EFICIÊNCIA E PRECISÃO NO CÁLCULO DE ILUMINAÇÃO POR RADIOSIDADE COM GPUS

ALESSANDRO RIBEIRO DA SILVA
Orientador: Renato Antônio Celso Ferreira

EFICIÊNCIA E PRECISÃO NO CÁLCULO DE ILUMINAÇÃO POR RADIOSIDADE COM GPUS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Belo Horizonte
Novembro de 2008

Silva, Alessandro Ribeiro
S586e Eficiência e precisão no cálculo de iluminação por radiosidade com GPUs [manuscrito] / Alessandro Ribeiro da Silva. — Belo Horizonte, 2008
xiv, 66 f. , enc. : il : 29cm

Orientador: Renato Antônio Celso Ferreira

Dissertação (mestrado) — Universidade Federal de Minas Gerais. Departamento de Ciência da Computação.

1. Computação - Teses. 2 Processamento digital de imagens - Teses. 3 Processamento paralelo - Teses. I Ferreira, Renato Antônio Celso. II Universidade Federal de Minas Gerais. Departamento de Ciência da Computação. III Título.

CDU 519.6*83

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Eficiência e Precisão no Cálculo de Iluminação por Radiosidade com GPUs

ALESSANDRO RIBEIRO DA SILVA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. RENATO ANTÔNIO CELSO FERREIRA - Orientador
Departamento de Ciência da Computação - UFMG

DR. JOÃO LUIZ ELIAS CAMPOS
Bolsista de Pesquisa DTI/FINEP - DCC - ICEx - UFMG

PROF. LUIZ CHAIMOWICZ
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 24 de novembro de 2008.

Resumo

Esse trabalho aborda um estudo sobre o método de iluminação global conhecido como radiosidade, num esforço para obter uma implementação eficiente para determinar a mesma.

A radiosidade é um método clássico e muito importante para a computação gráfica ao sintetizar imagens foto-realísticas, pois foi um dos primeiros a considerar informações de vários objetos para determinar a iluminação de uma cena. Como o método apresenta uma grande demanda de processamento, foi utilizada a GPU como forma de otimização, pois a mesma possui uma capacidade de processamento massivo considerável.

Foram realizadas duas implementações distintas que utilizam-se da GPU para acelerar alguma parte ou todo o algoritmo. A primeira é semelhante ao trabalho original sobre radiosidade, onde foi utilizada a resolução de sistemas lineares na GPU. A segunda é semelhante a uma extensão do algoritmo de radiosidade voltada para execução em uma GPU.

Como resultado, foram comparados diferentes métodos para resolver sistemas de equações lineares em ambos, GPUs e CPUs, e foi proposta uma abordagem para melhorar a determinação de visibilidade da implementação relacionada a GPU. As modificações melhoram a qualidade das imagens geradas enquanto preservam as características interativas da variação do método.

Abstract

This work consists of a study about the global illumination model known as radiosity, in an effort to achieve an efficient implementation on modern hardware.

Radiosity is a classic and very important method in computer graphics. It is the first proposed method to achieve photo-realism as it considers interactions among the objects of the scene. As the method has a high processing demand, we propose the use of modern GPUs, with their massive processing capacity, in an attempt to speed up the computation.

We experimented with two distinct implementations that use GPUs to accelerate part or all of the method. The first implementaion is similar to the original proposal of radiosity, and the GPUs were used to solve the simultaneous linear equations. The second implementation is derived from an extension to the original radiosity method that was designed for processing in GPUs.

As results, we compared different methods to solve the simultaneous linear equations, on both GPUs and CPUs and we proposed an improvement on the GPU implementation related to the determination of visibility. Our modifications improves the quality of the output image while preserving the interactive charateristic of the method variation.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Um dos principais objetivos da computação gráfica é a reprodução, através de renderizações, de fenômenos percebidos na natureza. Dentre os diversos fenômenos como sombras, reflexões, nuvens, água, atmosféra e etc, foram estudadas formas de se representar e determinar a iluminação de objetos opacos. Um modelo que se tornou bastante conhecido na área é o modelo de iluminação de Phong [1973]. Foram criados modelos mais precisos para representar tipos específicos de objetos, como é o caso do modelo de Cook e Torrance [1982].

Os modelos utilizavam como referência somente o objeto, ou seja, o cálculo da iluminação dos mesmos era realizado considerando somente um objeto e uma fonte de iluminação. Esse tipo de modelo é conhecido como modelo local. Em contrapartida, no modelo global, se procura utilizar mais informações para calcular a iluminação de um objeto, como é o caso do *raytracing* [Whitted, 1979] e da radiosidade [Goral et al., 1984].

O *raytracing* modela fenômenos ópticos, considerando o caminho inverso que a luz se propaga em um dado ambiente. E a radiosidade modela fenômenos de transferência de energia luminosa considerando reflexões de objetos. A partir desses modelos, surgiram novos modelos que os utilizam como base.

O objetivo desse trabalho é o estudo sobre o modelo de iluminação global conhecido como radiosidade, pois o mesmo é um modelo importante em computação gráfica por ser um dos primeiros a permitir o acréscimo de características foto-realísticas de iluminação global a renderizações.

A radiosidade procura modelar a reflexão difusa entre superfícies, conseguindo aproximar a mistura de cores que ocorre entre as superfícies em virtude das interações de reflexões entre os próprios objetos. É um método numérico e é computacionamente caro. Tem sido efetivamente usando em computação gráfica para renderização offline de cenas realísticas ou como pré processamento para visualização interativa [Goral

et al., 1984; Cohen et al., 1988].

O principal problema da determinação de radiosidade se encontra na demanda por processamento para determinação de um fator geométrico chamado de fator de forma, que inclui a determinação da visibilidade de cada elemento para com os outros, e na resolução do sistema de equações que é realizado no método original.

Como uma possível solução a esse problema é possível utilizar a capacidade de processamento de GPUs[1] modernas para implementar a determinação de radiosidade, mas por sua arquitetura ser diferente da arquitetura de CPUs[2] comuns é necessário que se realize uma adaptação ao algoritmo, quando o mesmo é aplicável à GPU.

Nesse trabalho foram implementadas duas formas de se determinar a radiosidade que utilizam a GPU. No método original foi verificado a utilização da GPU para resolução do sistema linear e no refinamento progressivo foi proposta uma forma de melhorar a qualidade da função de visibilidade para cálculo do fator de forma.

1.1 Objetivo e motivação

O objetivo desse trabalho é realizar um estudo sobre os algoritmos de radiosidade original e por refinamento progressivo voltados a execução na GPU.

Existem duas motivações principais para a realização desse trabalho, a primeira é em relação ao método de determinação de radiosidade em si, pois o mesmo é um método clássico em computação gráfica que foi um dos primeiros a conseguir gerar imagens com características foto-realísticas. A segunda motivação está relacionada a utilização da GPU para acelerar o processamento de algoritmos, ou de parte de algoritmos, em virtude da alta capacidade de processamento que as mesmas proporcionam a um baixo custo, visto que a adaptação ou implementação nesses *hardwares* devem ser pensadas de acordo com sua arquitetura.

1.2 Contribuições

A principal contribuição desse trabalho está relacionada ao método de radiosidade progressiva em GPU, onde foi apresentada uma forma de determinação da função de visibilidade para que seja possível aproximar melhor do resultado de uma projeção esférica, eliminando alguns pontos de *aliasing* na cobertura da visibilidade. Dessa forma fazendo com que a determinação do fator de forma seja realizada com maior precisão.

[1] GPU - *Graphics Processing Unit*
[2] CPU - *Central Processing Unit*

1.3 Resumo dos capítulos

O capítulo Radiosidade aborda a definição de radiosidade, onde é apresentada a equação de radiosidade, os fatores de forma, a radiosidade por refinamento progressivo e como resolvê-la na forma de um sistema linear.

O capítulo Programação de GPU aborda alguns conceitos relacionados a programação de GPUs utilizando *shaders* que foram utilizados nesse trabalho.

O capítulo Implementação descreve as decisões de implementação dos algoritmos de radiosidade original e por refinamento progressivo.

O capítulo Resultados apresenta os experimentos realizados sobre os dois métodos que implementam a determinação de radiosidade, os resultados e a discussão sobre os mesmos.

Finalmente o capítulo Conclusão apresenta um fechamento do trabalho com algumas propostas para trabalhos futuros.

Capítulo 2

Radiosidade

Esse capítulo aborda o estado da arte sobre radiosidade, onde se descreve a equação e seus termos. O objetivo desse capítulo é fornecer informações que auxiliem no entendimento do modelo de radiosidade.

A radiosidade como modelo de iluminação em computação gráfica apareceu em uma época onde não existia um modelo que considerasse a reflexão entre objetos [Goral et al., 1984]. Ela é definida como a quantidade de energia luminosa que "sai" de uma superfície, sendo composta da energia que é refletida sobre a superfície e energia que é emitida pela mesma.

Em sua forma original, a radiosidade assume que os objetos são refletores difusos ideais, pois a maior parte da energia percebida em um ambiente é fruto da reflexão difusa entre as superfícies [Goral et al., 1984]. É possível determiná-la, pois as reflexões atingem uma situação de equilíbrio [Akenine-Moller e Haines, 2002].

A energia é modelada em forma de intensidade, sendo que a mesma pode ser representada como uma banda no espectro luminoso ou um conjunto finito de intervalos do comprimento de onda [Foley et al., 1996]. Ao utilizar o sistema de cor RGB[1], a intensidade é representada por três valores no intervalo fechado de zero(0) até um(1) para cada aproximação do comprimento de onda das cores primárias: vermelho, verde e azul. Cada valor representa a quantidade de energia mínima e a máxima suportada pelo dispositivo de saída (monitor) para as três bandas.

A reflexão difusa é modelada de acordo com a Figura 2.1, onde ao se especificar um ângulo de visão, uma intensidade é definida e a intensidade final percebida em todas as direções para esse ângulo é constante em todo o hemisfério de reflexão. Para saber a intensidade que deixa uma dada superfícies é necessário apenas multiplicar a intensidade em um dado ângulo por π [Goral et al., 1984].

As superfícies no modelo são definidas como polígonos convexos (quadriláteros),

[1]RGB - *Red, Green, Blue*

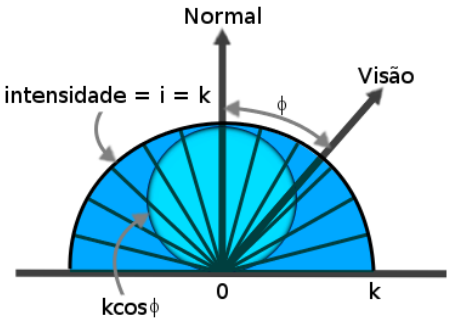

Figura 2.1. Reflexão difusa ideal sobre uma superfície. Imagem adaptada de Goral et al. [1984].

sendo chamados de *patches*, onde se assume que a radiosidade de um *patch* é constante em toda a sua área. Geralmente os objetos não são modelados em forma de *patches*, por isso em alguns casos, é necessário dividí-los em *patches*.

O resultado da radiosidade é independente da posição do observador, o que torna possível utilizar o cálculo da radiosidade de uma cena em várias visões da mesma. Esse resultado não necessariamente precisa ser aplicado de forma isolada, como ele fornece o coeficiente difuso de reflexão para qualquer ponto de observação da cena, o mesmo pode ser aplicado como o termo ambiente em outros modelos de iluminação existentes, como o modelo de Phong [1973], Cook e Torrance [1982] ou *raytracing* [Whitted, 1979].

Como a cena é composta por *patches*, então podem existir regiões onde a diferença entre radiosidades vizinhas é grande, que podem ser regiões de sombra. Quanto maior a quantidade de *patches*, mais suave serão essas transições, por isso é importante existir um algoritmo capaz de subdividir a cena.

2.1 Equação de radiosidade

Como dito anteriormente, a radiosidade é definida como a intensidade que sai de uma superfície, então a mesma pode ser definida como:

$$Radiosidade = LuzEmitida + LuzRefletida \tag{2.1}$$

A luz refletida é a intensidade vinda dos *patches* da cena. A equação para um *patch* i pode ser reescrita da seguinte forma:

$$B_i = E_i + \rho_i \sum_{1 \leq j \leq n} B_j F_{i \to j} \tag{2.2}$$

Onde B_i representa a radiosidade da superfície i, E_i é a energia emitida, ρ_i é o coeficiente de reflexão difusa, $B_j F_{i \to j}$ é a radiosidade do *patch* j ponderada pelo fator de forma de i para j, e ao considerar a influência de todos os n *patches* sobre a superfície de i se tem o somatório. O fator de forma é a fração visível de um *patch* para outro.

Existe uma equação dessa para cada um dos n *patches* da cena, dessa forma é possível escrevê-las como um sistema linear de n equações e n variáveis na forma $Ax = b$:

$$\begin{bmatrix} 1-\rho_1 F_{1-1} & -\rho_1 F_{1-2} & \cdots & -\rho_1 F_{1-n} \\ -\rho_2 F_{2-1} & 1-\rho_2 F_{2-2} & \cdots & -\rho_2 F_{2-n} \\ \vdots & \vdots & \ddots & \vdots \\ -\rho_n F_{n-1} & -\rho_n F_{n-2} & \cdots & 1-\rho_n F_{n-n} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix}$$

Note que a diagonal da matriz A utiliza o fator de forma de um *patch* para ele mesmo. Como assumimos que os *patches* são polígonos convexos, então o mesmo não é visível para si, fazendo com que o fator de forma seja zero. O sistema linear fica da seguinte forma:

$$\begin{bmatrix} 1 & -\rho_1 F_{1-2} & \cdots & -\rho_1 F_{1-n} \\ -\rho_2 F_{2-1} & 1 & \cdots & -\rho_2 F_{2-n} \\ \vdots & \vdots & \ddots & \vdots \\ -\rho_n F_{n-1} & -\rho_n F_{n-2} & \cdots & 1 \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix}$$

Os valores de B podem ser computados utilizando métodos numéricos. Nesse trabalho, essa forma de encontrar a radiosidade é referenciada como método ou algoritmo original.

2.2 Fator de forma

O fator de forma é a fração visível de um *patch* para outro, dessa forma, o somatório dos fatores de forma de um *patch* é uma unidade [Goral et al., 1984; Cohen e Greenberg, 1985]. O fator de forma de j para i é dado pela Equação 2.3, e sua geometria é mostrada na figura 2.2.

$$F_{dAi \to dAj} = \frac{\cos \phi_i \cos \phi_j}{\pi r^2} \tag{2.3}$$

Para que o fator de forma seja aplicado sobre toda a área de ambas as superfícies, Cohen e Greenberg [1985] apresenta a derivação do mesmo de modo a representá-lo

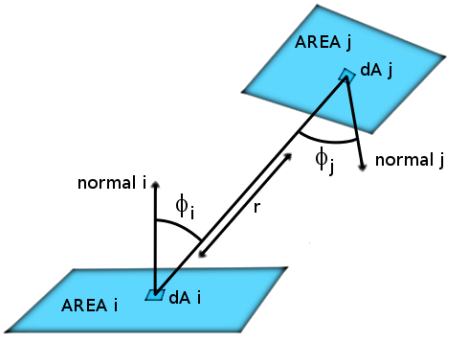

Figura 2.2. Geometria Fator de forma. Imagem adaptada de Cohen e Greenberg [1985].

como uma integral dupla sobre as áreas das superfícies envolvidas, como pode ser visto na Equação 2.4.

$$F_{Ai \to Aj} = \frac{1}{Ai} \int_{Ai} \int_{Aj} \frac{\cos \phi_i \cos \phi_j}{\pi r^2} \, V(i,j) \, dAi \, dAj \tag{2.4}$$

Essa equação apresenta uma função de visibilidade V(i,j) que serve apenas para dizer se um dado ponto na área j está visível ou não para um outro ponto na área i. Essa integral dupla pode ser usada para resolver a radiosidade para cenas de modo geral.

Goral et al. [1984] apresenta uma forma de resolver essa integral, mas essa abordagem limitava a utilização da radiosidade somente para cenas onde não existem oclusões, pois a função de visibilidade não era contabilizada de forma simples.

O fator de forma pode ser separado em duas partes: o cálculo analítico e a parte de visibilidade. A visibilidade pode ser determinada utilizando técnicas de remoção de superfícies ocultas como a renderização utilizando *z-buffer* ou através de algoritmos baseados em *raytracing*. A parte analítica apresenta um problema, pois considera que os *patches* possuem área infinitesimal. Como tentativa de solucionar esse problema, Wallace et al. [1989] propõe utilizar a aproximação por área de discos, que considera a área do *patch* como um disco orientado e utiliza a mesma para determinar o fator de forma. Ao utilizar essa aproximação por área de disco, torna-se possível o cálculo do fator de forma para *patches* mesmo que eles estejam próximos ou se suas áreas forem grandes. A equação apresentada por Coombe et al. [2004] e Coombe e Harris [2005] que utiliza a aproximação por disco de Wallace é apresentada da seguinte forma:

$$F_{i \to j} = \triangle j \frac{\cos \theta_i \cos \theta_j}{\pi r^2 + \triangle j} V(i,j)$$

Existe uma forma análoga ao fator de forma que foi desenvolvida por Nusselt e apresentada em Cohen e Greenberg [1985] como sendo uma projeção sobre um hemisfério na superfície, no qual essa é novamente projetada ortogonalmente sobre a base do hemisfério. O fator de forma é então a fração ou proporção ocupada na base do hemisfério, como pode ser visto na Figura 2.3.

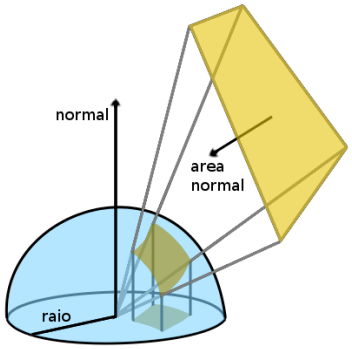

Figura 2.3. Projeção sobre o hemisfério. Imagem adaptada de Cohen e Greenberg [1985].

Uma característica interessante, ao pensar no fator de forma dessa maneira, é que diferentes superfícies podem possuir o mesmo fator de forma.

Uma forma simples de criar uma aproximação para a projeção sobre esse hemisfério é criar um conjunto de elementos discretos dividindo o hemisfério em várias regiões com um dado ângulo sólido. O *hemi-cube* [Cohen e Greenberg, 1985] foi a primeira proposta para utilizar o cálculo discreto do fator de forma.

O *hemi-cube* é a metade de um cubo que possui suas faces discretas e uma tabela de mapeamento de ocupação dos elementos(*pixels*) de modo a permitir o cálculo da integral do fator de forma, o mesmo pode ser visto na Figura 2.4.

Para determinar o fator de forma a partir dos valores preenchidos do *hemi-cube* basta somar os valores dessa tabela de mapeamento que a projeção de uma superfície ocupa como descrito pela Equação 2.5. Mais detalhes sobre o *hemi-cube* estão no capítulo Implementação desse trabalho.

$$F_{ij} = \sum_{q=1}^{R} \Delta F q \tag{2.5}$$

O termo R é a quantidade de *pixels* ocupada pela projeção da superfície e ΔFq é um valor da tabela de mapeamento para aproximar a projeção sobre o hemisfério.

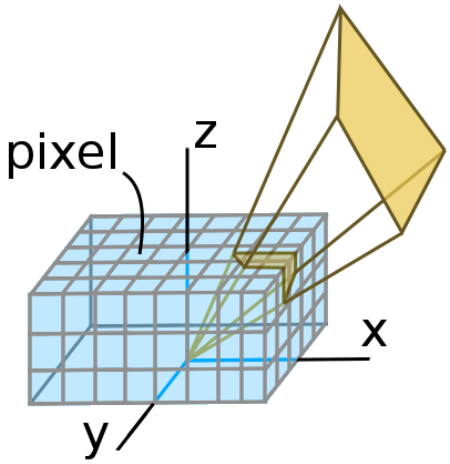

Figura 2.4. *Hemi-cube.* Imagem adaptada de Cohen e Greenberg [1985].

2.3 Resolução de sistemas lineares

As primeiras formas de calcular a radiosidade foram implementadas utilizando métodos para resolução de sistemas lineares, dos quais se destacam os métodos de Jacobi e Gauss-Seidel. Para que os mesmos possam ser aplicados, a matriz A deve possuir dominância pela diagonal, como a matriz de radiosidade apresenta essa dominância [Cohen e Greenberg, 1985], então é possível aplicá-los na resolução desse sistema linear. Esses métodos são conhecidos por convergirem rapidamente [Ruggiero e Lopes, 1988].

Esses métodos servem para resolver sistemas do tipo $Ax = b$, onde A é uma matriz $n \times n$, x um vetor de dimensão n e b um vetor de dimensão n. A partir dos valores de A e b, que são conhecidos, se procura estimar os valores de x.

Jacobi

O método iterativo de Jacobi realiza a estimativa do conjunto solução x baseado no valor da iteração passada $k-1$ como visto na equação 2.6.

$$x_i^{(k)} = \frac{b_i - \sum_{j=1, i \neq j}^{n} a_{ij} x_j^{(k-1)}}{a_{ii}} \tag{2.6}$$

Esse método pode ou não convergir para uma solução. Uma condição suficiente para garantir a convergência é a matriz A apresentar dominância pela sua diagonal, ou seja, o valor da diagonal ser maior que o somatório dos valores dos elementos da mesma linha. Esse critério suficiente para garantir a convergência do método pode ser verificado de acordo com a equação 2.7.

$$|a_{ii}| > \sum_{j=1, i \neq j}^{n} |a_{ij}| \tag{2.7}$$

Uma característica importante desse método é que uma solução é atingida independente do conjunto solução que foi estimado inicialmente.

A partir da equação 2.6 é possível montar um pseudo-algoritmo desse processo iterativo mostrado no Algoritmo 1.

Algoritmo 1 Iteração de Jacobi

```
//Considerando uma matriz 'a' de dimensão 'n'x'n'
// com vetores 'b' e 'x' com dimensão 'n'
// uma iteração do método de Jacobi
for i ← 1 to n do
    x[i]← b[i]
    for j ← 1 to n do
        if j ≠ i then
            x[i]← x[i]−a[i, j]∗x_old[j]// utiliza o vetor x da iteração passada
        end if
        x[i]← x[i]/a[i, i]
    end for
end for
```

O critério de parada do algoritmo pode ser a variação de um conjunto de estimativas x para o próximo conjunto de estimativas x, de acordo com a precisão numérica desejada.

Gauss-Seidel

O método iterativo de Gauss-Seidel é uma extensão de método de Jacobi, que contem apenas um conjunto de resultados, ou seja, os valores de x utilizados em uma iteração k podem ser utilizados na própria iteração. No caso do método de Jacobi, um conjunto x deve ser totalmente estimado antes do resultado ser utilizado na iteração seguinte. A equação 2.8 mostra a definição desse método.

$$x_i^{(k)} = \frac{b_i - \sum_{j=1, i \neq j}^{n} a_{ij} x_j^{(k)}}{a_{ii}} \tag{2.8}$$

O principal ponto positivo é que a solução "anda" na direção dos vetores da matriz A, o que possibilita a aproximação da solução real com menos iterações [Ruggiero e Lopes, 1988].

O ponto negativo é que o laço interno agora é dependente de um valor calculado anteriormente na mesma iteração k, fazendo o algoritmo apresentar uma serialização de processamento. O mesmo pode ser visto no Algoritmo 2.

Algoritmo 2 Iteração de Gauss-Seidel

```
//Considerando uma matriz 'a' de dimensão 'n'x'n'
// com vetores 'b' e 'x' com dimensão 'n'
// uma iteração do método de Jacobi
for i ← 1 to n do
    x[i]← b[i]
    for j ← 1 to n do
        if j ≠ i then
            x[i]← x[i]−a[i, j]∗x[j]// vetor x é utilizado diretamente
        end if
        x[i]← x[i]/a[i, i]
    end for
end for
```

2.4 Refinamento progressivo

O refinamento progressivo é uma forma de determinar a radiosidade proposta por Cohen et al. [1988], no qual foram feitas modificações no algoritmo original de modo a permitir que o resultado do processamento seja visualizado antes que a solução esteja completamente estimada.

O refinamento progressivo se baseou no trabalho de Bergman et al. [1986], que apresenta uma técnica de renderização simples e rápida, onde o realismo da cena é incrementado a cada quadro, até que uma mudança na visão da cena ocorra para que o processo recomece. O objetivo principal é oferecer a melhor imagem o quanto possível de acordo com a frequência de manipulação do usuário, onde existe um processo único que refina a cena, e a cada refinamento, a mesma é acrescida de mais realismo.

A radiosidade é representada como uma propriedade global dos objetos onde, mesmo que a visão mude, ainda é possível continuar o refinamento para se obter uma solução com maior realismo. A radiosidade ao contrário de uma técnica de renderização simples, precisa ser reiniciada somente se algum objeto da cena se mover ou se sua forma for alterada.

O método original determina todos os fatores de forma, monta a matriz de radiosidade para depois calcular uma solução. Por isso esse método impede a implementação interativa, já que o custo de determinar os fatores de forma e armazenar todos os dados de cenas complexas é alto.

No refinamento progressivo todos os *patches* são atualizados a cada determinação de fator de forma, o que não ocorre no método original. Além disso, apresenta uma convergência melhor, pois processsa os *patches* de acordo com sua contribuição para o ambiente.

No método original, um fator de forma fornece informação para preencher uma linha na matriz de radiosidade. Essa linha permite a estimativa somente de um valor de radiosidade, como pode ser visto na Figura 2.5 esquerda. Essa forma de preenchimento da matriz é chamada de "*gathering*", pois a radiosidade de apenas um *patch* é determinada a partir da influência dos outros *patches* da cena.

No refinamento progressivo, o fator de forma atualiza uma coluna da matriz de radiosidade, dessa forma, uma iteração permite atualizar todas as estimativas de radiosidade de uma vez. O método é chamado de "*shooting*" pois seria o equivalente a distribuir a energia entre os *patches* visíveis a partir de um fator de forma. O cálculo do fator de forma é dado diretamente, uma vez que existe uma relação de um *patch* para o outro que pode ser utilizada para calcular o fator de forma do caminho inverso. A Figura 2.5 direita mostra o exemplo do "*shooting*".

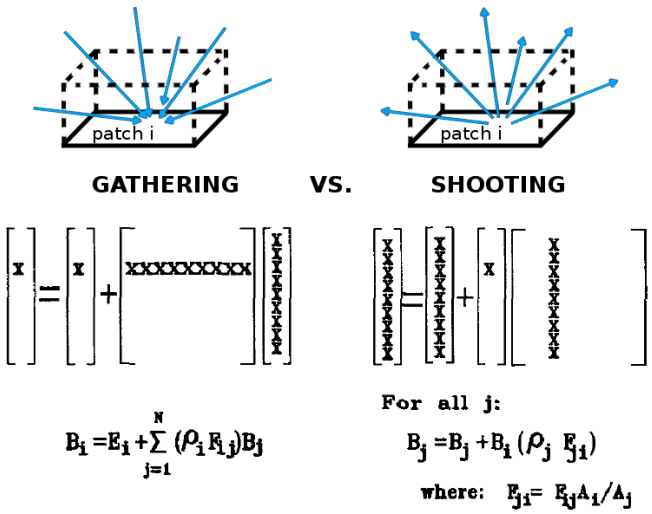

Figura 2.5. Esquerda *ghathering*; Direita *shooting*. Imagem adaptada de Cohen et al. [1988].

Para aplicar o "*shooting*" sobre a cena, é necessário armazenar para cada *patch* o coeficiente de reflexão, a radiosidade final e a radiosidade residual. Sempre que se lançar energia de um *patch*, deve se limpar a energia residual do mesmo.

O algoritmo base para a implementação da radiosidade por refinamento progressivo é semelhante ao utilizado nesse trabalho e será abordado no capítulo Implementação.

2.5 Subdivisão

Como visto anteriormente a subdivisão é importante para diminuir o gradiente entre regiões vizinhas.

No trabalho original foi utilizado um tipo de subdivisão uniforme, onde cada *patch* dá origem a novos *patches* de acordo com uma resolução predefinida. Cohen et al. [1986] apresenta um algoritmo baseado em dividir um dado *patch* mantendo a referência de hierarquia dessa divisão que pode ser uniforme ou adaptativa utilizando o critério de gradiente, onde a referência de cada novo item é dada como um elemento. Essa técnica, conhecida como *substructuring*, foi utilizada no refinamento progressivo desse trabalho.

O resultado do *substructuring* não é um *patch*, por isso não é necessário determinar o fator de forma a partir dos elementos gerados, tornando possível aumentar a definição de sombras sem aumentar a quantidade de elementos que lançam energia na cena.

Outra forma de calcular a subdivisão é apresentada por Hanrahan et al. [1991] como radiosidade hierárquica, onde utiliza o fator de forma juntamente com a área de cada superfície para determinar quanto deve subdividir um *patch* ou elemento.

2.6 Visualização

O resultado da radiosidade são apenas coeficientes associados aos *patches* da cena. Por esse motivo, ao utilizar somente esse resultado, a cena apresenta uma aparência discretizada, ou seja, parece ser composta por vários elementos com intensidade constante na área de cada um. Para tentar evitar esse tipo de resultado, desde o trabalho original, se utiliza algum tipo de função para interpolar os valores de intensidades sobre a superfície dos *patches*.

A interpolação pode ser bilinear [Cohen e Greenberg, 1985], de Gourand [Foley et al., 1996] ou utilizando uma textura sobre a geometria simples [Möller, 1996; Bastos et al., 1997]. Quando o número de geometrias for muito grande, a utilização de texturas é uma boa opção, visto que dessa forma a renderização pode se dar de forma interativa. Pois exige apenas que o *hardware* utilizado possua suporte a texturização simples.

2.7 Radiosidade e o hardware gráfico programável

Apesar de GPUs modernas possuirem um alto poder de processamento, poucos trabalhos utilizam-se da mesma para acelerar a determinação da radiosidade.

O trabalho de Carr et al. [2003] apresenta testes sobre o algoritmo de Jacobi para resolução de sistemas lineares implementado na GPU, onde o mesmo foi utilizado para

encontrar a solução da matriz de radiosidade e *subsurface scattering*. Os trabalhos de Coombe et al. [2004] e Coombe e Harris [2005] adaptam o algoritmo de refinamento progressivo com subdivisão adaptativa para a GPU e Wallner [2008] apresenta uma extensão desse trabalho adicionando ao cálculo de radiosidade uma função de reflectância arbitrária e resolvendo alguns problemas relacionados com a determinação de visibilidade que foi utilizada para calcular o fator de forma. Finalmente Lum et al. [2005] apresenta uma forma de implementar a radiosidade hierárquica utilizando texturas com MIPMAP para representar a hierarquia de subdivisões juntamente com o cálculo analítico do fator de forma.

Nesse trabalho se propôs uma forma para resolver o problema de visibilidade levantado em Coombe et al. [2004] e Coombe e Harris [2005] utilizando recursos do *hardware* gráfico, que ainda se diferencia da forma apontada por Wallner [2008] por não utilizar a primitiva gráfica do *hardware*(triângulo) como forma de mapeamento.

Capítulo 3

Programação de GPU

Nesse capítulo são apresentados alguns conceitos relacionados a programação de GPUs utilizando *shaders* que foram utilizados nesse trabalho. O conteúdo desse capítulo se baseia principalmente nos trabalhos de Owens et al. [2007] e Harris e Buck [2005].

As GPUs modernas apresentam uma capacidade de processamento massivo considerável, principalmente devido a sua especialização em operações paralelas relacionadas a computação gráfica tradicional. O desenvolvimento de novas série das principais fabricantes de GPUs vêem flexibilizando sua programação, com isso fazendo com que as APIs utilizadas se tornem também mais flexíveis. Como consequência disso, a GPU vem sendo utilizada para processamento de propósito geral.

Para utilizar os recursos desses *hardwares* eram necessárias APIs gráficas como OpenGL ou DirectX, mas com a flexibilização do *hardware*, as próprias fabricantes criaram APIs específicas como o CUDA[1], CTM[2] ou STREAM[3].

Ao utilizar a GPU para realizar computações não voltadas para a computação gráfica tradicional, emprega-se o termo GPGPU(*General Purpose computation on GPU*) que significa computação de propósito geral sobre a GPU [Harris e Buck, 2005; Owens et al., 2007].

A GPU pode apresentar maior eficiência para execução de instruções paralelas que as CPUs, pois a CPU geralmente é otimizada para execução de instruções sequenciais, onde grande parte dos transistores são utilizados para controle de desvios e execuções não ordenadas. A GPU por não possuir uma estrutura para tratamento de desvios condicionais elaborada, possibilita a utilização desses transistores diretamente para computações, fazendo com que a quantidade de operações aritméticas sejam processadas em maior quantidade com a mesma contagem de transistores [Owens et al., 2007].

De acordo com Owens et al. [2007], apesar da evolução tanto do *hardware* quanto

[1]http://www.nvidia.com/object/cuda_home.html - Acessado em Julho/2008
[2]http://ati.amd.com/companyinfo/researcher/Documents.html - Acessado em Julho/2008
[3]http://ati.amd.com/technology/streamcomputing/ - Acessado em Julho/2008

das linguagens, a GPU ainda é difícil de ser utilizada em computações não gráficas, pois para sua utilização eficiente é necessário se conhecer o modelo de computação baseado em termos gráficos.

3.1 APIs e Linguagens

As linguagens que foram surgindo para as GPUs eram voltadas exclusivamente para o *pipeline* de renderização. Por esse motivo foi dado o nome de *shader* ao programa (conjunto de instruções) executado sobre a GPU. Um programa de *shader* pode ser utilizado para diversos tipos de processamento. Hoje existem APIs de programação para séries de *hardwares* específicos como o CUDA, CTM ou STREAM, mas deve-se observar que a portabilidade das mesmas é limitada. Para apresentar uma solução com maior portabilidade entre os fabricantes, ainda é necessário se utilizar as linguagens de *shader*.

As linguagens e APIs existentes hoje criam uma camada de abstração sobre a programação de GPUs, pois os programas que as utilizam não controlam sincronismos internos da GPU nem locks de memória [Owens et al., 2007].

A linguagem escolhida para esse trabalho é uma linguagem virtual desenvolvida pela Nvidia chamada Cg [Mark et al., 2003] (*C-for graphics*). Que engloba as principais funcionalidades das principais outras linguagens existentes como HLSL, GLSL, FP, HLSL assembly, entre outras. Além de possuir uma API que converte automaticamente os programas que são escritos para qualquer uma dessas linguagens, o que possibilita utilizar Cg para programar sobre qualquer uma das principais APIs de renderização que existem. Ainda é possível escrever um conjunto de programas em um único arquivo através da API CGFX (*Effect for Cg*), que gerencia automaticamente a carga dos programas na GPU independente da API gráfica que está em execução.

3.2 GPGPU

O principal motivador para o desenvolvimento da programação para GPUs foi o sucesso do sistema de renderização offline da Pixar[4], pois era possível realizar operações mais flexíveis com vários *shaders*, o que permitiu a criação de imagens mais realísticas, principalmente no aspécto de iluminação.

Apesar de GPUs modernas possuírem vários processadores, não é qualquer tipo de computação que pode obter ganhos consideráveis ao ser portada para a GPU. A arquitetura de uma GPU é planejada para executar tarefas relacionadas a computação

[4]https://renderman.pixar.com/products/rispec/index.htm - Acessado em Julho/2008

gráfica tradicional, ou seja, as principais características que permitem um algoritmo alcançar um alto desempenho são o paralelismo de dados e independência da computação de cada elemento [Harris e Buck, 2005].

Para utilizar a programação de GPUs é essencial conhecer como funciona o *pipeline* gráfico, pois alguns conceitos de processamento paralelo são aplicados à GPU por meio desse. A Figura 3.1 mostra um *pipeline* semelhante ao apresentado em Owens et al. [2007], onde existem três estágios programáveis.

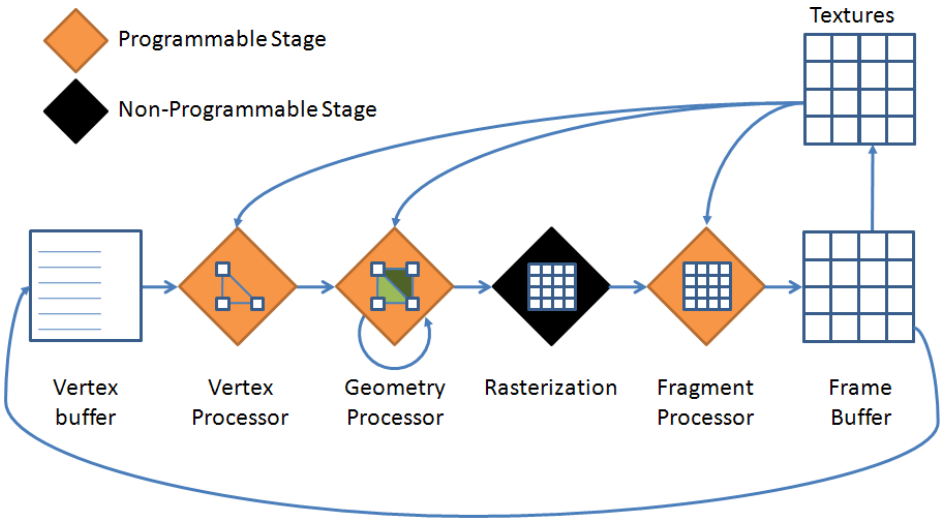

Figura 3.1. *Pipeline* de renderização.

O *pipeline* de renderização, como mostrado na Figura 3.1, é uma organização de operações que são a base para estruturação lógica de uma GPU para computação gráfica. O *hardware* não precisa seguir exatamente essa divisão de funções, como é o caso da arquitetura Tesla [Lindholm et al., 2008] criada pela Nvidia, pois ela não faz distinção das unidades de processamento interno (os *streams processors*). Nessa arquitetura, as unidades de processamento podem exercer a tarefa de qualquer estágio do *pipeline*, além de ainda estarem divididas em 16 conjuntos de threads que executam simultaneamente de forma independente.

Os estágios programáveis destacados na figura 3.1 são: Processamento de vértice, Processamento de geometria e Processamento de fragmentos.

O processamento de vértices geralmente realiza operações relacionadas a transformação dos vértices do modelo utilizado e a saída desse estágio é a posição que o mesmo terá no *framebuffer*.

O processamento de geometria recebe uma primitiva gráfica(linha, ponto, triângulo ou quadrado) já transformada e tem a capacidade de criar novas primitivas.

Todas as primitivas visíveis que vieram do processamento de vértices ou do gerador de geometria vão para o estágio fixo de rasterização, que gera os conjuntos de fragmen-

tos. Cada fragmento pode ser visto como um elemento que pode virar um *pixel* e sua posição de gravação no *framebuffer* é determinada aqui.

Finalmente, o processamento de fragmentos recebe um fragmento e deve determinar a saída do mesmo no *framebuffer*.

Kernel e *Streams*

Um *kernel* é um conjunto de instruções que são aplicadas a cada um dos elementos do *stream* de saída. Pode ser pensado como um laço em um programa convencional.

Geralmente o *kernel* opera sobre um conjunto de dados onde esses podem ser armazenados em forma de textura. Esse conjunto de dados é referenciado como *stream* ou textura, e pode ser utilizado apenas para leitura ou apenas para escrita. A Figura 3.2 mostra o exemplo da execução de um *kernel* sobre um *stream* que possui quatro elementos. É possível fazer uma analogia do laço do Algoritmo 3 com o *kernel* mostrado na Figura 3.2 .

As texturas são estruturas gráficas em forma retangular que possuem um conjunto de *texels*. Quando um *kernel* utiliza texturas para realizar computações, o conteúdo das mesmas passam a representar o conjunto de dados que será processado ou lido.

Algoritmo 3 Laço de exemplo de *kernel*

```
for i ← 0 to m − 1 do
    for j ← 0 to n − 1 do
        //KernelOperation
        ...
        result ← result + stream_in[i][j]
        ...
        stream_out[i][j]← result
    end for
end for
```

Existem dois tipos de operações de escrita que podem ser implementadas utilizando os *kernels*, são elas o agrupamento(*gather*) e o espalhamento(*scatter*).

No agrupamento, o *kernel* agrega informações de outras regiões de memória, e no espalhamento, o *kernel* distribui informações para outros *streams*.

Como visto no *pipeline* de renderização da Figura 3.1, o estágio de processamento de fragmentos consegue realizar somente operações do tipo agrupamento, pois a posição de gravação final de um elemento não é determinada por ele. Já os outros processadores podem alterar a posição final de um vértice ou conjunto de vértices(primitiva) alterando dessa forma a posição de escrita.

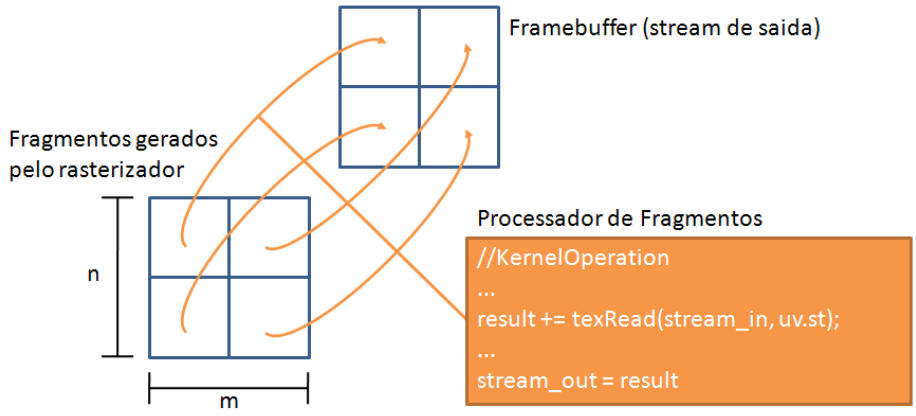

Figura 3.2. Exemplo de execução de um *kernel*.

Geralmente para utilizar a GPGPU é necessário identificar partes do algoritmo que possuem dados que possam ser processados em paralelo. A partir desses dados então um *kernel* é codificado para realizar a computação adequada.

Execução de *kernels*

A execução de um *kernel* é análoga ao processamento do laço que esse *kernel* define. Para isso, um ou mais *streams* de entrada devem estar disponíveis e um ou mais *streams* de saída devem ser configurados.

Geralmente o *kernel* atua sobre uma região retangular e são chamados através de uma função de desenho da API gráfica utilizada.

Alguns algoritmos não podem ser implementados com apenas uma única execução de um *kernel*. Quando essa situação ocorre é necessário criar uma cadeia de processamento e executar vários *kernels* em sequência. Exemplo: Um *stream* que foi a saída de um *kernel* A agora deve ser configurado como entrada de outro *kernel* B.

Reduções paralelas

As reduções paralelas são operações realizadas sobre um *stream* a fim de diminuir sua dimensão através da aplicação de operações de soma, média, máximo, mínimo, etc.

As reduções podem ser implementadas utilizando um par de *streams*, onde a cada passo de uma cadeia de processamento, o limite de gravação é reduzido por alguma fração. Geralmente uma redução é realizada em $O(\log n)$ passos. A Figura 3.3 mostra um exemplo de redução onde se tem um *stream* maior a que passa por uma redução de $\frac{1}{4}$ até restar somente um valor para ser utilizado em c.

Existem algumas funções de *hardware* disponíveis na API gráfica que podem ser utilizadas para otimizar as reduções, como a criação automática de mipmaps para obter

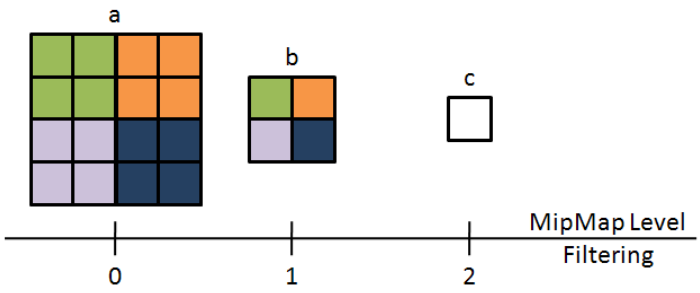

Figura 3.3. Exemplo de redução em relação a pirâmide de mipmap: a)*stream* original; b)*stream* com $\frac{1}{4}$ da dimensão de a; c)*stream* com $\frac{1}{4}$ da dimensão de b.

a média ou o *z-buffer* para determinar o valor máximo ou mínimo de um conjunto de valores.

Desempenho

Apesar da GPU possuir uma arquitetura com grande capacidade de transferência e processamento aritmético, existem dois principais pontos que fazem uma aplicação GPGPU ter uma queda de desempenho, que são relacionados a transferência CPU-GPU/GPU-CPU e desvios condicionais.

Se um programa utilizar a GPU como um co-processador massivo, então é necessário carregar os dados na GPU, executar o *kernel* e ler os valores de volta. Para se obter um ganho substancial nesse tipo de aplicação, deve-se utilizar uma massa de dados grande o suficiente para que o ganho sobreponha o *overhead* da transferência que é necessária.

Para aplicações gráficas onde a GPU processa elementos e utiliza os mesmos na própria GPU, não existe a transferência GPU-CPU. Em algumas situações onde existem alguns pontos de sincronização no algoritmo que requerem uma transferência de poucos dados para a CPU, para verificação de cálculos ou obtenção de resultados, é importante realizar o mínimo de transferências o quanto possível.

Outro ponto que pode fazer uma aplicação perder desempenho é a utilização de desvios condicionais dinâmicos como if, for ou while. A execução dos desvios ocorre em paralelo dentro de grupos de processadores, sendo que os fragmentos que não sofreram o desvio ficam ociosos até o termino de uma execução. Geralmente, quando o teste é muito simples e o conteúdo que será executado é pequeno, utilizar uma multiplicação ao invés do teste condicional pode fazer o programa apresentar um ganho de desempenho.

Capítulo 4

Implementação

Nesse capítulo são descritas as decisões de implementação dos algoritmos de radiosidade original e por refinamento progressivo.

Foram implementados dois algoritmos de radiosidade para serem executados na GPU, o original que constrói a matriz de sistemas lineares e o algoritmo apresentado por Coombe et al. [2004] e Coombe e Harris [2005] que implementa o algoritmo de radiosidade por refinamento progressivo.

Nas implementações não foram considerados algoritmos alternativos para subdivisões. Foi utilizado somente o algoritmo de subdivisão uniforme sobre a malha de entrada.

O método original utiliza a determinação dos fatores de forma utilizando o *hemicube* e o método de refinamento progressivo utiliza a projeção esférica juntamente com o cálculo de fator de forma baseado em aproximação por disco [Wallace et al., 1989; Wallner, 2008].

O fluxo de execução da aplicação se baseia no fluxo apresentado por Goral et al. [1984], onde existe uma etapa de pre-processamento em que a radiosidade é determinada, e a etapa de renderização, que utiliza o valor da radiosidade para apresentar o resultado de forma interativa.

A Figura 4.1 mostra um fluxo que começa com a aplicação inserindo a geometria com suas propriedades (reflexão e emissão), passa pelo algoritmo de subdivisão uniforme, vai para o cálculo da radiosidade e renderiza a cena. Em todas as partes que envolvem a utilização da geometria, foi utilizada uma *display list* [Opengl et al., 2005] que facilita a renderização, dessa forma é necessário chamar apenas uma função da API do OpenGL.

As partes que se diferenciam nos algoritmos implementados do método original e refinamento progressivo são a determinação da radiosidade, a geração das texturas finais e a renderização que utiliza como base texturas mapeadas de formas diferentes.

Buffer de identificação de objetos

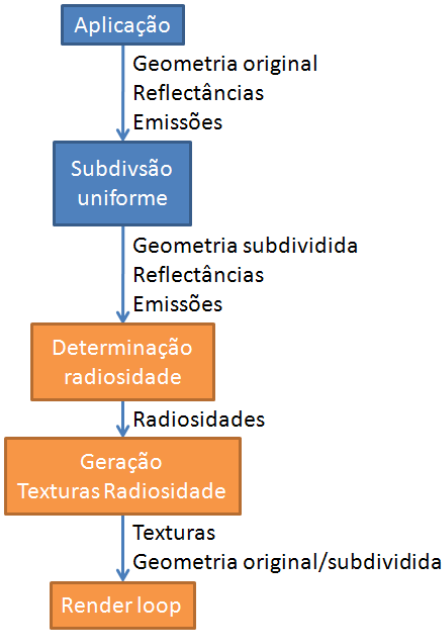

Figura 4.1. Fluxo lógico para computar e exibir o resultado da radiosidade.

Os dois algoritmos implementados na GPU utilizam um *buffer* de identificação de objetos para auxilar no cálculo dos fatores de forma. Como exemplo: no método do *hemi-cube*, é necessário projetar a cena e utilizar, para cada *patch* processado, um identificador único que geralmente é atribuído a um inteiro, cor ou ponto-flutuante.

A identificação serve para determinar a proporção visível de um objeto a um dado ponto de vista.

Subdivisão uniforme

A subdivisão utilizada em ambos os algoritmos é realizada de forma estática. A função de subdivisão recebe como parâmetro um polígono bem como suas propriedades (reflectância e emissão) e um valor inteiro para controlar o nível de subdivisão.

O polígono de origem é dividido em partes iguais em ambas as orientações de acordo com um parâmetro n. Esse processo é executado até que se atinja um dado limite especificado. A Figura 4.2 mostra um exemplo de subdivisões de um polígono em vários polígonos de acordo com um parâmetro. O algoritmo de subdivisão é mostrado no Algoritmo 4.

4.1 Radiosidade original

Para determinar a radiosidade de acordo com o método original é necessário efetivamente montar a matriz com os fatores de forma e reflectâncias na forma de equações

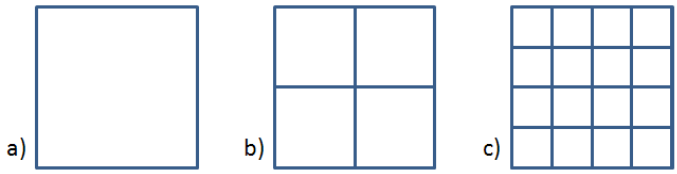

Figura 4.2. Exemplo de subdivisão uniforme para diferentes parâmetros: a)subdivisão=1; b)subdivisão=2; c)subdivisão=4.

Algoritmo 4 Subdivisão de geometria

```
function SUBDIVIDE(GeometryManager result,StructToAdd toAdd)
    int x ← toAdd.subdivision
    int y ← toAdd.subdivision
    vec3 ad ← toAdd.d - toAdd.a //vetor vertical esquerdo do patch
    vec3 bc ← toAdd.c - toAdd.b //vetor vertical direito do patch
    vec3 a, b, c, d
    for j ← 0 to y − 1 do
        float jFirst = j/y
        float jSecond = (j+1)/y
// Definição dos pontos de uma linha subdividida
        a = toAdd.a + ad * jFirst
        d = toAdd.a + ad * jSecond
        b = toAdd.b + bc * jFirst
        c = toAdd.b + bc * jSecond
        vec3 ab = b - a // Vetor horizontal superior da linha
        vec3 dc = c - d // Vetor horizontal inferior da linha
        for i ← 0 to x − 1 do
            float iFirst = i/y
            float iSecond = (i+1)/y
// Definição dos pontos de uma coluna subdividida
            vec3 refa = a + ab * iFirst
            vec3 refb = a + ab * iSecond
            vec3 refc = d + dc * iSecond
            vec3 refd = d + dc * iFirst
// Emitindo um novo patch subdividido
            ADD_QUAD(refa,refb,refc,refd, toAdd.emission,toAdd.reflectancy)
        end for
    end for
end function
```

simultâneas. A partir da mesma, é aplicado um método numérico para encontrar a solução aproximada do sistemas de equações.

A textura de equações simultâneas que é montada possui 3 componentes de cor e 1 componente que é o fator de forma. Já as texturas de emissão, reflectância e radiosidade possuem apenas 3 componentes válidos. Isso faz com que ao se calcular a

solução das equações se obtenha as soluções para os 3 componentes de cor (RGB).

O algoritmo utiliza a estrutura do *hemi-cube* para determinar os fatores de forma. O mesmo é aplicado sobre os *patches* subdivididos a fim de montar a matriz de equações.

4.1.1 Hemi-cube

O *hemi-cube* é a primeira estrutura que apareceu para otimizar o cálculo de fatores de forma utilizando o *hardware* gráfico. São utilizados os centros dos *patches* como o centro de projeção, são definidos os valores de orientações a partir da normal da superfície e um dos vetores é escolhido para ser a orientação "cima" do *hemi-cube*.

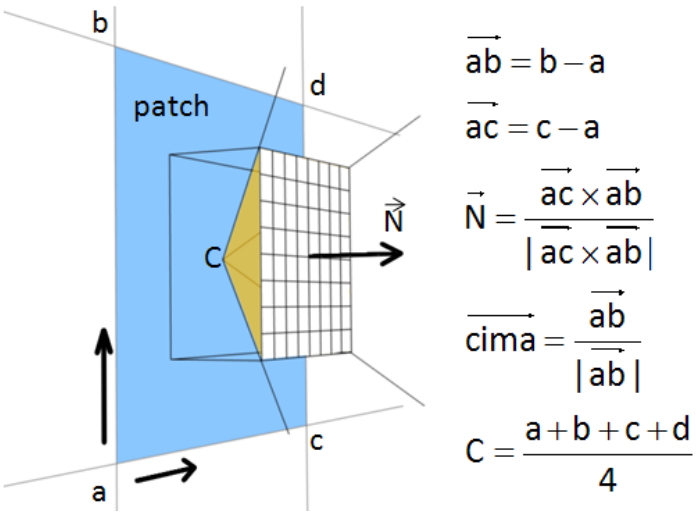

Figura 4.3. Exemplo de utilização dos valores dos pontos a, b, c e d para construção da base para orientação do plano da frente do *hemi-cube*.

A Figura 4.3 mostra o exemplo de um *patch* com os pontos a, b, c e d e quais equações foram utilizadas para determinar a normal, vetor cima e o centro que são utilizados para montar a matriz de transformação e projeção de referência.

Ao todo são cinco transformações e projeções, onde a transformação principal é obtida de acordo com a matriz inversa construída a partir de informações do polígono e as projeções são delimitadores de um tronco de pirâmide(*frustum*) com campo de visão de 45^o. No final, cada face do *hemi-cube* possui sua transformação que é utilizada para projetar os elementos nas respectivas faces.

A transformação principal Tp é montada a partir da normal da superfície e do vetor que indica a orientação *cima*. Antes de construir a matriz, são definidos os vetores que irão montar o quadro de referência de acordo com as equações:

$$\vec{z} = -\vec{N}$$

$$\vec{y}=\vec{cima}$$

$$\vec{x}=\vec{z} \times \vec{y}$$

A partir dos vetores $\vec{x}$, $\vec{y}$, $\vec{z}$ e do ponto c, é criada a matriz inversa de transformação da face frontal do *hemi-cube*.

$$Tp^{-1} = \begin{bmatrix} X\ \vec{x} & Y\ \vec{x} & Z\ \vec{x} & -Xc \\ X\ \vec{y} & Y\ \vec{y} & Z\ \vec{y} & -Yc \\ X\ \vec{z} & Y\ \vec{z} & Z\ \vec{z} & -Zc \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

As outras transformações são derivadas dessa principal, com alteração somente da rotação em torno dos eixos x e y.

$$xRotate(\phi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$yRotate(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{front} = Tp^{-1}$$

$$T_{right} = yRotate(90^o) * Tp^{-1}$$

$$T_{left} = yRotate(-90^o) * Tp^{-1}$$

$$T_{up} = xRotate(-90^o) * Tp^{-1}$$

$$T_{down} = xRotate(90^o) * Tp^{-1}$$

Para as projeções, foram utilizadas matrizes construídas da seguinte forma:

$$frustum(left, right, bottom, top, near, far) =$$

$$\begin{bmatrix} \frac{2\ near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2\ near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2\ far\ near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Onde cada face possui uma configuração distinta de projeção.

$$P_{front} = frustum(-0.001, 0.001, -0.001, 0.001, 0.001, 100)$$

$$P_{right} = frustum(-0.001, 0, -0.001, 0.001, 0.001, 100)$$

$$P_{left} = frustum(0, 0.001, -0.001, 0.001, 0.001, 100)$$

$$P_{up} = frustum(-0.001, 0.001, -0.001, 0, 0.001, 100)$$

$$P_{down} = frustum(-0.001, 0.001, 0, 0.001, 0.001, 100)$$

Ao especificar as projeções e transformações de cada uma das faces, é possível determinar a matriz final das faces realizando a multiplicação $T_{\{face\}} * P_{\{face\}}$. A Figura 4.4 exemplifica a projeção no espaço de cada uma das faces.

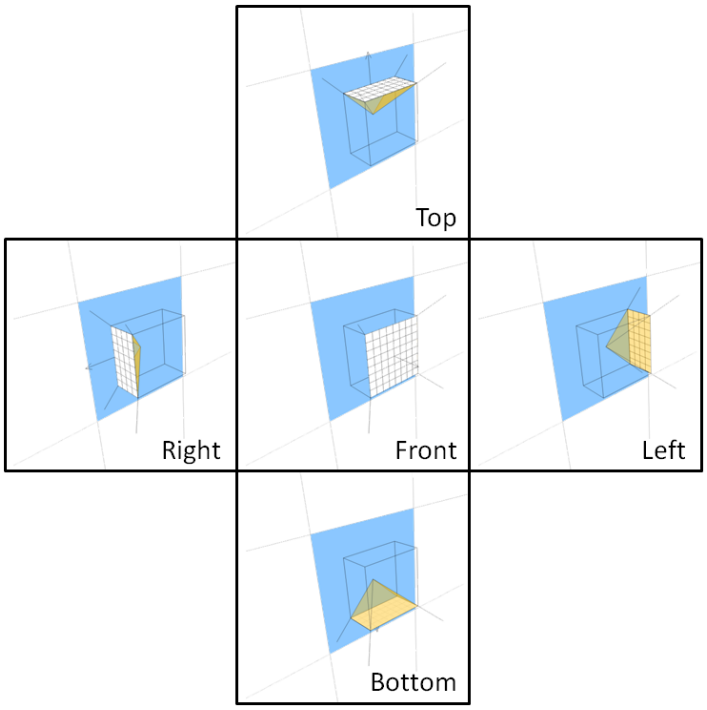

Figura 4.4. Transformações finais de cada face do *hemi-cube*.

Renderização

Para preencher todo o conteúdo do *hemi-cube* são necessárias cinco renderizações e armazenar a visão da cena de forma discreta em uma ou mais texturas. Se fossem utilizadas cinco texturas, uma para cada face, então elas possuiriam as dimensões $n \times n$ para a face da frente e $n \times \frac{n}{2}$ ou $\frac{n}{2} \times n$ para as faces das laterais. Nesse trabalho foram

utilizadas três texturas de dimensão $n \times n$ onde duas delas possuem um conjunto de duas projeções, como exemplificado na Figura 4.5.

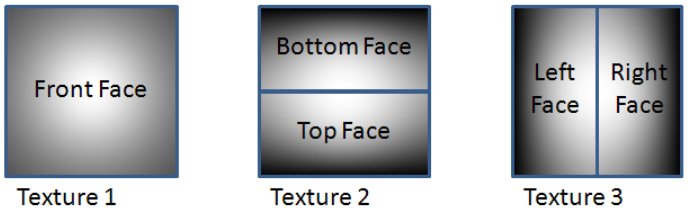

Figura 4.5. Distribuição das faces do *hemi-cube* em três texturas e intensidade dos fatores de forma delta (tom de cinza).

Para eliminar as distorções devido ao formato não hemisférico do *hemi-cube*, foram utilizados os fatores de forma delta que consideram o fator de angulação para superfícies difusas.

A fórmula para a face da frente é

$$\triangle_{FrontFormFactor} = \frac{1}{\pi|(x - n/2, y - n/2, 1)|^2}$$

E para as laterais é

$$\triangle_{SideFormFactor} = \frac{z}{\pi|(1, y, z)|^2}$$

Um exemplo desde mapeamento sobre as texturas criadas pode ser observado através dos tons de cinza na Figura 4.5.

Para se calcular o fator de forma de um elemento que foi projetado, basta somar todos fatores de forma delta que o elemento ocupa após a projeção.

Como o fator de forma é uma função que considera a proporção projetada sobre o hemisfério, então se toda a área do *hemi-cube* for ocupada, o somatório de todos os fatores de forma delta devem resultar em uma unidade. Por essa razão, após calcular o somatório, é necessário dividir esse valor pelo somatório total de toda a área do *hemi-cube*, considerando todos os *pixels* de acordo com a dimensão das texturas.

Matriz de radiosidade

Assim que um *hemi-cube* recebe a informação de visibilidade referente a um dado *patch* i, é possível realizar o preenchimento de uma linha i inteira da matriz de reflectância a partir dos elementos projetados.

É importante lembrar que o fator de forma totalmente preenchido deve resultar em um, ou seja, ele precisa ser normalizado de acordo com a resolução das texturas

utilizadas. Quanto maior for essa resolução, maior será o valor de normalização e consequentemente mais precisão será necessária para realizar os cálculos.

Para montar a matriz de radiosidade por completo é necessário realizar n projeções, onde n é o número de *patches* que a cena possui.

4.1.2 Resolução do sistema linear

Após montar a matriz de equações simultâneas é necessário obter uma solução para a mesma. Foram implementados os algoritmos de Gaus-seidel e Jacobi em C++ e também foi adaptado o algoritmo de Jacobi para ser executado na GPU.

O algoritmo de Jacobi foi escolhido para ser adaptado à GPU, porque o mesmo dentre os algoritmos de resolução de sistemas lineares vistos, é o que apresenta um potencial de paralelização maior.

A estratégia de implementação em GPU foi baseada no vetor de resultado "x" que deve ser calculado a cada iteração. Dessa forma foram criadas duas etapas de GPGPU e uma etapa de CPU:

- GPGPU: O vetor "x" é a saída de renderização e os vetores "a", "x_old" e "b" são as entradas;
- GPGPU: O valor da maior diferença "maxDifference" é a saída de renderização e os vetores "x_old" e "x" são utilizados como entradas. Como os vetores de entrada possuem uma dimensão maior que a saída, então foi implementada uma redução paralela;
- CPU: realiza uma leitura do valor de "maxDifference" para saber se a variação de uma solução para a outra é menor que um Epsilon. Se for, então não precisa mais iterar sobre os valores.

Note que o único local onde é realizada uma leitura de dados GPU-CPU é na terceira etapa e mesmo assim é realizada apenas a leitura de um elemento. É importante não efetuar leituras dessa natureza excessivamente, pois o gargalo de aplicações GPGPU geralmente se concentram na transmissão GPU-CPU ou CPU-GPU.

O algoritmo paralelo funciona de forma idêntica ao implementado para executar em CPU, o mesmo pode ser visto abaixo na figura 4.6:

Para verificar essa implementação, foi realizada uma comparação entre os resultados obtidos com o resultado da versão que funciona somente em CPU que foi implementada anteriormente.

Depois do algoritmo estar funcionando, foi observado que existe um teste condicional dentro do loop principal. Branch dinâmico em uma GPU geralmente possui um

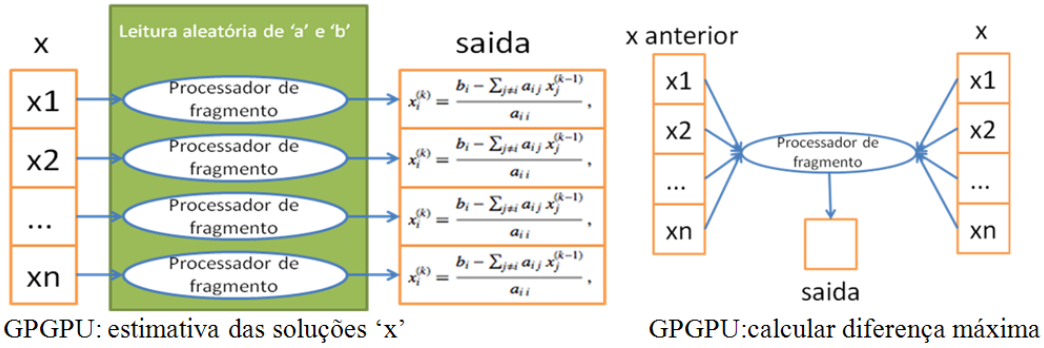

Figura 4.6. Diagrama dos passos que ocorrem na GPU do método de Jacobi.

custo alto, o que se verificou ao trocar esse teste por uma operação de multiplicação que filtra o resultado, da mesma forma que o "if" atuaria sobre o resultado da estimativa de "x".

O novo código apresentou uma diferença de 20ms no tempo de execução somente retirando o "if" do programa que roda na GPU. O novo código é mostrado no Algoritmo 5.

Algoritmo 5 Jacobi na GPU

```
function FS(in Vert2Frag IN, out FragOut OUT)
    float i ← IN.uv.s
    float solution ← readVetor(b, i)
    for jInt ← 0 to y − 1 do
        float j ← (float)jInt + 0.5
        //XORfilter faz a funcionalidade do if
        float XOR_filter ← step(0.0002, abs(j - i))
        solution ← solution - XOR_filter * ARRAY(a, i, j) * readVetor(x, j)
    end for
    OUT.out1 ← solution / ARRAY(a, i, i)
end function
```

A matriz de reflectâncias apresenta, além da necessidade de armazenamento $O(n^2)$, uma característica negativa com relação a precisão dos valores internos. Uma vez que os elementos da diagonal são sempre 1 (por não serem visiveis para sí mesmos), então o somatório de todos os outros elementos de uma linha deve ser menor do que 1. Portanto, quanto maior o número de *patches* no modelo, menor serão os valores que preenchem essa matriz.

Ao utilizar a GPU para processar sistemas dessa natureza, deve se observar a precisão que a mesma provê. Pois no caso da arquitetura Tesla, a GPU opera sobre valores de ponto flutuante de 32 bits em algumas operações internas com precisões diferentes, fazendo a parte significativa variar de 20 a 24 bits. Para modelos muito grandes, talvez

nem mesmo uma CPU comum possa realizar o processamento utilizando um ponto flutuante de 32 bits.

4.1.3 Visualização

Ao final do processo, existirá apenas um valor de radiosidade por *patch* subdividido. A visualização pode ser feita utilizando uma interpolação bilinear das radiosidades de *patches* vizinhos, e ao utilizar texturas, é possível exibí-las com um filtro bilinear que o próprio *hardware* gráfico implementa.

Como a cena foi subdividida antes do processamento, então é necessário utilizar a referência de um dado *patch* com o polígono que o originou. Permitindo que cada polígono na geometria original possua uma textura de radiosidade ou radiosidades de acordo com a resolução da subdivisão utilizada.

É interessante renderizar a cena dessa forma, pois permite utilizar o resultado da radiosidade com poucos polígonos em *hardwares* não tão modernos.

4.2 Radiosidade por refinamento progressivo

O processo de determinar a radiosidade por refinamento progressivo dessa dissertação utiliza o algoritmo adaptado à GPU apresentado por Coombe et al. [2004] e Coombe e Harris [2005]. Cada *patch* da cena possui uma textura de radiosidade e uma textura de energia residual.

De acordo com a radiosidade por refinamento progressivo, é necessário atualizar todos os elementos visíveis de um ponto de vista, mas como o processo de espalhamento na GPU é pouco eficiente, então Coombe et al. [2004] Coombe e Harris [2005] adaptam esse problema para poder ser processado com operações de agrupamento.

As texturas de radiosidade/residual são atualizadas uma a uma de acordo com a sua visibilidade. Essa abordagem facilitou utilizar a técnica de *substructuring* apenas aumentando a resolução das texturas, pois ao aumentar esse número, aumenta-se o número de *patches* que recebem energia. Um *kernel* é aplicado sobre as texturas de todos os objetos visíveis que estão sendo processados.

Como no refinamento progressivo a ordem de processamento dos *patches* deve ser realizada do elemento de maior potência para o de menor potência até atingir um certo limite, foi utilizada a geração de mipmap automática do *hardware* para calcular a energia total residual de um *patch*. Além disso, o algoritmo de *z-buffer* foi utilizado para selecionar o próximo *patch* a ser processado.

O processo para determinar a radiosidade por refinamento progressivo na GPU é descrito no Algoritmo 6.

Algoritmo 6 Radiosidade

```
Inicializar radiosidades com valor nulo
Inicializar residual com valores de emissão
i ← patch com maior potência
repeat
    Renderizar cena a partir da visão i
    for todos patches visíveis a partir de i do
        Calcular Fator de Forma
        Somar resultado em radiosidade e residual
        Gerar mipmap residual
    end for
    Limpar residual i
    i ← Próximo elemento de maior potência
until potencia i > limiar
```

4.2.1 Cálculo do fator de forma

Para calcular o fator de forma, foi utilizada a determinação de visibilidade separada do cálculo analítico. A projeção esférica determina a visibilidade, pois a mesma não necessita de cinco renderizações como o *hemi-cube*. O cálculo analítico é o mesmo apresentado por Coombe et al. [2004] e Coombe e Harris [2005], onde utiliza a aproximação por área de disco.

Como a determinação de visibilidade é feita separadamente através da utilização de uma projeção esférica, na equação abaixo, Tp^{-1} é a matriz de visão configurada da mesma forma que a mostrada no *hemi-cube* como transformação principal, P é o vértice da geometria, *near* é o plano que fica mais próximo ao centro da projeção e far é o limite de alcance máximo da projeção.

$$pos = \begin{bmatrix} Xpos \\ Ypos \\ Zpos \\ Wpos \end{bmatrix} = Tp^{-1} * P$$

$$ProjectionPosition = \begin{bmatrix} Xpos/|pos| \\ Ypos/|pos| \\ \frac{-2Zpos-near-far}{far-near} \\ 1 \end{bmatrix}$$

A partir dos pontos projetados, a rasterização escreve todos os *pixels* visíveis no *framebuffer*. Da mesma forma que no *hemi-cube*, se cria um *buffer* com identificadores para os objetos visíveis.

Essa projeção era utilizada juntamente a um *raytracer* e ao adaptá-la ao *hardware* gráfico, que realiza somente rasterizações lineares, fez com que aparessessem deformações de áreas que deveriam ser curvas, que agora passam a ser lineares. A figura 4.7 mostra uma rasterização linear e a curva que deveria ser "rasterizada".

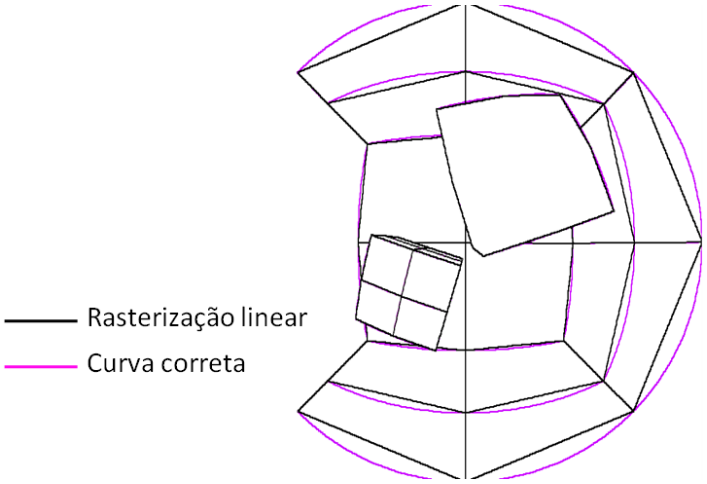

Figura 4.7. Exemplo de projeção esferica com rasterização linear e curva projetada de forma ideal.

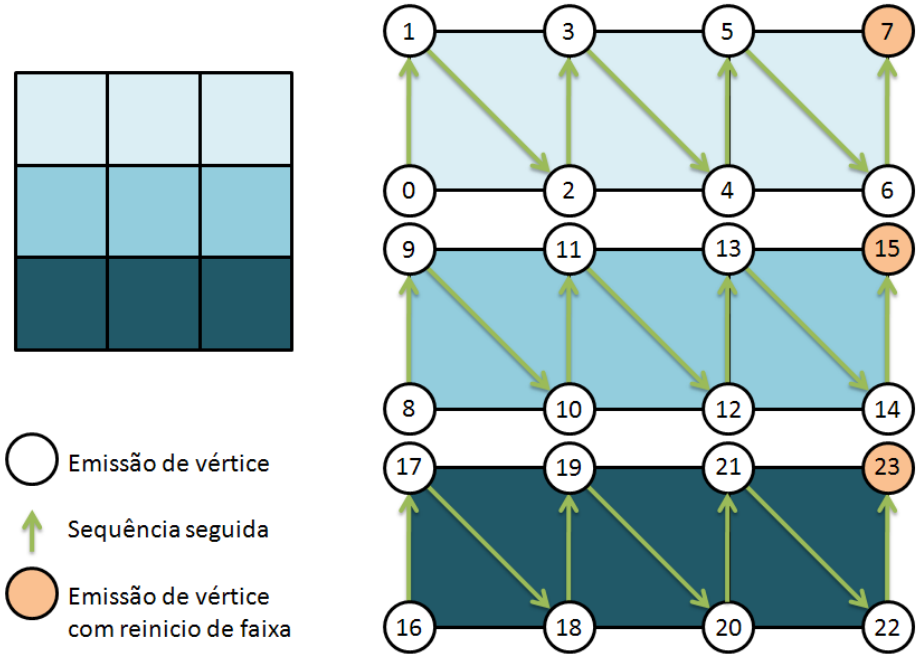

Figura 4.8. Exemplo da subdivisão utilizando emissão de vértice em forma de faixas.

Para mitigar o problema foi utilizada a capacidade de geração de geometria das GPUs modernas para subdividir os *patches* em tempo de renderização a fim de deixar

a renderização mais próxima da projeção esférica desejada. O Algoritmo 7 foi utilizado para subdividir a geometria de entrada. Como a emissão de primitivas utilizada é um triângulo, a cada três emissões de vértice um triângulo deveria ser rasterizado, mas foi utilizada a configuração de emissão em faixa [Opengl et al., 2005] e a cada nova "linha" do conjunto de faixas é dado um comando pra marcar o início da triangulação. A Figura 4.8 mostra um exemplo da subdivisão voltada para criação de geometrias com o padrão de faixas.

Algoritmo 7 Subdivisão para o processador de geometria

```
float4 a, b, c, d //vertices de um patch
float4 nc, nd
float4 oa, ob, oc, od
oa ← a
oc ← c
for j ← 1 to subdivision do
    float lerp_Factor_J ← j/subdivision
    ob ← a*lerp_Factor_J + b*(1-lerp_Factor_J)
    od ← c*lerp_Factor_J + d*(1-lerp_Factor_J)
    Emmit Vertice oa
    Emmit Vertice ob
    for i ← 1 to subdivision do
        float lerp_Factor ← i/subdivision
        nc ← oa*lerp_Factor + oc*(1-lerp_Factor)
        nd ← ob*lerp_Factor + od*(1-lerp_Factor)
        Emmit vertice nc
        Emmit vertice nd
    end for
    Reset Triangulation
    oa ← ob
    oc ← od
end for
```

Ao utilizar a capacidade de geração de geometria do *hardware* é possível ter uma boa aproximação para a projeção, além de evitar o gasto de memória para subdividir e armazenar a geometria subdividida.

Após ter a projeção dos elementos visíveis de um *patch*, todas as texturas dos *patches* visíveis são processadas de acordo com um *kernel* que calcula o fator de forma por aproximação de disco e faz uma busca por elementos projetados no *buffer* de identificadores pra determinar a visibilidade. Mesmo aproximando a projeção esférica com a subdivisão da malha, ainda podem haver problemas de amostragem na busca por visibilidade,pois os elementos da textura são discretos. Como tentativa de diminuir esse problema são realizadas leituras de vizinhos de uma posição já projetada para verificar

se o ponto em questão está ou não visível. O algoritmo de cálculo dos fatores de forma com busca por vizinhos é mostrado no Algoritmo 8.

4.2.2 Geração de texturas

No refinamento progressivo não foram utilizados os *patches* da geometria original, pois as texturas com *substructuring* fariam facilmente o tamanho final da textura não poder ser armazenado como uma única textura.

Foram criadas texturas com 9 vezes o tamanho original do *substructuring*, onde foi mapeado o elemento central da mesma. Isso permite que a aplicação do filtro trilinear sobre a pirâmide de resoluções gerada pelo mipmap, podendo ainda controlar o nível de filtragem pela seleção do mipmap. A Figura 4.9 mostra um exemplo da relação de mapeamento para uma textura com o tamanho aumentado.

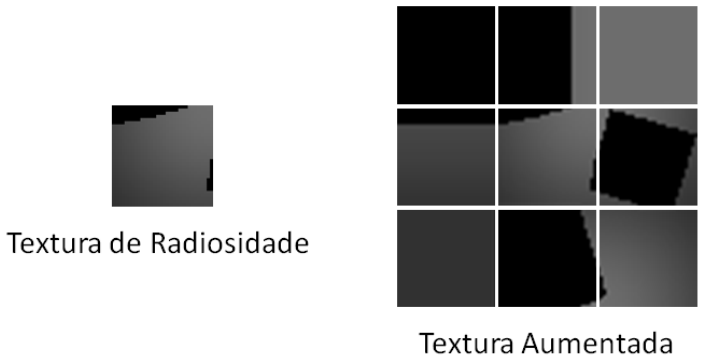

Figura 4.9. Textura com tamanho original e textura aumentada para permitir a aplicação de filtros com acesso a texturas vizinhas.

Para a renderização, é necessário criar essas texturas e associá-las ao modelo.

Algoritmo 8 Visibilidade

```
function Visible(float3 ProjPos, float4 RecvID, sampler2D HemiItemBuffer):bool
    float3 proj ← normalize(ProjPos)//Projetar o texel no hemisfério
    proj.xy ← proj.xy*0.5+0.5//Escala para acessar a textura [0..1]
    float4 xtex
//Busca por 3 vizinhos
    const int iterator ← 3
    bool result ← false
    for j ← -iterator to iterator do
        for i ← -iterator to iterator do
            xtex ← tex2D(HemiItemBuffer, proj.xy + float2(i,j) * SphereUnitScale)
            result ← result or all(xtex == RecvID)
        end for
    end for
    return result
end function

//Função analitica do fator de forma
function FormFactorEnergy(float3 RecvPos, float3 ShootPos, float3 RecvNormal, float3 ShootNormal, float3 ShootEnergy, float ShootDArea, float3 RecvColor):float3
//Vetor normalizado em direção ao patch receptor de energia
    float3 r ← ShootPos - RecvPos
    float distance2 ← dot(r,r)
    r ← normalize(r)
//Angulos entre os patches i(receptor) e j(lançador)
    float cosi ← dot(RecvNormal,r)
    float cosj ← -dot(ShootNormal,r)
//Aproximação pela area de disco do fator de forma
    const float pi ← 3.1415926535
    float Fij ← max(cosi*cosj,0)/(pi*distance2+ShootDArea)
//Modular a energia de saida pela reflectância do patch
//de recepção e área do patch de saida
    float3 delta ← ShootEnergy * RecvColor * ShootDArea * Fij
    return delta
end function

// Operações realizadas no processador de fragmentos,
// acesso a visibilidade e fator de forma
function FragmentShader_formFactor(...)
    float3 ff ← FormFactorEnergy(...)
    bool visibility ← Visible(...)
    OUT.result ← float4(ff*visibility,1)
end function
```

Capítulo 5

Resultados

Nesse capítulo são apresentados os experimentos realizados com dois métodos que implementam a determinação de radiosidade, os resultados e a discussão sobre os mesmos.

Foram implementados dois algoritmos para determinar a radiosidade, o método original e o método por refinamento progressivo. O método original primeiramente era executado na CPU e o *hemi-cube* era renderizado na GPU. A adaptação para execução em GPU teve o foco em resolver o sistema linear na GPU como apontado por Carr et al. [2003]. Já o método de refinamento progressivo se baseou nos trabalhos de Coombe et al. [2004] e Coombe e Harris [2005], onde se fez uma alteração no algoritmo de determinação de visibilidade.

O modelo utilizado nos testes foi a caixa de Cornell[1], onde os valores das frequências foram substituidos por intensidades nos canais RGB. Somente as paredes laterais refletem uma dada cor, e o restante da cena reflete somente a cor branca, sendo que a cena possui somente uma luz no teto.

Os testes desse trabalho foram restritos à cena da caixa de Cornell estática, onde a radiosidade é determinada uma vez e seu resultado é exibido de forma interativa. O resultado esperado em ambos os métodos é a renderização apresentar sombras suaves e a mistura de cores refletidas pelos *patches* da cena. A interpolação do resultado foi utilizada baseada em filtro de texturas somente para dar a aparência suave à renderização.

Os testes foram realizadas em um ambiente com as seguintes configurações:

- **CPU**: Atlhon 64 X2 4200+ 939
- **RAM**: 2GB DDR1 400 Mhz - dual channel
- **SO**: Windows Vista 64bits

[1]Homepage da caixa de Cornell: http://www.graphics.cornell.edu/online/box/ - Acessado em Junho de 2008

- **GPU**: Geforce 8800 GT 512 RAM
- **Versão do *driver* video**: 181.22
- **Compilador**: Microsoft (Visual Studio 2005)
- **Modo compilação**: release (/O2 - *Maximize Speed*)

5.1 Radiosidade original

O método original de radiosidade utilizando o *hemi-cube* para determinar os fatores de forma foi primeiramente implementado parcialmente na CPU com a renderização do *hemi-cube* feita através da GPU.

As primeiras renderizações apresentaram as situações esperadas e podem ser vistas na Figura 5.1. É importante notar que a qualidade da imagem é diretamente afetada pelo nível de subdivisão utilizado.

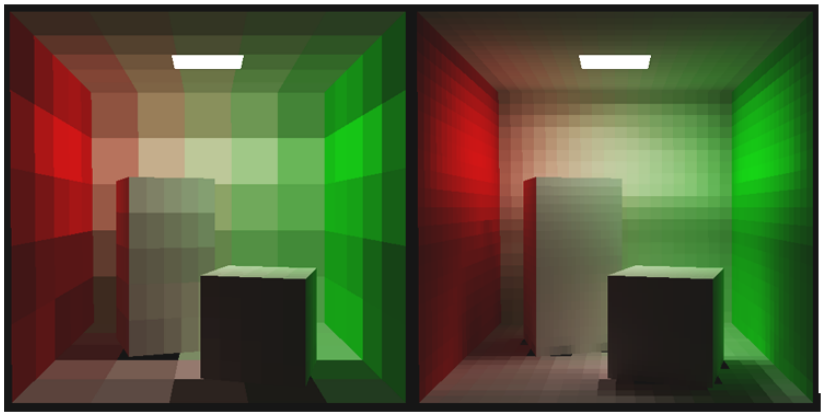

Figura 5.1. Resultado do teste de renderização de radiosidade pela técnica original.

Para verificação dos identificadores do *hemi-cube* foram utilizadas cores para cada objeto da cena e uma câmera foi configurada no piso, como pode ser visto na Figura 5.2.

A Figura 5.3 mostra os testes que foram realizados na GPU, mas com a variação do número de subdivisões. Apesar de vários autores utilizarem interpolações bicúbicas ou bilineares, foi utilizado o filtro bilinear provido pelo *hardware* gráfico.

Como visto nas Figuras 5.1 e 5.3, é necessário subdividir bastante a cena para se obter uma boa definição de sombras. No entanto, o aumento do número de elementos tem impacto na quantidade de memória necessária e precisão para os cálculos. Por

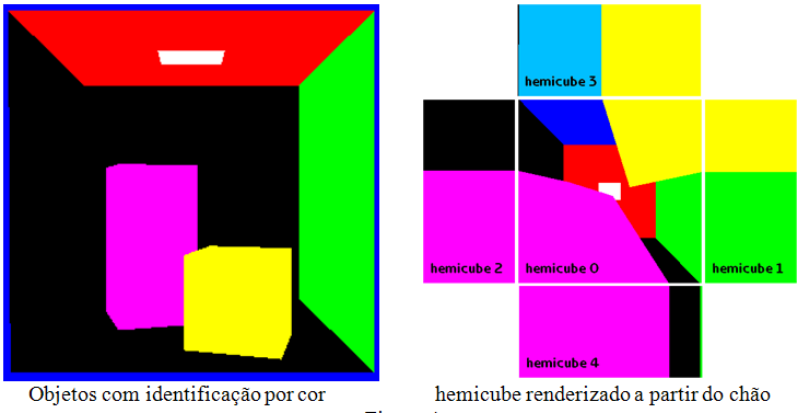

Figura 5.2. Resultado do teste de criação do *buffer* de identificação de objetos no *hemi-cube*.

exemplo: a limitação de bits da GPU fez o método de Jacobi divergir em alguns testes com muitos *patches*.

Desempenho da resolução do sistema linear

Como o alvo de otimização no algoritmo original foi a resolução do sistema de equações, então foi realizado um teste a fim de medir o desempenho dos algoritmos. Foram implementados três algoritmos, o de Gauss-Seidel, o de Jacobi executado na CPU e o de Jacobi executado na GPU.

Foram obtidas 30 amostras do tempo de execução de cada implementação, onde as entradas para o sistema linear (matrizes A e b) foram as matrizes geradas pela radiosidade original com a caixa de Cornell. Os parametros de subdivisão foram variados de forma a processar 69, 154, 273, 426, 613, 834, 1089, 1378, 1701, 2058 e 2449 patches (equações no sistema).

O critério de parada dos métodos foi configurado para iterar até que a variação máxima do conjunto de soluções for menor que duas casas decimais.

A Tabela 1 mostra a média dos tempos em milissegundos juntamente com o desvio padrão para os algoritmos. Cada coluna segue de acordo com a classificação: Seidel (Gaus-Seidel), JacCPU (Jacobi na CPU) e JacGPU (Jacobi na GPU). A Figura 5.4 mostra os tempos dos algoritmos em um gráfico.

Ao considerar os algoritmos que executam na CPU, não existe uma diferença muito grande dos tempos de execução dentro do intervalo utilizado. O algoritmo de Jacobi na GPU obteve os melhores resultados para um quantidade de equações maior que 426. Apesar de ser o melhor algoritmo, ele não seria adequado para cenas grandes, pois foi codificado com apenas uma textura. As texturas possuem um limite de dimensão máxima que depende do *hardware* gráfico.

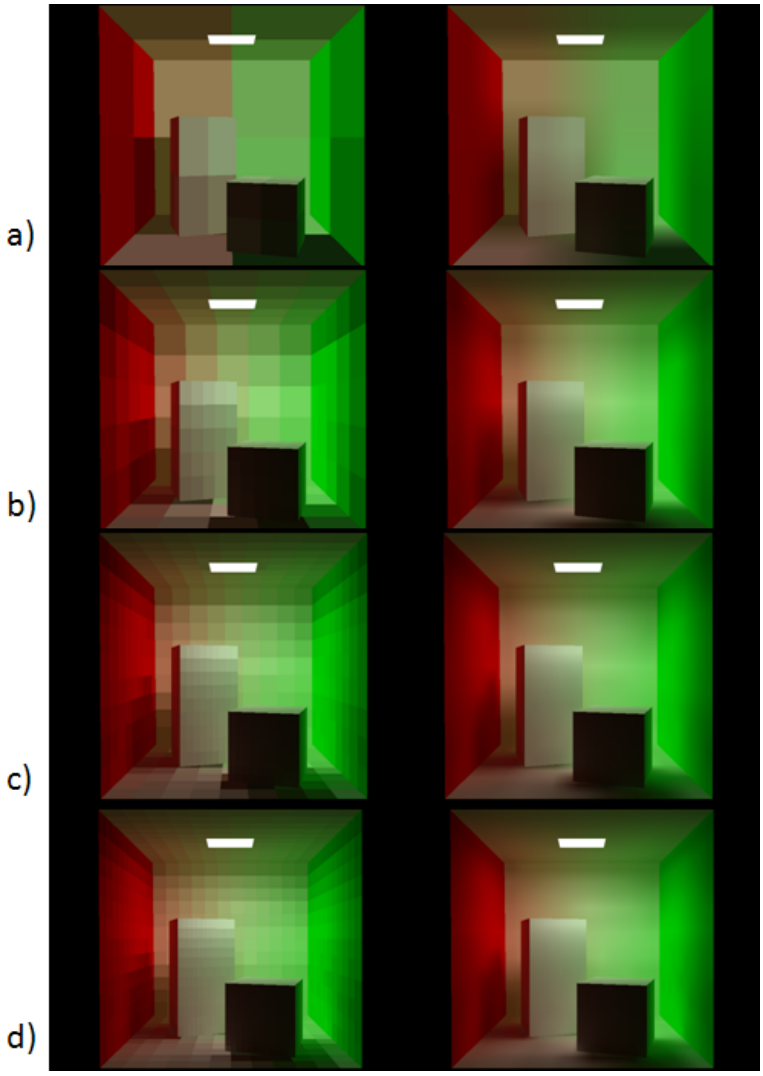

Figura 5.3. Esquerda: imagem com radiosidade dos *patches*. Direita: imagem com interpolação bilinear do *hardware*. a)subdivisão = 2; b)subdivisão = 4; c)subdivisão = 9; d)subdivisão = 12.

5.2 Radiosidade por refinamento progressivo

Como no método original, as imagens apresentaram a mistura de cores e também as sombras suaves. A aproximação para a solução é realizada do elemento de maior para o de menor potência, então é possível visualizar o resultado gradativamente. Geralmente o critério de parada para o método é até a energia residual se torne pequena ou quando o número de "*shooters*" chegar a um certo limite.

Nesse trabalho a radiosidade progressiva pára quando a energia residual for pequena. Caso se considerasse um número máximo de iterações e esse número fosse pequeno, poderia realizar o cálculo de radiosidade a taxas interativas para cenas dinâmicas, restringindo a qualidade da determinação final da radiosidade. A Figura 5.5

Quante. Elementos	Seidel Média	Seidel Desv. Pdr.	JacCPU Média	JacCPU Desv. Pdr.	JacGPU Média	JacGPU Desv. Pdr.
69	0,72	0,06	0,86	0,33	15,09	4,25
154	2,19	0,21	3,35	0,69	15,01	3,62
273	11,79	1,05	10,1	0,67	14,81	4,09
426	37,73	0,78	26,82	2,36	14,35	2,53
613	79	1,8	69,52	2,09	20,15	2,26
834	164,22	2,63	163,33	2,8	41,41	3,66
1089	278,34	3,64	279,46	3,32	54,65	4,33
1378	466,73	8,86	455,07	3,78	71,65	4,48
1701	682,77	6,14	713,92	4,92	109,99	6,62
2058	992,7	5,64	1213,14	6,39	173,87	5,69
2449	1403,55	7,32	1558,16	8,31	217,12	7,16

Tabela 5.1. Tempos em milissegundos para resolução da matriz de radiosidade.

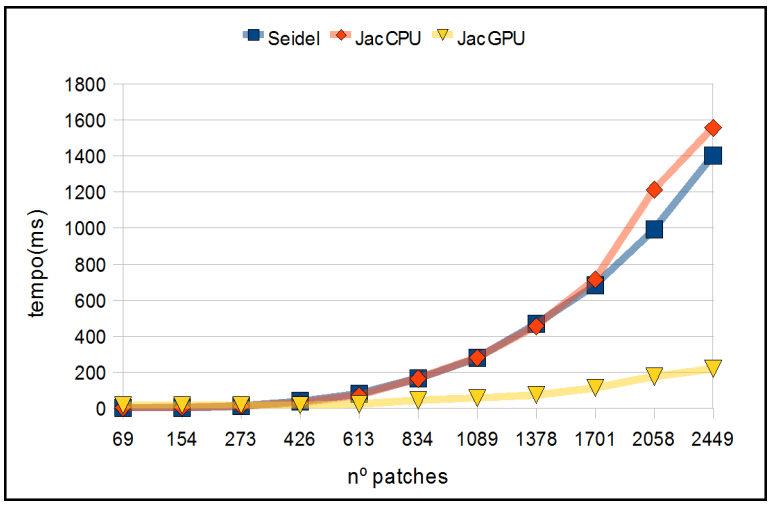

Figura 5.4. Gráfico com tempo entre os algoritmos de resolução de sistemas lineares.

mostra o resultado da radiosidade progressiva com a variação de subdivisão de *patches*, o resultado mesmo com poucas subdivisões ao utilizar o *substructuring* com 32x32 elementos consegue definir bem as sombras.

A função de visibilidade utilizada foi baseada nos trabalhos de Coombe et al. [2004] e Coombe e Harris [2005], mas a mesma apresenta dois problemas com relação à restrição de rasterização do *hardware* e amostragem da projeção.

Para verificar o resultado da função de visibilidade foi utilizada uma renderização da projeção esférica posicionada na luz da cena da caixa de Cornell, onde utilizando a mesma, foi renderizada a cena da câmera principal com valores de verde(claro) para locais visíveis e azul(escuro) para locais não visíveis, dessa forma mostrando a cobertura da visibilidade, assim tornando possível ver o resultado da função de visibilidade de

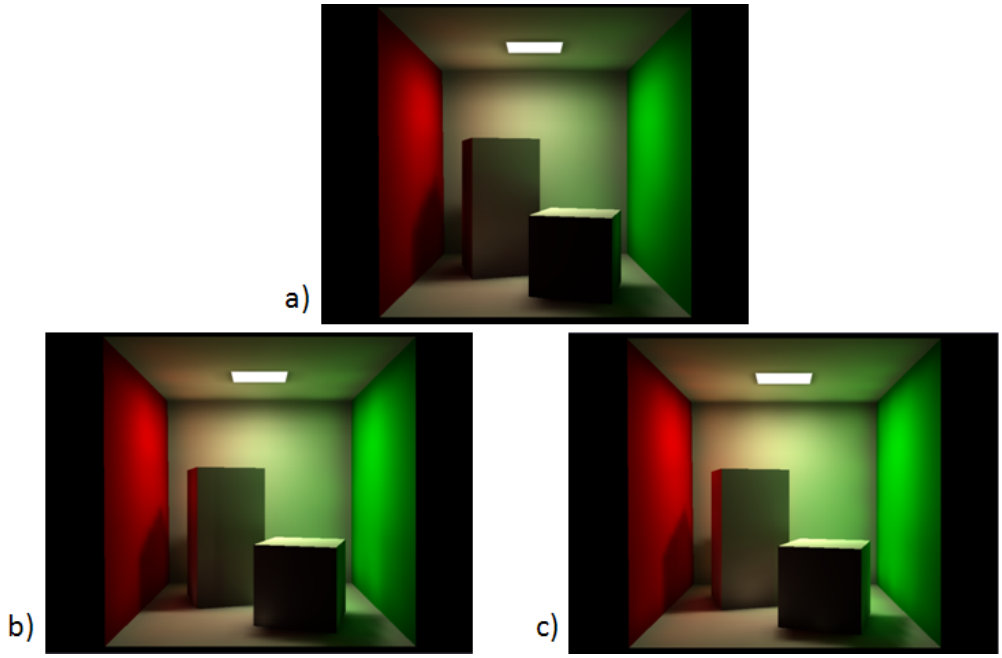

Figura 5.5. Método de refinamento progressivo: a)subdivisão = 2; b)subdivisão = 3; c)subdivisão = 4.

Coombe et al. [2004] e Coombe e Harris [2005] e das propostas de melhora dessa função.

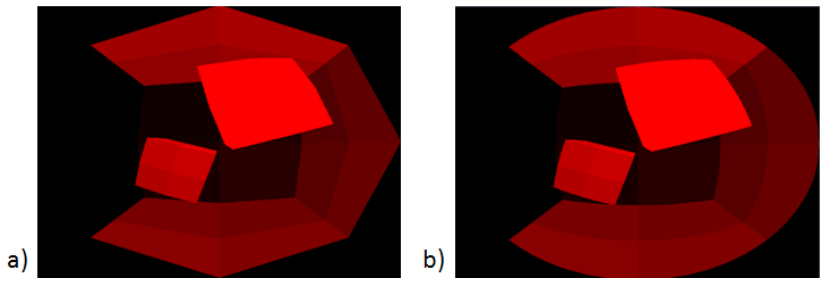

Figura 5.6. Renderização com a câmera posicionada no teto da cena: a)Projeção com rasterização linear sem subdivisão; b) Projeção utilizando o processador de geometria para subdividir os *patches*.

Como pode ser visto na Figura 5.6, após utilizar a subdivisão no processador de geometria, o resultado da visibilidade ficou melhor, pois apresenta uma aproximação para a projeção esférica com maior definição. Apesar disso, ainda apresenta problemas nas bordas dos objetos que foi melhorado com a pesquisa por vizinhos na textura projetada. A Figura 5.7 mostra os três casos: a visibilidade original, com subdivisão e com pesquisa por vizinhos. É importante lembrar que o fator de forma é calculado com base na área visível, ou seja, de acordo com a Figura 5.7 a, o fator de forma da área das paredes não considerariam a real área visível, por apresentar alguns pontos de área não visível que deveriam estar visíveis.

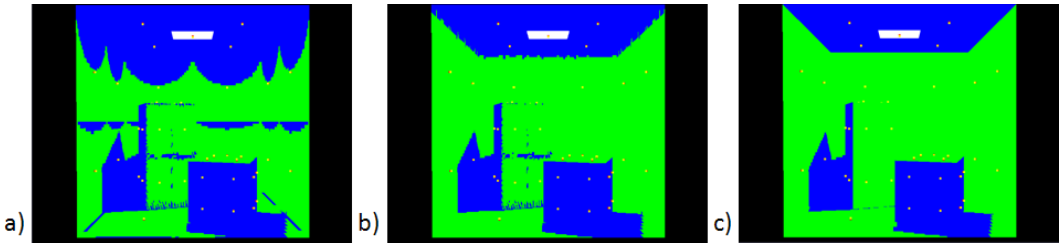

Figura 5.7. Imagens com a relação de cobertura da função de visibilidade: a)Função de visibilidade original proposta por Coombe et al. [2004] e Coombe e Harris [2005]; b)Função de visibilidade utilizando o processador de geometria; c)Função de visibilidade utilizando o processador de geometria e a busca por vizinhos.

Recentemente Wallner [2008], apresentou um método de determinar radiosidade também baseado nos trabalhos de Coombe et al. [2004] e Coombe e Harris [2005], onde o mesmo altera a função de reflectância de Lambert para utilizar mapas de normais na renderização e também propõe uma outra forma de determinar a visibilidade. Ele utiliza a profundidade ao invés de identificadores de objetos, realiza a pesquisa na textura de projeção esférica utilizando a mesma interpolação linear utilizada pelo *hardware* para gerar essa textura e realiza a pesquisa por vizinhos.

Utilizar a profundidade ao invés dos identificadores pode acarretar em problemas de proximidade envolvendo testes com ponto flutuante. Com a utilização de identificadores, esse problema fica presente somente na hora que a geometria é projetada, e não na pesquisa por visibilidade. Qualquer adaptação ou ajuste desses testes ficam concentrados em uma única parte do algoritmo.

Já a projeção com a mesma função de interpolação utilizada sobre os vértices pode fazer a área de pesquisa por visibilidade apresentar problemas de continuidade em relação aos triângulos da cena. Inicialmente essa era uma das soluções que seria utilizada, mas a função de visibilidade não apresenta um resultado bom, a não ser que a área do *patch* seja muito pequena. Como pode ser visto na Figura 5.8, a imagem com a projeção direta apresenta regiões visíveis coerentes e a projeção interpolada não.

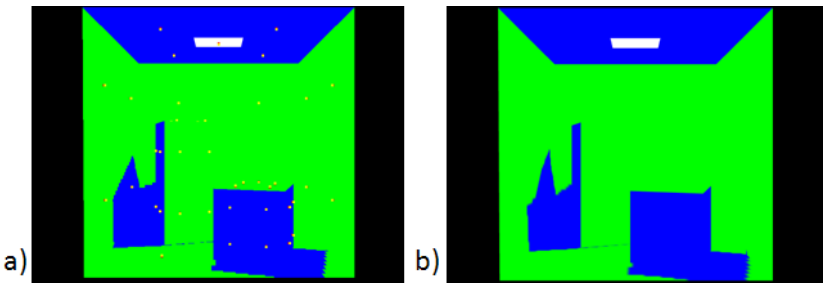

Figura 5.8. Cobertura da visibilidade: a)Projeção direta; b)Projeção com interpolação linear e problema de continuidade.

A solução de visibilidade de Wallner [2008] apresenta o problema de continuidade em virtude da interpolação linear na superfície dos triângulos da cena. Esse problema ocorre porque a interpolação do *hardware* segue a configuração convexa de 3 pontos (triângulos). Na figura 5.9 é possível notar o que acontece ao mapeamento quando se utiliza a configuração convexa de 3 e 4 pontos.

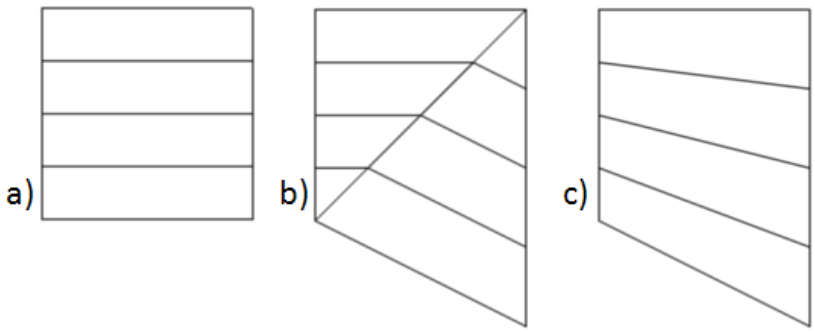

Figura 5.9. Exemplo de continuidade para interpolação: a)Mapeamento original; b)Configuração convexa de 3 pontos; c)Configuração convexa de 4 pontos.

5.3 Radiosidade original e por refinamento Progressivo

Após ter os dois métodos implementados, foi possível também comparar características das renderizações dos mesmos. As diferenças básicas entre os método original e o refinamento progressivo são:

- O método original necessita que todos os fatores de forma sejam processados, o que inclui a determinação de visibilidade e o refinamento progressivo não;
- O resultado do método original considera todos os elementos, independente de sua contribuição sobre o ambiente. O método de refinamento progressivo utiliza somente os elementos que possuem uma quantidade de energia considerável;
- No método original, por necessitar de muita precisão de bits nos cálculos, não permite a determinação de muitos elementos com a limitação de 32 bits para ponto flutuante. No refinamento progressivo esse não é um problema, visto que não é necessário calcular a influência de todos os elementos com todos na forma das equações simultâneas;
- No método original, é impossível exibir o resultado antes da solução do sistema ser atingida, o que não é verdade para o refinamento progressivo, pois o mesmo aproxima da solução gradativamente.

- O método original por necessitar da determinação de todos os fatores de forma restringe sua aplicação no cálculo interativo da radiosidade. No refinamento progressivo é possível realizar poucas renderizações de *patches* para atingir um resultado, o que pode permitir atualização interativa.

Com relação a qualidade das imagens, no método de refinamento progressivo, por ser implementado com *substructuring* apresenta uma definição melhor das áreas de sombra e iluminação sobre a superfície.

Capítulo 6

Conclusão

Nesse trabalho foi realizado um estudo sobre o método de determinação de radiosidade bem como sua estruturação para execução utilizando uma GPU.

Dois algoritmos distintos foram implementados de forma eficiente, onde se desenvolveu uma abordagem melhor para determinação de visibilidade acrescentando qualidade à função do algoritmo tanto de Coombe et al. [2004] e Coombe e Harris [2005] quanto de Wallner [2008].

A implementação realizada, apesar de apresentar várias etapas nos dois algoritmos, não possui todas as características ou extensões que já foram propostas para a radiosidade, como é o caso da informação de radiosidade por *patch*, que utiliza excessivamente identificadores de texturas no *hardware*; ou a subdivisão uniforme juntamente ao *substructuring*. Dessa forma, seria interessante realizar estudos que tenham o foco em utilizar um número menor de texturas, mas que possam ser mapeadas aos objetos da mesma forma que as intensidades separadas, ou utilizar formas adaptativas de realizar/determinar a subdivisão.

Existem dois pontos que são: a aplicação da radiosidade em cenas estáticas; e a restrição de aplicar o cálculo de radiosidade considerando objetos difusos. Seria interessante estudar formas de aplicar a radiosidade em cenas dinâmicas ou mesmo utilizar a forma que Wallner [2008] propôs para calcular a radiosidade utilizando funções de reflectâncias arbitrárias.

Finalmente, esse trabalho se restringe a utilizar a radiosidade aplicada a elementos discretos como foi proposto em sua forma original. Existem outros estudos baseados em resolver a radiosidade de forma estocástica, através de algoritmos baseados principalmente no trabalho de Kajiya [1986], que define a equação de renderização, adapta o problema da radiosidade para ser resolvido por essa equação e propõe um algoritmo chamado path tracing que é semelhante a um *raytracing*, mas é possível determinar a radiosidade de uma cena através do mesmo.

Apêndice A

GPGPU com OpenGL utilizando CGFX

Esse apêndice tem por objetivo exemplificar o método utilizado para programação de GPUs utilizando *shaders* através das APIs CGFX e OpenGL, onde é feita uma abordagem de como os dados podem ser mapeados para a API e é mostrado um exemplo de multiplicação simples.

Apesar desse trabalho utilizar a API gráfica, atualmente essa é a opção que apresenta uma curva de aprendizado maior do que a utilização por exemplo do CUDA da Nvidia ou mesmo do OpenCL que foi lançado recentemente, pois além de se preocupar com a estratégia de paralelização dos algoritmos, ainda é necessário conhecer a API gráfica para realizar uma implementação eficiente.

A primeira sessão apresenta o OpenGL, onde são listadas algumas extensões importantes. A segunda sessão aborda o CGFX, onde se descreve o que é um perfil de compilação. A terceira sessão aborda o mapeamento de dados de um programa convencional para uma textura e como seria um *pipeline* capaz de processar esses dados. Por último, na quarta sessão, são mostrados dois programas, um que é o kernel que executa sobre a GPU escrito em CG e outro em C/C++ que carrega as estruturas necessárias para utilizar a GPU.

É importante ressaltar que esse apêndice não aborda detalhadamente cada uma das APIs envolvidas, pois as mesmas são extensas e possuem diversos exemplos e documentos que podem ser consultados[1,2,3,4].

[1]OpenGL - http://www.opengl.org/
[2]GLee - http://www.elf-stone.com/glee.php
[3]CG Toolkit - http://developer.nvidia.com/object/cg_toolkit.html
[4]FreeGLut - http://freeglut.sourceforge.net

A.1 OpenGL

O OpenGL é uma API mantida pelo grupo Khronos voltada para renderizações. Existem diversos fabricantes de placas de vídeo que dão suporte à API através de drivers para sistemas operacionais específicos. Ele possui um conjunto básico de funções que é chamado de OpenGL1.0. A medida que novas funcionalidades são criadas, são criadas também extensões para a API, e a cada conjunto de extensões é atribuida uma nova versão ao OpenGL.

A API está especificada em forma de cabeçalhos escritos em C, onde a mesma pode ser utilizada em todas as linguagens que possuam alguma forma de ligação com bibliotecas de vínculo dinâmico. As extensões são adquiridas através de buscas (*query*) dentro da API pelos ponteiros de função. Geralmente, a utilização de uma extensão envolve o conhecimento de todas as funções que a mesma define (para quais *queries* realizar) e de todas as constantes que podem ser utilizadas por essas funções. Para evitar o manuseio diretamente das extensões, existem algumas bibliotecas que realizam a busca de funções das extensões automaticamente além de definirem as constantes necessárias, como é o caso do GLew(OpenGL Extension Wrangler Library) e GLee(OpenGL Easy Extension).

Nesse trabalho foi utilizada a GLee, pois para utilizá-la basta adicionar o cabeçalho da biblioteca e compilar o arquivo objeto GLee juntamente à fonte do programa, sem a necessidade de instalação de bibliotecas dinâmicas no sistema.

As extensões utilizadas pelo exemplo de GPGPU desse apêndice são:

- **GL_ARB_texture_non_power_of_two**: Para a criação de texturas com dimensões que não são potência de 2.
- **GL_ARB_texture_float**: Para utilizar texturas onde os componentes das mesmas possam ser do tipo *float* (ponto flutuante).
- **GL_EXT_framebuffer_object**: Para que se possa implementar o método de *render to texture* através da associação com texturas.

As extensões somente podem ser adquiridas pela aplicação se existir um contexto OpenGL criado, por isso foi utilizada a biblioteca FreeGLut para criar o contexto de renderização do OpenGL.

A.2 CGFX

CGFX é um formato de arquivo que utiliza a API CG da nvidia para programação de *shaders*. Através dele, é possível definir vários *shaders* para serem executados sobre

um conjunto de primitivas gráficas (triângulos, linhas ou pontos).

A linguagem CG é semelhante à linguagem C, onde existem tipos de dados que contém um conjunto de valores em uma única variável, como o float2 (conjunto de 2 valores em ponto flutuante), float3, float4. Existem também tipos de matrizes como o mat2x2, mat3x3 e mat4x4.

A linguagem define várias operações entre os tipos de dados através de operadores aritméticos(+ - * /) ou funções como a **mul(A,B)** que realiza a multiplicação matricial de uma matriz com uma matriz ou de uma matriz com um vetor. A especificação que acompanha o CG Toolkit possui uma lista de todas as operações possíveis entre os tipos, além da definição das funções auxiliares.

A.2.1 Perfil de compilação

Quando um *shader* é compilado pela API, o mesmo deve possuir uma associação a um conjunto de recursos que o hardware tem disponível. Esse conjunto de recursos é definido por um determinado perfil. Como exemplo de um perfil, tem-se o gp4vp, que possui as funcionalidades da série 8 das placas da NVidia (como o suporte a branch dinâmico).

Se um hardware não atende as especificações de um perfil, o resultado da compilação pode ser satisfeito, mas a ligação com o *driver* da GPU retorna uma falha.

A.3 Mapeamento de dados

Geralmente os dados de apliações GPGPU são armazenados em forma de textura, pois é uma estrutura eficiente que pode ser utilizada para leitura pelos *shaders*(*kernels*).

As texturas são matrizes ou imagens que podem possuir 1, 2 ou 3 dimensões. Cada elemento da textura, chamado de *texel*, pode ser composto por 1, 2, 3 ou 4 componentes em um formato específico, que podem ser inteiros ou ponto flutuante com quantidades de bits variadas (o formato depende do suporte que o hardware possui).

A.3.1 Carregar e ler dados

A função de carga (upload) ou leitura (download) de texturas do OpenGL considera a entrada como uma área de memória contínua, onde as linhas da imagem são posicionadas seqüencialmente no vetor. A Figura A.1 mostra uma seqüência de dados contínua, em seguida como poderia ser a divisão dessas linhas de modo a formar uma matriz e no final uma matriz de textura. É possível utilizar texturas unidimensionais, mas o limite de elementos em uma dimensão pode não ser suficiente para armazenas todos os

dados necessários, por isso geralmente são utilizadas texturas bidimensionais mesmo para representar uma sequencia de dados unidimensional.

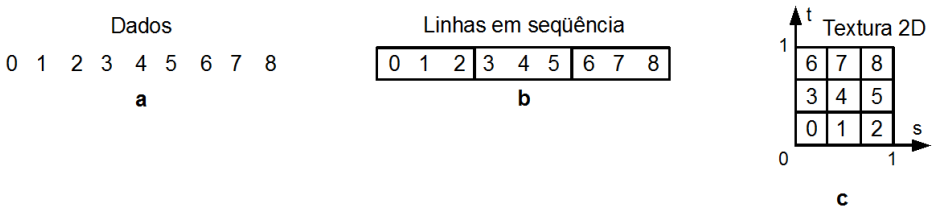

Figura A.1. a) Dados em um vetor; b) Separação dos dados em linhas consecutivas; c) Dados na disposição de uma textura.

A.3.2 Acesso a textura

A forma de acessar os elementos de uma textura são feitos através do mapeamento das dimensões da textura para um quadrado de dimensão 1x1. Assim, ao acessar um determinado elemento, é necessário realizar uma tranformação da coordenada de acesso para esse espaço normalizado (espaço de textura).

Como exemplo temos a textura de dimensão 3x3 e queremos acessar algum elemento dessa matriz. Para isso, podemos utilizar a função de acesso $f(x, y) = TEX(\frac{x+0,5}{w}, \frac{y+0,5}{h})$, onde w e h são as dimensões da textura sobre os eixos x e y respectivamente. O deslocamento de $0,5$ aplicados às coordenadas garante ao endereçamento indexar o centro de um *texel* no espaço de textura. Essa função de acesso pode ser utilizada no OpenGL, mas não na API da Microsoft (DirectX), pois a forma de endereçamento ja realiza esse deslocamento.

As texturas quando utilizadas em renderizações, podem ser filtradas de várias formas para suavizar a transição de um *texel* para outro, geralmente esses filtros não são bons pra GPGPU, uma vez que é necessário utilizar os valores discretos dos texels.

Além dos filtros, ainda existem os modos de repetições de coordenadas de texturas quando um valor de acesso estiver fora da faixa de mapeamento normalizado, mas normalmente as repetições são desabilitadas para evitar acessos cíclicos nos eixos das texturas.

A.3.3 Executando o kernel sobre uma textura

A forma mais simples de utilizar um kernel é definir uma textura de entrada e saida com a mesma dimensão e desenhar um quadrado sobre a textura de saida.

São necessários dois kernels para realizar essa execução: um para transformas as coordenadas (vértices) do quadrado para o espaço normalizado da API de renderização

e o outro que efetivamente realiza o processamento para cada um dos *texels* da textura de saida.

A Figura A.2 mostra um exemplo do que acontece no *pipeline* de renderização quando um quadrado é aplicado sobre a API que possui um processador de vértice e um processador de fragmentos.

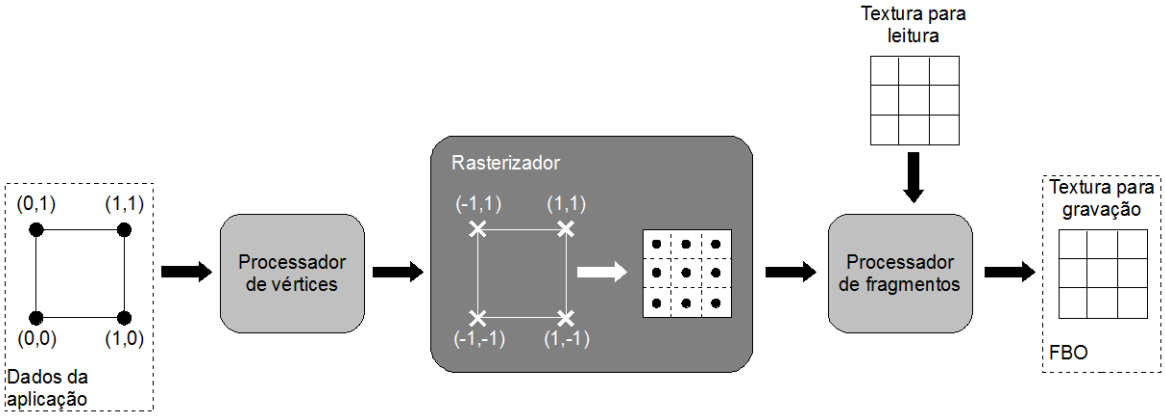

Figura A.2. Exemplo de um *pipeline* com entrada de coordenadas normalizadas e saida direcionada para textura através de um FBO(*framebuffer object*).

A.4 Exemplo

Nessa sessão é mostrado um exemplo de *shader* em CGFX que possui três blocos principais. Em todo o *shader* que executa sobre o processador de vértice é necessário gravar na saida uma determinada posição, da mesma forma, todo o processador de fragmento deve escrever, no final, a cor de um *pixel* no *framebuffer*.

Exemplo de *shader* em CGFX:

```
-----------------------------------------------------------------------------------------
// Arquivo "./shaders/shaders.cgfx"

//sampler(textura) utilizada para leitura
sampler2D input_stream:input_stream;

/*****************************************************************
 1) Definição de estruturas de entrada e saida
 *****************************************************************/

//entrada para o processador de vértice
struct VertIn{
  float4 pos : POSITION;
};
```

```
//saida do processador de vértice e
//entrada do processador de fragmentos
struct Vert2Frag{
  float4 pos : POSITION;
  float2 texAccess : TEXCOORD0;
};

//saida do processador de fragmentos
struct FragOut{
  float4 output_stream: COLOR0;
};

/*************************************************************
 2) Shaders
 *************************************************************/

//Shader utilizado para processar vértice
void VS(in VertIn IN, out Vert2Frag OUT){

  // atribui diretamente a coordenada normalizada para o rasterizador
  OUT.pos = IN.pos;

  // calcula a coordenada de enderecamento de textura de acordo com a
  //coordenada que é passada para o rasterizador
  //    Transforma as coordenadas no intervalo de [-1..1] para [0..1]
  OUT.texAccess = IN.pos.xy*0.5+0.5;
}

//Shader utilizado para processar fragmentos
void FS(in Vert2Frag IN, out FragOut OUT){

  // realiza uma leitura de textura input_stream, utilizando a
  // coordenada calculada anteriormente
  float value = tex2D(input_stream,IN.texAccess).r;

  //  Cálculo da função f(x) = x*2
  value = value*2;

  // gravação do resultado
  OUT.output_stream = float4(value);
}

/*************************************************************
 3) Tecnicas
 *************************************************************/

//Técnica exemplo1 que define quais shaders
//irão ser executados nos processadores de vértice e de fragmentos
technique exemplo1 {

 // Definição de 1 passo de execução
 pass P0{

  // atribui ao processador de vértice o shader VS com o perfil arbvp1
```

```
  VertexProgram = compile arbvp1 VS();

  // atribui ao processador de fragmentos o shader FS com o perfil arbfp1
  FragmentProgram = compile arbfp1 FS();
 }

}
-----------------------------------------------------------------------------------------
```

O *shader* acima possui 3 blocos principais, que são eles:

1. **Definição de estruturas**: Aqui as informações vindas da aplicação e as informações que são transferidas de um processador para outro no *shader* devem ser especificadas. Uma forma de se fazer essa definição é criar uma *struct* para cada informação.

2. **Definição dos shaders**: Os *shaders* são os programas que irão ser executados para cada elemento que for de responsabilidade de um determinado processador. O processador de vertice irá executar o *shader* de vértice para cada vértice que for passado para a API e o processador de fragmentos irá processa todos os *pixels* gerados pelo rasterizador.

3. **Definição das técnicas**: A técnica é o que define quais *shaders* estarão associados a quais processadores e qual o perfil de compilação mínimo (exigido) para um dado *shader*.

A.4.1 Interface em C/C++

A definição do *shader* é simples pois é necessário apenas definir um kernel e seus parâmetros. A interface da API gráfica com o *shader* precisa da criação de diversos identificadores e inicialização dos mesmos, pois aqui é que se determina a execução do programa.

O código mostrado não possui a parte de inicialização de janela, pois a mesma pode ser facilmente encontrada como exemplo de uma aplicação FreeGLut.

Exemplo de código em C/C++ para utilização do *shader*:

```
-----------------------------------------------------------------------------------------
/*************************************************************
 1) Definição de dados do programa principal
 *************************************************************/
float dados[5*5]={ 1,  2,  3,  4,  5,
                   6,  7,  8,  9, 10,
                  11, 12, 13, 14, 15,
                  16, 17, 18, 19, 20,
                  21, 22, 23, 24, 25};
```

```
float resultado[5*5];

/*************************************************************
 2) Variaveis das APIs
 *************************************************************/

//identificadores de texturas
GLuint inputStreamTexture;
GLuint outputStreamTexture;

//identificadores auxiliares
GLuint outputFBO;
CGcontext context;
CGeffect effect;
CGtechnique activedTech;
CGparameter inputParam;
CGpass pass;

//confiurando a textura para ser modulada pela cor do poligono
glTexEnvi(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_MODULATE);

//ativando textura 0, pois a multitexturização pode estar
//ativa com outro identificador
glActiveTextureARB(0);

//ativando a textura no pipeline de renderização
glEnable(GL_TEXTURE_2D);

/*************************************************************
 3.A) Configurando e carregando inputStreamTexture
 *************************************************************/

//gerando um identificador de textura no OpenGL
glGenTextures(1,&inputStreamTexture);

//ativando o identificador de textura inputStreamTexture
glBindTexture(GL_TEXTURE_2D,inputStreamTexture);

//configuração de modo de wrap da textura entrada
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

//configuração de modo de filtro da textura entrada
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

//criando textura de ponto flutuante com o tipo GL_RGBA32F_ARB, ou seja,
//   textura de ponto flutuante de 32 bits
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F_ARB, 5/*w*/, 5 /*h*/,
             0, GL_RGBA, GL_UNSIGNED_BYTE, 0);

//carregando os dados na textura de entrada
glTexSubImage2D(GL_TEXTURE_2D,0,0,0, 5/*w*/, 5 /*h*/,
                GL_RED,//formato dos dados de entrada
```

```
                GL_FLOAT,//tipo dos dados que serão carregados na textura
                dados);//ponteiro

/*************************************************************
 3.B) Configurando outputStreamTexture
 *************************************************************/

//gerando um identificador de textura no OpenGL
glGenTextures(1,&outputStreamTexture);

//ativando o identificador de textura outputStreamTexture
glBindTexture(GL_TEXTURE_2D,outputStreamTexture);

//configuração de modo de wrap da textura entrada
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

//configuração de modo de filtro da textura entrada
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

//criando textura de ponto flutuante com o tipo GL_RGBA32F_ARB, ou seja,
//   textura de ponto flutuante de 32 bits
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F_ARB, 5/*w*/, 5 /*h*/,
             0, GL_RGBA, GL_UNSIGNED_BYTE, 0);

/*************************************************************
 3.C) Configurando FBO(Framebuffer Object)
 *************************************************************/

glGenFramebuffersEXT(1, &outputFBO);

//ativando outputFBO
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, outputFBO);

//associando textura de saida ao FBO
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
                          GL_TEXTURE_2D, outputStreamTexture, 0);

//setando dimensões do viewport de render para a dimensão do FBO
glViewport(0,0,w,h);

/*************************************************************
 3.D) Configurando shader
 *************************************************************/

//criando o contexto
context = cgCreateContext();
cgGLRegisterStates(context);
cgGLSetOptimalOptions(cgGLGetLatestProfile(CG_GL_VERTEX));
cgGLSetOptimalOptions(cgGLGetLatestProfile(CG_GL_FRAGMENT));

//carregando arquivo de shader
effect = cgCreateEffectFromFile(context, "./shaders/shaders.cgfx", NULL);
```

```
//ativando técnica dentro do shader
activedTech = cgGetNamedTechnique(effect,"exemplo1");

//pegando identificador de entrada do shader
inputParam = cgGetEffectParameterBySemantic(effect, "input_stream");

//setando textura de entrada do shader
cgGLSetTextureParameter(inputParam, inputStreamTexture);

/*************************************************************
 4) Executando o Kernel
 *************************************************************/

pass = cgGetFirstPass(activedTech);
while(!pass){
  cgSetPassState(pass);

//desenhando poligono com as dimensões do FBO
  glBegin(GL_QUADS);
  glVertex2f(-1,-1);
  glVertex2f( 1,-1);
  glVertex2f( 1, 1);
  glVertex2f(-1, 1);
  glEnd();

  cgResetPassState(pass);
  pass = cgGetNextPass(pass);
}

/*************************************************************
 5) Lendo resultado do processamento da textura
 *************************************************************/

glBindTexture( GL_TEXTURE_2D, outputStream);
glGetTexImage( GL_TEXTURE_2D, 0, GL_RED, GL_FLOAT, resultado );

/*************************************************************
 6) Liberando recursos
 *************************************************************/

cgDestroyContext(context);
glDeleteFramebuffersEXT(1, &outputFBO);
glDeleteTextures(1,&inputStreamTexture);
glDeleteTextures(1,&outputStreamTexture);
-------------------------------------------------------------------------------------------
```

Os blocos que constituem o programa são:

1. **Definição de dados**: Geralmente os dados que serão processados são definidos como um vetor de vários elementos simples, nesse caso é um vetor de 25 números em ponto-flutuante.

2. **Variaveis das APIs**: Como são utilizadas duas APIs em conjunto, então existirá uma quantidade considerável de variáveis destinadas a gerenciar os recursos de cada uma das APIs.

3. **Configurando texturas e shader**: A configuração de texturas deve ser feita de acordo com os dados que foram definidos, uma vez que temos um vetor de 25 números, devemos criar uma textura que armazene esses dados. O *shader* deve estar preparado para receber a textura como entrada, por isso é necessário configurar a textura na forma de um *sampler*. E o FBO(*framebuffer object*) deve ser criado associado a uma textura para que possa ser possível gravar dados na mesma.

4. **Executando o Kernel**: A execução do kernel, sem o FBO, escreveria no framebuffer do contexto de OpenGL criado. Para que o mesmo escreva na textura de saida, foi utilizado o FBO para redirecionar a renderização para uma área de memória interna da placa de vídeo(textura).

5. **Lendo o resultado da GPU**: Após a execução, é necessário somente realizar uma leitura da textura que está associada ao FBO.

6. **Liberando memória**: Antes do programa ser finalizado, é importante liberar todos os recursos alocados, uma vez que a placa de vídeo é compartilhada por vários programas, se essa etapa não for bem estruturada, pode fazer com que outras aplicações apresentem uma perda de desempenho.

Referências Bibliográficas

Akenine-Moller, T. e Haines, E. (2002). *Real-time rendering. Second edition.* A K Peters, Ltd.

Bastos, R.; Goslin, M. e Zhang, H. (1997). Efficient radiosity rendering using textures and bicubic reconstruction. In *Symposium on Interactive 3D Graphics*, pp. 71–74, 184.

Bergman, L.; Fuchs, H.; Grant, E. e Spach, S. (1986). Image rendering by adaptive refinement. *SIGGRAPH Comput. Graph.*, 20(4):29–37.

Carr, N.; Hall, J. e Hart, J. (Proc. Graphics Hardware 2003). Gpu algorithms for radiosity and subsurface scattering.

Cohen, M.; Greenberg, D.; Immel, D. e Brock, P. (1986). An efficient radiosity approach for realistic image synthesis. *Computer Graphics and Applications, IEEE*, 6(3):26–35.

Cohen, M. F.; Chen, S. E.; Wallace, J. R. e Greenberg, D. P. (1988). A progressive refinement approach to fast radiosity image generation. *SIGGRAPH Comput. Graph.*, 22(4):75–84.

Cohen, M. F. e Greenberg, D. P. (1985). The hemi-cube: a radiosity solution for complex environments. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pp. 31–40, New York, NY, USA. ACM Press.

Cook, R. L. e Torrance, K. E. (1982). A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24.

Coombe, G. e Harris, M. (2005). *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter Global Illumination Using Progressive Refinement Radiosity, pp. 635–647. Addison Wesley.

Coombe, G.; Harris, M. J. e Lastra, A. (2004). Radiosity on graphics hardware. In *GI '04: Proceedings of Graphics Interface 2004*, pp. 161–168, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada. Canadian Human-Computer Communications Society.

Foley, J. D.; van Dam, A.; Feiner, S. K. e Hughes, J. F. (1996). *Computer Graphics: Principles and Pratice, 2nd ed. in C.* Addison-Wesley.

Goral, C. M.; Torrance, K. E.; Greenberg, D. P. e Battaile, B. (1984). Modeling the interaction of light between diffuse surfaces. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pp. 213–222, New York, NY, USA. ACM.

Hanrahan, P.; Salzman, D. e Aupperle, L. (1991). A rapid hierarchical radiosity algorithm. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pp. 197–206, New York, NY, USA. ACM.

Harris, M. e Buck, I. (2005). *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter GPU Flow-Control Idioms, pp. 547–555. Addison Wesley.

Kajiya, J. T. (1986). The rendering equation. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pp. 143–150, New York, NY, USA. ACM Press.

Lindholm, E.; Nickolls, J.; Oberman, S. e Montrym, J. (2008). Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55.

Lum, E. B.; Ma, K.-L. e Max, N. (2005). Calculating hierarchical radiosity form factors using programmable graphics hardware. *journal of graphics tools*, 10(4):61–71.

Mark, W. R.; Glanville, R. S.; Akeley, K. e Kilgard, M. J. (2003). Cg: a system for programming graphics hardware in a c-like language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pp. 896–907, New York, NY, USA. ACM Press.

Möller, T. (1996). Radiosity techniques for virtual reality - faster reconstruction and support for levels of detail. In *WSCG '85: Proceedings of the Fourth International Conference in Central Europe on Computer Graphics and Visualization*, pp. 209–216, Czech Republic. Plzen.

Opengl; Shreiner, D.; Woo, M.; Neider, J. e Davis, T. (2005). *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional.

Owens, J. D.; Luebke, D.; Govindaraju, N.; Harris, M.; Krüger, J.; Lefohn, A. E. e Purcell, T. J. (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*.

Phong, B. T. (1973). *Illumination for computer-generated images.* PhD thesis.

Ruggiero, M. A. G. e Lopes, V. L. R. (1988). *Cálculo numérico: aspectos teoricos e computacionais*. São Paulo : McGraw-Hill.

Wallace, J. R.; Elmquist, K. A. e Haines, E. A. (1989). A ray tracing algorithm for progressive radiosity. *SIGGRAPH Comput. Graph.*, 23(3):315–324.

Wallner, G. (2008). Gpu radiosity for triangular meshes with support of normal mapping and arbitrary light distributions. In Skala, V., editor, *WSCG'2008 - The 16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2008*, Journal. University of West Bohemia.

Whitted, T. (1979). An improved illumination model for shaded display. *SIGGRAPH Comput. Graph.*, 13(2):14.