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*.

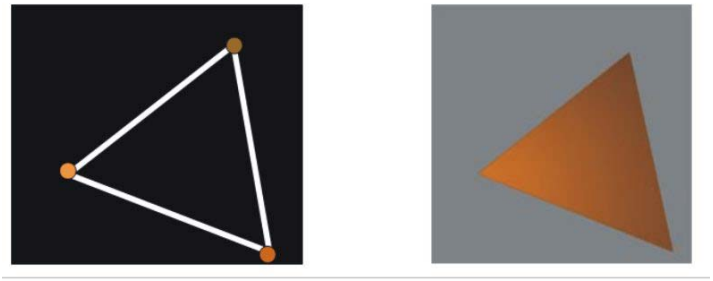


figura 6.1 – A iluminação de um polígono normalmente é feita pela interpolação das cores de cada um dos vértices que o compõem. Esta interpolação dará a ilusão de que cada *pixel* do interior foi iluminado individualmente, quando na verdade apenas os vértices o foram.

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.

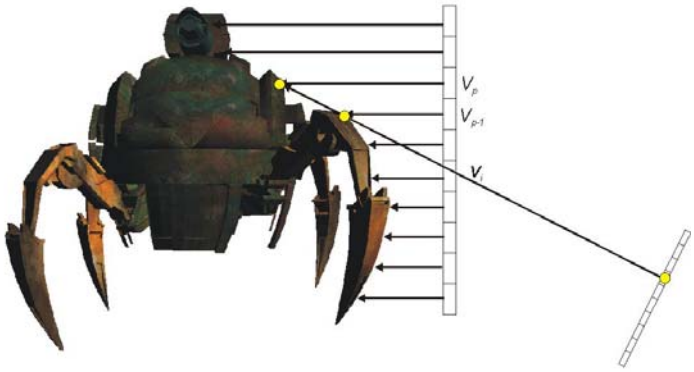


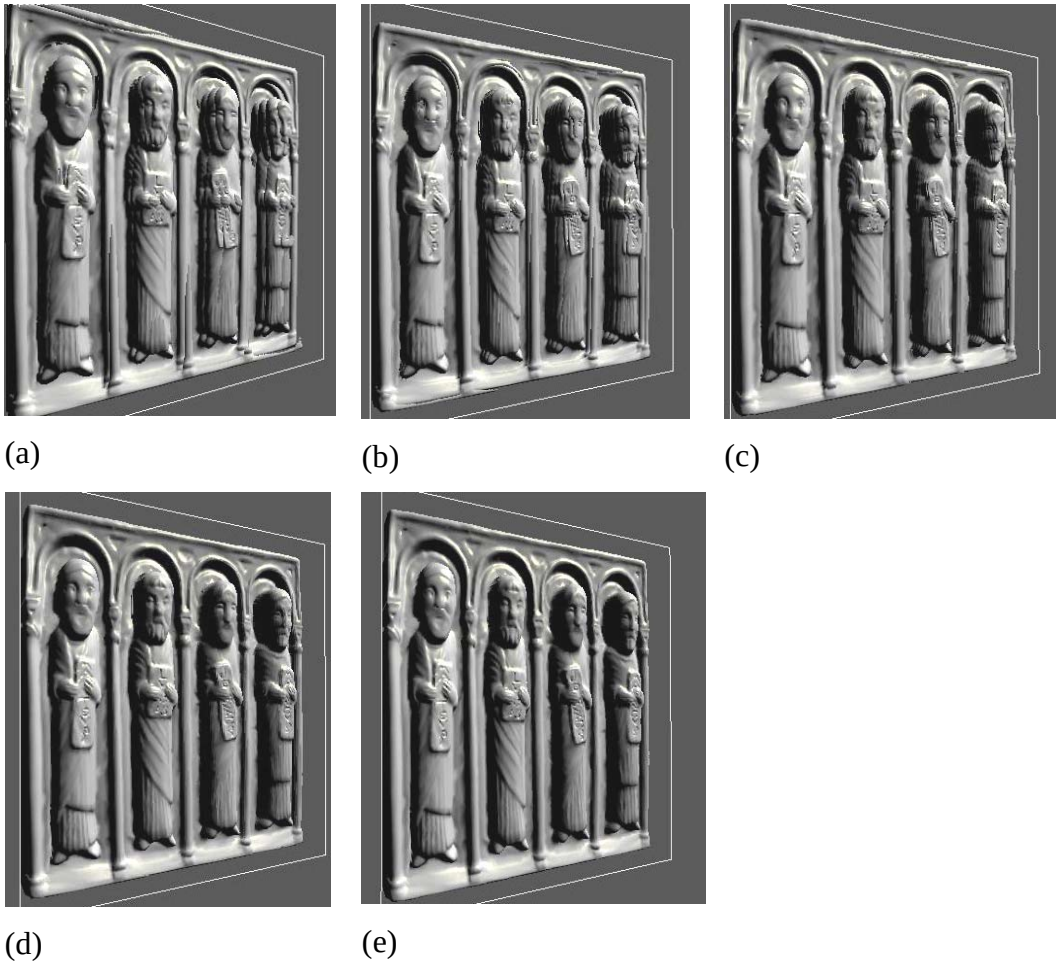
figura 6.2 – Dois *texels* distintos são mapeadas para o mesmo ponto no processo de *pre-warping* inverso.

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)

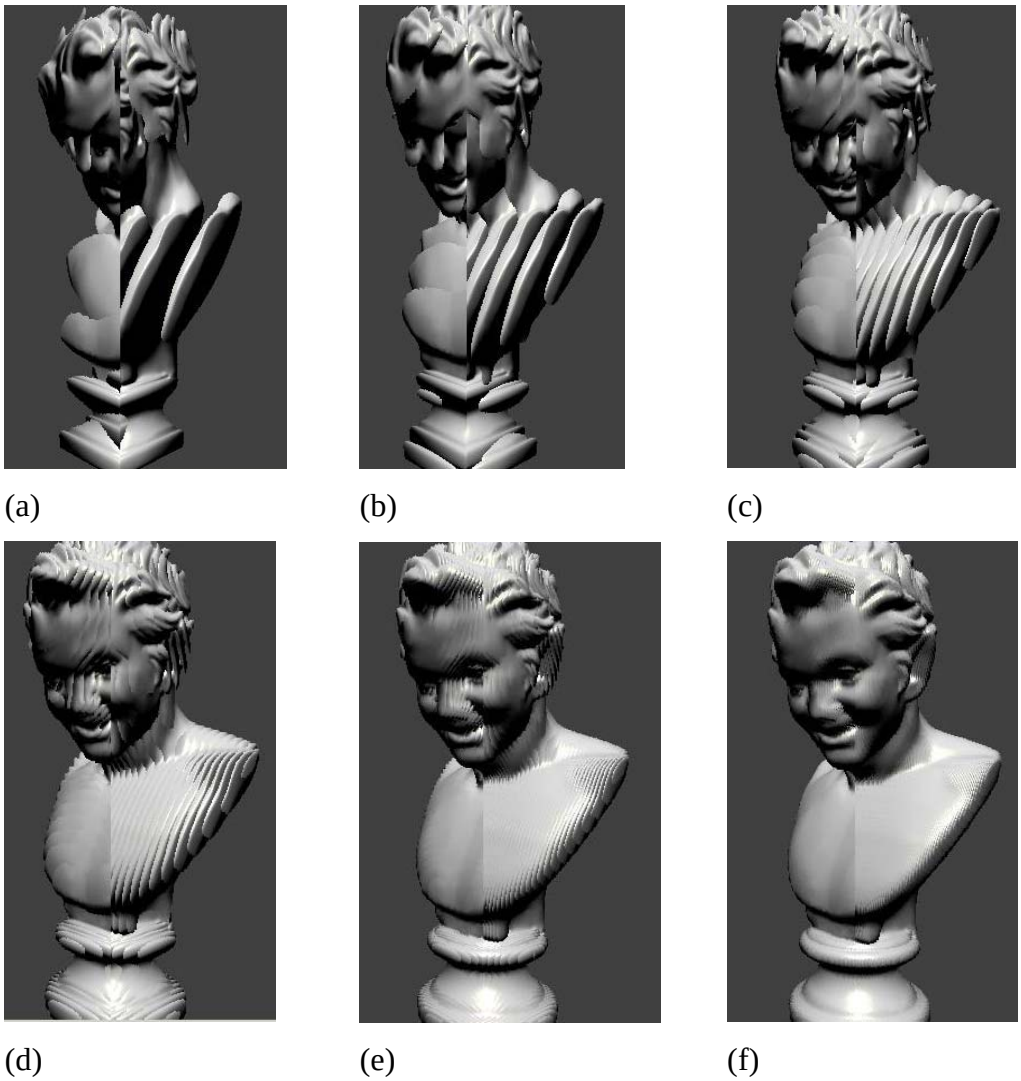
(d) (e)

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.