Pontifícia Universidade Católica
do Rio de Janeiro

Esteban Walter Gonzalez Clua

Impostores com Relevo

Tese de Doutorado

Tese apresentada ao Programa de Pós-graduação em Informática da PUC-Rio como requisito parcial para obtenção do título de Doutor em Informática.

Orientador: Bruno Feijó
Co-orientador: Marcelo Dreux

Rio de Janeiro, Abril de 2004

Esteban Walter Gonzalez Clua

Impostores com Relevo

Tese de Doutorado

Tese apresentada como requisito parcial para obtenção do título de Doutor pelo Programa de Pós-Graduação em Informática da PUC-Rio.

Orientador: Bruno Feijó
Co-orientador: Marcelo Dreux

Rio de Janeiro, abril de 2004

Esteban Walter Gonzalez Clua

Impostores com Relevo

Tese apresentada como requisito parcial para obtenção do título de Doutor pelo Programa de Pós-Graduação em Informática da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Bruno Feijó
Orientador
PUC-Rio

Marcelo Dreux
Co-orientador
PUC-Rio

Waldemar Celes
PUC-Rio

Manuel Menezes Oliveira
UFRGS

Hélio Lopes
PUC-Rio

Luiz Eduardo Sauerbronn
UFRJ

Edilberto Strauss
UFRJ

Sidnei Paciornik
Puc-Rio

José Eugênio Leal
Coordenador(a) Setorial do Centro Técnico Científico - PUC-Rio

Rio de Janeiro, 2 de abril de 2004

Esteban Walter Gonzalez Clua

Graduou-se em Ciência da Computação pelo IME-USP em 1996. É mestre em informática na área de Computação Gráfica pela PUC-Rio. Atualmente é pesquisador do ICAD-Igames, Puc-Rio e trabalha no desenvolvimento de games e ferramentas para a área.

Ficha Catalográfica

Clua, Esteban Walter Gonzalez

Impostores com Relevo / Esteban Walter Gonzalez Clua; orientador: Bruno Feijó, co-orientador: Marcelo Dreux. – Rio de Janeiro: PUC, Departamento de Informática, 2004. v., 127 f.: il. ; 29,7 cm

1. Tese (doutorado) – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática. Inclui referências bibliográficas.

Modelagem baseada em imagens, Rendering baseado em imagens, jogos para computador, texturas com relevo, impostores, impostores com relevo, sistemas distribuídos para visualização, pipeline gráfico programável.

Para meus pais, meus irmãos e meus orientadores.

Agradecimentos

Se cheguei a este ponto, é porque consegui acabar a tese! E ter conseguido terminá-la se deve a muitas pessoas, para quem palavras são pouco para retribuir.

Agradeço aos meus pais, que sempre foram modelo para mim, até no aspecto acadêmico. Posso dizer que cresci dentro de um centro de pesquisas... De igual maneira, agradeço aos meus irmãos, que sempre foram grandes amigos para mim e tiveram um papel importante por ter me apaixonado pela área de games.

Agradeço aos Bruno Feijó e Marcelo Dreux, a quem considero como verdadeiros amigos, antes de orientadores. OBRIGADO MESMO, do fundo do coração!

Agradeço aos que me ajudaram na maior das boas vontades a poder implementar muitas coisas: Francisco Fonseca, Fábio Policarpo, César Pozzer, Lauro Kozovitz, Gilliard Lopes, Lucas Machado... O que seria de mim sem vocês?... Também devo muito a alguns professores e pesquisadores que em algum momento me ajudaram. Se não fossem algumas conversas e dicas, não teria tido as idéias deste trabalho: Luiz Velho, Waldemar Celes, Manuel Oliveira, Noemi Rodrigues, Maria das Graças, Luiza Novaes, Marcelo Gattass...

Finalmente, agradeço a todos os que me ajudaram e orientaram, de alguma forma, a encontrar minha paixão pela computação gráfica, pelo entretenimento digital (games...) e obviamente ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), pela ajuda financeira; Agradeço também à FINEP pela concretização do VisionLab, ao qual este trabalho está vinculado.

Dedico este trabalho a todos vocês, a todos os membros da tripulação do ICAD-IGames e a muitos outros cuja amizade considero o que de melhor tenho.

Resumo

Clua, Esteban Walter Gonzalez. **Impostores com relevo.** Rio de Janeiro, 2004. 127 p. Tese de Doutorado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

O presente trabalho introduz o conceito de impostores com relevo: uma maneira eficiente para representar objetos por imagens em sistemas que requerem visualização em tempo real, especialmente jogos 3D e ambientes de realidade virtual. Para tanto, mesclam-se métodos tipicamente pertencentes à área de renderização baseada em imagens com métodos tradicionais de visualização baseada em geometria. A técnica requer do usuário apenas a modelagem geométrica da entidade a ser representada. Posteriormente o sistema sintetiza texturas com relevo, dinamicamente atualizadas quando necessário, e as visualiza utilizando o método de mapeamento de texturas com relevo. Esta abordagem permite inserir modelos complexos, tanto pela sua natureza geométrica, como pelo seu processo de visualização, no *pipeline* gráfico em tempo real. Além disso, os impostores com relevo procuram aproveitar o tempo ocioso ou recursos paralelos disponíveis no processador, de forma a balancear a carga de processamento de visualização entre CPU/GPU. Estes impostores também tornam possível a representação de qualquer tipo de objeto geométrico através de mapeamento de texturas com relevo.

Palavras-chave

Modelagem baseada em imagens, Rendering baseado em imagens, jogos para computador, texturas com relevo, impostores, impostores com relevo, sistemas distribuídos para visualização, pipeline gráfico programável.

Abstract

Clua, Esteban Walter Gonzalez. **Relief Impostors.** Rio de Janeiro, 2004. 127 p. Tese de Doutorado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The present work introduces the concept of relief impostors: an efficient manner of representing objects by images in systems that require real time rendering, such as 3D games and virtual reality environments. For this purpose, typical methods of image-based rendering are mixed with traditional geometry based rendering methods. This technique only requires from the user the geometric modeling of the entity to be represented. After this, the system synthesizes relief textures, dynamically refreshed when necessary, and renders them using the method of relief texture mapping. This approach allows complex models to be inserted into the real time pipeline system. This complexity arise either from the geometric nature of the model or its process of visualization. Also, the relief impostors try to use the idle time or parallel resources available on the processor, in order to balance the work to be done between the CPU and GPU. Furthermore, they make possible the representation of any kind of geometric object by the relief texture mapping technique.

Keywords

Image-based modeling, image-based rendering, 3D computer games, relief textures, impostors, relief impostors, distributed visualization systems, programmable graphic pipeline.

Sumário

Lista de figuras

Lista de tabelas

I am the Architect. I created the Matrix. I've been waiting for you.

The Matrix

1 Introdução

I know because I must know. It's my purpose. It's the reason I'm here. (The Matrix)

1.1 Objetivos do trabalho

Os *hardwares* gráficos atualmente podem ser considerados como verdadeiros processadores e, em muitos casos, mais poderosos que os processadores da CPU. Algumas placas comerciais ultrapassaram a barreira dos 120 milhões de transistores (mais do que um chip Pentium IV), podem acessar até 16 GB de memória, processam 4 ou mais *pixels* simultaneamente e as placas com suporte à programação de seu *pipeline* podem possuir vários processadores programáveis (ver capítulo 6). É por esta razão que a tendência hoje é de chamar estes dispositivos de GPU's (*Graphic Processor Units*). Assim, pode-se assumir que um computador munido de uma boa placa gráfica é uma máquina paralela, embora um dos processadores seja de uso dedicado a aplicações gráficas (GPU) e o outro seja de uso genérico (CPU).

Pode-se comprovar, no entanto, que nas aplicações gráficas cada vez mais têm crescido a tendência de que a GPU assuma a maior parte do *pipeline* de visualização, deixando por outro lado que a CPU tenha um tempo ocioso (*idle time*) cada vez maior. Na realidade, o trabalho gráfico da CPU tem se tornado cada vez mais burocrático e limitado a alimentar a placa gráfica com os dados relacionados às geometria e texturas e, quando muito, a resolver alguns cálculos de otimização e *culling*. A figura 1.1 mostra o baixo consumo de tempo do processamento de uma CPU na execução de um jogo 3D, dotado de extensa modelagem geométrica. Diversas aplicações têm-se aproveitado deste tempo livre procurando, com este recurso, resolver operações mais complexas, tais como cálculos de inteligência artificial ou simulação física. Entretanto, poucos trabalhos têm sido feitos no sentido de assumir a CPU|GPU como uma máquina paralela para um mesmo *pipeline* gráfico.

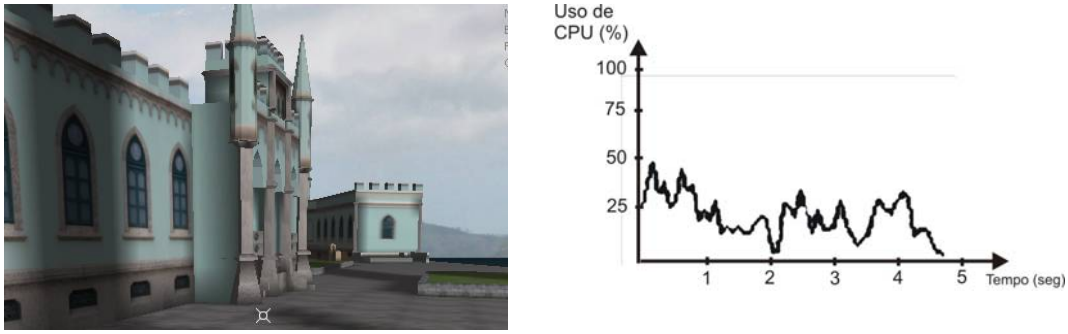


figura 1.1 – Consumo de CPU (em %) em movimentos típicos de um jogo 3D, utilizando um computador munido de GPU. O cenário utilizado possui 14.108 polígonos, sendo que 4.540 estão dentro do campo de visão.

A primeira idéia neste trabalho consiste em aproveitar o tempo ocioso da CPU para operações gráficas em tempo real, realizando operações que são inapropriadas para a GPU sem, no entanto comprometer o desempenho de visualização da aplicação. Estas operações consistem sobretudo em realizar processamento de imagens típicas da área de renderização baseada em imagens, alimentando o *pipeline* da placa gráfica com os resultados obtidos. Assim sendo, tem-se um *pipeline* gráfico duplo: um que é executado pela placa gráfica (*hardware*) e outro que é executado pela CPU (*software*). Este tipo de arquitetura por si só já corresponde, na prática, a ver um sistema com uma CPU e uma GPU como sendo uma máquina paralela. Nesta máquina paralela, a arquitetura proposta toma os cuidados apropriados para que a GPU nunca tenha outra latência, a não ser aquelas geradas pela sua própria limitação de processamento.

Neste sentido, um dos objetivos deste trabalho consiste em apontar que os métodos de renderização baseada em imagens podem ser utilizados para reutilização de resultados obtidos de uma visualização previamente realizada pela CPU ou pela GPU. Isto é feito através de uma extensão do conceito de impostores, primeiramente definido em (Schaufler, 1997) e que consiste num *sprite*, gerado a partir de uma primitiva geométrica, com suporte a profundidade. O nome “impostor com relevo” indica que este tipo de primitiva é um impostor, cuja profundidade serve para se realizar uma operação de *warping* sobre a textura, utilizando o método de mapeamento de texturas com relevo. O método básico para texturas com relevo foi primeiramente apresentado em (Oliveira, 2000a). A figura 1.2 ilustra um objeto modelado como impostor com relevo.

figura 1.2 – Exemplo de um objeto modelado através de impostores com relevo.

Uma vez que as operações de renderização baseada em imagem não são apropriadas para serem processadas pela GPU, o método de *pipeline* híbrido proposto sugere que o processamento da função plenóptica seja na CPU, deixando o *hardware* gráfico dedicado às operações de visualização dos dados geométricos. Esporadicamente o sistema permite que haja uma transferência dos resultados entre a memória da CPU para a memória da GPU. Isto é realizado de forma inteligente para evitar que o gargalo da visualização esteja ou no processamento da aplicação de renderização baseada em imagens ou no processo de tráfego de dados CPU/ GPU.

Um outro objetivo deste trabalho consiste em realizar abordagens paralelas para resolver o mapeamento de texturas com relevo. Para tanto, é explorado o recurso de programação por *threads*, permitindo que haja processos dedicados a certas tarefas, dando-lhes diversos graus de prioridade. São apresentadas diversas abordagens envolvendo paralelismo entre CPU-GPU e paralelismo de recursos de CPU. De forma a comprovar que este paralelismo é um recurso acessível e barato, toda a implementação é realizada em processadores com *hyper-threading*, uma vez que este recurso está se tornando barato e comum no mercado de micro-computadores.

Impostores com relevo, a contribuição inédita desta tese, possibilita que a técnica inventada por Oliveira (2000 a) modele qualquer tipo de objeto geométrico – um resultado de grande impacto em aplicações de tempo real.

1.2 Conceitos Envolvidos

Os conceitos que são tratados nesta dissertação referem-se a:

a) Renderização baseada em imagens - área da computação gráfica que procura criar objetos e cenários utilizando apenas imagens;

b) *Pipeline* de visualização em tempo real, dando-se especial enfoque a aplicações de jogos 3D;

c) Aceleração gráfica por *hardware*, bem como programação para GPU´s;

d) Processamento paralelo e distribuído para o *pipeline* de visualização.

1.3 Estrutura da dissertação

O capítulo 2 introduz primeiramente o conceito da função plenóptica, para em seguida resumir os principais trabalhos e pesquisas na área de renderização baseada em imagens. Para descrevê-los, cria-se uma classificação separada por aplicações que envolvem modelagem de cenários, modelagem de panoramas, modelagem de objetos e métodos de aceleração e otimização para visualização de estruturas geométricas. A seguir, apresenta-se com detalhes o conceito de 3D *image warping*, conceito este que é utilizado na elaboração dos impostores com relevo. O conceito de *view morphing* também é apresentado com detalhes porque, apesar de não ter sido utilizado no desenvolvimento desta pesquisa, aponta-se que este conceito também pode ser explorado para chegar a resultados semelhantes.

A proposta de modelagem deste trabalho se enquadra dentro da categoria de modelagem de objetos por imagens. Desta forma, o capítulo 3 aprofunda-se nas diversas técnicas de renderização baseada em imagens que solucionam esta classe de problemas. O mapeamento de texturas com relevo é detalhadamente exposto e discutido ainda neste capítulo, uma vez que o conceito de impostores com relevo é uma extensão deste trabalho.

O capítulo 4 propõem o conceito de Impostores com relevo, expondo como é adaptado partindo das texturas com relevo.

Os impostores com relevo são texturas com relevo dinamicamente atualizáveis: quando uma delas se torna obsoleta, gera-se, através de um processo paralelo, uma nova textura para substituí-la. O capítulo 5 apresenta um critério capaz de indicar quando esta troca deve ocorrer. Neste capítulo também se apresenta uma possível otimização no processamento do mapeamento de relevo, utilizando a equação de Schaufler.

O capítulo 6, denominado de “A GPU” apresenta o papel do *hardware* gráfico no processo de visualização dos impostores com relevo, que basicamente consiste em tratar a iluminação para cada *pixel*, utilizando o mapa de normais da textura. Nesta parte da dissertação também se discute a conveniência de se utilizar a aceleração por *hardware* para aplicações de renderização baseada em imagens. Finalmente, ainda neste capítulo, apresenta-se uma implementação do mapeamento de texturas com relevo, feita totalmente na GPU. Esta abordagem é posteriormente comparada com o desempenho da proposta deste trabalho.

No capítulo 7, após uma breve introdução aos conceitos de sistemas paralelos e distribuídos, apresentam-se as propostas de paralelismo que são utilizadas para a implementação do *framework*.

O capítulo 8 introduz detalhadamente este *framework,* bem como todas as etapas de implementação realizadas. Ainda neste capítulo podem-se encontrar os resultados obtidos pelas diversas propostas da tese, especialmente em relação aos impostores com relevo.

Finalmente, o capítulo 9, após breves conclusões, discute sobre diversas possibilidades para se estender esta pesquisa.

1.4 Contribuições Alcançadas

A principal contribuição original desta tese é a criação do conceito de Impostores com Relevo, como uma extensão do mapeamento de texturas com relevo de Oliveira (2000). Este novo conceito aumenta a capacidade de representação do mapeamento de texturas com relevo, tornando-o capaz de modelar qualquer tipo de objeto geométrico.

A paralelização da renderização entre GPU e CPU é uma outra contribuição original, porém dividida com Fonseca (2004) que realiza suas pesquisas em parceria com o presente autor. Uma contribuição do presente trabalho está no processamento paralelo para renderização do impostor, enquanto que em Fonseca (2004) o impostor já vem pronto.

Duas outras contribuições originais são as seguintes:

- Processo de otimização de *warping* em texturas com relevo, através do critério de Schaufler (1995);

- Proposta de taxonomia para os métodos de renderização baseada em imagens e suas principais técnicas.

- Uma heurística capaz de apontar quando uma textura com relevo não é mais válida, e portanto convém ser substituída por uma nova.

2 Renderização Baseada em Imagens

What is real? How do you define real? If you're talking about what you can feel, what you can smell, what you can taste and see, then real is simply electrical signals interpreted by your brain. (The Matrix)

2.1 Introdução

O trabalho apresentado nesta pesquisa pertence, em parte, à área da computação gráfica denominada de renderização baseada em imagens. Esta área tem como objetivo representar um cenário completo ou objetos que façam parte dele a partir de imagens, ao invés de elementos geométricos. Neste capítulo descreve-se inicialmente o conceito de função plenóptica, juntamente com os trabalhos mais significativos na área. Em seguida discute-se, resumidamente, um sistema para classificação e comparação dos métodos existentes. Finalmente, expõe-se com detalhes o *3D Image Warping*, que é fundamental para desenvolver o conceito dos impostores com relevo.

2.2 A Função Plenóptica

A área de renderização baseada em imagens surgiu a partir de uma combinação entre as técnicas de Computação Gráfica 3D e Visão Computacional, tendo em vista adquirir resultados foto-realistas e ao mesmo tempo rápidos na visualização de cenas. Atualmente, com o avanço do poder computacional das placas gráficas, esta técnica ganhou grande importância em sistemas de visualização de tempo real, tais como jogos, aplicações de realidade virtual e simulações.

Para uma definição mais formal do que vem a ser o *ibr* – abreviatura comumente utilizada para referir-se a renderização baseada em imagens, extraída da expressão em inglês *image-based rendering* -, deve-se recorrer ao conceito de

função plenóptica[1], definida inicialmente por Adelson (1991). Esta é uma função parametrizada que descreve tudo o que é possível de ser visto a partir de qualquer posição e orientação do espaço, a qualquer momento e em qualquer comprimento de onda da luz. Em (McMillan, 1995) a função é definida como:

$$cor = \mathrm{P}\,(\theta, \phi, \lambda, V_x, V_y, V_z, t) \qquad (2\text{-}1)$$

onde (V_x, V_y, V_z) é a posição do observador, θ e ϕ são os ângulos de azimute e elevação da direção do vetor do observador (ver figura 2.1) e λ é o comprimento da onde de luz capaz de ser vista. No caso de uma cena dinâmica, a variável t descreve o tempo.

Pode-se entender a função plenóptica como sendo uma função capaz de descrever todas as possíveis vistas de uma determinada cena. Diz-se que esta função está completa se, para um ponto coberto pela função, toda a abóboda que está à sua volta pode ser vista; diz-se incompleta quando apenas uma parte desta abóbada é visível. Vários autores afirmam que todos os algoritmos de *ibr* se resumem a uma abordagem particular da função plenóptica.

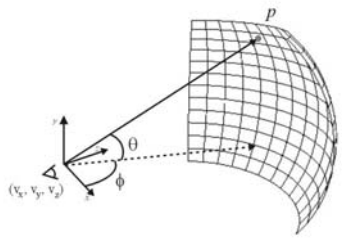


figura 2.1 – Qualquer ponto visível para qualquer posição de um observador pode ser descrito pela função plenóptica.

Desta maneira, um possível enunciado para a maioria dos problemas relacionados a *ibr* pode ser descrito da seguinte forma: "dado um conjunto de amostras (completas ou incompletas) da função plenóptica, o objetivo da solução proposta consiste em gerar uma representação contínua desta função" (McMillan, 1995).

A seguir descrevem-se os trabalhos de maior destaque na área de *ibr*,

[1] *plenus* = completa e *optic* = visão

classificados de acordo com o que se propõem a representar: cenários completos (tudo o que é visível é modelado por imagens), panoramas (objetos que estão em volta e infinitamente afastados do observador são imagens), transformação de elementos da cena em imagens com o objetivo de realizar *culling* ou otimização da visualização e objetos específicos (apenas alguns elementos da cena são descritos por imagens).

2.2.1 Modelagem de cenários completos

Os trabalhos mais próximos da definição de função plenóptica e que procuram representar um cenário completo são o *Ligth Field* (Levoy, 1996) e o *Lumigraph* (Gortler, 1996), embora antes destes Lippman (1980) tenha desenvolvido o *MovieMap*, que pode ser considerado como o precursor de todas as implementações de *ibr* e da definição da função plenóptica. O *Ligth Field* e o *Lumigraph* poderiam também ser chamados de técnicas "força bruta", no sentido de que criam um banco de dados com todos os possíveis raios de visão existentes dentro de uma região. Para tanto, estas técnicas sugerem descrever cada raio de luz através das duas coordenadas de interseção do mesmo com dois planos de suporte distintos. Desta maneira, é possível descrever um sub-conjunto da função plenóptica através de dois pares de coordenadas bidimensionais. Deve-se ressaltar que apenas 2 planos não são suficientes para representar todos os raios do interior de uma cena, sendo necessário portanto definir vários pares de planos distintos e não paralelos. Não é difícil concluir que ambos requerem uma densidade de amostras enorme para poder haver um mínimo de continuidade nos movimentos.

O *Unstructural Lumigraph Rendering* (Buheler, 2001) tem uma idéia semelhante aos dois trabalhos citados, porém não precisa de uma seqüência de imagens de entrada com uma ordem pré-estabelecida, como ocorre no *Lumigraph* ou no *Light Field*, embora ambos possuam soluções de pré-processamento que solucionam, em parte, este problema. Isto possibilita que a modelagem da cena seja mais simples, bastando capturar imagens do local através de aparelhos convencionais. Entretanto, o sistema requer que seja feito uma estimativa da posição da câmera, bem como uma aproximação geométrica da cena. Quanto melhor é esta aproximação, menos imagens são necessárias. O algoritmo baseia-se no conceito de *camera blending field,* que consiste numa tabela de distribuição de

pesos para cada ponto visto pela câmera, indicando qual é a imagem que deve ser usada para gerá-lo e quanto é a sua contribuição (um mesmo ponto é descrito por várias imagens, desde que a soma de todos os pesos seja igual a 1). Para construir esta distribuição de pesos, realiza-se o cálculo das imagens que possuem o menor ângulo formado entre o seu centro de projeção e o ponto que está sendo observado com a posição real da câmera com o mesmo ponto. Embora a aproximação da geometria das imagens possa ser um processo complexo, e não ser conveniente para representar ambientes extensos, o algoritmo demonstra ter bom comportamento quando se utilizam aproximações grosseiras da geometria, como por exemplo criando uma malha de polígonos coincidente com o plano de projeção da câmera e com alguns poucos vértices deslocados. À aproximação geométrica é possível somar-se a geometria de outros objetos, possibilitando que haja uma mistura de *ibr* com modelagem tradicional.

Procurar extrair informações geométricas da cena tem sido uma abordagem muito utilizada para reduzir o número de imagens necessárias para modelar a função plenóptica. Em (Debvec, 1996) propõe-se que a partir de um conjunto de imagens arquitetônicas de um ambiente real, seja gerado um modelo geométrico aproximado e a seguir se aplique as imagens sobre os modelos, com técnicas tradicionais de projeção de texturas. O sistema auxilia o usuário, detectando linhas horizontais e verticais para posteriormente indicar onde colocar primitivas geométricas, cujos vértices possuirão correspondência com coordenadas de texturas referentes às imagens fonte. Esta técnica introduz o conceito de view-dependent texture mapping e prevê ainda que detalhes da modelagem sejam estimados utilizando métodos de visão estéreo para calcular um mapa de profundidade de cada imagem original.

Apesar desta solução possuir um grau de liberdade alto (5 dimensões: θ, ϕ, V_x, V_y e V_z) o espaço de navegação é pequeno e restrito. Quanto mais se deseje ampliar esta região, maior será o trabalho de modelagem, tornando-se em alguns casos impraticável. Além disso, sua ferramenta de autoria prevê que os objetos das imagens tenham um estilo arquitetônico (linhas retas), o que limita a utilização do sistema para cenas particulares. Borshukov (1997) estendeu o sistema para algumas primitivas curvas.

2.2.2 Modelagem de panoramas

Uma forma mais simples e mais genérica para representar cenas com um suporte geométrico consiste na técnica de panoramas, inicialmente proposto por Chen (1995). Esta técnica, que se tornou conhecida devido à sua aplicação comercial *Quicktime VR*, utiliza a idéia básica de *enviroment-maps*[*].

O *QuickTime VR* utiliza um panorama cilíndrico como suporte geométrico, tendo em vista a facilidade no processo de autoria (gerar a biblioteca de imagens que serão necessárias). No espaço da função plenóptica este método é classificado como bidimensional, pois o movimento da câmera não é contínuo e está restrito a um conjunto finito de pontos do espaço, sendo as únicas variáveis que podem variar continuamente θ e ϕ - ângulos de azimute e elevação do observador. No caso do panorama ser cilíndrico o ângulo de elevação está restrito. Isto é resolvido criando-se suportes cúbicos ou esféricos, do panorama.

Esta técnica vem sendo amplamente utilizada em jogos, e é conhecida também como *skyboxes*. Neste caso, ao invés de um suporte cilíndrico utiliza-se um cubo, onde o observador está sempre no seu centro. Através deste método, representam-se objetos afastados e sem interação com o jogador, como o céu, paisagens de fundo, estrelas, etc.

Na tentativa de se poder realizar mudanças contínuas entre diversas imagens panorâmicas, um dos trabalhos de mais destaque é o proposto em (Aliaga, 2001), chamado de *Plenoptic Stitching*.

O método consiste inicialmente em capturar imagens de um ambiente com um pequeno carro rádio-controlado que percorre caminhos dentro de um determinado ambiente. Para o bom funcionamento do sistema, de acordo com os autores, são necessárias em torno de 30 imagens por metro, utilizando uma câmera omnidirecional (capaz de adquirir fotos de 360^o de latitude por 180^o de longitude). Estes caminhos percorridos devem cruzar-se, de maneira a formar uma malha irregular, sendo cada célula da malha denominada de ciclo de imagem. Pode-se entender estes ciclos como diversos panoramas justapostos, não havendo necessidade de uma forma pré-definida para cada um. Dada a grande quantidade de imagens necessárias, o sistema foi previsto apenas para ambientes internos.

---

[*] Imagens mapeadas sobre uma esfera que engloba completamente o objeto (Blinn, 76)

Feita a captura dos panoramas, o sistema calcula uma aproximação da posição da câmera para cada imagem. Sugere-se para isto o algoritmo chamado de *bacon-based*, capaz de estimar o movimento feito pela câmera partindo de uma imagem inicial. De maneira a realizar esta estimativa com mais precisão, os autores sugerem que se coloquem duas lâmpadas no ambiente, indicando-se suas posições na imagem inicial.

A interpolação de imagens será necessária sempre que o observador estiver no interior de uma célula. Neste caso, a composição é feita utilizando-se colunas de pixels das imagens que estiverem nas bordas dos ciclos de imagens mais próximos da câmera.

Para que o sistema possa ser tempo real, é necessário haver uma fase de pré-processamento, onde sobretudo se cria uma estrutura de dados para que se possa ter rápido acesso às linhas radiais e às suas funções de mapeamento requeridas na reconstrução.

2.2.3 Aplicações de *ibr* para *cache* e *culling*

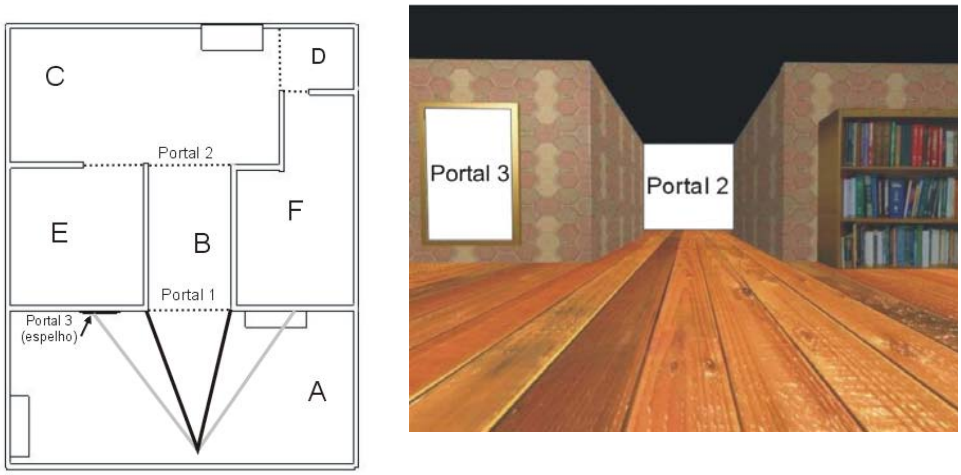
Algoritmos de *ibr* também podem ser utilizados para otimização do processo de visualização de uma cena. Enquadram-se dentro desta área, os algoritmos de portais e impostores.

Os impostores são discutidos com mais detalhes na seção 3.3, mas visam sobretudo minimizar o número de cálculos de visualização de um objeto, reaproveitando resultados já obtidos anteriormente. Neste sentido é que funcionam como um *cache*.

Os portais (Airey 1990, Teller 1991, Eberly 2000) possuem um conceito que em parte é semelhante ao dos impostores, porém para regiões visíveis de uma cena ao invés de objetos individuais. São comumente utilizados como um método eficiente para *culling*[*] de ambientes que podem ser divididos em células. Este método procura otimizar a visualização, fazendo com que apenas as células visíveis num determinado instante sejam processadas. Aliaga (1997) propõem que esta idéia seja usada para substituir parte da geometria de uma cena por um

[*] Processo de eliminação de polígonos não necessários para a visualização da imagem correspondente.

polígono especial, que é o portal, onde se mapeia a imagem resultante da visualização de uma câmera posicionada no mesmo local do observador inicial, mas com um campo de visão menor que o original e limitado pelo próprio polígono corretamente recortado (figura 2.2).



(a) (b)

figura 2.2 – Em (a) pode-se ver um mapa de um ambiente fechado. Cada célula está indicada por uma letra maiúscula e os portais estão representados com uma linha tracejada. Em (b) pode-se ver o resultado parcial da visualização da cena: o portal 1 já está sendo mostrado e os portais 2 e 3 ainda estão em branco.

O algoritmo de portais inicia-se dividindo a cena em células convexas, o que garante a propriedade de que um observador posicionado em qualquer ponto do interior da célula pode ver qualquer outra parte da mesma (não se está considerando a existência de objetos oclusivos não pertencentes à estrutura). Estando-se dentro de uma célula, apenas se podem ver outras através dos polígonos especiais, que são os portais (figura 2.2). Assim, diz-se que todas as células estão separadas por polígonos normais (o que impede que o observador veja a célula adjacente) ou por portais. Para a representação de células não convexas, criam-se paredes invisíveis, onde os portais correspondem a estas paredes.

O *pipeline* de renderização que utiliza os portais deve garantir que os polígonos sejam renderizados na ordem de traz para frente, de maneira a garantir a ordem correta de polígonos visíveis. A visualização pode ser resumida pelo seguinte algoritmo:

Fazer o Clipping e Culling da célula onde se encontra o observador
Visualizar a cena utilizando o observador na sua posição original
Se um polígono da cena é um portal então
Recalcular o campo de visão do observador
Visualizar a sala que está sendo vista pelo portal
Chamar recursivamente o algoritmo de visualização

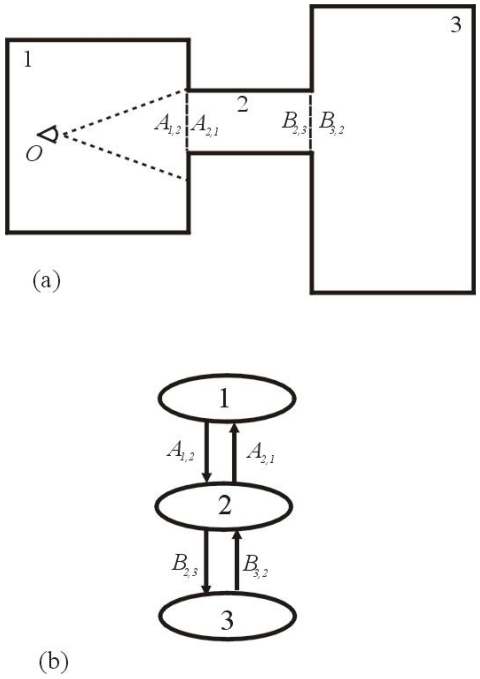


figura 2.3 - (a) Exemplo de um conjunto de células e seus respectivos portais bidirecionais. (b) grafo direcionado que representa a estrutura do exemplo da figura.

A relação dos portais com a área de *ibr* é explorada em (Aliaga, 1997), onde se apresenta uma otimização, fazendo com que os portais correspondam a texturas pré-calculadas (figura 2.4). Quando o observador se aproxima do portal, este passa a ser tratado como geometria, utilizando-se um algoritmo de *warping* para a transição. Com esta idéia torna-se necessário calcular a visualização apenas da célula onde o observador se encontra, uma vez que as demais células serão tratadas como imagens. Se por um lado este método traz uma grande aceleração, por outro lado traz problemas relacionados à falta de sensação espacial, já que as imagens das células anexas ficam estáticas, problema semelhante ao que ocorre com os *sprites*, conforme se verá na seção 3.2. Em (Rafferty, 1998a) o autor procura solucionar este problema sugerindo fornecer a cada portal um conjunto de

imagens pré-calculadas, com vistas da célula anexa, a partir de ângulos diferentes (figura 2.4). Dependendo da posição do observador, escolhe-se a vista que lhe é mais adequada. Este método aumenta a sensação de imersão, mas ainda gera transições descontínuas ao intercalar as imagens. Neste mesmo trabalho, assim como em (Rafferty, 1998b), apresenta-se como uma possível solução para este problema a utilização de algoritmos de 3D *image warping*, baseado na equação de *warping* de McMillan, conforme será apresentado na seção 2.4. Desta maneira, além de eliminar o efeito de transição descontínua, possibilita-se que haja um número menor de imagens para cada portal. Surgem, no entanto, alguns problemas relacionados à falta de informação para algumas posições em que o observador se encontra (buracos). Em (Popescu, 1998) apresenta-se uma solução para este mesmo problema utilizando a técnica de *Layered Depth Images*, que consiste basicamente numa imagem com várias camadas de cores para cada *pixel* (Gortler 1997, Max 1995).

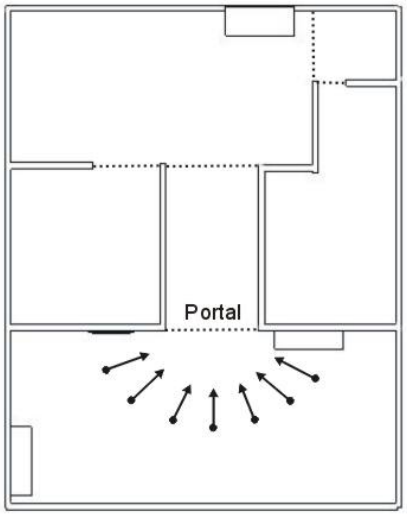


figura 2.4 – Em (Aliaga, 1997) sugere-se que várias possíveis vistas de um portal sejam pré-renderizadas e armazenadas na forma de textura. Dependendo da posição do observador, escolhe-se a imagem mais adequada deste conjunto.

Na pesquisa desta dissertação, procura-se de certa forma uma otimização semelhante à obtida nestes trabalhos de portais: reduzir cenas de natureza geométrica complexa num conjunto de imagens. Em (Rafferty, 1998a) e (Aliaga, 1999) uma cena é dividida em diversas células. Apenas a geometria da célula onde o observador se encontra é visualizada a cada frame. Para as outras células

são geradas imagens separadamente e utilizadas como texturas projetadas nas faces adjacentes à célula onde o observador se encontra, utilizando-se técnicas de *warping* de imagens para corrigir erros de paralaxe* que podem surgir.

Imagens com centros de projeção múltipla (Rademacher, 1998) podem ser vistas como uma adaptação otimizada do *Lumigraph* e *Light Field* para objetos baseados em imagens: os objetos são representados como conjuntos de imagens unidimensionais, tomadas cada uma a partir de uma direção específica.

2.2.4 Modelagem de objetos por imagens

Várias abordagens de *ibr* foram desenvolvidas com o objetivo de modelar objetos individuais, ao invés de cenários ou panoramas. Como os impostores com relevo se enquadram dentro deste tema, apesar de serem também uma forma de *cache* e *culling*, todo o capítulo 3 é dedicado a uma abordagem mais detalhada sobre o tema. Neste capítulo, depois de realizar uma descrição dos principais trabalhos existentes, introduz-se o conceito de impostores com relevo.

2.3 Componentes de classificação para os métodos de *ibr*

Como foi visto, a impraticabilidade da implementação da função plenóptica no seu caso geral faz com que os sistemas de *ibr* procurem criar amostras da mesma, a partir de um conjunto finito e discreto de dados. Isto implica em que toda abordagem possui restrições inerentes à solução proposta, apesar dos grandes progressos que se vêm obtendo recentemente. Não existe - pelo menos até agora - uma solução ótima para todos os casos, o que obriga que sejam elaborados métodos classificativos capazes de analisá-los e compará-los entre si, possibilitando uma escolha adequada para necessidades específicas. A seguir expõem-se resumidamente alguns tópicos que permitem realizar uma análise comparativa para as diversas técnicas existentes. No capítulo 3 os impostores com relevo são discutidos de acordo com estes critérios.

a) Processo de autoria: A viabilidade de um sistema de *ibr* pode ser ditada em

* Deslocamento da posição aparente de um corpo devido à mudança de ponto de vista do observador.

muitos casos pela dificuldade ou facilidade que o seu processo de autoria possui. Um sistema ideal seria aquele onde qualquer usuário poderia construir o ambiente ou os objetos, sem a necessidade de equipamentos especiais ou sofisticados. Algumas aplicações necessitam, por exemplo, a profundidade de cada pixel. Esta informação pode ser obtida em alguns casos na fase de captura da imagem, sendo necessário para tal a utilização de equipamentos especiais, como o que está descrito em (Nyland, 2001). No caso de imagens sintéticas, esta informação é trivial de ser obtida, bastando armazenar o *Z-buffer* correspondente a cada *pixel*. Em alguns casos, pode ser que a aplicação apresente soluções para extrair estes dados, como em (Oh, 2001) em que se realiza uma aproximação geométrica da imagem.

Caso o sistema consista em criar um panorama para modelar um ambiente, é importante distinguir qual a geometria do objeto que serve de suporte. Para aplicações de panoramas cilíndricos é necessária uma ferramenta de autoria, como apresentado em (Szeliski, 1996) para compor uma seqüência de fotos. Já para panoramas esféricos ou cúbicos serão necessários aparelhos de captura ou métodos de composição mais sofisticados, como os sugeridos em (Aliaga, 2001). Pode ser interessante indicar qual o método utilizado para estimar a posição de câmera, caso isto seja necessário para o algoritmo. Em (Takahashi, 2000), por exemplo, o método indica a utilização de um sistema de *Global Positioning System* (GPS), o que inviabiliza sua implementação para muitos casos.

Para o *Light Field* e o *Lumigraph*, bem como os trabalhos que se derivam destes, o critério de autoria revela-os como métodos custosos - pois é necessário um conjunto grande de amostras. No processo de aquisição das imagens pode-se imaginar que ambos os planos serão malhas com um espaçamento de tamanho fixo entre os nós. A câmera será colocada numa coordenada (s, t) e são tiradas M x N fotos, onde M e N é a resolução do outro plano (alvo da câmera). Para cada posição da câmera o seu alvo percorre cada uma das coordenadas (u, v) do outro plano. Embora, a rigor, não seja necessário nenhum equipamento especial, Gortler (1996) sugere a montagem de uma plataforma baseada em padrões de cores para determinar a posição da câmera durante a captura de amostras. Já Levoy (1996) utiliza um aparelho controlado por computador para posicionar uma câmera corretamente.

b) **Número de variáveis da função plenóptica que podem ser manipuladas continuamente e em Tempo Real (graus de liberdade):** Na prática, todos os métodos de visualização baseados em *ibr* possibilitam a manipulação de um sub-conjunto de variáveis (menor ou igual a 7) da função plenóptica. Quanto maior o número de variáveis livres, maior é o grau de liberdade permitido na interatividade do sistema implementado.

Por exemplo, restringindo-se o problema para cenas estáticas e para um comprimento de onda fixo, reduz-se a função plenóptica para 5 dimensões. McMillan (1995) usa imagens com valores de profundidade para os *pixels* de forma a reconstruir uma função de 5 dimensões. Outros exemplos de trabalhos que reduzem a função plenóptica para esta dimensão são (Chen 1993, Kang, 1996).

Para espaços sem obstrução pode-se reduzir a função para 4 dimensões, fazendo com que a cena ou o observador estejam presos a uma caixa ou a um cilindro. São exemplos desta redução os trabalhos de Levoy (1996) e Gortler (1996).

No trabalho de Shum (1999) os autores capturam imagens e fazem com que o movimento da câmera esteja preso a círculos concêntricos e paralelos ao chão, sendo um exemplo de redução da função para 3 dimensões.

Finalmente, para o caso de se fixar o observador e se permitir alterar apenas a direção do observador e o fator de *zoom*, chega-se a uma função plenóptica bidimensional. Exemplos típicos para estes casos são os panoramas esféricos e cilíndricos, onde se destacam os trabalhos de Chen (1995), Szeliski (1996) e Szeliski (1997), bem como a técnica de *skyboxes*.

c) **Continuidade e limitações impostas na mudança do valor dos parâmetros V_x, V_y, V_z da função plenóptica:** Estes parâmetros descrevem a posição onde se encontra o observador. Alterar o valor de alguma destas variáveis corresponde a caminhar com o observador numa cena. Alguns sistemas apenas permitem alterar estes valores de forma discreta, o que faz o observador dar saltos de um local para outro, tais como (Chen 1995, Szeliski 1997). A vantagem destes sistemas é que na maioria das vezes, o processo de autoria é mais fácil e não é necessário obter informações de profundidade dos *pixels*.

São exemplos de sistemas onde esta mudança é contínua (Buehlere 2001, Aliaga 2001). Para fazer esta mudança do observador ser continua, é necessário deformar e interpolar um conjunto de imagens, para pontos de vistas distintos. Estas interpolações em geral trazem alguns problemas, tais como *image fold* (mais de um *pixel* da imagem de referência são mapeados para um mesmo *pixel* da imagem resultante) e surgimento de buracos (informações que estavam oclusas na imagem inicial passam a ser necessárias na imagem gerada).

d) Densidade de amostras para situações ideais: Cada sistema tem sua base de dados ideal. Em muitos casos, para que a interatividade e imersão dentro de um ambiente sejam completas, é necessário um grande volume de amostras de imagens. Em geral, sistemas que levam em conta a profundidade do *pixel* têm a vantagem de solicitar amostras mais esparsas de imagens (Debvec, 1996). Para se fazer uma medida objetiva deste valor, pode-se criar uma relação entre o tamanho da região que se permite navegar com o número de imagens necessárias para montar este cenário. Em (Chai, 2000), por exemplo, realiza-se um estudo detalhado da quantidade mínima de imagens necessárias para métodos baseados no *Light-Field* (Levoy, 1996).

e) Extração de dados geométricos a partir das imagens: Costuma-se dividir os sistemas de *ibr* em dois conjuntos distintos: aqueles que requerem informações da geometria da cena, tendo como um extremo algoritmos denominados *View Dependent Texture Mapping* e aqueles que não requerem nenhuma informação da geometria, tendo como entrada apenas conjuntos de imagens. Alguns autores preferem inclusive separar os métodos, chamando o primeiro de modelagem baseada em imagens (Debvec, 1996) e o segundo de renderização baseada em imagens. Para os casos onde se requer informações de geometria devem ser desenvolvidas ferramentas especiais de autoria, que permitam esta extração manual ou automática. Possuem em geral a vantagem de necessitar menos imagens de entrada. Entretanto, por ser necessário realizar a estimativa da geometria da cena, o sistema depende da complexidade do local ou do objeto que está sendo criado.

f) Inclusão da variável tempo da função plenóptica: Poucos trabalhos foram

feitos até agora tendo em vista este propósito. Grande parte das aplicações de *ibr* consistem em caminhar dentro de cenários ou realizar a visualização de alguns objetos, mas não prevêem que o tempo transcorra linearmente, permitindo que diversos objetos estejam movendo-se individualmente. Isto porque a densidade de amostras se torna extremamente grande ao armazenar sub-conjuntos de imagens animadas.

Para um estudo classificatório das técnicas de *ibr* mais aprofundado veja-se (Buehlere, 2001) e (Clua, 2003b).

2.4 3D *Image Warping*

Para as técnicas que requerem uma alteração contínua em pelo menos uma das variáveis da função plenóptica, tendo-se em conta que é impraticável ter um conjunto infinitamente grande de imagens da cena ou do objeto, uma solução frequentemente adotada consiste na deformação (*warping*) das imagens fonte.

Assim, uma deformação sobre uma imagem pode ser feita para, por exemplo, dar a impressão de que o observador andou para frente ou olhou para outra direção.

Existem diversas abordagens matemáticas para se realizar esta deformação bidimensional, tais como as apresentadas detalhadamente em (Gomes, 1997). Nas sessões 2.4.2 e 2.4.3 apresentam-se respectivamente dois métodos convenientes para a elaboração do conceito estendido de impostores: O *view-morphing* (Seitz, 1996) e o 3D *Image Warping* (McMillan, 1997), que é o modelo escolhido para a implementação deste trabalho.

2.4.1 Definição de *Warping* em Imagens

O *warping* de uma imagem consiste numa função $w: U \rightarrow W$, onde $U, W \subset R^2$. Esta função transforma a posição de um ponto pertencente a uma imagem entrada ou fonte U, produzindo uma nova imagem, que será chamada de imagem destino W.

Intuitivamente, o *warping* de imagens pode ser entendido como o deslocamento de um *pixel* de uma imagem fonte para outra posição. O resultado obtido ao deslocar todos os *pixels* da imagem fonte gera uma imagem destino.

Um exemplo de aplicação direta do processo de *warping* consiste no *morphing* de imagens: transições suaves de uma imagem para outra.

2.4.2 View-Morphing

Chen (1993) e Sauer (2002) descrevem uma técnica capaz de criar transições suaves de imagens, de maneira a poder gerar situações intermediárias entre elas, denominada de *optical flow*. Esta técnica requer que sejam fornecidos ou calculados vetores com a direção e a intensidade do movimento que cada *pixel* deverá sofrer para alterar corretamente a imagem fonte. Entretanto, esta técnica tem uma série de limitações com relação à preparação das imagens para serem utilizadas, já que fornecer ou calcular estes vetores pode em alguns casos ser impraticável. Uma série de trabalhos mais recentes aborda este problema, tais como (Horry 1997, Szeliski 1997, Kanade 1997). A técnica do *view morphing* (Seitz, 1996) requer um conjunto de imagens de diversas vistas do objeto sendo representado e é capaz de gerar vistas do elemento para ângulos não fornecidos pelo conjunto de figuras. Isto é feito sem a necessidade de vetores de direção de movimento para cada *pixel*, mas sim realizando uma interpolação suave entre duas imagens (figura 2.5).

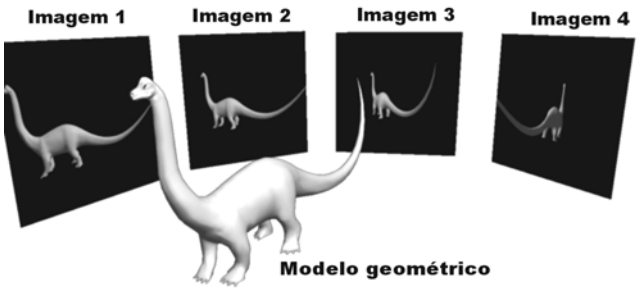


figura 2.5 – Para criar um objeto baseado na técnica de *view morphing*, deve-se ter um conjunto de imagens tiradas ou geradas ao redor do objeto, de maneira a poder interpolar as vistas intermediárias.

O *morphing* tradicional é determinado por duas imagens I_0 e I_1 e pela função de transição C_0: $I_0 \rightarrow I_1$ e C_1: $I_1 \rightarrow I_0$. Existe uma correspondência entre pontos de

uma imagem e outra. A correspondência para alguns destes pontos chaves é dada pelo usuário, sendo que os demais pontos são calculados automaticamente por uma função de interpolação. Fornecer estes pontos chaves é um dos principais inconvenientes do *view-morphing* em relação ao 3D *image warping*, pois requer um processo de autoria mais sofisticado.

Uma função de *warping* será construída a partir da função de transição:

$$W_0(p_0, s) = (1-s)p_0 + sC_0 p_0$$
$$W_1(p_1, s) = (1-s)C_1 p_1 + sp_1 \qquad (2\text{-}2)$$

W_0 e W_1 são assim funções que retornam o valor de deslocamento de cada ponto $p_0 \in I_0$ e $p_1 \in I_1$ em função de $s \in [0, 1]$. Assim, uma imagem intermediária I_s é criada aplicando a função de *warping* às duas images I_0 e I_1 e calculando a média da cor dos pixels das imagens resultantes do *warping*.

O problema maior que se tem ao aplicar este cálculo de *warping* sobre duas imagens tiradas de um mesmo objeto com uma posição de câmera ligeiramente alterada consiste em que as retas (contornos, por exemplo) do objeto não serão consistentes ao longo da transição. Isto faz que, por exemplo, uma reta possa temporariamente converter-se numa curva, como mostra a figura 2.6.

Diz-se que uma transformação é *shape preserving* no caso em que dadas duas imagens de um mesmo objeto visto de ângulos diferentes, esta transformação seja capaz de gerar uma imagem representando uma vista intermediária do mesmo objeto, sem sofrer nenhuma deformação.

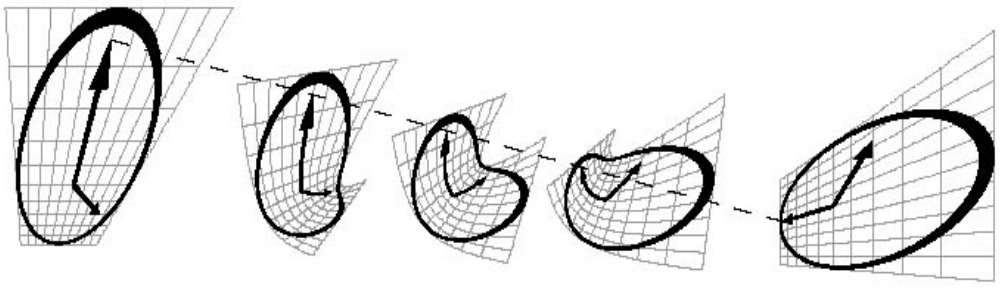


figura 2.6 – O *warping* que ocorre nestas duas imagens não é *shape preserving*, pois ocorre uma deformação nas imagens intermediárias. (Seitz, 1996)

Os algoritmos tradicionais de *morphing* (Wolberg 1990, Lee 1992, Beier 1992) não podem ser utilizados para esta finalidade, pelo fato de não serem *shape*

preserving. Já o algoritmo de *view morphing* descrito por Seitz (1996) é capaz de garantir esta propriedade.

Para calcular o *morphing* são necessárias, além das duas imagens representando vistas diferentes do mesmo objeto I_0 e I_1, suas respectivas matrizes de projeção Π_0 e Π_1 e um conjunto de *pixels* de correspondência de uma imagem para outra. A princípio, esta correspondência deve ser fornecida pelo usuário.

Estes dados são em geral necessários para qualquer técnica de *morphing.* Em geral, o *morphing* é correto quando a correspondência for correta e completa (quando isto não ocorrer, surgem "buracos" nas imagens intermediárias, devido à falta de informações). No caso do *view morphing*, esta correspondência correta garante também que as imagens geradas sejam *shape preserving*.

Resumidamente, o algoritmo funciona da seguinte maneira: sejam I_0 e I_1 as imagens a serem interpoladas e $\Pi_0 = [H_0 \mid -H_0C_0]$, $\Pi_1 = [H_1 \mid -H_1C_1]$ suas respectivas matrizes de projeção. De maneira a simplificar os cálculos, convém escolher um sistema de coordenadas, cujo eixo X coincide com a reta em que o observador se moveu. Isto permite ter-se $C_0 = (X_0, 0, 0)$ e $C_1 = (X_1, 0, 0)$. O eixo Y deste sistema pode ser obtido pelo produto vetorial entre as normais das duas imagens. Feito isto, imagens interpoladas de I_0 e I_1, conforme ilustra a figura 2.7, podem ser obtidas pela posição do observador C_s, dada pela equação de interpolação e pela matriz de projeção $\Pi_s = [H_s \mid -H_sC_s]$. Assim, o processo de *view morphing* se resume a três etapas:

1) Aplicar a matriz de transformação de projeção H_0^{-1} a I_0 e H_1^{-1} a I_1, produzindo as imagens intermediárias I^*_0 e I^*_1. Esta primeira etapa faz com que os planos das duas imagens estejam paralelos, sem alterar o centro de cada uma. É interessante notar que I^*_0 e I^*_1 representam vistas dadas pelas matrizes de projeção $\Pi^*_0 = [I_d \mid -C_0]$ e $\Pi^*_1 = [I_d \mid -C_1]$, sendo I_d a matriz identidade de dimensão 3. Estas duas imagens intermediárias possuem uma propriedade importante que é o fato de os pontos que se correspondem estarem na mesma linha da imagem, o que permite que a interpolação necessária para gerar I_s seja feita numa única dimensão.
2) Gerar I^*_s através de uma interpolação linear da posição dos pontos e das cores de pontos correspondentes entre I^*_0 e I^*_1 utilizando alguma equação de

interpolação. Esta etapa corresponde ao *morphing* propriamente dito e move o centro da imagem para C_s.

3) Aplicar H_s a I^*_s de maneira a obter a imagem interpolada I_s. Aqui finalmente se transforma o plano da imagem para a sua posição e orientação corretas.

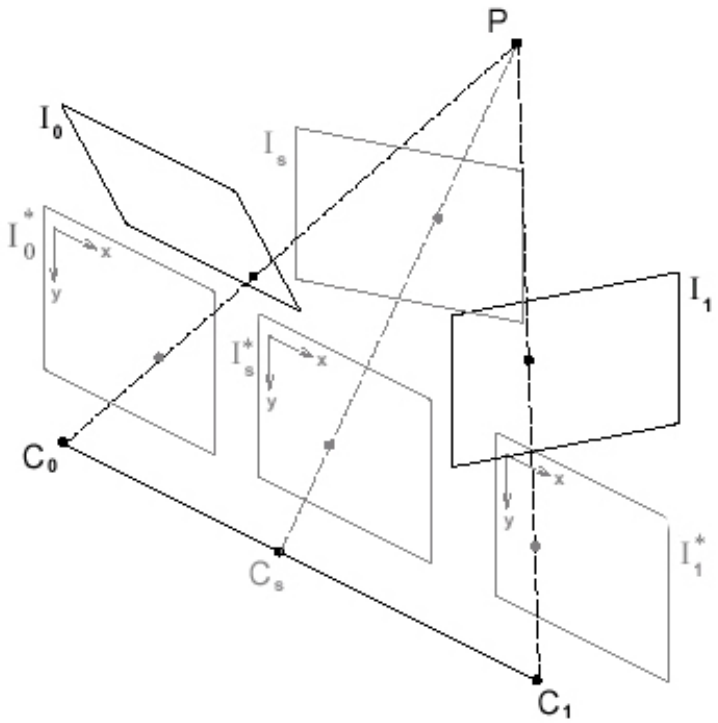


figura 2.7 – I_0 e I_1 correspondem às imagens originais. I^*_0 e I^*_1 são as imagens intermediárias obtidas a partir da multiplicação da matriz de projeção com H_0^{-1} e H_1^{-1}. I^*_s consiste na interpolação entre as imagens intermediárias e I_s será o resultado final do *view morphing* (Seitz, 1996).

2.4.3 3D Image Warping

O *3D image warping* definido por McMillan (1997) consiste numa função de transformação geométrica $w:U' \rightarrow W \subset \boldsymbol{R}^2$ capaz de mapear uma imagem fonte i_f para uma imagem destino i_d. A imagem fonte deve conter, além das cores dos *pixels*, a profundidade correspondente à superfície que está sendo representada pelo respectivo *pixel*. Além disso, conhecem-se os dados relacionados à câmera desta imagem fonte (a posição $\dot{C}_f$ bem como o seu plano de projeção). Utilizando o modelo de câmera perspectiva tem-se que um ponto $\dot{x}$ (figura 2.8) pertencente à cena geométrica pode ser descrito pela equação 2-3:

$$\dot{x} = \dot{C}_f + (\vec{c} + u_f \vec{a} + v_f \vec{b}).t_f(u_f, v_f) \tag{2-3}$$

Onde (u_f, v_f) são as coordenadas da projeção deste ponto sobre o plano de projeção da câmera, os vetores $\vec{a}$ e $\vec{b}$ formam a base para o plano da imagem e os comprimentos correspondem às medidas de cada *pixel* no espaço euclidiano, $\vec{c}$ é o vetor da direção dada pelo centro de projeção à origem do plano da imagem e o coeficiente t_f (u_f, v_f) é dado pela razão da distância de $\dot{C}_f$ até $\dot{x}$ pela distância de $\dot{C}_f$ até a projeção de $\dot{x}$ no plano, dado pela coordenada (u_f, v_f). A equação 2-4 corresponde ao cálculo deste coeficiente:

$$t_f(u_f, v_f) = \frac{|\dot{x} - \dot{C}_f|}{|\vec{c} + u_f \vec{a} + v_f \vec{b}|} \tag{2-4}$$

A equação (2-3) pode ser rescrita utilizando operação de matrizes da seguinte maneira:

$$\dot{x} = \dot{C}_f + \begin{bmatrix} a_i & b_i & c_i \\ a_j & b_j & c_j \\ a_k & b_k & c_k \end{bmatrix} \begin{bmatrix} u_f \\ v_f \\ 1 \end{bmatrix} t_f(u_f, v_f) = \dot{C}_f + P.\vec{x}.t_f(u_f, v_f) \tag{2-5}$$

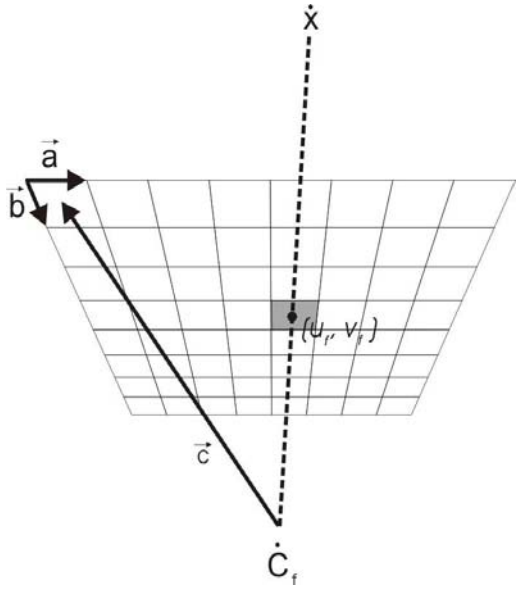


figura 2.8 – O ponto (u_f, v_f) é a projeção de $\dot{x}$ para o modelo da câmera fonte $\dot{C}_f$.

De forma semelhante, para uma outra posição do observador dada por $\dot{C}_d$ (figura 2.9), o mesmo $\dot{x}$ pode ser descrito da seguinte maneira:

$$\dot{x} = \dot{C}_d + P_d.\vec{x}_d.t_d(u_d, v_d) \tag{2-6}$$

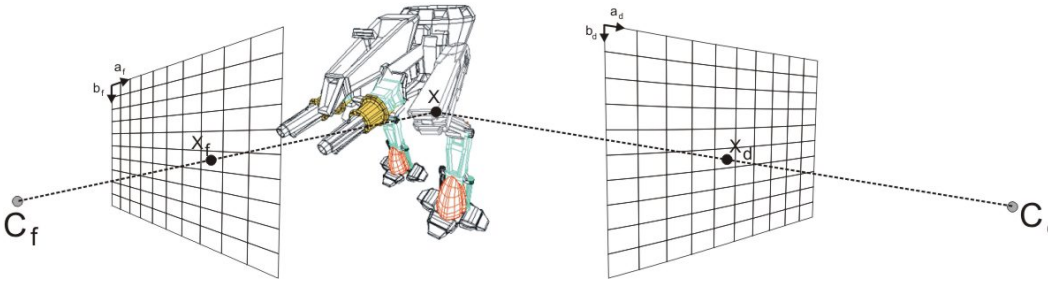


figura 2.9 – x_f e x_d são as projeções do ponto x sobre os planos de projeção das câmeras C_f (fonte) e C_d (destino), respectivamente.

Como $\dot{x}$ corresponde ao mesmo ponto, visto por observadores colocados em posições diferentes, pode-se criar a relação de equivalência entre as equações (2-5) e (2-6):

$$\dot{x} = \dot{C}_d + P_d.\vec{x}_d.t_d(u_d, v_d) = \dot{C}_f + P.\vec{x}.t_f(u_f, v_f) \tag{2-7}$$

Esta igualdade pode ser interpretada como sendo duas retas que se interceptam no ponto $\dot{x}$. Desenvolvendo a equação (2-7) tem-se:

$$P_d.\vec{x}_d.t_d(u_d, v_d) = (\dot{C}_f - \dot{C}_d) + P.\vec{x}.t_f(u_f, v_f)$$

$$\vec{x}_d.t_d(u_d, v_d) = P_d^{-1}[(\dot{C}_f - \dot{C}_d) + P.\vec{x}.t_f(u_f, v_f)] \tag{2-8}$$

Desta igualdade pode-se concluir que vale a seguinte equivalência projetiva[*]:

[*] O vetor resultante possui a mesma direção e sentido, podendo ser apenas o módulo diferente.

$$\vec{x} \doteq P_d^{-1}[P_f.\vec{x}_f.t_f(u_d,v_d)+(\dot{C}_f-\dot{C}_d)] \tag{2-9}$$

Esta equação explicita uma propriedade importante em relação à imagem que se deseja gerar: qualquer nova posição do observador não requer a informação de profundidade dos *pixels* da imagem a ser gerada (que na verdade é uma informação que não se dispõe), sendo suficiente apenas a profundidade dos *pixels* da imagem fonte.

Para chegar à formulação final da equação de 3D *image warping* de McMillan (equação 2-10), divide-se a equação (2-8) por $t_f(u_f, v_f)$ e distribui-se a matriz inversa de projeção P_d^{-1}:

$$\vec{x} \doteq P_d^{-1}P_f.\vec{x}_f + P_d^{-1}(\dot{C}_f-\dot{C}_d).\partial_f(u_f,v_f) \tag{2-10}$$

Onde $\partial_f(u_f,v_f)=1/t_f(u_f,v_f)$ é denominado de disparidade generalizada do *pixel* (u_f, v_f) da imagem fonte.

Esta equação pode ser vista como a composição de duas transformações bidimensionais: A primeira parcela representa uma transformação perspectiva planar homográfica sobre a imagem fonte (que pode ser vista como uma projeção de textura) e a segunda equivale a uma transformação *pixel* a *pixel*[**], proporcional ao valor da disparidade generalizada (dada pelo termo $\partial_f(u_f,v_f)$) na direção do ponto epipolar do plano da imagem destino (Oliveira, 2000).

A parcela de transformação perspectiva pode ser resolvida pelas implementações convencionais de projeção de textura por hardware. Quanto à transformação *pixel* a *pixel*, é possível reescrevê-la através de uma estrutura unidimensional simples, o que permite uma eficiente implementação por *software*.

O capítulo 3, após apresentar diversos métodos existentes para modelar elementos por imagens, discute com mais detalhes a técnica de texturas com relevo proposta por Oliveira (2000), que consiste numa implementação da equação de 3D *image warping* apresentada. Os impostores com relevo são uma

[**] Também conhecido com o nome de operação *per-pixel*

extensão desta abordagem, procurando aumentar sua capacidade para modelar objetos quaisquer.

2.5 Discussão

As duas formas de *warping* apresentadas podem servir para se elaborar os impostores com relevo. Como foi apresentado, o *view morphing* apresenta uma deficiência que consiste em necessitar dos pontos de correspondência entre as duas imagens. Entretanto, dado que os impostores com relevo trabalham com imagens sintetizadas dinamicamente pela aplicação, este problema pode ser resolvido, da seguinte forma: cada vértice do modelo recebe um identificador. Ao renderizar cada imagem, associa-se à estrutura do *pixel* que representa a projeção de cada vértice o identificador correspondente. Ao gerar outra imagem, os *pixels* dos mesmos vértices receberão um identificador equivalente ao da primeira imagem. Os pontos de correspondência serão aqueles com os mesmos identificadores.

A principal vantagem do *warping* apresentado por McMillan (1997) consiste em que, para cada posição de câmera, apenas é necessário uma imagem fonte. Esta abordagem requer, no entanto, que seja fornecida a profundidade de cada *pixel*, o que é trivial para o caso de imagens sintetizadas. No próximo capítulo, após uma síntese dos principais métodos para modelar objetos através de imagens, detalha-se o funcionamento do mapeamento de texturas com relevo e introduz-se o conceito de impostores com relevo.

3 Modelagem de Objetos Baseada em imagens

But... Look, see those birds? At some point a program was written to govern them. A program was written to watch over the trees, and the wind, the sunrise, and sunset. There are programs running all over the place. (The Matrix)

3.1 Introdução

Há diversas aplicações que podem requerer que apenas alguns grupos de objetos sejam representados por imagens (*image-based objects*), deixando que o restante da cena seja representado por geometria tradicional (geo*metry-based objects*). Isto permite que se possam utilizar as vantagens existentes em cada tipo de representação: geometrias simples e eficientes para serem processadas pelo *pipeline* da GPU são encaminhadas diretamente ao *hardware*. Objetos de natureza mais complexa ou que requerem uma visualização cujo *hardware* gráfico seja incapaz de calcular são pré-processados como objetos baseados em imagens e re-encaminhados ao *pipeline* da GPU na forma de *sprites*, *billboards*, impostores, portais ou panoramas.

Modelar um objeto através de imagens pode resumir-se a projetar uma textura com a figura do elemento que se deseja criar sobre um plano inserido no espaço. Entretanto, apenas fazer isto traz uma série de problemas: ao mover o objeto ou a câmera este plano texturizado não reproduz corretamente as novas vistas do objeto. Mesmo girando o plano, de forma que a sua normal esteja sempre apontada para o observador, não se tem idéia de tridimensionalidade do mesmo, pois a imagem permanece sempre a mesma. Neste capítulo são apresentadas diversas abordagens para representar objetos deste tipo, bem como soluções para o problema recém apresentado: *sprites*, impostores, objetos com texturas com relevo e finalmente a proposta deste trabalho, que são os impostores com relevo.

3.2 Sprites e Billboards

Sprites são planos (comumente representados por polígonos) com texturas aplicadas sobre si. Ao se criar um objeto através do mapeamento de uma imagem (foto ou imagem sintetizada) sobre um plano, surge um inconveniente: o objeto sempre será retangular. Isto ocorre devido à natureza retangular da imagem a ele aplicada, fazendo com que os elementos não possuam uma silhueta adequada. Para contornar este problema, costuma-se utilizar texturas com transparência: o valor da transparência de cada *pixel* será descrito por um byte extra, chamado de canal alfa (*alpha channel*). Para resolver a aparência plana do *sprite*, pode-se realizar uma rotação do mesmo sempre que ele ou o observador se movem, garantindo que a normal do *sprite* esteja sempre apontada para a câmera. Este processo também é conhecido como *billboarding* e o polígono correspondente é denominado de *billboard* (McReynolds, 1999).

figura 3.1 – Um *sprite* é normalmente implementado por um polígono, cuja normal aponta sempre para o observador e com uma textura com transparência. Neste exemplo pode-se reparar que, independente do ponto de vista, o cactos é sempre igual.

As técnicas de *billboarding* combinadas com alfa e animações podem ser usadas para representar fumaça, fogo, *fog*, nuvens e outros fenômenos não sólidos. Há várias formas de *billboards*, sendo as mais comuns:

- Alinhadas ao plano de visão (bons para *sprites* circulares, como partículas);
- Orientados ao ponto de vista (bons para fumaça, fogo, nuvens e outros fenômenos cujas aleatoriedades disfarçam bem as distorções provocadas por este tipo de *billboard*);
- Axiais (bons para árvores e outros objetos de natureza cilíndrica);

Em (Dally, 1996) estende-se o conceito de *sprite* para um objeto representado por uma esfera envolvente (*bounding sphere*). Várias vistas deste objeto são pré-calculadas para diversas posições de câmera ao redor da esfera. Estas imagens são armazenadas numa estrutura chamada de árvore delta (*Delta Tree*). Para cada posição de câmera calcula-se um polígono inscrito na projeção da esfera sobre o plano de projeção da câmera e escolhe-se a imagem mais adequada para esta posição. Este modelo tem como grande desvantagem o fato do centro de projeção do objeto ter que ser sempre o centro da esfera, fazendo com que objetos mal distribuídos ao redor deste ponto sejam deformados.

O trabalho de Pulli (1997) utiliza uma malha simplificada para representar um objeto 3D com poucos polígonos. Associado a esta malha, há um conjunto de imagens que são usadas como texturas e aplicadas sobre a geometria simplificada.

Os *Sprites with Depth* (Shade, 1998) permitem aumentar o poder de representação dos *sprites* realizando deslocamentos ortogonais dos *texels* da textura que os representam. O método utiliza um algoritmo de dois passos para calcular a cor dos *pixels* da imagem sendo gerada a partir da textura fonte do *sprite*. O primeiro passo consiste em gerar um mapa de profundidade intermediário através de um mapeamento que utiliza uma transformação 2D sobre o mapa de profundidade da imagem fonte. No segundo passo, cada *pixel* da imagem desejada passa por uma transformada homográfica (projeção perspectiva sobre um plano), sendo que as coordenadas resultantes são usadas para indexar o mapa de profundidade calculado no primeiro passo. Os valores de deslocamento encontrados são finalmente multiplicados pelo ponto epipolar da imagem sendo formada e adicionada ao resultado da homografia. Estas coordenadas são usadas para indexar a cor dos *pixels* de destino.

Uma abordagem mais recente para as LDI's (*Layered Depth images*) pode ser encontrada em (Bayakovski, 2002), onde as imagens são representadas através das cores dos *pixels* e uma tabela com a distância de cada *pixel* até a superfície que está sendo vista no mesmo. A vantagem desta representação sobre o LDI padrão é que as imagens podem representar objetos mais realistas, com características que tornam complexa sua visualização em tempo real. Além disso, a complexidade da visualização se torna apenas proporcional ao tamanho da imagem e não à complexidade geométrica sendo representada. Na implementação de Bayakovski (2002) utilizam-se duas maneiras de representar tais dados:

- *DepthImages* com texturas simples: Conjunto de imagens com seus respectivos mapas de profundidade. Estas imagens podem compor um objeto através de *Box-Texture* (faces de um *bounding box*) ou através de *Generalized Box Textures* (texturas posicionadas arbitrariamente sobre o objeto);
- *OctreeImages*: Representação através de um volume (*Binary Volumetric Octree*), na forma de *octree* (Hunter, 1978), dizendo se um *voxel* está ocupado ou não. Para otimizar tal estrutura, a *octree* é antes de mais nada convertida para a forma "*breadth-first traversal linkless*", onde cada nó é representado por um simples *byte* cujos *bits* indicam se o seu sub-cubo correspondente deve também ser dividido. Informações de cores são armazenadas num conjunto de imagens de referência, que são obtidas pela projeção dos *voxels* representados no plano das imagens pré-definidas.

Jakulin (2000), procurando representar vegetação através de sprites, apresenta uma outra extensão, onde se usam vários planos para representar a folhagem e fazendo um *blending* entre as texturas, dependendo da posição de onde se encontra o observador.

3.3 Impostores

Os impostores (Maciel, 1995) consistem também em métodos eficientes para representar objetos através de imagens. A idéia é representar um objeto tridimensional através de um *sprite*, porém diferentemente que no conceito padrão, estes objetos estão realmente definidos como modelos geométricos e são

calculados durante a aplicação corrente e projetados num plano como textura com transparência.

Impostores são *billboards* gerados em tempo real que distorcem a imagem de maneira semelhante ao que ocorreria com a geometria real do objeto. Devido à própria natureza dos impostores, os *billboards* "orientados ao ponto de vista" são os mais adequados. A imagem de textura de um impostor também pode ser tratada em tempo real para simular determinados efeitos (p. ex. imagens fora de foco para um efeito de profundidade de campo).

Na prática, um impostor deve ser reusado por vários pontos de vista suficientemente próximos (explorando a coerência quadro-a-quadro). Por esta razão, impostores são mais adequados para objetos estáticos pequenos (ou suficientemente distantes). Objetos lentos à grande distância são igualmente bons candidatos para esta técnica. Os testes para verificar se um determinado impostor é ainda válido para o ponto de vista corrente são uma questão fundamental. O presente trabalho trata exatamente desta questão quando propõe os impostores com relevo.

Schaufler (1995) e (1997) descreve os impostores como métodos adequados para minimizar o número de vezes em que se deve visualizar um determinado objeto que possui certa complexidade geométrica e que é, portanto, caro para o *pipeline* de tempo real (Forsyth, 2001). Sob este ponto de vista, os impostores são uma espécie de *buffer* para estes objetos, de forma que sua visualização é reaproveitada enquanto ainda "serve" para uma determinada posição do observador, como foi discutido na seção 2.2.3.

O tamanho da imagem gerada para estes objetos pode ser proporcional ao tamanho que ocupam na tela de projeção. Assim, se estão muito distantes, estes objetos são renderizados numa resolução pequena; porém, na medida em que se aproximam do observador e, portanto, aumentam em tamanho no plano de visualização, estas imagens são refinadas para resoluções maiores. Em (Damon, 2003) descreve-se uma forma de implementar impostores utilizando técnicas de renderização em memória de vídeo, recurso que está disponível nas placas gráficas mais recentes, podendo aumentar o número de objetos sendo representados com este método, desde que haja memória de vídeo suficiente.

Impostores podem conter a profundidade associada a cada *pixel*. Neste caso, Schaufler (1997) passa a chamar este tipo de impostor de *nailboard*. Entretanto,

na presente tese, não se faz distinção entre estes dois termos. A profundidade armazenada em impostores pode ser utilizada no *Z-Buffer* do *pipeline*, de forma que o objeto possa ser penetrado por outros elementos, não havendo problemas de oclusão. Esta profundidade pode ser armazenada no próprio canal alfa da imagem, deixando-se apenas um *bit* reservado para indicar se o *texel* é transparente ou não.

Schaufler (1998) implementa um impostor com mapa de profundidade utilizando camadas de polígonos: o *sprite* consiste numa série de polígonos mapeados com texturas correspondentes às camadas respectivas. Dependendo da informação de profundidade presente no *sprite*, um *pixel* é desenhado ao nível de profundidade que lhe corresponda. O uso de transparência é fundamental neste caso.

3.4 Texturas com Relevo

A técnica de *displacement-mapping* (Cook, 1984) permite modificar a geometria de uma superfície tridimensional a partir de um mapa de profundidade, podendo assim criar superfícies complexas utilizando apenas um conjunto de imagens aplicadas sobre superfícies. Poder-se-ia então criar um sprite com *displacement-mapping*, para dar profundidade ao polígono. Esta operação necessita, no entanto, de uma malha de polígonos refinada, uma vez que deforma a geometria no sentido da normal de cada ponto da superfície (figura 3.2 (b)). Além disso, esta técnica não possui uma solução adequada para um tratamento em tempo real, uma vez que não possui uma formulação que permite uma transformação direta da textura para a coordenada da tela (vários pontos da geometria podem ser projetados no mesmo *pixel* da imagem final).

A técnica de *bump-mapping* (Catmull, 1974), por outro lado, possui esta formulação direta, não precisa de uma malha de polígonos refinada, mas não permite modelar a geometria de um objeto, pois somente manipula as normais de cada ponto da superfície, alterando apenas o seu sombreamento (figura 3.2 (a)). Justamente por não alterar a geometria do objeto, o *bump-mapping* é incapaz de corrigir problemas relacionados à paralaxe. A técnica de mapeamento de texturas com relevo possibilita representar a geometria de uma superfície tridimensional e possui uma formulação direta para o mapeamento, podendo ser implementada por

hardware e, portanto, podendo ser utilizada em aplicações que exigem tratamento em tempo real.

O mapeamento de texturas com relevo (Oliveira, 2000b), ao contrário do *bump-mapping*, permite criar primitivas geométricas, baseadas unicamente em imagens com profundidade. Esta técnica é utilizada na implementação do presente trabalho para resgatar objetos pré-renderizados, utilizando o *Z-Buffer* previamente armazenado, possibilitando dar-lhes a profundidade e tentar garantir uma coerência de paralaxe do objeto em relação à cena.

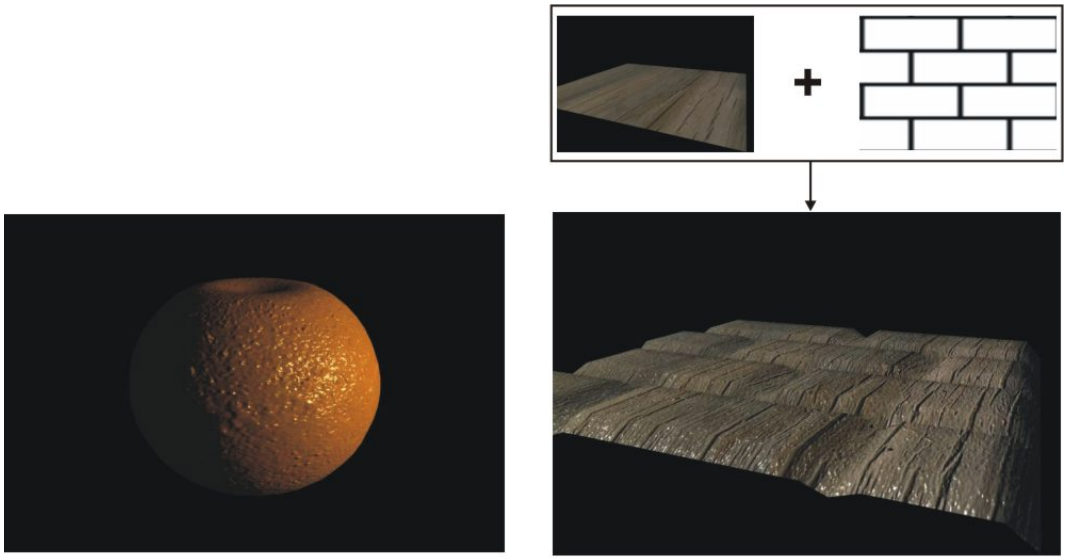


figura 3.2 – Em (a) o objeto possui um *bump-mapping* que lhe dá a aparência de ser rugoso. Perceba-se que o contorno do objeto não é deformado, uma vez que esta técnica apenas altera o sombreamento da superfície. Em (b) se está aplicando a textura de ladrilhos ao plano como *displacement-mapping*. Pode-se perceber que a geometria está sendo de fato alterada.

O método de texturas com relevo consiste numa fatorização da equação de 3D *image warping* (McMillan 1995, McMillan 1997), apresentado na seção 2.4.3, em duas etapas separadas: a primeira, denominada de *pre-warping*, corresponde à transformação *pixel* a *pixel,* proporcional ao valor da disparidade generalizada; a segunda, nada mais é do que um mapeamento de textura convencional, que corresponde à transformação perspectiva.

O *pre-warping* é aplicado a imagens que possuem mapa de profundidade para cada *texel* e consiste sobretudo no movimento (*warping*) destes *texels*. Este movimento é realizado de tal forma que procura corrigir ou reduzir o efeito de paralaxe, resultante do deslocamento do observador em relação ao objeto que

possui a textura com relevo. Cabe a esta etapa resolver o problema de preenchimento de espaços vazios que surgem. A etapa seguinte - texturização - realiza as operações de escala, rotação, filtragem e as deformações de perspectiva necessárias para o mapeamento correto nos polígonos da textura com relevo.

De acordo com Oliveira (2000a), uma imagem com profundidade é definida pelo par [i_d, K], onde i:$U' \rightarrow C'$ é uma imagem digital e K é o modelo de câmera associada a i. Cada elemento do espaço de cores C' de i possui também um valor escalar que corresponde à distância, no espaço euclidiano, entre o ponto amostrado e um ponto de referência de K. No caso de K ser o modelo de uma câmera com projeção perspectiva, este ponto de referência é o centro de projeção da câmera e a imagem é chamada de imagem de projeção perspectiva com profundidade. Entretanto, na implementação proposta por Oliveira (2000a), é utilizado um modelo de câmera ortográfica. Neste caso, o ponto de referência é o plano da imagem e a imagem correspondente é chamada de imagem de projeção paralela com profundidade. A figura 3.3 mostra os dois modelos de câmera.

A princípio, a imagem *i* pode ser vista como uma textura RGBA (*Red*, *Green*, *Blue* e *Alpha*), onde os canais RGB armazenam a cor deste *texel* e o canal *alpha* armazena a profundidade do mesmo. No entanto, fazendo-se isto, não é possível calcular uma iluminação correta para o objeto com a textura, pois não se pode reconstituir a normal original para este ponto. Portanto, na implementação feita, armazena-se em RGB a normal que o ponto sendo representado pelo *texel* possui na superfície geométrica, seguindo a indicação de Oliveira (2000a). Existe uma segunda imagem RGB que possui a cor de cada *pixel*, sem a sua iluminação difusa (o cálculo de iluminação é feito por *hardware* e é implementado através de um *Pixel Shaders*, na linguagem Cg, como será visto no capítulo 5).

Além das duas imagens, para se representar o objeto por uma textura com relevo é necessário o modelo da câmera ortográfica que lhe corresponde. Também é necessário definir um polígono, que é visto pelo *pipeline* como uma entidade geométrica da cena e sobre o qual a textura com relevo é aplicada.

A estrutura que armazena um objeto do tipo textura com relevo e o seu modelo de câmera é da seguinte forma:

Tipo Relief_Object

{

Imagem_RGBA: Mapa_Normais_Profundidade;

Imagem_RGB: Mapa_de_cor;

Camera: Câmera_fonte;

Polígono: Plano_do_Relief_Object;

}

Tipo Camera

{

Vetor3D a; // Dimensão horizontal da câmera

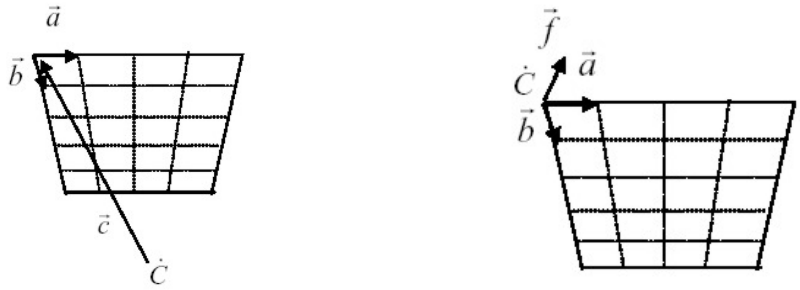
Vetor3D b; // Dimensão vertical da câmera

Vetor3D c; // Centro de Projeção da câmera (projeção paralela)

Vetor3D f; // $\vec{f} = \left|\vec{a} \times \vec{b}\right|$

Vetor3D C; // $\vec{c} = \dot{C} - Posição\_Corrente\_da\_camera$

}



(a) (b)

figura 3.3 – (a) corresponde a um modelo de câmera perspectiva, onde os vetores $\vec{a}$ e $\vec{b}$ formam a base para o plano da imagem. O comprimento destes vetores correspondem às medidas de cada pixel no espaço euclidiano, $\dot{C}$ é o centro de projeção e o ponto de referência de *K* e $\vec{c}$ é o vetor cuja direção é dada pelo centro de projeção e a origem do plano da imagem. (b) corresponde ao modelo de câmera ortográfica, onde o ponto de referência $\dot{C}$ coincide com a origem do plano da imagem e o vetor $\vec{f}$ é ortogonal ao plano da imagem e possui módulo unitário.

A profundidade armazenada no canal alpha deve estar quantizada para valores inteiros que estão entre 0 (pontos pertencentes à face frontal do *bounding box* do modelo) e 254 (pontos pertencentes à face de traz do *bounding box* do modelo), com valores intermediários para pontos que estão entre as duas faces. Reserva-se um valor (no caso, 255) para representar pontos vazios da textura com relevo. Esta limitação de valores ocorre porque o alpha de um RGBA é do tamanho de um byte. Caso isto não fosse feito, dever-se-ia utilizar uma matriz de pontos flutuantes, fazendo com que o tempo de transferência destes valores para a memória de textura fosse consideravelmente maior (Oliveira, 2000a).

A fatorização descrita inicialmente pode também ser interpretada da seguinte forma: a imagem fonte sofre primeiro uma pré-deformação (*pre-warping*) que depende das informações de profundidade de cada pixel e da posição do observador destino. Esta pré-deformação é feita no próprio plano de projeção da imagem para o observador fonte. Feito isto, utiliza-se a técnica padrão de mapeamento de textura para projetar a imagem deformada sobre o plano correspondente ao observador destino (figura 3.4).

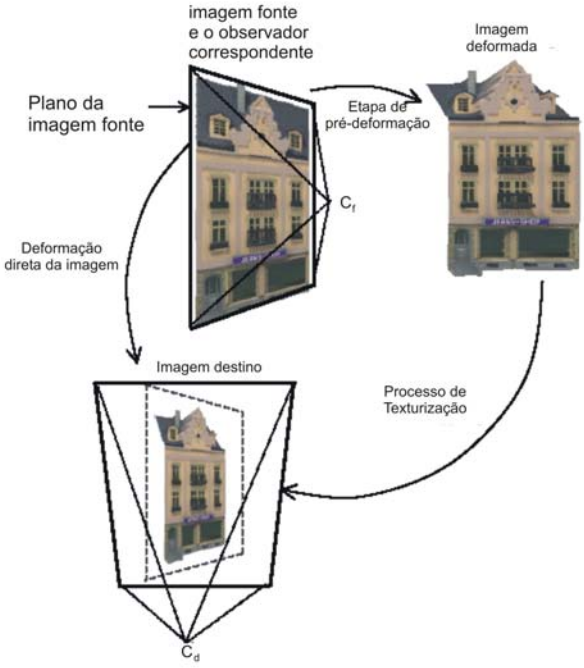


figura 3.4 – Etapas da fatorização da equação de McMillan (figura retirada de (Oliveira, 2000a))

A etapa de *pre-warping*, por sua vez, corresponde ao seguinte problema: encontrar o par (u_i, v_i) para cada pixel (u_f, v_f) de forma que ao ver a imagem deformada a partir do observador destino C_d, este novo par corresponda ao local onde (u_f, v_f) deveria estar no plano de projeção do observador fonte (figura 3.5).

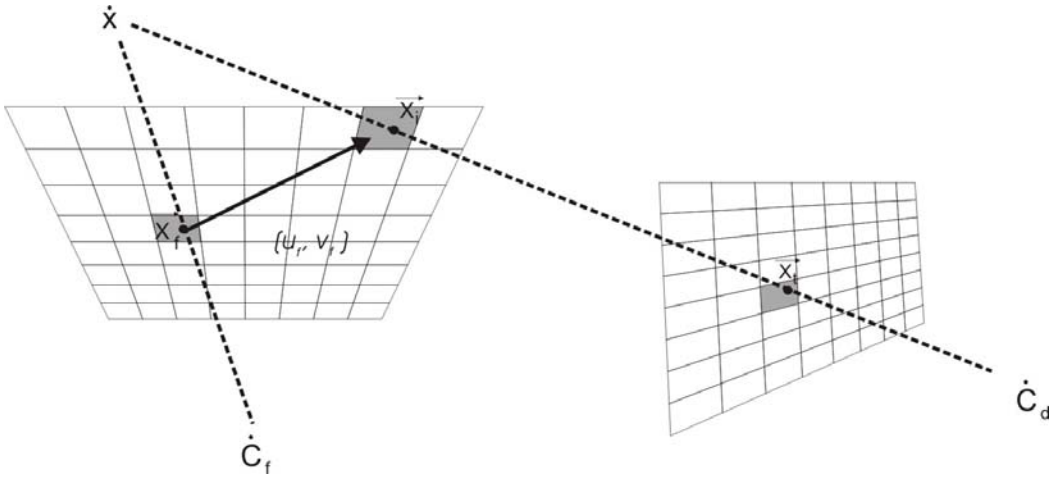


figura 3.5 – A etapa de *pre-warping* pode ser vista como uma transformação sobre o pixel x_f sobre o plano da imagem do observador fonte de forma que corresponda à projeção de $\dot{x}$ sobre este plano para o observador destino.

Como foi discutido no capítulo 2.4.3, a equação de 3D *image warping* que descreve o deslocamento relativo de um *pixel* possui, em relação ao observador destino, dependência apenas da sua posição final. Desta maneira, escolhendo-se adequadamente valores para $\vec{a}_d$, $\vec{b}_d$ e $\vec{c}_d$ uma série de simplificações podem ser feitas na equação de McMillan, chegando-se à seguinte formulação reduzida para a etapa de *pre-warping*:

$$u_i = \frac{u_f - k_1 \partial(u_f, v_f)}{1 + k_3 \partial(u_f, v_f)} \tag{3-1}$$

e $$v_i = \frac{v_f - k_2 \partial(u_f, v_f)}{1 + k_3 \partial(u_f, v_f)} \tag{3-2}$$

onde $k_1 = \frac{\vec{f}.(\vec{b} \times \vec{c})}{\vec{a}.(\vec{b} \times \vec{c})}$, $k_2 = \frac{\vec{f}.(\vec{c} \times \vec{a})}{\vec{b}.(\vec{c} \times \vec{a})}$, $k_3 = \frac{\vec{f}.(\vec{a} \times \vec{b})}{\vec{c}.(\vec{a} \times \vec{b})}$, sendo que *a*, *b*, *c* e *f* são os vetores ilustrados na figura 3.3. As formulações (3-1) e (3-2) são denominadas de equações de *pre-warping* para texturas com relevo.

Da fatorização destas equações, verifica-se que, apesar da etapa de *pre-warping* ser da mesma natureza bidimensional da imagem, pode também ser vista como um processo unidimensional, já que existe total independência entre as coordenadas u_i e v_i, ou seja, para determinar u_i não é necessário o valor de v_f e para determinar v_i não é necessário o valor de u_f. Isto implica na possibilidade de realizar o *warping* horizontal independentemente do *warping* vertical, podendo-se fazer um processo linear de *warping*, aplicando-o primeiro sobre as linhas e depois sobre as colunas. Esta independência possibilitou que no presente trabalho fossem implementadas diversas versões da etapa de *pre-warping*, conforme se apresenta na seção 8.2. Também se pode notar que quando $\partial(u_f, v_f) = 0$ a etapa de p*re-warping* não deforma a imagem original ($u_i = u_f$ e $v_i = v_f$), tornando o 3D *image warping* no processo padrão de texturização (segunda etapa do mapeamento de texturas com relevo).

O algoritmo que realiza o mapeamento de texturas com relevo pode ser descrito resumidamente da seguinte maneira:

Determinar o $\dot{C}_d$ e calcular as constantes k_1, k_2 e k_3;

Realizar o pre-warping partindo-se da imagem fonte:

- *Calcular a posição (u_d, v_d) para cada pixel (u_f, v_f), usando as eq.(3-1) e (3-2)*
- *Copiar a cor do pixel da imagem fonte para as posições de pre-warping;*
- *Tratar o conflito de pixels sobre uma mesma área e os buracos surgidos;*

Renderizar os polígonos aplicando as imagens com pre-warping como textura;

3.4.1 Ordem Compatível com Oclusão

Ao realizar o *warping*, surge o problema de conflito de *pixels* sobre uma mesma área: isto ocorre porque a profundidade de cada *pixel* da imagem pertence a pontos de superfícies com profundidades diferentes no mundo real. Assim sendo, um *pixel* pode deslocar-se mais do que o seu vizinho no processo de *warping* (isto pode ser facilmente visto na equação de McMillan observando-se o parâmetro $\partial(u_f, v_f)$).

Além disso, quando um *pixel* sofre um *warping* para uma determinada posição, mas nenhum outro *pixel* é movido para a posição que era ocupada por ele

anteriormente, surge o problema dos buracos. Caso estes espaços não sejam preenchidos com alguma cor, a imagem resultante do *warping* conterá regiões não pintadas.

Para resolver o problema de conflito de *pixels* (Oliveira, 2000a) sugere que seja utilizado o seguinte algoritmo:

Determinar a interseção do centro de projeção da imagem destino na imagem fonte[*]*;*
Dividir a imagem fonte em quadrantes, usando o ponto de interseção como pivô;
Se o Centro de Projeção da imagem fonte estiver atrás do centro de projeção da imagem destino, então para cada quadrante:
Começar o warping a partir do ponto de interseção encontrado e avançar em direção às bordas do quadrante;
Senão
Começar o warping a partir das bordas do quadrante e avançar em direção ao ponto de interseção encontrado;

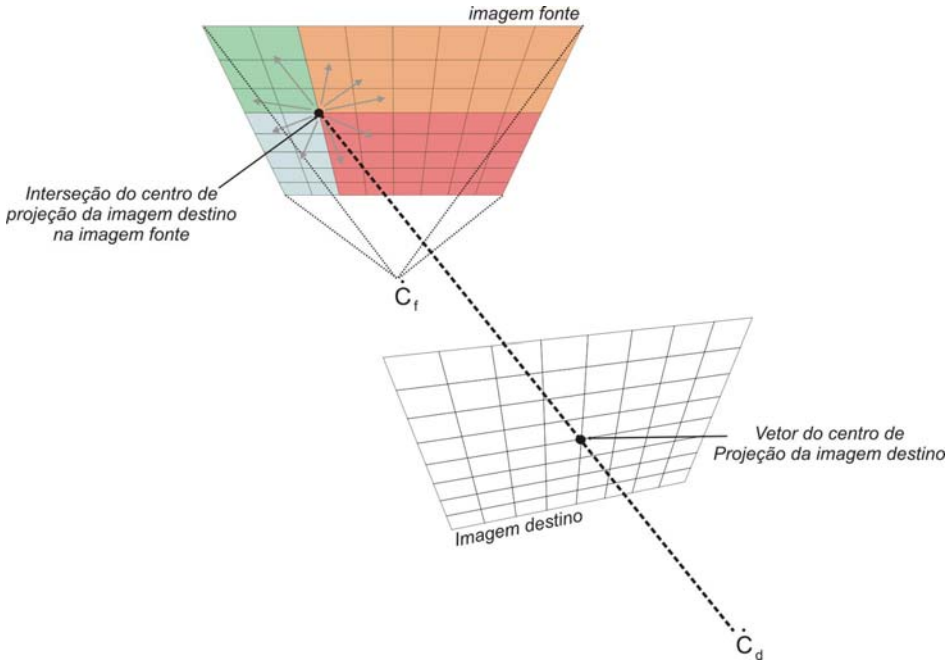


figura 3.6 – Algoritmo do Pintor adaptado para critério de ordem de plotagem dos *pixels* com profundidade, implementado no mapeamento de texturas com relevo. O ponto de interseção do Centro de Projeção da imagem destino com o plano da imagem fonte está ilustrado e cada quadrante criado possui uma cor. Note-se que neste exemplo $\dot{C}_d$

[*] Como foi visto, trata-se do epipolo.

(Centro de Projeção da imagem destino) está atrás $\dot{C}_f$ (Centro de projeção da imagem fonte) e portanto o *warping* ocorre do ponto epipolar em direção às bordas.

A figura 3.6 ilustra este algoritmo, que na verdade é uma adaptação do algoritmo do pintor. Intuitivamente, pode-se dizer que este algoritmo dita a ordem em que os *pixels* da imagem fonte devem sofrer o *warping* para que não seja necessário realizar um teste de profundidade no caso de dois ou mais *pixels* caírem numa mesma posição na imagem destino, como se pode ver na figura 3.7. No exemplo ilustrado da figura 3.6, realiza-se primeiro o *warping* nos *pixels* que estão mais próximos do ponto epipolar, já que estão mais próximos do centro de projeção da imagem destino.

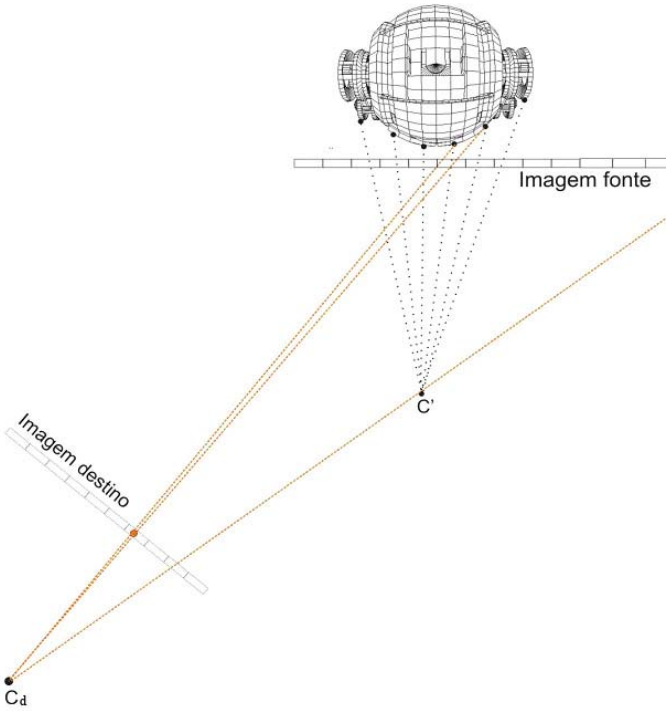


figura 3.7 – Sempre que várias amostras são projetadas sob um mesmo *pixel* da imagem destino, aquela cujo *pixel* da imagem fonte estiver mais distante da linha epipolar é a que está mais próxima do centro de projeção da imagem destino e portanto deverá ser o último a ser plotado.

3.4.2 Texturas com relevo em panoramas cilíndricos

Em (ElHelw, 2003), estende-se o conceito de texturas com relevo para imagens mapeadas em cilindros. Uma das aplicações deste trabalho consiste em criar panoramas com detalhes 3D e com uma pequena correção para paralaxe nos

movimentos realizados em seu interior. Como se pode ver na figura 3.8, as etapas deste algoritmo consistem em:

- Transformar uma textura com relevo que possui uma projeção plana para um espaço cilíndrico, através de um mapeamento simples (observe-se que neste caso o centro de projeção da textura, que era no infinito, passa a estar no eixo do cilindro);

- Gerar uma imagem intermediária, através de uma deformação da textura com relevo transformada na etapa anterior utilizando uma equação de *warping* cilíndrico;

- Projetar a imagem intermediária para o cilindro, por meio de mapeamento de textura padrão.

De forma resumida, pode-se dizer que este método faz o *warping* dos *pixels* da textura com relevo, transformando-o para um sistema espacial cilíndrico.

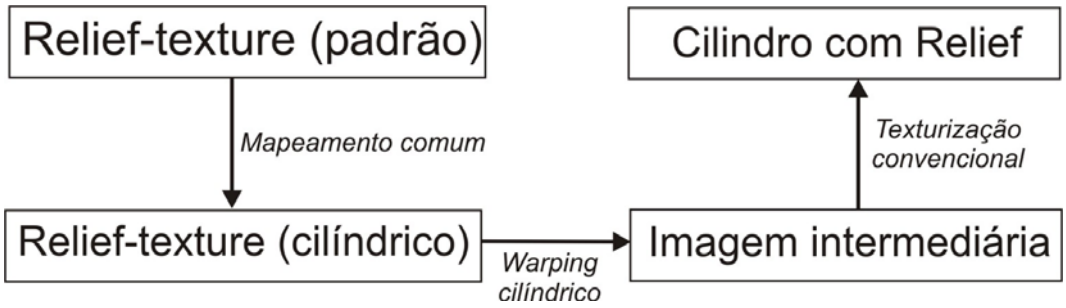


figura 3.8 – Etapas do mapeamento de texturas com relevo para espaços cilíndricos.

3.4.3 Representação de objetos 3D utilizando um conjunto de texturas com relevo

Apenas uma textura com relevo não permite que um objeto seja representado por completo. Como não se disponha de dados para partes oclusas do objeto, chega um momento em que o *warping* acumula demasiados erros, invalidando a textura inicial. Para solucionar este problema, Oliveira (1999) e Oliveira (2000a) propõem representar um objeto através de um paralelepípedo, onde cada uma das 6 faces possui uma textura com profundidade gerada para a vista correspondente, através de uma câmera ortogonal. Entretanto, utilizando-se apenas o 3D *image warping* e o critério de oclusão discutido na seção 3.4.1 para cada face separadamente, *pixels* do objeto podem ser projetados em ordem errada,

pois partes da superfície de uma face podem cair no espaço de outra face, dependendo do *warping* que está sofrendo. Desta forma, Oliveira (1999) descreve uma ordem adequada para a escolha das faces a sofrerem o *warping*. Este método possui uma grande vantagem que consiste em permitir que o objeto representado sofra as transformações básicas (translação, rotação e escala) com simples manipulações sobre o seu Centro de Projeção e os vetores $\vec{a}, \vec{b}$ e $\vec{c}$ que definem as câmeras.

Reduzindo-se o problema ao espaço bidimensional, sem perda de generalidade, pode-se descrever o algoritmo da seguinte forma:

Dividir o espaço em 8 regiões, conforme ilustra a figura 3.9. Se o observador estiver numa região ímpar (como é o caso de O_1 da figura 3.9), as 3 faces mais próximas e que contribuem para a visibilidade do objeto, são classificadas de frente (face *a* para O_1), esquerda (face *b* para O_1) e direita (face *d* para O_1). Desta forma, aplica-se um *pre-warping* nas faces esquerda e direita sobre o polígono que representa a face da frente. Depois disto, realiza-se o *pre-warping* da face frontal sobre o mesmo polígono, sobre-escrevendo as regiões que foram pintadas pelos *pre-warpings* anteriores.

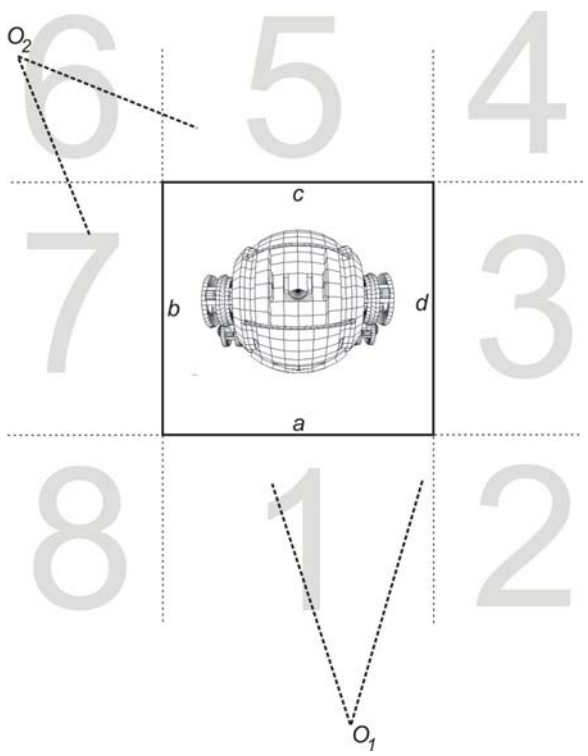


figura 3.9 – Objeto sendo representado por 6 texturas com relevo, visto de cima. O espaço é dividido em 8 regiões, que podem ser de 2 tipos: região par e região ímpar.

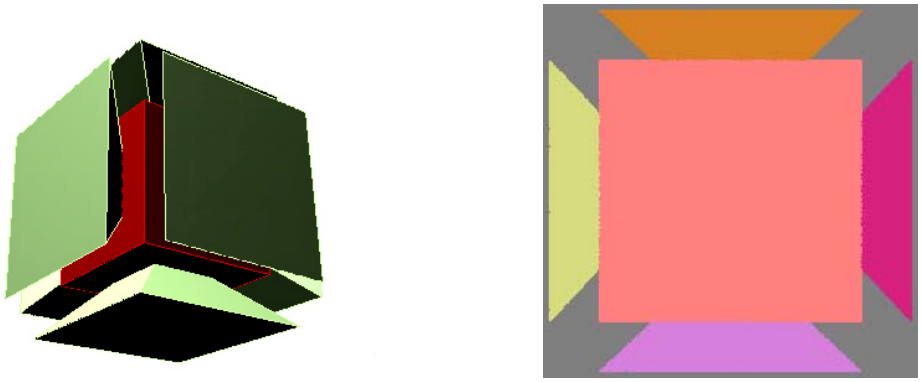
Se o observador estiver numa região par (como é o caso de O_2 na figura 3.9), as duas faces potencialmente visíveis serão classificadas de face esquerda (face *c* para o caso de O_2) e face direita (face *b* para o caso de O_2). Desta maneira, realiza-se o *pre-warping* da face esquerda sobre o polígono que sustenta a face da direita e a seguir se realiza o *pre-warping* da face direita sobre seu próprio polígono, sobre-escrevendo as regiões já preenchidas. De igual maneira, realiza-se o *pre-warping* da face direita sobre o polígono que sustenta a face esquerda e a seguir o *pre-warping* da face esquerda sobre seu próprio polígono.

4 Impostores com Relevo

I just wish... I wish I knew what I'm supposed to do. That's all. I just wish I knew. (The Matrix)

4.1 Introdução

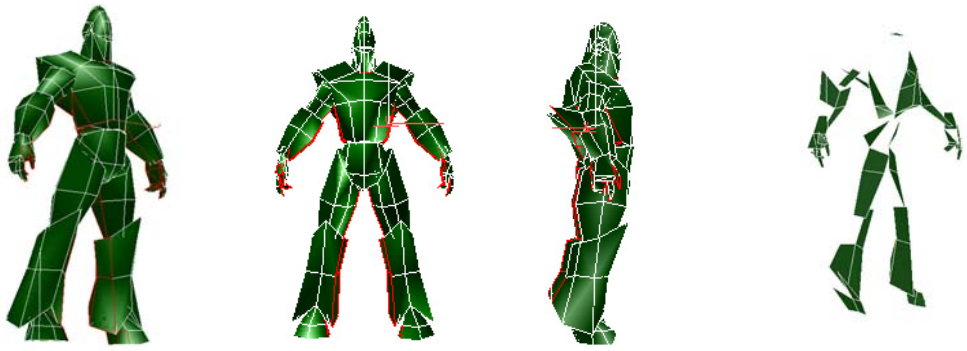
Os objetos modelados por texturas com relevo, por corrigirem o problema de paralaxe, possuem um tempo de vida extremamente maior do que objetos modelados por simples *sprites*. Entretanto, como foi visto na seção 3.4, apenas uma textura não é suficiente para representar um objeto por inteiro. Ao representar um objeto através de um paralelepípedo de texturas com relevo, conforme descrito em 3.4.3, tem-se uma limitação do tipo de objetos possíveis de serem representados. A figura 4.1 mostra um exemplo de topologia extrapolada e incorreta para tal método.



(a) (b)

figura 4.1 – A parte vermelha do objeto representado em (a) não pode ser vista utilizando o método proposto em 3.4.3. Todas as 6 texturas com relevo são iguais ao mapa de normais e profundidade representado em (b).

De forma geral, partes de objetos que não são visíveis por nenhuma das 6 vistas ortogonais, desaparecerão na sua representação por texturas com relevo. Tal caso se dará com frequência em objetos com características côncavas. A figura 4.2 (d) mostra um personagem com os polígonos que não são alcançados através deste método.



(a) (b) (c) (d)

figura 4.2 – Nas vistas ortogonais muitos dos polígonos do objeto não podem ser vistos e são incorretamente interpolados pelo mapeamento de texturas com relevo padrão para uma posição de câmera, como a de (a). Em (d) mostra-se quais são estes polígonos: de um total de 768 faces, 107 estão dentro deste grupo.

A idéia inicial dos *impostores com relevo* consiste em representar um objeto através da textura com relevo correta, para a posição corrente do observador. Para cada movimento de câmera ou do objeto, verifica-se se esta textura ainda é válida. Para fazer este teste, desenvolve-se uma métrica, no presente trabalho, onde se estima o erro a partir da quantidade de *texels* que se devem interpolar, escolhendo-se para isto um *texel* especial, denominado de ponto crítico. Esta métrica está detalhadamente discutida no capítulo 5 desta tese. Diferentemente da modelagem apresentada em 3.4.3, o objeto é representado por apenas um polígono e uma textura com relevo para cada instante.

Quando se verifica que uma textura com relevo está prestes a se tornar obsoleta, o sistema proposto gera uma nova textura, tendo em conta os dados correntes da câmera e do objeto, procurando por outro lado não prejudicar a visualização em andamento dos demais elementos da cena.

A criação destas novas texturas pode ser feita por *software* (CPU) ou por *hardware* (GPU). Gerá-la por *software* pode ser conveniente para sintetizar objetos em tempo real com algoritmos pouco adequados para a GPU ou para explorar e aproveitar melhor o tempo livre existente na CPU. Gerar as texturas com relevo por GPU pode servir para se fazer uma abordagem de renderização distribuída para placas gráficas.

Atualizar as texturas com relevo equivale a calcular um novo mapa de profundidade, um mapa de normais e um mapa de textura, dizendo a cor de cada *texel*. Na seção 8.8 discute-se com mais detalhes como isto é implementado.

Excetuando os casos de objetos com transparência, este método não possui restrições quanto ao tipo de objeto geométrico capaz de ser representado. Entretanto, modelos com geometria complexa fazem com que o tempo de validade da textura seja menor do que em objetos com topologia mais simples (ver figura 8.11). Este fato intrigante está explicado na seção 8.8. A figura 4.3 mostra diversas regiões onde, para cada uma, um único impostor é suficiente para representar o objeto.

No caso de se fazer o cálculo da nova textura com relevo por *software*, pode-se utilizar modelos de visualização impossíveis de serem implementados por *hardware*, tais como o ray-tracing, que exige recursão na sua implementação, ou volume-rendering, que exige um volume grande de dados. Também pode servir para tratar objetos cuja natureza geométrica seja inadequada para a GPU que se dispõe (objetos volumétricos, por exemplo).

figura 4.3 – Cada cor representa uma região onde uma mesma textura com relevo pode ser utilizada. Através da ilustração, pode-se notar que o tempo de vida de cada textura deste tipo é muito maior que o tempo de vida de um *sprite*.

Quanto à classificação apresentada na seção 2.3, pode-se dizer a respeito aos impostores com relevo que:

a) A autoria que os impostores requerem é equivalente às dos demais elementos geométricos da cena, já que o usuário não interfere no processo de criação e atualização das texturas com relevo. Pode-se, no entanto, permitir que alguns parâmetros sejam controlados: resolução máxima da textura, condição para que um objeto seja transformado num impostor com relevo, constante de erro tolerada pela métrica, algoritmo de visualização por software a ser usado para o objeto, etc.

b) Os graus de liberdade para objetos deste tipo correspondem a cinco, ou seja, permite-se um movimento livre pela cena, bem como uma variação do vetor de azimute e elevação do observador em relação ao mesmo.

c) A mudança da posição do observador é contínua para qualquer região do espaço, exceto para o caso do observador entrar na *bounding box* do elemento. Neste caso é necessário tratá-lo como geometria e não mais como textura com relevo.

d) A quantidade de amostras para visualizar o objeto corretamente se resume a uma textura para cada instante. Entretanto, conforme se apresenta no capítulo 8, havendo um gerenciamento de *cache* de texturas com relevo ou um sistema de previsão para gerar texturas futuras, mais de uma textura com relevo pode estar presente na memória.

e) Pode-se dizer que os impostores com relevo são do tipo *View Dependent Texture Mapping* e portanto estão dentro da área de modelagem baseada em imagens (*image-based modeling*). No entanto, a extração dos dados geométricos é feita de forma automática e transparente para o usuário, comportando-se, neste caso, como um objeto puramente imagem.

f) A variável tempo pode vir a ser inserida nos impostores com relevo, sendo necessária uma extensão da pesquisa neste sentido. Para tanto, deve-se ter uma sequência de texturas com relevo, que representem o objeto animado. Uma discussão mais detalhada sobre esta possibilidade pode ser encontrada no capítulo 9 e nas páginas 107-108 de Oliveira (2000a).

4.1.1 Multi-resolução para Impostores com relevo

Em algumas situações pode ser conveniente diminuir a resolução original do impostor com relevo, de forma a não ter que calcular *pixels* que são

posteriormente desperdiçados por problemas de amostragem. A figura 4.4 mostra um objeto sendo representado por um impostor que possui uma resolução de 256 x 256, sendo projetado numa tela na resolução de 800 x 600. Para a câmera em questão, o objeto ocupa aproximadamente 120 x 120 *pixels* da tela. Assim sendo, desperdiça-se tempo ao realizar o *warping* para todos os 256^2 *texels* da textura com relevo original. É conveniente, portanto, reescalar esta textura.

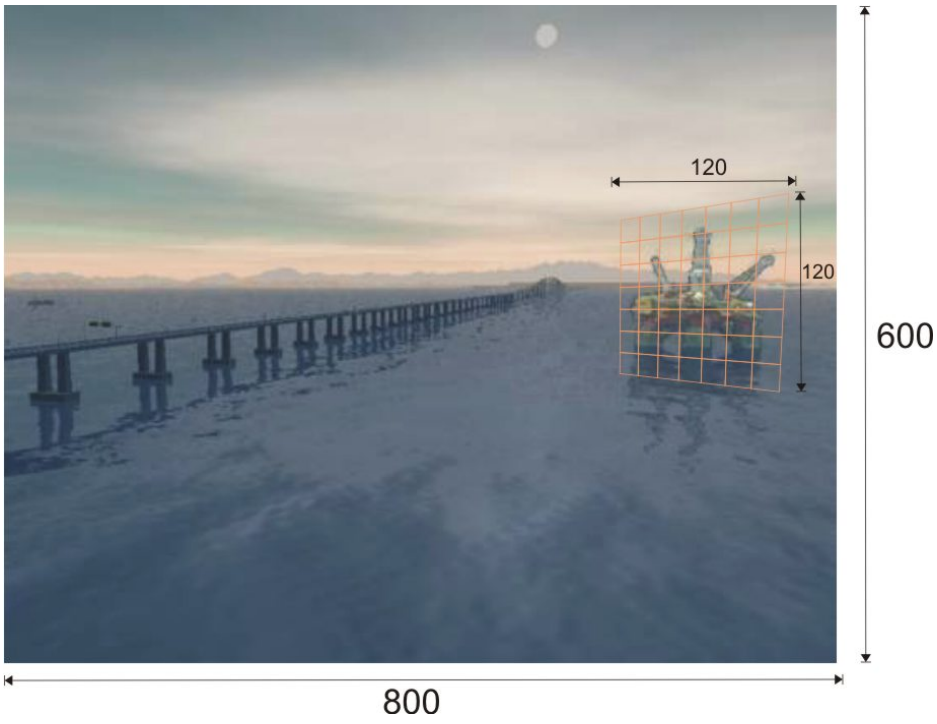


figura 4.4 – Neste exemplo, dos 256^2 *texels* do impostor, apenas 120^2 estão sendo aproveitados para a imagem final. Assim sendo, ao se fazer o *warping* da imagem no tamanho original, se está desperdiçando mais de ¾ do tempo de processamento com *texels* que não são vistos.

Para realizar este reescalonamento, é possível testar se um *texel* do impostor se tornou maior que um *pixel* da imagem gerada do observador, realizando-se o seguinte cálculo (ilustrado na figura 4.5):

$$\beta_{impostor} > K.\beta_{tela} \tag{4-1}$$

Sendo que

$$\beta_{impostor} = \frac{FOV}{resolução\_impostor} \text{ e } \beta_{tela} = \frac{FOV}{resolução\_tela} \tag{4-2}$$

A constante *K* permite que se coloque uma margem de erro, permitindo que o *texel* do impostor possa ser um pouco maior do que o *pixel* da tela, uma vez que a diferença pode ser imperceptível.

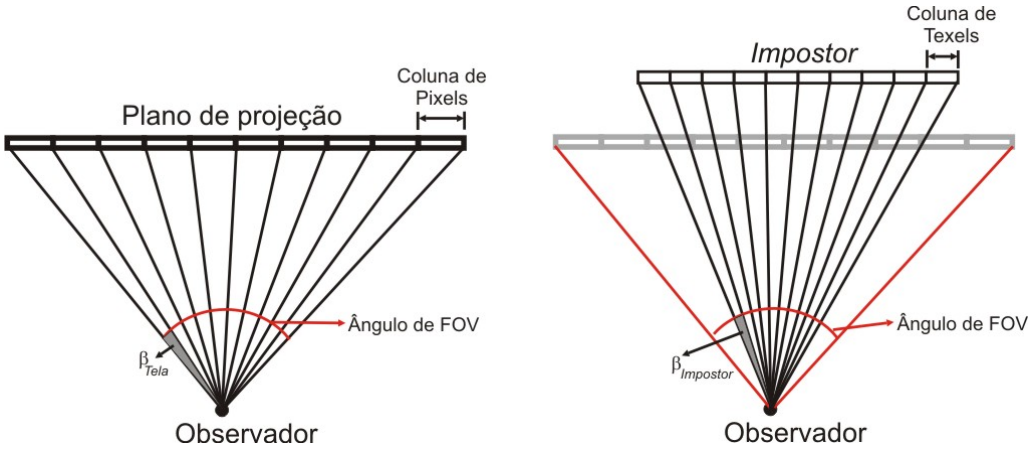


figura 4.5 – $\beta_{impostor}$ deve ser obtido a cada *frame*, para determinar se o impostor se tornou obsoleto em relação à resolução da tela e β_{tela} apenas em caso de alteração do FOV ou da resolução da imagem.

4.2 Discussão

Neste capítulo, após uma breve discussão sobre as possíveis formas de se representar modelos de objetos através de imagens, introduziu-se o conceito de impostores com relevo, como uma extensão na capacidade de modelagem através de texturas com relevo. Este método, diferentemente que a maioria das técnicas de modelagem baseada em imagem e da própria técnica texturas com relevo, apresentada inicialmente por Oliveira (1999), mantém dependência da geometria do objeto. Esta dependência se origina do próprio conceito de impostores, que consiste numa atualização dinâmica da textura. No capítulo seguinte apresenta-se uma forma de medir quando estas atualizações são necessárias. Quanto mais acurada for esta medida, menos texturas devem ser geradas e, portanto, mais eficiente é o impostor com relevo.

5 Medida de Erro para Impostores com relevo

Have you ever had a dream, Neo, that you were so sure was real? What if you were unable to wake from that dream? How would you know the difference between the dream world and the real world? (The Matrix)

5.1 Introdução

No pipeline padrão da GPU todos os objetos poligonais devem ser visualizados a cada frame, mesmo que nada tenha sido alterado em relação aos objetos e ao observador. Como foi discutido na seção 3.3, os impostores convencionais otimizam a visualização porque a geometria dos objetos que representam não precisa ser transformada e projetada a todo instante. De igual maneira, um impostor com relevo não precisa ser recalculado enquanto haja uma coerência visual, podendo assim ser reutilizado durante vários *frames* seguidos. Um impostor com relevo tem seu tempo de vida estendido devido a sua capacidade de corrigir a paralaxe. Divide-se, assim, a validade do impostor com relevo em 3 estágios, separados por dois momentos críticos:

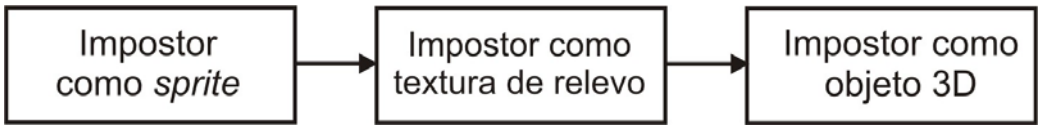


figura 5.1 – Estágios de um impostor com relevo.

Um impostor como *sprite* corresponde aos instantes em que o observador está dentro do limite de tolerância descrita pela equação de Schaufler (Schaufler, 1995). Este impostor é limitado pelo momento crítico em que esta tolerância deixa de ser válida. Neste caso o impostor usa a sua textura com relevo corrente para realizar um *warping* adequado, voltando logo em seguida a comportar-se como um impostor *sprite*. Este ciclo será realizado até o momento em que o *warping* começa a deformar o objeto que está sendo representado. Este momento é descrito

pela equação 5-7, que está descrita na seção 5.5. Neste caso, chega-se ao segundo momento crítico, momento este em que a textura com relevo do impostor não é mais válida, sendo necessário gerar uma nova textura, utilizando para isto os dados da posição do observador no ponto em que se dá este momento crítico e o modelo geométrico do objeto sendo representado.

Em qualquer um destes estágios, pode ocorrer do impostor deixar de ser válido por problemas de amostragem, o que na prática se dá devido a um movimento de aproximação do observador ao objeto. Este problema deve ser resolvido através de uma nova renderização do impostor numa resolução maior.

5.2 Criação do Impostor com Relevo

Num primeiro instante, deve-se determinar qual é a resolução inicial do impostor. Isto depende da distância em que o objeto se encontra em relação ao observador, pretendendo evitar que se desperdice tempo gerando *pixels* que depois são perdidos por motivo de amostragem.

Para determinar o tamanho do impostor a ser criado utiliza-se a seguinte aproximação proposta por Schaufler (1995):

$$resolução\_impostor = resolução\_tela \frac{Tamanho\_do\_objeto}{2 \times dist\_cam \times tg(FOV/2)} \qquad (5\text{-}1)$$

Onde *Tamanho_do_objeto* é a medida da *Bounding Box* paralelo ao plano de projeção da tela, dist_cam é a distância do objeto à câmera e FOV é o ângulo de abertura da lente da câmera.

5.3 Atualização do Impostor com Relevo

Como foi visto, existem basicamente dois movimentos do observador que fazem um impostor estático se tornar obsoleto, sendo portanto necessário um novo cálculo de *warping* ou de visualização para atualizá-lo:

1) O observador se aproximou do objeto, sendo necessário gerar uma textura com relevo para o impostor com uma resolução maior. No presente trabalho, denomina-se este movimento de translação normal ao plano da imagem (*transN*), i.e., na direção correspondente ao *Z* da câmera.

2) O observador se moveu num plano paralelo ao plano de projeção, podendo surgir erro de paralaxe do objeto mapeado. Este erro pode ser corrigido com um *warping* da textura com relevo ou através da geração de uma nova textura, no caso desta já não servir. No presente trabalho, denomina-se este movimento de translação paralela do observador (*transP*).

Para testar o primeiro caso, utiliza-se a aproximação de Schaufler (Schaufler, 1995): toma-se um dos extremos de um *bounding box* criado para o objeto sendo representado pelo impostor, conforme se pode ver na figura 5.2. A idéia é avaliar como estes pontos extremos se projetam no plano do impostor quando o observador se aproxima do mesmo. Tem-se que C_1 corresponde à posição inicial do observador e C_2 à posição que se deseja testar a necessidade de um acréscimo da resolução. Tendo-se que $\beta_{tela} = \frac{FOV}{resolução\_tela}$ e $\beta_{TransN} = B_1\hat{C}_2B_0$, sempre que $\beta_{transN} > \beta_{tela}$ deve-se refinar a textura do impostor. Esta correção pode ser realizada através de um cálculo de visualização apenas para os novos pixels que surgiram, reaproveitando os que já estavam renderizados, de maneira a não recalcular os *texels* já existentes.

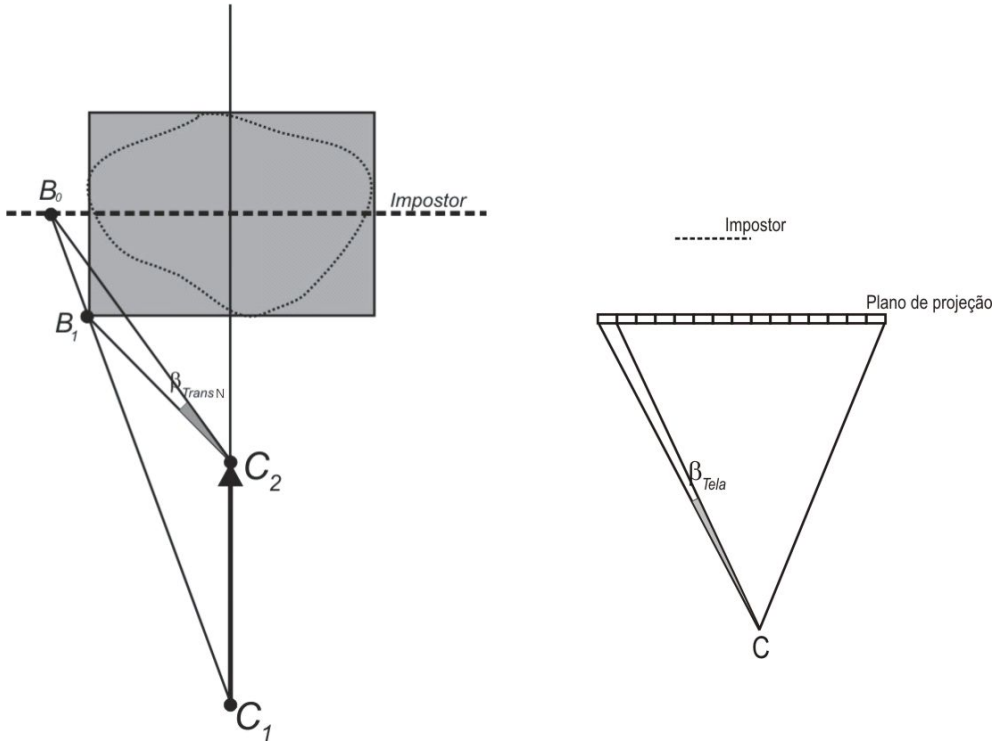


figura 5.2 –Medida de erro de Schaufler para movimento de aproximação do observador ao objeto representado pelo impostor (Schaufler, 1995).

O segundo caso requer mais cuidado: Antes de mais nada, através da aproximação de Schaufler, realiza-se um teste para saber se o impostor ainda é válido como *sprite* (momento crítico 1). Chama-se de ângulo de erro da translação paralela (β_{transP}) o ângulo criado pelos pontos extremos do *bounding box* quando o observador se move (figura 5.3). O deslocamento aparente do objeto devido ao movimento do observador é chamado de paralaxe. Caso $\beta_{transP} > \beta_{tela}$, o impostor não pode mais ser visto como o *sprite* que era, sendo necessário realizar um *warping* de sua textura para corrigir o paralaxe. Antes disto, porém, deve-se fazer uma estimativa para prever se o impostor a ser produzido pelo *warping* é válido, ou seja, se ao realizar o *warping* na sua textura, o *sprite* do objeto não possuirá excessivos erros acumulados provindos de uma extrapolação. Quando isto ocorrer, significa que se chegou ao segundo momento crítico, sendo necessário gerar uma nova textura com relevo para o objeto, utilizando a posição corrente da câmera.

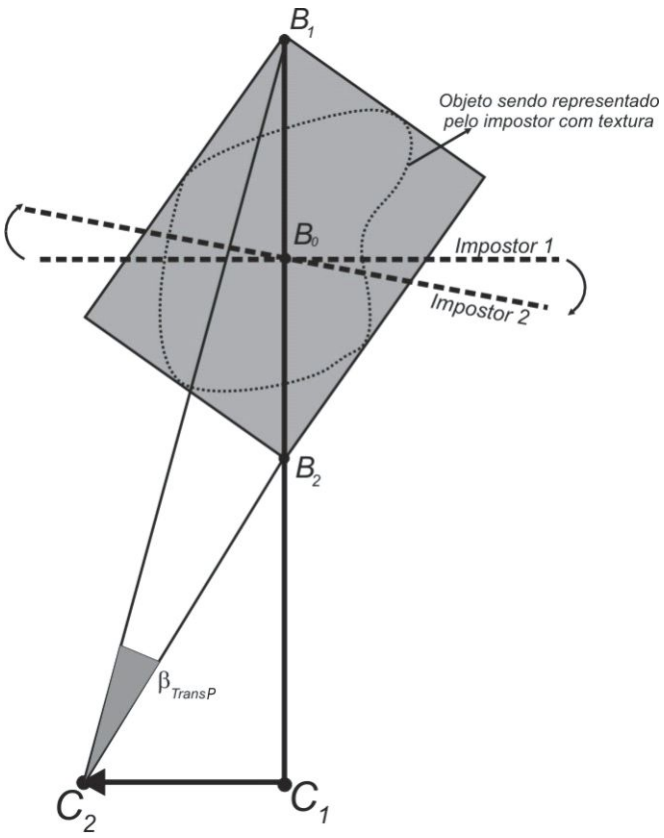


figura 5.3 – Medida de erro para movimento paralelo ao plano de projeção (Schaufler,1995).

Como foi visto na seção 3.4, a interpolação consiste no preenchimento dos buracos que surgiram entre *pixels* que eram vizinhos na figura original. Este

preenchimento gera num erro provocado pela falta de informação da imagem sobre o local. De um modo geral, quanto maior a descontinuidade da superfície que está sendo representada, maior é este erro. No presente trabalho desenvolve-se uma métrica baseada no erro acumulado nesta interpolação para uma textura com relevo, numa dada posição (figura 5.4).

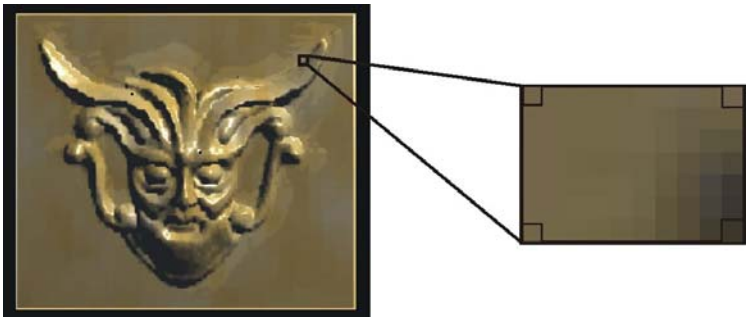


figura 5.4 – Dois *texels* vizinhos, ao se afastarem, criam um "buraco" que é preenchido por *texels* interpolados. Uma possível métrica de erro consiste em monitorar esta interpolação numa região crítica, que são pontos sujeitos a um erro maior.

5.4 Métrica de Erro Acumulado para Impostores com relevo

Para a estimativa desenvolvida é conveniente ver o impostor com relevo conforme ilustra a figura 5.5. Nela pode-se observar a seguinte equivalência de triângulos:

$$\frac{\Delta u}{B_u} = \frac{\partial(u_f, v_f)}{H} \tag{5-2}$$

onde B_u corresponde à distância entre a componente horizontal da projeção do ponto epipolar do impostor com relevo u_e até u_f, que é a projeção de $\dot{X}$ sobre o plano da textura com relevo e $H = h + \partial(u_f, v_f) = \vec{c}.\vec{f} + \partial(u_f, v_f)$, sendo estes parâmetros os mesmos já descritos na seção 3.4.

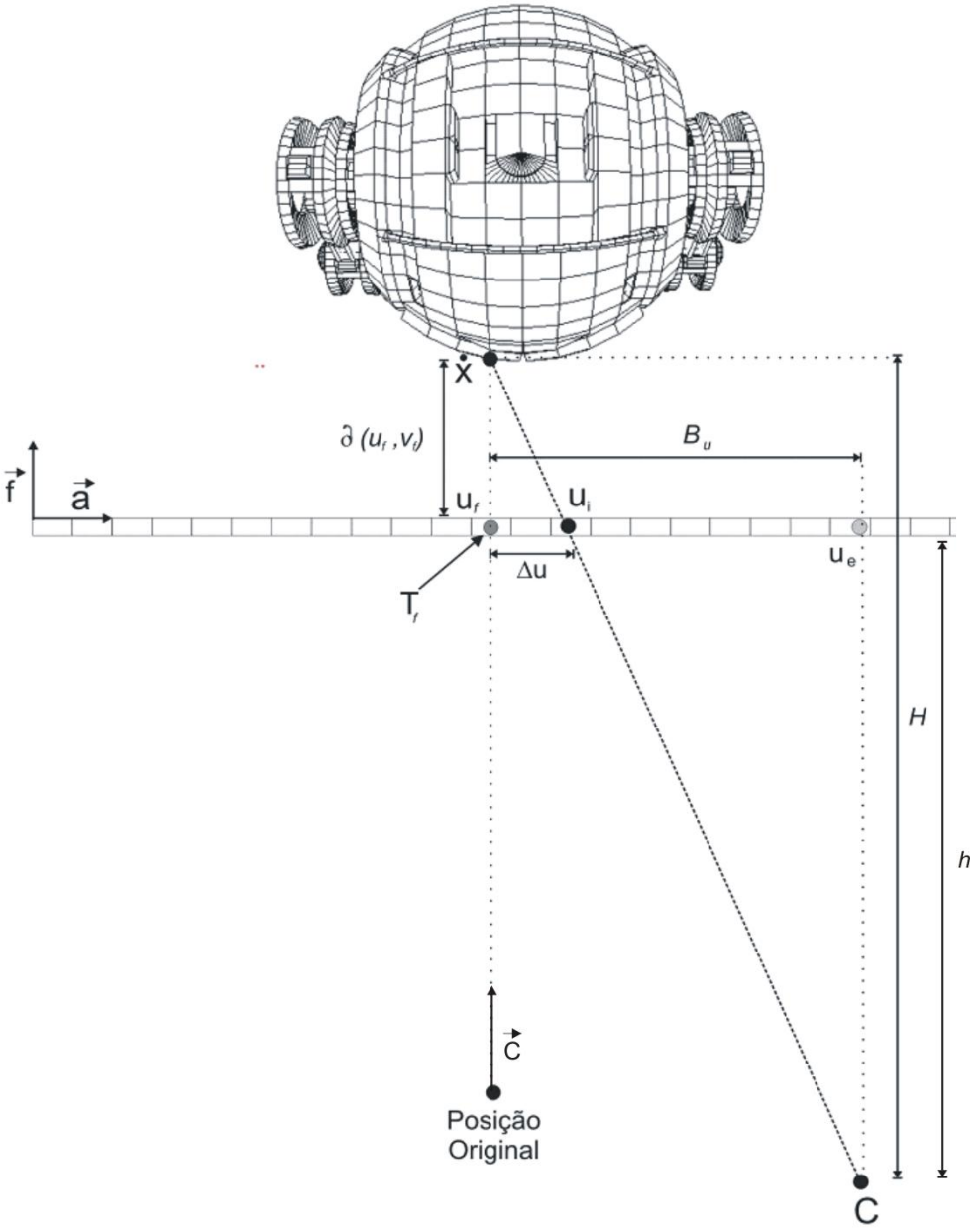


figura 5.5 - O *warping* de uma textura com relevo pode ser descrito da seguinte forma: para cada *pixel* da imagem T_f, realiza-se um deslocamento sobre as colunas da imagem Δv e sobre as linhas da imagem Δu. Estes deslocamentos dão a nova posição v_i e u_i correspondente ao *pixel* para a posição da câmera *C* a partir da sua posição original v_f e u_f, fazendo-se $v_i = v_f + \Delta v$ e $u_i = u_f + \Delta u$. A presente figura ilustra o deslocamento horizontal Δu.

Assim sendo, tem-se que $B_u = u_e - u_f$. Pode-se interpretar através da equação (5-2) que o deslocamento Δu a ser aplicado sobre o *pixel* de $\dot{X}$ é inversamente proporcional à distância de *C* ao plano do impostor, para valores de $\partial(u_f, v_f) \neq 0$.

Desta forma, chega-se à seguinte equação (Oliveira, 2000):

$$\Delta u = \frac{(u_e - u_f)\partial(u_f, v_f)}{\vec{c}.\vec{f} + \partial(u_f, v_f)} \tag{5-3}$$

De forma semelhante, pode-se chegar à seguinte formulação para o deslocamento vertical a ser aplicado sobre o *texel*:

$$\Delta v = \frac{(v_e - v_s)\partial(u_f, v_f)}{\vec{c}.\vec{f} + \partial(u_f, v_f)} \tag{5-4}$$

Intuitivamente, pelas equações (5-3) e (5-4) pode-se notar que quanto maior for o deslocamento do observador *C* em relação a posição original *C'*, maior tendem a ser os valores absolutos de Δu e Δv para um *texel* do impostor e portanto maior a tendência a serem criados *texels* interpolados.

Assim, sejam T_1 e T_2 dois *texels* pertencentes à mesma textura com relevo, tal que T_1 e T_2 são vizinhos e que $|\partial(u_1, v_1) - \partial(u_2, v_2)|$ é máximo para qualquer par nestas condições. Então T_1 e T_2 são fortes candidatos para que $|(u_{f_1} + \Delta u_1) - (u_{f_2} + \Delta u_2)|$ seja máximo em toda a textura.

Esta afirmação pode ser facilmente comprovada com o auxílio da figura 5.6, onde $\Delta h = \partial(u_1, v_1) - \partial(u_2, v_2)$, e pode ser interpretado intuitivamente da seguinte maneira: a região onde pode ocorrer maior descontinuidade no campo de profundidade da textura com relevo é o local onde se acumula o maior erro de interpolação durante o *warping* (figura 5.6).

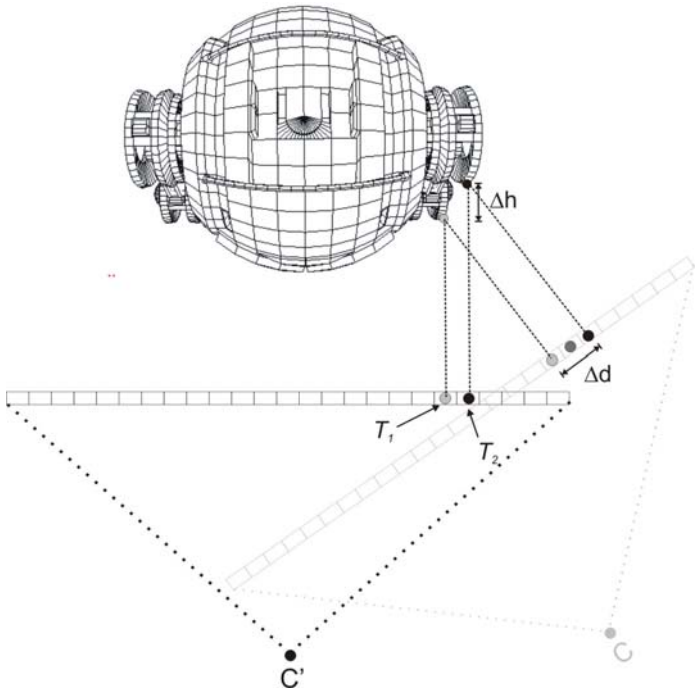


figura 5.6 – Δ*h* corresponde ao valor da maior descontinuidade para a textura com relevo. Neste caso, Δ*d* representa a distância entre os *pixels* da textura depois de sofrerem o *warping*. De acordo com a afirmação apresentada, para a posição de câmera *C,* Δ*d* é o maior valor que pode ocorrer de distanciamento entre *pixels* que são vizinhos no impostor original.

5.5 Métrica de Erro baseado no ponto crítico do Impostor com Relevo

Denomina-se ponto crítico de uma textura com relevo o par de *texels* T_1 e T_2 descritos na afirmação acima. Uma das métricas de erro utilizada neste trabalho consiste em prever como é o erro no ponto crítico, antes de realizar o *warping* para imagem completa. Caso o erro previsto esteja acima de um determinado valor, ao invés de ser feito o *warping* o sistema deve requerer a geração de uma nova textura para o impostor com relevo, apropriado para a posição corrente do observador. O erro deste ponto crítico (Δ*d* na figura 5.6), com relação ao deslocamento horizontal, pode ser calculado da seguinte forma:

$$Erro_u = |(u_{f_1} + \Delta u_1) - (u_{f_2} + \Delta u_2)| = |u_{f_1} - u_{f_2} + \Delta u_1 - \Delta u_2| \qquad (5\text{-}5)$$

Como $u_{f_1} - u_{f_2} = 1$, pois são vizinhos, tem-se que:

$$Erro_u = |1 + \Delta u_1 - \Delta u_2|$$

Substituindo-se Δu_1 e Δu_2 pela equação 5-3 chega-se a

$$Erro_u = \left| 1 + \frac{(k_1 - u_{f_1} k_3)\partial(u_1, v_1)}{1 + k_3 \partial(u_1, v_1)} - \frac{(k_1 - u_{s_2} k_3)\partial(u_2, v_2)}{1 + k_3 \partial(u_2, v_2)} \right|$$

Ou, em termos da equação 5-2:

$$Erro_u = \left| 1 + \frac{\partial(u_1, v_1) B_{u_1}}{h + \partial(u_1, v_1)} - \frac{\partial(u_2, v_2) B_{u_2}}{h + \partial(u_2, v_2)} \right| = \left| \frac{h(K + \Delta h(u_e - u_{f_1}))}{(K - \Delta h)K} \right|$$

Onde $K = h + \partial(u_1, v_1)$. Isto permitiria armazenar em cada texel o maior valor de Δh entre este texel e todos os seus vizinhos.

Como $B_u = u_e - u_f$, pode escrever-se que

$$Erro_u = \left| 1 + \frac{\partial(u_1, v_1)(u_e - u_{f_1})}{h + \partial(u_1, v_1)} - \frac{\partial(u_2, v_2)(u_e - u_{f2})}{h + \partial(u_2, v_2)} \right|$$

Até este ponto assume-se apenas um movimento horizontal. De forma semelhante se pode chegar à seguinte equação:

$$Erro_v = \left| 1 + \frac{\partial(u_1, v_1)(v_e - v_{f_1})}{h + \partial(u_1, v_1)} - \frac{\partial(u_2, v_2)(v_e - v_{f2})}{h + \partial(u_2, v_2)} \right| \quad (5\text{-}6)$$

Finalmente, a previsão do erro para o ponto crítico da textura com relevo é calculado da seguinte forma:

$$Erro = \sqrt{Erro_u^2 + Erro_v^2}$$

Porém, como este erro é apenas usado de forma comparativa, economiza-se uma operação de extração de raiz fazendo-se

$$Erro \cong Erro_u^2 + Erro_v^2 \quad (5\text{-}7)$$

Esta estimativa de erro é adequada para impostores que representam objetos com superfície contínua, onde o erro do ponto crítico não é muito maior do que o erro médio de todos os pontos.

5.6 Métrica de Erro baseado em amostragem de pontos críticos

Caso o objeto possua alguma descontinuidade nos valores de profundidade dos *texels*, o método proposto sobreestima o erro de toda a textura, dando-lhe um tempo de vida muito curto apenas por causa de uma região. É conveniente para estes casos não colocar todo o sistema dependente de apenas um ponto crítico, mas sim de um conjunto destes.

O método proposto sugere que a textura com relevo seja dividida em *N* x *M* regiões. Para cada uma destas regiões existe um ponto crítico baseado nos mesmos critérios da seção 5.5, conforme ilustra a figura 5.7.

O critério para se gerar uma nova textura com relevo para o impostor será o de que a somatória do erro de cada ponto crítico ultrapasse um valor de erro estipulado.

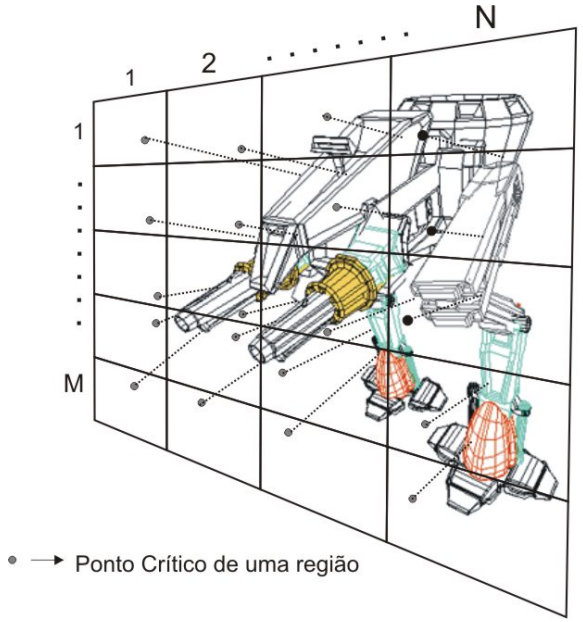


figura 5.7 – Nesta medida de erro uma textura com relevo é dividida em *N* x *M* regiões, tomando-se um ponto crítico para cada região.

5.7 Discussão

A importância da métrica consiste sobretudo em garantir que a textura com relevo dure o máximo possível, sem prejudicar demasiadamente o resultado da visualização. Quanto mais tempo durar a textura, melhor. Este tempo economizado pode ser utilizado para processar as componentes geométricas da cena com mais detalhes, dedicar mais tempo a uma visualização correta do impostor através de *software* e eventualmente realizar a previsão das futuras texturas com relevo com maior precisão. No próximo capítulo é apresentado como se encaixa a GPU no processo de cálculo e visualização destes objetos.

6 A GPU

These machines are keeping us alive, while other machines are coming to kill us. Interesting, isn't it? Power to give life, and the power to end it. (The Matrix)

6.1 Introdução

Há poucos anos, falar em programar o pipeline do hardware gráfico limitava-se a ativar/desativar alguns estados disponíveis na placa. Atualmente, tornaram-se populares placas que podem executar programas inteiros antes de processar um vértice ou um *pixel*, permitindo que o desenvolvedor não esteja mais preso aos estados pré-existentes no hardware.

Diferentemente que na CPU, onde há apenas um processador programável, as GPU com suporte a programação de pipeline podem possuir dois tipos de processadores programáveis: O processador de vértices e o processador de fragmentos[*].

O Processador de vértices recebe como entrada um vértice juntamente com seus atributos: normalmente a posição, cor, coordenada de texturas e a sua normal. Para cada um destes vértices, o processador executa uma determinada sequência de instruções, que consiste no *vertex shader*. Este pequeno programa fará alterações nos parâmetros do vértice, de acordo com a sua lógica, possibilitando criar efeitos complexos em tempo real. Ao terminar de processar o *vertex shader*, o vértice é encaminhado para o restante do pipeline gráfico. Efeitos comumente implementados desta forma são texturas procedurais, movimentos complexos de vértices, manipulação dinâmica das cores, etc.

Para se calcular a iluminação de uma malha 3D por hardware, normalmente calcula-se a iluminação que corresponde a cada um de seus vértices durante o

[*] Também é chamado de processador de *pixels*

estágio de geometria. No estágio de rasterização, o hardware realiza uma interpolação dos fragmentos que estão no interior deste polígono (figura 6.1), dando a impressão de que foi calculada a iluminação para cada *pixel* do interior.

Este processo de interpolação é responsável por uma série de limitações impostas ao modelo de iluminação utilizado para tempo real, tal como impossibilidade de especularidade e de *bump-mapping*, uma vez que estes precisam de um cálculo *pixel* a *pixel*, ou seja, um cálculo específico para cada *pixel*.

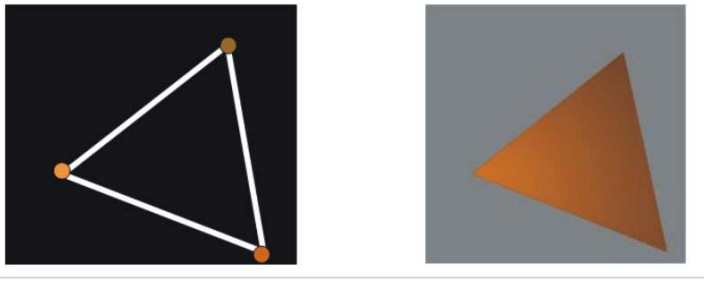


figura 6.1 – A iluminação de um polígono normalmente é feita pela interpolação das cores de cada um dos vértices que o compõem. Esta interpolação dará a ilusão de que cada *pixel* do interior foi iluminado individualmente, quando na verdade apenas os vértices o foram.

No pipeline gráfico, o estágio de rasterização processa um fragmento individualmente, calculando sua cor através de uma interpolação da cor dos vértices do polígono a que pertence, bem como das coordenadas da sua textura. De forma semelhante ao *vertex shader*, um *pixel shader* consiste num conjunto de instruções que são executadas para cada fragmento, antes que este seja plotado na tela. A saída de um programa deste tipo consiste numa cor. Tal recurso permite que alguns efeitos de iluminação, antes impossíveis para visualização em tempo real, possam ser eficientemente implementados.

Normalmente, um efeito requer que haja pelo menos um *vertex shader* e um *pixel shader*. Este trabalho implementa um cálculo de iluminação *pixel* a *pixel*, utilizando-se o mapa de normais do impostor com relevo em linguagem Cg.

6.2 GPU's e Renderização Baseada em Imagens

Na abordagem de *ibr* que se está fazendo o problema central consiste em realizar *warping* (deformações) nas imagens. Foram desenvolvidos alguns trabalhos de destaque para criar arquiteturas de *hardware* dedicadas a esta operação, tais como o *Warp Engine* (Popescu, 2000) e (Popescu, 2001) e o *Talisman* (Toborg, 1996).

O *Warp Engine* consiste numa arquitetura de *hardware* que propõe a utilização de imagens como primitivas, ao invés de polígonos. Estas imagens devem ser capturadas ou geradas juntamente com a informação da profundidade de cada pixel, para poder assim seguir a manipulação de *ibr* através da equação de 3D *image warping* de McMillan.

O *Talisman* consiste numa implementação onde as primitivas são camadas de imagens: cada uma destas camadas possui informação de profundidade e receberá um tratamento separado. Movimentos são gerados a partir de deformações individuais para cada camada e a imagem final é gerada através de uma composição de todas as camadas.

Na prática, entretanto, estes sistemas não têm se popularizado: em parte devido à seu propósito muito específico. Por outro lado, placas com arquitetura programável mais genérica vêem ganhando espaço no mercado e tornando-se muito baratas. No presente trabalho desenvolve-se uma adaptação para uma plataforma comum de CPU-GPU para um sistema com suporte a *ibr* em seu *pipeline gráfico*.

Como os *shaders programs* possuem como entrada um único vértice ou um *pixel* de cada vez, o programa não é capaz de ver um modelo ou uma imagem como um todo, não sendo possível uma análise de contexto da imagem.

A GPU recebe como entrada cada um dos *pixels* da tela e para o mesmo executa-se um pequeno programa (*fragment program*) que retornará uma nova cor para pintar tal *pixel*. Assim sendo, uma possível forma de se realizar o *warping* de uma imagem, seria implementar o mapeamento inverso do *warping*: dado um *pixel* da tela, determina-se qual é o *texel* de cada imagem, com o *warping* aplicado sobre ela, que lhe corresponde. Note-se que, no caso de haver mais de uma imagem, um teste de profundidade se faz necessário. Afirma-se que este procedimento não é adequado para ser implementado nos *hardwares* atuais,

devido à complexidade que este mapeamento inverso pode vir a ter e a escassez de recursos que se dispõem no âmbito da GPU. Isto será comprovado na seção 6.5.

Neste trabalho propõe-se que sejam criados dois *pipelines* paralelos: um controlado pela CPU, que se encarrega por fazer o *warping* para as imagens e outro controlado pela GPU, que se encarrega de plotar polígonos. A CPU envia o resultado do *warping* para o dispositivo gráfico na forma de uma textura.

6.3 A Linguagem Cg

Cg (*C for Graphics*) é uma linguagem de programação para GPU's de autoria da NVidia em parceria com a Microsoft, na tentativa de abstrair a linguagem de máquina necessária para se programar *shaders* para o *hardware* gráfico. O desenvolvedor é capaz de escolher a API que é alvo deste código (OpenGL ou Direct3D) (Fernando, 2003).

O Cg implementa a sintaxe, funções e operadores equivalentes ao C, permite suporte a condicionais e gera um código otimizado para a GPU.

Um *vertex* e *pixel* Cg *program* descrevem um *pass*: uma realização de rasterização completa no *frame buffer*. Entretanto pode-se agrupar alguns programas num CgFX, capaz de executar *multi-passes* para uma visualização.

Enquanto numa aplicação padrão tem-se apenas uma instância do programa para cada CPU, um programa para GPU é executado várias vezes, uma para cada componente: vértices ou *pixels*, podendo facilmente ser paralelizado por placas que tenham mais de um processador de vértice e/ou *pixel*.

6.4 Cálculo de Iluminação *Per-Pixel* utilizando *pipeline* programável e mapa de normais

A iluminação dos impostores com relevo é feita através da GPU, utilizando-se o mapa de normais disponível.

O *vertex shader* para este cálculo é simples e resume-se a realizar a projeção dos vértices para o plano de projeção da câmera:

```
void main_vertex_shader (float4 position : POSITION          // posição original do vértice
      float2 texCoord : TEXCOORD0                 // coordenada da textura do vértice
```

```
        out    float4 oPosition : POSITION          // vértice projetado
        out    float4 objPosition: TEXCOORD0        // Posição original do vértice
        out    float2 oTexCoord: TEXCOORD1          // coord.  textura no vértice projetado
        uniform float4x4 modelViewProj              // matriz da câmera concatenada com a
                                                    // de projeção
{
        // Calcula a projeção do vértice no plano da câmera
        oPosition = mul (modelViewProj, position);
        objPosition = position;     // Armazena a posição do vértice antes da projeção
        oTexCoord = texCoord;    // Mantém a mesma coordenada de textura
}
```

O cálculo de iluminação propriamente dito reside no *pixel shader*. Neste, para um determinado *pixel*, testa-se a validade do *texel*, verificando o valor de profundidade do mesmo. Caso seja um valor correspondente a 255, o *texel* é vazio e não se deve calcular nenhuma iluminação para o mesmo, pois não há nenhuma superfície em tal ponto. Caso seja válido, busca-se a sua normal correspondente no mapa de normais e a sua cor no mapa de textura. Feito isto, calcula-se a iluminação utilizando o algoritmo de *Phong*. Maiores detalhes para este cálculo podem ser encontrados em (Fernando, 2003) e em (Fonseca, 2004).

```
void main_Pixel_Shader (float4 position : TEXCOORD0,
    float2 texCoord : TEXCOORD1,          // A posição do pixel e a  sua coordenada de textura
                                          // são resultados da interpolação para o pixel
                                          // corrente, provindos do vertex shader
    out float3 oColor : COLOR,            // Cor resultante do cálculo de iluminação
    uniform float3 globalAmbient,         // Luz ambiente da cena
    uniform float3 lightPosition,         // Posição da fonte de luz na cena
    uniform float3 lightColor,            // Cor da fonte de luz
   uniform float3 eyePosition,            // Posição do observador em coordenadas da tela
   uniform float3 ka,                     // Componente ambiente do material do objeto
   uniform float3 ks,                     // Componente especular  do material do objeto
   uniform float shininess,               // Índice de especularidade do material do objeto
   uniform sampler2D normalMap,           // Mapa de normais com profundidade
   uniform sampler2D textureMap)          // Mapa de cores da textura com relevo
{
// Obtêm-se as normais correspondentes ao pixel corrente
float4 normalTexel = tex2D (normalMap, texCoord);
if (normalTexel.w == 1.0) // Testa-se se o pixel é válido, ou seja, se sua profundidade
```

```
        discard;            // é diferente de 255. Caso seja invalido termina o shader
else
{       // Busca-se o texel do mapa de texturas para o pixel
        float3 textureTexel = tex2D (textureMap, texCoord);
        // Inicialização dos parâmetros do material e da luz para o cálculo de iluminação
        float3 p = position.xyz;
        float3 n = normalize (normalTexel.xyz);
        TLight light;
        light.position = lightPosition;
        light.color = lightColor;
        TMaterial material;
        material.ka = ka;
        material.kd = textureTexel;
        material.ks = ks;
        material.shininess = shininess;
        // Cálculo de iluminação para o pixel
        oColor.xyz = lighting (material, light, globalAmbient, p, n, eyePosition);
        oColor.w = 1.0;
        }
}
```

6.5 Implementação de Texturas com Relevo em Hardware

Como foi visto, uma das principais contribuições de Oliveira (2000) consiste em decompor a equação de 3D *image warping* de McMillan numa operação unidimensional sobre uma imagem (etapa de *pre-warping*) seguida de uma operação de texturização tradicional. Entretanto, para que este processo possa ser implementado por *hardware*, como foi discutido em 5.2, devido à natureza do funcionamento do processador de fragmentos, deve-se fazer o mapeamento inverso do que é proposto na equação original de 3D *Image Warping*: dado um *texel* (u_f, v_f) determinar qual o seu resultado em relação posição corrente da câmera C_d. Esta equação inversa é dada por (Oliveira, 2000):

$$u_f = u_i(1 + k_3 disp(u_f, v_f)) - k_1 disp(u_f, v_f)$$

e

$$v_f = v_i(1 + k_3 disp(u_f, v_f)) - k_1 disp(u_f, v_f) \qquad (6\text{-}1)$$

Observando-se (6-1) percebe-se que a equação exige a profundidade $disp(u_f,v_f)$. O problema é que este valor não pode ser descoberto sem uma inspeção sobre o modelo representado pela textura com relevo. Pior ainda: pode haver mais de uma solução para tal componente, como ilustra a figura 6.2.

Assim sendo, para saber qual é o ponto que está na frente de todos, é necessário pesquisar a profundidade de um conjunto de projeções vizinhas a v_i. Em (Oliveira, 2000) apresenta-se um critério para limitar esta pesquisa apenas para uma pequena parte da imagem.

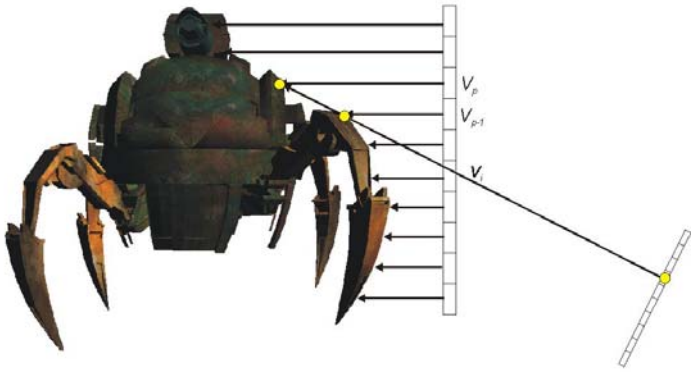


figura 6.2 – Dois *texels* distintos são mapeadas para o mesmo ponto no processo de *pre-warping* inverso.

Policarpo (2003) implementa o método de mapeamento inverso de texturas com relevo tal como está descrito, fazendo com que a equação (6-1) esteja inserida no *pipeline* de *hardware* da GPU, através de um *pixel shader* feito em Cg. Nesta implementação, a busca da profundidade é feita criando-se "fatias" do modelo: para um cálculo preciso é necessária uma subdivisão detalhada do volume envolvente do objeto. Entretanto, como a interseção deve ser calculada para cada uma destas fatias, quanto maior a precisão que se deseja obter, mais fatias deve haver e mais lento se torna o processo.

A tabela 6.1 mostra o desempenho obtido com esta implementação, para um objeto sendo representado por uma textura com relevo aplicada a um polígono. Este objeto está ilustrado na figura 6.3. O *hardware* utilizado para este teste foi um Pentium IV, com 512 MB de memória RAM e uma GPU da Nvidia Gforce FX 5600, com 128 MB de memória de vídeo DDR.

Número de fatias	Taxa de FPS	Imagem correspondente (figura 5.3)
8	17	(a)
16	7	(b)
32	3	(c)
64	1	(d)
128	0,4	(e)

tabela 6.1 – Desempenho obtido para o modelo da figura 5.3, com diversos números de fatias.

(a) (b) (c)

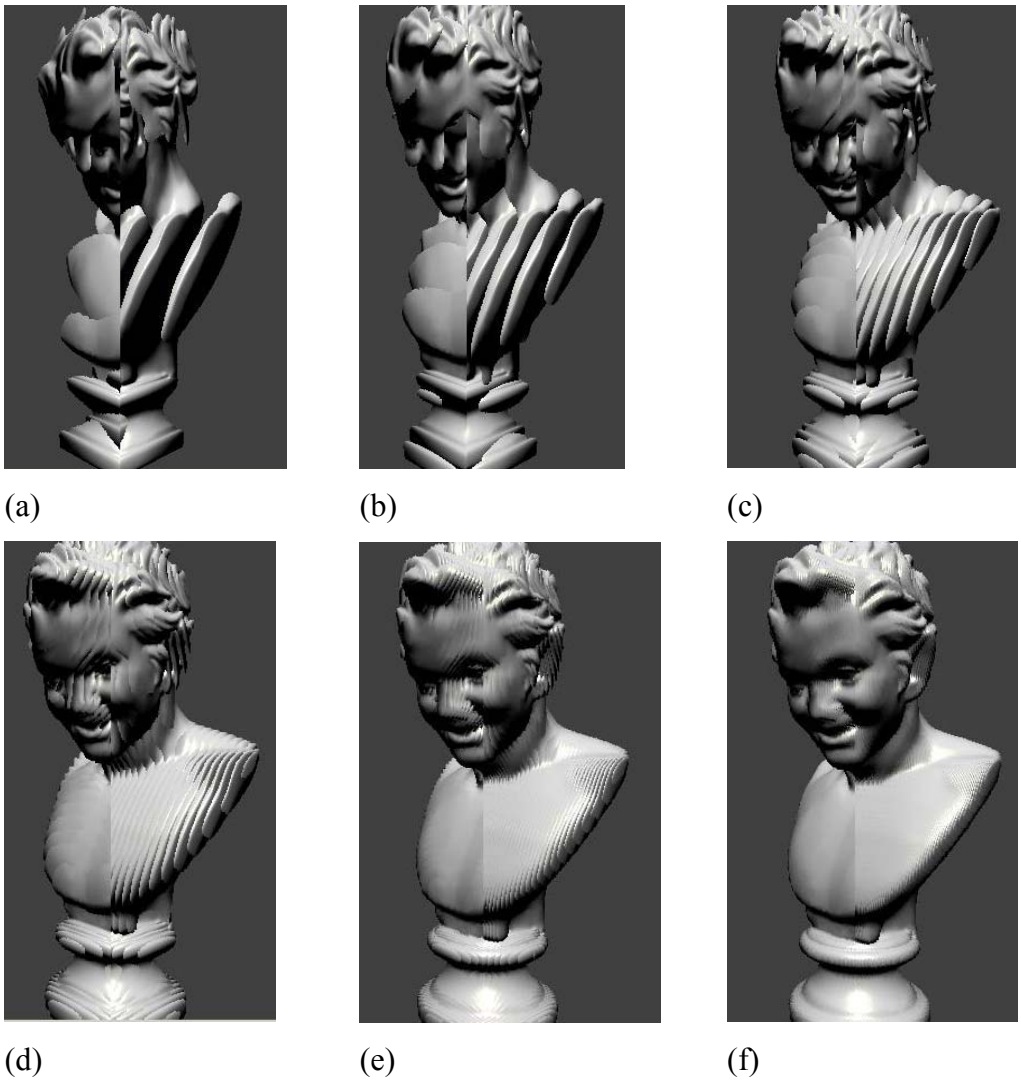
(d) (e)

figura 6.3 – Resultados obtidos com diversas resoluções para as fatias usadas pelo *shader* implementado na GPU: 8 fatias (a), 16 fatias (b), 32 fatias (c), 64 fatias (d) e 128 fatias (e). A tabela 5.1 contém o desempenho obtido em cada caso.

Para o mesmo *hardware*, a tabela 6.2 mostra o desempenho obtido para a implementação do sistema de representação de um objeto por um paralelepípedo de texturas com relevo, conforme discutido em 3.6.3.

Número de fatias	Taxa de FPS	Imagem correspondente (figura 5.4)
8	17	(a)
16	7	(b)
32	3	(c)
64	1	(d)
128	0,4	(e)
256	0,2	(f)

tabela 6.2 – Desempenho obtido para o modelo da figura 5.4, com diversos números de fatias para a modelagem por paralelepípedos de texturas com relevo.



(a) (b) (c)

(d) (e) (f)

Figura 6.4 – Resultados obtidos com diversas resoluções para as fatias usadas pelo *shader* implementado na GPU: 8 fatias (a), 16 fatias (b), 32 fatias (c), 64 fatias (d), 128 fatias (e) e 256 fatias (f). A tabela 5.2 contém o desempenho obtido em cada caso.

Pode-se reparar pelos resultados obtidos que, para representar um objeto por completo, torna-se necessário utilizar pelo menos 128 fatias de forma que não ocorram deformações. Entretanto, pela tabela 6.2, pode-se afirmar que a taxa de FPS obtida nesta configuração não é aceitável para tempo real. Ressalta-se todavia que esta complexidade é independente da complexidade geométrica do objeto representado, mas apenas da resolução da imagem e do número de fatias. É possível que ao longo dos próximos anos esta taxa aumente significativamente, especialmente com o aumento do paralelismo das GPU´s. Mesmo assim, uma afirmação que se faz neste trabalho consiste em que o processo de *warping* para aplicações de *ibr* não é adequado para uma GPU de programação genérica, mas apenas em arquiteturas dedicadas, como as apresentadas em (Popescu 2000, Toborg 1996). Afirma-se ainda que com as tecnologias atuais, é mais conveniente resolver esta etapa dos algoritmos de *ibr* através de *threads* paralelos de CPU.

Em (Fujita, 2002) se apresenta outra abordagem para a implementação em *hardware* para mapeamento de texturas com relevo*:* para cada alteração da câmera, ao invés de se gerar uma imagem com o *pre-warping* da textura*,* gera-se um mapa de *off-set*: cada elemento deste mapa contém a informação da coordenada de onde se encontra o *texel* que lhe corresponde na imagem fonte após o cálculo de 3D *image warping*. Para esta implementação, utiliza-se a função *OFFSET_TEXTURE_2D*, que é um *texture shader*. Esta função, que está implementada em *hardware* na maioria das placas atuais, permite pintar um *texel* de uma determinada textura buscando a cor num *texel* de outra, com um determinado *offset* (dado pelo *offset map*).

Como o mapa de *offset* é gerado pela CPU, não se pode dizer com propriedade que o *warping* está sendo feito por *hardware*. Este método é apropriado para calcular efeitos de iluminação num objeto sendo representado por mapas de relevo, pois o mesmo mapa de *offset* gerado pode ser reutilizado para determinar outros elementos do *pixel*, tal como a sua normal, dentro de um mapa de normais*,* ou o seu reflexo dentro de um mapa de reflexos.

6.6 Simulação de Shading para sprites sem normal-maps

A iluminação apresentada em 6.4 leva em conta que existe um mapa de normais previamente calculado e armazenado. Caso este mapa não exista, devido

à dificuldade para sua aquisição, pode-se criar um mapa aproximado, realizando-se uma interpolação, como se propõe em (Clua, 2001).

6.7 Discussão

Um sistema composto por CPU-GPU pode ser visto por si só como uma máquina paralela, fazendo com que cada componente fique a cargo de uma parte da visualização mais adequada à sua arquitetura. Dentro do âmbito da CPU, também é possível criar outros graus de paralelismo, caso se disponha de mais processadores. O capítulo 7, após realizar um breve resumo sobre os diversos tipos de abordagens paralelas, apresenta alguns métodos para distribuir ou paralelizar a resolução dos problemas envolvidos nos impostores com relevo.

7 Processamento Paralelo

Yes, of course, who has time? Who has time? But then if we do not ever take time, how can we ever have time? (The Matrix)

7.1 Introdução – Classificação de Sistemas Paralelos

Diversas aplicações procuram lançar mão de multi-processamento para obter melhor performance, a começar pelos próprios sistemas operacionais ou serviços de busca para Internet. Os objetivos do paralelismo podem ser categorizados como:

- Permitir que sistemas munidos de mais de um processador possam acelerar a resolução de um problema de natureza paralelizável;
- Poder executar vários processos simultaneamente (define-se neste caso, segundo Carissimi (2001), que processo é um programa em execução)
- Otimizar o tempo ocioso (*idle time*) das CPU's;

A divisão de processos pode ocorrer tanto ao nível de várias aplicações independentes sendo executadas simultaneamente, bem como uma mesma aplicação dividida em vários *threads* (*thread level paralelism* -TLP) (Marr, 2002).

De acordo com Andrews (2000), toda aplicação paralela pode ser categorizada como sendo de uma das seguintes formas:

- Sistema *Multi-Threaded*: um programa que contém mais processos do que o número de processadores disponíveis. Estes sistemas têm como objetivo gerenciar múltiplas tarefas independentes. Um sistema operacional se encontra dentro desta categoria;
- Sistema Distribuído: vários processos são executados em várias máquinas, conectadas entre si de alguma forma. Estes processos se comunicam mutuamente através de mensagens;

- Computação Paralela: Um programa se divide em vários processos para resolver o mesmo problema num tempo menor. Geralmente haverá o mesmo número de processos quanto o de processadores.

Também pode-se categorizar uma aplicação paralela de acordo com o modelo de programação adotado (Andrews, 2000):

- Variáveis compartilhadas: os processos trocam informações entre si através de uma memória compartilhada, numa área comum a todos;
- Troca de mensagens: Não há disponibilidade de uma área comum de memória, ou isto não convém para a aplicação (são máquinas diferentes e separadas, por exemplo). Para tanto, a comunicação entre os processos é feita através de envio e recebimento de mensagens;
- Dados Paralelos: cada processo executa as mesmas operações sobre diferentes partes do dado distribuído. Este modelo também é conhecido como *Single Instruction Multiple Data* (*SIMD*).

Em aplicações de visualização para tempo real, são inúmeros os trabalhos que paralelizam as tarefas. Merecem destaque (Kempf 1998, Igehy 1998).

7.2 *Multi-threading* e *Hyper-threading*

Um *thread* pode ser entendido também como um processo, que pode pertencer a uma aplicação ou a aplicações completamente independentes. Sistemas com apenas um processador podem gerenciar vários *threads* simultaneamente através de métodos de *Time Slice*, concedendo para cada processo uma fatia de tempo da CPU, sendo que este tempo não precisa ser necessariamente o mesmo para cada um. Além disso, cada processo pode conter um nível de prioridade: *threads* com alta prioridade podem interromper *threads* com menor prioridade. Entretanto, por melhor que seja o gerenciamento dos *threads* por parte da aplicação ou do sistema operacional, o *Time Slice* sempre pode trazer latências, devido a compartilhamento de *cache* ou dificuldade de gerenciamento entre o trânsito de processos.

Há muitos anos têm-se adotado paralelismo para gerenciamento de vários processos. Entretanto, esta solução costuma ser cara e requer equipamentos

especiais, como *clusters* de CPU's, por exemplo, não sendo adequados para aplicações de usuários padrões, tais como jogos.

O *hyper-threading* é uma tecnologia que permite que um único processador possa ser visto por uma aplicação ou sistema operacional como dois processadores separados. Isto é possível, devido à arquitetura interna da CPU, que é composta por dois processadores lógicos. Do ponto de vista do *software* equivale a poder desenvolver uma aplicação para um sistema composto por mais de um processador e do ponto de vista de *hardware*, significa que instruções são alocadas para partes diferentes da CPU e são executadas individual e simultaneamente. Cada um destes pode usufruir das suas próprias interrupções, pausas, finalizações, etc. A nível de *hardware*, no entanto, a CPU consiste em dois processadores lógicos, mas que diferentemente do que processadores duais, compartilham uma série de componentes: os recursos do *core* do processador, *cache*, barramento de interface, *firmware*, etc. A arquitetura interna de *hyper-threading* permite que haja um ganho de até 30% em relação a sistemas de multi-processamento padrões, de acordo com INTEL (2001).

A INTEL implementou esta tecnologia de *hyper-threading* para servidores da série XEON e mais recentemente para a nova linha de processadores Pentium IV, sendo que já pode ser considerada uma tecnologia barata e acessível. Como este trabalho visa sobretudo aplicações de jogos 3D, que requer tipicamente recursos populares, escolheu-se esta arquitetura para realizar a implementação do paralelismo.

7.3 Paralelismo em pipelines de visualização tempo real

De acordo com Akenine-Möller (2002), um sistema paralelo de visualização em tempo real pode ser dividido em dois métodos: paralelismo temporal e paralelismo espacial.

O paralelismo temporal consiste em modelar o problema semelhantemente a uma linha de produção: cada processo possui a capacidade de resolver uma parte do problema. Uma vez terminada a sua parte, o processo envia os resultados para o processo seguinte e fica disponível para receber novos dados. Neste modelo, pode-se denominar cada processo como sendo um estágio do problema. A resolução completa do problema consiste em percorrer por todos os estágios. O

gargalo, neste caso, consiste no estágio que leva mais tempo para ser processado. A configuração ideal para este sistema é ter um processador para cada estágio.

Já o modelo de paralelismo espacial consiste em criar vários processos capazes de fazer a mesma coisa: ganha-se performance distribuindo partes menores do problema para cada processo. Deve-se observar que neste modelo apenas se podem implementar problemas que sejam de natureza paralelizável. Surge neste sistema um problema extra: o programa deve implementar uma função que se encarregua de juntar todos os dados, após terem sido devolvidos por cada processo. Em muitos casos, esta etapa pode inviabilizar a abordagem paralela, pois faz com que se gaste um tempo adicional que antes não era necessário.

7.4 Paralelismo e os Impostores com relevo

Neste trabalho, ressalta-se que a etapa de *pre-warping* envolvida num sistema de *ibr* é apropriada para um estágio que é processado pela CPU e que a etapa de texturização é um estágio para a GPU. Tendo em conta que os impostores com relevo constituem apenas parte da modelagem da cena e que a outra parte é descrita através de modelagem geométrica convencional, desenvolveu-se um *framework* que encara a CPU e a GPU como dois processadores separados e paralelos. Além disso, lançando-se mão de um processador com *hyper-threading*, vê-se a CPU como uma máquina paralela por si só.

Inicialmente, o sistema modelado é do tipo variáveis compartilhadas: a CPU e a GPU disputam os mesmos recursos. A CPU cuida do estágio de aplicação (*framework*) e a GPU utiliza estes resultados para realizar os estágios de geometria e rasterização. Há uma dependência temporal entre ambos, pois a GPU necessita dos resultados da CPU para seguir adiante, como se pode ver na figura 7.1. Desta maneira, diz-se que o sistema segue o modelo de paralelismo temporal.

Para realizar o *pre-warping* cria-se um *thread*. Este processo pode correr em paralelo ao estágio de aplicação da CPU, pois não há dependência de dados. Assim, o *thread* de *pre-warping* é visto como um sistema de paralelismo espacial em relação ao processo do *framework*.

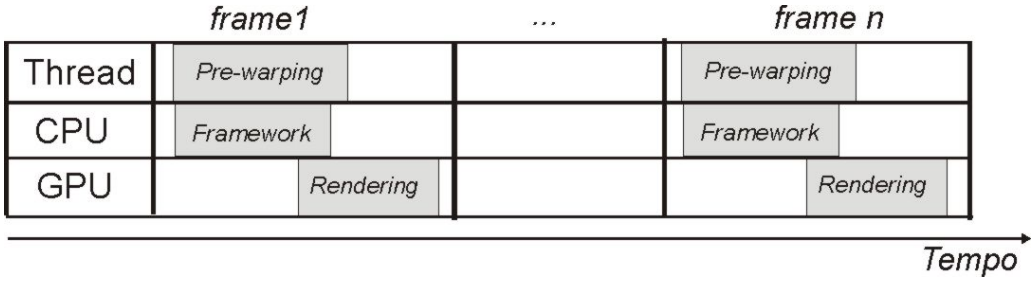


Figura 7.1 – Enquanto o processo de rendering da GPU é do tipo de paralelismo temporal em relação ao *framework*, o *thread* de *pre-warping* é do tipo de paralelismo espacial em relação ao mesmo.

Na modelagem de objetos por impostores com relevo torna-se necessário mais um processo, que é o responsável por gerar dinamicamente novas texturas com relevo. Como o processo de *pre-warping* precisa destas texturas para ser inicializado, o seu *thread* correspondente não pode ser disparado enquanto não receber o resultado da nova textura, já pronta. Surge assim um tempo de espera do sistema todo, uma vez que os estágios de geometria e de rasterização da GPU não podem dar inicio à visualização da textura enquanto este não lhe for enviado. A figura 7.2 ilustra esta situação. Vale ressaltar que este tempo de espera da GPU se dá apenas para a visualização do impostor com relevo, uma vez que os objetos geométricos da cena estão sendo tratados independentemente pelo *framework* e pela GPU.

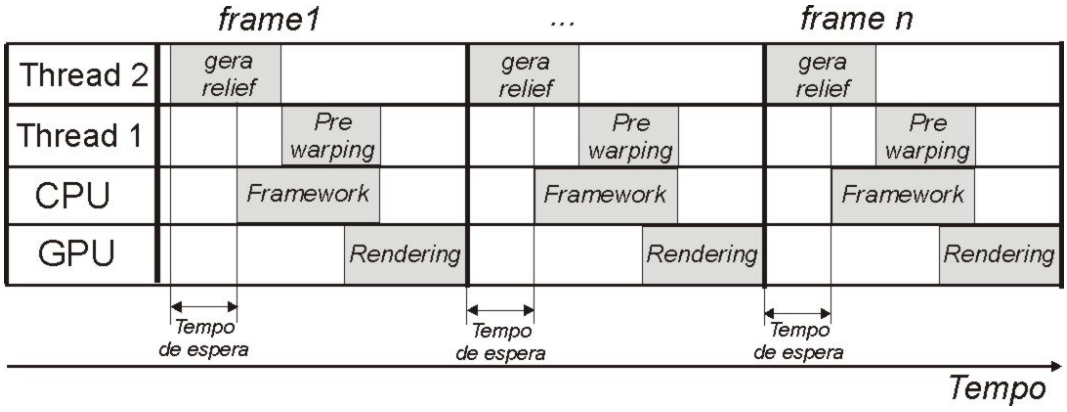


figura 7.2 – O processo responsável por gerar uma nova textura com relevo faz com que o tempo necessário para a visualização do impostor com relevo seja aumentado, devido à cadeia de dependências que se cria.

Para solucionar este problema, sugere-se fazer um sistema de previsão de texturas com relevo. A idéia é que, fazendo-se isto, quando a métrica de erro

indicar que uma textura com relevo caducou e uma nova textura se tornou necessária, há uma grande probabilidade desta já se encontrar pronta, evitando-se assim o tempo de espera que antes existia. Esta previsão pode ser feita através de uma inferência no vetor velocidade referente ao observador. Também se pode desenvolver uma estrutura de dados que armazena as texturas invalidadas num *cache*, para poderem ser utilizadas novamente, evitando-se assim um novo cálculo das mesmas. No capítulo 10, esta possibilidade é descrita com mais detalhes.

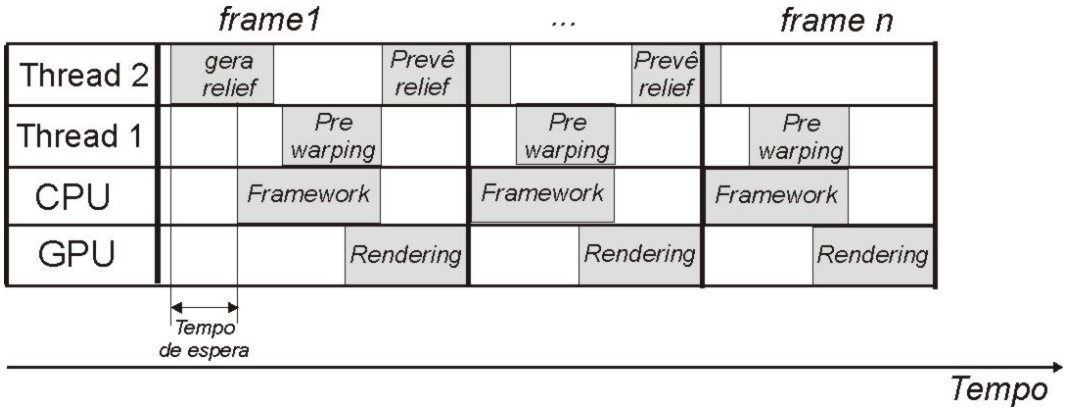


figura 7.3 - Um sistema de previsão é capaz de minimizar o tempo de espera de todo o *pipeline* para a visualização do impostor com relevo.

No capítulo seguinte discute-se com mais detalhes a arquitetura destes sistemas paralelos, bem como resultados obtidos por meio da sua implementação.

8 Implementação e Resultados Práticos

...there's a difference between knowing the path and walking the path. (The Matrix)

8.1 Framework utilizado

Para o sistema de visualização deste trabalho utilizou-se um *framework* capaz de visualizar objetos 3D de forma interativa. Os cenários podem ser modelados utilizando o software 3DSMAX e devem ser diferenciados os objetos que serão tratados por geometria com os que serão tratados como impostores como relevo. A estrutura básica do *framework* está ilustrada na figura 8.1:

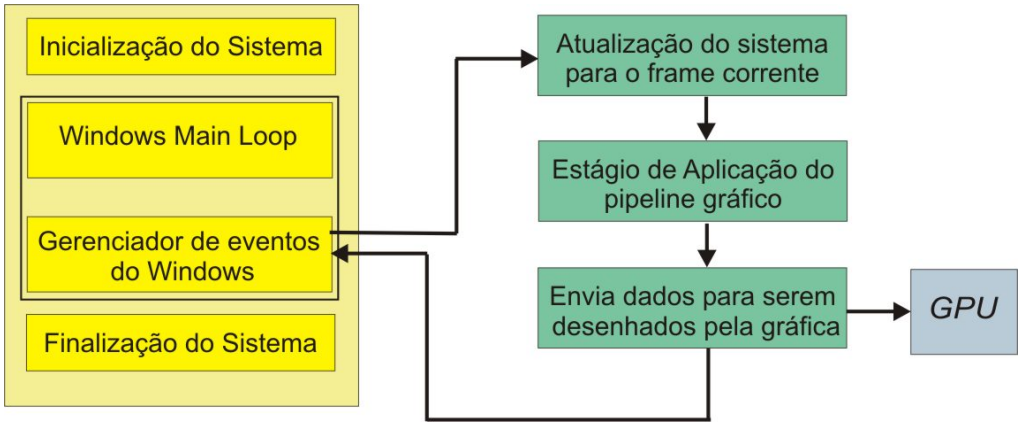


figura 8.1 – Diagrama da estrutura do *framework* utilizado para as implementações que se seguem

O sistema inicializa todas as variáveis globais antes de entrar no *loop* principal do *windows*. Uma vez dentro deste *loop*, para cada *frame*, executa-se a sequência da direita do diagrama, que a grosso modo pode ser resumido como sendo o estágio de aplicação do *pipeline* gráfico. No término deste estágio, antes de voltar ao início do *loop* principal, enviam-se os resultados para serem desenhados pela placa gráfica, correspondendo aos estágios de geometria e rasterização, que estão ambas a cargo da GPU. Uma vez feito este envio, a CPU já terminou sua parte do *pipeline* gráfico e apenas a placa gráfica possui tarefas pare

serem feitas. Neste âmbito, pode-se encarar o sistema como uma máquina paralela temporal, uma vez que o processador pode iniciar o processo de novas etapas da visualização enquanto a GPU está tratando a visualização dos dados anteriores.

8.2 Implementação básica do estágio de *pre-warping*

Foram implementadas quatro versões da técnica de mapeamento de texturas com relevo (Fonseca, 2004), variando o processo de amostragem e reconstrução da imagem. A estrutura do *framework* implementado é da seguinte forma:

Relief_Mapping (sprite, Relief, camera) // Chamado a cada frame

Reinicialização dos parâmetros para a câmera corrente

Configuração do polígono da textura com relevo para nova posição

Constrói_Look_Up_Table

Amostragem e Reconstrução da imagem

Mapeamento da textura com pre-warping sobre o polígono

Para todas as versões, procurando otimizar o cálculo das equações 3-1 e 3-2, que devem ser calculadas para cada *texel* do mapa de relevos, lançou-se mão de uma *lookup table*. Esta tabela deve ser reconstruída cada vez que qualquer parâmetro da câmera destino seja alterado e é feito pelo seguinte algoritmo (Oliveira, 2000):

Função Constrói_Look_Up_Table

$e_1 = -(\vec{c} \cdot \vec{a})/(\vec{a} \cdot \vec{a})$ *// e_1 e e_2 são as coordenadas do ponto epipolar*

$e_2 = -(\vec{c} \cdot \vec{b})/(\vec{b} \cdot \vec{b})$ $k_3 = 1/(\vec{c} \cdot \vec{f})$ *//k_1, k_2, k_3 são as constântes da eq. 3-1*

k_1 = e_1 . k_3 // referentes à posição corrente da câmera

k_2 = e_2 . k_3

Para i=0 até i >= 255 faça

Depth = Q(i) / 255 // Profundidade aproximada para um
// índice quantizado i

Coef_1[i] = k_1(Depth) // Coeficientes das Equações 3-1 e 3-2

Coef_2[i] = k_2(Depth)

Coef_3[i] = 1/(1+ k_3)(Depth)

Através desta *look up table*, os cálculos para cada *texel* que correspondiam a

$$u_i = \frac{u_f - k_1 \partial(u_f, v_f)}{1 + k_3 \partial(u_f, v_f)} \quad \text{e} \quad v_i = \frac{v_f - k_2 \partial(u_f, v_f)}{1 + k_3 \partial(u_f, v_f)}$$

são substituídos pelas seguintes operações:

u_{next} = *(u + Coef_1[Depth_Texel]) x Coef_3[Depth_Texel]*
v_{next} = *(v + Coef_2[Depth_Texel]) x Coef_3[Depth_Texel]*

o que se resume a 2 adições e 2 multiplicações por *texel*.

8.2.1 Amostragem Unidimensional Realizada em dois passos

O primeiro método consiste em criar dois passos totalmente separados e independentes para o processo de amostragem e reconstrução (Oliveira, 2000). Para esta versão cria-se uma função que realiza o cálculo de *warping* apenas para as linhas e outra que, terminada a primeira, realiza o cálculo de *warping* apenas para as colunas, conforme ilustra a figura 8.2. Como foi discutido na seção 4.2, ao realizar o *warping* de um *texel*, podem surgir buracos, ou seja, dois *texels* que eram vizinhos na textura original podem se afastar de tal forma que surgem *pixels* entre os dois que antes não existiam. (O tamanho destes buracos é delimitado pela medida de erro apresentada no capítulo 5, de forma que nunca são maiores do que um determinado valor estipulado).

Para solucionar este problema, o algoritmo implementado cria *texels* para esta região, fazendo uma interpolação linear entre os dois elementos pertencentes à imagem original. Esta interpolação deve ser efetuada não apenas para a cor dos *texels*, mas também para o valor das normais e da profundidade. Neste método, as interpolações horizontais e verticais são feitas separadamente. É importante notar que, ao atingir o passo vertical, faz-se uma interpolação sobre outra já feita no passo anterior.

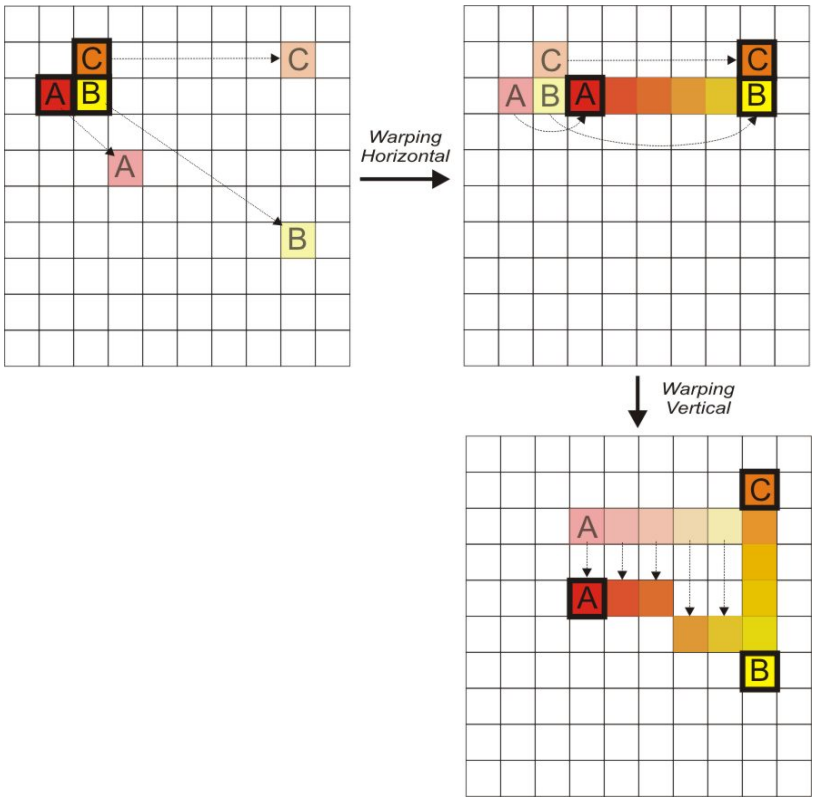


figura 8.2 – O método da amostragem unidimensional dividido em dois passos realiza primeiro o *warping* para as linhas, fazendo a interpolação necessária para preencher os buracos que surgem devido ao distanciamento de dois *texels* que eram vizinhos na imagem original. Depois de processar toda a imagem, realiza-se um segundo *warping*, restrito às colunas, também fazendo a interpolação de *texels* no caso de afastamentos dos mesmos.

Este método possui um inconveniente, especialmente mais notório, em locais de interpolação que existe maior descontinuidade na informação de profundidade. Este efeito pode-se tornar mais proeminente em linhas retas, que tenderão a parecer curvas. O erro se deve sobretudo ao fato de se realizar uma interpolação (no passo vertical) sobre um valor que já era interpolado (no passo horizontal)[*].

8.2.2 Amostragem Assimétrica Realizada em dois passos

Nesta versão da implementação das texturas com relevo se procura sanar o erro obtido na primeira abordagem. Neste caso, evita-se realizar uma interpolação

[*] Uma discussão mais detalhada das limitações desta abordagem pode ser obtida em (Oliveira, 99) e em (Fonseca, 04).

sobre outra, armazenando o valor final da coordenada vertical do *texel* já no primeiro passo do *pre-warping* (Oliveira, 2000), ou seja, no passo horizontal. Para isto, deve-se utilizar uma tabela auxiliar responsável por armazenar estes resultados. Denomina-se a esta versão de assimétrica porque o passo horizontal não é simétrico ao vertical.

8.2.3 Amostragem Realizada em Dois Passos com Compensação de Deslocamento

Neste método, sugerido e demonstrado em (Oliveira, 1999), utiliza-se um cálculo mais exato para a interpolação linear no cálculo dos valores de profundidade para os *texels* interpolados durante o passo horizontal. Esta interpolação, para os *texels* que estão entre A e B (figura 8.2) é dada pela equação 8-1:

$$disp(t) = \frac{\beta_1 - t\beta_2 - (1-t)\beta_3}{\gamma_1 + t\gamma_2 + (1-t)\gamma_3} \tag{8-1}$$

Onde *disp* (*t*) é o valor de profundidade interpolado, $\beta_1 = v_{fA}(1 + k_3 disp(B) - v_{pA}$, $\beta_2 = v_{fB} - v_{pA} + k_2 disp(B)$, $\beta_3 = v_{pA} k_3 disp(B)$, $\gamma_1 = v_{pA} k_3 - (1 + k_3 disp(B))k_2$, $\gamma_2 = (v_{fB} - v_{pA} + k_2 disp(B))k_3$ e $\gamma_3 = v_{pA}.k_3^2.disp(B)$, sendo que se denomina de v_{fN} ao *pixel N* da imagem fonte e v_{pN} ao *pixel N* da imagem com o *pre-warping*. Os demais coeficientes são os mesmos da equação 3-1.

8.2.4 Amostragem Intercalada Realizada em um Passo

Nas implementações descritas ainda é possível de ser observada a presença de ruídos durante o *pre-warping*. Tais ruídos surgem pelo fato de que múltiplas amostras são mapeadas sobre um mesmo *pixel* na imagem resultante, durante a primeira fase do *pre-warping*. Ao se realizar a segunda etapa, alguns *texels* que se tornam necessários para se fazer a interpolação já não existem, pois foram ocludidos. Neste método implementado, utilizou-se a solução indicada em (Oliveira, 1999) para este problema: intercalar os passos horizontal e vertical do *pre-warping*. Para cada *texel* intermediário produzido no passo horizontal realiza-

se imediatamente sua interpolação dentro da coluna apropriada. Como cada iteração vertical recebe e processa os *texels* numa ordem compatível de oclusão, a visibilidade correta é preservada na imagem resultante.

Este método, diante dos três apresentados anteriormente, é o que menos operações realiza na implementação do *pre-warping*, devido às intercalações. Por esta razão é o que possui maior velocidade, como se pode verificar na tabela 8.1, e é o método utilizado para o restante do desenvolvimento que se apresenta nesta dissertação.

Algoritmo	Tempo médio de processamento (ms)	Desvio Padrão (ms)
Amostragem unidimensional	47.406	3.001
Amostragem Assimétrica	47.859	4.612
Amostragem com compensação	49.625	6.075
Amostragem intercalada	40.859	7.734

Tabela 8.1 – Tempo obtido para realizar o *pre-warping* utilizando cada um dos 4 algoritmos de amostragem discutidos previamente. O modelo usado para esta medida é o da figura 5.4, numa resolução de 256 x 256 e utilizando uma máquina com processador Pentium IV 2.6 GHz.

8.3 *Pre-warping* serial no *pipeline* gráfico

Numa primeira implementação, inseriu-se toda a etapa de *pre-warping* das texturas com relevo no estágio de aplicação, conforme ilustra a figura 8.3:

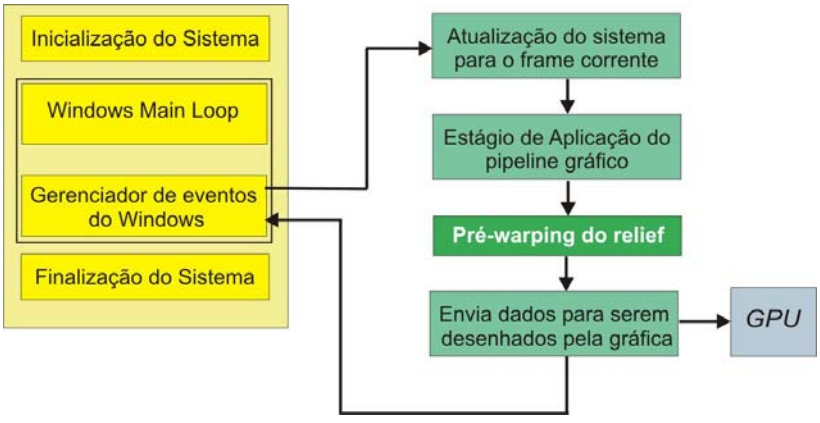


figura 8.3 – Primeira implementação do mapeamento de texturas com relevo utilizando o *framework* apresentado. Apenas se mostra no diagrama a etapa de *pre-warping*, já que a etapa de texturização está a cargo da GPU.

Nesta implementação não houve nenhuma preocupação com paralelismo. Assim sendo, enquanto se está realizando a etapa de *pre-warping*, todo o sistema está parado, à espera do resultado a ser gerado. Ao se fazer isto, pôde-se observar que a etapa de *pre-warping* tornou-se o gargalo de toda a aplicação, ou seja, ela tornou-se a responsável por restringir a performance do sistema. Num Pentium IV, 3GHz com uma RADEON 9700, enquanto numa cena sem objetos com texturas com relevo a performance era de 95 FPS (*frames* por segundo), ao se inserir um objeto deste tipo a performance caiu para 11 FPS.

Além disso, nesta primeira implementação utiliza-se uma textura com relevo já pronta e armazenada na memória. Como existe apenas uma textura para o objeto, torna-se impossível vê-lo de qualquer ângulo. Desta maneira, o *framework* restringe o movimento da câmera. A figura 8.4 mostra um diagrama das etapas envolvidas no processo, bem como uma identificação do local onde esta etapa é executada: CPU ou GPU.

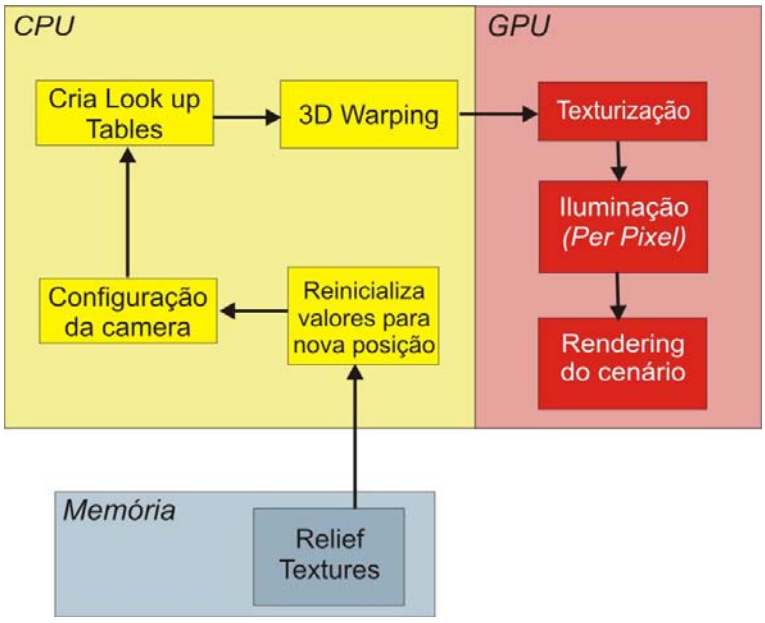


figura 8.4 – Distribuição entre CPU e GPU na implementação do sistema.

8.4 *Pre-warping* com *Time-Slice* fixo

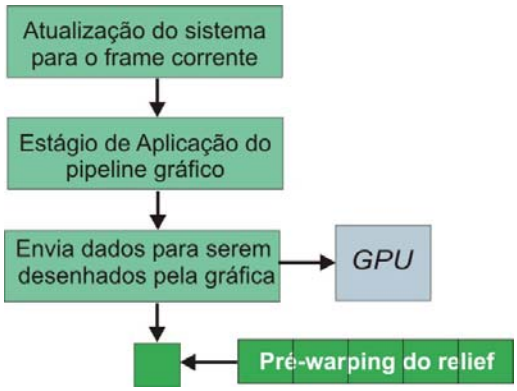
Numa segunda implementação, procurando encarar um sistema com CPU e GPU como uma máquina paralela, criou-se uma distribuição baseada em *time-slice*: dividiu-se a etapa de *pre-warping* em vários pedaços. Para cada *frame*, faz-se com que a CPU realize o *pre-warping* de uma parte da textura. Apenas ao

terminar o *pre-warping* por completo se envia esta textura para a placa gráfica, de modo a tratar da etapa de texturização. Perceba-se no diagrama da figura 8.5 (a) que o *pre-warping* foi inserido após enviar os dados para a GPU. Fez-se isto porque se observou que neste momento a CPU estava com maior tempo ocioso. Enquanto a GPU realiza todo o estágio de geometria e de rasterização, a CPU está tratando de uma parte do *warping*. Ao utilizar esta abordagem, a taxa de visualização da cena subiu para 75 FPS, mas a taxa de atualização das texturas com relevo caiu para 6 por segundo. Embora a atualização destas texturas tenha sido menor, esta abordagem foi melhor porque comprometeu menos a interatividade da aplicação e a visualização dos objetos geométricos.

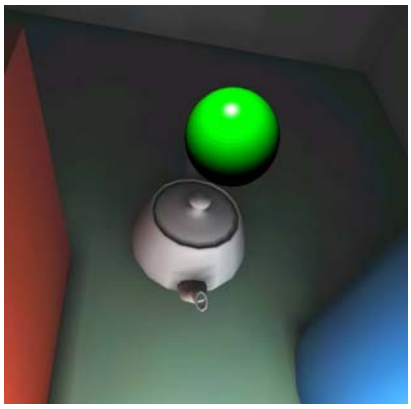
8.5 *Pre-warping* com *time-slice* variável

Uma variação feita na implementação anterior foi a de não fixar o tempo concedido ao *warping*: permite-se que a CPU faça o cálculo de *warping* apenas enquanto dura o processo de visualização da placa gráfica. Desta maneira, garante-se que a CPU apenas está processando a textura com relevo enquanto a GPU está de fato ocupada. Quando o *framework* volta a precisar da CPU, esta interrompe o *warping* que estava realizando. Sempre que se termina uma varredura completa da textura com relevo, a CPU avisa que no próximo frame a GPU pode usar uma textura nova para o objeto com relevo, que está disponível na memória RAM. (constatou-se que não é conveniente realizar o *warping* diretamente na memória de vídeo, pois isto interrompe o processamento que está ocorrendo na GPU).

Esta implementação deixou a taxa de FPS praticamente igual à que existia antes de se inserir o objeto de textura com relevo, mas por outro lado, a taxa de atualização da textura com relevo ficou mais lenta (em torno de 3 a 4 atualizações por segundo, para a cena da figura 8.5). Vale ressaltar também que quanto mais complexa for a cena para a GPU, mais tempo a CPU tem para realizar o *pre-warping* e, portanto, maior será a taxa de atualizações da textura com relevo.



(a)



(b)

figura 8.5 – (a) Estrutura do *framework* para *time slice*. (b) enquanto a cena é desenhada a uma taxa de 75 FPS, a esfera, que é representada por textura com relevo, é atualizada a 6 vezes por segundo, na primeira versão desta implementação. Já na segunda versão, a taxa ficou em torno de 95 FPS, mas a taxa de *warpings* por segundo caiu para 3.

8.6 *Pre-warping* com *multi-threading*

Numa terceira abordagem implementou-se o mapeamento de texturas com relevo utilizando *multi-threading*. Neste caso, o *pre-warping* ficou a cargo de um *thread* exclusivo para isto (para tanto é necessário utilizar uma máquina com mais de um processador; no caso deste trabalho utilizou-se um processador com *hyper-threading*). Com isto um processador fica dedicado ao *pipeline* de visualização padrão, sem ser interrompido nenhuma vez para fazer as tarefas relacionadas ao mapeamento de texturas com relevo.

Através de mensagens, este processador avisa ao *thread* se houve mudanças de posição do observador, o que requer que um novo *warping* seja criado para a textura. O *thread*, ao terminar de fazer o *pre-warping*, avisa ao processador do *framework*, que por sua vez, permite que o *thread* envie o resultado para a GPU. Este processo de sincronização é implementado através de uma máquina de estados, ilustrado na figura 8.6. Novamente vale ressaltar que o *warping* foi feito inteiramente na memória RAM, para que o processamento da placa gráfica não seja interrompido a todo instante. Apenas quando se termina uma sequência de *warping*, realiza-se a transferência para a memória de textura.

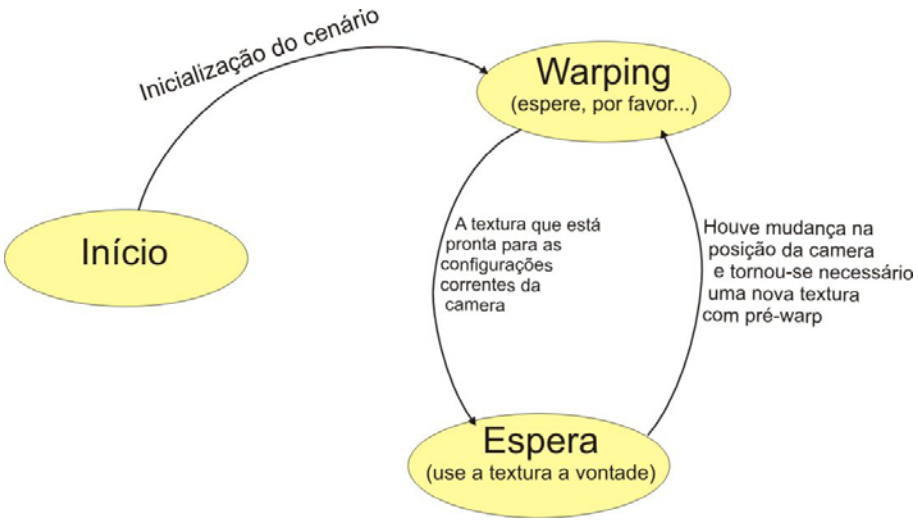


figura 8.6 – Máquina de estados responsável por sincronizar o processo de *pre-warping*.

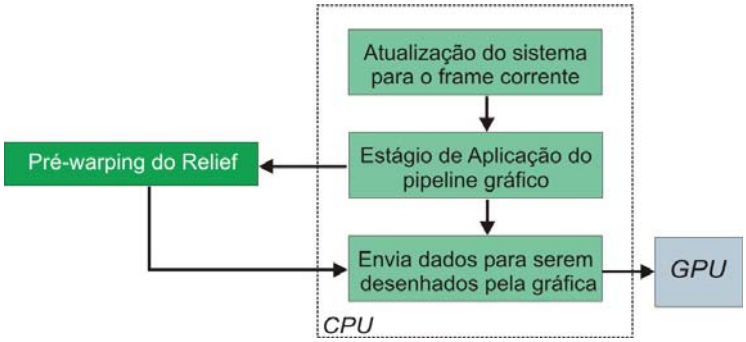


figura 8.7 – Estrutura do *framework* para texturas com relevo utilizando *multi-thread*.

A máquina de estados é controlada pelo *framework* (figura 8.8). Perceba-se que não há mais nenhuma conexão entre o laço de visualização do cenário normal e o laço do *pre-warping*. A máquina de estados garante que não se passe para a placa gráfica uma textura com relevo que está no meio do processo de *pre-warping*.

Testou-se este sistema também numa máquina sem possuir *hyper-threading*. Neste caso, o resultado foi superior ao apresentado no *pre-warping* com *time-slice* variável. Nesta versão deu-se ao processo de *pre-warping* uma prioridade pequena, de forma que o sistema operacional gerenciasse os momentos em que a CPU tem tempo disponível para este processo. Assim sendo, sempre que a CPU encontra tempo ocioso passa a dedicar-se ao *pre-warping*. Observou-se que numa aplicação de visualização com o *framework* desenvolvido, sem *culling* por software, o tempo ocioso da CPU chega a 90% para uma cena de 1000 polígonos.

Desta forma, o tempo que existe à disposição para o passo de amostragem e reconstrução da textura com relevo é grande.

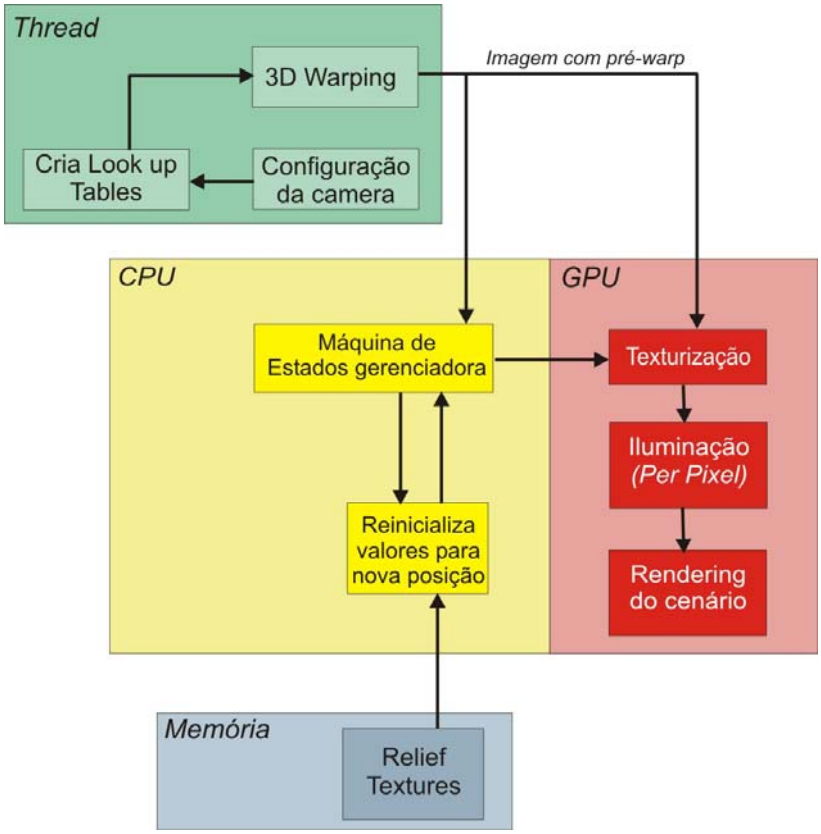


figura 8.8 – Arquitetura mais detalhada do *framework* com implementação de processo distribuído para o *pre-warping*.

8.7 *Pre-warping* multi-processado

Na implementação anterior é apresentada uma arquitetura que cria um processo dedicado à etapa de *pre-warping*. Esse método é especialmente interessante quando se dispõe de uma máquina com dois processadores ou com tecnologia de *hyper-threading*, mas não é capaz de explorar sistemas que possuem mais do que dois processadores. A abordagem presente é a que pode ser considerada com mais propriedade de paralela, pois divide a textura com relevo em *n* pedaços, onde *n* é o número de processadores disponíveis. Neste caso, cada um se encarrega do *pre-warping* de um pedaço. Quando todos terminam, um processo deve se responsabilizar por unir os diversos resultados numa só textura. Este método é classificado como paralelismo espacial, de acordo com as categorias apresentadas na seção 7.3.

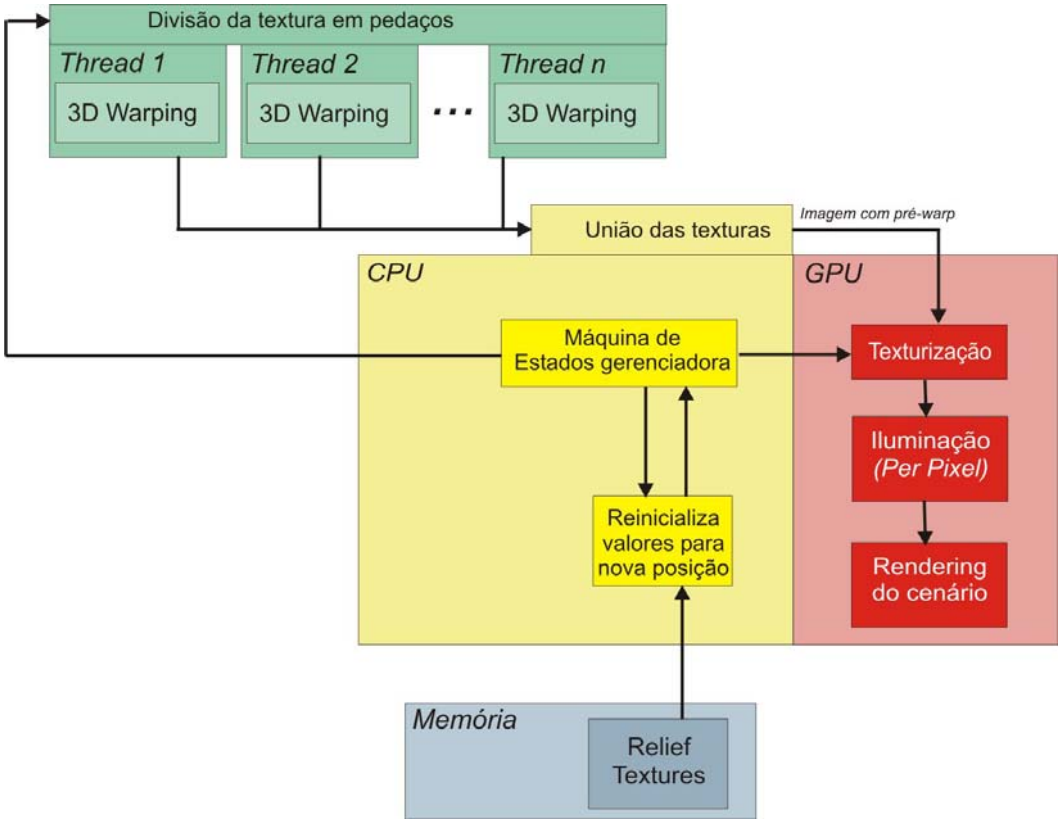
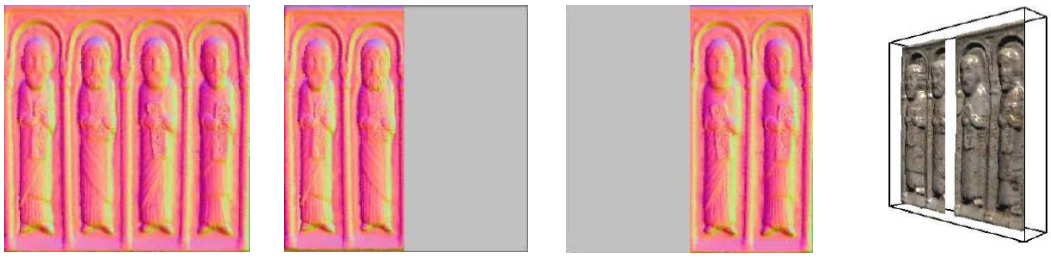


figura 8.9 – Arquitetura do sistema para multi-processamento. Cada *thread* recebe um pedaço da textura original e se encarrega de fazer o *pre-warping* apenas neste pedaço.

Deve-se ressaltar que o processo de divisão da textura com relevo não consiste simplesmente em quebrar a imagem original em *n* pedaços menores. Ao se fazer o *pre-warping* de um pedaço, pode ocorrer de que um *texel* seja deslocado para uma região que pertence a outro *thread*. Se o processo não possuir acesso a esta parte, o *texel* que se está movendo é simplesmente perdido. O efeito disto consiste em "buracos" nas emendas dos pedaços, conforme se pode ver na figura 8.10 (d).



(a) (b) (c) (d)

figura 8.10 – Ao se dividir a imagem em *n* partes, o *warping* de uma pode "invadir" o espaço que era de outra, ocorrendo a perda destes *texels* e formando um "buraco" na imagem com relevo.

Para evitar este problema, a divisão da textura com relevo em pedaços deve ser feita da seguinte forma:

- Alocar um espaço de memória para cada processo com a mesma resolução da textura com relevo original;
- Inicializar os valores de profundidade para todos os *texels* como sendo um valor inválido;
- Copiar as informações da textura original apenas para a região que é usada para este pedaço.

Estes passos podem ser observados na figura 8.10. Em (a) está representada a textura original. (b) e (c) representam dois pedaços diferentes que serão enviados a processadores distintos para sofrerem o *pre-warping*. A área cinza de cada uma destas imagens indica que há valores de profundidade equivalentes a *texels* inválidos. Fazendo-se isto, permite-se que cada processo escreva valores nos *texels* da região de outra imagem, caso seja necessário.

O processo necessário para unir as texturas com o *pre-warping* terminado também deve levar em conta esta possível invasão de áreas. O algoritmo abaixo descreve esta união:

Para i de 1 a n faça
Para cada texel pertencente ao pedaço i faça
Se texel é válido então copie este texel para textura final
Senão verificar se pedaço vizinho apresenta este mesmo texel como válido
Caso seja válido, copie-o para a textura final

A tabela 8.2 mostra resultados obtidos para 3 implementações: *pre-warping* serial, *pre-warping* com *multi-threading* e *pre-warping* paralelo. Neste caso, a medida do tempo é feita pela taxa de FPS do processo de visualização completa e não do número de *warpings* por segundo que estão sendo feitos.

Pode-se perceber que o desempenho obtido no *pre-warping* paralelo é muito inferior ao obtido com a abordagem *multi-threading*. Isto ocorre porque no primeiro existem 3 processos para 2 processadores (um processo do *framework* e dois processos do *pre-warping*) e no segundo 2 processos para 2 processadores. Deve-se ter em conta que na abordagem paralela existe ainda o processo de separação e junção das texturas, que não é necessário para o *multi-threading*.

Outra observação que se faz é de que o valor de FPS corresponde ao do *framework* como um todo, e não à taxa de *warpings* por segundo que se está realizando. Esta tabela mostra que, para a arquitetura de *hardware* escolhida, o segundo método deixa o *framework* mais livre para o processamento do restante da cena.

Abordagem	Taxa de FPS	Desvio padrão
pre-warping Serial	24.93	1.54
pre-warping Multi-threading	96.45	5.01
pre-warping multiprocessado	31.09	7.67

tabela 8.2 – Comparação da performance obtida para as diversas abordagens do *pre-warping*. A textura com relevo utilizada é a mesma da tabela 8.1, com um Pentium IV 2.6 GHz com *hyper-threading*.

Outra abordagem, que pode ser mais eficiente, porém não implementada, consiste em criar *n threads*, deixando cada um responsável pela linha *i mod Altura_Imagem*, sendo *i* o número do processo e *Altura_Imagem* a resolução vertical da textura com relevo. Utilizando o método de amostragem intercalada (seção 8.2.4), poderia-se escrever o resultado diretamente na textura final, não sendo necessário realizar a etapa de junção.

8.8 *Pre-warping* com atualização dinâmica dos Impostores com relevo

Nesta versão, implementou-se primeiramente a estimativa de Schaufler, de forma a minimizar o número de *warpings* realizados sobre o impostor. Quando o *warping* não for necessário, o impostor é tratado como um simples *sprite*. Isto é especialmente conveniente para o caso de se implementar um sistema de previsão para gerar texturas com relevo, em máquinas que apenas possuem 2 processadores (e.g., processadores com *hyper-threading*), pois neste caso pode-se liberar o *thread* de *warping* para esta função. Apenas com esta otimização, observou-se a seguinte melhoria, para a câmera parada:

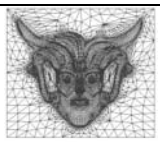
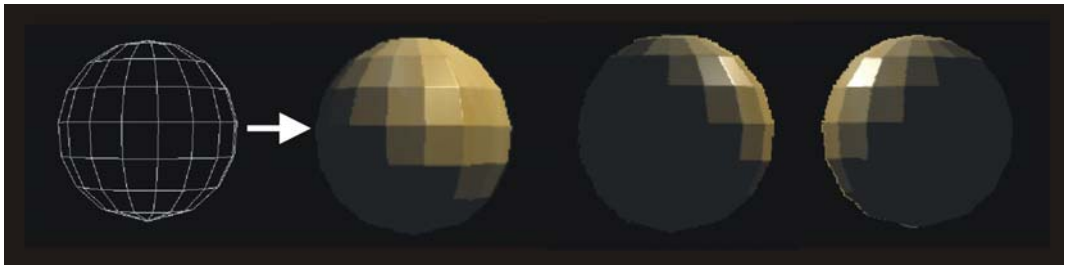
Objeto	FPS (sem Schaufler)	FPS (com Schaufler)
	35	120

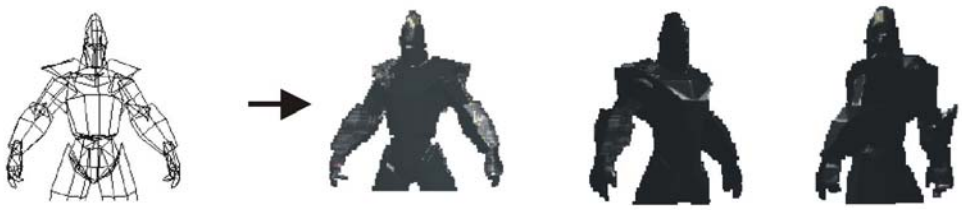
tabela 8.3 – Comparação de performance com implementação do teste de Schaufler para otimizar número de *warpings*. Os testes foram feitos sem ativar o *hyper-threading*. Esta otimização está apresentada com detalhes na seção 5.3.

Sempre que ocorrer um movimento do observador e o teste de Schaufler falhar, então o sistema permite que se inicie um *thread* para realizar um *warping* sobre a textura com relevo. Quando este termina, aplica-se o teste da métrica apresentado na seção 5.4. Caso se comprove que o erro seja maior do que o valor tolerado, inicia-se o cálculo de uma nova textura com relevo, que é usada tão logo fique pronta.

A figura 8.11 mostra dois exemplos de impostores com relevo: Em (a) foram necessários 14 atualizações para se dar uma volta completa sobre a esfera, que é um objeto com topologia simples, já em (b) foram necessárias 26 atualizações para o mesmo movimento sobre um objeto com geometria complexa.



(a)



(b)

figura 8.11 – Dois exemplos de Impostores com relevo: (a) representa uma geometria simples, onde são necessárias 14 atualizações para se dar uma volta por completo, já no caso de (b) se representa uma geometria complexa e são necessárias 26 atualizações para que o erro do *warping* passe desapercebido ao longo de uma volta completa.

No capítulo 5.1 discutiu-se a necessidade de gerar uma nova textura com relevo para o objeto, quando estes critérios de erro forem atingidos. Nesta implementação criou-se um sistema paralelo que gera uma nova textura com relevo quando o observador chega a uma região crítica. Esta região corresponde a um local próximo ao de um ponto em que o erro superará o valor tolerado pela equação (5-7). Caso existam apenas dois processadores, o *thread* para a geração da nova textura com relevo deve ter prioridade menor do que o processo de *pre-warping*.

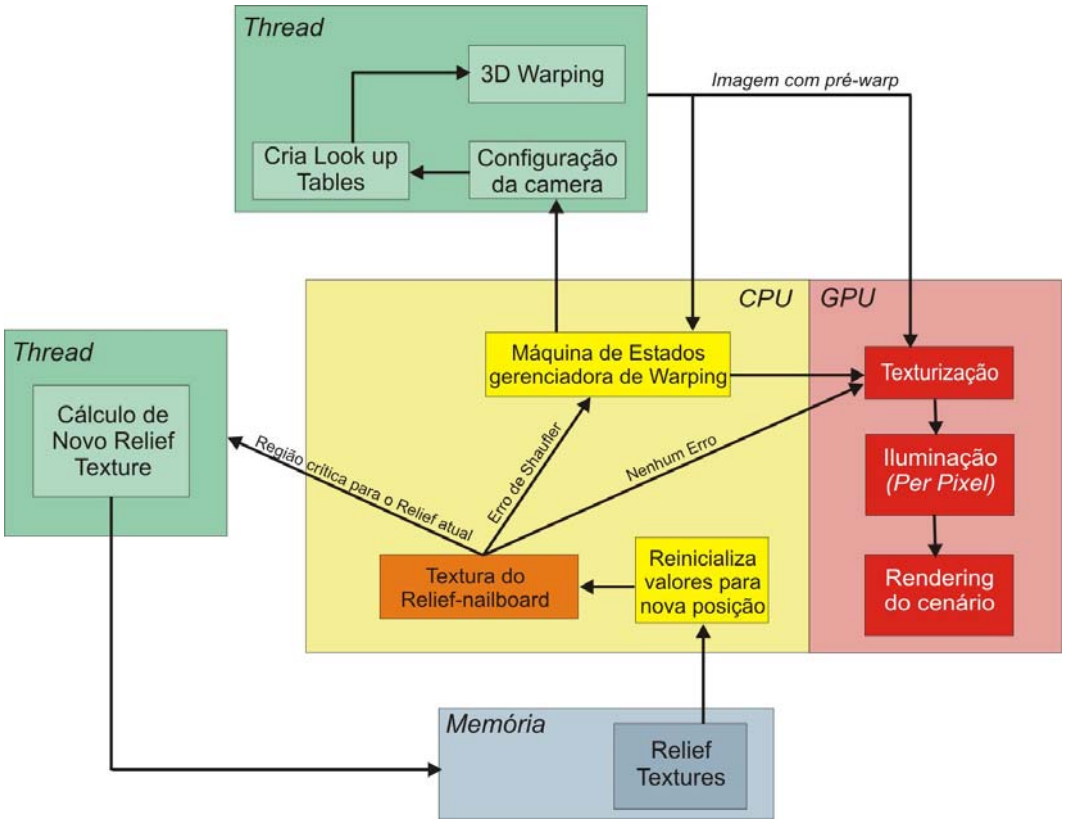


figura 8.12 – Arquitetura do sistema com otimização e métricas de erro.

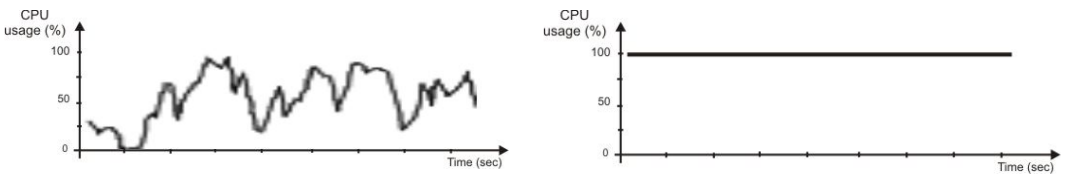
A geração de uma nova textura corresponde a uma visualização do objeto pela CPU, podendo-se utilizar qualquer algoritmo que forneça a profundidade de um ponto. Denominam-se os programas que realizam este cálculo de *software shaders*. Como foi discutido, estes possuem a capacidade de calcular efeitos que podem ser impossíveis de serem obtidos pela GPU. Neste trabalho implementou-se um *software shader* para *ray-tracing*, omitindo a computação da iluminação, já que este fica a cargo do *pixel shader*, utilizando o mapa de normais do objeto em questão. Este é um algoritmo que não é conveniente de ser implementado na placa

gráfica, por envolver recursão (até a data do presente trabalho as linguagens de programação para placa gráfica impossibilitam que haja recursão). A figura 8.13 mostra um mesmo objeto sendo renderizado pelo *software shader* e pela GPU.

figura 8.13 – O personagem da esquerda foi renderizado pelo *software shader* de *Ray-tracing* e o personagem da direita foi renderizado diretamente pela GPU.

O tempo consumido para se realizar o warping da uma imagem, que é de 256 x 256 pixels, num Pentium IV 2.6GHz foi de 11 ms, resultando numa taxa de 91 warpings por segundo. Já o tempo para gerar um novo impostor com relevo pelo *software shader* foi de 228 ms, para um nível de Ray-tracing, num objeto composto por 1500 polígonos. Para manter um erro de warping acumulado quase imperceptível, o valor colocado para a heurística gerou 17 imagens ao dar uma volta completa sobre o objeto da figura 8.13. Para esta cena, a taxa de frames por segundo foi de 131 no caso de ser totalmente renderizada em GPU. Já para o caso de se usar software shader, a taxa foi de 124. Apesar de serem valores muito próximos, esta pequena perda se deve especialmente ao tempo de transferência que ocorre da imagem na memória do sistema para a memória de vídeo.

A figura 8.14 mostra o tempo de consumo de CPU para ambos os casos.



(a) (b)

figura 8.14 – Tempo de consumo de CPU sem o *software shader* (a) e com o *software shader* (b)

Pelo exemplo apresentado, pode-se comprovar que houve um pequeno comprometimento da taxa de frames por segundos ao inserir um *software shader* responsável por gerar impostores com o algoritmo de ray-tracing. Entretanto o acréscimo de precisão e realismo na imagem gerada é consideravelmente maior. Manter a taxa de frames por segundos praticamente inalterada, porém acrescentando recursos de visualização mais realistas é justamente uma das principais propostas dos impostores com relevo. A figura 8.14 mostra que o processamento extra para obter tal resultado foi retirado do tempo que antes estava sendo desperdiçado pela CPU.

As texturas com relevo que foram usadas e deixaram de valer, podem ser armazenadas num *cache*, juntamente com as posições no espaço onde são válidas. Estas posições podem ser geradas através de volumes, estimados a partir da métrica de erro apresentada. Para objetos com superfícies contínuas, e portanto com valores pequenos no gradiente da profundidade, estes volumes tendem a ser maiores. Objetos com muitas descontinuidades possuem um mosaico maior de volumes ao seu redor (figura 4.3 e 8.11). Desta forma, antes de efetivamente ser gerado uma nova textura com relevo, verifica-se se há um volume na nova posição do observador e, neste caso, utiliza-se uma textura previamente calculada ao invés de gerar uma nova.

Quando o espaço de memória disponível para as texturas com relevo estiver saturado e for necessário descartar alguma textura para dar espaço a uma nova, é conveniente analisar o volume que se encontra mais distante da posição corrente do observador ou analisar a direção do movimento e estimar o local menos provável de se estar nos próximos *frames*. Esta idéia, bem como outras possíveis melhorias, estão discutidas no próximo capítulo: Conclusão.

9 Conclusão

I don't know the future. I didn't come here to tell you how this is going to end. I came here to tell you how it's going to begin. (The Matrix)

9.1 Contribuições

Foi apresentado que através da técnica de impostores com relevo é possível aumentar consideravelmente o tempo de vida de um objeto sendo representado por uma imagem. Diferentemente do método tradicional dos *sprites,* entretanto, faz-se necessário fornecer ao sistema de visualização a malha do modelo a ser utilizado, bem como o algoritmo de renderização. Outra alternativa que se pode seguir para atingir tal fim consiste nos *Layered Depth Images*, mas destaca-se que os impostores com relevo são mais simples de serem gerados dinamicamente, por se adaptarem facilmente aos algoritmos de visualização padrão.

Mostrou-se que os impostores com relevo podem representar qualquer geometria, estendendo a capacidade de modelagem por texturas com relevo, apresentado inicialmente por Oliveira (1999).

No trabalho, pode-se comprovar também que em alguns casos é mais conveniente dividir o processamento de um sistema de *ibr* em duas etapas (uma para a CPU e outra para a GPU) do que implementá-la por completo em *hardware.* Isto porque o processo de *warping* inverso possui um inconveniente que é a falta de conhecimento da profundidade do *pixel*.

A complexidade inerente à visualização de texturas com relevo é independente da complexidade da geometria do objeto representado. No caso da implementação de (Policarpo, 2003), para um grau de realismo razoável este processamento é demorado, especialmente devido as limitações impostas pelos recursos de *hardware*. Já no caso dos impostores com relevo, apesar da velocidade para atualizar as texturas ser dependente da geometria e do algoritmo de visualização sendo utilizado, comprovou-se que há uma grande eficiência na

atualização. Mais ainda: para o *hardware* disponível na época da elaboração deste trabalho, o desempenho foi muito superior para esta segunda abordagem. Isto porque o *hardware* fica dedicado a tarefas mais simples, adequadas à sua capacidade atual.

Comprovou-se também nesta tese, em colaboração com (Fonseca, 2004) que é possível paralelizar o processo de mapeamento de texturas com relevo a diversos níveis: CPU/GPU, *multi-threading* e multi-processamento. Os resultados de algumas abordagens são apresentados nos capítulos 6 e 7.

Finalmente, uma importante contribuição da tese consiste na construção de um *framework* que permite inserir elementos geométricos e impostores com relevo com bastante facilidade. Este sistema simplifica consideravelmente a continuação da pesquisa aqui iniciada.

9.2 Trabalhos Futuros

Uma possível extensão ao que foi discutido no capítulo 7.8 consiste em criar um sistema de previsão para gerar impostores com relevo: quando o *thread* dedicado a gerar novas texturas com relevo estiver ocioso, este poderia ser aproveitado para gerar texturas para futuras posições do observador, armazenando estes resultados no *cache*, enquanto ainda não sejam requisitados. Este sistema de previsão pode ser baseado numa estimativa sobre futuras posições do observador, tendo em vista a análise do vetor velocidade. O gerenciamento do *cache* pode possuir certo grau de "inteligência", procurando descartar texturas para posições menos prováveis em que o observador pode vir a se encontrar, quando se torna necessário liberar espaço de memória.

Para a tarefa de gerar dinamicamente novas texturas com relevo, pode-se desenvolver um sistema de cálculo progressivo. Desta forma, no caso de uma textura ser solicitada mas esta ainda não ter sido gerada por completo, envia-se ao *pipeline* a textura com uma resolução menor porém completa.

Neste trabalho apenas se discutiu sobre a utilização de impostores com relevo estáticos. Uma extensão da pesquisa deve considerar objetos animados. Como sugerido em (Oliveira, 2000), a translação do impostor com relevo pode ser feita com uma simples translação do polígono de suporte, com possíveis atualizações da textura, de acordo com a métrica apresentada; a rotação também

consiste num movimento deste tipo no polígono de suporte e a escala também pode ser feita com uma simples transformação do polígono, levando em conta a utilização de texturas com relevo com multiresolução. Neste caso, um sistema de previsão pode ser muito importante, pois ao contrário de que uma previsão de posições do observador, onde não há certeza, a previsão de transformações provindas de uma animação já existente possui 100% de acerto. Esta abordagem corresponderia a inserir o parâmetro tempo na função plenóptica, com a limitação de animações poderem ser apenas de translação, rotação e escala.

Para o caso de não existir o mapa de normais para a textura com relevo em questão, a seção 5.6 apresenta uma forma de criar valores estimados. Para esta finalidade, pode-se criar uma modelagem mais complexa, no entanto mais realista, realizando uma pré-avaliação e um reconhecimento de padrões da imagem.

Ao gerar dinamicamente as texturas com relevo, é possível fazer uma abordagem onde se acrescenta para cada *texel* o cálculo de sombreamento. Isto pode ser particularmente útil quando as luzes do cenário e os objetos forem sempre estáticos, ou para sistemas que não dispõem de *fragment processors* na GPU.

Dentro de um ambiente de visualização 3D, como num *engine* gráfico para jogos 3D, por exemplo, poderia-se desenvolver critérios e métodos inteligentes de identificar quais objetos devem ser tratados através de geometria e quais através de imagem (no caso, impostores com relevo). Pode ser que um mesmo objeto, num espaço de tempo deva ser visto como geometria e em outro momento deva ser tratado como imagem. Para isto, deveria-se analisar como realizar um processo de transição suave.

Uma possível aplicação dos impostores com relevo poderia dar-se em sistemas de visualização distribuídos por *hardware*: dentro de uma rede constata-se que se uma CPU distribui *frames* para serem calculados por várias GPU's, em máquinas separadas, devido à velocidade de transmissão de dados da rede, há um tempo de atraso grande na devolução das imagens para o servidor. Este tempo impede que se possa manter uma taxa de *frames* por segundo adequada para um sistema de visualização em tempo real. Uma possível paralelização seria enviar às demais GPU's requisições de possíveis impostores com relevo que serão necessários para os objetos do cenário. Isto atenuaria o atraso, uma vez que o impostor com relevo tem um tempo de vida maior que um impostor simples.

Utilizando-se o sistema de previsão discutido, poderia-se aliviar grandemente a CPU/GPU principal.

Por fim, uma interessante pesquisa consistiria na implementação de impostores com relevo através do algoritmo de *view-morphing*, ao invés do 3D *image warping*. Neste caso os objetos deveriam ser chamados de impostores com *morphing*...

Referências Bibliográficas

ADELSON, E. and BERGSEN, J. 1991. **The Plenoptic Function and the Elements of Early Vision.** In: Computacional Models of Visual Processing, MIT Press, Cambridge, MA, p. 3-29.

AIREY, J., ROHLF, J. and BROOKS, F. 1990. **Toward Image Realism with Interactive Update Rates in Complex Virtual Building Environments.** In: Proceedings of ACM Symposium on Interactive 3D Graphics, vol. 24, no. 2, p. 41-50, March.

AKENINE-MÖLLER, T. and HAINES, E. 2002. **Real-Time Rendering**, A K Peters, Natick, Massachusetts, second edition edition.

ALIAGA, D. and LASTRA, A. 1997. **Architectural Walkthroughs Using Portal Textures.** In: Proceedings of IEEE Visualization'97, p. 355-362, October.

ALIAGA, D. 1999. **MMR: An Integrated Massive Model Rendering System Using Geometric and Image-Based Acceleration.** In: Proceedings of ACM Symposium on Interactive 3D Graphics. Atlanta, GA, p. 199-206, April.

ALIAGA, D. and CARLBOM, I. 2001. **Plenoptic Stitching: A Scalable Method for Reconstructing 3D Interactive Walkthroughs**. In: ACM SIGGRAPH Computer Graphics Proceedings, p. 443-450.

ANDREWS, G. R. 2000. **Foundations of Multithreaded, Parallel, and Distributed Programming.** Addison Wesley, USA.

BAYAKOVSKI, Y., LEVKOVICH, M., et al. 2002. **Depth Image-based Representation for Static and Animated 3D Objects.** Moscow State University Techical Report.

BEIER, T. and NEELY, S. 1992. **Feature-based image metamorphosis**. In: ACM SIGGRAPH Computer Graphics Proceedings, p. 35-42.

BLINN, J. and NEWELL, M. 1976. **Texture and Reflection in Computer Generated Images.** Communications of the ACM, vol. 19, no. 10, p. 542-547, October.

BORSHUKOV, Georgi D. 1997. **New Algorithms for Modeling and Rendering Architecture from Photographs.** Master Thesis, Department of Computer Science, Berkeley.

BUEHLERE, C., BOSSE, M., MCMILLAN, L., GORTLER, S., COHEN, M. 2001. **Unstrucutural Lumigraph Rendering**. In: ACM SIGGRAPH Computer Graphics Proceedings.

CARISSIMI, A., OLIVEIRA, R., TOSCANI, S. 2001. **Sistemas Operacionais. Série Livros Didáticos**. Instituto de Informática da UFRGS. Sagra Luzzatto, Porto Alegre.

CATMULL, E. 1974. **A Subdivision Algorithm for Computer Display of Curved Surfaces**. PhD thesis, Department of Computer Science, University of Utha, December.

CHAI, J., TONG, X., et al. 2000. **Plenoptic Sampling**. In: ACM SIGGRAPH Computer Graphics Proceedings.

CHEN, S., WILLIAMS, L. 1993. **View Interpolation for Image Synthesis**. In: ACM SIGGRAPH Computer Graphics Proceedings.

CHEN, S. 1995. **Quicktime VR - An Image-based Approach to Virtual**

Enviroment Navegation. In: ACM SIGGRAPH Computer Graphics Proceedings, p. 29-38, August.

CLUA, Esteban W. G., Dreux, M, Feijó, B. 2001. **A Shading Model for Image-based object rendering.** In: Proceedings of SIBGRAPI'01, p.138-145, October.

CLUA, Esteban W. G., Feijó, B., Dreux, M. 2003a. **Utilization of Nailboards for optimization in rendering distribution between CPU / GPU using Relief Texture Mapping**. Monografias em Ciência da Computação ISSN 0103-9741, MCC n. 28/03, Departamento de Informática, PUC-Rio.

CLUA, Esteban W. G., Feijó, B., Dreux, M. 2003b. **A Generic Avaliation Method for Image-Based Rendering Systems.** Monografias em Ciência da Computação, MCC n. 27/03, Departamento de Informática, PUC-Rio.

COOK, R. 1984. **Shade trees**. In: ACM SIGGRAPH Computer Graphics Proceedings, p. 223–231, July.

DALLY, W., et al. 1996. **The Delta Tree: An Object-Centered Approach to Image-Based Rendering.** MIT AI Lab Technical Memo 1604, May.

DAMON, W. 2003. **Impostors Made Easy**. Intel Technical Report. Disponível em http://cedar.intel.com.

DEBEVEC, P., TAYLOR, C. and MALIK, J. 1996. **Modeling and Rendering Architecture From Photographs: A Hybrid Geometry - and Image-Based Approach.** In: ACM SIGGRAPH Computer Graphics Proceedings, p. 11-20, August.

DEBEVEC, P., YU, Y., BORSHUKOV, G. 1998. **Efficient view-dependent image-based rendering with projective texture-mapping.** In: Proceedings of the 9th Eurographics Workshop on Rendering, p. 105–116. Springer-Verlag, Vienna, Austria.

EBERLY, D. 2000. **3D Game Engine Design - A Practical Approach to Real-Time Computer Graphics**. Morgan Kaufmann Publishers Inc, San Franciso.

ELHELW, M., YANG, G. 2003. **Cylindrical Relief Texture Mapping**. In: Journal of WSCG, Vol.11, No.1., Plzen, Czech Republic, February.

FERNANDO, R. and KILGARD, M. 2003. **The Cg Tutorial - The definitive guide to programmable Real-Time Graphics.** Addison Wesley and NVidia, Boston.

FLY3D, 2004. Disponível em www.fly3d.com. Acesso em: 23 jan. 2004.

FONSECA, F. 2004. **Texturas com Relevo utilizando Iluminação por Pixel e Processamento Paralelo.** Departamento de Informática, Puc-Rio, Rio de Janeiro, Dissertação de Mestrado.

FORSYTH, Tom. 2001. **Impostors: Adding Clutter**. In Mark DeLoura, ed., Game Programming Gems 2, Charles River Media, p. 488-496.

FUIJITA, M., KANAI, T. 2002. **Hardware-Assisted Relief Texture Mapping**. Proc. Eurographics 02, short presentation.

GOMES, J., VELHO, L. 1997. **Image Processing for Computer Graphics**. Springer-Verlag.

GORTLER, S., GRZESZUCZUK, R., SZELISKI, R. and COHEN, M. 1996. **The Lumigraph.** In: ACM SIGGRAPH Computer Graphics Proceedings, p. 43-54, August.

GORTLER, S., HE, L. and COHEN, M. 1997. **Rendering Layered Depth Images**. Technical Report, MSTR-TR-97-09.

GREENE, N. 1986. **Enviroment Mapping and other aplications of Worls projections.** In: IEEE Computer Graphics and Aplications, Vol. 6, no. 11, p. 21 - 29. November.

HEIGI, B., KOCH, R., et al. 1999. **Plenoptic Modeling and rendering from image sequences taken by hand-held camera.** In: Proceedings of DAGM 99, p. 94 – 101.

HUNTER, G., M. 1978. **Efficient Computation and Data Structure for Graphics.** Department of Electrical Engineering and Computer Science, Princeton University, Ph. D. thesis.

IGEHY, H., GORDON, S. and HANRAHAN, P. 1998. **The Design of a Parallel Graphics Interface.** In: ACM SIGGRAPH Computer Graphics Proceedings, p. 141-150, July.

INTEL CORPORATION. 2001. **Introduction to Hyper-Threading Technologies**. White paper from INTEL Data Research. Document Number 250008-002.

JAKULIN, Aleks. 2000. **Interactive Vegetation Rendering with Slicing and Blending.** In: Proceedings of Eurographics Workshop on Rendering. August.

KANG, S. and SZELISKI, R. 1996. **3D Scene Data Racovery Using Omnidirectional Baseline Stereo.** In: IEEE Computer Vision and Pattern Recognition (CVPR 96), p. 364-370.

KEMPF, R. and HARTMAN, J. 1998. **OPENGL on Silicon Graphics Systems,** Silicon Graphics Inc.

KOZOVITS, L. 2004. **Otimização de Mensagens e Balanceamento de Jogos Multi-Jogador**. Departamento de Informática, Puc – Rio, Rio de Janeiro, Dissertação de Mestrado, 170p.

LEE, S., CHWA, K., SHIN, S. and WOLBERG G. 1992. **Image metamorphosis using snakes and freeform deformations**. In: ACM SIGGRAPH Computer Graphics Proceedings, p. 439-448.

LEVOY, M. and HANRAHAN, P. 1996. **Light Field Rendering**. In: ACM SIGGRAPH Computer Graphics Proceedings, p. 31-42, August.

LIPPMAN, A. 1980. **Movie Maps: An Application on the Optical Videodisc to Computer Graphics**. In: ACM SIGGRAPH Computer Graphics Proceedings.

MACIEL, PAULO W. and SHIRLEY, PETER. 1995. **Visual Navigation of Large Environments Using Textured Clusters.** Symposium on Interactive 3D Graphics pp 95-102. April.

MARR, D., BINNS, F., et al. 2002. **Hyper-Threading Technology Architecture and Microarchitecture**. Intel Technology journal Q1.

MAX, N. and OHSAKI, K. 1995. **Rendering Trees from Precomputed Z-Buffer Views.** In: Proceedings of the 6th Eurographics Workshop on Rendering, Dubiln, Ireland, p. 45-54, June.

MCMILLAN, L. and BISHOP, G. 1995. **Plenoptic Modeling: An Image-Based Rendering System.** In: ACM SIGGRAPH Computer Graphics Proceedings, p. 39-46, August.

MCMILLAN, L. 1997. **An Image-Based Approach to Three Dimensional Computer Graphics.** Department of Computer Science, University of North Carolina at Chapel Hill, Ph. D. thesis.

MCREYNOLDS, T., BLYTHE, D., et al. 1999. **Advanced Graphics Programming Techniques Using OpenGL.** In: ACM SIGGRAPH course notes.

NYLAND, L., LASTRA, A., et al. 2001. **Capturing, Processing and Rendering Real-World Scenes.** Department of Computer Science, University of North Carolina Technical Report.

OH, B., CHEN, M., et al. 2001. **Image-based Modeling and Photo Editing**. In: ACM SIGGRAPH Computer Graphics Proceedings.

OLIVEIRA, M. and BISHOP, G. 1999. **Relief Textures**. Department of Computer Science, University of North Carolina at Chapel Hill, Technical Report TR99-015, March.

OLIVEIRA, M. 2000 (a). **Relief Texture Mapping.** Department of Computer Science , University of North Carolina, PhD thesis.

OLIVEIRA, M., BISHOP, G. and MCALLISTER, D. 2000 (b). **Relief texture mapping**.In: ACM SIGGRAPH Computer Graphics Proceedings, p. 359–368, November.

OLIVEIRA, M., BISHOP, G. 2002. **Image-Based Objects.** University of North Carolina at Chapel Hill, Technical Report.

PIGHIN, F., HECKER, J., et al. 1998. **Synthesizing realistic facial expressions from photographs.** In: ACM SIGGRAPH Computer Graphics Proceedings.

POLICARPO, F. 2002. **CG Relief Texture Mapping**. CG shaders contest. Disponível em: <http://www.cgshaders.org/shaders>. Acesso em: 9 mar. 2004.

POPESCU, V., LASTRA, A., et al. 1998. **Efficient Warping for Architectural Walktroughs Using Layered Depth Images.** In: Proceedings of IEEE Visualization.

POPESCU, V., Eyles, J., et al. 2000. **The WarpEngine: An Architecture for the**

Post-Polygonal Age. In: ACM SIGGRAPH Computer Graphics Proceedings, p. 433-442, July.

POPESCU, V. 2001. **Forward Rasterization: A Reconstruction Algorithm for Image-based Rendering**. Department of Computer Science, University of North Carolina, PhD thesis.

PULLI, K., COHEN, M., et al., 1997. **View-based Rendering: Visualizing real objects from scanned range and color data**. In: Eurographics Rendering Workshop, p. 23–34.

RADEMACHER, P. and BISHOP, G. 1998. **Multiple-Center-of-Projection Images**. In: ACM SIGGRAPH Computer Graphics Proceedings, p. 199-206.

RAFFERTY, M., ALIAGA, D. and LASTRA, A. 1998(a). **3-D Warping in Architectural Walkthroughs.** In Proceedings of VRAIS, p. 228-233, March.

RAFFERTY, M., ALIAGA, D., POPESCU, V. and LASTRA, A. 1998(b). **Images for Accelerating Architectural Walkthroughs.** In: IEEE Computer Graphics & Applications, vol. 18, no. 6, p. 38-45, November/December.

SAUER, L. **titulo da tese**. Departamento de Engenharia Mecânica, Puc – Rio, Rio de Janeiro, Dissertação de Doutorado.

SCHAUFLER, G. 1995. **Dynamically Generated Impostors**. In: Workshop on Modeling – Virtual Worlds – Distributed Graphics, D. W. Fellner, ed., Verlag, p. 129-135, November.

SCHAUFLER, G. 1997. **Nailboards: A Rendering Primitive for Image Caching in Dynamic Scenes.** In: Proceedings of the 8th Eurographics

Workshop on Rendering. St. Ettiene, France, Springer-Verlag, p. 151-162, June.

SCHAUFLER, G. 1998. **Per-Object Image Warping with layered Impostors**. In: Proceedings of the 9th Eurographics Workshop on Rendering. Vienna, Austria, p. 145-156, June.

SEITZ, S. and DYER, C. 1996. **View Morphing.** In: ACM SIGGRAPH Computer Graphics Proceedings, p. 21-30, August.

SHADE, J., GORTLER, S., et al. 1998. **Layered Depth Images**. In: ACM SIGGRAPH Computer Graphics Proceedings, p. 231-242, July.

SHUM, H., HE, L. 1999. **Rendering with concentric mosaics**. In: ACM SIGGRAPH Computer Graphics Proceedings, August.

SHUM, H-Y. and KANG, S. 2000. **A Review of Image-based Rendering Techniques.** In: IEEE/SPIE Visual Communications and Image Processing, p.2-13.

SZELISKI, R. 1996. **Video mosaics for virtual environments**. In: IEEE Computer Vision and Applications, p. 22-30.

SZELISKI, R. and SHUM, H. 1997. **Creating full view panoramic image mosaics and texture-mapped models**. In: ACM SIGGRAPH Computer Graphics Proceedings.

TAKAHASHI, T., KAWASAKI, H., et al. 2000. **Arbitrary View Position and Direction Rendering for Large-Scale Scenes.** In: IEEE Computer Vision and Pattern Recognition (CVPR 00), p. 296-303.

TELLER, S. and SÉQUIN, C. 1991. **Visibility Preprocessing For Interactive Walkthroughs**. In: ACM SIGGRAPH Computer Graphics Proceedings, p. 61-69, July.

TOBORG, J. and KAJIYA, J. 1996. **Talisman: Commodity Realtime 3D Graphics for PC.** In: ACM SIGGRAPH Computer Graphics Proceedings, p. 353-363, August.

WATT, A. and POLICARPO, F. 2001. **3D games, Real-time Rendering and Software Technology.** ACM Press.

WESTOVER, L. 1990. **Footprint Evaluation for Volume Rendering**. In: ACM SIGGRAPH Computer Graphics Proceedings, p. 367-376, August.

WOLBERG, G. 1990. **Digital Image Warping**. IEEE Computer Society Press, Los Alamitos, CA.

WOOD, D., AZUMA, D., et al. 2000. **Surface Light Fields for 3D photography**. In: ACM SIGGRAPH Computer Graphics Proceedings, July.