Scan Paralelo em Arquitetura CUDA para Jogos 3D

Aluna: Patricia Zalmon Rosenberg
Orientador: Bruno Feijó

Introdução

Jogos 3D requerem altíssimo desempenho de tempo real que só pode ser alcançado através de algoritmos paralelos. Existem blocos de construção de algoritmos paralelos que são usados em um grande número de situações. Um dos mais simples e úteis destes blocos de construção é a operação de somas acumulativas de prefixos (*all-prefix-sums operation*) também chamada de **soma de prefixos** ou simplesmente **scan**.

O scan recebe um operador associativo binário $\oplus$ com identidade I e um vetor de n elementos

$$[a_0, a_1, \cdots, a_n]$$

e retorna o vetor

$$[I, a_0, (a_0 \oplus a_1), (a_0 \oplus a_1 \oplus a_2), \cdots, (a_0 \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_{n-2})]$$

Por exemplo, se $\oplus$ é a *adição* + com identidade 0, então o scan no vetor [3 1 7 0 4 1 6 3] retorna [0 3 4 11 11 15 16 22]. Outros exemplos de operadores binários são o *min* (mínimo) e o *max* (máximo).

O scan pode ser empregado para implementar vários algoritmos, tais como *quicksort*, *quickhull*, compactação de *stream*, *summed-area tables*, entre outros. Além disso, podemos também usá-lo para comparar lexicamente strings de caracteres, somar números de precisão maior do que o *Double* (os chamados *big numbers*), avaliar polinômios, resolver sistemas lineares tridiagonais e alocar processadores dinamicamente.

O primeiro passo na pesquisa de scan paralelo em jogos 3D é o domínio do scan sequencial e de sua aplicação em computação gráfica. Posteriormente, como segunda etapa, os seguintes tópicos deveriam ser atacados: scan paralelo usando a biblioteca **CUDPP – CUDA Data Parallel Primitives Library** CUDA, scan paralelo replicando o algoritimo usado pela CUDPP e, por fim, a busca por um algoritmo de scan paralelo mais eficiente.

Objetivos

O primeiro objetivo desta pesquisa é estudar Computação Gráfica e OpenGL [1] [2] e usar a soma de prefixos sequencial (scan sequencial) na técnica de *Summed Area Table* para uma aplicação típica de renderização 3D: profundidade de campo (*Depth Field*). Depois, o objetivo é indicar a direção para a segunda etapa da pesquisa que consiste em usar o scan paralelo.

Metodologia

O Realismo de uma cena:

Uma cena tridimensional sintetizada pelo computador tem a particularidade de possuir uma *profundidade de campo* infinita. Ao visualizarmos uma imagem com os objetos perfeitamente detalhados, vemos uma falta de realismo visual. Para termos um aspecto real de uma imagem, esta deve ser aproximada ao máximo daquela que um ser humano poderia captar com seus próprios olhos ou com câmaras ópticas.

A *profundidade de Campo* (em inglês denominada de *Depth Field)* é um dos fatores responsáveis pelo realismo de uma imagem. A profundidade de campo existe no processo

natural da visualização humana e atua na nitidez das imagens. Na imagem realista, a área focada é vista de forma nítida, enquanto as restantes são exibidas com certo desfoque, conforme seu afastamento.

Utilizando OpenGL [1][2], primeiro implementamos um algoritmo que gera uma imagem de várias chaleiras com profundidade de campo. Depois estudamos o método criado por Frank Crow, em 1984 [7], que introduziu o conceito de *Summed Area Table* (**SAT**) para permitir uma maior filtragem de textura. Em seguida, estudamos a proposta de Hensley et al. [5], publicada em 2005, que consiste em uma técnica ainda mais eficiente.

Geração de um SAT e sua aplicação em profundidade de campo:

A imagem gerada na tela é composta de uma matriz retangular de pixels (Figura 1) [8], cada um capaz de exibir um pequeno quadrado de cor num determinado ponto. Para realizar o desenho, precisamos saber a cor que cada pixel possui. Essa informação é encontrada armazenada no *buffer* de cor.

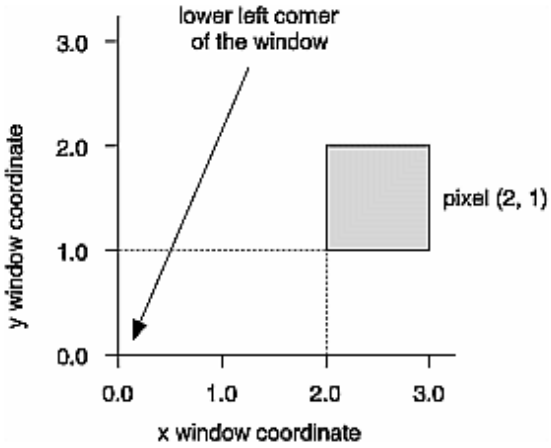

Figura 1: Região ocupada por um pixel (fonte: [8]).

O *Summed Area Table* [4] de uma imagem corresponde à chamada **imagem integral**, onde cada pixel tem um valor de intensidade que é igual à soma das intensidades de todos os pixels que o precedem acima e à esquerda (e mais a dele próprio). Obtemos a imagem SAT aplicando a soma scan (operação *all-prefix-sums* em uma matriz) para todas as linhas (Figuras 2b e 3b) da imagem original (Figuras 2a e 3a), seguida da soma scan de todas as colunas do resultado. A Figura 3c é a representação da imagem integral da Figura 3a (*e.g.* o pixel i=3 e j=2 tem valor 11 na Figura 3c, pois é igual a 2+3+2+3+0+1 na Figura 3a).

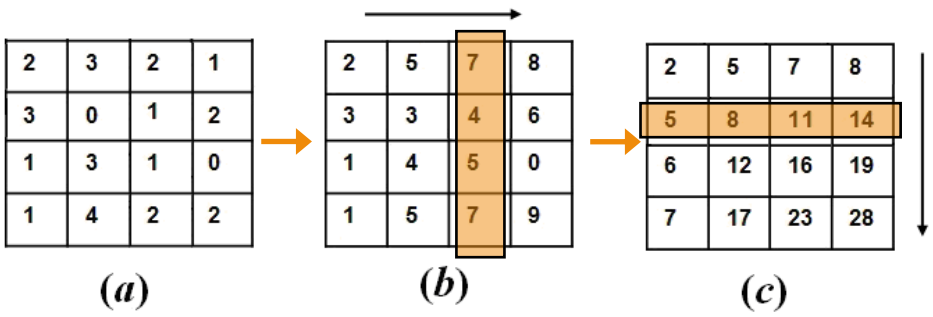

Figura 2: Imagem original em (a). Soma scan para todas as linhas em (b). Soma para as colunas em (c).

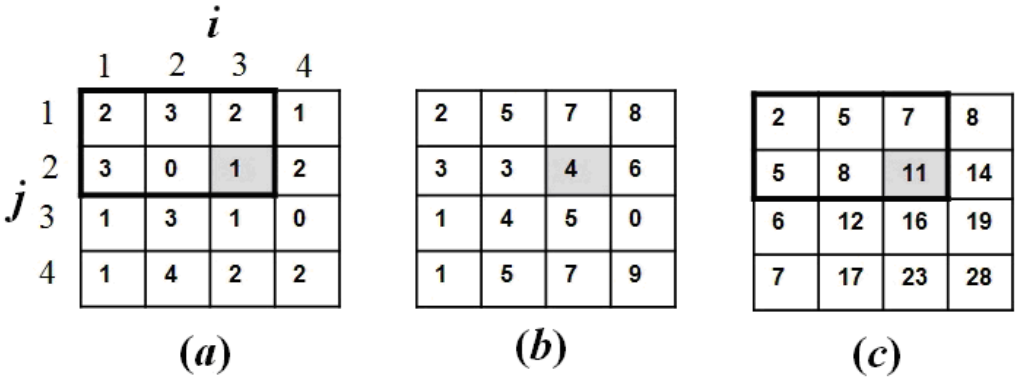

Figura 3: Processo de calcular o SAT

Usando como fonte uma textura com elementos a[i, j], podemos construir uma *Summed Area Table t*[*i,j*] (Figura 4), para que isso ocorra, usamos a equação a seguir:

$$t[i,j] = \sum_{x=0}^{i}\sum_{y=0}^{j} a[x,y]. \qquad (0)$$

Observamos que ao escolhermos um ponto (*x*, *y*) de maneira aleatória podemos ver que em todos os casos se obedece à condição de soma mencionada acima. Podemos computar a soma dos valores de uma área (região sombreada da Figura 4a), que é 3+2+4+7=16, a partir dos valores dos pixels que a delimitam na imagem integral (pixels sombreados na Figura 4b): 28-8-6+2=16. Esta propriedade está generalizada no parágrafo a seguir.

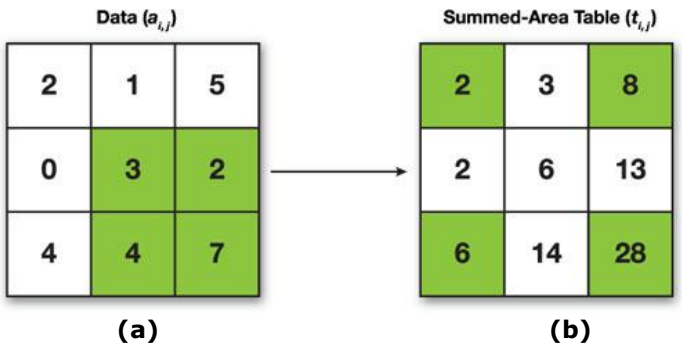

Figura 4: Outro exemplo de SAT.

A soma de todas as intensidades de um conjunto de pixels no retângulo $m \times n$ (chamado de **máscara de filtro**) da Figura 5a pode ser obtido pela seguinte equação (Figura 5):

$$T_R = s_{lr} - s_{ur} - s_{ll} + s_{ul} \qquad (1)$$

onde s_{lr} é o valor de intensidade do pixel mais embaixo e à direita da máscara de filtro (*lower-right*) na imagem integral (*i.e.* no SAT), s_{ul} é o valor do pixel mais acima e à esquerda (*upper-left*) da máscara e assim por diante. Na Equação (1), s_{ul} corresponde à soma das intensidades dos pixels da região R_{ul} na imagem original. Esta soma é subtraída duas vezes quando usamos os termos s_{ur} e s_{ll} na Equação (1). Por esta razão, somamos s_{ul} no final desta equação (ver subtração das áreas na Figura 5b. O valor final da intensidade do pixel (no centro da máscara)

é dado pelo valor médio da intensidade dos pixels cobertos pela máscara; portanto, o valor da aplicação deste filtro (chamado de *box filter* e que é um filtro de convolução) é dado por:

$$s_{filtro} = T_R / m \times n \qquad (2)$$

No exemplo da Figura 5a, a máscara é 5×5.

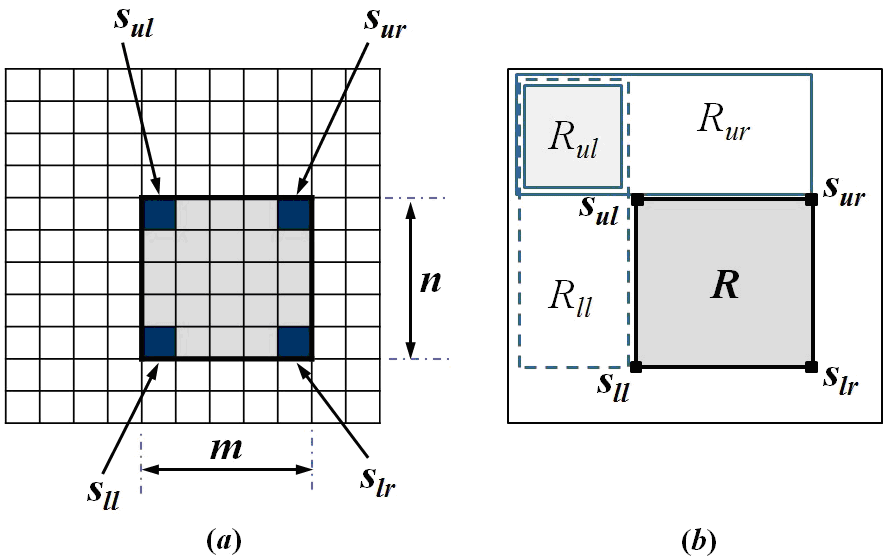

Figura 5: (a) Conjunto de pixels no retângulo *m* × *n* (chamado de máscara de filtro). (b) Soma de intensidades na região R de uma imagem.

Devemos notar que o tamanho da máscara deve sempre ser definido por números ímpares para ter um pixel no centro da máscara. Uma outra observação é que problemas sempre ocorrem nas bordas, dado que não há pixels de um dos lados. Os problemas de **efeitos de borda** ocorrem quando precisamos aplicar o filtro em pixels próximos às bordas da imagem. Neste caso, existem várias soluções paliativas e destacamos duas: ***constant pad*** (onde definimos todos os pixels fora da imagem original como tendo um valor constante, que pode ser 0) e ***clamp*** (onde repetimos os pixels das bordas indefinidamente).

Para a geração do efeito de profundidade de campo (*depth field*) numa imagem, usamos o valor da profundidade (dada pelo *depth buffer*) para calcular um fator de desfocagem e estabelecer o tamanho da máscara. Desta maneira, para a maior distância da câmera na cena, estabelecemos como sendo o fator de desfocagem máximo igual a 1.0 (e, para este valor, arbitramos o maior valor ímpar de máscara *m=n=M*). Então, para cada pixel, lemos a profundidade correspondente e calculamos um tamanho de máscara proporcional. Na prática, geramos 3 matrizes de imagens integrais, uma para cada canal de cor (R, G, B).

Implementação do algoritmo do Summed Area Table:

No algoritmo, devemos alocar uma memória para o SAT e os outros vetores. Para isso, implementamos a função: **int AllocateSATs(GLint sizeX, GLint sizeY);**

Posteriormente escrevemos a função: **void CalculateSummedAreTable(GLsizei sizeX, GLsizei sizeY, GLenum colorBuffer);** que irá calcular a SAT. Essa função lê o *buffer* e os *pixels*, conforme está escrito a seguir.

```
glReadBuffer(colorBuffer);
glReadPixels(0,0,sizeX,sizeY,GL_BGR, GL_FLOAT, SummedAreaTable_BGR);
glReadPixels(0,0,sizeX,sizeY,GL_BGR, GL_FLOAT, transformed_pixels);

glReadBuffer(GL_DEPTH_COMPONENT);
glReadPixels(0,0,sizeX,sizeY,GL_DEPTH_COMPONENT,GL_FLOAT,DepthBuffer);
```

Os *buffer* de cor (***colorBuffer)*** contêm dados de cor RGB e também podem ter valores de alpha. Já os de *profundidade ou buffer de z* (z vem do fato dos valores de X e Y medirem um deslocamento horizontal e vertical à tela e o valor de z ser responsável pela medida das distâncias perpendiculares), armazenam os valores de profundidade de cada pixel, que são geralmente medidos em termos da distância (estabelecendo, assim, uma relação com o olho).

O comando ***glReadBuffer( )*** é usado para selecionar o *buffer* como a fonte para o ***glReadPixels( )***. Dentro dessa mesma função acima, fazemos os laços (*loops*) com suas respectivas condições (*if* e *else*) para a geração de todos os valores do SAT.

Através do algoritmo que segue no quadro abaixo podemos calcular a área inversa e em seguida aplicamos a fórmula para o cálculo do embaçamento (*blur*) da imagem.

```
area_inverse = 1.0f/((highest_i - lowest_i)*(highest_j - lowest_j));

                        (...)

BGR_blur[BLUE] = SummedAreaTable_BGR[lowest_j*sizeX + lowest_i + BLUE] +
SummedAreaTable_BGR[highest_j*sizeX + highest_i + BLUE] -
SummedAreaTable_BGR[highest_j*sizeX + lowest_i + BLUE] -
SummedAreaTable_BGR[lowest_j*sizeX + highest_i + BLUE];


BGR_blur[GREEN] = SummedAreaTable_BGR[lowest_j*sizeX + lowest_i + GREEN] +
SummedAreaTable_BGR[highest_j*sizeX + highest_i + GREEN] -
SummedAreaTable_BGR[highest_j*sizeX + lowest_i + GREEN] -
SummedAreaTable_BGR[lowest_j*sizeX + highest_i + GREEN];


BGR_blur[RED] = SummedAreaTable_BGR[lowest_j*sizeX + lowest_i + RED] +
SummedAreaTable_BGR[highest_j*sizeX + highest_i + RED] -
SummedAreaTable_BGR[highest_j*sizeX + lowest_i + RED] -
SummedAreaTable_BGR[lowest_j*sizeX + highest_i + RED];

BGR_blur[BLUE] *= area_inverse;
BGR_blur[GREEN] *= area_inverse;
BGR_blur[RED] *= area_inverse;
```

Posteriormente escrevemos funções de: inicialização, de renderização das chaleiras e de display (que desenha as chaleiras para o *buffer* de acumulação, ou em inglês *accumulation buffer*, várias vezes, cada qual com uma perspectiva *jitter*). Utilizamos um ponto focal em que z=5.0, de modo que, na Figura 6, possamos ver a segunda chaleira da esquerda para a direita em foco. Escrevemos também funções de *reshape* (remodelamento, que aloca o SAT com relação ao seu comprimento e altura) e a *main* (que chamará todas as funções, abrindo uma

janela com o tamanho inicial, seu respectivo título, modo de exibição RGBA, profundidade de campo e eventos de entrada).

Resultados

Obtivemos através dos algoritmos escritos a Figura 6a que é a cena gerada pelo OpenGL (sem depth field) e a Figura 6b que é o resultado do *depth field*. Obtivemos também as imagens com aplicação do SAT para obter depth field (imagem original na Figura 7a e imagem com o filtro baseado em SAT na Figura 7b).

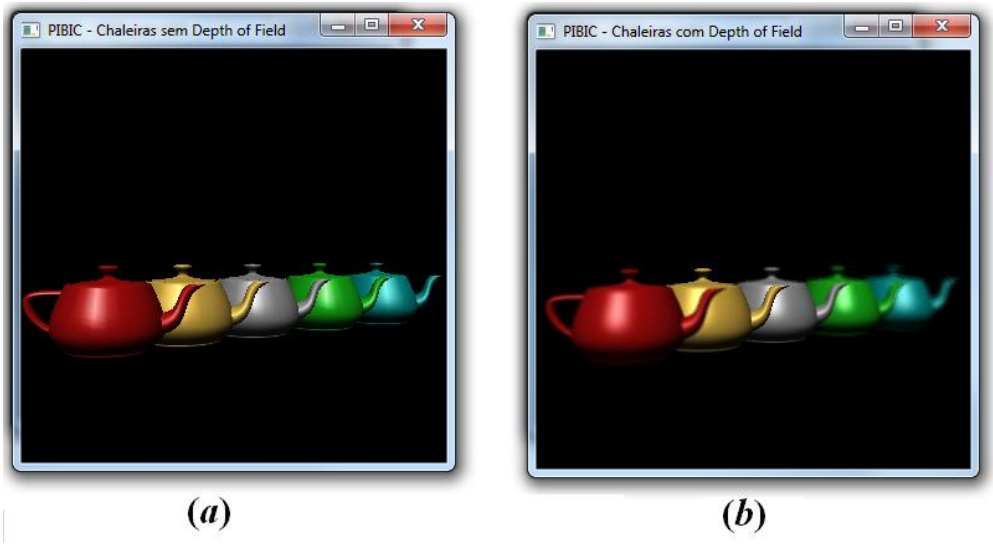

Figura 6: Imagem gerada sem Depth Field em (a) e com Depth Field em (b), utilizando OpenGL.

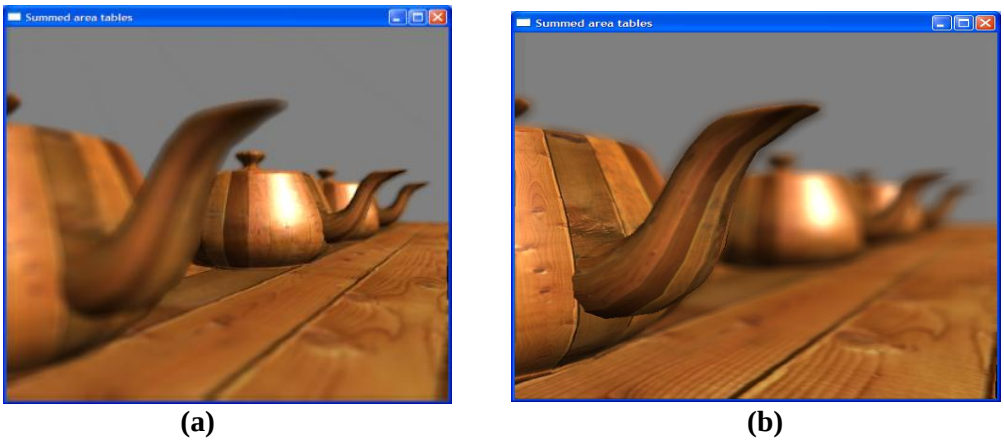

Figura 7: (a) Imagem original. (b) Aplicação do filtro.

Conclusões

O processo investigado revelou que o scan sequencial é lento, mas muito apropriado para ser paralelizado. Observamos, também, que a qualidade da profundidade de foco depende de ajustes no cálculo do tamanho da máscara. Devido a esse fato, na Figura 6b (OpenGL) obtivemos um resultado melhor do que na Figura 7b.

Observamos que gostaríamos de encontrar uma versão paralela do scan que pudesse utilizar os processadores paralelos de uma GPU para acelerar a computação. Através de

estudos [3], vimos que a implementação de um algoritmo **CUDA** com scan paralelo poderia aumentar a rapidez do algoritmo em até 20 vezes com relação a implementações diretas com OpenGL. O algoritmo proposto em [3] pode ser facilmente aplicado através da biblioteca **CUDPP – CUDA Data Parallel Primitives Library** [6] e esta é a nossa primeira indicação para a continuidade do presente trabalho. Destacamos também a técnica de Duplicação Recursiva (*Recursive Doubling*) proposta por Hensley et al. [5]), que utiliza uma alta performance e computação paralela. Por fim, encontramos referências a técnicas ainda mais eficientes, porém protegidas por patentes [9] [10]. Nossa segunda indicação para a continuidade desta pesquisa é estudar estas patentes e procurar melhorá-las.

Referências

1 - WRIGHT Jr, R. S., LIPCHAK, B., HAEMEL, N. **OpenGL SuperBible – Comprehensive Tutorial and Reference**, 4th Ed., Addison-Wesley, 2007.

2 - HILL Jr, F.S., KELLY, S.M. **Computer Graphics Using OpenGL**, 3rd. Ed., Prentice Hall, 2006.

3 - HARRIS, M., SENGUPTA, S., OWENS, J.D. **Parallel Prefix Sum (Scan) with CUDA**. In: H. Nguyen (Ed.), GPU Gems 3, Addison-Wesley, Chapter 39, p. 851-876, 2007.

4 - LANSDALE, R.C. **Texture Mapping and Resampling for Computer Graphics**. MSc Thesis, University of Toronto, Canada, January 1991. [disponível em: http://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.57.4266&rep=rep1&type=pdf] [acessado em 2/07/2011].

5 - HENSLEY, J., SCHEUERMANN, T., COOMBE, G., SINGH, M. & LASTRA, A. **Fast summed-area table generation and its applications**, *Computer Graphics Forum*, 24(3), p. 547-555, 2005.

6 - GPGPU.ORG. **CUDPP – CUDA Data Parallel Primitives Library**. Disponível em http://gpgpu.org/developer/cudpp [acessado em 29/04/2010].

7 – CROW, F. **Summed –area tables for texture mapping**. In: ACM SIGGRAPH 84, p. 207-212, July 1984.

8 – SHREINER, D., WOO, M., NEIDER, J., DAVIS, T. **OpenGL Programming Guide: The Official Guide to Learning OpenGL**, Version 2.1, 6th Edition, Addison-Wesley Professional, 2007 [também em http://glprogramming.com/red, conhecido por The Red Book].

9 - DOTSENKO, Y., GOVINDARAJU, N.K., SLOAN, P-P., BOYD, C. & MANFERDELLI, J. 2008. **Fast scan algorithms on graphics processors**. In: ACM ICS'08, June 7-12, 2008, Grécia, p. 205-213.

10 - DOTSENKO, Y., GOVINDARAJU, N.K., BOYD, C. & MANFERDELLI, J., SLOAN, P-P. **Matrix-based scans on parallel processors**. Patent US 2010/0076941, Assignee: Microsoft, Mar. 25, 2010.