = new int[numeroDeIndices];
```

```
      // Calcula os vértices do skyplane
      float tamanhoDoPlano = 2.0f * (float)sqrt((quadrado(raio2) -
                                          quadrado(raio1)));
      float delta = tamanhoDoPlano / (float)divisoes;
      float deltaTextura = 2.0f / (float)divisoes;
      float distanciaX   = 0.0f;
      float distanciaZ   = 0.0f;
      float alturaX = 0.0f;
      float alturaZ = 0.0f;
      float altura = 0.0f;
      Vertice verticeTemporario;
      for (int i=0; i <= divisoes; i++) {
        for (int j=0; j <= divisoes; j++){
          distanciaX = (-0.5f * tamanhoDoPlano) + ((float)j * delta);
          distanciaZ = (-0.5f * tamanhoDoPlano) + ((float)i * delta);
          alturaX = (distanciaX * distanciaX) / raio2;
          alturaZ = (distanciaZ * distanciaZ) / raio2;
          altura = alturaX + alturaZ;
          verticeTemporario.x = distanciaX;
          verticeTemporario.y = 0.0f - altura;
          verticeTemporario.z = distanciaZ;
          verticeTemporario.u = hTile * ((float)j * deltaTextura * 0.5f);
          verticeTemporario.v = vTile*(1.0f-(float)i*deltaTextura*0.5f);
          vertices[i * (divisoes + 1) + j] = verticeTemporario;
        }
      }
      int indice = 0;
      for (int i = 0; i < divisoes; i++)  {
        for (int j = 0; j < divisoes; j++) {
          int verticeInicial = (i * (divisoes + 1) + j);
          indices[indice++] = verticeInicial;
          indices[indice++] = verticeInicial + 1;
          indices[indice++] = verticeInicial + divisoes + 1;
          indices[indice++] = verticeInicial + 1;
          indices[indice++] = verticeInicial + divisoes + 2;
          indices[indice++] = verticeInicial + divisoes + 1;
        }
      }
}

// Desenha o skyplane a partir do vetor de vértices.
void desenhaSkyplane() {
  static float wrap = 0; // Variável usada para animar a textura da nuvens.
   // Ativa a textura 0, que contém o céu.
  glActiveTextureARB(GL_TEXTURE0_ARB);
  glEnable(GL_TEXTURE_2D);
  glBindTexture(GL_TEXTURE_2D,  textura[11]);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
   // Ativa a textura 1, que contém as nuvens.
  glActiveTextureARB(GL_TEXTURE1_ARB);
  glEnable(GL_TEXTURE_2D);
  glBindTexture(GL_TEXTURE_2D,  textura[10]);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
      glPushMatrix();
        glTranslatef(0.0, raio, 0.0); // Centraliza o plano.
        glBegin(GL_TRIANGLES);
          // Desenha todos os vértices.
          for (int i=0; i < numeroDeIndices; i++) {
            // Mapea a primeira textura.
             glMultiTexCoord2fARB(GL_TEXTURE0_ARB, vertices[indices[i]].u,
                                  vertices[indices[i]].v);
```

```
            // Mapea a segunda textura.
            glMultiTexCoord2fARB(GL_TEXTURE1_ARB,
                                 vertices[indices[i]].u - wrap,
                                 vertices[indices[i]].v);
            glVertex3f(vertices[indices[i]].x, vertices[indices[i]].y,
                       vertices[indices[i]].z);
          }
        glEnd();
      glPopMatrix();
      glActiveTextureARB(GL_TEXTURE1_ARB);
  glDisable(GL_TEXTURE_2D);
  glActiveTextureARB(GL_TEXTURE0_ARB);
  glDisable(GL_TEXTURE_2D);
  wrap += 0.0001f; // Move a textura das nuvens.
}

// Libera a memória utiliza pelos vetores.
// Função chamada a partir de liberaLimpaTudo.
void apagaSkyplane() {
  if (vertices) {
    delete vertices;
    vertices = NULL;
  }
  if (indices) {
    delete indices;
    indices = NULL;
  }
}

void liberaLimpaTudo() {
  apagaSkyplane();
  glDeleteTextures(12, textura);
  SDL_Quit();
}

void renderizarQuadro() {
  compoeOitavas();
  gluLookAt(camera.posicao.x, camera.posicao.y, camera.posicao.z,
            camera.direcao.x, camera.direcao.y, camera.direcao.z,
            camera.topo.x, camera.topo.y, camera.topo.z);
  desenhaSkyplane();
  glEnable(GL_TEXTURE_2D);
  SDL_GL_SwapBuffers();
}

// A cada determinado tempo, regenara-se as oitavas.
// No caso, a oitava de resolução 128 regenara a cada 1 segundo;
// a de 64 a cada 2 segundos; e a de 32 a cada 4 segundos.
// Desta forma, a evolução das nuvens é mais visível nas partes menores.
void fazerLogica() {
  static float n = 1.0f;
  taxa1 = ((float)SDL_GetTicks() * 0.001 - t1) / (4*n);
  taxa2 = ((float)SDL_GetTicks() * 0.001 - t2) / (2*n);
  taxa3 = ((float)SDL_GetTicks() * 0.001 - t3) / n;
  if (taxa1 > 1.0) {
    copiaTextura(32, 2, 1);
    geraOitava(32, 2);
    t1 = (float)SDL_GetTicks() * 0.001;
    taxa1 = 0.0;
  }
  if (taxa2 > 1.0) {
```

```
    copiaTextura(64, 4, 3);
    geraOitava(64, 4);
    t2 = (float)SDL_GetTicks() * 0.001;
    taxa2 = 0.0;
  }
  if (taxa3 > 1.0) {
    copiaTextura(128, 6, 5);
    geraOitava(128, 6);
    t3 = (float)SDL_GetTicks() * 0.001;
    taxa3 = 0.0;
  }
}

// Aqui faz-se processamento das entradas do usuário.
// "Esc" sai; mouse rotaciona a câmera.
void pegarEntradas() {
  while(SDL_PollEvent(&evento)) {
    switch (evento.type) {
      case SDL_QUIT:
        estado = SAINDO;
        break;
      case SDL_KEYDOWN:
        if(evento.key.keysym.sym == SDLK_ESCAPE) { estado = SAINDO; }
        break;
      case SDL_MOUSEMOTION:
        camera.ajustaDirecaoComMouse();
        break;
      default: break;
    }
  }
}

// Esta função faz todo procedimento de inicialização.
void inicializa() {
  // Inicialização da SDL, responsável pela interface
  // entre o sistema e a OpenGL.
  SDL_Init(SDL_INIT_VIDEO);
  SDL_GL_SetAttribute( SDL_GL_DOUBLEBUFFER, 1 );
  SDL_GL_SetAttribute( SDL_GL_DEPTH_SIZE, COR_DA_TELA);
  SDL_GL_SetAttribute( SDL_GL_STENCIL_SIZE, 0);
  SDL_GL_SetAttribute( SDL_GL_ACCUM_RED_SIZE, 0);
  SDL_GL_SetAttribute( SDL_GL_ACCUM_GREEN_SIZE, 0);
  SDL_GL_SetAttribute( SDL_GL_ACCUM_BLUE_SIZE, 0);
  SDL_GL_SetAttribute( SDL_GL_ACCUM_ALPHA_SIZE, 0);
  SDL_WM_SetCaption("Textura Procedural", "Textura Procedural");
  SDL_SetVideoMode(LARGURA_DA_TELA, ALTURA_DA_TELA, COR_DA_TELA,
                   SDL_OPENGL | SDL_HWSURFACE | SDL_HWACCEL);
  SDL_EnableKeyRepeat(100, SDL_DEFAULT_REPEAT_INTERVAL);
  SDL_WM_GrabInput(SDL_GRAB_ON);
  SDL_ShowCursor(SDL_DISABLE);

  // Ajustes iniciais da OpenGL.
  glActiveTextureARB = (PFNGLACTIVETEXTUREARBPROC)
                       SDL_GL_GetProcAddress("glActiveTextureARB");
  glMultiTexCoord2fARB = (PFNGLMULTITEXCOORD2FARBPROC)
                         SDL_GL_GetProcAddress("glMultiTexCoord2fARB");
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_TEXTURE_2D);
  glEnable(GL_CULL_FACE);
  glViewport(0, 0, LARGURA_DA_TELA, ALTURA_DA_TELA);
  glMatrixMode(GL_PROJECTION);
```

```
  glLoadIdentity();
  gluPerspective(45.0f,(GLfloat)LARGURA_DA_TELA/(GLfloat)ALTURA_DA_TELA,
                 0.5f ,900.0f);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

  // Carregamento das texturas da skybox, realizada através de textura.h.
  // Se não for possível abrir a imagem, erro será igual a 1.
  erro = criaTextura(textura, "corte.tga", 0);
  // São geradas dez texturas vazias:
  // - Duas para cada oitava/resolução (tempo inicial, tempo final);
  // - Uma para cada mistura/interpolação (números 7, 8, 9);
  // - A textura final contendo as nuvens (número 10).
  criaTexturaVazia(32, 3, GL_RGB, 1);
  criaTexturaVazia(32, 3, GL_RGB, 2);
  criaTexturaVazia(64, 3, GL_RGB, 3);
  criaTexturaVazia(64, 3, GL_RGB, 4);
  criaTexturaVazia(128, 3, GL_RGB, 5);
  criaTexturaVazia(128, 3, GL_RGB, 6);
  criaTexturaVazia(256, 3, GL_RGB, 7);
  criaTexturaVazia(256, 3, GL_RGB, 8);
  criaTexturaVazia(256, 3, GL_RGB, 9);
  criaTexturaVazia(512, 3, GL_RGB, 10);
  erro = criaTextura(textura, "fundo.tga", 11);
  // Inicialização das oitavas (tempo inicial e final).
  geraOitava(32, 1);
  geraOitava(32, 2);
  geraOitava(64, 3);
  geraOitava(64, 4);
  geraOitava(128, 5);
  geraOitava(128, 6);
  // Inicialização dos relógios que controlam quandro gerar novas oitavas.
  t1 = (float)SDL_GetTicks() * 0.001;
  t2 = t1;
  t3 = t1;
  criaSkyplane(16, 100.0f, 900.0f, 1.0f, 1.0f);
}

// Função principal (main) começa aqui
int main(int argc, char *argv[]) {
  // Implementação genérica do loop principal de jogo
  while (estado != SAINDO) {
    switch (estado) {
      case INICIALIZANDO:
        inicializa();
        estado = RODANDO;
        break;
      case RODANDO:
        limparTela();
        pegarEntradas();
        fazerLogica();
        renderizarQuadro();
        break;
      case SAINDO:
        liberaLimpaTudo();
        break;
      default: break;
    }
  }
  return erro;
}
```

7.1.4 billboard.cpp

```
#include <SDL/SDL.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include "camera.h"
#include "textura.h"

// Definição dos estados do loop do jogo.
enum {INICIALIZANDO, RODANDO, SAINDO};
// Numeração das imagens da skybox.
enum {FRENTE, ESQUERDA, TRASEIRA, DIREITA, EMCIMA, EMBAIXO};

// Variáveis globais
const int LARGURA_DA_TELA = 800;
const int ALTURA_DA_TELA = 600;
const int COR_DA_TELA = 32;
int estado = INICIALIZANDO; // Começa neste estado.
int erro = 0; // Usado para informar erros ao sistema operacional.
Camera camera(0, 0, 0,  0, 0, -6,  0, 1, 0,
              LARGURA_DA_TELA, ALTURA_DA_TELA);
unsigned int textura[7] = {0};
SDL_Event evento;
float deslocamento = 0.0;
float tempo = 0.0;
char wireframe = 0;

void geraSkybox(float largura, float altura, float espessura) {
  // As variáveis x, y, z definem o centro da caixa.
  float x = -largura  / 2.0;
  float y = -altura / 2.0;
  float z = -espessura / 2.0;

  // Aqui é gerada a caixa. Basicamente o trabalho é pegar a textura
  // específica da face e mapeá-la na mesma.
  // Face frontal.
  glBindTexture(GL_TEXTURE_2D, textura[FRENTE]);
  glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(x, y, z);
    glTexCoord2f(1.0, 0.0); glVertex3f(x+largura, y, z);
    glTexCoord2f(1.0, 1.0); glVertex3f(x+largura, y+altura, z);
    glTexCoord2f(0.0, 1.0); glVertex3f(x, y+altura, z);
  glEnd();
  // Face da direita
  glBindTexture(GL_TEXTURE_2D, textura[DIREITA]);
  glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(x+largura, y, z);
    glTexCoord2f(1.0, 0.0); glVertex3f(x+largura, y, z+espessura);
    glTexCoord2f(1.0, 1.0); glVertex3f(x+largura, y+altura, z+espessura);
    glTexCoord2f(0.0, 1.0); glVertex3f(x+largura, y+altura, z);
  glEnd();
  // Face traseira.
  glBindTexture(GL_TEXTURE_2D, textura[TRASEIRA]);
  glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(x+largura, y, z+espessura);
    glTexCoord2f(1.0, 0.0); glVertex3f(x, y, z+espessura);
    glTexCoord2f(1.0, 1.0); glVertex3f(x, y+altura, z+espessura);
    glTexCoord2f(0.0, 1.0); glVertex3f(x+largura, y+altura, z+espessura);
  glEnd();
```

```
  // Face da esquerda
  glBindTexture(GL_TEXTURE_2D, textura[ESQUERDA]);
  glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(x, y, z+espessura);
    glTexCoord2f(1.0, 0.0); glVertex3f(x, y, z);
    glTexCoord2f(1.0, 1.0); glVertex3f(x, y+altura, z);
    glTexCoord2f(0.0, 1.0); glVertex3f(x, y+altura, z+espessura);
  glEnd();
  // Face de cima
  glBindTexture(GL_TEXTURE_2D, textura[EMCIMA]);
  glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(x, y+altura, z);
    glTexCoord2f(1.0, 0.0); glVertex3f(x+largura, y+altura, z);
    glTexCoord2f(1.0, 1.0); glVertex3f(x+largura, y+altura, z+espessura);
    glTexCoord2f(0.0, 1.0); glVertex3f(x, y+altura, z+espessura);
  glEnd();
  // Face de baixo.
  glBindTexture(GL_TEXTURE_2D, textura[EMBAIXO]);
  glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(x, y, z+espessura);
    glTexCoord2f(1.0, 0.0); glVertex3f(x+largura, y, z+espessura);
    glTexCoord2f(1.0, 1.0); glVertex3f(x+largura, y, z);
    glTexCoord2f(0.0, 1.0); glVertex3f(x, y, z);
  glEnd();
}

// Desenha um billboard na posição (x, y, z)
// com largura, altura e textura especificada.
void billboard(float x, float y, float z,
               float largura, float altura, int texturaID) {
  // No plano xz a rotação do billboard é determinada pela intersecção
  // da direção do polígono neste plano com a paralela ao eixo x
  // na posição z do observador.
  double catetoAdjacente = camera.posicao.x - x;
  double catetoOposto = camera.posicao.z - z;
  double tangente = catetoOposto / catetoAdjacente;
  // atan() retorna o valor em radianos, necessitando conversão para graus.
  double anguloComplementar = atan(tangente) * 180 / M_PI;
  double anguloXZ = 90.0 - anguloComplementar;
  if (anguloXZ > 90.0) { anguloXZ -= 180.0; }
  if (anguloXZ < -90.0) { anguloXZ += 180.0; }

  // No plano yz a rotação do billboard é determinada pela intersecção
  // da direção do polígono neste plano com a paralela ao eixo y
  // na posição z do observador.
  catetoAdjacente = y - camera.posicao.y;
  catetoOposto = z - camera.posicao.z;
  tangente = catetoOposto / catetoAdjacente;
  anguloComplementar = atan(tangente) * 180 / M_PI;
  double anguloYZ = 90 - anguloComplementar;
  if (anguloYZ > 90.0) { anguloYZ -= 180.0; }
  if (anguloYZ < -90.0) { anguloYZ += 180.0; }

  glEnable(GL_BLEND);
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  glColor4f(1.0f,1.0f,1.0f,0.9f);
  if (wireframe) {
   glDisable(GL_BLEND);
   glDisable(GL_TEXTURE_2D);
   glPolygonMode( GL_FRONT_AND_BACK, GL_LINE );
  }
```

```
  glPushMatrix();
    glTranslatef(x, y, z);
    glRotatef(anguloXZ, 0, 1, 0);
    glRotatef(anguloYZ, 1, 0, 0);
    glBindTexture(GL_TEXTURE_2D, textura[texturaID]);
    glBegin(GL_QUADS);
      glTexCoord2f(0.0, 0.0); glVertex3f(-largura/2,  altura/2, 0.0);
      glTexCoord2f(0.0, 1.0); glVertex3f(-largura/2, -altura/2, 0.0);
      glTexCoord2f(1.0, 1.0); glVertex3f( largura/2, -altura/2, 0.0);
      glTexCoord2f(1.0, 0.0); glVertex3f( largura/2,  altura/2, 0.0);
    glEnd();
  glPopMatrix();

      if (wireframe) {
    glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );
    glEnable(GL_TEXTURE_2D);
  }

  glDisable(GL_BLEND);
}

void liberaLimpaTudo() {
  glDeleteTextures(7, textura);
  SDL_Quit();
}

void renderizarQuadro() {
  glLoadIdentity();
  gluLookAt(camera.posicao.x, camera.posicao.y, camera.posicao.z,
            camera.direcao.x, camera.direcao.y, camera.direcao.z,
            camera.topo.x, camera.topo.y, camera.topo.z);
  geraSkybox(400, 400, 400);
  glDepthMask(GL_FALSE);
  billboard(-85.0 - deslocamento, 0.0, -190.0, 20.0, 20.0, 6);
  billboard(-35.0 - deslocamento, 0.0, -190.0, 20.0, 20.0, 6);
  billboard(15.0 + deslocamento, 0.0, -190.0, 20.0, 20.0, 6);
  billboard(65.0 + deslocamento, 0.0, -190.0, 20.0, 20.0, 6);

  billboard(-60.0 - deslocamento, 0.0, -150.0, 20.0, 20.0, 6);
  billboard(-10.0, 0.0, -150.0, 20.0, 20.0, 6);
  billboard(40.0 + deslocamento, 0.0, -150.0, 20.0, 20.0, 6);

  billboard(-35.0 - deslocamento, 0.0, -110.0, 20.0, 20.0, 6);
  billboard(15.0 + deslocamento, 0.0, -110.0, 20.0, 20.0, 6);

  billboard(-10.0, 0.0, -70.0, 20.0, 20.0, 6);
  glDepthMask(GL_TRUE);


  SDL_GL_SwapBuffers();
}

void fazerLogica() {
  static char sentido = 0;
  if (!sentido) {
    deslocamento += ((float)SDL_GetTicks() * 0.001 - tempo)/8.0;
    if (deslocamento > 20.0) {
      tempo = (float)SDL_GetTicks() * 0.001;
      sentido = 1;
    }
```

```
  }
  else {
    deslocamento -= ((float)SDL_GetTicks() * 0.001 - tempo)/8.0;
    if (deslocamento < -20.0) {
      tempo = (float)SDL_GetTicks() * 0.001;
      sentido = 0;
    }
  }
}

// Aqui faz-se processamento das entradas do usuário.
// "Esc" sai; mouse rotaciona a câmera.
void pegarEntradas() {
  while(SDL_PollEvent(&evento)) {
    switch (evento.type) {
      case SDL_QUIT:
        estado = SAINDO;
        break;
      case SDL_KEYDOWN:
        if(evento.key.keysym.sym == SDLK_w) {
          if (wireframe) { wireframe = 0; }
          else { wireframe = 1; }
        }
        if(evento.key.keysym.sym == SDLK_ESCAPE) { estado = SAINDO; }
        if(evento.key.keysym.sym == SDLK_UP) { camera.move(0.5f); }
        if(evento.key.keysym.sym == SDLK_DOWN) { camera.move(-0.5f); }
        if(evento.key.keysym.sym == SDLK_LEFT) { camera.desliza(-0.5f); }
        if(evento.key.keysym.sym == SDLK_RIGHT) { camera.desliza(0.5f); }
        break;
      case SDL_MOUSEMOTION:
        camera.ajustaDirecaoComMouse();
        break;
      default: break;
    }
  }
}

void limparTela() {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}

// Esta função faz todo procedimento de inicialização.
void inicializa() {
  // Inicialização da SDL, responsável pela interface entre o sistema e a
OpenGL.
  SDL_Init(SDL_INIT_VIDEO);
  SDL_GL_SetAttribute( SDL_GL_DOUBLEBUFFER, 1 );
  SDL_GL_SetAttribute( SDL_GL_DEPTH_SIZE, COR_DA_TELA);
  SDL_GL_SetAttribute( SDL_GL_STENCIL_SIZE, 0);
  SDL_GL_SetAttribute( SDL_GL_ACCUM_RED_SIZE, 0);
  SDL_GL_SetAttribute( SDL_GL_ACCUM_GREEN_SIZE, 0);
  SDL_GL_SetAttribute( SDL_GL_ACCUM_BLUE_SIZE, 0);
  SDL_GL_SetAttribute( SDL_GL_ACCUM_ALPHA_SIZE, 0);
  SDL_WM_SetCaption("Billboard", "Billboard");
  SDL_SetVideoMode(LARGURA_DA_TELA, ALTURA_DA_TELA, COR_DA_TELA, SDL_OPENGL
| SDL_HWSURFACE | SDL_HWACCEL| SDL_FULLSCREEN);
  SDL_EnableKeyRepeat(100, SDL_DEFAULT_REPEAT_INTERVAL);
  SDL_WM_GrabInput(SDL_GRAB_ON);
  SDL_ShowCursor(SDL_DISABLE);
```

```
  // Ajustes iniciais da OpenGL.
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_TEXTURE_2D);
  glEnable(GL_CULL_FACE);
  glViewport(0, 0, LARGURA_DA_TELA, ALTURA_DA_TELA);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0f,(GLfloat)LARGURA_DA_TELA/(GLfloat)ALTURA_DA_TELA,
                 .5f ,500.0f);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

  // Carregamento das texturas da skybox, realizada através de textura.h.
  // Se não for possível abrir a imagem, erro será igual a 1.
  erro = criaTextura(textura, "traseira.tga", TRASEIRA);
  // Para não aparecer emendas nas imagens do cubo, as texturas devem estar
  // configuradas para não repitir (GL_CLAMP_TO_EDGE).
  texturaNaoRepete();
  erro = criaTextura(textura, "frente.tga", FRENTE);
  texturaNaoRepete();
  erro = criaTextura(textura, "embaixo.tga", EMBAIXO);
  texturaNaoRepete();
  erro = criaTextura(textura, "emcima.tga", EMCIMA);
  texturaNaoRepete();
  erro = criaTextura(textura, "esquerda.tga", ESQUERDA);
  texturaNaoRepete();
  erro = criaTextura(textura, "direita.tga", DIREITA);
  texturaNaoRepete();
  erro = criaTextura(textura, "nuvem.tga", 6);
  tempo = (float)SDL_GetTicks() * 0.001;
}

// Função principal (main) começa aqui
int main(int argc, char *argv[]) {
  // Implementação genérica do loop principal de jogo
  while (estado != SAINDO) {
    switch (estado) {
      case INICIALIZANDO:
        inicializa();
        estado = RODANDO;
        break;
      case RODANDO:
        limparTela();
        pegarEntradas();
        fazerLogica();
        renderizarQuadro();
        break;
      case SAINDO:
        liberaLimpaTudo();
        break;
      default: break;
    }
  }
  return erro;
}
```

7.1.5 particulas.cpp

```
#include <SDL/SDL.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glext.h>
#include <windows.h>
#include "camera.h"
#include "textura.h"

// Definição dos estados do loop do jogo.
enum {INICIALIZANDO, RODANDO, SAINDO};
// Numeração das imagens da skybox.
enum {FRENTE, ESQUERDA, TRASEIRA, DIREITA, EMCIMA, EMBAIXO};

// Variáveis globais
const int LARGURA_DA_TELA = 800;
const int ALTURA_DA_TELA = 600;
const int COR_DA_TELA = 32;
int estado = INICIALIZANDO; // Começa neste estado.
int erro = 0; // Usado para informar erros ao sistema operacional.
Camera camera(0, 0, 0,  0, 0, -6,  0, 1, 0,  LARGURA_DA_TELA,
ALTURA_DA_TELA);
unsigned int textura[1];
SDL_Event evento;
const int MAXIMO_DE_PARTICULAS = 100;

struct Particula {
  float vida;  // Vida da particula.
  float desaparecimento;  // Velocidade de desaparecimento
  float x;
  float y;
  float z;
  float direcaoX;
  float direcaoY;
  float direcaoZ;
};

Particula particula[MAXIMO_DE_PARTICULAS];
int loop;
float V, angulo;

void liberaLimpaTudo() {
  glDeleteTextures(1, textura);
  SDL_Quit();
}

void particulas() {
  glEnable(GL_BLEND);
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  glTranslatef(0.0f, 0.0f, -15.0f);
  for (loop = 0; loop < MAXIMO_DE_PARTICULAS; loop++) {
    float x = particula[loop].x;
    float y = particula[loop].y;
    float z = particula[loop].z;

    glColor4f(1.0f, 1.0f, 1.0f, particula[loop].vida);
    glBindTexture(GL_TEXTURE_2D, textura[0]);
    glBegin(GL_TRIANGLE_STRIP);
      glTexCoord2f(1.0f, 1.0f); glVertex3f(x + 0.2f, y + 0.2f, z);
```

```
      glTexCoord2f(0.0f, 1.0f); glVertex3f(x - 0.2f, y + 0.2f, z);
      glTexCoord2f(1.0f, 0.0f); glVertex3f(x + 0.2f, y - 0.2f, z);
      glTexCoord2f(0.0f, 0.0f); glVertex3f(x - 0.2f, y - 0.2f, z);
    glEnd();

    particula[loop].x += particula[loop].direcaoX / 250;
    particula[loop].y += particula[loop].direcaoY / 250;
    particula[loop].z += particula[loop].direcaoZ / 250;

    // Slow down the particulas
    particula[loop].direcaoX *= .99;
    particula[loop].direcaoY *= .99;
    particula[loop].direcaoZ *= .99;

   // Reduz a vida da partícula baseado na velocidade de desaparecimento.
    particula[loop].vida -= particula[loop].desaparecimento;

    if (particula[loop].vida < 0.05f) {
      particula[loop].vida = 1.0f; // Dá vida nova.
      // Determina aleatoriamente o desaparecimento.
      particula[loop].desaparecimento = float(rand() % 100) / 10000 +
0.005f;
      // Centraliza a partícula.
      particula[loop].x = 0;
      particula[loop].y = 0;
      particula[loop].z = 0;
      V = (float((rand() % 9)) + 1);
      angulo = float(rand() % 360);

      particula[loop].direcaoX = sin(angulo) * V;
      particula[loop].direcaoY = cos(angulo) * V;
      particula[loop].direcaoZ = ((rand() % 10) - 5) / 5;
    }
  }
  glDisable(GL_BLEND);
}

void renderizarQuadro() {
  glLoadIdentity();
  gluLookAt(camera.posicao.x, camera.posicao.y, camera.posicao.z,
            camera.direcao.x, camera.direcao.y, camera.direcao.z,
            camera.topo.x, camera.topo.y, camera.topo.z);
 glPushMatrix();
   glDepthMask(GL_FALSE);
   particulas();
   glDepthMask(GL_TRUE);
 glPopMatrix();

  SDL_GL_SwapBuffers();
}

// Aqui faz-se processamento das entradas do usuário.
// "Esc" sai; mouse rotaciona a câmera.
void pegarEntradas() {
  while(SDL_PollEvent(&evento)) {
    switch (evento.type) {
      case SDL_QUIT:
        estado = SAINDO;
        break;
      case SDL_KEYDOWN:
        if(evento.key.keysym.sym == SDLK_ESCAPE) { estado = SAINDO; }
```

```
        if(evento.key.keysym.sym == SDLK_UP) { camera.move(0.05f); }
        if(evento.key.keysym.sym == SDLK_DOWN) { camera.move(-0.05f); }
        if(evento.key.keysym.sym == SDLK_LEFT) { camera.desliza(-0.05f); }
        if(evento.key.keysym.sym == SDLK_RIGHT) { camera.desliza(0.05f); }
        break;
      case SDL_MOUSEMOTION:
        camera.ajustaDirecaoComMouse();
        break;
      default: break;
    }
  }
}

void limparTela() {
  glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
}

// Esta função faz todo procedimento de inicialização.
void inicializa() {
  // Inicialização da SDL, responsável pela interface entre o sistema e a
OpenGL.
  SDL_Init(SDL_INIT_VIDEO);
  SDL_GL_SetAttribute( SDL_GL_DOUBLEBUFFER, 1 );
  SDL_GL_SetAttribute( SDL_GL_DEPTH_SIZE, COR_DA_TELA);
  SDL_GL_SetAttribute( SDL_GL_STENCIL_SIZE, 0);
  SDL_GL_SetAttribute( SDL_GL_ACCUM_RED_SIZE, 0);
  SDL_GL_SetAttribute( SDL_GL_ACCUM_GREEN_SIZE, 0);
  SDL_GL_SetAttribute( SDL_GL_ACCUM_BLUE_SIZE, 0);
  SDL_GL_SetAttribute( SDL_GL_ACCUM_ALPHA_SIZE, 0);
  SDL_WM_SetCaption("Skybox", "Skybox");
  SDL_SetVideoMode(LARGURA_DA_TELA, ALTURA_DA_TELA, COR_DA_TELA,
                   SDL_OPENGL | SDL_HWSURFACE | SDL_HWACCEL);
  SDL_EnableKeyRepeat(100, SDL_DEFAULT_REPEAT_INTERVAL);
  SDL_WM_GrabInput(SDL_GRAB_ON);
  SDL_ShowCursor(SDL_DISABLE);

  // Ajustes iniciais da OpenGL.
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_TEXTURE_2D);
  //glClearColor(0, 0, 1, 1.0f);
  glShadeModel(GL_SMOOTH);
  glViewport(0, 0, LARGURA_DA_TELA, ALTURA_DA_TELA);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0f,(GLfloat)LARGURA_DA_TELA/(GLfloat)ALTURA_DA_TELA,
                 .5f ,5000.0f);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

  texturaNaoRepete();
  erro = criaTextura(textura, "nuvem.tga", 0);

  for (loop = 0; loop < MAXIMO_DE_PARTICULAS; loop++) {
    particula[loop].vida = 1.0f; // Dá vida total a todas as partículas.
    particula[loop].desaparecimento = float(rand() % 100) / 1000.0f +
0.05f;
    V = float(rand() % 25);
    angulo = float(rand() % 360);
    particula[loop].x = 0;
    particula[loop].y = 0;
    particula[loop].z = 0;
```

```
    particula[loop].direcaoX = sin(angulo) * V;
    particula[loop].direcaoY = cos(angulo) * V;
    particula[loop].direcaoZ = float(((rand() % 10) - 5) / 10) * V;
      }
}

// Função principal (main) começa aqui
int main(int argc, char *argv[]) {
  // Implementação genérica do loop principal de jogo
  while (estado != SAINDO) {
    switch (estado) {
      case INICIALIZANDO:
        inicializa();
        estado = RODANDO;
        break;
      case RODANDO:
        limparTela();
        pegarEntradas();
        renderizarQuadro();
        break;
      case SAINDO:
        liberaLimpaTudo();
        break;
      default: break;
    }
  }
  return erro;
}
```

7.2 Apêndice B – Códigos auxiliares

7.2.1 camera.h

```
#ifndef _CAMERA_H
#define _CAMERA_H

#include <SDL/SDL.h>
#include <math.h>

class Vetor {
  public:
    Vetor() {}
    Vetor(float X, float Y, float Z) { x = X; y = Y; z = Z; }
    Vetor operator+(Vetor vetor) {
      return Vetor(vetor.x + x, vetor.y + y, vetor.z + z);
    }
    Vetor operator-(Vetor vetor) {
      return Vetor(x - vetor.x, y - vetor.y, z - vetor.z);
    }
    Vetor operator*(float numero) {
      return Vetor(x * numero, y * numero, z * numero);
    }
    Vetor operator/(float numero) {
      return Vetor(x / numero, y / numero, z / numero);
    }
    float x, y, z;
};

class Camera {
  public:
    Camera(float posicaoX = 0.0f,
           float posicaoY = 0.0f,
           float posicaoZ = 0.0f,
           float direcaoX = 0.0f,
           float direcaoY = 0.0f,
           float direcaoZ = -1.0f,
           float topoX = 0.0f,
           float topoY = 1.0f,
           float topoZ = 0.0f,
           int novaLargura = 800,
           int novaAltura = 600);
    void posiciona(float posicaoX = 0.0f,
                   float posicaoY = 0.0f,
                   float posicaoZ = 0.0f,
                   float direcaoX = 0.0f,
                   float direcaoY = 0.0f,
                   float direcaoZ = -1.0f,
                   float topoX = 0.0f,
                   float topoY = 1.0f,
                   float topoZ = 0.0f);
    void rotaciona(float angulo, float X, float Y, float Z);
    void rotacionaAoRedor(Vetor centro, float X, float Y, float Z);
    void ajustaDirecaoComMouse();
    void move(float velocidade);
    void desliza(float velocidade);
```

```
    Vetor posicao;
    Vetor direcao;
    Vetor topo;
    int largura;
    int altura;
};

#endif
```

7.2.2 camera.cpp

```
#include "camera.h"

// Retorna um vetor perpendicular à dois dados vetores.
Vetor produtoVetorial(Vetor vetor1, Vetor vetor2) {
  Vetor normal;

  // Calculo do produto vetorial com equação não comunicativa.
  normal.x = ((vetor1.y * vetor2.z) - (vetor1.z * vetor2.y));
  normal.y = ((vetor1.z * vetor2.x) - (vetor1.x * vetor2.z));
  normal.z = ((vetor1.x * vetor2.y) - (vetor1.y * vetor2.x));

  // Retorna o produto vetorial.
  return normal;
}

// Retorna o módulo do vetor.
float magnitude(Vetor normal) {
  return (float)sqrt((normal.x * normal.x) +  (normal.y * normal.y) +
                     (normal.z * normal.z));
}

// Retorna um vetor unitário (módulo igual a 1).
Vetor normaliza(Vetor vetor) {
  float modulo = magnitude(vetor);

  // De posse da magnitude do vetor, pode-se dividir esse por aquele.
  // Isto fará nosso vetor ter módulo igual a 1.
  vetor = vetor / modulo;

  return vetor;
}

////////////////////////////////////

Camera::Camera(float posicaoX, float posicaoY, float posicaoZ,
               float direcaoX, float direcaoY, float direcaoZ,
               float topoX, float topoY, float topoZ,
               int novaLargura, int novaAltura) {
  largura = novaLargura;
  altura = novaAltura;
  posiciona(posicaoX, posicaoY, posicaoZ,
            direcaoX, direcaoY, direcaoZ,
            topoX, topoY, topoZ);
}

void Camera::posiciona(float posicaoX, float posicaoY, float posicaoZ,
                       float direcaoX, float direcaoY, float direcaoZ,
                       float topoX, float topoY, float topoZ) {
```

```
  posicao.x = posicaoX;
  posicao.y = posicaoY;
  posicao.z = posicaoZ;
  direcao.x = direcaoX;
  direcao.y = direcaoY;
  direcao.z = direcaoZ;
  topo.x = topoX;
  topo.y = topoY;
  topo.z = topoZ;
}

void Camera::rotaciona(float angulo, float x, float y, float z) {
  Vetor novaDirecao;

  // Calcula o vetor direção.
  Vetor direcaoAtual = direcao - posicao;

  // Calcula cosseno e seno do ângulo para uso a seguir.
  float cosTheta = (float)cos(angulo);
  float sinTheta = (float)sin(angulo);

  // Calcula um ponto da nova direção.
  novaDirecao.x  = (cosTheta + (1 - cosTheta) * x * x) * direcaoAtual.x;
  novaDirecao.x += ((1-cosTheta) * x * y - z * sinTheta) * direcaoAtual.y;
  novaDirecao.x += ((1-cosTheta) * x * z + y * sinTheta) * direcaoAtual.z;
  novaDirecao.y  = ((1-cosTheta) * x * y + z * sinTheta) * direcaoAtual.x;
  novaDirecao.y += (cosTheta + (1-cosTheta) * y * y)     * direcaoAtual.y;
  novaDirecao.y += ((1-cosTheta) * y * z - x * sinTheta) * direcaoAtual.z;
  novaDirecao.z  = ((1-cosTheta) * x * z - y * sinTheta) * direcaoAtual.x;
  novaDirecao.z += ((1-cosTheta) * y * z + x * sinTheta) * direcaoAtual.y;
  novaDirecao.z += (cosTheta + (1-cosTheta) * z * z)     * direcaoAtual.z;

  // Agora é só adicionar o vetor rotacionado à posição para
  // obter a direção rotacinada da câmera.
  direcao = posicao + novaDirecao;
}

void Camera::ajustaDirecaoComMouse() {
  int posicaoMouseX, posicaoMouseY;
  int meioX = largura / 2;
  int meioY = altura / 2;
  float anguloY = 0.0f;
  float anguloZ = 0.0f;
  static float atualRotacaoX = 0.0f;

  SDL_GetMouseState(&posicaoMouseX,&posicaoMouseY);

  if((posicaoMouseX == meioX) && (posicaoMouseY == meioY)) { return; }

  SDL_WarpMouse(meioX, meioY);

  anguloY = ((float)(meioX - posicaoMouseX)) / 1000.0f;
  anguloZ = ((float)(meioY - posicaoMouseY)) / 1000.0;

  atualRotacaoX -= anguloZ;

  // A rotação (em radianos) tem que ser no máximo 1.0.
  if(atualRotacaoX > 1.0f) { atualRotacaoX = 1.0f; }
  // A rotação (em radianos) tem que ser no mínimo -1.0.
  else if(atualRotacaoX < -1.0f) { atualRotacaoX = -1.0f; }
  // Caso contrário, pode-se rotacionar ao redor da posição atual.
```

```
  else {
    Vetor eixo = produtoVetorial(direcao - posicao, topo);
    eixo = normaliza(eixo);

    rotaciona(anguloZ, eixo.x, eixo.y, eixo.z);
    rotaciona(anguloY, 0, 1, 0);
  }
}

void Camera::move(float velocidade) {
  Vetor vetorTemporario = direcao - posicao;
  vetorTemporario = normaliza(vetorTemporario);

  posicao.x += vetorTemporario.x * velocidade;
  posicao.z += vetorTemporario.z * velocidade;
  direcao.x += vetorTemporario.x * velocidade;
  direcao.z += vetorTemporario.z * velocidade;
}

void Camera::desliza(float velocidade) {
  Vetor produto = produtoVetorial(direcao - posicao, topo);
  Vetor deslizamento = normaliza(produto);

  posicao.x += deslizamento.x * velocidade;
  posicao.z += deslizamento.z * velocidade;

  direcao.x += deslizamento.x * velocidade;
  direcao.z += deslizamento.z * velocidade;
}
```

7.2.3 textura.h

```
#ifndef _TEXTURA_H
#define _TEXTURA_H

#include <SDL/sdl.h>
#include <GL/gl.h>
#include <GL/glu.h>

#ifndef GL_CLAMP_TO_EDGE
#define GL_CLAMP_TO_EDGE 0x812F
#endif

// Carrega a imagem "arquivo" na posição "identificacao" do vetor
"destino".
// Retorna 0 se não houver problema ou 1 se não existir o arquivo.
int criaTextura(unsigned int destino[],char* arquivo,int identificacao);

void texturaNaoRepete();

#endif
```

7.2.4 textura.cpp

```
#include "textura.h"

// Informam o tipo de TGA
#define TGA_RGB      2   // Um arquivo RGB
#define TGA_A        3   // Um arquivo ALPHA
#define TGA_RLE     10   // Um arquivo compactado (RLE)

struct TGA {
  int canais;            // Canais na imagem (3 = RGB : 4 = RGBA)
  int largura;           // Largura da imagem em pixels
  int altura;            // Altura da imagem em pixels
  unsigned char *dados;  // Os pixels da imagem
};

TGA* carregaTGA(const char* arquivo) {
  TGA* imagem = new TGA;
  int largura = 0;
  int altura = 0;
  unsigned char tamanho = 0;
  unsigned char tipoDaImagem = 0;
  unsigned char bits = 0;
  FILE *fp = NULL;
  int canais = 0;
  int stride = 0;
  int i = 0;

  if((fp = fopen(arquivo, "rb")) == NULL) {
    return NULL;
  }

  fread(&tamanho, sizeof(unsigned char), 1, fp);
  fseek(fp, 1, SEEK_CUR);
  fread(&tipoDaImagem, sizeof(unsigned char), 1, fp);
  fseek(fp, 9, SEEK_CUR);
  fread(&largura, sizeof(unsigned short), 1, fp);
  fread(&altura, sizeof(unsigned short), 1, fp);
  fread(&bits, sizeof(unsigned char), 1, fp);
  fseek(fp, tamanho + 1, SEEK_CUR);
  if(tipoDaImagem != TGA_RLE) {
    if(bits == 24 || bits == 32) {
      canais = bits / 8;
      stride = canais * largura;
      imagem->dados = new unsigned char[stride * altura];

      for(int y = 0; y < altura; y++) {
        unsigned char *linha = &(imagem->dados[stride * y]);
        fread(linha, stride, 1, fp);
        for(i = 0; i < stride; i += canais) {
          int temp = linha[i];
          linha[i] = linha[i + 2];
          linha[i + 2] = temp;
        }
      }
    }
    else if(bits == 16) {
      unsigned short pixels = 0;
```

```
    int r = 0, g = 0, b = 0;

    canais = 3;
    stride = canais * largura;
    imagem->dados = new unsigned char[stride * altura];

    for(i = 0; i < (largura * altura); i++) {
      fread(&pixels, sizeof(unsigned short), 1, fp);
      b = (pixels & 0x1f) << 3;
      g = ((pixels >> 5) & 0x1f) << 3;
      r = ((pixels >> 10) & 0x1f) << 3;
      imagem->dados[i * 3 + 0] = r;
      imagem->dados[i * 3 + 1] = g;
      imagem->dados[i * 3 + 2] = b;
    }
  }
  else { return NULL; }
}
else {
  unsigned char rleID = 0;
  int coresLidas = 0;
  canais = bits / 8;
  stride = canais * largura;
  imagem->dados = new unsigned char[stride * altura];
  unsigned char *cores = new unsigned char[canais];

  while(i < largura * altura) {
    fread(&rleID, sizeof(unsigned char), 1, fp);
    if(rleID < 128) {
      rleID++;
      while(rleID) {
        fread(cores, sizeof(unsigned char) * canais, 1, fp);
        imagem->dados[coresLidas + 0] = cores[2];
        imagem->dados[coresLidas + 1] = cores[1];
        imagem->dados[coresLidas + 2] = cores[0];
        if(bits == 32) { imagem->dados[coresLidas + 3] = cores[3]; }
        i++;
        rleID--;
        coresLidas += canais;
      }
    }
    else {
      rleID -= 127;
      fread(cores, sizeof(unsigned char) * canais, 1, fp);
      while(rleID) {
        imagem->dados[coresLidas + 0] = cores[2];
        imagem->dados[coresLidas + 1] = cores[1];
        imagem->dados[coresLidas + 2] = cores[0];
        if(bits == 32) { imagem->dados[coresLidas + 3] = cores[3]; }
        i++;
        rleID--;
        coresLidas += canais;
      }
    }
  }
}
fclose(fp);
imagem->canais = canais;
imagem->largura = largura;
imagem->altura = altura;
```

```
  return imagem;
}

void liberaTGA(TGA* imagem) {
  if (imagem) {
    if (imagem->dados) { delete imagem->dados; }
    delete imagem;
  }
}

int criaTextura(unsigned int destino[],char* arquivo,int identificacao) {
  if(!arquivo) { return 1; }
  TGA* imagem = carregaTGA(arquivo);
  if(imagem == NULL) { return 1;}

  glGenTextures(1, &destino[identificacao]);
  glBindTexture(GL_TEXTURE_2D, destino[identificacao]);
  int tipoDeTextura = GL_RGB;
  if(imagem->canais == 4) { tipoDeTextura = GL_RGBA; }
  gluBuild2DMipmaps(GL_TEXTURE_2D, imagem->canais, imagem->largura,
                    imagem->altura, tipoDeTextura, GL_UNSIGNED_BYTE,
imagem->dados);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

  liberaTGA(imagem);
  return 0;
}

void texturaNaoRepete() {
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
}
```

8 GLOSSÁRIO

Aliasing – Problemas de interferência resultando em traços "serrilhados" devido a rasterização funcionar por amostragem, pois a representação vetorial é um domínio contínuo enquanto a representação matricial é um domínio discreto.

Alpha-blending – Técnica de processamento gráfico para obter transparência do *pixel*.

Antialiasing – Qualquer técnica utilizada para reduzir o impacto visual do *aliasing*.

API – Interface de programação da aplicação (*application programming interface*) é um conjunto de definições que permitem a um software se comunicar com outro. É um método de se obter abstração, usualmente (mas não necessariamente) entre software de baixo nível e alto nível.

Arcades – No Brasil conhecidos como fliperamas, são máquinas de jogos que funcionam a base de moedas ou fichas (uma partida, uma ficha) e podem ser encontradas em bares, parques de diversão e casas especializadas.

Billboard – Face (polígono) texturizada sempre orientada segundo uma determinada vista. À medida que a vista muda, a orientação da face muda.

Bitmap – Mesmo que imagem rasterizada.

Buffer de profundidade – Usado internamente pelo sistema para determinar a profundidade do *pixel* em relação a outros na mesma posição durante a renderização.

Bump mapping – Técnica que através de uma textura extra, o mapa de rugosidade, perturba-se a normal à superfície no ponto onde se está calculando a intensidade luminosa. Isto equivale (visualmente) a ter-se uma superfície ondulada, embora não haja o aumento do número de polígonos.

Colorização – *Shading*, determinação da cor de cada ponto da imagem rasterizada, que é função da cor do objeto, da sua textura, da posição em relação às fontes luminosas, e nos métodos mais complexos da reflexão indireta da luz de outras superfícies da cena.

Colorizar – Aplicar colorização.

Curvas de Bézier – Tipo de curva paramétrica.

Framebuffer – Memória dedicada ao processador gráfico e usada para guardar os *pixels* renderizados antes de serem mostrados na tela.

Imagem rasterizada – Matrizes retangulares de pontos chamados *pixels* (*picture element*). Também conhecidas como imagens matriciais, *pixelmaps* (mapas de pixels) ou ainda *bitmaps*.

Impostores – *Billboards* que ao invés de usar como texturas imagens feitas por artistas, usam imagens geradas em tempo de execução, pelo sistema em si.

Mapas de iluminação – Textura especial que contém as informações das luzes estática de um certo ambiente, que é utilizada para determinar a intensidade dos *pixels* das texturas de fato.

Motor 3D – Software que processa os dados do mundo tridimensional, incluindo luzes, ações, estados em geral e renderiza segundo o ponto de vista do jogador ou câmera.

Motor do jogo – Conjunto de motores integrados, cada um responsável por uma determinada área.

MUD – *Multi-User Dungeon*, *Dimension*, ou *Domain*; RPG computacional multiusuário em rede, baseado em descrições textuais.

Multitextura – Múltiplas texturas; aplicação de mais de uma textura no polígono.

Mundo virtual – Modelos computacionais com os quais os usuários interagem nas simulações.

Normal – O vetor normal indica a orientação angular de um plano ou superfície.

Pixelmap – Mesmo que imagem rasterizada.

Procedural – Relativo a procedimento. Forma estabelecida por um conjunto de regras.

Profundidade de cor – Possibilidades de cores que o *pixel* pode assumir.

Rasterização – Mapeamento dos pontos colorizados no dispositivo de saída gráfico, obtendo-se a imagem rasterizada.

Renderização – Ato de renderizar a cena.

Renderizar – *To render*, na computação é a interpretação de algum código. Neste trabalho, trata-se de todo processamento e conversão dos dados tridimensionais em imagem bidimensional rasterizada.

Resolução – Número de *pixels* na imagem, representados pela multiplicação da largura pela altura.

RPG – *Role-Playing Game*, tipo de jogo, originalmente em tabuleiro (não-computacional), em que os jogadores assumem papeis de personagens ficcionais.

Script – Código baseado numa linguagem de programação interpretada que conecta diversos componentes pré-existentes para cumprir determinada tarefa.

Shader – Função escrita a partir de uma linguagem para colorização usada para definir cada *pixel*. *Shaders* de vértices permitiam controle das posições, cores e

coordenadas de texturas dos vértices enquanto os *shaders* de *pixels* permitiam controle na maneira como cada *pixel* individual seria renderizado na tela.

Sistema de partículas – Formalismo matemático usado para descrever fenômenos que são dinâmicos (temporais), compostos por pequenos componentes individuais e complexos.

Skybox – Cubo ou esfera com as normais voltadas para dentro (faces internas) e texturas cobrindo todos os polígonos que representam o céu.

Skyplane – Geometria que engloba o mundo virtual, desempenhando o papel de céu ou atmosfera, em forma de plano deformado (lençol com as pontas puxadas para baixo).

Sprites – Na computação gráfica, e particularmente nos jogos bidimensionais, são uma categoria de imagens desenhadas na tela. Usualmente pequenos (em relação às dimensões e resolução da tela) e parcialmente transparentes (ganhando formas não retangulares) são usados para personagens e objetos móveis.

Stencil buffer – Seção da memória gráfica que guarda os dados do "estêncil", usado para restringir o desenho de certas partes da tela.

Tempo real – Resposta correta de um sistema dentro de um prazo imposto pelas limitações do ambiente. Em termos gráficos, é quando não se tem percepção da renderização, conseguindo um mínimo de 15 imagens (quadros) por segundo e criando uma animação com sensação de movimento natural.

Textura – imagem “colada” no polígono como se fosse um adesivo, com objetivo de dar uma aparência mais realística.

Texturização – Ato de texturizar.

Texturizar – Aplicação de uma textura no polígono ou objeto.