UTILIZAÇÃO DE NAILBOARDS PARA OTIMIZAÇÃO EM DISTRIBUIÇÃO DE RENDERING ENTRE CPU E GPU UTILIZANDO RELIEF TEXTURE MAPPING

ESTEBAN W. GONZALEZ CLUA[1]
MARCELO DREUX[2]
BRUNO FEIJÓ[1]

[1]ICAD – IGames, Departamento de Informática - PUC-Rio
{esteban, bruno}@inf.puc-rio.br

[2]ICAD – IGames, Departamento Engenharia Mecânica
dreux@mec.puc-rio.br

Resumo

Com o crescimento do poder dos hardwares gráficos e especialmente com os recursos de pipeline programável, uma aplicação gráfica disponibiliza cada vez mais tempo ocioso para a CPU. No sentido de aproveitar este tempo, este trabalho apresenta uma forma de paralelizar o rendering entre CPU e GPU, utilizando um thread da CPU e a técnica de Relief Texture Mapping para inserir os resultados no pipeline do hardware. Esta proposta permite que uma série de efeitos inapropriados ou difíceis de serem atingidos pela visualziação de hardware possam ser tratados pela CPU sem perdas na performance total da aplicação.

Keywords: *Image-based rendering, Sprites, Per Pixel shading*

1 Introdução

Os hardwares gráficos atualmente podem ser considerados como verdadeiros processadores e, em muitos casos, melhores que as CPU's. Algumas placas gráficas comerciais ultrapassaram a barreira dos 120 milhões de transistores (mais do que um chip Intel Pentium IV), podem acessar até 16GB de memória, são capazes de processar 4 ou mais pixels simultaneamente, renderizar para 2 monitores também simultaneamente e algumas placas com suporte a programação para o seu pipeline possuem 2 processadores. Para ressaltar o caráter de que este hardware dedicado não é simplesmente um acessório, mas sim um processador dedicado, costuma-se denominá-los também de GPU (Graphic Processor Unit) ou VPU (Visual Processor Unit). Assim sendo, pode-se considerar que uma máquina munida de uma GPU é uma máquina paralela, embora um dos processadores seja de uso dedicado a aplicações gráficas (GPU) e o outro seja de uso genérico (CPU).

Até poucos anos o pipeline de rendering por hardware era pré-definido pela sua arquitetura interna. Assim, por melhor que fosse a sua performance ou a performance da CPU, as imagens geradas em tempo real apresentavam certos padrões ditados pelos algoritmos implementados em hardware. Atualmente é possível programar este pipeline gráfico, alterando o pipeline de visualização quantas vezes for necessário numa aplicação, podendo assim, num mesmo frame, utilizar vários algoritmos diferentes, cada um mais apropriado para determinada situação. Com estas técnicas é possível gerar imagens em tempo real com efeitos de iluminação mais sofisticados, simulação de reflexos, cartoon rendering, efeitos metálicos, caustics, etc.

Assim sendo, em aplicações gráficas, cada vez mais têm crescido a tendência de que o hardware gráfico assuma a maior parte do pipeline de visualização, deixando a CPU com um tempo ocioso (IDLE) cada vez maior. Na realidade, o trabalho gráfico da CPU têm se tornado cada vez mais burocrático e limitado a

alimentar a placa gráfica com os dados relacionados à geometria, texturas e quando muito resolver alguns cálculos de otimização. Diversas aplicações têm-se aproveitado deste tempo ocioso para realizar operações mais complexas, tais como operações de inteligência artificial. Entretanto poucos trabalhos têm sido feitos no sentido de assumir a CPU|GPU como uma máquina paralela para um mesmo pipeline gráfico.

A idéia deste trabalho consiste em aproveitar o idle time da CPU para operações gráficas tempo real, escolhendo operações que seriam inapropriadas para a GPU, sem no entanto comprometer a performance da aplicação. Assim sendo, têm-se um pipeline gráfico duplo: um que é executado pela GPU (hardware) e outro que é executado pela CPU (software). Este tipo de arquitetura por si só já corresponderia, na prática, a encarar um sistema com uma CPU e uma GPU como sendo uma máquina paralela. Entretanto, deve-se desenvolver uma arquitetura que permita que a GPU nunca tenha outra latência, a não ser aquelas geradas pela sua própria limitação de processamento, de forma a garantir que a visualização não deixe de ser tempo-real.

Neste trabalho será apresentado um sistema capaz de distribuir a visualização para a CPU, que calculará a visualização através de uma técnica de ray-tracing adaptado, repassando o resultado para a GPU através de métodos de image-based rendering implementado em hardware, e que calculará o restante da visualização através do seu pipeline convencional.

2 Trabalhos Relacionados

2.1 Impostors [14] e Nailboard [15]

A idéia que está por trás dos impostors ou nailboards consiste em representar um objeto tridimensional através de um sprite, porém diferentemente que no conceito padrão de sprites ou billboards, estes objetos estão realmente definidos como modelos e são renderizados durante a aplicação corrente e projetados num plano como textura com alpha. [14] e [15] descrevem estes métodos como adequados para minimizar o número de vezes em que se deve visualizar algum determinado objeto, que possui certa complexidade geométrica[5] e portanto cara para o pipieline de tempo real (os impostors serviriam como uma espécie de buffer para estes objetos, de forma que são reaproveitas enquanto ainda "servem" para uma determinada posição do observador). Além disso, a imagem gerada para estes objetos pode ser proporcional ao tamanho que ocupam na imagem final. Assim, se estiverem muito distantes, serão renderizados numa resolução pequena, 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 [3] descreve-se uma forma de implementar impostors utilizando técnicas de rendering em memória de vídeo, recurso que está disponível nas placas gráficas mais recentes, podendo aumentar o número de objetos sendo representados com esta técnica, desde que haja memória de vídeo suficiente.

Nesta pesquisa, o conceito de impostors será estendido para realizar a separação dos objetos gerados por CPU dos gerados por GPU. Os impostors corresponderão aos elementos gerados pela CPU e serão passados diretamente para a GPU, na forma de sprites com relief mapping.

Em trabalhos semelhantes usa-se o mesmo conceito para reduzir cenas de natureza geométrica complexa num conjunto de imagens. Em [12] e [1] 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 serão geradas imagens separadamente e utilizadas como texturas projetadas nas faces adjacentes à célula onde o observador se encontra, utilizando-se alguma técnica de image warping para corrigir erros de parallax que possam surgir.

2.2 Sprites com profundidade [13]

A idéia desta técnica consiste em aumentar o poder de representação dos sprites realizando deslocamentos ortogonais dos pixels da textura que o representa. 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 pela *epipole* (projeção do centro de projeção de uma determinada camera sobre o plano de visualização de outra camera) da imagem sendo formada e adicionada ao resultado da homografia. Estas coordenadas serão usadas para indexar a cor dos pixels de destino.

2.4 Per-Pixel Shading[4]

Esta tecnologia, que está se tornando comum entre os diversos fabricantes de hardware gráficos. Permite que algumas operações do pipeline gráfico sejam alterados pelo desenvolvedor. Muitas destas operações consistem em manipulações diretas sobre os texels, tais como operações sobre sua cor, deslocamento e offset. Permite-se que algumas operações sejam feitas com os fragmentos de uma imagem (um pixel ou um conjunto de pixels), podendo-se por exemplo fazer um produto escalar, comparações, multiplicações entre estes pixels e outros ou entre os pixels e outros números presentes em variáveis armazenadas em registradores do hardware ou em tabelas.

Utilizando-se Per-Pixel Shading, vários efeitos que antes eram impossíveis de serem obtidos em tempo real, passaram a ser viáveis, tais como o bump-mapping ou o reflection-mapping. Mais especificamente, alguns recursos de processamento de imagens poderão ser tratados em tempo real pela placa gráfica.

3 3D Image Warping [8, 9]

O warping 3D de imagens consiste numa função de transformação geométrica $w:U' \rightarrow W\grave{I}\ \boldsymbol{R^2}$ capaz de mapear uma imagem fonte i_f para uma imagem destino i_d. A imagem fonte contém, além das cores dos pixels, um mapa de profundidade para cada pixel. Além disso, sabe-se os dados relacionados ao observador desta imagem (posição e plano de projeção). Utilizando o modelo de camera perspectiva tem-se que um ponto $\dot{x}$ pertencente à cena geométrica pode ser representado pela equação:

$$\dot{x} = \dot{C}_f + (\vec{c} + u_f \vec{a} + v_f \vec{b}).t_f(u_f, v_f) \quad (1)$$

Onde $\dot{C}_f$ é a posição geométrica da camera, (u_f, v_f) é a coordenada da projeção deste ponto sobre o plano de projeção da camera, 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 com a direção dada pelo centro de projeção à origem do plano da imagem e o coeficiente $t_f(u_f, v_f)$ é dado pela fração 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 corresponde a este cálculo:

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

A equação (1) pode ser rescrita utilizando uma operação de matrizes da seguinte maneira:

$$\dot{x} = \dot{C}_f + \begin{bmatrix} a_i & b_i & c_i \\ a_j & b_j & c_j \\ a_k & b_k & c_k \end{bmatrix} \begin{bmatrix} u_f \\ v_f \\ 1 \end{bmatrix} t_f(u_f, v_f) = \dot{C}_f + P.\vec{x}.t_f(u_f, v_f) \quad (3)$$

Para uma posição do observador $\dot{C}_d$, é trivial ver que $\dot{x}$ pode ser descrito através da equação:

$$\dot{x} = \dot{C}_d + P_d.\vec{x}_d.t_d(u_d, v_d) \quad (4)$$

Em [8], o autor demonstra que $\vec{x}_d.t_d(u_d, v_d) = P_d^{-1}[P_f \vec{x}_f t_f(u_f, v_f) + (\dot{C}_f - \dot{C}_d)]$, de onde se pode concluir que vale a seguinte equivalência projetiva (o vetor resultante possui a mesma direção e sentido, podendo ser apenas o módulo diferente):

$$\vec{x} \doteq P_d^{-1}[P_f.\vec{x}_f.t_f(u_d, v_d) + (\dot{C}_f - \dot{C}_d)] \quad (5)$$

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

Para chegar a formulação final da equação de 3D image warping de McMillan, divide-se a equação (5) por $t_f(u_f, v_f)$ e distribui-se a matriz inversa de projeção P_d^{-1}:

$$\vec{x} \doteq P_d^{-1} P_f.\vec{x}_f + P_d^{-1}(\dot{C}_f - \dot{C}_d).\partial_f(u_f, v_f) \quad (6)$$

Onde $\partial_f(u_f, v_f) = 1/t_f(u_f, v_f)$ será denominada de disparidade generalizada do pixel (u_f, v_f) da imagem fonte. A primeira parcela da soma da equação de McMillan representa uma transformação perspectiva planar homográfica sobre a imagem fonte (que pode ser visto como uma projeção de textura) e a segunda parcela equivale a uma transformação per-pixel proporcional ao valor da disparidade generalizada do pixel, dada por $\partial_f(u_f, v_f)$ na direção da epipolar do plano da imagem destino.

4 Relief Mapping [10, 11]

O método consiste numa fatorização da equação de 3D image warping de McMillan [8, 9] em duas etapas separadas: uma de pré-warping e outra de texturização padrão. O pré-warping é aplicado a imagens com mapa de profundidade para cada texel, transformando-as em imagens simples, corrigindo o efeito de parallax resultante do deslocamento do observador e de partes da textura. A etapa seguinte realizará as operações de escala, rotação e as deformações de perspectiva necessárias para o mapeamento correto nos polígonos do Relief-Texture.

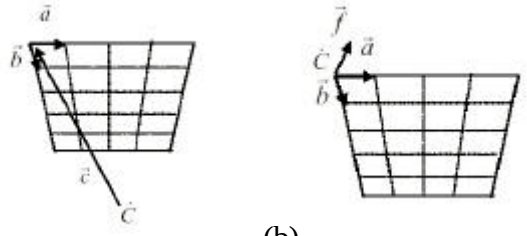
Esta fatorização pode também ser interpretada da seguinte forma: a imagem fonte sofre primeiramente uma pré-deformação, que depende das informações da imagem e da posição do observador destino. Esta pré-deformação é feita no próprio plano de projeção da imagem para o observador fonte. Feito isto, utilizando a técnica padrão de mapeamento de textura, projeta-se a imagem deformada sobre o plano correspondente ao observador destino.

A etapa de pré-warping, por sua vez, pode ser interpretada como sendo o 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 este novo par corresponda ao local onde (u_f, v_f) deveria estar no plano de projeção do observador fonte.

Em [8] demonstra-se que este deslocamento relativo ao pixel da imagem do plano do observador fonte possui dependência apenas do parâmetro posição em relação ao observador destino. Desta maneira, uma série de simplificações podem ser feitas na equação de McMillan, chegando-se à seguinte formulação:

$$u_i = \frac{u_f - k_1 \partial(u_f, v_f)}{1 + k_3 \partial(u_f, v_f)} \text{ e } v_i = \frac{v_f - k_2 \partial(u_f, v_f)}{1 + k_3 \partial(u_f, v_f)} \quad (7)$$

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 os coeficientes de *a*, *b*, *c* e *f* são referentes aos vetores ilustrados na figura 1.



(a) (b)

Figura 2 – (a) corresponde a um modelo de camera 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 que do centro de projeção à origem do plano da imagem. (b) corresponde ao modelo de camera ortogonal, 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.

Resumidamente, o algoritmo que realizará o Relief-Mapping pode ser descrito da seguinte maneira:

Determine-se o $\dot{C}_d$ e calcule-se as constantes k_1, k_2 e k_3;

Realizar o pré-warping partindo-se da imagem fonte:
- *Calcular a posição (u_d, v_d) para cada pixel (u_f, v_f), usando a eq. (7);*
- *Copiar a cor do pixel da imagem fonte para as posições de pré-warp;*
- *Tratar o conflito de pixels sobre uma mesma área e os buracos surgidos;*

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

5 Implementação de Relief Mapping por hardware[6]

Como foi visto, uma das principais contribuições de [10] consiste em decompor a equação de 3D image warping de McMillan numa operação unidimensional sobre uma imagem (etapa de pré-warping) seguida de uma operação de texturização tradicional. Esta decomposição torna possível que o 3D Image Warping seja implementada num hardware com pipeline programável. Embora o autor de [10] cite esta possibilidade, apenas em [6] se apresenta uma forma de implementação, utilizando per-pixel shading, embora possam haver várias abordagens diferentes, dependendo especialmente dos recursos de hardware disponíveis.

O algoritmo de Relief Mapping apresentado no capítulo anterior será substituído pelo seguinte:

Determine-se o $\dot{C}_d$ e calcule-se as constantes k_1, k_2 e k_3;

Para cada pixel da imagem fonte (u_f, v_f) gere-se um mapa de deslocamento (offset map):
- *Utilizando-se a equação (7) de pré-warping, calcule-se a posição de cada (u_d, v_d);*
- *Copie-se o endereçamento de um pixel (u_f, v_f) da imagem fonte como um valor de RGB para a imagem de pré-warping, criando-se um offset map;*
- *Tratar o conflito de pixels sobre uma mesma área, bem como os buracos;*

Renderizar a imagem destino, usando a tabela offset map para buscar a referência do pixel que lhe corresponde na imagem fonte;

Para efetuar o último passo do algoritmo, usa-se a função *OFFSET_TEXTURE_2D*, que é um texture shader. Esta função, que está implementada em hardware, permite pintar um texel de uma determinada textura buscando a cor num texel de outra, com um determinado offset (dado pelo *offset map*). Este mesmo *offset map* pode ser usado para determinar a normal correspondente ao pixel, no caso de haver um

mapa de normais, e através de um pixel shader calcular a iluminação correspondente.

6 3D Warping com visualização progressiva utilizando Relief Mapping

Num nailboard comum, o tempo de validade da imagem depende diretametne do movimento do observador ou do movimento do objeto sendo representado pelo sprite. Utilizando um relief mapping, o tempo de vida do nailboard aumenta consideravelmente, devido à sua capacidade de corrigir os problemas relacionados ao parallax até um determinado ponto de vista.

A idéia central deste trabalho consiste em criar um algoritmo capaz de sintetizar nailboards de alguns objetos de uma cena através de métodos implementados em CPU. Estas imagens são armazenadas em buffers do sistema e copiadas para frame buffers da GPU. Além de gerar a imagem, deve-se gerar um mapa de profundidade da mesma.

Simultaneamente será gerado um mapa de offset para o pré-warping e que será necessário para visualizar esta imagem como relief mapping.

Os objetos de uma cena serão divididos em 2 clusters distintos, sendo que um cluster receberá um tratamento por hardware (cluster de GPU) e outro por software (cluster de CPU). O segundo conjunto deve ser limitado, dependendo da capacidade do sistema e da complexidade da cena e deve haver uma razão para que os objetos se encontrem neste grupo:

- Deseja-se utilizar para estes um modelo de iluminação inapropriado de se implementa por hardware;
- Estes objetos não podem ser tratados como polígonos (NURBS, volumes, partículas, etc.);
- Deseja-se aplicar-lhes algum efeito especial que exija cálculos complexos (motion blur, refração) e inapropriados para a GPU.

O algoritmo inicia-se criando um thread que será responsável pelo cálculo de visualização através de um algoritmo implementado em software. Este processo será chamado de thread de rendering paralelo e deve ser criado com uma prioridade pequena, para garantir que a CPU não prejudique ao pipeline do hardware. Opcionalmente, no caso de uma máquina com processadores paralelos ou com hyper-threading, um processador pode ser designado a estar exclusivamente dedicado a este algortimo.

O pipeline da GPU deve ser considerado prioritário, sendo que poderá interromper a CPU a qualquer momento, quando lhe for necessário (somente assim se poderá garantir que a aplicação seja em tempo real).

O thread de rendering paralelo deve calcular uma nova imagem apenas quando uma das duas condições for observada:

- Houve uma transformação espacial ou geométrica do cluster de CPU;
- O observador deslocou-se o suficiente para que a imagem disponível do cluster de CPU passe a estar obsoleta.

No caso de nenhuma destas condições serem satisfeitas, a GPU não precisa acessar à CPU, pois ainda dispõem de um nailboard válido. Neste caso, a CPU deve aproveitar o tempo que estaria ociosa, gerando conjuntos de relief-mapping para o cluster de CPUs com um sistema de previsão inteligente: Se o observador está se movendo com uma determinada velocidade, deve-se prever a posição em que estará passados alguns frames e começar a calcular o nailboard que seria necessário para esta futura posição. O mesmo vale para o caso do cluster estar se movendo.

Imagens que se tornam obsoletas para a GPU, através do critério de [14] não devem ser descartadas pela CPU: na medida do possível, deve-se criar um cachê com os relief textures mais recentes, pois em caso de necessidade, não será necessário calculá-las novamente. Em alguns casos podem vir a ser necessárias imagens para um mesmo plano de projeção de um $\dot{C}_d$ previamente calculado, porém com

resolução diferente. Neste caso, é conveniente reaproveitar a imagem que já se havia gerado, fazendo apenas uma reamostragem.

Dependendo do método de rendering que se estiver utilizando, é conveniente utilizar métodos progressivos, pois no caso de não dar tempo da CPU terminar e a GPU fizer a requisição de um nailboard, este envio deve ser imediato, sendo necessário enviar o que se tem disponível no momento.

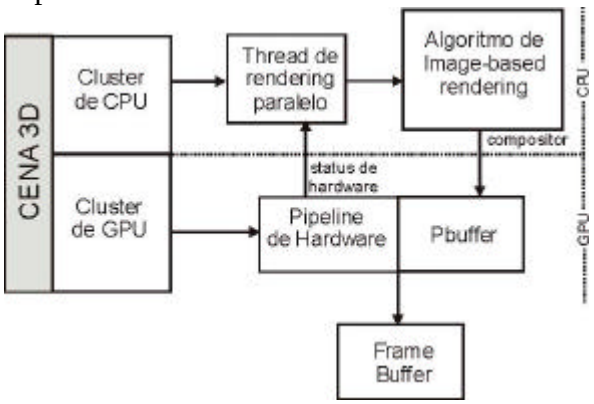


Figura 3 - Diagrama do modelo de rendering paralelo CPU|GPU proposto.

7 Medida de Erro

Enquanto no pipeline padrão da GPU todos os objetos devem ser renderizados a cada frame, mesmo que nada tenha sido alterado em relação aos objetos e ao observador, os nailboards com relief mapping não precisam ser recalculados enquanto haja uma coerência visual, podendo assim ser reutilizada durante vários frames seguidos uma mesma imagem.

Além disso, a resolução que será adotada para a imagem a ser gerada dependerá da distância em que o objeto se encontra em relação ao observador. Isto evitará que seja desperdiçado um tempo grande gerando pixels que depois serão perdidos por motivo de amostragem.

Para determinar o tamanho inicial da imagem a ser criada pela CPU utiliza-se a seguinte equação:

$$res\_sprite = res\_tela \frac{Tamanho\_do\_objeto}{2.distância.tg(FOV/2)} \qquad (8)$$

Exitem basicamente duas situações em que a imagem do nailboard se torna obsoleto, sendo portanto necessário um novo cálculo de visualização para atualizá-lo:

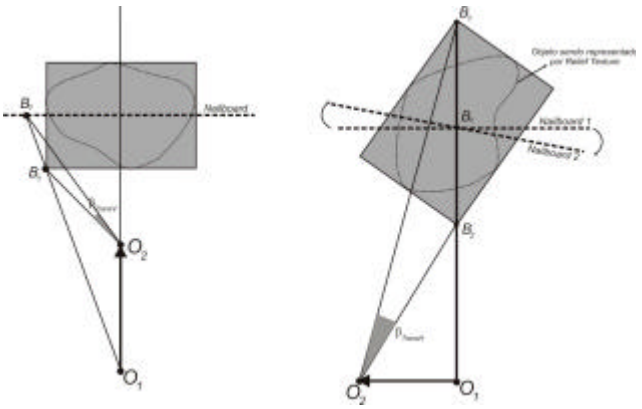
- O observador se aproximou do objeto do cluster de CPU, sendo necessária uma imagem do nailboard com maior resolução do que a anterior;
- O observador se moveu num plano paralelo ao plano de projeção, podendo surgir erro de paralax do objeto mapeado.

Note-se que qualquer alteração do objeto na imagem gerada consiste numa composição destas duas transformações.

Para determinar o momento em que se torna necessário calcular uma nova imagem, utiliza-se o método proposto em [14], que realiza um teste para cada um dos casos mencionados acima.

Para o primeiro caso, utiliza-se um dos extremos de uma Bounding Box criada para o objeto sendo representado pelo nailboard, conforme se pode ver na figura 3 (a). O_1 corresponde à posição inicial do observador e O_2 à posição que se deseja testar se há erro e portanto é necessário gerar uma nova imagem do nailboard. Sempre que $b_{transV} > b_{tela}$ deve-se recalcular um novo nailboard.

Para o segundo caso, mede-se o deslocamento de parallax, que consiste no deslocamento aparente do objeto para duas posições distintas. O ângulo do erro é determinado pela diferença provocada pela projeção dos dois extremos da Bounding Box do elemento sendo representado pelo nailboard (figura 3 - b) e neste caso um novo sprite deve ser renderizado quando $b_{transH} > b_{tela}$.



(a) (b)

Figura 3 – (a) medida de erro para movimento de aproximação do observador ao objeto. (b) medida de erro para movimento paralelo ao plano de projeção [14].

Além disso, é fundamental minimizar a resolução do nailboard, de forma a não ter que calcular pixels que depois serão desperdiçados por problemas de amostragem. Para isso, deve-se determinar se um texel do nailboard se tornou maior que um pixel da imagem gerada do observador, realizando-se o seguinte teste (ilustrado na figura 4):

$$b_{naiboardl} > K.b_{tela} \qquad (9)$$

Sendo que $b_{nailboard} = \frac{FOV}{resolução\_nailboard}$ e $b_{tela} = \frac{FOV}{resolução\_tela}$.

A constante K permite que se coloque uma margem de erro, permitindo que o texel do nailboard possa ser um pouco maior do que o pixel da tela, uma vez que a diferença pode ser imperceptível. (o valor desta constante aumenta consideravelmente ao utilizar-se o Relief-mapping ao invés de imagens normais).

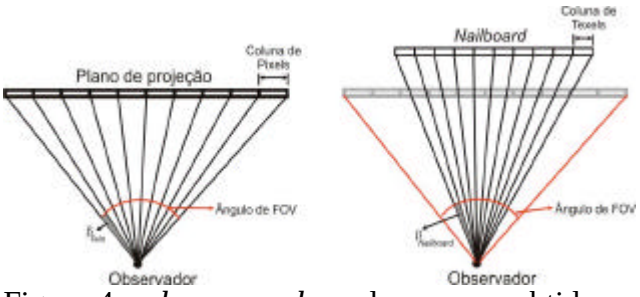


Figura 4 – $b_{naiboard}$ e b_{tela} devem ser obtidos a cada frame, para determinar se o nailboard se tornou obsoleto em relação à resolução da tela.

8 Implementação

O sistema implementado utiliza um algoritmo de Ray-tracing padrão para renderizar os objetos do cluster de CPU. A cena é descrita por 2 conjuntos distintos de objetos, de forma que a visualização da CPU não tenha que realizar interseção com todos os objetos presentes na cena. O processo de distribuição foi implementado de duas formas distintas: uma com time-slice e outra com threads. Como a máquina utilizada inicialmente não possui processamento paralelo ou sistema de hyper-threading, a performance foi muito parecida para os 2 métodos.

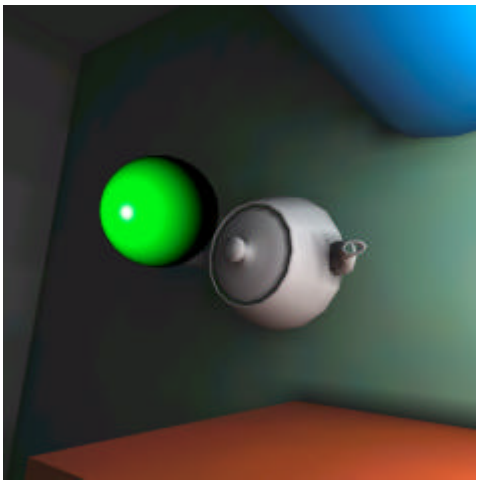


Figura 5 – Enquanto a esfera é calculada pela CPU com algoritmo de ray-tracing, o restante da cena está sendo calculado pela GPU.

Num Pentium IV 2.4GHz, com uma placa gráfica ATI Radeon 9700 Pro, a taxa real de visualização possui uma média de 171 fps. Sem reaproveitamento inteligente de imagens e sem o sistema métrico de Schaufler, o cluster de CPU possui uma taxa de atualização de 7 fps. Sem este cluster, a taxa de visualização permanece a mesma.

8 Conclusão e Trabalhos Futuros

Pode ocorrer que uma aplicação exija muito trabalho da CPU, dando pouco espaço de tempo para que o thread de rendering paralelo calcule a visualização dos objetos que lhe correspondam. Para tanto, pode-se criar um thread dedicado num processador paralelo. Fazendo-se isto, sempre haverá espaço de tempo para que a visualização da CPU seja calculada. Para uma implementação adequada deste sistema, sugere-se a utilização da tecnologia de Hyper-threading. 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 a arquitetura interna da CPU, que é composto por dois processadores lógicos. Do ponto de vista de 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 serão alocadas para partes diferentes da CPU e serão executadas individual e simultaneamente. Cada um destes poderá usufruir de suas próprias interrupções, pausas, finalizações, etc., porém a nível de hardware, a CPU consiste em dois processadores lógicos, mas que diferentemente do que processadores duais, compartilham uma série de componentes como os recursos do core do processador, cachê, 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 [7].

Outra possível otimização consiste em verificar a distância em que o objeto se encontra e, caso esteja mais afastado do que um determinado valor, o objeto pode deixar de ser tratado com relief-mapping e passa a ser tratado como simples sprite. Além disso, pode-se colocar uma adaptação dinâmica para a métrica de erro de Schaufler, fazendo com que na medida que o objeto esteja mais afastado do observador a constante k seja cada vez maior, uma vez que o erro do paralax irá diminuindo.

9 Bibliografia

1. Aliaga, D. *MMR: An Integrated Massive Model Rendering System Using Geometric and Image-Based Acceleration. Proceedings of 1999 ACM Symposium on Interactive 3D Graphics*. Atlanta, Ga, April 26-28, 1999, pp. 199-206.

2. Clua, E.; Dreux, M.; Feijó, B. A Shading Model for Image-based object rendering. Sibgrapi 2001.

3. Damon, William. *Impostors Made Easy*. Intel Technical Report. http://cedar.intel.com, 2003.

4. Fernando, Randima; Kilgard, Mark J. *The CG Tutorial – The definitive guide to programmable Real-Time Graphics*. Addison-Wesley and NVidia, February 2003.

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

6. Fujita, Masahiro; Kanai, Takashi. *Hardware-Assisted Relief Texture Mapping*. Proc. Eurographics 02, *short presentation*, 2002.

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

8. McMillan, L.; Bishop, G. *Plenoptic Modeling: An Image-Based Rendering System.* Siggraph'95 Computer Graphics Proceedings.

9. McMillan, L. *An Image-Based Approach to Three-Dimensional Computer Graphics.* Ph.D. Dissertation.

UNC Computer Science Technical Report TR97-013, University of North Carolina, April 1997.

10. Oliveira, Manuel M.; Bishop, Gary. *Relief Textures*. Department of Computer Science, University of North Carolina at Chapel Hill, Technical Report TR99-015, March 1999.

11. Oliveira, Manuel; Bishop, Gary; McAllister, D. *Relief texture mapping*. In *Computer Graphics (Proc. SIGGRAPH 2000)*, pages 359–368. ACMPress, New York, 2000.

12. Rafferty, M.; Aliaga, D.; e Lastra, A. *3-D Warping in Architectural Walkthroughs. VRAIS'98*. March 14-18, 1998. Pp. 228-233.

13. Shade, J., et al. *Layered Depth Image*s. Proc. SIGGRAPH 98 (Orlando, FL, July 19-24, 1998). In Computer Graphics Proceedings. Annual Conference Series, 1998, ACM SIGGRAPH, pp. 231-242.

14. Schaufler, G. *Dynamically Generated Impostors*. GI Workshop on Modeling – Virtual Worlds – Distributed Graphics, D. W. Fellner, ed., Infix Verlag, pp. 129-135, November 1995.

15. Schaufler, G. Nailboards: A Rendering Primitive for Image Caching in Dynamic Scenes. Proceedings of the 8th Eurographics Workshop on Rendering. St. Ettiene, France June 16-18, 1997. Rendering Techniques '97, Springer-Verlag, pp. 151-162.