OTIMIZAÇÃO DE AMBIENTES VIRTUAIS TRIDIMENSIONAIS
PARA PROCESSAMENTO EM TEMPO REAL NA WEB

Andrea Dantas Hecksher

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA CIVIL.

Aprovada por:

_____
Prof. Nelson Francisco Favilla Ebecken, D.Sc.

_____
Prof. Gerson Gomes Cunha, D.Sc.

_____
Prof. José Luis Drummond Alves, D.Sc.

_____
Profª. Isis Fernandes Braga, D.Sc.

RIO DE JANEIRO, RJ - BRASIL
SETEMBRO DE 2006

HECKSHER, ANDREA DANTAS

Otimização de Ambientes Virtuais Tridimensionais para Processamento em Tempo Real na Web [Rio de Janeiro] 2006

VIII, 66 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia Civil, 2006)

Dissertação – Universidade Federal do Rio de Janeiro, COPPE

1. Otimização de Ambientes 3D
2. Processamento em Tempo Real
3. Ambientes Virtuais Interativos
4. Ambientes 3D

I. COPPE/UFRJ II. Título (série).

Aos meus pais, Magno e Ana Maria,
por toda dedicação e incentivo,
e a meu filho, Bruno, por compreender
os meus momentos solitários de concentração.

AGRADECIMENTOS

Ao meu orientador, professor Nelson Francisco Favilla Ebecken, por acreditar e apoiar este trabalho e por sua presteza e objetividade.

Ao professor Gerson Gomes Cunha, a quem devo grande parte do conhecimento que sustenta esta dissertação.

Aos professores Isis Fernandes Braga e José Luis Drummond Alves, por aceitarem o convite para esta banca.

Ao professor Alexandre Gonçalves Evsukoff, pelo apoio prestado em várias etapas durante este curso de mestrado.

Aos amigos e professores Paulo de Oliveira Reis Filho e Fabio Siqueira D'Alessandri Forti, por terem apontado caminhos.

A todos os colegas do grupo GRVa / LAMCE, pelas dicas trocadas em nossa lista de discussão.

Aos funcionários da secretaria do PEC, pela qualidade e responsabilidade na prestação de serviços administrativos.

Ao professor Esteban Gonzalez Clua, do Departamento de Informática da PUC-Rio; a Leonardo Cardarelli Leite, mestre em Design pela PUC-Rio (tema: Jogos Multiplataformas); e a José Lúcio Mattos da Gama, vencedor do Festival de Jogos da CDG-Rio 2006, pela cortesia e atenção prestadas nas respostas aos questionários durante a CDG-Rio 2006.

A minha família e amigos, pelo incentivo.

Ao CNPq, pelo apoio financeiro recebido.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

OTIMIZAÇÃO DE AMBIENTES VIRTUAIS TRIDIMENSIONAIS PARA PROCESSAMENTO EM TEMPO REAL NA WEB

Andrea Dantas Hecksher

Setembro/2006

Orientador: Nelson Francisco Favilla Ebecken

Programa: Engenharia Civil

Este trabalho expõe e analisa técnicas e processos utilizados para otimização de ambientes virtuais tridimensionais interativos, visando o aumento da qualidade visual e a rapidez no processamento em tempo real.

O escopo desta dissertação é fazer um levantamento das técnicas aplicadas na construção e otimização de cenários para *games* 3D e, a partir de um exame crítico, propor a aplicação de algumas dessas técnicas em interfaces tridimensionais – ambientes para navegação interativa – para *web*.

O resultado da pesquisa é um estudo comparativo de técnicas atuais utilizadas para criação e aplicação de textura que componha resultados de efeitos de iluminação e de malhas complexas de modelos tridimensionais em um único mapa de imagem (*bitmap*) para ser aplicado a objetos 3D com baixo nível de detalhamento na modelagem.

Uma importante contribuição deste trabalho é a descrição do processo e a análise dos resultados da aplicação dessas técnicas na construção de um modelo de ambiente virtual tridimensional a ser processado em tempo real na *web*.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

OPTIMIZATION OF THREE-DIMENSIONAL VIRTUAL ENVIRONMENTS FOR REAL-TIME RENDERING IN THE WEB

Andrea Dantas Hecksher

September/2006

Advisors: Nelson Francisco Favilla Ebecken

Department: Civil Engineering

This work displays and analyzes techniques and processes used to optimize three-dimensional virtual environments, aiming to increase visual quality and accelerate real-time rendering.

The objective of this work is to survey techniques applied for construction and optimization of scenes for 3D video games and, from a critical examination, considering the application of some of these techniques in three-dimensional interfaces – interactive navigation – for web.

The result of the research is a comparative study of current techniques used for creation and application of a texture that composes results of illumination effects and complex meshes of three-dimensional models in a single image map (bitmap) to be applied to 3D objects with low level detailing in the modeling.

An important contribution of this work is the description of the process and the analysis of the results of the application of these techniques in the construction of a three-dimensional virtual environment model to be processed in real-time in the web.

Índice

capítulo 1. Introdução 1

capítulo 2. Ambientes Virtuais Interativos 4

2.1. Ambientes Virtuais Tridimensionais na Web 6

capítulo 3. Renderização e Processamento em Tempo Real 8

capítulo 4. Processamento Geométrico 11

capítulo 5. Superfícies de Modelos Tridimensionais 13

5.1. Materiais 13

5.2. *Shaders* 16

5.3. Texturas 18

5.4. Mapas de Imagem – *Bitmaps* 21

capítulo 6. Iluminação em Ambientes 3D 23

6.1. Luzes e Superfícies 23

6.2. *Raytracing* e *Radiosity* 25

capítulo 7. *Mapping* e *Unwrapping* 27

capítulo 8. Técnicas e Processos para Otimização de Ambientes 3D Interativos 29

8.1. *Vertex Color* 29

8.1.1. *Vertex Lighting* 31

8.2. ***Vertex Shader*** **e** ***Pixel*** **(ou** ***Fragment*****)** ***Shader*** 32

8.3. ***Render to Texture*** 35

8.4. ***Bump Map*** **e** ***Normal Map*** 36

8.5. Redução de Complexidade Geométrica e Visual 40

8.5.1. LOD 40

8.5.2. ***Mipmaps*** 41

capítulo 9. Aplicação das Técnicas de Otimização em um Protótipo de Ambiente 3D para Web 43

capítulo 10. Conclusão 60

capítulo 11. Bibliografia 64

1. Introdução

O desenvolvimento de *softwares* e algoritmos para geração de gráficos tridimensionais e de *hardwares* dedicados a renderização e processamento geométrico impulsionou a área de visualização em diversos setores: entretenimento, visualização científica e simulação de processos industriais. Acelerando e aumentando a qualidade tanto na etapa de produção quanto de exibição de projetos de animação; composição de cenas virtuais com imagens reais para cinema, TV e impressos publicitários; jogos digitais; maquetes e projetos arquitetônicos, cenográficos ou de engenharia.

Visualização [1] é o conjunto de tecnologias que, compartilhando uma mesma base tecnológica de informação e comunicação, permite a simulação virtual de ambientes e processos em vários setores econômicos. Assim, as tecnologias de visualização mantêm relação estreita com diversos outros conceitos como: simulação, computação gráfica, efeitos especiais, interatividade, realidade virtual, produção de conteúdo digital para cinema e televisão, indústrias de energia e defesa, agronegócio, aeronáutica e jogos digitais.

Com aplicações em diversos setores da economia, as tecnologias que permitem a representação gráfica visual interativa – efeitos especiais, realidade virtual, simulações, hipermídia – constituem uma área promissora da indústria internacional de *software*. O conceito de Tecnologias de Visualização (TecVis) estende-se além do subconjunto, relativamente amplo, das Tecnologias da Informação e Comunicação (TICs), que têm em comum a simulação de ambientes e processos virtuais, utilizando como base a matemática, a física, a lógica e a informática [1].

Os ambientes virtuais tridimensionais interativos, tais como jogos digitais 3D ou maquetes virtuais, precisam de um tratamento artístico apurado nas etapas de modelagem, texturização e iluminação para serem realmente atrativos, persuasivos e proporcionarem a sensação de imersão. Por outro lado, a visualização e a interatividade desses ambientes dependem de capacidade de processador, memória e placa gráfica do computador, do peso do arquivo que será executado e da complexidade dos cálculos que serão necessários para exibir cada *frame* (quadro) em tempo real. Portanto a preocupação na etapa de construção desses ambientes não deve ser apenas estética, mas também funcional. Deve ser considerada a aplicação de técnicas para otimização da

malha poligonal dos modelos e para preparação de texturas que minimizem o tamanho do arquivo e a quantidade de cálculos para sua exibição sem muita perda de qualidade visual no ambiente, visando não sobrecarregar o processamento em tempo real.

Partindo do conhecimento prévio de que para um bom processamento em tempo real de um ambiente virtual tridimensional os objetos que fazem parte desse cenário devem ter o menor número de polígonos possível, que acarreta perda de qualidade visual, serão expostas e analisadas técnicas que através de imagens de *bits* (*bitmaps*) conseguem resgatar ou, ainda, aumentar a qualidade visual de objetos tridimensionais com baixo nível de detalhamento. Essas técnicas, que serão descritas no capítulo 8, consistem na criação de materiais que utilizam o canal de mapa de *bump* com mapas de saliências (*bump map*) ou com mapas de perpendiculares, também conhecidos como mapas normais (*normal map*).

Além da redução do número de polígonos, outra forma de otimização de modelos 3D são as técnicas que permitem reunir características da superfície de um objeto e efeitos que a iluminação pode gerar sobre sua superfície em um único *bitmap* ou diretamente nos vértices de sua malha, diminuindo a quantidade de cálculos na etapa de renderização.

Todas as técnicas abordadas nesta dissertação foram escolhidas e descritas com base na bibliografia apontada no final deste trabalho e nas palestras e questionários feitos para pesquisadores e desenvolvedores na área de *games 3D*, durante as *Conferências de Desenvolvedores de Games* realizadas no Rio de Janeiro – *CDG 2005 e 2006*.

Apesar de existirem diversos *softwares* para criação de ambientes virtuais tridimensionais, ricos no quesito qualidade visual, que disponibilizam ferramentas e algoritmos para criação de texturas para superfícies de modelos 3D e para simulação de iluminação global, a maior parte dessas aplicações está voltada para produção de imagens e filmes pré-renderizados. Este trabalho apresenta um estudo sobre ferramentas e técnicas aplicadas à construção de ambientes virtuais para renderização em tempo real com a mesma preocupação de agregar valor a partir da aplicação de texturas e reprodução de iluminação direta e indireta. A escolha das técnicas será ditado com base no levantamento de requisitos e limitações da criação desses ambientes para navegação em tempo real na *web*, a partir da linguagem VRML (*Virtual Reality Modelling Language*); linguagem aberta e multi-plataforma para descrição de cenas tridimensionais.

A preparação desses mundos virtuais para *web* é muito semelhante à construção de cenários tridimensionais para jogos digitais 3D. As técnicas descritas nesta dissertação, em sua maioria, são aplicadas no preparo de ambientes para *games* e podem ser úteis, também, para acelerar o processo de renderização de animações e filmes.

2. Ambientes Virtuais Interativos

Os *adventures*, denominação dada aos primeiros jogos digitais, em sua maioria, não possuíam gráficos, apenas textos, porque eram desenvolvidos para atender ao mercado de micros da geração de 8 bits. A interface era essencialmente interativa e textual. O procedimento interativo, geralmente, era executado pelo usuário através de comandos por digitação de texto e as respostas a suas ações eram exibidas na tela também em formato de texto. Quando o jogo exibia imagens, eram gráficos simples, estáticos ou com pouca animação.

SpaceWar, desenvolvido no *MIT* em 1962, foi o primeiro jogo a apresentar gráficos. Apesar da simplicidade na simulação física e na apresentação gráfica e embora tivesse origem um ambiente acadêmico, foi o precursor de uma nova era para o mercado de *games*, marcando uma mudança na mentalidade dos desenvolvedores, que passariam a visar um público cada vez mais exigente em qualidade visual. A demanda por *games* com animações complexas, simulações físicas, efeitos especiais e ambientes mais realísticos justifica a organização de equipes multidisciplinares para o desenvolvimento de jogos, bem como o surgimento de novos cursos com ênfase em desenvolvimento e em artes e design, especializantes ou profissionalizantes, voltados para desenvolvimento e design de jogos e aplicações 3D interativas. Explica também o aparecimento de novas especializações, como a do *level designer*, responsável pela criação de cenários 3D complexos para exploração virtual e jogos.

A classificação de um jogo, em relação ao aspecto gráfico, pode ser: 2D, 3D e misto. Nos jogos 3D as imagens são processadas pelo computador em tempo real a partir de dados de modelos criados em *softwares* de modelagem 3D.

> Jogos mistos apresentam tanto gráficos 3D quanto gráficos 2D, como o caso do *Escape from Monkey Island*, da *Lucas Arts*, que apresenta personagens 3D integrados em um cenário formado por um conjunto de imagens bidimensionais organizadas em camadas.
>
> Enquanto jogos 2D necessitam apenas de procedimentos para posicionar e desenhar as imagens, jogos 3D ou mistos necessitam de um conjunto de procedimentos para gerar as imagens a partir de dados descritivos dos objetos. A esse conjunto de procedimentos foi atribuído o nome de Renderizador [2].

O ambiente virtual de um *game* 3D define objetos, formas de iluminação e tem grande influência na definição das funcionalidades do motor de um jogo, como a determinação dos limites de movimentação.

> Do ponto de vista de programadores, um motor é uma ferramenta que permite novos jogos serem construídos a partir das interfaces e funcionalidades que ele provê, cabendo aos programadores implementar as peculiaridades de um jogo específico. Do ponto de vista de requisitos, espera-se que um motor assuma uma série de tarefas e disponibilize uma série de módulos de fácil utilização e extensão para cada novo jogo.
>
> Motores são considerados peças chave no desenvolvimento de um jogo, pois são os responsáveis pelo controle e processamento básico de todas as mídias envolvidas em um jogo. O motor abstrai a complexidade da implementação de módulos básicos do jogo e, assim, facilita o desenvolvimento permitindo que se concentre os esforços de projeto nas questões lógica, visual e sonora do produto e não em questões de codificação [2].

Os ambientes de *games* 3D podem ser classificados como: ambientes internos, ambientes externos, espaço sideral e ambientes mistos. Os ambientes internos, também denominados de *dungeons* ou *masmorras*, são caracterizados por ambientes fechados, constituídos por salas e suas conexões, com iluminação geralmente artificial, provenientes de fontes de luz estáticas, em sua maioria, como tochas ou *spotlights*. O *Quake* e o *Doom*, ambos da *Id Software*, são exemplos de jogos de ambientes internos.

Os ambientes externos são caracterizados por ambientes abertos, construídos a partir de geradores de terreno ou por mapas de altitude. A iluminação, geralmente, é ambiente e o céu é representado por um *SkyBox* (caixa ou domo texturizado). Os ambientes externos são encontrados, principalmente, em jogos de simulação, como o *F1 World Grand Prix* da *Eidos Interactive*.

Espaço Sideral são ambientes, como o próprio nome já diz, que simulam o espaço sideral. A iluminação é variável porque simula a iluminação dos astros luminosos disponíveis no sistema sideral em que o personagem se encontra. *Privateer I* e *II* da *Origin Systems* possuem a ambientação de Espaço Sideral.

Ambientes mistos são compostos por mais de um dos três tipos de ambientes anteriores, no intuito de criar uma aplicação mais atrativa ao consumidor.

2.1. Ambientes Virtuais Tridimensionais na Web

As primeiras idéias de mundos virtuais na *web* caracterizavam-se por interfaces bidimensionais utilizando a linguagem HTML (*Hipertext Markup Language*). A linguagem VRML (*Virtual Reality Modeling Language*), ou Linguagem para Modelagem de Realidade Virtual, é um padrão de aplicativos de realidade virtual utilizado na *Internet* que viabilizou a descrição de objetos e ambientes virtuais tridimensionais interativos para *web* com navegação em tempo real. Por tempo real entende-se um modo de processamento cujo resultado é obtido em um curto espaço de tempo após a entrada dos dados.

Em um ambiente VRML, a velocidade na renderização das cenas deve ser suficientemente rápida, como um processo contínuo, para que o usuário não perceba a troca de imagens, como nos *games* 3D. A caracterização de navegação e interatividade em tempo real se deve ao fato dos ambientes em VRML permitirem interação através da rotação e translação do espaço e dos objetos e através de sensores que habilitam eventos.

VRML é uma linguagem simples utilizada para descrever objetos e mundos interativos 3D, apesar de existirem programas específicos para criação de arquivos VRML, esses podem ser substituídos por um editor de texto básico – como o *Bloco de Notas* da *Microsoft*. Todo o ambiente, com objetos, câmeras, luzes e sensores, que possibilitam a interatividade, são descritos como texto. VRML não é compilada, é um simples arquivo texto que pode ser interpretado.

> Por meio desta linguagem, escrita em modo texto, é possível criar objetos tridimensionais podendo definir cor, transparência, brilho, textura (associando-a a um *bitmap*). Os objetos podem ser formas básicas, como esferas, cubos, ovóides, hexaedros, cones, cilindros, ou formas criadas pelo próprio programador, como as extrusões.
>
> Além dos objetos, também é possível acrescentar interatividade a estes por meio de sensores, podendo assim deslocá-los de posição, acrescentar luz, produzir um som quando o objeto é clicado ou o *avatar* simplesmente se aproxima dele, e abrir um arquivo ou página da Web, ou ainda outra página em VRML, quando o objeto é acionado [3].

O VRML permite, também, a inserção de recursos multimídia, como imagens, áudio e vídeo integrados ao ambiente 3D, propiciando uma interface atrativa com maior envolvimento para o usuário.

> A primeira versão (VRML 1.0) que inicialmente se chamava *Virtual Reality Markup Language*, foi desenvolvida pela *Silicon Graphics* baseada no formato dos arquivos do *Open Inventor* e comportava apenas objetos estáticos. A segunda versão (VRML 2.0) foi inicialmente desenvolvida por um consórcio formado pela *Silicon Graphics* com contribuição da *Sony Research*, *Mitra* e muitos outros. Esta versão foi então comentada e revisada por uma lista de discussão moderada formada pela comunidade *web3d*. Em dezembro de 1997, com pequenas modificações o VRML97 substituiu oficialmente o VRML 2.0 como padrão internacional (ISO/IEC 14772) e em 2002 foi adicionada a segunda parte do padrão descrevendo o EAI (*External Authoring Interface*). [4]

Os programas interpretadores de VRML – como o *Blaxxun Contact* da *Blaxxun* [5], *Cortona VRML Cliente* da *ParallelGraphics* [6], *Cosmo Player* da *SGI* [7] e o *Octaga Player* da *Octaga* [8] – são chamados de *3D player* ou *plug-ins* para VRML, alguns possibilitam também a visualização de X3D. Esses interpretadores de VRML e X3D podem funcionar como um aplicativo autônomo ou como *plug-ins* para a maioria dos internet *browsers*.

O padrão X3D [9] é uma evolução do VRML, também é um padrão aberto e *royalty-free* para distribuir conteúdo 3D na web. Incorpora os avanços dos recursos disponíveis nos últimos dispositivos gráficos e pode representar e comunicar as cenas e os objetos 3D usando XML (*Extensible Markup Language*).

3. Renderização e Processamento em Tempo Real

Renderização ou *rendering* é a decodificação para exibição em determinada mídia de um mundo virtual criado digitalmente pela descrição de pontos (vértices), que definem polígonos, determinados por coordenadas bi ou tridimensionais em um espaço geométrico. Assim como, os diversos algoritmos, formas e metodologias de cálculos de textura, efeitos de iluminação e cores aplicados nesse universo vetorial [10].

Renderizar é o processo de gerar uma imagem de um modelo, através de um programa. O modelo é uma descrição, geralmente, de objetos tridimensionais, por meio de uma linguagem computacional ou uma estrutura de dados, contendo a geometria, o ponto de vista, a textura e as informações de iluminação. A imagem gerada é um *bitmap*; um arquivo de dados ou uma estrutura que representa uma grade, geralmente retangular, de *pixels*, pontos de cor, para saída em um monitor de computador, em papel, ou em qualquer outro dispositivo de exposição que tenha como suporte o espaço bidimensional. A cor de cada *pixel* é definida individualmente, geralmente, determinada por três *bytes*, um para cada canal de cor: vermelho, verde e azul.

Os termos *render* ou *rendering* também podem ser usados para designar o processamento do resultado da edição digital de um vídeo, ou seja, o cálculo dos efeitos inseridos durante a etapa de edição.

O *render* de um projeto deve ser feito tendo em mente a mídia que será veículo para exibição desse, considerando-se as especificações e limites técnico desses meios. O resultado pode ser uma única imagem ou uma seqüência de imagens que simulem movimento.

Existe uma grande variedade de renderizadores: integrados em *softwares* de modelagem e animação, autônomos ou projetos livres (*open-source*). Analisando os sistemas de renderização de alguns dos *softwares* de modelagem mais populares encontram-se métodos híbridos: *3ds max* [11] – *scanline*, *raytrace* e *radiosity*; *Lightwave* [12] – *Z-buffer*, *raytrace* e *radiosity*; *Maya* [13] – *A-buffer*, *scanline* e *raytrace*. O *Mental ray* [14] – renderizador desenvolvido pela *Mental Images* – se tornou bastante conhecido e agora já vem integrado aos softwares *Maya* e *3ds max* como uma alternativa aos *renderers* padrões desses aplicativos. O *Mental ray* é um render excelente porém requer a utilização dos seus próprios *shaders*. Existem outros *plug-ins* de *render* para o *3ds max*,

um muito recomendado é o *V-Ray* [15] da *Chaos Group*. Para o *Lightwave* o *plug-in* de *render* mais indicado é o *G2* [16] da *Worley Labs*.

As tecnologias de renderização que dão saída para imagens ou animações vetorizadas e têm integração com o programa *Flash* [17] são uma boa alternativa para diminuir o tamanho dos arquivos e acelerar o processamento de modelos e ambientes virtuais tridimensionais para a *web*. No entanto, essas tecnologias não são o foco desta dissertação porque criam imagens com estilo *cartoon* ou com sombreamento através de dégradé, modelos e cenas pouco realísticos. São exemplos dessas tecnologias: o *Swift 3D* [18] da *Electric Rain* – programa e *plug-in* para o *3ds Max* e o *Lightwave* – e o *Vector Renderer* incorporado ao *Maya*.

Render é, portanto, um algoritmo ou aplicativo que possibilita às tecnologias de visualização a simulação de ambientes e processos virtuais, utilizando como base a matemática, a física, a lógica e a informática. Nesse contexto, existem dois tipos de renderizações: o "*pre-rendering*" (pré-processamento computacional) usado, por exemplo, para criação de filmes e o "*real-time rendering*" (processamento em tempo real) usado para criar ambientes interativos, como nos *games* 3D, em que o desempenho depende da potência das placas gráficas e dos aceleradores de *hardware* 3D.

Renderizar é sintetizar por meio de uma imagem um ambiente virtual tridimensional. No contexto de um jogo 3D, várias cenas devem ser renderizadas por segundo, portanto torna-se extremamente importante a velocidade no processamento dessas cenas. Essa velocidade pode ser expressa em quadros por segundo (FPS), que é o número de imagens geradas por segundo, ou em *hertz* (HZ). Esse sistema de renderização é denominado renderização ou processamento em tempo real porque o resultado deve ser obtido em um curto espaço de tempo após a entrada dos dados. A renderização em tempo real, até pouco tempo atrás, só era viável em estações gráficas dedicadas ou com otimizações por *software*. No caso dos motores de jogos, sistemas fechados, pouco modulares e caros. O avanço na área de *hardware* proporcionou a criação de aplicações para as mais diversas áreas.

Sistemas de renderização em tempo real constituem a área mais interativa da computação gráfica. Nesses sistemas não existe uma linearidade, a seqüência é definida ou interrompida pelo usuário. Esse interfere na cena visualizada e, cada ação ou reação sua, afeta e define o quadro seguinte que será gerado. Para propiciar a sensação

de imersão em um processo contínuo, a velocidade de renderização deve ser rápida o suficiente para que o usuário não percebe imagens em uma taxa discreta [19].

Em uma cena com muitos elementos diferentes, a função principal de um *render* está em determinar e calcular que partes da geometria devem ser representadas nos *pixels* da imagem. Na renderização são realizadas operações como remoção de polígonos que não estejam contidos no volume de visão definido pelo plano de recorte da câmera e calculado pelo modelo de projeção ou, ainda, a remoção de objetos (ou partes desses) que estejam escondidos por outros.

De acordo com *Kerlow* [20], existem três métodos de renderização que podem ser utilizados: *Z-buffer*, *Raytracing* e *Radiosity*. O método *Z-buffer* [21] leva em consideração o valor do eixo *Z* da posição de cada objeto de um ambiente virtual, ou seja, a profundidade na cena. Esses valores são armazenados em um *buffer*; o *Z-buffer* que é a área da memória de vídeo da placa gráfica destinada a armazenar o valor do eixo *Z*. O *Z-buffering* correspondente a um algoritmo que verifica e define quais objetos (ou partes deles) são visíveis em relação à câmera e quais se apresentam escondidos atrás de outros objetos. Nos *games* o método provavelmente mais usado é o *Z-buffer real-time rendering*. Os métodos *raytracing* e *radiosity* serão explicados no capítulo 6.

Quando um objeto é renderizado pelo algoritmo de *z-buffer*, a profundidade (coordenada de *z*) de uma parte do objeto, que será representado em um determinado *pixel*, fica armazenada no *z-buffer* (ou buffer de profundidade). Ficam registrados no *buffer*, também os valores de cor e intensidade para cada *pixel*. Se no enquadramento da cena, de acordo com a posição da câmera, existe um outro objeto (ou parte desse) que deve ser renderizado no mesmo *pixel*, a placa gráfica compara as duas profundidades e escolhe a que está mais perto do observador. A profundidade escolhida é conservada então no *z-buffer*, descartando o valor anterior armazenado. No final, o *z-buffer* permitirá que a placa gráfica reproduza corretamente a percepção de profundidade: um objeto próximo esconde o que está mais distante. Este processo também é conhecido como *z-culling*. Após o processamento de todos os polígonos, se não for solicitado nenhum cálculo de *raytracing* ou de *radiosidade*, o buffer terá toda a cena armazenada em uma disposição bidimensional para ser representada em *pixels* na tela do computador ou em qualquer dispositivo 2D.

4. Processamento Geométrico

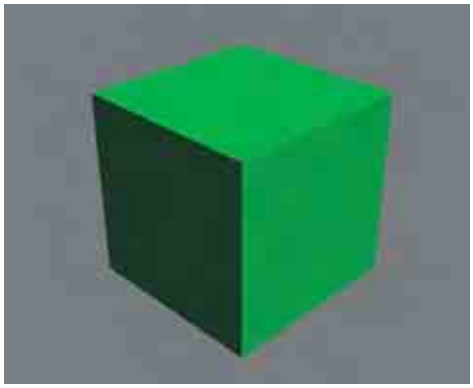
A maior parte do *pipeline* de renderização em tempo real de ambientes tridimensionais é de natureza matemática, com isso faz-se necessário a definição de um conjunto de estruturas de dados descritivos para representar os objetos que irão compor o sistema. Essa estrutura descritiva é, geralmente, a maior responsável pela performance no sistema de visualização. As estruturas podem ser vetoriais ou paramétricas. O processamento de uma cena descrita com uma quantidade muito grande de dados será problemático para um sistema com memória limitada. Da mesma forma, uma cena representada por uma função pode demandar muito tempo para o processamento, diminuindo o desempenho na renderização para visualização *real-time*.

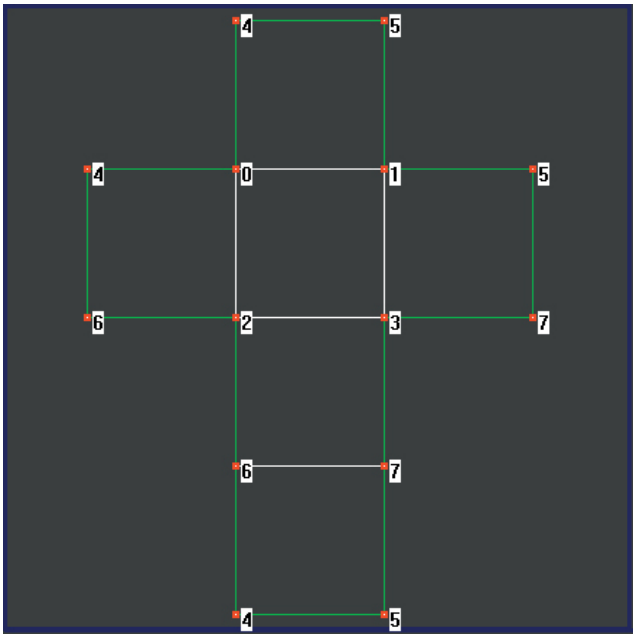
> Algumas bibliotecas gráficas podem implementar métodos paramétricos de descrição de objetos. Isto é, objetos tridimensionais podem ser descritos a partir de uma função f(u,v), muito útil para terrenos, objetos quádricos ou superfícies não “fechadas”, como um cilindro, um círculo ou superfícies NURBS.
>
> Por não ter um volume grande de dados essa representação consome pouca memória, porém caso a biblioteca ou a placa gráfica não suporte essa representação ela pode (e deve) ser convertida para uma representação poligonal, o que pode consumir tempo de processamento [2].

Apesar de exigir uma enorme quantidade de dados para representação de um único modelo, a estrutura vetorial é a mais utilizada nas aplicações 3D atuais. Nessa estrutura, os objetos poligonais são descritos através de um conjunto de vértices ou vetores que se correlacionam. Existem basicamente duas regras de descrição de uma estrutura vetorial: Conjunto de Faces e Ordenação de Vértices.

No Conjunto de Faces o algoritmo de renderização percorre cada face, buscando, ordenadamente, os vértices de sua borda para formar o polígono. Esse tipo de representação é contraproducente para objetos poligonais mais complexos onde um mesmo vértice pode ser compartilhado por duas ou mais faces. Dessa forma, cada vértice será processado tantas vezes quanto for o número de faces que o compartilham, retardando o processamento devido ao volume de dados.

O método de malha por ordenação de vértices ou *stripping* soluciona o problema apresentado no Conjunto de Faces. Nesse método os objetos poligonais são descritos por uma lista ordenada de vértices. As faces poligonais que irão definir o objeto são formadas por uma regra de composição que segue a ordem da lista de vértices.






```
*MESH {
        *TIMEVALUE 0
        *MESH_NUMVERTEX 8
        *MESH_NUMFACES 12
        *MESH_VERTEX_LIST {
                *MESH_VERTEX    0       -1.5000 -1.5000 -1.4873
                *MESH_VERTEX    1       1.5000  -1.5000 -1.4873
                *MESH_VERTEX    2       -1.5000 1.5000  -1.4873
                *MESH_VERTEX    3       1.5000  1.5000  -1.4873
                *MESH_VERTEX    4       -1.5000 -1.5000 1.5127
                *MESH_VERTEX    5       1.5000  -1.5000 1.5127
                *MESH_VERTEX    6       -1.5000 1.5000  1.5127
                *MESH_VERTEX    7       1.5000  1.5000  1.5127
        }
```

FIGURA 4.a. A primeira imagem é um render de um cubo, ao lado a planificação (*unwrap*) do cubo com a indicação de seus vértices. A lista é um trecho do código ASCII que representa o cubo acima, é um exemplo de lista de ordenação de vértices.

5. Superfícies de Modelos Tridimensionais

> "É importante lembrar que computadores são apenas máquinas velozes de calcular. O objetivo de um bom projeto de *software* é fazer o usuário esquecer isto. (...) Se o usuário não souber quais combinações de cores e formas são visualmente agradáveis, o computador não poderá ajudá-lo. Se o usuário não tiver o conhecimento e o domínio sobre um conjunto de ferramentas e práticas para definir materiais e iluminações de ambientes convincentes, o *software* não poderá compensar esta deficiência. Se o usuário não entender a teoria de como luz e superfície dos materiais trabalham juntas para alcançar a aparência desejada, ele ficará preso a tentativas e erros e talvez nunca alcance o resultado esperado. Um bom artista compreenderá inteiramente a estética, a teoria e a tecnologia do *software*." *Dennis Summers* [10]

5.1. Materiais

A criação de materiais para serem aplicados a modelos virtuais tridimensionais é um processo de simulação das propriedades da superfície de objetos que pertencem ao mundo real ou, ainda, a criação de superfícies com aparência inusitada.

No universo 3D os termos material, *material surface*, *surface material* são usados para descrever cor, padrão, transparência, reflexo, ou seja, todas as características que possam definir o aspecto da superfície de uma malha geométrica.

Nos programas de modelagem 3D existem os painéis para criação e edição de materiais (*Material Editor*), onde podem ser construídos materiais com baixa complexidade, como no caso de um material que apresente apenas uma cor sólida, ou de alta complexidade, como uma textura multi-imagem.

Um material pode conceder a um objeto diversas características diferentes; uma aparência de velho ou de novo, de sujeira ou de limpeza. Um material, também, poderá ser brilhoso ou fosco, rugoso ou liso, características que influenciarão na forma como a luz será refletida ou absorvida pela superfície de um objeto.

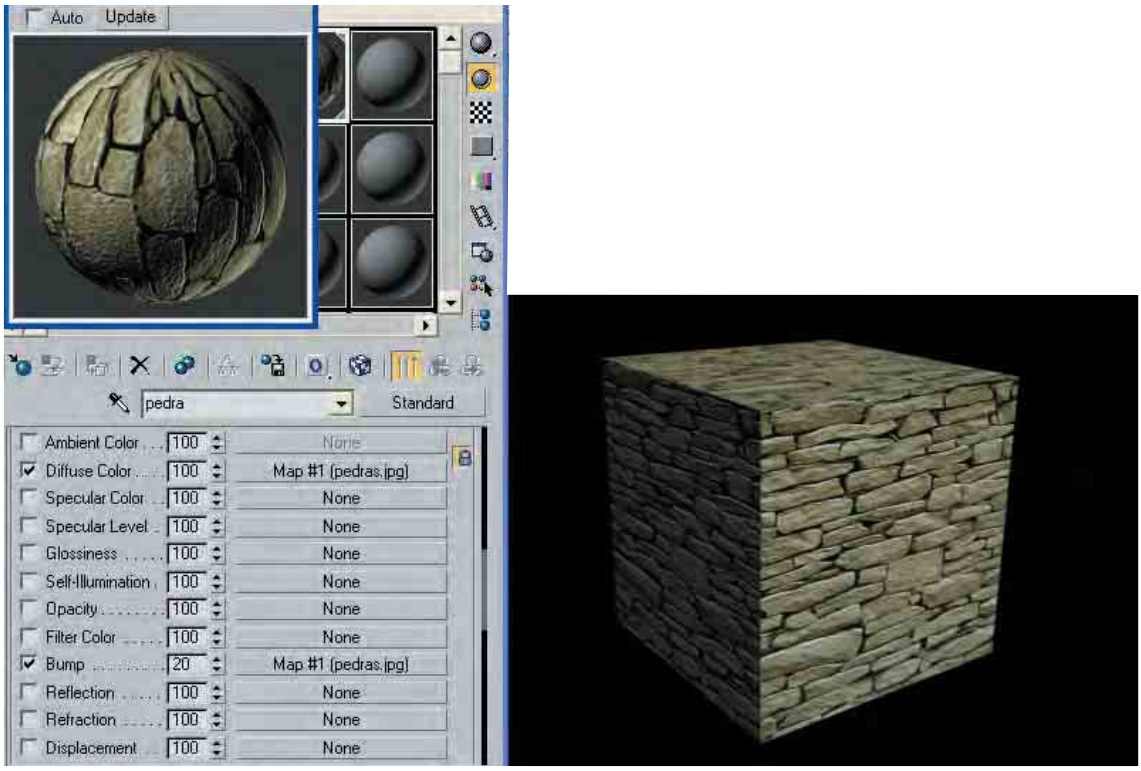
Na construção de um material é fundamental considerar como o material e a iluminação irão interagir. Nos editores de materiais dos programas de modelagem 3D

existem diversos tipos de materiais disponíveis para que a partir deles o usuário possa construir seu próprio material. Alguns materiais tentam simular superfícies, mas há o caso do *Ray-Trace*, por exemplo, que tenta modelar fisicamente uma superfície. O *Ray-Trace*, quando aplicado a um objeto, calcula, no momento da renderização, os raios de luz da cena que incidem neste objeto para gerar efeitos como luminosidade, iluminação extra, translucidez e fluorescência. Por isso, os materiais *Ray-Trace* levam mais tempo para serem renderizados [22].

Um modelo 3D é criado por equações matemáticas ou representado por uma lista que descreve a localização de cada vértice no espaço 3D e que serão processados ou interpretados para serem projetados na tela do monitor apenas em duas dimensões criando a ilusão de perspectiva e profundidade. A superfície dos objetos 3D também podem apresentar cores e outras características associadas a cada vértice ou por aplicação de textura (mapas de imagens).

Portanto, a criação de superfície diversas e complexas para modelos 3D pode ser possível através da aplicação de materiais. Os materiais, por sua vez, são construídos com o auxílio de *shaders* (algoritmos de sombreamento) e, freqüentemente, enriquecidos visualmente por mapas de imagens: mapas de *bits* (*bitmaps*) ou mapas de procedimento (construídos por algoritmos).

Algumas características comuns de superfícies de materiais incluem *diffuse*, *specular*, *ambient*, *transparency*, *self-illumination*, *bump*, *reflection*, *refraction* e outras. A maioria das características podem ser alcançadas com o uso de mapas de textura – mapas procedurais e de imagem – e algumas são obtidas somente pelos valores cinzentos do mapa da textura [10].




```
*MATERIAL_LIST {
        *MATERIAL_COUNT 1
        *MATERIAL 0 {
                *MATERIAL_NAME "pedra"
                *MATERIAL_CLASS "Standard"
                *MATERIAL_AMBIENT 0.4471        0.4471  0.4471
                *MATERIAL_DIFFUSE 0.4471        0.4471  0.4471
                *MATERIAL_SPECULAR 0.9000       0.9000  0.9000
                *MATERIAL_SHINE 0.0000
                *MATERIAL_SHINESTRENGTH 0.0000
                *MATERIAL_TRANSPARENCY 0.0000
                *MATERIAL_WIRESIZE 1.0000
                *MATERIAL_SHADING Blinn
                *MATERIAL_XP_FALLOFF 0.0000
                *MATERIAL_SELFILLUM 0.0000
                *MATERIAL_FALLOFF In
                *MATERIAL_XP_TYPE Filter
                *MAP_DIFFUSE {
                        *MAP_NAME "Map #1"
                        *MAP_CLASS "Bitmap"
                        *MAP_SUBNO 1
                        *MAP_AMOUNT 1.0000
                        *BITMAP "D:\Andrea\Tese2006\junho_disserta__o\Figuras\Material01\pedras.jpg"
                        *MAP_TYPE Screen
                        *UVW_U_OFFSET 0.0000
                        *UVW_V_OFFSET 0.0000
                        *UVW_U_TILING 0.5000
                        *UVW_V_TILING 0.5000
                        *UVW_ANGLE 0.0000
                        *UVW_BLUR 1.0000
                        *UVW_BLUR_OFFSET 0.0000
                        *UVW_NOUSE_AMT 1.0000
                        *UVW_NOISE_SIZE 1.0000
                        *UVW_NOISE_LEVEL 1
                        *UVW_NOISE_PHASE 0.0000
                        *BITMAP_FILTER Pyramidal
                }
                *MAP_BUMP {
                        *MAP_NAME "Map #1"
                        *MAP_CLASS "Bitmap"
                        *MAP_SUBNO 8
                        *MAP_AMOUNT 0.2000
                        *BITMAP "D:\Andrea\Tese2006\junho_disserta__o\Figuras\Material01\pedras.jpg"
                        *MAP_TYPE Screen
                        *UVW_U_OFFSET 0.0000
                        *UVW_V_OFFSET 0.0000
                        *UVW_U_TILING 0.5000
                        *UVW_V_TILING 0.5000
                        *UVW_ANGLE 0.0000
                        *UVW_BLUR 1.0000
                        *UVW_BLUR_OFFSET 0.0000
                        *UVW_NOUSE_AMT 1.0000
                        *UVW_NOISE_SIZE 1.0000
                        *UVW_NOISE_LEVEL 1
                        *UVW_NOISE_PHASE 0.0000
                        *BITMAP_FILTER Pyramidal
                }
        }
}
```

FIGURA 5.1.a. Acima o editor de materiais do *3ds max* com um material preparado que contém uma imagem no canal *diffuse* e uma instância dessa imagem no canal de *bump*. Ao lado, o *render* do cubo com a textura aplicada. A lista é um trecho do código ASCII que representa o material do cubo acima.

5.2. *Shaders*

A primeira parte a ser definida em um material é o *shader* ou sombreador, um algoritmo que determina a forma como uma superfície é afetada pela luz. O trabalho de iluminação de uma cena não termina com a criação e definição da posição de fontes de luzes. No processo de renderização, a maneira como uma luz é refletida ou é absorvida pela superfície de um objeto é tão importante quanto o posicionamento das luzes em relação a este objeto. Para ter um bom controle deste processo e conseguir alcançar os resultados esperados, é preciso ter um conhecimento prévio de como as superfícies 3D respondem a exposição à luz e como ajustar os *shaders* dos materiais [23].

Cada shader possui um conjunto de parâmetros que podem ser ajustados pelo usuário. Existem parâmetros que são comuns entre os sombreadores como: *ambient*, *diffuse*, *self-illumination*, *opacity* e *specular highlights*. Existem vários *shaders*, alguns atendendo a mesma finalidade, porém com nomes diferentes em cada *software* 3D. Os mais comuns, como *Phong* e *Blinn*, são geralmente úteis para materiais isotrópicos.

Shaders são modelos matemáticos projetados para simular e ajustar a aparência de tipos diferentes de superfície de materiais sob circunstâncias diferentes de iluminação. Um shader determina, em seu modo de utilização mais básico, a uniformidade ou a lisura (*smoothing*) através das bordas de polígonos contíguos. Tem como principal função determinar o modo como uma superfície responde a propagação da luz, se a superfície terá um reflexo especular ou difuso. Uma superfície difusa é aquela em que a luz quando é refletida se espalha para todas as direções porque contém pequenas irregularidades, às vezes imperceptíveis. Uma superfície especular é definida por uma reflexão coerente, bons exemplos são as superfícies refletoras como alguns tipos de metais polidos e, principalmente, o espelho.

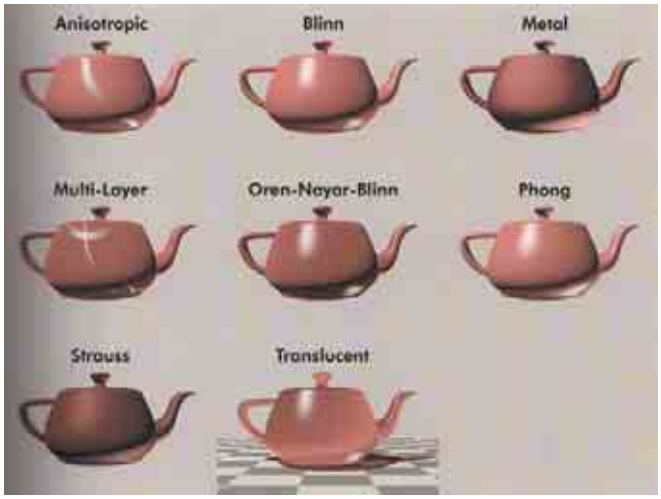


FIGURA 5.2.a. *Shaders* do *3ds max.*
Imagem do livro *Texturing: Concepts and Techniques.*

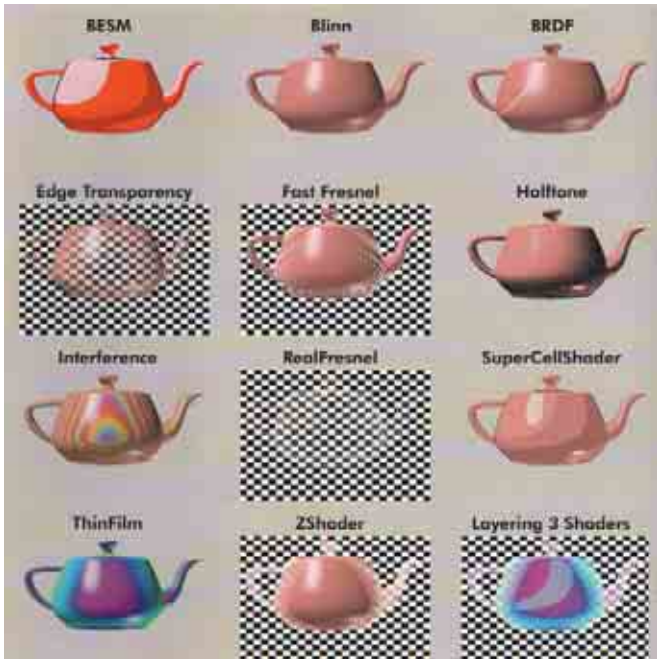


FIGURA 5.2.b. *Shaders* do *Lightwave.*
Imagem do livro *Texturing: Concepts and Techniques.*

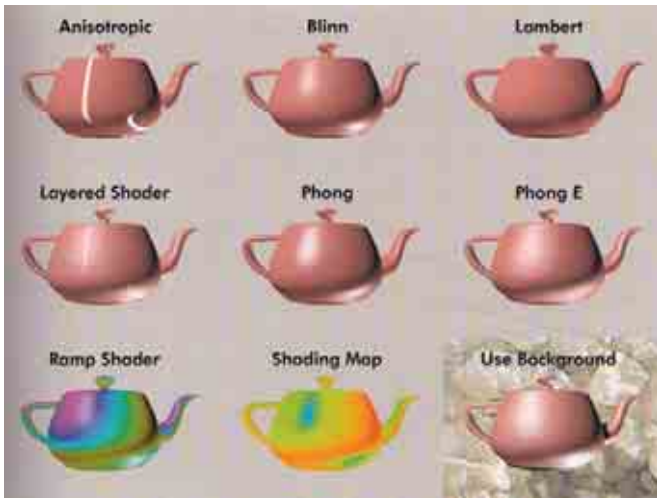


FIGURA 5.2.c. *Shaders* do *Maya.*
Imagem do livro *Texturing: Concepts and Techniques.*

5.3. TEXTURAS

Na história das artes plásticas e das artes visuais, a textura é parte importante e, às vezes, fundamental para compor várias obras. Na natureza existe uma grande variedade de texturas e em cada textura de uma mesma superfície diversos ruídos, deformações de fidelidade na transmissão de uma mensagem visual, que conferem a cada objeto e a cada parte desse, características únicas. Os materiais no mundo real, em sua maioria, não são isotrópicos e isso determina o grande abismo ainda percebido entre o universo real e o virtual. Neste último ainda é muito comum encontrar objetos 3D com aparência sintética e muito uniforme.

De acordo com *Summers* [10], outro aspecto da textura é sua relação com a forma e o volume em um objeto. Com o advento da modelagem digital e da texturização artificial de modelos, pesquisas têm sido feitas na tentativa de determinar se variações no modo como a textura é aplicada em um objeto 3D podem afetar a percepção dos observadores quanto a forma e a tridimensionalidade de um objeto virtual. Segundo o autor, os resultados têm mostrado que a compreensão da forma pelo observador pode ser ampliada ou anulada de acordo com as diferentes orientações – mudanças na aplicação do mapeamento – de uma mesma textura sobre um mesmo objeto.

Em ambientes sem iluminação, a textura também cumpre o importante papel de ajudar na percepção da tridimensionalidade do objeto. Modelos sem textura, com apenas uma única cor sólida aplicada, não aparentam ter profundidade, parecem figuras achatadas (2D).

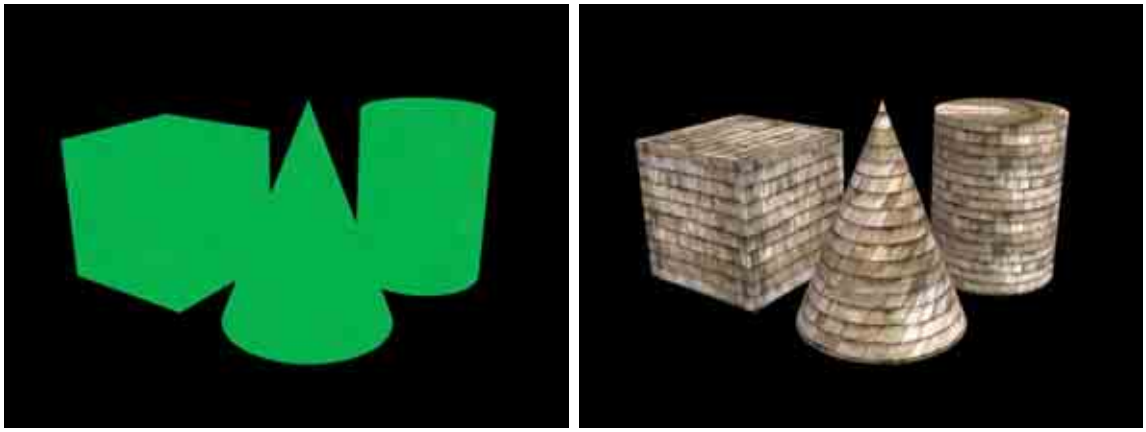


FIGURA 5.3.a. Na imagem da esquerda, ambiente sem iluminação com modelos apenas com aplicação da mesma cor. Na direita, os mesmos modelos, ainda no ambiente sem iluminação, mas com aplicação de uma mesma textura.

Textura, *texture*, mapas de textura e *texture maps* são palavras usadas para mapas de imagens que podem ser criados através de pintura digital em algum programa de edição de imagens, como o *Adobe Photoshop* ou obtidas a partir de uma câmera digital ou de um *scanner*. Esses tipos de imagens são chamados de mapas de imagem, *image maps* ou *bitmaps*. Existem, também, texturas criadas através de algoritmos que são conhecidas como mapas procedurais, mapas de procedimento 3D ou *procedural maps*, criados a partir de fórmulas matemáticas como o *Noise*, *Perlin Marble* e *Speckle*, e possuem parâmetros que podem ser alterados [10, 22].

Os mapas de textura são muito usados na construção de materiais porque enriquecem a qualidade visual ou conferem mais realismo aos materiais que vão compor superfícies de modelos 3D.

A texturização para ambientes a serem processados em tempo real, seja para jogos ou para *web*, tem a importante função de definir as características e os detalhes da superfície de todos os objetos da cena. Os processos e técnicas para aplicação de textura nesses ambientes não são os mesmos utilizados em cenas pré-processadas com renderização de alta resolução. O processamento em tempo real impõe limites. A textura de um único objeto em um *render* de alta definição pode ser uma combinação de diversos mapas, diversas imagens *bitmaps* em canais diferentes. Esse mesmo objeto teria um ganho maior no processamento em tempo real se o resultado visual da combinação de todos os seus mapas fosse aplicado a um único *bitmap* que seria, então, sua textura final.

Outra diferença é a necessidade de compensar na textura, através de pintura ou foto, a falta de detalhes na modelagem de objetos *low-poly* (baixo nível de detalhamento). Como a renderização em tempo real requer uma quantidade limitada de geometria, a textura é usada para compensar os detalhes que não podem ser representados por polígonos. Objetos que compõem cenas pré-processadas, geralmente, são *high-poly*, portanto já possuem alto nível de detalhes em sua malha geométrica.

Para ser melhor compreendido o processo de texturização, é fundamental distinguir o que é *shader* e o que é textura. *Shader*, às vezes conhecido como o próprio material, define as propriedades da superfície de um objeto, seu comportamento diante de iluminação, sua rugosidade ou lisura, se é brilhante ou fosca. Textura é a imagem que define a aparência visual do objeto, em geral, um *bitmap*. Por muito tempo, usar *shaders*

em ambientes e personagens virtuais para *games* era incomum. Agora já existe essa possibilidade que favorece a criação de materiais complexos para definir a superfície de objetos. Ainda que muitos jogos utilizem *bump*, *specular*, *displacement*, *pixel* e *vertex shaders*, o uso dessas técnicas e de outras que precisem ser processadas no momento da visualização ainda deve ser limitado em ambientes *real-time*.

A texturização através de um *bitmap* é a oportunidade de gerar soluções criativas para compensar as limitações. Definidos o conceito e a direção de arte do *game*, após um levantamento de referências imagéticas, os *level designers* através da textura reproduzem e aumentam o nível de detalhes e realismo em ambientes virtuais [24].

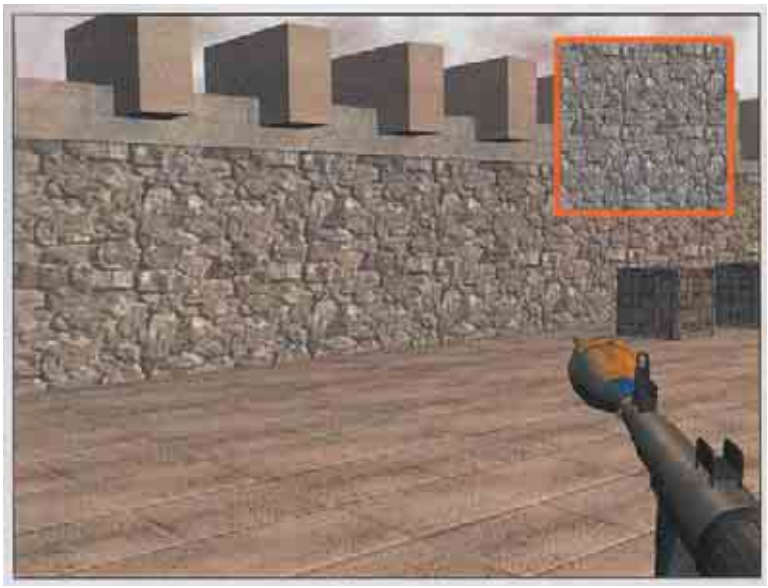


FIGURA 5.3.b. *Bitmap* repetido para a texturização do muro.
Imagem do livro *The Dark Side of Game Texturing.*



FIGURA 5.3.c. Exemplos de *bitmaps* usados na texturização de ambientes de *games*, aplicados com repetição (*tileable*).
Imagem do livro *The Dark Side of Game Texturing* (Cortesia de *Ben Ridgway*).

5.4. Mapas de Imagem – *Bitmaps*

Os monitores de vídeo dos primeiros computadores exibiam apenas imagens monocromáticas. Com o passar do tempo, a demanda por uma visualização em cores e por melhores resoluções de imagens levou ao desenvolvimento de monitores e placas gráficas mais potentes, com capacidade de reproduzirem *pixels* coloridos que através da combinação de três cores primárias – vermelho, verde e azul – pudessem formar uma imagem com uma rica gama de tons [25].

Mapa de imagem ou *raster*, mais conhecido como *Bitmap* (mapa de *bits*), é um padrão para representação de imagens em dispositivos de saída como um monitor de computador e, também, para meios não visuais, como na memória RAM de um computador ou em disco magnético, sob a forma de arquivo.

Um *bitmap* é composto por uma grade, geralmente, retangular de pequenos pontos de cor, conhecidos como *pixel* (*picture element*). O mapa de *bits* é uma imagem digital armazenada como dados, um conjunto finito de valores numéricos dispostos como uma matriz de *m* linhas e *n* colunas. *Pixel* é a menor unidade que pode ser endereçada para exibição de uma imagem, por isso quanto maior for a resolução, ou seja, quanto mais linhas e colunas de *pixels*, maior a quantidade de informação, portanto melhor será a definição da imagem e maior o peso do arquivo.

Existem diferentes formatos de arquivos para mapas de imagem, como:

- TARGA (*.tga) *Truevision Graphics Adapter*
- TIFF (*.tif) *Tagged Image File Format*
- BMP (*.bmp) *Bitmap*
- PICT (*.pct ou *.pic) *Windows Picture File*
- GIF (*.gif) *CompuServe Graphics Interchange Format*
- JPEG (*.jpg) *Joint Photographic Experts Group*
- PNG (*.png) *Portable Network Graphics*

Os formatos de arquivo GIF, JPEG e PNG foram desenvolvidos a partir da necessidade de criar arquivos de imagem que ocupassem menos espaço em disco. Esses formatos permitem controle de compressão. Por gerarem arquivos leves, se tornaram os formatos preferidos para publicação de imagens na *web*. Os plugins VRML também foram projetados para aceitar esses formatos em nós (*nodes*) *ImageTexture*.

Como o computador é baseado no sistema binário, para melhorar a performance e o processamento em tempo real dos arquivos, a cor em *bits* e as dimensões da imagem em *pixels* devem ser números que sejam potências de dois.

As resoluções (dimensões em *pixels*) mais utilizadas em *games* para mapas de texturas são [24]:

32 x 32	64 x 64	128 x 128	256 x 256	512 x 512
64 x 32	128 x 64	256 x 128	512 x 256	1024 x 1024

A cor, a tonalidade, o brilho e outras características de cada parte da imagem são representadas pelos valores numéricos de cada *pixel*. Cada pixel tem uma cor atribuída em *bits*. Na tela de um computador, um simples *bit* tem dois estados diferentes; ligado e desligado [25]. Portanto, o termo *bitmap*, usado para designar mapas de imagem, seria mais apropriado se fosse aplicado apenas para mapas de imagem em preto e branco. O termo exato para mapas de imagem coloridos é *pixmap* [10].

O *color deph* (profundidade de cor), também denominado definição de cor e medido em *bits*, deve seguir a mesma regra da resolução da imagem, preferencialmente, números que sejam potências de dois. É importante escolher um tamanho e uma definição de cor que ofereçam a melhor qualidade possível dentro das limitações impostas pela visualização em *real-time*; manter os arquivos com o menor tamanho possível para garantir um bom desempenho em processamento.

A seguir algumas definições de cores, aquelas em negrito são as mais adotadas em games para mapas de texturas, já as definições de 3, 5, 6, e 7-*bits* não trabalham bem na memória [24].

1 bit = 2 cores

2 bits = 4 cores

3 bits = 8 cores

4 bits = 16 cores

5 bits = 32 cores

6 bits = 64 cores

7 bits = 128 cores

8 bits = 256 cores

16 bits = 65,536 cores

24 bits = 16,777,215 cores

32 bits = 16,777,215 cores +
8-bits do canal alpha

6. Iluminação em Ambientes 3D

A iluminação é o que torna visíveis materiais, texturas, cores, enfim um ambiente seja ele real ou virtual. A aparência de qualquer material depende, em grande parte, da iluminação.

As fontes de luz são adicionadas à cena com o intuito de criar imagens mais realistas de modelos e ambientes virtuais tridimensionais. As luzes, nos programas de modelagem, são colocadas na cena, movimentadas, rotacionadas e também animadas, como os objetos e possuem parâmetros ajustáveis. Os tipos mais comuns e mais usados, entre os *softwares* mais populares, são: *directional light*, *spotlight* e *point* ou *omni light*.

A *directional light* ou luz direta produz um feixe de raios luminoso paralelos. Por isso, costuma ser usada para simular a luz do Sol ou da Lua, que por estarem tão distantes da Terra, projetam sombras visualmente paralelas, para os objetos de uma determinada região.

A *spotlight* ou refletor é muito usada em ambientes 3D, principalmente para os ambientes internos. Produz um feixe de luz em forma de cone e possibilita maior controle dos focos de iluminação.

A *omni light* é um ponto de luz que emite raios em todas as direções. A área de atuação de seus raios pode ser entendida como uma esfera luminosa. São muito utilizadas para simular lâmpadas domésticas.

Os objetos podem apresentar cores, associando-as aos vértices de sua geometria ou por aplicação de texturas. Dessa maneira, a iluminação afeta as cores, seja dos vértices ou da textura, criando um realismo maior.

6.1. *Luzes e Superfícies*

O relacionamento entre as superfícies que nós vemos e a luz que nos permite vê-las é fortemente integrado. Há tanta integração que os controles para *raytracing,* em *softwares* 3D, são encontrados , geralmente, no mesmo lugar que os outros parâmetros para a criação de materiais de superfície [10]. Nesses *softwares*, existe também a possibilidade de visualizar o material com iluminação e fazer o ajuste dessa no próprio painel de edição de materiais.

No editor de materiais do programa 3ds Max, por exemplo, o material pode sofrer influência de três tipos de iluminação: superior, *backlighting* e ambiente. A iluminação superior, conhecida como iluminação principal (*key light*), geralmente é a luz mais intensa e a que fornece a maior parte da iluminação da superfície. No editor de materiais, assim como na cena iluminada, a *key light* é a que mais interfere na aparência da superfície do material. A iluminação traseira ou *backlighting*, posicionada atrás de um objeto, tem a função, assim como na fotografia, de ressaltar o objeto do plano de fundo. No editor de materiais ela também é usada e ajuda na visualização do material. A iluminação ambiente não é uma fonte de luz direcional, como o próprio nome diz, ela cria uma atmosfera na cor selecionada que envolve todo o ambiente, como se a luz estivesse sendo refletida por todos os objetos de uma cena [22].

No mundo real, ao contrário do mundo virtual que fica limitado pela capacidade de *hardware* e de *software*, uma quantidade infinita de eventos gerados por iluminação ocorre a cada instante. O Sol, a cada segundo, emite 3.8×10^{44} fótons (partículas luminosas de energia). Uma pequena porção incide sobre metade da Terra, gerando a luz do dia. A luz é o que nós chamamos a parcela visível do espectro eletromagnético [10].

A radiação é a energia que se propaga de um ponto a outro no espaço ou num meio material. Os processos físicos de emissão e propagação de energia podem ser descritos como fenômenos ondulatórios ou por meio de partículas elementares conhecidas como fótons.

Quando um fóton atinge um objeto, ele se propaga e muda de direção, mas deixa alguma de sua energia para trás. Continua irradiando-se para outros objetos até que não tenha mais energia.

A luz que vem do Sol é mais ou menos branca, o que significa que ela carrega todas as cores. Quando ela incide em um objeto, a estrutura atômica desse objeto permite a absorção de parte desta energia. Por exemplo, quando a luz do Sol atinge um objeto laranja, este objeto tem uma aparência alaranjada porque todas as outras cores foram absorvidas exceto o laranja que é refletido e percebido pelo olho humano. Ocasionalmente, é possível observar que os objetos que estão próximos a esse, também pareçam um pouco alaranjados, porque assim como as ondas atingem a vista humana, elas também atingem objetos vizinhos. A mudança na direção de propagação de uma onda é chamado de reflexão.

6.2. *Raytracing* e *Radiosity*

Quando os raios de luz são refletidos por uma superfície em direções aleatórias, é chamado de reflexão difusa. Quando os raios de luz são refletidos por uma superfície em um ângulo igual ao ângulo de incidência, trata-se de uma reflexão especular [10].

Os dois modelos, complementares ou não, que conseguem resultados próximos à iluminação no mundo real, através de soluções de Iluminação Global, são: *radiosity* e *raytracing*. *Radiosity* é um método que calcula o componente difuso de uma superfície, independente da posição do observador (ponto de vista da câmera). *Raytracing* é um método que calcula o componente especular de uma superfície e é dependente do ponto de observação.

Os processos *raytracing* e *radiosity* são sistemas baseados em física para cálculo de iluminação em um ambiente virtual e da influência dessa sobre a superfície dos objetos que compõem o cenário, simulando a iluminação no universo real. O *raytracing* e o *radiosity* são também sistemas de renderização porque calculam a iluminação e geram uma imagem. Os dois processos redefinem a aparência da superfície dos objetos a partir da iluminação e dos reflexos dessa sobre o ambiente, determinando novas cores e tons para os pixels da imagem final renderizada.

O *raytracing* realiza um cálculo das trajetórias dos raios emitidos pelas fontes de luz do ambiente virtual até o ponto de vista da câmera e possibilita, dessa forma, a visualização do efeito da luz em materiais transparentes e reflexivos [21].

O *radiosity* calcula a trajetória dos raios de luz em todas as direções possíveis do ambiente virtual, diferente do método de *raytracing* que calcula apenas a direção dos raios entre a câmera e a fonte de luz. Dessa forma, o *radiosity* calcula a transferência de quantidade de luz entre as superfícies de todos os objetos envolvidos na renderização até que a energia de luz seja totalmente absorvida pelas superfícies ou dissipada pelo espaço.

Diferentes dos *shaders* que são algoritmos que determinam a forma como uma superfície reage à luz, o *raytrace* e o *radiosity* determinam os efeitos da reflexão da luz entre as superfícies, seguem o caminho da luz até que essa perca sua força, colhendo os resultados e representando-os em valores de cor nos pixels da imagem renderizada.

> O processo de renderização consiste em um conjunto bem definido de regras matemáticas de projeção e iluminação que tem por objetivo gerar uma imagem bidimensional através do cálculo de projeção de uma câmera virtual que visualiza objetos tridimensionais com uma determinada textura, a qual pode sofrer modificações na coloração e na luminância de acordo com fontes de luz definidas no ambiente. A imagem gerada depende de como o sistema calcula a intensidade da refletância da luz provinda das fontes de luz e dos outros objetos. Tal cálculo é denominado modelo de iluminação [2].

Em uma cena com vários elementos, a função primordial de um *render* é determinar que partes da geometria devem ser representadas nos *pixels* da imagem. Dependendo do ângulo de visualização, alguns objetos podem esconder parcialmente ou totalmente outros. Como esta técnica de determinação de superfície visível (ou de remoção de superfície escondida) é fundamental e imprescindível no processo de renderização, porém gera cálculos bastante complexos, a adição de soluções de *radiosity* e *raytracing* podem aumentar muito esta complexidade e o tempo para o processamento da cena.

Se todas as cenas fossem renderizadas com radiosidade ou *raytracing*, os resultados seriam mais interessantes e convincentes. No entanto, mesmo que o ambiente virtual não esteja sendo preparado para processamento em tempo real, que seja uma renderização de uma seqüência de cenas para gerar um filme, o tempo de processamento será penoso. O *RenderMan*, *software* usado pela *Pixar* que renderizou filmes como o *Toy Story* (1995), é um render *scanline* que não possui *raytracing*. Entretanto, acelera a etapa de renderização do filme e consegue resultados semelhantes ao *raytracing* usando mapas de imagem de reflexão [10].

7. *Mapping* e *Unwrapping*

Os programas de modelagem, geralmente, oferecem duas maneiras de aplicar um mapa de textura em um modelo geométrico: *mapping* e *unwrapping*. A denominação desses dois métodos pode variar de *software* para *software*, mais o fundamento é o mesmo [10].

Em um espaço tridimensional, onde os objetos são construídos pela definição de pontos em coordenadas X, Y e Z, os mapas de textura aplicados aos modelos, para envolvê-los com exatidão, precisam acompanhar essas coordenadas. Arbitrariamente, para não confundir os processos de modelagem e texturização, convencionou-se para coordenadas de mapeamento de textura o uso das letras U, V e W para definir as dimensões X, Y e Z, respectivamente.

O *mapping* é o ajuste da textura, em relação às coordenadas do objeto, a partir de modelos de mapeamento pré-definidos e ajustáveis por parâmetros. Os *softwares* mais populares de modelagem possuem pelo menos quatro tipos de *mapping* comuns: *box*, *spherical*, *cylindrical* e *planar*. O *mapping* define a forma de como um mapa, geralmente uma matriz bidimensional, pode embrulhar um objeto. Assim como, permite manipulações na textura já aplicada; posição, tamanho, deslocamento, rotação e quantidades de repetições desse mapa.

Para ajustar uma imagem em objetos com modelagem mais complexa e menos uniforme, o *mapping* não é a melhor alternativa. Para esses casos, existe o *unwrapping* que oferece maior controle na texturização através de um processo inverso ao *mapping*. O *unwrapping* planifica o modelo geométrico, rotacionando suas faces para que fiquem todas em um mesmo plano e as organiza em *clusters*. Esses agrupamentos de faces são definidos por algoritmos, mas podem ser reorganizados manualmente com auxílio das ferramentas de edição do *unwrapping*.

Alguns algoritmos de *unwrapping* permitem definir um ângulo para dividir a malha em *clusters*. O algoritmo começa com uma única face e continua acrescentando faces até atingir uma, que é girada em um valor maior que o do ângulo definido. A partir dessa face, ele começa a organizar novo agrupamento. Se for definido um ângulo muito pequeno, ocorrerão muitas faces individuais. Por outro lado, se for um ângulo próximo a 90 graus, haverá muitas faces distorcidas nas bordas dos *clusters*. Existem, também,

algoritmos que dividem a malha em *clusters* de acordo com as normais das faces, a partir de suas direções [22].

A geometria planificada e organizada em *clusters* pode ser copiada (*print screen*, geralmente) para um programa de edição de imagens, como o *Photoshop*. Esse mapa da geometria serve de gabarito para uma pintura digital ou para aplicação de imagens. Concluída a texturização da malha geométrica exposta, a imagem é salva e pode ser aplicada sobre o modelo 3D.



FIGURA 7.a. Exemplos de *mapping.*
Imagem do livro *Texturing: Concepts and Techniques.*

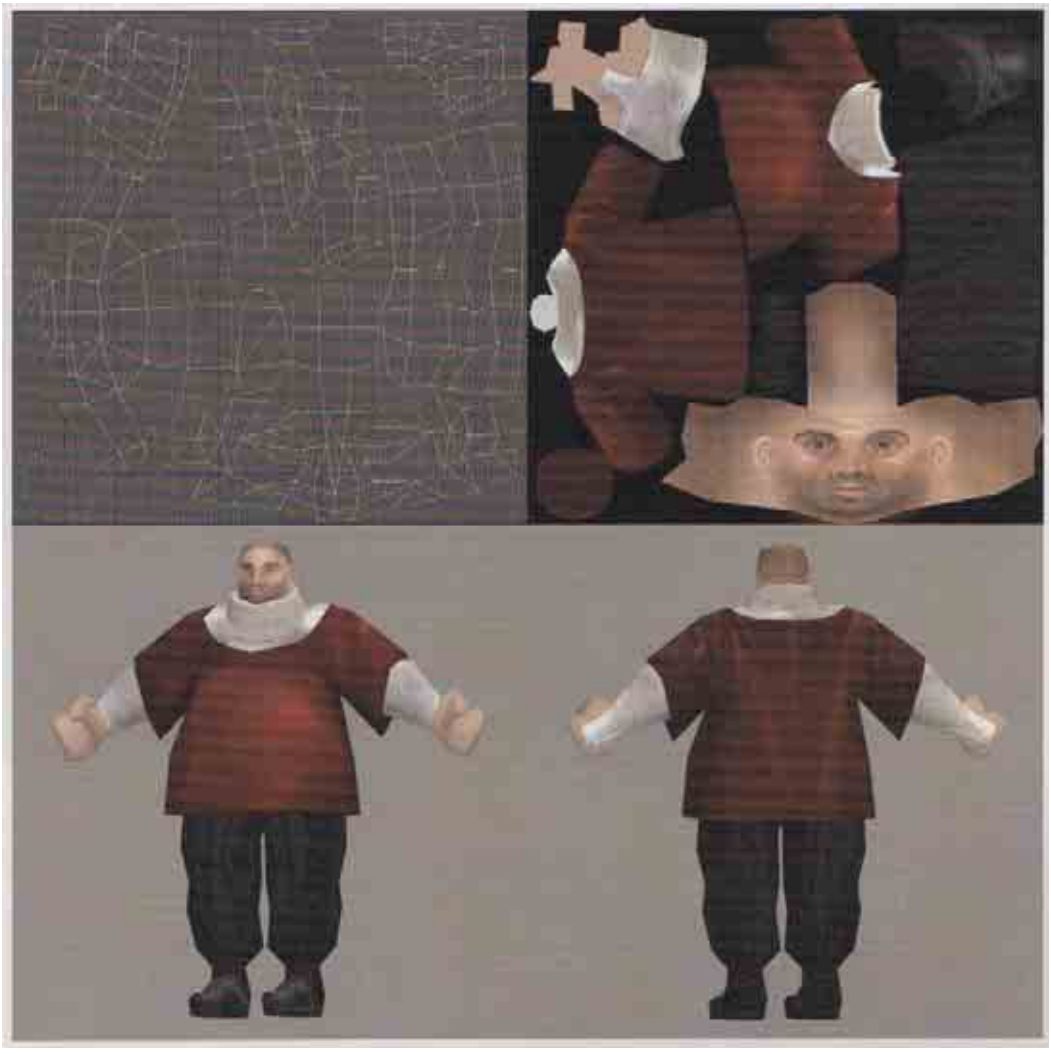


FIGURA 7.b. Exemplo de *unwrapping.*
Imagem do livro *Texturing: Concepts and Techniques.*

8. Técnicas e Processos para Otimização de Ambientes 3D Interativos

8.1. *Vertex Color*

Muitos aplicativos 3D e *engines* (motores) de games suportam o uso de *vertex color* (cores nos vértices). Nesses programas, quando se inicia a construção de um modelo, cada vértice tem uma cor associada e, geralmente, a cor do vértice é cinza (RGB - *red*: 128, *green*: 128 e *blue*: 128), no *3ds max* é branca (RGB: 255, 255, 255) [24].

Através do uso dessa técnica, sem a necessidade da aplicação de materiais, *shaders* e mapas de textura, um objeto pode ser colorido, ter transparência e simular áreas de brilho, apenas usando informações de cores diretas nos vértices de sua malha. As cores são formadas por uma combinação de quantidades de vermelho, verde e azul; a escala RGB (*red*, *green* e *blue*). Os vértices também podem carregar informações de opacidade através do *Alpha* (canal de opacidade).

O *vertex color* também pode ser associado à aplicações de materiais. Um objeto pode ter um um *bitmap* aplicado como textura e através de alterações nas cores dos vértices simular modificações nesse *bitmap*, proporcionando uma nova aparência para o material da superfície de um objeto. As informações nos vértices funcionam como uma camada sobre o *bitmap*.

É amplo o universo quantitativo de aplicações dessa técnica, da diversidade de efeitos que ela pode gerar e de soluções para melhoria no processamento em tempo-real.

Algumas aplicações do *vertex color* incluem:

- Pode ser usado para disfarçar o uso da repetição de um *bitmap* em uma superfície. Mudando as cores de alguns vértices, pode-se obter resultados que simulem a aparência de sujeira, brilho, profundidade, sem ter que usar novos *bitmaps*.
- Pode ser usado de outra forma, mas também para disfarçar a reutilização de um *bitmap*, no caso de um *bitmap* que será aplicado em diversos objetos. Por exemplo, vários lápis de cores diferentes compartilhando o mesmo *bitmap* como

mapa de textura, porém em cada malha geométrica existe alterações na coloração de determinados vértices.

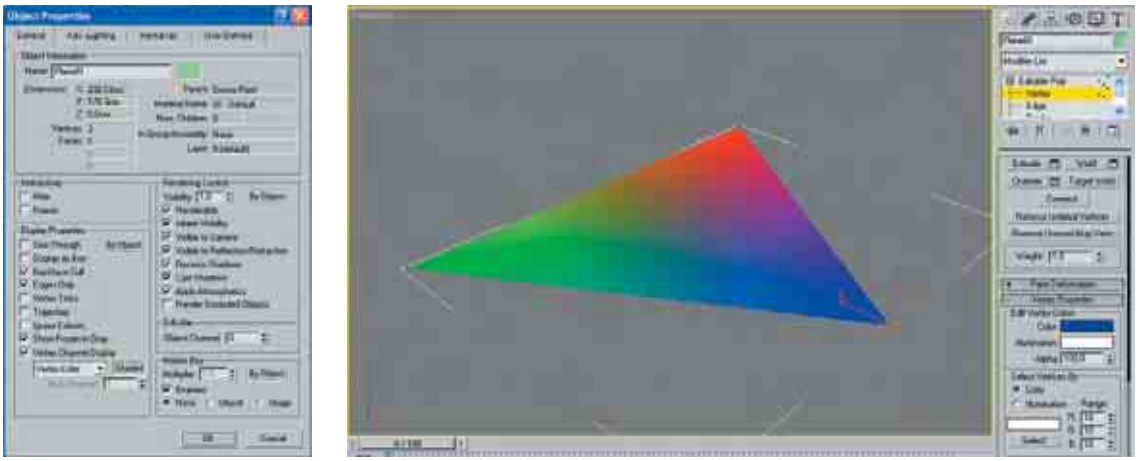


FIGURA 8.1.a. No 3ds Max, para visualizar as cores aplicadas aos vértices, deve-se acessar as propriedades do objeto pelo painel *Object Properties*, selecionar *Vertex Color* em *Display Properties* e clicar em *Shaded*. Para alterar as cores, basta selecionar os vértices da malha e alterá-los – *color*, *illumination* e *alpha* – pelo painel *Vertex Properties*.

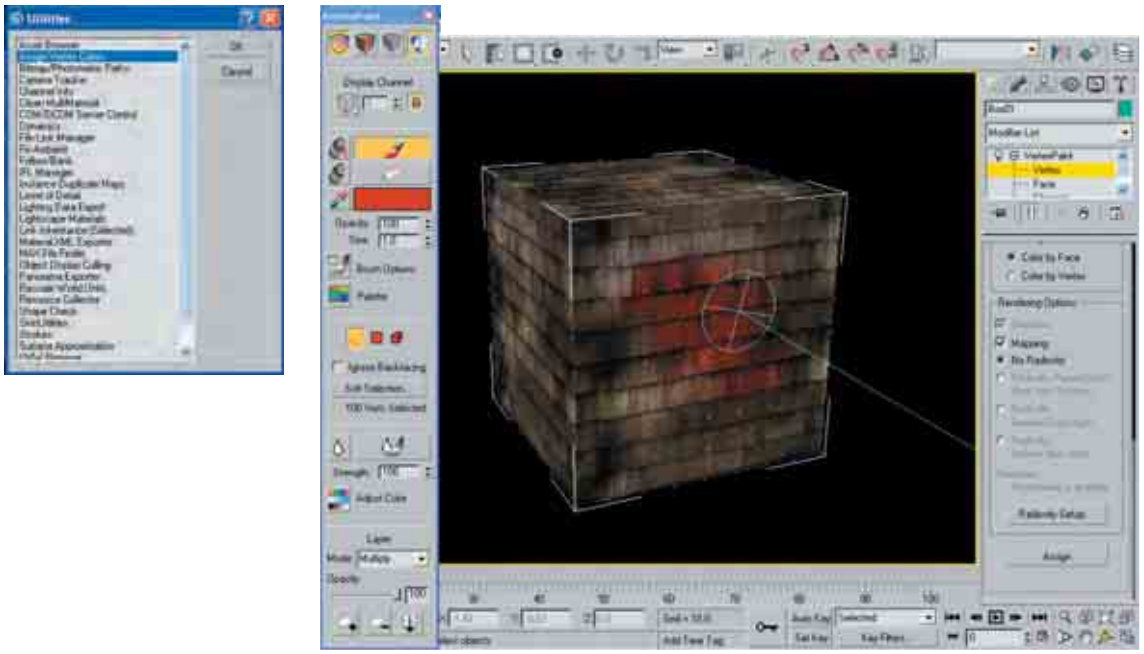


FIGURA 8.1.b. Também é possível aplicar ou alterar propriedades dos vértices, como cor e iluminação, em um objeto que já possui textura aplicada. No *3ds max*, em *Utilities* (*More...*), escolher *Assign Vertex Color*, e nesse painel, clicar em *Assign to Selected* e depois em *Edit*. O *VertexPaint* entra como um modificador sobre o polígono editável, selecionar os vértices que se deseja alterar, a cor e o filtro que define a forma como o layer da pintura dos vértices será aplicado sobre a textura.

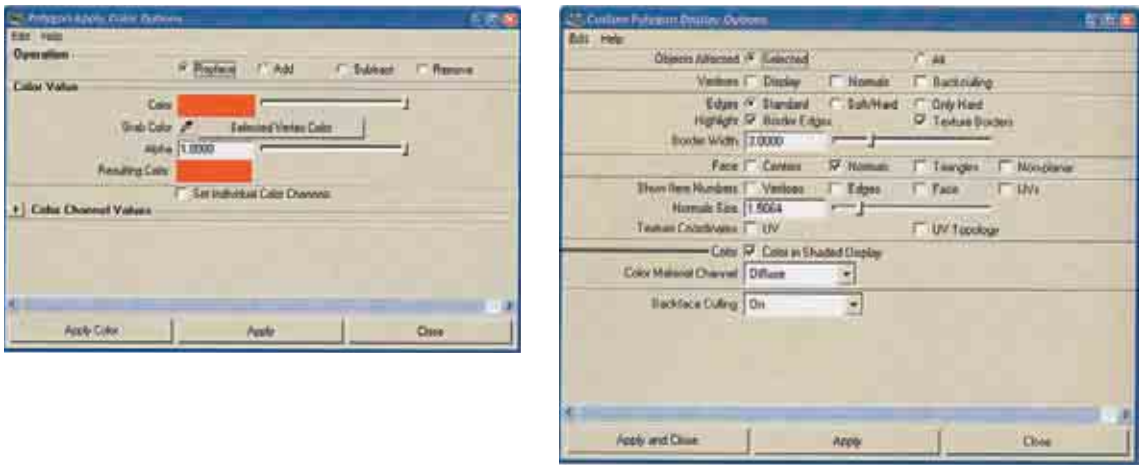


FIGURA 8.1.c. As imagens acima mostram os painéis para aplicação de cores nos vértices e para visualização dos resultados no *Maya*. Figuras do livro *Creating the Art of the Game*.

8.1.1. *VERTEX LIGHTING*

É uma técnica que reúne todas as informações e efeitos de iluminação da superfície de um modelo, gerados pelas luzes do ambiente, e aplica esse resultado direto na geometria desse objeto, mas precisamente para os vértices da malha do modelo, usando a técnica de *Vertex Color*, informações de cor: canais RGB e, eventualmente, o canal *Alpha* [24].

O *Vertex Lighting* não é uma técnica de iluminação, não significa colocar luzes em uma cena, mas sim, usar as luzes que incidem sobre um objeto para colorir os vértices de sua malha. No *3ds max*, esse processo é gerado pela ferramenta "*Assign Vertex Colors"* que se encontra no "*Utilities"* (figura 8.1.b). No *Maya*, o *Vertex Lighting* é chamado de *"Prelighting"* porque esse método também é usado para visualizar iluminação em modelos *high-poly* sem precisar renderizar a cena e pode ser acessado escolhendo-se *Edit Polygons > Color > Prelighting*.

Depois de aplicar a técnica, as luzes podem ser retiradas da cena, dispensando, assim, o processamento dessas em tempo real. O efeito resultante será sempre ligeiramente diferente do que é visto na janela de visualização ou no render, antes de executar esse processo.

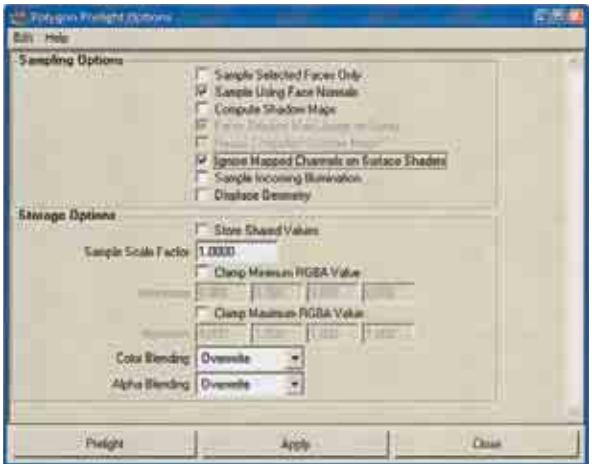


FIGURA 8.1.d. *Prelighting* no *Maya*. Figura do livro *Creating the Art of the Game*.

Assim como o *Vertex Lighting*, existem outras técnicas geradas a partir do *Vertex Color*; o *Vertex Alpha* e a adição de radiosidade (*Radiosity*) para os vértices da malha. A desvantagem dessas técnicas está na relação de dependência entre o número de vértices da malha e a qualidade do resultado gerado. Quanto menor for o número de vértices do modelo, menor será a definição das áreas iluminadas ou coloridas (figura 8.1.e.).

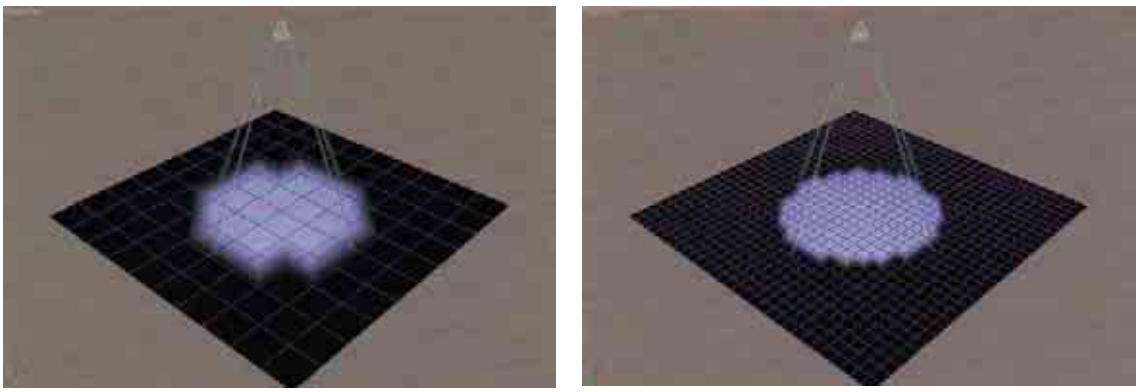


FIGURA 8.1.e. *Vertex Lighting* aplicado a partir de um *spotlight* sobre dois planos com diferentes níveis de detalhamento na malha geométrica. Figura do livro *Creating the Art of the Game*.

8.2. *Vertex Shader* e *Pixel* (ou *Fragment*) *Shader*

Muito além de ser capaz de processar ambientes virtuais constituídos por milhões de triângulos, as atuais placas gráficas aceleradoras permitem a realização de diversos efeitos gráficos complexos em tempo real através de programação das funcionalidades existentes na arquitetura do *pipeline* gráfico da placa [26]. Por *pipeline gráfico* entende-se a descrição do processo que uma estrutura de representação de um modelo gráfico sofre, até a sua apresentação num dispositivo.

> Determinados processos do *pipeline* gráfico são mais suscetíveis ao aspecto de programação do que outros. A unidade ou processo do *pipeline* que trata os vértices dos objetos gráficos, por exemplo, é flexível a operações de programação. A unidade de programação de vértices (*vertex shaders*) poderá realizar várias operações e cálculos com os atributos dos vértices (posição, cor e normal, por exemplo) para produzir efeitos, dentre outros, de iluminação, *skinning*, *morphing* e *bump mapping*.

> Outro processo importante do *pipeline* que pode ser programado é aquele que gera a cor final de um *pixel* na tela. A unidade de programação de *pixels* ou fragmentos (*pixel* ou *fragment shader*) opera com os atributos de cada *pixel* para a composição final da cor de cada fragmento. Esta unidade de programação produz efeitos muito interessantes, já que se pode fazer acessos a várias texturas próprias do *hardware* para combinar o resultado destes acessos aos próprios atributos dos *pixels*. Estes acessos a texturas vêm sendo explorados por pesquisas utilizando texturas 3D na visualizaçào de dados volumétricos [26].

A finalidade original das placas de vídeo [27] era determinar apenas a cor final dos *pixels* de uma geometria submetida ao fluxo de visualização. Com a popularização, em 2001 [28], das placas de vídeo programáveis, a malha geométrica de um modelo pode ser deformada durante o processo de visualização a partir de algoritmos programados diretamente para os vértices, que alteram a posição e a aparência desses, ou adquirir uma melhor definição de modelagem através de processamento gráfico de fragmentos em tempo real, utilizando um dispositivo de entrada 2D. São modificações que afetam a aparência física do modelo, geradas por *hardware* para renderização em tempo real e visualização interativa, que processam resultados de forma semelhante aos, até então, conseguidos apenas por *softwares* através de especificações para renderização cinemática, como o *RenderMan Interface Specification* (RISpec) [29]; API desenvolvida pela *Pixar* para descrever cenas tridimensionais e convertê-las em imagens digitais de alta qualidade. O modificador de vértices é, portanto, um modelador, informa ao processador de vértices mudanças em relação à posição e normal à superfície dos vértices que compõem a geometria do modelo original. O modificador de fragmentos, por sua vez, é um simulador, porque altera os atributos de cada *pixel*, tais como profundidade e vetor normal, após o estágio de rasterização [30, 31].

> A abordagem de deformação em placas de vídeo começa a ser largamente utilizada em técnicas de geração de terrenos, em mapeamento de deslocamento e *skinning* de geometria, uma vez que isso minimiza o tráfego de dados entre a CPU e a unidade de processamento gráfico (GPU, *Graphics Processing Unit*) necessário para visualizar os modelos. Por exemplo, para geração de terrenos, a geometria é armazenada com antecedência na placa de vídeo

> como uma grade plana e estática. Ao mesmo tempo é armazenada a textura contendo os valores de altura do terreno. Para cada quadro de exibição, o usuário precisa apenas enviar a posição atual do observador à placa. O modificador de vértices responsabiliza-se pela deformação da grade de acordo com as informações de altura do terreno correspondentes com a posição atual do observador. De forma parecida, na técnica de *skinning* de vértices, a malha triangular que compõe o modelo é armazenada com antecedência na placa gráfica, considerando que cada vértice já contém, em atributos adicionais, os pesos de influência de cada matriz que compõe a hierarquia de *bones*. Para cada quadro de exibição, basta enviar à placa de vídeo essa hierarquia de *bones*, composta de matrizes, para posicionar cada vértice do modelo no modificador de vértices [27].
>
> Para programar estas unidades do pipeline gráfico do *hardware*, uma API (*Application Program Interface*) foi desenvolvida pela empresa *NVIDIA Coorporation*, denominada *C for Graphics* (Cg). A linguagem *Cg* permite um alto nível de abstração do *hardware* gráfico semelhante à linguagem C, tanto em sua sintaxe quanto no próprio nível de abstração, podendo ser compilada para gerar código em aplicações que utilizam *DirectX* ou *OpenGL* [26].

A técnica de *vertex shader* assim como as de *vertex color* não foram adotadas na construção do protótipo de ambiente virtual tridimensional realizado para demonstrar a otimização desses ambientes nesta dissertação. Essa opção se deve ao fato do resultado da aplicação dessas técnicas, em termos de qualidade visual, dependerem da resolução da malha geométrica dos objetos que compõem o ambiente. A exigência de uma maior quantidade de vértices contraria a finalidade desta pesquisa porque aumenta o esforço computacional no processamento em tempo real.

8.3. *RENDER TO TEXTURE*

Render to Texture, *Baking de Texturas*, Preparação de Texturas são nomes adotados para a técnica muito utilizada em diversos setores da indústria 3D que acelera o processo de renderização porque combina atributos de uma superfície, como textura, iluminação e sombras e salva-os como um mapa de imagem para o objeto.

Neste processo existe a possibilidade de adotar apenas um ou vários aspectos da textura para preparar o mapa. Pode-se salvar um *bitmap* apenas com as informações de iluminação de um objeto (*Lighting Map*). Também pode ser usado para preparar vários mapas em um único mapa de imagem e para transformar mapas procedurais em *bitmap*, como um reflexo gerado pelo mapa *Raytracer*.

Outra aplicação comum é preparar um mapa de imagem com o resultado da adição de iluminação avançada em um ambiente, efeitos de iluminação criados por *raytracing* ou *radiosity*. As soluções de iluminação avançada podem aumentar consideravelmente o tempo de processamento de renderização, por isso a transformação delas em textura é um processo fundamental na preparação de ambientes virtuais tridimensionais para serem visualizados em tempo real.

É necessário expor os UVWs de um objeto para que o *Render to Texture* funcione. No *3ds max* o mapeamento dos UVWs pode ser feito antes de aplicar o *Render to Texture* através do recurso *Unwrap UVW* e esta é a melhor opção, mas o *Unwrap UVW* também pode ser criado automaticamente pela própria caixa do *Render to Texture*. A desvantagem de usar a técnica de exposição automática é que dificulta uma posterior edição do mapa, caso se faça necessário, porque o *Unwrap UVW* do *Render to Texture* não cria uma exposição combinada em *clusters*, como a descrita no capítulo 7 [22].

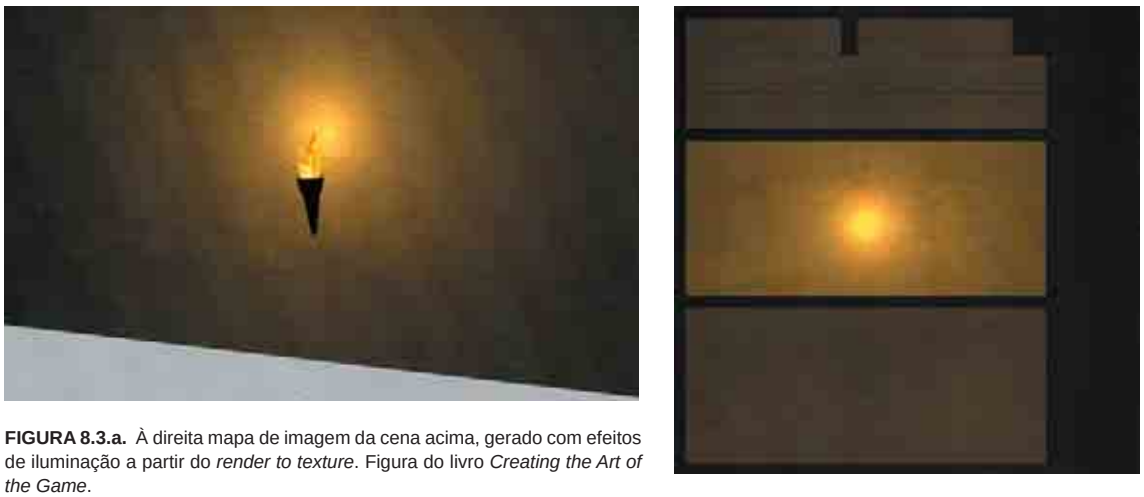


FIGURA 8.3.a. À direita mapa de imagem da cena acima, gerado com efeitos de iluminação a partir do *render to texture*. Figura do livro *Creating the Art of the Game*.

8.4. *Bump Map* e *Normal Map*

Normal map ou mapa de normais é considerado a solução mais inovadora para criação e otimização de ambientes virtuais tridimensionais. A técnica é muito similar à aplicação de *bump map*, mapa de saliência. Ambas utilizam o canal de *bump* de um material e, na presença de luz, simulam superfícies com relevos e detalhes inexistentes em uma malha geométrica de um modelo 3D [24]. O *normal map* é uma solução superior porque produz resultados mais verossímeis. Essa técnica tem sido usada cada vez mais na criação de cenários e personagens para os "*advanced games*", como: *Far Cry* (*Crytek*), *Deus Ex: Invisible War* (*Eidos Interactive*), *Thief: Deadly Shadows* (*Eidos Interactive*), *The Chronicles of Riddick: Escape from Butcher Bay* (*Vivendi Universal*), *Halo 2* (*Microsoft*), *Doom 3* (*id Software*), *Half-Life 2* (*Valve Software*), *Call of Duty 2* (*Activision*), *Tom Clancy's Splinter Cell: Chaos Theory* (*Ubisoft*) e na terceira versão do *Unreal engine* (*Epic Games*) [32].

O processo inicial dessa técnica consiste em extrair informações das propriedades da superfície – seu relevo e a forma como reage a incidência de iluminação – de um modelo *high-poly* (alto nível de detalhes). A partir das normais dessa geometria, transferir as informações de sua superfície para os *pixels* de um *bitmap*; o *normal map*. Esse mapa de normais é aplicado a um modelo *low-poly* (baixo número de polígonos), geralmente, um objeto originário das primeiras fases de modelagem de um objeto mais complexo ou uma versão minimizada deste último, pós aplicação de um algoritmo ou por um trabalho manual de redução de polígonos.

A técnica, com o auxílio de iluminação, produz uma impressão de *high-detail* (alto nível de detalhamento) em um modelo *low-poly*. Quando a luz atinge a superfície desse objeto, mapeado com o *normal map*, cria-se a ilusão de detalhamento que enriquece visualmente o objeto. O processo, portanto, contribui com a otimização do processamento *real-time* do ambiente virtual.

Conceitualmente o *normal map* é muito semelhante ao *bump map*, porém o primeiro utiliza os valores de RGB do *bitmap* para guardar informações de propriedades da superfície em suas três coordenadas X, Y e Z, produzindo resultados mais realísticos no processamento em tempo real do que o *bump map*. Um mapa de normais contém para cada *pixel* três canais de informação (as coordenadas do vetor normal em X,Y e

Z). Essa informação se codifica em um mapa de cores RGB, correspondendo cada coordenada X, Y e Z com as cores R (*red*), G (*green*) e B (*blue*), respectivamente [33]. O mapa de *bump* é calculado, geralmente, baseado em uma imagem de apenas um canal de cor e é interpretado como uma escala de cinzas, as variações dos cinzas do *bitmap* são utilizadas para simular informações de altura na superfície.

As técnicas e aplicativos que geram mapas de normais possibilitam a escolha do espaço referente ao sistema de coordenadas, onde as normais serão medidas e armazenadas, que pode ser: o espaço do mundo, o espaço do objeto e o espaço da tangente [33].

Não existe nenhuma dependência em relação ao tamanho do *bitmap* usado como mapa de *diffuse* e a dimensão do *bitmap* usado como mapa de normais. O mapa de *diffuse* pode ser maior, melhorando a definição de detalhes na pintura do objeto, ou ser menor, no caso de uma imagem que será repetida várias vezes. Nesse caso a dimensão do *bitmap* pode ser bem pequena e o mapa de normais não poderá seguir o seu formato, terá que ser maior.

O problema da aplicação dessa técnica na construção de um ambiente é a demanda de tempo, já que é necessária a geração de um mapa de normais para cada modelo. É um processo de produção árduo, mas a recompensa em redução no peso do arquivo e no processamento em tempo real lhe torna valorosa e, quase, fundamental para criação de mundos virtuais. Outro fator que merece destaque é que apesar do normal map, assim como o *bump map*, criar uma aparência de detalhamento acurado para um modelo 3D de baixa resolução, a geometria não é alterada. A silhueta do modelo será definida pelos vértices de sua geometria, o que revela a sua condição de modelo *low-poly*.

Bump mapping e técnicas afins têm sido aplicadas com sucesso em combinação com uma variedade de algoritmos de renderização e, graças aos avanços nos *hardwares* gráficos, seu uso está se tornando comum em aplicações interativas. Desde a concepção e introdução à utilização de mapas de *bump*, por *J. Blinn* [34], diversas técnicas vêm sendo desenvolvidas para simular detalhes em uma geometria a partir da aplicação de texturas. Muitas técnicas utilizando mapas de *bump* para renderização interativa foram propostas, como por exemplo; melhora na aparência do mapa de *bump* com aplicação de mapas de horizonte (*horizon maps*) [35]; uma técnica de *self-shadowing*

para mapas de *bump* que foi introduzida em [36]; uma descrição de como os mapas de horizonte podem ser acelerados usando o *hardware* [37]. Técnicas semelhantes ao *bump* que utilizam mapas de imagem para gerar o relevo, porém ao invés de simular, realmente alteram a geometria, como os mapas de deslocamento (*displacement maps*) [38, 39] e os mapas de relevo (*relief maps*) [40, 41]. Artigos recentes já tratam da técnica de *normal map* [42].

Os métodos de geração de mapas de normais mais utilizados atualmente são através de *software*. Podem ser gerados por meio dos programas 3D de edição, como por exemplo *3DS Max* ou *Maya*. Também existem aplicativos dedicados exclusivamente à criação de mapas de normais, como o *NormalMapper* da ATI [43] o *nVidia Melody* [44]. Desse modo são programadas as instruções necessárias para a geração dos mapas de normais e, somente, o processamento, ou seja, a decodificação das informações contidas nesse mapa, é feito em tempo real. Existem artigos que propõem também a geração do mapa de normal por *hardware*, como em [33], utilizando vértices e *pixels shaders*.

O uso da técnica de *normal map* ganhou popularidade entre os desenvolvedores de *games* devido à combinação de aumento da qualidade visual gráfica com a diminuição na exigência de processamento para renderização em tempo real. A redução na etapa de processamento pode ser entendida como um melhor desempenho para a visualização e navegação interativa e pode ser ainda maior quando aplicada juntamente com a técnica de *mipmapping*, que será descrita nas próximas páginas. A técnica de *normal map* também pode ser conjugada ao “LOD”, para troca de objetos *high-poly* por *low-poly*, de acordo com a distância do ponto de observação.

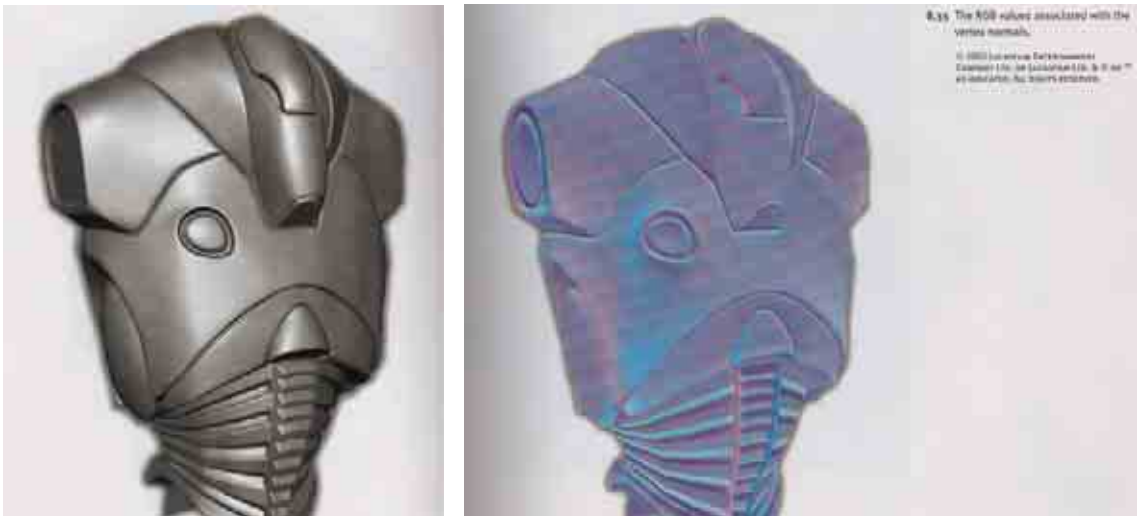
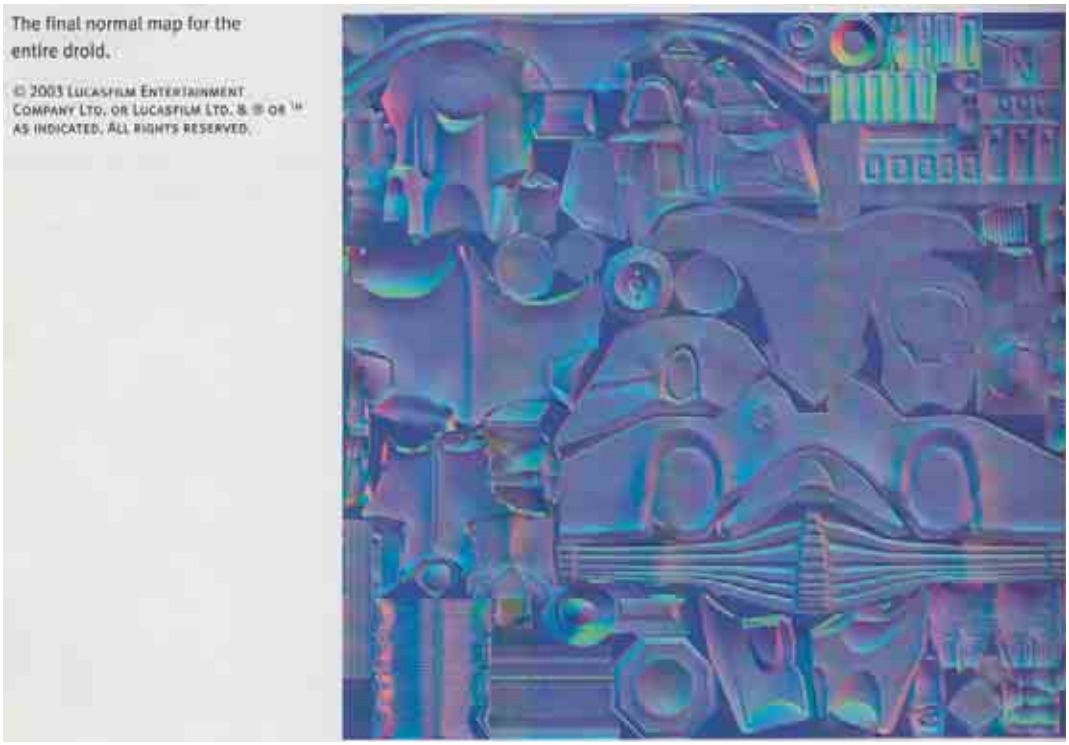


FIGURA 8.4.a. A imagem à esquerda é um modelo *high-poly*. À direita, visualização dos valores de RGB no mapa de normais aplicado ao modelo. Figura do livro *Creating the Art of the Game*.



The final normal map for the entire droid.

© 2003 Lucasfilm Entertainment Company Ltd. or Lucasfilm Ltd. & ® or ™ as indicated. All rights reserved.



8.35 The final in-game model showing far more detail than is actually there, thanks to a normal map.

© 2003 Lucasfilm Entertainment Company Ltd. or Lucasfilm Ltd. & ® or ™ as indicated. All rights reserved.

FIGURA 8.4.b. A primeira imagem é um mapa de normais que foi aplicado no modelo *low-poly* acima.
Figuras do livro *Creating the Art of the Game*.

8.5. Redução de Complexidade Geométrica e Visual

O processamento de um objeto com uma malha poligonal complexa e texturizada com *bitmap* de alta resolução é custoso para o sistema *real-time*. Esse esforço computacional de processamento só será válido se o objeto estiver próximo ao ponto de visão. Entretanto, quando o mesmo objeto se encontra distante da câmera, essa enorme quantidade de polígonos e os detalhes do *bitmap* serão transformados em uma pequena porção de *pixels* na imagem final da cena renderizada. Nesse caso, o tempo despendido de processamento desse objeto não será compensador.

A escolha do nível de detalhamento em geometria e do tamanho da textura deve sempre considerar a relação custo e benefício entre os fatores: qualidade visual e tempo de processamento.

Com a preocupação de manter sempre um nível satisfatório de qualidade visual durante a visualização em tempo real de ambientes tridimensionais, muitas vezes são adotados procedimentos como o LOD e o *mipmapping*, que reduzem a complexidade geométrica e visual de objetos, apenas quando forem dispensáveis.

8.5.1. LOD (*Level of Detail*)

A quantidade de detalhes na geometria de modelo 3D não precisa ser constante durante a visualização de um ambiente *real-time*. O LOD define o nível de detalhamento geométrico. Quando os objetos são afastados ou se movem, ficando mais distantes do observador, o nível de detalhe diminui.

No padrão VRML, o nó (*node*) de LOD especifica vários níveis de detalhe para a geometria de um determinado objeto e fornece sugestões permitindo que os *browsers* escolham automaticamente a versão apropriada do objeto baseado na distância do usuário. O campo *field* contém uma lista de nós que representam o mesmo objeto em níveis variados de detalhe. A lista deve começar pela representação do objeto com maior nível de detalhe e seguir em ordem decrescente de detalhamento. O campo *range* especifica as distâncias ideais entre as mudanças de níveis [45].

```
#VRML V2.0 utf8

LOD {
   range [ 10, 50 ]
   level [
      LOD {
         level [
            Shape { geometry Sphere { } }
            DEF LoRes Shape { geometry Box { } }
         ]
      }
      USE LoRes,
      Shape { } # Display nothing
   ]
}
```

Neste exemplo, o *browser* está livre para escolher uma versão mais detalhada (*Sphere*) ou menos detalhada (*Box*) do objeto quando o ponto de vista está próximo de 10 unidades (medida de coordenada do espaço de LOD) do objeto. O *browser* deve indicar a versão menos detalhada do objeto se o observador estiver entre 10 e 50 e não exibir nenhuma imagem do objeto caso o ponto de vista esteja mais distante do que 50.

8.5.2. *Mipmaps*

Quanto às texturas, a maioria dos motores de jogos criam automaticamente LODs diferentes para suas imagens, chamados de *mipmaps*, que são exibidos com diferentes resoluções dependendo também da proximidade do objeto ao observador [25].

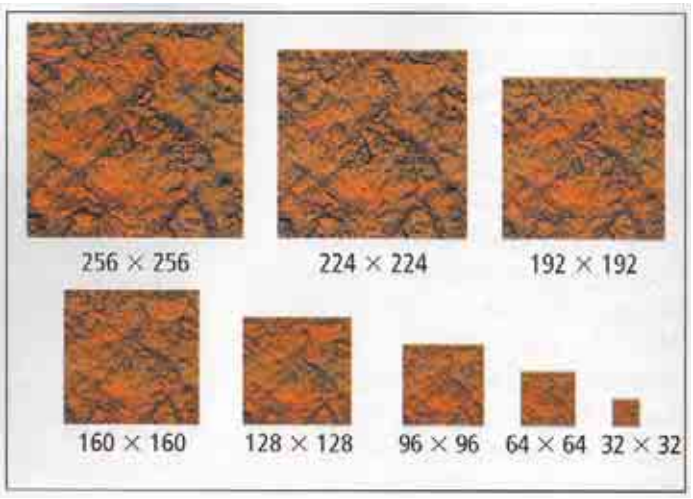


FIGURA 8.5.a. Imagens relativas aos tamanhos de *mipmaps* gerados automaticamente por um *game engine*. Figura do livro *The Dark Side of Game Texturing*.

Mipmaps são coleções de imagens para um mesmo objeto, *bitmaps* otimizados e pré-calculados, que ficam armazenados, geralmente, no mesmo diretório em que está a textura principal do modelo. Os *mipmaps* são usados para substituir o *bitmap* principal quando a distância do ponto de observação ao objeto não exigir muita definição, aumentando a performance e diminuindo o tempo de renderização da cena.

O VRML possui o mecanismo EXTERNPROTO para extensibilidade que possibilita a definição de novos nós. O site do Cortona VRML Client apresenta um exemplo de um EXTERNPROTO com um nó de *mipmap* [46].

```
#VRML V2.0 utf8

EXTERNPROTO       MipMap [
  exposedField MFNode          levels
]
[ “urn:ParaGraph:MipMap” “http://www.parallelgraphics.com/vrml/proto/Cortona/extensions.wrl#MipMap” ]

DEF VP Viewpoint {
  position 0 0 5
}

Shape {
  appearance Appearance {
            material Material {

            }

            texture MipMap {
                  levels [
                        ImageTexture { url “256.jpg” }
                        ImageTexture { url “128.jpg” }
                        ImageTexture { url “64.jpg” }
                        ImageTexture { url “32.jpg” }
                        ImageTexture { url “16.jpg” }
                  ]
            }
  }

  geometry Box {size 3 3 3
  }
}
```

O nó de *mipmap* especifica uma lista de nós de textura que contém versões em baixa e alta resolução da mesma textura a ser aplicada em uma mesma geometria. O campo *level* contém a lista dos nós de textura (*ImageTexture*, *PixelTexture* ou *MovieTexture*) que especificam níveis diferentes de *mipmap* para uma textura original. A textura original, com maior resolução, tem que aparecer na primeira posição na lista do campo *level*. Cada textura seguinte deve ter a metade do tamanho (para cada uma das dimensões) da textura precedente. Deve, portanto, manter a proporção da original, senão a textura seguinte é esticada e distorcida para ser aplicada da mesma forma que a original. Se alguns dos níveis faltarem, serão gerados automaticamente.

9. Aplicação das Técnicas de Otimização em um Protótipo de Ambiente 3D para Web

Como a proposta desta dissertação é demonstrar a importância dos *bitmaps* para a otimização de ambientes virtuais, alguns objetos do cenário também foram criados a partir de fotos digitais (*bitmaps*) de modelos físicos reais. Esses modelos foram reconstruídos no programa *3DSOM Pro* da *Creative Dimension Software Ltd*. [47], ratificando a importância da imagem 2D, dessa vez otimizando o tempo de produção do ambiente, acelerando as etapas de modelagem e texturização.

Uma série de fotos são tiradas com uma câmera fotográfica digital fixada em um tripé. A câmera fica imóvel, o que gira é a base com o objeto. A base possui uma imagem que determina o sistema de calibração que servirá para o programa reconhecer a posição da câmera e reconstruir o objeto. Cada seqüência de fotos registra o objeto num giro de 360º de uma determinada posição da câmera. Estas seqüências de fotos são tiradas de pelo menos duas posições diferentes da câmera (figura 9.a).

No *3DSOM* as fotos são manipuladas para isolar o fundo e indicar apenas o objeto que o programa deve reconstruir através de uma máscara. Quando o programa termina o processamento de todas as imagens, ele gera o modelo e, caso seja solicitado, sua textura.

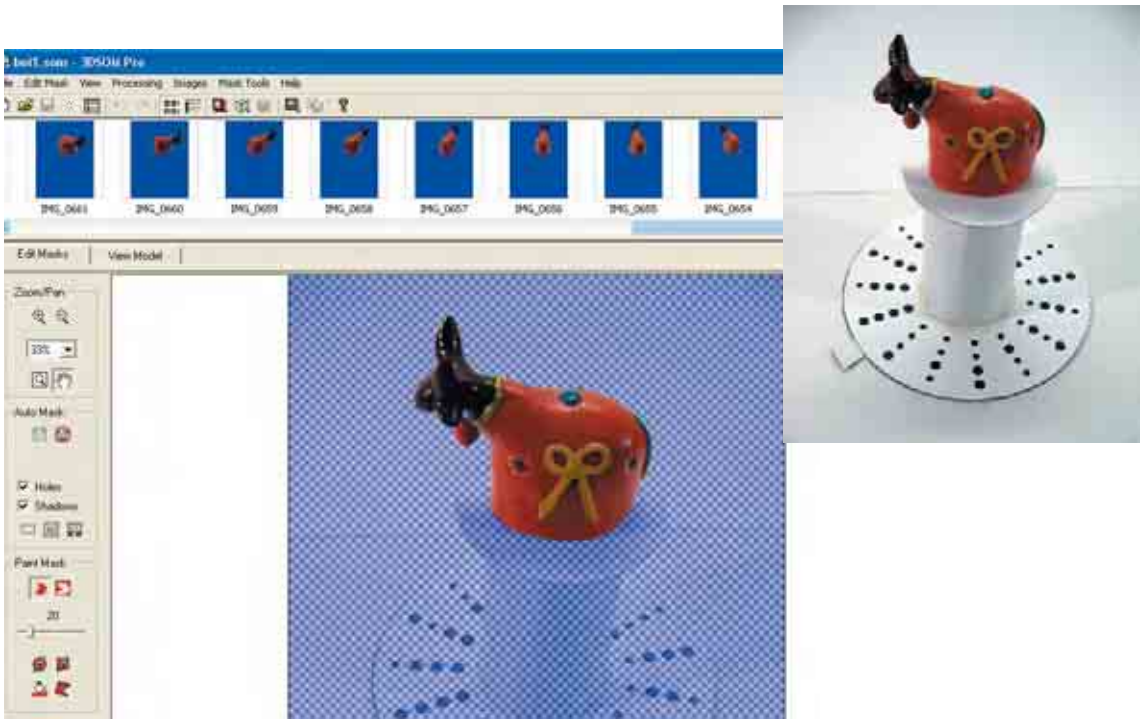


FIGURA 9.a. Programa *3DSOM Pro*. Fotos com máscaras preparadas para gerar o modelo. Ao lado uma das fotos da sequência de imagens do boi, mostra a base com o sistema de calibração.

Alguns objetos foram modelados no 3ds max. A organização do ambiente e a aplicação das técnicas, em sua maioria, feitas também no *3ds max*. Algumas técnicas de otimização, como o LOD e o *Mipmap*, são escritas direto no código de VRML, nos programas *VRML pad* ou no *Note pad*.

Para a boneca foram gerados dois modelos no *3DSOM*. O primeiro com 3.328 polígonos e o segundo com 1.000 (figura 9.c e 9.d), os dois modelos com suas respectivas texturas. Os modelos foram exportados como *3ds*, formato aceito para importação no *3ds max*. As texturas foram exportadas no formato *jpg*. No 3ds max foi gerado o mapa de normais (figura 9.e) para o modelo de baixa resolução, a partir de uma projeção do modelo de alta resolução, criado com o recurso *projection mapping* disponível no *render to texture* (figura 9.b). O mapa de normais aplicado no canal de *bump* do material do modelo de baixos polígonos permite uma visualização desse modelo muito aproximada ao modelo de alta resolução (figura 9.f).

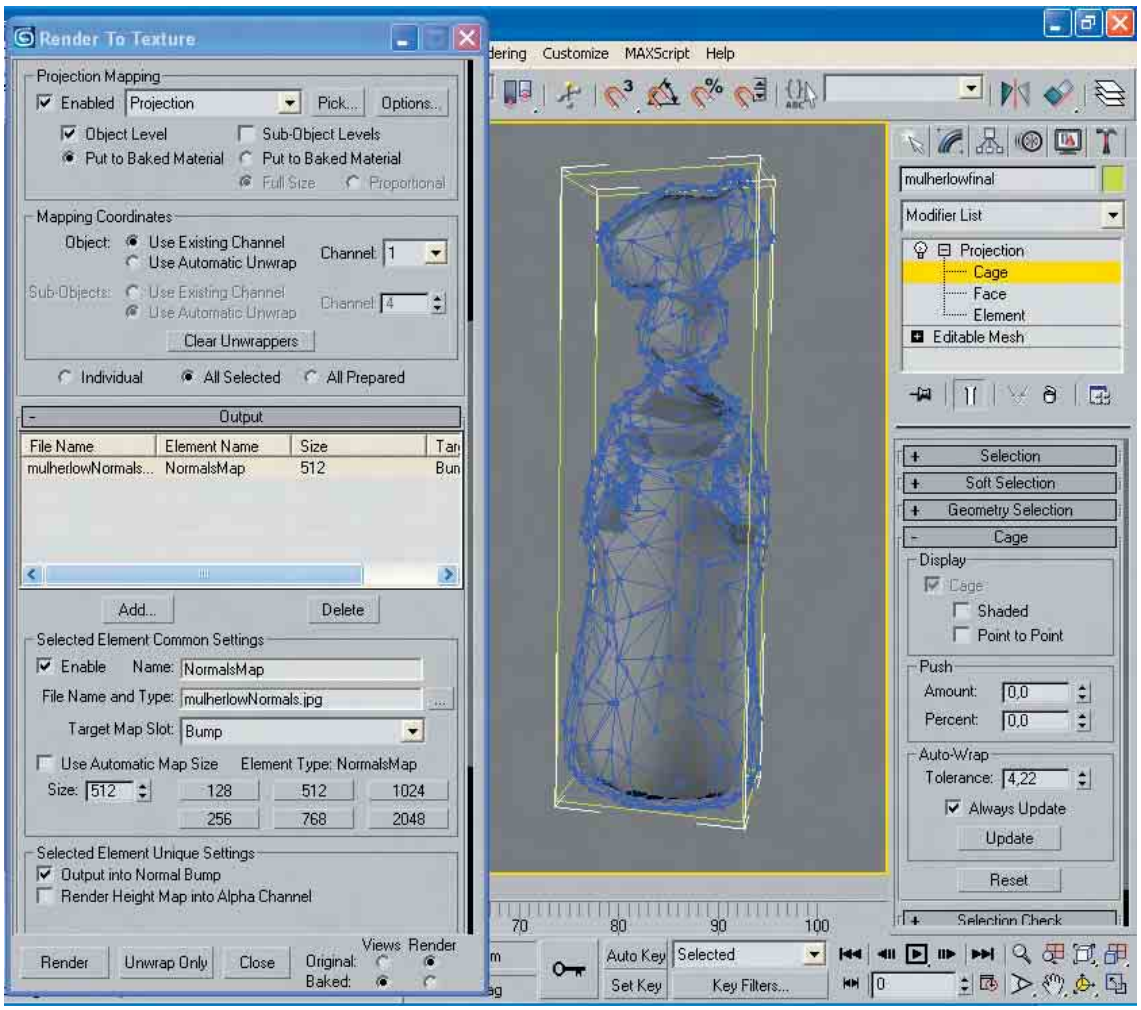


FIGURA 9.b. Modelo *low-poly* com a gaiola (*cage*) de projeção do modelo *high-poly*, *render to texture* preparado para gerar o mapa de normais.



FIGURA 9.c. À esquerda modelo *high-poly*, à direita modelo *low-poly*, modelos originais sem textura.

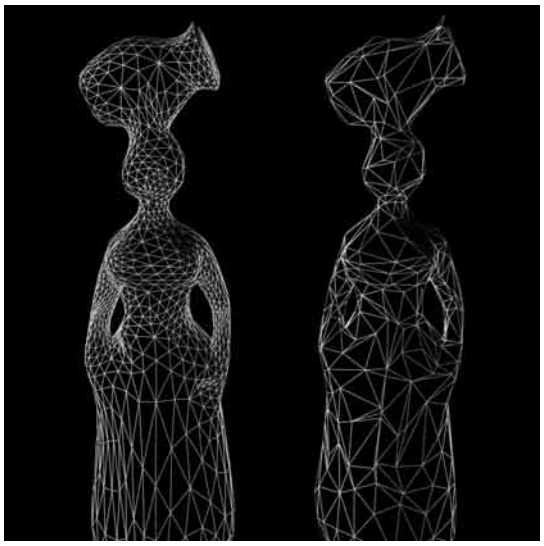


FIGURA 9.d. Wireframe dos modelos *high-poly* e *low-poly*.

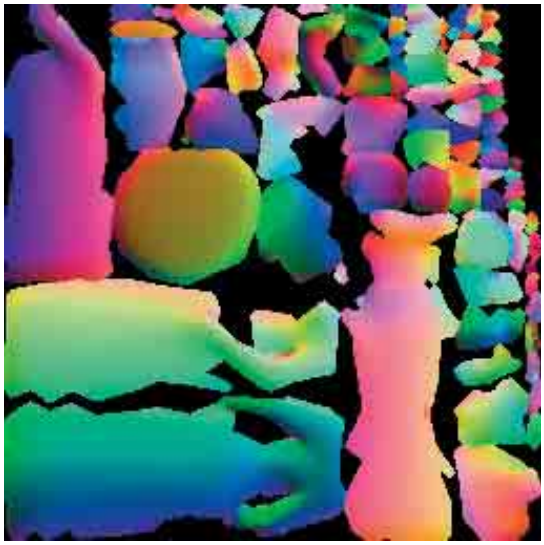


FIGURA 9.e. *Normal map* gerado pelo *render to texture* e aplicado ao modelo *low-poly*.



FIGURA 9.f. À esquerda modelo *high-poly*, à direita modelo *low-poly*, com o *normal map* aplicado.

O boi foi criado também no *3DSOM* a partir de fotos (figura 9.a). O modelo foi gerado com 3.200 polígonos e com textura. O segundo boi é uma instância do primeiro e sua textura é uma manipulação, feita no *Photoshop*, com alteração de cor da textura do primeiro boi. Foram criadas mais três texturas para cada um, para serem usadas como *mipmaps* (figura 9.g).

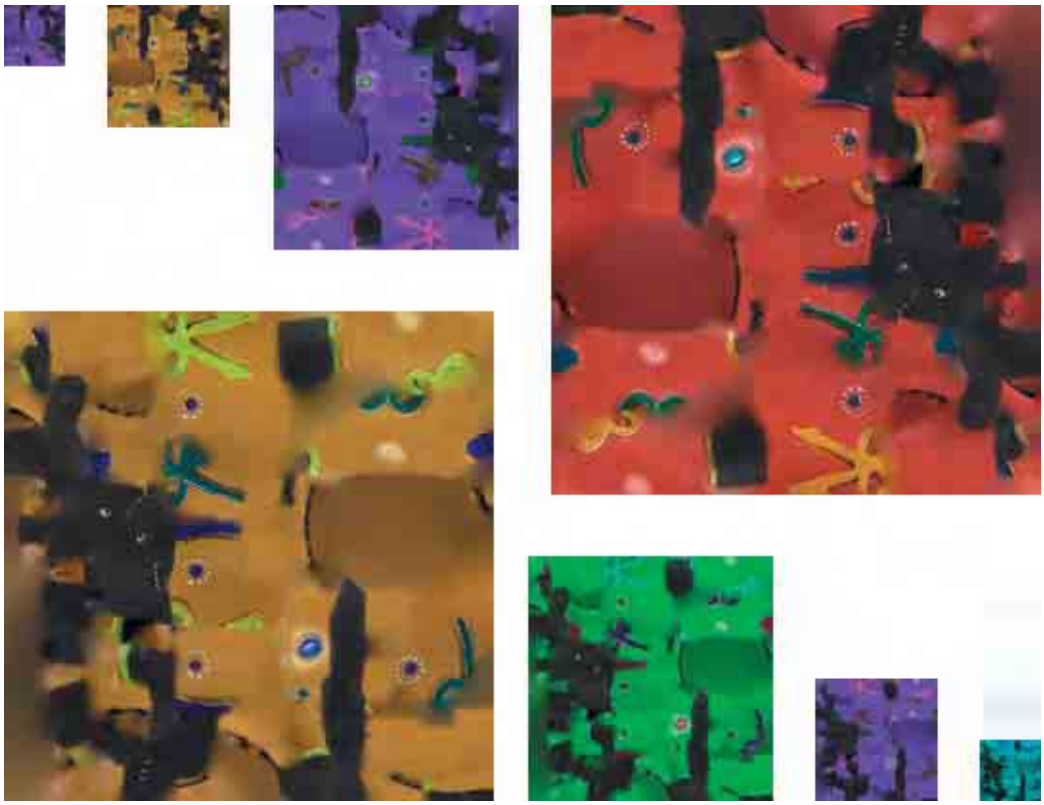


FIGURA 9.g. Mipmaps dos bois, feitas a partir do mesmo mapa, ambos os conjuntos de mipmaps possuem imagens com as dimensões: 32 x 32, 64 x 64, 128 x 128 e 256 x 256 pixels.

O muro foi construído no *3ds max* com uma linha pela vista de topo (*line* do *splines/shapes*). Foi aplicado o *outline* no *spline* para gerar uma espessura para as paredes e o modificador *extrude* para definir a altura do muro. A *line* com o modificador *extrude* foi transformada em *editable poly* para diminuir a quantidade de informações. Para a textura do muro, assim como a do terreno, um mesmo bitmap é repetido (*tiling*) diversas vezes, como uma parede preenchida de azulejos. Para que as emendas, junção dos *bitmaps*, não sejam percebidas, é necessário uma manipulação na imagem para que as bordas, esquerda/direita e superior/inferior, sejam iguais. Essa transformação foi feita no *Photoshop*, cortando com a seleção por *feather* as partes que emendam uma imagem na outra e espelhando partes copiadas da borda da imagem original (figura 9.h).

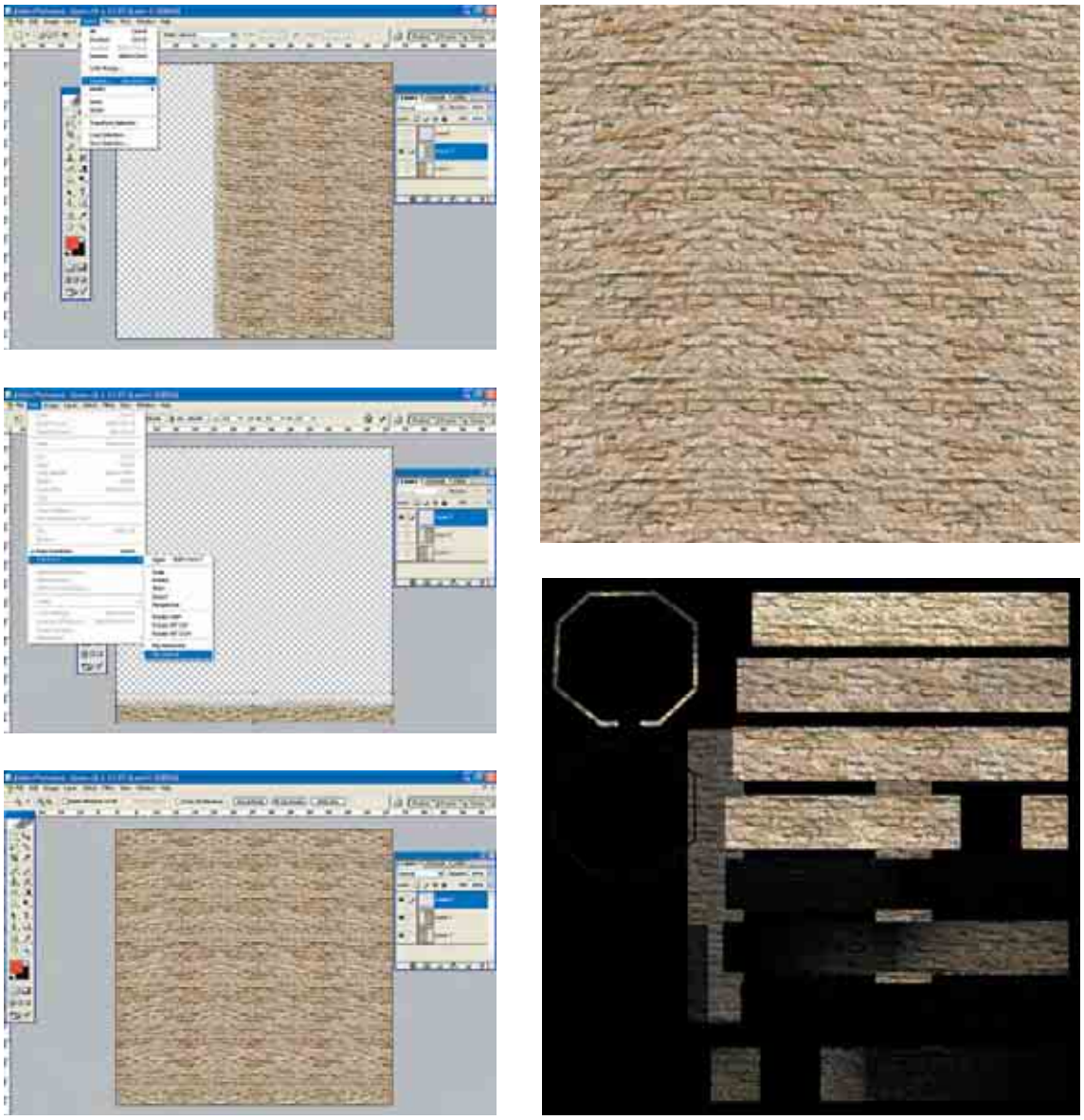


FIGURA 9.h. À esquerda, algumas etapas da manipulação da imagem no *Photoshop*. À direita, em cima, imagem que foi repetida (*tiling*) no material do muro. Abaixo, imagem do mapa completo, com efeitos de iluminação, gerado pelo *render to texture*.

Como para o bom funcionamento do *render to texture* é necessário expor os UVWs de um objeto, foi aplicado no muro o modificador *unwrap UVW*. O *render to texture* foi usado para criar o mapa completo do muro (*compete map*) com a textura e os efeitos da iluminação (figura 9.h).

Para a casa foi gerado no *3DSOM* um modelo com 2.400 polígonos e com textura. O *3DSOM* é ótimo para reconstrução de modelos orgânicos com linhas sinuosas, mas para objetos com formas mais retas, como a casa, o programa gera polígonos demais e acaba distorcendo o modelo. Caso seja solicitada uma reconstrução com poucos polígonos, o modelo chega a um nível tão grande de distorção que deixa o objeto irreco-

nhecível. Por isso, o modelo em baixa foi construído no *3ds max* e possui apenas 104 polígonos (figura 9.i). Ainda no *3ds max* foram gerados, para o modelo de baixa resolução, o mapa de normais e um mapa completo (*complete map*), a partir de uma projeção do modelo de alta resolução, criado com o recurso *projection mapping* disponível no *render to texture*.

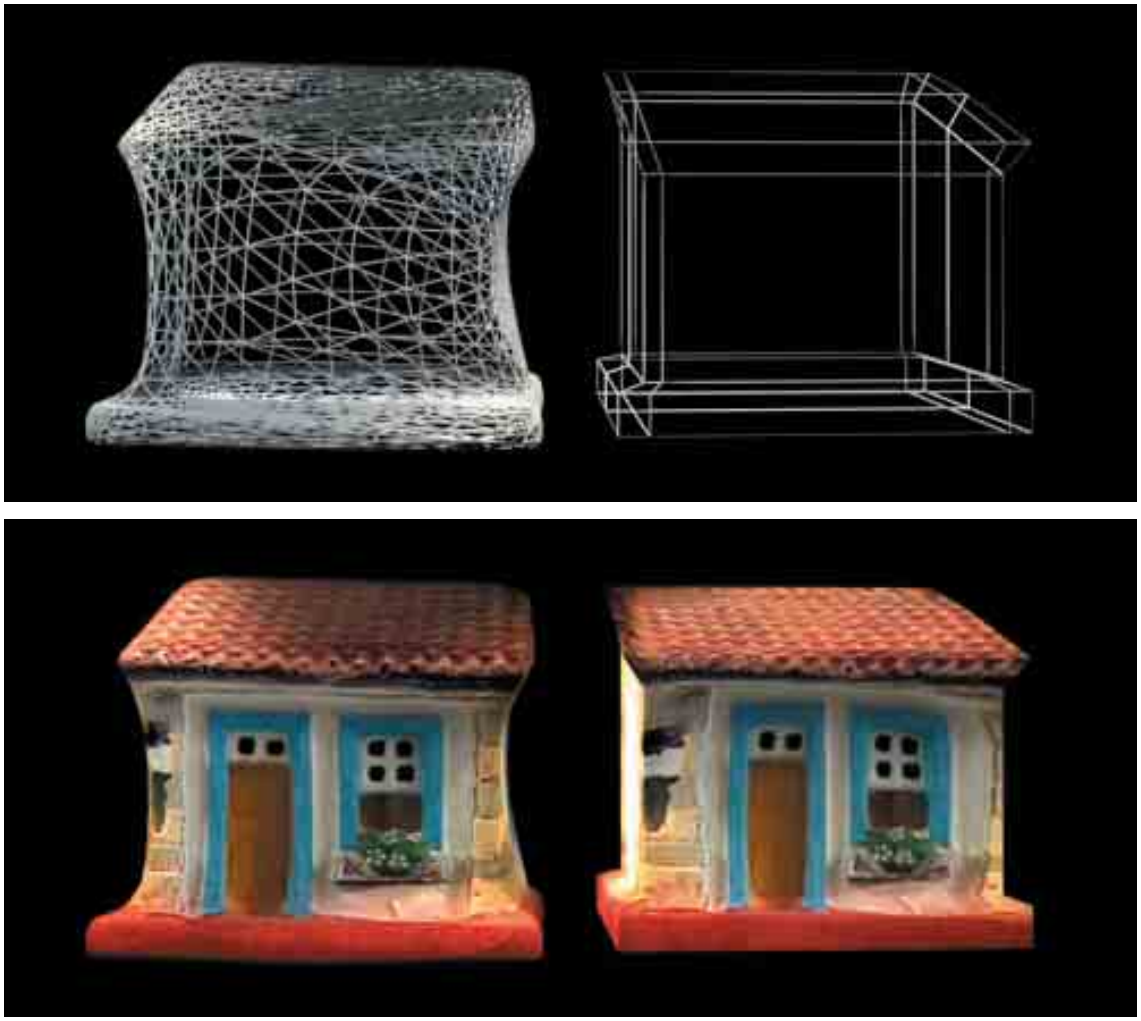


FIGURA 9.i. Wireframe e modelos com textura aplicada: *high-poly* e *low-poly*.

O terreno, base do ambiente virtual, foi gerado a partir de um plano com 3.200 polígonos. Essa numerosa quantidade de polígonos serviu para que o modificador *displace* funcionasse como era esperado. O *displace* usa uma imagem ou um algoritmo para alterar a geometria de um objeto, diferente do *bump* e do *normal map* que apenas simulam o relevo. O *displace* interpreta um *bitmap*, mesmo os coloridos, como uma escala de tons de cinza, para criar os altos e baixos relevos na superfície de um modelo, a partir das regiões claras e escuras da imagem.

O material do terreno foi construído com os *bitmaps* “terra.jpg” no canal *diffuse* e “pedras.jpg” no canal *bump* (figura 9.j). Foram usados dois modificadores *displace* para deformar a geometria, no primeiro foi usado o *bitmap* “terreno.jpg” e no segundo “montanhas.jpg”.

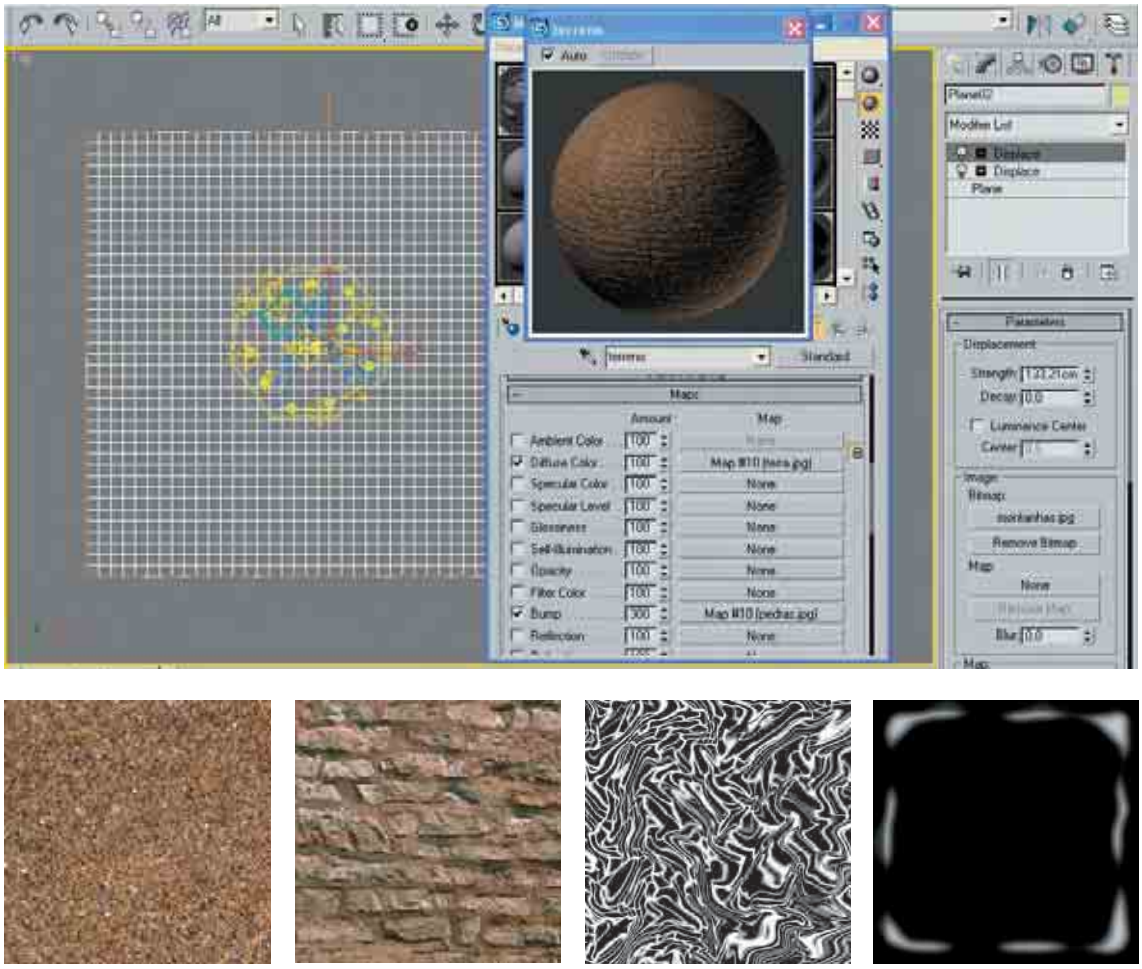


FIGURA 9.j. Material do terreno, visto pelo editor de materiais, e mapas usados para construir o material e o relevo na geometria do terreno.

Definida a modelagem do terreno foi gerada uma solução de iluminação avançada, o *radiosity*, através do painel *render* do *3ds max* (figura 9.l). O *render to texture* foi aplicado para gerar um mapa completo (*complete map*), o resultado de todas as texturas aplicadas e dos efeitos de iluminação em um único *bitmap*. O mapa completo foi gerado duas vezes, com e sem a solução de radiosidade (figura 9.n e 9.o). Nos testes feitos, foi observado que o *render to texture* não renderiza corretamente o mapa de textura de um objeto submetido a opção de controle de exposição (*exposure control*) do *radiosity*.

Os dois mapas, com e sem o *radiosity*, foram mesclados e manipulados no *Photoshop* para produzir a imagem final (figura 9.m). Esse último procedimento não é

necessário, foi uma escolha para atingir o resultado pretendido. Ainda no render to texture, foi criado o *normal map* que também foi manipulado no *Photoshop*.

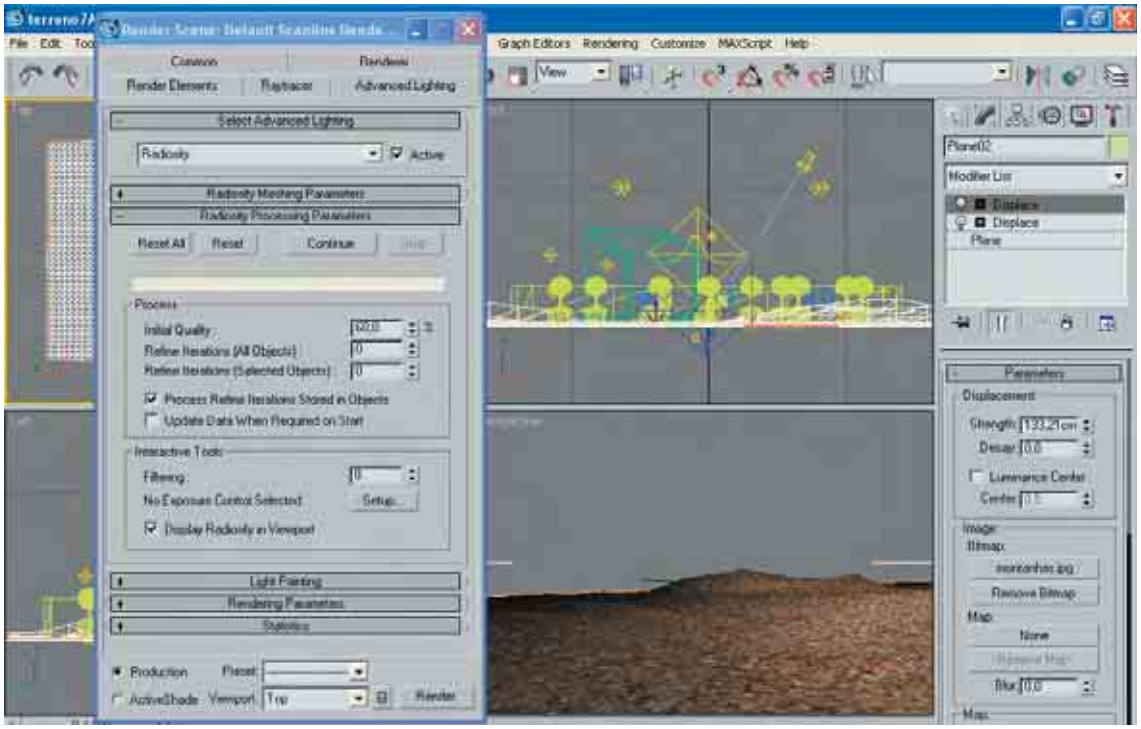


FIGURA 9.l. Solução de radiosidade sendo aplicada no ambiente.

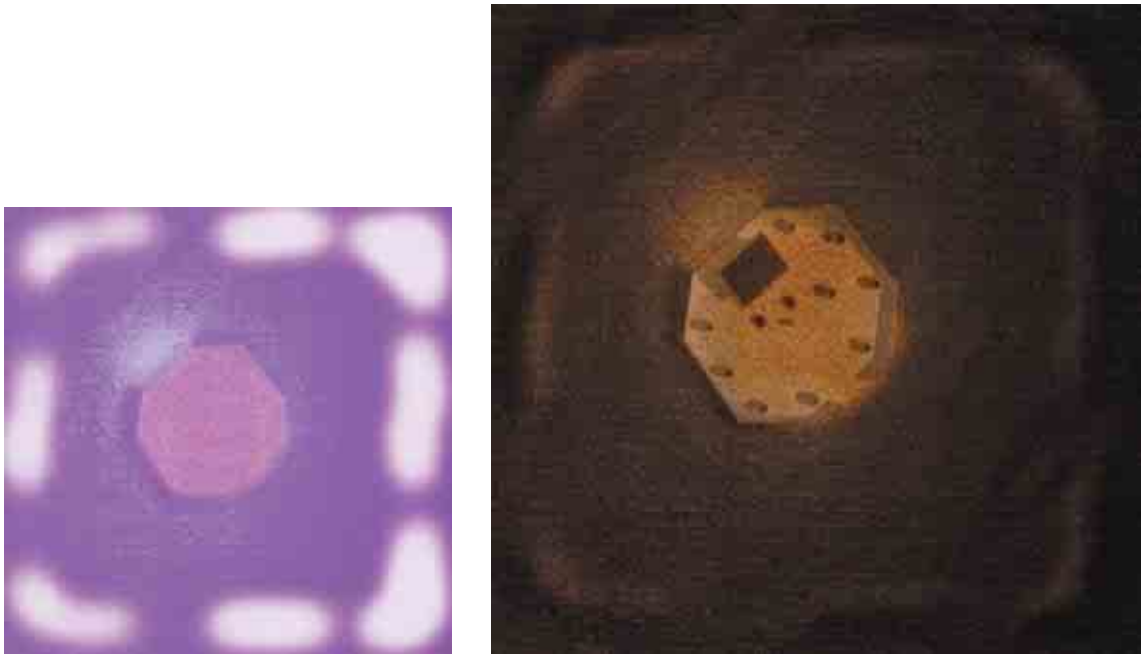


FIGURA 9.m. Mapas finais após manipulação no *Photoshop*. *Normal map* e *compete map*, respectivamente.

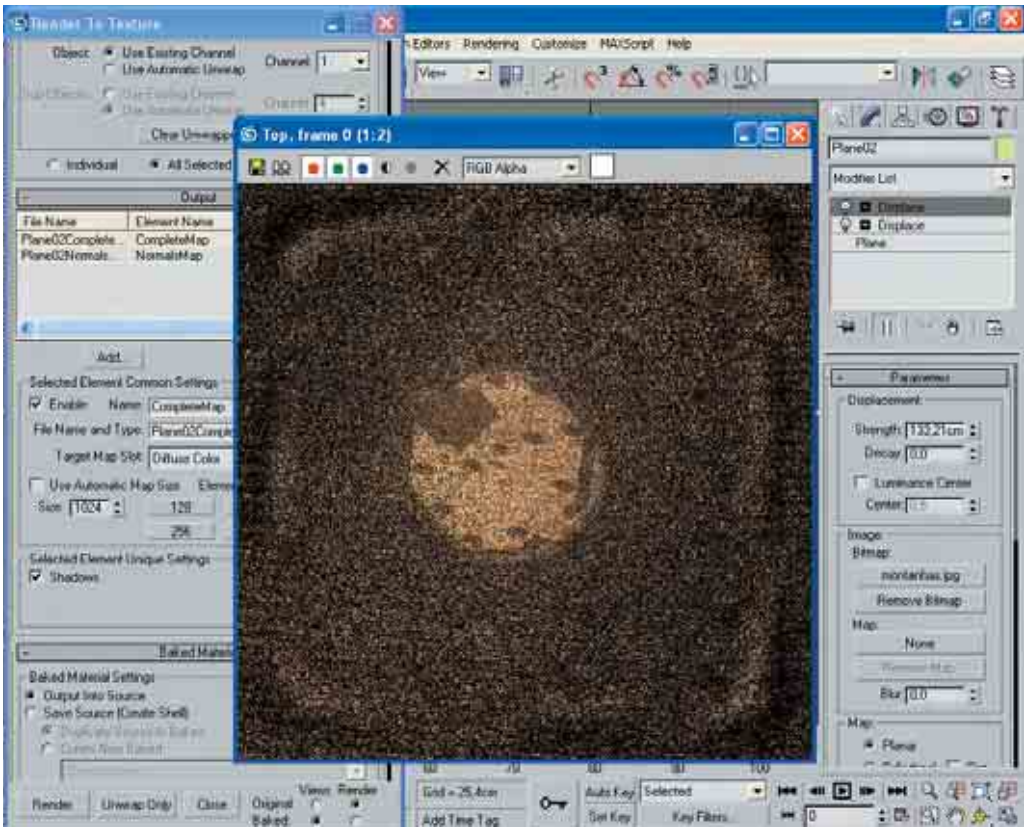


FIGURA 9.n. *Compete map* gerado sem *radiosity*.

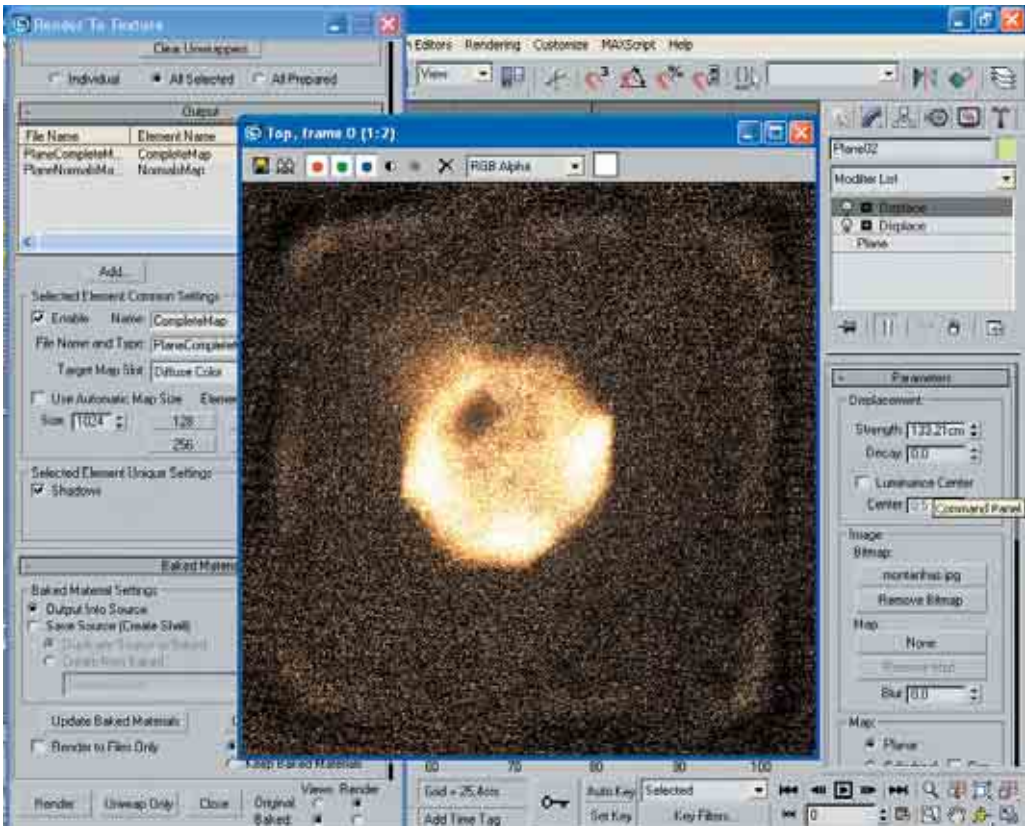


FIGURA 9.o. *Compete map* gerado com *radiosity*.

O terreno final usado no ambiente virtual otimizado possui apenas 2 polígonos e tem como mapas de textura – o *complete map* e o *normal map* – as imagens finais geradas a partir do terreno de 3.200 polígonos e manipuladas no *Photoshop* (figura 9.m).

A árvore também foi gerada no *3d som*. Só existe um modelo de árvore, as outras árvores são instâncias da primeira. No código VRML, a primeira árvore é definida através da descrição de sua geometria, mas as outras árvores têm apenas as coordenadas de localização, a geometria é construída usando a descrição da primeira.

```
DEF arvore_menor02 Transform {
 translation 0.009056 -3.04 -10.5
 scale 0.02333 0.02333 0.02333
 children [
  Shape {
        appearance AdvancedAppearance {
              material Material {diffuseColor 0 0 0}
              textures [
                    ImageTexture      {url “arvotimiz.jpg”}
                    DEF bm BumpMap {
                          direction 1 0 1
                          texture   ImageTexture {url “arvnormals.jpg”}
                    }
              ]
              backgroundFactor [“ZERO”,”ZERO”]
              foregroundFactor [“ONE”,”BACK_COLOR”]
        }
  geometry USE arvore_menor01-FACES
  }
 ]
}
```

(**Definição da segunda árvore do protótipo em VRML.** Parte do código do arquivo final em VRML do protótipo otimizado.)

As árvores também usam dois mapas de textura, um como *ImageTexture* e outro, feito a partir do *render to texture* no *3ds max*, o *normal map*.

O céu poderia ser feito através de geometrias texturizadas, como *box*, cilindro ou *domo*. Porém, para não sobrecarregar o processamento geométrico, foi usado no próprio VRML o *node* (nó) de *background*. O nó de *background* é usado para especificar uma determinada cor ou uma textura para cada vista do ambiente: *back*, *bottom*, *front*, *left*, *right* e *top*. A preparação do *background* também foi feita através da opção para preparação de arquivos para exportação em VRML97, no *helpers*, em *create* do painel de comando do *3ds max*, escolhendo *background* no *object type*. Esse nó de *background* pode ser escrito diretamente no código em VRML.

```
DEF Background01menor Background {
  skyColor [1 1 1, ]
  groundColor [0.7 0.4 0.3, ]
  backUrl “nuvens.jpg”
  bottomUrl “nuvens.jpg”
  frontUrl “nuvens.jpg”
  leftUrl “nuvens.jpg”
  rightUrl “nuvens.jpg”
  topUrl “nuvens.jpg”
}
```

(**Definição do background em VRML.** Parte do código do arquivo final em VRML do protótipo otimizado.)

Os nós de *anchor* definem os modelos geométricos – os bois e a mulher – como objetos clicáveis que apontam para *links*. A preparação do *anchor* dos bois da cena também foi feita no *3ds max*, em VRML97 no *helpers*, na opção *anchor* em *object type* (figura 9.p). Como para a mulher foram implementados no código VRML, o “LOD” e o “EXTERNPROTO AdvancedAppearance” para incluir o *normal map*, optou-se também por fazer o *anchor* direto no código em VRML.

```
Anchor {
  description “exibir high-poly”
  parameter “”
  url “mulher_high.html”
  children [
LOD {
range [ 30, 40 ]
level [
# level 0 mulher
Transform {
  translation -0.009704 -1.129 0.9133
  rotation 0.008904 0.9992 0.03923 -0.8471
  scale 0.009702 0.009702 0.009702
  children
Shape {
appearance DEF AP AdvancedAppearance {
                material Material {diffuseColor 0 0 0}
                textures [
                        ImageTexture    {url “mulher_low.jpg”}
                        DEF bm BumpMap {
                                direction 1 0 1
                                texture  ImageTexture {url “Normal07.jpg”}
                        }
                ]
                backgroundFactor [“ZERO”,”ZERO”]
                foregroundFactor [“ONE”,”BACK_COLOR”]
        }
geometry IndexedFaceSet {
   ccw TRUE
      solid TRUE
      coord DEF mulherlowmenor-COORD Coordinate { point [
```

(***Anchor*, LOD e *bump map* da personagem feminina do protótipo em VRML.** Parte do código do arquivo final em VRML.)



FIGURA 9.p. *Anchor* sendo gerado a partir do *3ds max*.

Para que o *bump map* seja compreendido pelo *plugin Cortona VRML Client,* é preciso usar, no início do código, as extensões desenvolvidas pela *ParallelGraphics*: o “EXTERNPROTO AdvancedAppearance” e o “EXTERNPROTO BumpMap”.

```
#VRML V2.0 utf8

EXTERNPROTO       MipMap [
   exposedField MFNode          levels
]
[ “urn:ParaGraph:MipMap” “http://www.parallelgraphics.com/vrml/proto/Cortona/extensions.wrl#MipMap” ]

EXTERNPROTO AdvancedAppearance [
   exposedField SFNode material
   exposedField MFNode textures
   exposedField MFString mappingTypes
   exposedField MFFloat weights
   exposedField SFFloat materialBlending
   exposedField MFNode textureTransforms
   exposedField MFString backgroundFactor
   exposedField MFString foregroundFactor
]
[ “urn:ParaGraph:AdvancedAppearance”
  “http://www.parallelgraphics.com/vrml/proto/Cortona/extensions.wrl#AdvancedAppearance”
]

EXTERNPROTO       BumpMap [
   exposedField SFNode texture
   exposedField SFVec3f direction
]
[ “urn:ParaGraph:BumpMap”
  “http://www.parallelgraphics.com/vrml/proto/Cortona/extensions.wrl#BumpMap”
]
```

(**Início do código em VRML do arquivo final do protótipo otimizado.**)

O trecho do código acima exibe também a extensão utilizada para visualização de *mipmaps* no *Cortona VRML Client*. A aplicação de *mipmaps* na textura de um modelo já foi descrita na página 42, em *mipmaps* no capítulo 8.

As luzes, assim como as câmeras, são incluídas como objetos paramétricos na maioria dos programas de modelagem 3D. Nesse protótipo foram inseridas dez luzes *omni* e uma luz *target directional* (figura 9.q). A sombra foi ativada apenas para a *target directional light* que projeta os raios paralelos, simulando a iluminação do sol que, devido à extrema distância da Terra, parece também projetar os raios em paralelismo. As luzes *omni*, que iluminam em todas as direções, foram organizadas na forma de um domo que envolve todo os objetos do cenário, para simular iluminação ambiente e iluminação indireta existente no mundo real, mas a opção de projeção de sombras para essas luzes não foi ativada.

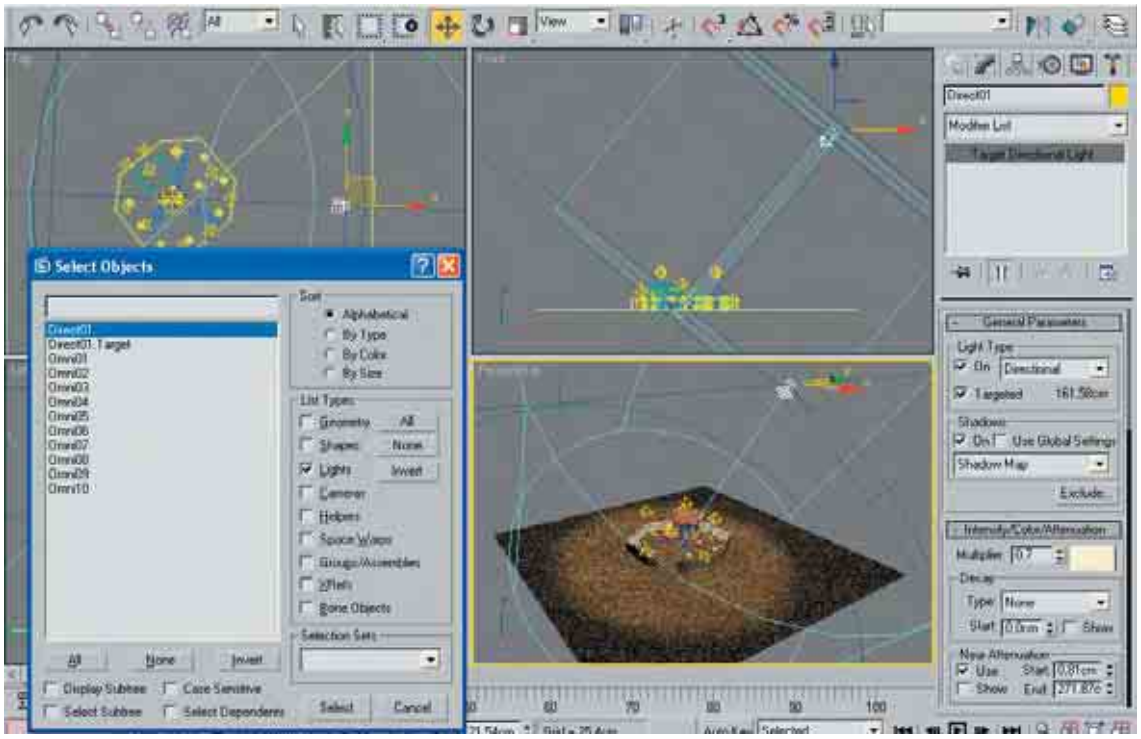


FIGURA 9.q. Organização das luzes no ambiente virtual.

Três câmeras foram colocadas no ambiente para auxiliar a visualização e navegação em tempo real. Deve-se ter o cuidado de colocar as câmeras na mesma altura especificada para o *avatar* no *NavInfo*. Caso contrário, ao iniciar a navegação a partir de um ponto de vista dado por determinada câmera, poderá ocorrer uma mudança brusca na altura do ponto de visualização.

Com o *helpers* do *create*, no painel de comando do *3ds max*, em VRML97, foi incluído o *NavInfo* (NavigationInfo) do *object type* (figura 9.r). O nó de *NavInfo* define as características de visualização e navegação do *avatar*. Entende-se por *avatar* o próprio

usuário que está navegando no ambiente virtual. O tamanho determinado para o *avatar* define a altura do ponto de vista do usuário em relação ao ambiente virtual. No 3ds max, o sistema de navegação pode ser indicado como: *walk*, *examine* e *fly*. No código em VRML, o sistema de navegação pode ser ampliado para o uso simultâneo de mais de um tipo de navegação.



FIGURA 9.r. NavInfo e seu painel para ajustes do controle de navegação.

Finalizada a construção e organização do cenário virtual no *3ds max*, o ambiente foi exportado para o formato VRML.

Algumas técnicas de otimização – LOD, EXTERNPROTO MipMap, EXTERNPROTO AdvancedAppearance, EXTERNPROTO BumpMap – foram escritas diretamente no código VRML, no programa *Vrml Pad* (ver parte do código nas págs. 53 e 54). Para aplicar o LOD na personagem feminina, ainda no *3ds max*, foi criado um *box* (figura 9.s) na mesma posição e do mesmo tamanho do modelo da personagem e exportado, também, para o formato VRML. A textura é um *print screen* da imagem já visualizada pelo *plugin Cortona*, apenas a personagem foi selecionada na imagem, ao fundo aparece um pouco da porta. O LOD dessa pesonagem possui o modelo *low-poly* com o *normal map* aplicado no *level 0* (ver pág. 53). Os *levels 1* e *2* foram preparados, escrevendo-se o código, de forma que o *level 1* contém a geometria e a textura simplificadas do *box* e o *level 2* não possui objeto algum. A partir de uma determinada distância, valor estipulado em "*range*", do ponto de vista à personagem, esta não será mais processada.

```
# level 1 - mulher box
Transform {
 translation -0.02467 0 0.8503
 rotation 0 1 0 -0.7953
 children [
  Shape {
   appearance Appearance {
    material Material {
     diffuseColor 0.588 0.588 0.588
     ambientIntensity 1.0
     specularColor 0 0 0
     shininess 0.145
     transparency 0
    }
    texture ImageTexture {
     url “mulherbox.jpg”
    }
   }
   geometry DEF Box01-FACES IndexedFaceSet {
    ccw TRUE
    solid TRUE
    coord DEF Box01-COORD Coordinate { point [
     -0.1971 0 0.2946, 0.1971 0 0.2946, -0.1971 0 -0.2946,
     0.1971 0 -0.2946, -0.1971 2.816 0.2946, 0.1971 2.816 0.2946,
     -0.1971 2.816 -0.2946, 0.1971 2.816 -0.2946]
    }
    normal Normal { vector [
     1 0 0, -1 0 0, 0 1 0, 0 0 1, 0 -1 0, 0 0 -1, ] }
    normalPerVertex TRUE
    texCoord DEF Box01-TEXCOORD TextureCoordinate { point [
     0 0, 1 0, 0 1, 1 1, 0 0, 1 0, 0 1, 1 1, 0 0, 1 0, 0 1,
     1 1]
    }
    coordIndex [
     0, 2, 3, -1, 3, 1, 0, -1, 4, 5, 7, -1, 7, 6, 4, -1, 0, 1, 5, -1,
     5, 4, 0, -1, 1, 3, 7, -1, 7, 5, 1, -1, 3, 2, 6, -1,
     6, 7, 3, -1, 2, 0, 4, -1, 4, 6, 2, -1]
    texCoordIndex [
     9, 11, 10, -1, 10, 8, 9, -1, 8, 9, 11, -1, 11, 10, 8, -1,
     4, 5, 7, -1, 7, 6, 4, -1, 0, 1, 3, -1, 3, 2, 0, -1,
     4, 5, 7, -1, 7, 6, 4, -1, 0, 1, 3, -1, 3, 2, 0, -1]
    normalIndex [
     4, 4, 4, -1, 4, 4, 4, -1, 2, 2, 2, -1, 2, 2, 2, -1, 3,
     3, 3, -1, 3, 3, 3, -1, 0, 0, 0, -1, 0, 0, 0, -1, 5,
     5, 5, -1, 5, 5, 5, -1, 1, 1, 1, -1, 1, 1, 1, -1, ]
    }
   }
  ]
}

# level 2 - empty
WorldInfo {}
]
}
]
}
```

(**Trecho do código do LOD em VRML: levels 1 e 2.**)

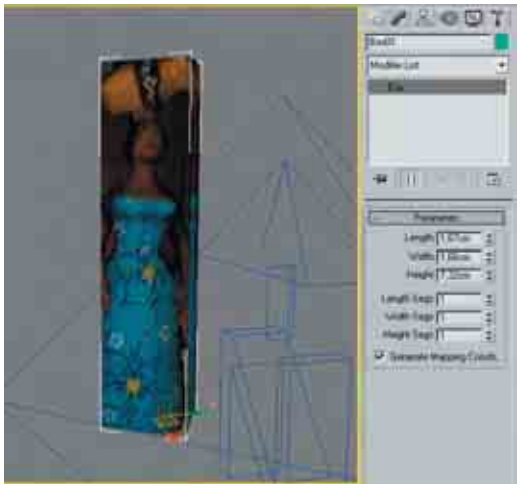


FIGURA 9.s. Box construído para o *level 1* do LOD.

A *ParallelGraphics* informa, na seção *Developer Zone* de seu site, que o EXTERNPROTO BumpMap – extensão criada para uso de bump map e de normal map – só funcionará quando forem selecionados, em *preferences* do *plugin Cortona VRML*, o *DirectX Renderer* ou, em computadores com o processador *Pentium IV*, o *R98 Renderer*. Apesar do aviso, os testes mostraram que o modelo, com sua textura aplicada como ImageTexture e seu mapa de normal no BumpMap, também conseguiu um bom resultado com a opção do *OpenGL Renderer*.

O arquivo VRML para ser publicado na *web* precisa ser incorporado em um arquivo HTML e para ser visualizado precisa que o usuário tenha instalado algum dos *plugins* (*3D players*) de VRML. O arquivo "index.html", construído para embutir o VRML deste protótipo, possui um código que detecta se o computador do usuário possui o *plugin Cortona VRM Client*, que foi o *plugin* escolhido neste trabalho. Caso o computador não possua esse *plugin*, uma janela é aberta para que o usuário decida se quer instalá-lo (figura 9.t). A instalação será feita a partir de um *download* diretamente do site da *ParallelGraphics* [6].

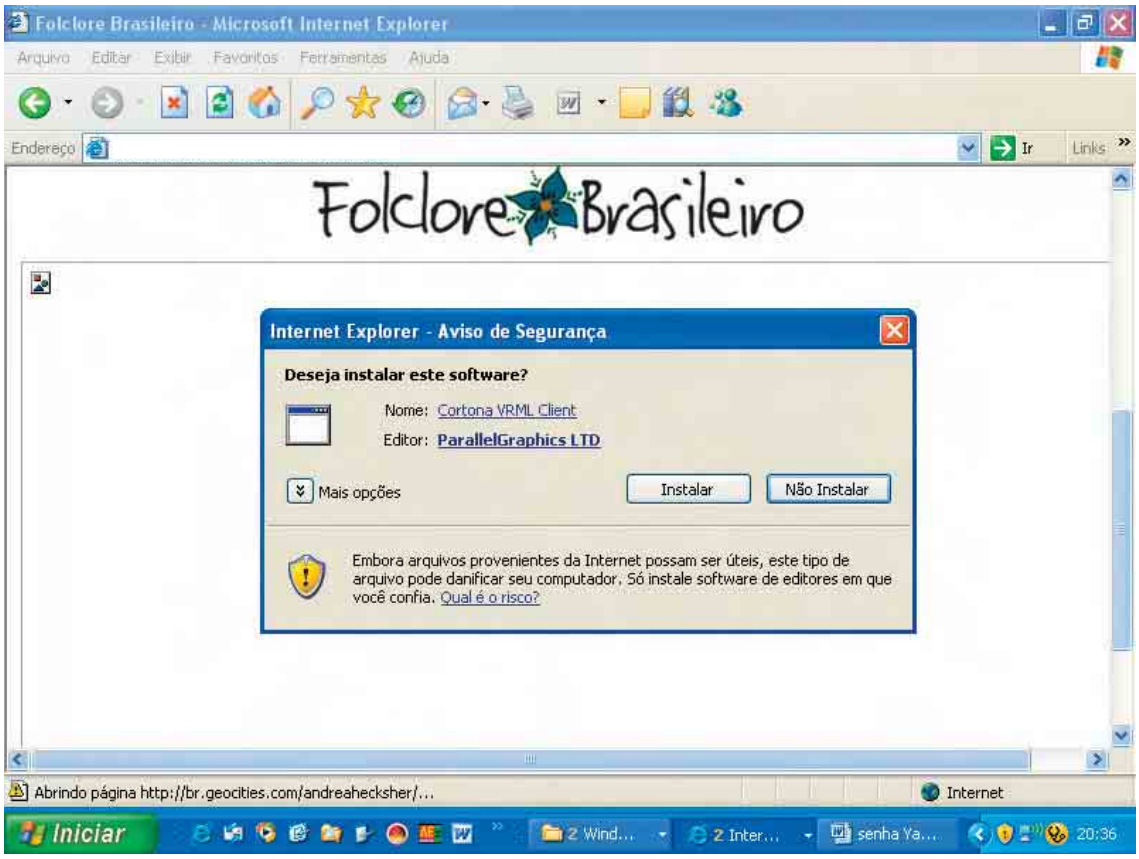


FIGURA 9.t. Aviso para a instalação do *plugin Cortona VRML*.

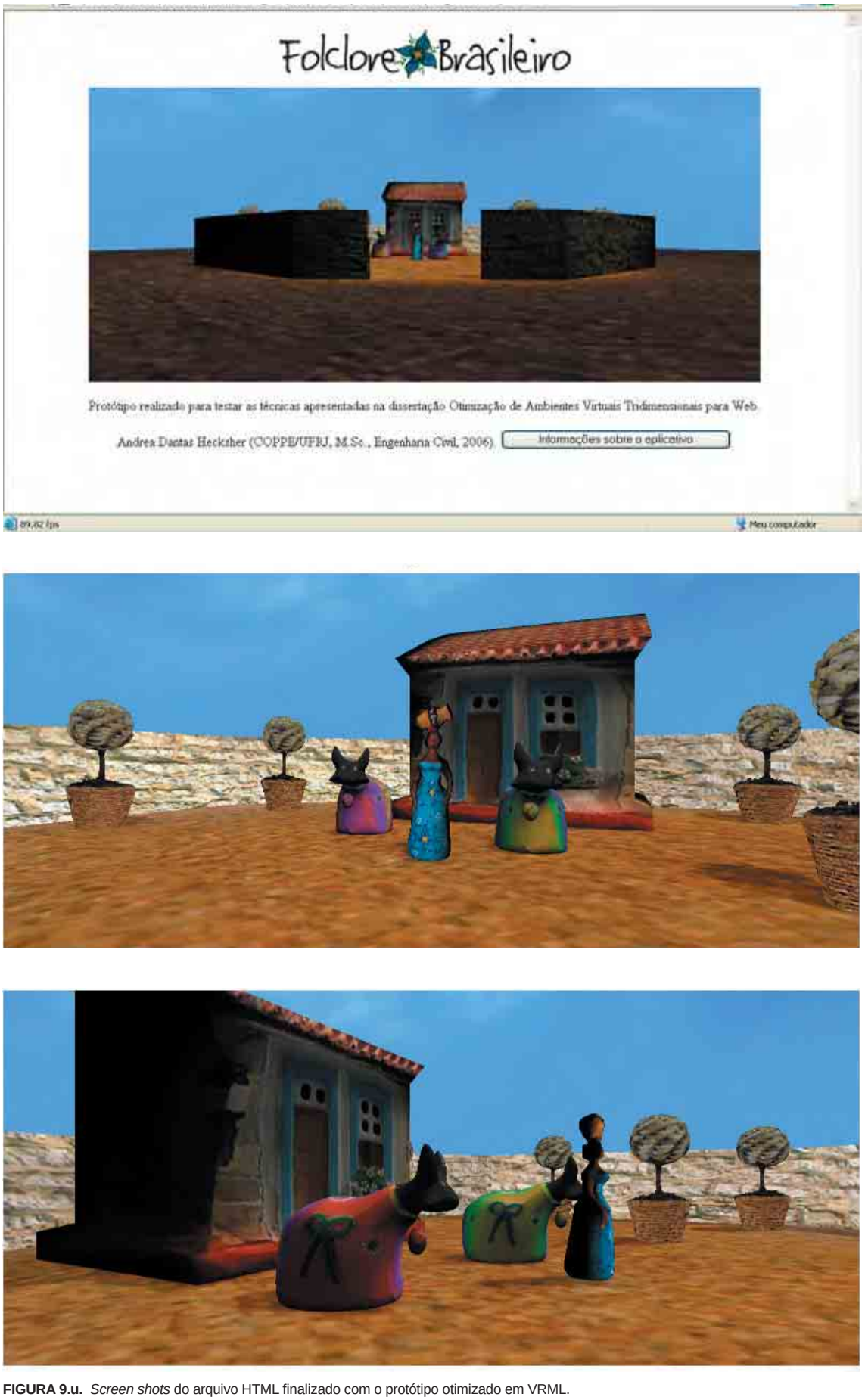


FIGURA 9.u. *Screen shots* do arquivo HTML finalizado com o protótipo otimizado em VRML.

10. Conclusão

Adiantando propostas para trabalhos futuros, foram implementados no protótipo outros arquivos, em HTML, VRML e *Flash*. Um dos objetos da cena é um *link* que ao ser clicado abre um novo arquivo HTML com um VRML incorporado. Apesar de não ser o escopo dessa dissertação, esse arquivo VRML serve para demonstrar a vantagem dessa linguagem também para exibir animações. Ao acessar esse arquivo, é como se o usuário estivesse assistindo um vídeo totalmente interativo, onde existe a possibilidade de ter várias câmeras para visualização de diversos ângulos e ainda navegar pela cena animada. Nesse arquivo, foi inserido um som no formato *mp3* através do nó *sound* do VRML. O *sound* é colocado na cena como se fosse um objeto porque é preciso definir uma posição para ele no sistema de coordenadas do ambiente virtual. Essa posição determina o local do ambiente onde o som pode ser ouvido com maior intensidade. Quando o usuário começa a se afastar desse ponto, o som vai diminuindo progressivamente até sumir.

O tempo de carregamento da animação e a possibilidade de manter uma boa qualidade visual, mesmo com compressão das imagens e do arquivo VRML, ratificam a importância dessa linguagem na *web*, podendo em certos casos substituir o uso de tipos de arquivos de vídeo convencionais em apresentações com animação, como apresentações de produtos e da forma de utilização desses. A questão da necessidade de *plugin* para visualização de arquivos VRML não invalida essa proposta porque todos os formatos de vídeo encontrados na *web* exigem também a instalação de *plugins*. Bons exemplos de como as aplicações 3D na *Internet* podem impulsionar o *e-business* e trazer soluções para o *e-commerce* são encontrados na seção "*showroom*" no *site* da *ParallelGraphics* [48].

Outro objeto da cena principal do protótipo também é um *link* que ao ser clicado abre um arquivo HTML com conteúdo em *Flash*. Mesmo não sendo o foco deste trabalho, evidencia a integração do VRML com outros tipos de arquivos. Um site pode ter uma parte em VRML e outra, por exemplo, em *Flash*. Botões em *Flash* podem abrir páginas HTML com conteúdo em VRML e objetos em VRML podem abrir páginas HTML com conteúdo em *Flash*. Além disso, uma animação em *Flash* ou em um formato de arquivo de vídeo (MPEG ou MOV) podem ser texturas de objetos em VRML.

Alguns artigos científicos, como o artigo de junho de 2006, do *site ExtremeTech* [49], especializado em tecnologias de informação, evidenciam que as pesquisas e desenvolvimentos na área de navegação em ambientes interativos tridimensionais na *web* prosseguem em busca de novas soluções. Nesse artigo foram apresentados três *browsers* 3D – *3B*, *Browse3D* e *SphereXPlorer*. Assim como os *browsers* mais convencionais, os *browsers* 3D apresentados permitem o acesso a todos os *sites* publicados na *web*, porém a forma de navegação e visualização é tridimensional.

O mesmo artigo, com base em testes de usabilidade, indica os *browser* 3D como um utilitário de entretenimento. No entanto, apontam a necessidade de maior desenvolvimento dessa tecnologia e destacam a lentidão no processamento como maior desvantagem em comparação aos *browser* 2D convencionais. Essas informações ratificam a importância desta pesquisa e apontam a necessidade de dar continuidade às pesquisas na área de visualização de ambientes tridimensionais interativos na *web*.

A partir dos estudos feitos sobre as técnicas de otimização, o LOD pode ser considerado imprescindível em ambientes virtuais amplos, como uma cidade. Quando o usuário se afasta muito de um determinado quarteirão em direção a outro, os objetos que vão ficando para trás podem sumir e assim diminuir a quantidade de modelos a serem processados em tempo real. O LOD também pode ser usado para causar efeitos; ao caminhar em direção a um determinado ponto do espaço virtual que possui geometrias armazenadas em LODs, o usuário poderá ser surpreendido com o aparecimento de um novo personagem ou objeto em cena, ou ainda, um modelo se transformar em outro.

O *mipmap* tem a mesma importância do LOD para cenários virtuais extensos e os dois podem ser usados, simultaneamente, no mesmo objeto. O *mipmap* entra como textura de uma geometria que está dentro de um LOD. O *mipmap* também pode ser usado para gerar efeitos em *real-time*, como a troca de cores da superfície de um objeto de acordo com a distância do ponto de vista, demonstrada no protótipo construído para esta dissertação. Várias soluções podem ser geradas com essa técnica; os *bitmaps* da lista de *mipmaps* podem ser imagens que variem apenas na intensidade ou posição da iluminação ou das sombras, conferindo um efeito de iluminação dinâmica.

O LOD e o *mipmap* são técnicas já muito utilizadas para organização e otimização de cenários para *games* 3D. Por ter a função de reduzir a complexidade geométrica e

visual de objetos de acordo com a distância são essenciais também na otimização de ambientes tridimensionais para processamento em tempo real na *web*.

O *displace* foi usado apenas para construir o relevo do terreno que serviu de base para gerar as texturas do terreno final otimizado. Essa técnica, assim como o *vertex shader* e as técnicas oriundas do *vertex color*, funciona melhor em malhas geométricas de alta resolução. Como a excelência no resultado visual está diretamente relacionado ao número de vértices, essas técnicas foram consideradas desapropriadas na implementação de um ambiente tridimensional para *real-time*. A exigência de modelos *high-poly* (grande quantidade de polígonos) sobrecarrega o processamento em tempo real do ambiente virtual.

O *render to texture* é uma poderosa ferramenta porque permite a criação de mapas completos, com efeitos de iluminação e de fusão de mapas de imagem. Permite também a criação de mapas de normais que simulam detalhamento na geometria de objetos com poucos polígonos, a partir da projeção de modelos semelhantes, porém complexos, com grande quantidade de polígonos, portanto pesados para o sistema *real-time*.

O *bitmap* gerado, a partir do *render to texture*, como mapa completo – com efeitos de iluminação e de texturas multi-imagem – é muito utilizado como mapa de *diffuse* em cenários de jogos 3D. Concluídos os testes, confirma-se que tem a mesma função de agregar valores em ambientes tridimensionais para *web*: melhorando a aparência e a performance do ambiente em tempo real. Da mesma forma, o *bump map*, muito aplicado em cenários de *games* 3D, comprovou a mesma importância e funcionalidade para ambientes 3D interativos na *web*.

O *normal map*, técnica recente para a criação de personagens e cenários para *games* 3D, estabeleceu-se como uma técnica fundamental, junto às técnicas de redução geométrica, para otimização de personagens e ambientes 3D sem perda de qualidade visual. Em alguns casos, como no protótipo desenvolvido para esta dissertação, percebe-se até um aumento de qualidade em relação ao modelo original de alta resolução porque suaviza mais a superfície do modelo.

A importância do uso de *bitmaps* com as técnicas apresentadas foi totalmente comprovada, mostrando otimização no processamento *real-time* (com aumento da taxa de quadros por segundos(FPS)), aumento da qualidade visual e diminuição no peso dos

arquivos VRML. O arquivo VRML original, criado a partir de um ambiente modelado e texturizado sem aplicação das técnicas de otimização, possui 6,31 MB e é 85,5% maior do que o arquivo VRML, com 908 KB, finalizado com as técnicas de otimização. Após compressão no programa *VRML Pad*, os arquivos ficam, respectivamente, com 1,60 MB e 236 KB.

A escolha das técnicas de otimização, na etapa de criação de modelos e ambientes tridimensionais para visualização em tempo real, deve resultar da consideração de três fatores: tempo de produção, tempo de processamento e qualidade visual. Percebe-se que quanto maior o tempo de produção, maior será a qualidade visual e menor será o tempo de processamento *real-time*. O fator tempo de produção vem criar uma relação mais harmônica entre qualidade visual e rapidez no processamento em tempo real.

Uma difusão maior das técnicas de otimização de modelos e ambientes tridimensionais para processamento em tempo real permitirá um melhor aproveitamento das vantagens e soluções propiciadas pela linguagem VRML ou X3D, bem como um aumento na qualidade e na quantidade de sites e aplicações 3D interativas na web.

Bibliografia

[1] SOFTEX. *Tecnologias de Visualização na Indústria de Jogos Digitais: Potencial Econômico e Tecnológico para a Indústria Brasileira de Software*. Campinas : Softex, 2005.

[2] DOMINGUES, R. G. *Projeto de um Framework para Auxilio no Desenvolvimento de Aplicações com Gráficos 3D e Animação*. São Carlos : UFSCar, 2003.

[3] ANÔNIMO, *VRML*. Disponível em: http://pt.wikipedia.org/wiki/VRML - Último acesso em 26/08/06

[4] CUNHA, G. G. *VRML97, Virtual Reality Modeling Language*. Disponível em: http://www.lamce.ufrj.br/grva/vrml/_index.htm - Último acesso em 22/07/06

[5] BLAXXUN, *Blaxxun Contact*. Disponível em: http://www.blaxxun.com - Último acesso em 10/06/06

[6] PARALLELGRAPHICS, *Cortona VRML Cliente*.
Disponível em: http://www.parallelgraphics.com - Último acesso em 10/06/06

[7] SGI, *Cosmo Player*. Disponível em: http://www.sgi.com/products/evaluation/6.5_cosmoplayer_2.1.5/ - Último acesso em 8/06/06

[8] OCTAGA, *Octaga Player*. Disponível em: http://www.octaga.com - Último acesso em 20/06/06

[9] WEB3D Consortium, *What is X3D?* Disponível em: http://www.web3d.org/about/overview - Último acesso em 15/06/06

[10] SUMMERS, D. *Texturing: Concepts and Techniques*. Charles River Media, Inc. Hingham, Massachusetts, 2004.

[11] AUTODESK, *3ds max*. Disponível em: http://usa.autodesk.com/adsk/servlet/index?id=5659302&siteID=123112 - Último acesso em 5/06/06.

[12] NEWTEK, *Lightwave*. Disponível em:http://www.newtek.com/lightwave - Último acesso em 18/06/06.

[13] AUTODESK, *Maya*. Disponível em: http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=7635018 - Último acesso em 12/06/06.

[14] MENTAL IMAGES, *Mental ray*. Disponível em: http://www.mentalimages.com/2_1_0_mentalray/index.html - Último acesso em 20/06/06.

[15] CHAOS GROUP, *V-Ray*. Disponível em: http://www.chaosgroup.com/software/vray - Último acesso em 20/06/06.

[16] WORLEY LABS, *G2*. Disponível em: http://www.worley.com/G2/g2_main.html - Último acesso em 20/06/06.

[17] ADOBE, *Flash*. Disponível em: http://www.adobe.com/products/flash/flashpro - Último acesso em 15/06/06.

[18] ELECTRIC RAIN, *Swift 3D*. Disponível em: http://www.erain.com/products - Último acesso em 27/06/06.

[19] MÖLLER, T., HAINES, E*. Real Time Rendering*, AK Peters, 1999. Disponível no web site do livro: http://www.realtimerendering.com - Último acesso em 12/06/06.

[20] KERLOW, I. V. *The Art of 3-D Computer Animation and Imaging*. Milton, Australia: John Wiley & Sons, 2000.

[21] SILVEIRA NETO, W. D., MELO, A. K. "*Técnicas de Animação em Ambientes Tridimensionais"*. UFSC, Florianópolis, SC. Edição 5: Ano IV, Volume IV, Número I, Maio de 2005. Disponível em: http://www.inf.ufsc.br/resi/edicao5.html - Último acesso em 12/06/06.

[22] *3DS MAX 7: GUIA AUTORIZADO DISCREET*. São Paulo: Ed. Campus, 2005.

[23] BIRN, J. *Digital Lighting & Rendering*. New Riders Publishing, Berkeley, California, 2000.

[24] OMERNICK, M. *Creating the Art of the Game*. New Riders Publishing, 2004

[25] FRANSON, D. *The Dark Side of Game Texturing*. Premier Press. 2004.

[26] BINOTTO, A. P. D. *Visualização em Tempo Real de Dados Volumétricos Dinâmicos usando Hardware Gráfico* – M.Sc. dissertação, UFRGS, Porto Alegre, 2003.

[27] BATAGELO, H. C. *Manipulação Direta em Geometria Deformada em Placas Gráficas*. Relatório Técnico. UNICAMP, DCA, 2005.

[28] PABST, T. *High-tech and Vertex Juggling - nVidia's new GeForce 3 GPU*. Toms Hardware Guide, February 2001.

[29] HANRAHAN, P., LAWSON, J. "*A Language for Shading and Lighting Calculations"*. *ACM SIGGRAPH Computer Graphics*, 24(4), 1990.

[30] SHREINER, D., WOO, M., NEIDER, J., DAVIS, T. *OpenGL Programming Guide: The Official Guide to Learning OpenGL*. Addison-Wesley Pub Co, 4th edition, 2003.

[31] MICROSOFT CORPORATION. *DirectX 9.0 Programmer's Reference*. October, 2004.

[32] ANÔNIMO. *Normal Mapping*. Disponível em: http://en.wikipedia.org/wiki/Normal_map - Último acesso em 25/05/06.

[33] GONZÁLEZ, C*.,* GUMBAU, J., CHOVER, M. "*Generación por Hardware de Mapas de Normales"*. CEIG, 2005. Disponível em: http://www.gametools.org/archives/publications/UJI_CEIG2005_2.pdf - Último acesso em 23/05/06.

[34] BLINN, J.F. "*Simulation of Wrinkled Surfaces*". In: Computer Graphics (Proceedings of SIGGRAPH 78), vol. 12, pp. 286–292, 1978.

[35] MAX, N.L. "*Horizon Mapping: Shadows for Bump-mapped Surfaces*". The Visual Computer 4(2), 109–117, 1988.

[36] MAX, N.L. “*Shadows for Bump-mapped Surfaces*”. Advanced Computer Graphics (Proceedings of Computer Graphics Tokyo ’86), pp. 145–156 (1986)

[37] SLOAN, P., COHEN, M.F. “*Hardware Accelerated Horizon Mapping*”. Rendering Techniques 2000: 11th Eurographics Workshop on Rendering, pp. 281–286, 2000.

[38] WANG, L., WANG, X., TONG, X., et al. “*View-dependent displacement mapping*”. ACM Transactions on Graphics 22(3), 334–339, 2003.

[39] WANG, X., TONG, X., LIN, S., , et al. “*Generalized Displacement Maps*”. Rendering Techniques 2004: 15th Eurographics Workshop on Rendering, pp. 227–234, 2004.

[40] OLIVEIRA, M.M., BISHOP, G., MCALLISTER, D. “*Relief Texture Mapping*”. Proceedings of ACM SIGGRAPH 2000, Computer Graphics Proceedings, Annual Conference Series, pp. 359–368, 2000.

[41] POLICARPO, F., OLIVEIRA, M. M., COMBA, J. L. D. “*Real-time Relief Mapping on Arbitrary Polygonal Surfaces*”. SI3D ’05: Proceedings of the 2005 symposium on Interactive 3D graphics and games, pp. 155–162. ACM Press, New York, NY, USA, 2005.

[42] PARILOV, E., ROSENBERG, I., ZORIN, D. “*Real-time Rendering of Normal Maps with Discontinuities*”. CIMS Technical Report, New York University, 2005

[43] ATI. *Normal Mapper Tool*, 2002. Disponível em: http://www.ati.com/developer/tools.html - Último acesso em 23/05/06.

[44] NVIDIA. *nVidia Melody User Guide*, 2004. Disponível em: http://developer.nvidia.com/object/melody_home.html - Último acesso em 23/05/06.

[45] Especificação Oficial ISO/IEC 14772 do VRML97. Disponível em: http://www.lamce.ufrj.br/grva/vrml/_index.htm#especificacao - Último acesso em 20/06/06.

[46] CORTONA VRML CLIENT, *MipMap*. Disponível em: http://www.parallelgraphics.com/developer/products/cortona/extensions/mipmap - Último acesso em 26/05/06.

[47] CREATIVE DIMENSION SOFTWARE*, 3DSOM Pro*. Disponível em: http://www.3dsom.com - Último acesso em 12/05/06

[48] PARALLELGRAPHICS, *Showroom*. Disponível em: http://www.parallelgraphics.com/products/showroom - Último acesso em 14/06/06

[49] MUCHMORE, M. W. “*Browse the Web in Three Dimensions*”. Disponível em: http://www.extremetech.com/article2/0,1697,1974390,00.asp - Último acesso em 17/06/06.