

Programação 3D - Assignment II

Grupo 02

Francisco Campaniço 83463

João Rafael 83482

Rodrigo Oliveira 83558

Março 2019

Uniform Grid Acceleration

Para a grelha uniforme usa-se o algoritmo de Amanatides e Woo (1987). Este algoritmo inicializa a grid com um número de células dado por:

$$N_{cells_{ax}} = \frac{M \cdot dim_{ax}}{\left(\frac{dim_x \cdot dim_y \cdot dim_z}{N_{obj}}\right)^{\frac{1}{3}}} + 1, ax = \{x, y, z\}$$

N_{obj} sendo o número de objetos, dim as dimensões da grelha, e M um fator dado pelo utilizador (usa-se 2 por defeito).

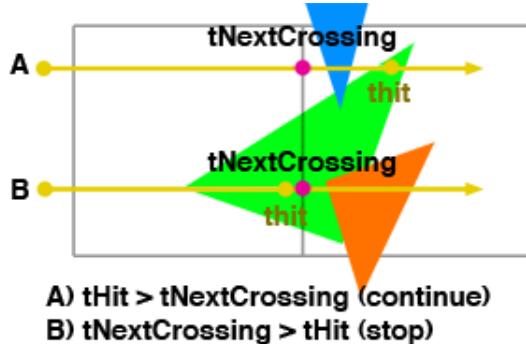
Após a criação das células, verifica-se que células é que contém cada objeto. Para isso, calculamos os índices das células correspondentes à *Bounding Box* do objeto:

$$i_{ax,min/max} = clamp\left(\frac{(obj.bb_{ax} - grid.bb_{ax,min/max}) \cdot N_{cells_{ax}}}{dim_{ax}}, 0, N_{cells_{ax}} - 1\right), ax = \{x, y, z\}$$

e percorremos todos os índices entre i_{min} e i_{max} , preenchendo cada célula com ponteiros para os objetos contidos nela.

Depois desta inicialização, começam-se a disparar raios sobre a grid, usando o algoritmo acima mencionado de *grid traversal*, que consiste em ver a primeira célula onde o raio acerta, e calcular a trajetória do raio através da grid (ou seja, calcula-se os índices *tnext*, *step* e *stop*).

Finalmente, calcula-se para cada objeto na célula a interseção com o raio, e se não houver nenhuma interseção, ou a interseção tiver uma distância superior à distância até à próxima célula (ou seja, $t_{near} > t_{next}$), continua-se o atravessamento para a próxima célula. Este último caso deve-se ao facto de se há interseção mas é mais distante do que a célula atual, não há garantias que na verdade há uma interseção mais próxima, mas numa célula mais distante:



Extra: Mailboxes

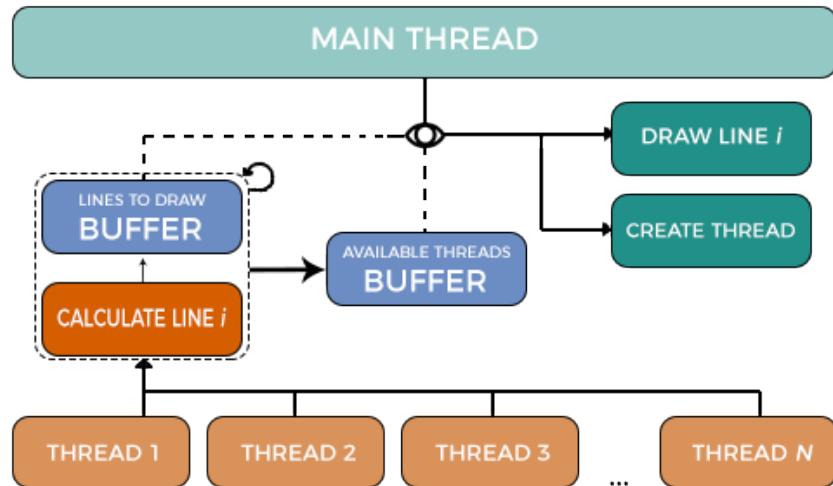
As mailboxes são uma otimização simples. Tem-se um *ID* inteiro único para cada raio. Se esse raio testar a interseção com um objeto e o resultado for falso, adiciona-se ao objeto o *ID*. Se o próximo raio a testar interseção com este objeto for o mesmo (ou seja, *ID* igual), passa-se os cálculos de interseção à frente.

Naturalmente, ter um *ID* único não funciona muito bem com paralelização. Como tal, as *mailboxes* são desligadas quando se tem a paralelização ligada.

Extra: Paralelização

Mesmo com a *Grid* observamos um pequeno uso da CPU em geral e chegamos a conclusão que recorrer à paralelização seria o ideal para tornar o processamento mais rápido. Logo, como em *OpenGL* apenas uma *Thread* pode ser a encarregada por desenhar (limitações de contexto) definimos um algoritmo para realizar a paralelização, no qual criamos *threads* auxiliares que são encarregadas por calcular as cores de cada pixel para um número determinado de linhas. Ao calcular todas as cores de uma linha de pixels a informação é posta num *buffer* (o qual é constantemente lido pela *Main Thread* que desenha a linha em questão).

Apesar de ter melhorado muito a performance notamos que algumas *threads* acabavam muito antes de outras já que as linhas que lhes foram pré-designadas exigiam muito menos cálculos que outras, logo, introduzimos ***Thread Scheduling*** na paralelização de forma a que todas as *threads* que já terminaram as linhas que lhes tinham sido pré-designadas ajudem outras *threads* calculando linhas que ainda não foram calculadas. Desta forma diminuímos **exponencialmente** o tempo de processamento final na maioria dos casos.



Resultados

Foram testadas várias cenas com as diferentes otimizações. O CPU usado foi um Intel i7-8750H com 6 cores e 12 threads.

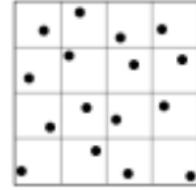
Teste	Sem Grid	Grid	Grid + Mailboxes	Grid + Paralelização
Balls (High)	12m14s	0m15s	0m14s	1s500ms
Mount (Very High)	2h21m3s	0m32s	0m29s	5s500ms
BFBoat (4000 triângulos)	14m45s	0m15s	0m12s	1s700ms
Distant (3 esferas, DoFx32)	0m29s	0m26s	0m25s	5s300ms

Anti-Aliasing

Jittering

É uma técnica de *Stochastic Sampling* que garante que os raios (número fixo) sejam aleatoriamente distribuídos. O pixel é subdividido em *SAMPLES* subpixels e cada subpixel possui uma pequena perturbação tendo em conta a posição na *Regular Grid de Subpixels*. De seguida, a soma das cores obtidas nos subpixels é normalizada. Sendo assim, cada subpixel tem a sua cor calculada da seguinte forma:

```
for each pixel  $(i, j)$  do
     $c = 0$ 
    for  $p = 0$  to  $n - 1$  do
        for  $q = 0$  to  $n - 1$  do
             $c = c + \text{ray-color}(i + (p + \xi)/n, j + (q + \xi)/n)$ 
         $c_{ij} = c/n^2$ 
```

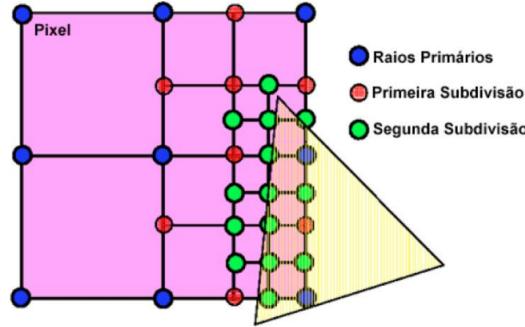


Onde ξ representa um número aleatório uniforme entre 0 e 1.

Monte Carlo Super Sampling

É uma técnica de *Super Sampling* recursiva que tem como vantagem principal o número reduzido de raios disparados. O primeiro passo é disparar quatro raios no pixel (um em cada canto) e de seguida verifica-se se as cores são todas similares entre si através de uma diferença que respeite um *threshold* determinado. Em caso positivo, a cor do pixel é a média dos quatro cantos. Em caso negativo, o pixel é separado em quatro sub-pixels (disparar mais cinco raios, um em cada canto e um ao centro) e aplica-se a mesma técnica em recursão até atingir um limite de sub-pixels determinado ou até atingir finalmente cores similares.

O resultado é filtrado fazendo médias dos resultados em cada passo de recursão nos sub-pixels, ou seja, a cor final do pixel é a média das cores calculadas em recursão.



É notável que a imagem feita com *Jittering* possui um resultado mais agradável e real por causa da *edge sharpness* apresentada, no entanto, a imagem feita com *Monte Carlo Super Sampling* exige menos tempo de processamento.

Soft Shadows

Random

Neste método tratamos *Area Lights* como infinitos pontos de luz e selecionamos um destes pontos de forma aleatória. Para atingir este resultado, apenas provocamos uma perturbação nas componentes *X* e *Y* de cada ponto de luz multiplicando estas por um número aleatório entre 0 e 1.

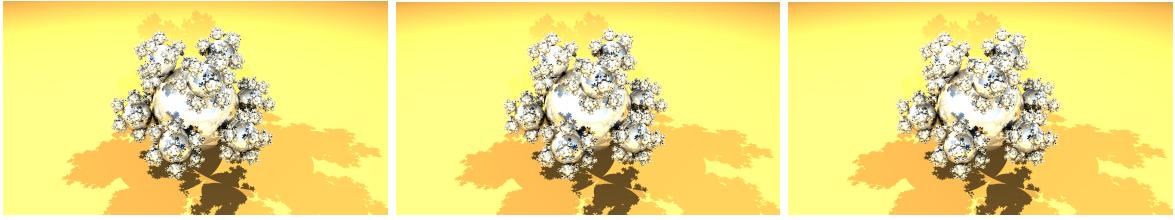


Figure 1: *AA* desligado, *Jittering* com 3 *samples*, *Monte Carlo super sampling*.

Area Light

Este método baseia-se em simular as luzes como áreas em vez de *pointlights*. Para isto, calculam-se "posições alternativas" para cada luz, que consistem de um deslocamento da posição de cada luz dentro de uma área definida previamente, e com um pouco de *jittering*:

$$light_alternate_pos = \{x + \frac{p + rand}{samples * area}, y + \frac{q + rand}{samples * area}, z\}, p, q \in \{0, samples - 1\},$$

rand sendo um valor aleatório entre 0 e 1. Após calcular estas posições, da-se *shuffle* ao vetor que as contém, para que cada pixel *jittered* não teste as sombras com a sua posição correspondente:

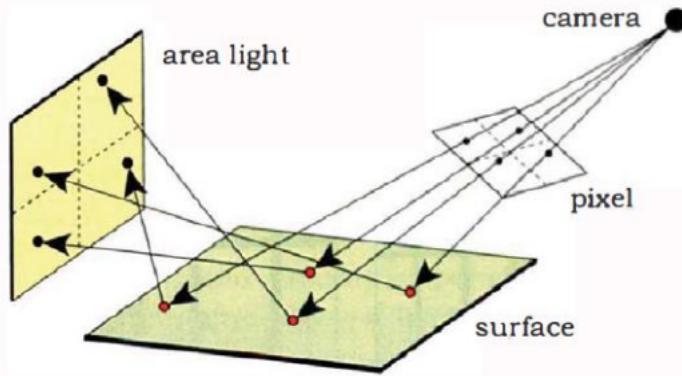


Figure 2: *SS* desligadas, *Random Area light* com a área a 0.25 e 0.5.

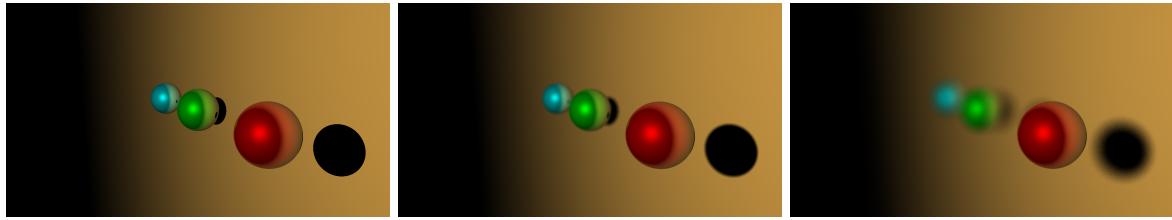


Figure 3: *DoF* desligado, com a abertura a 20, e a 100.

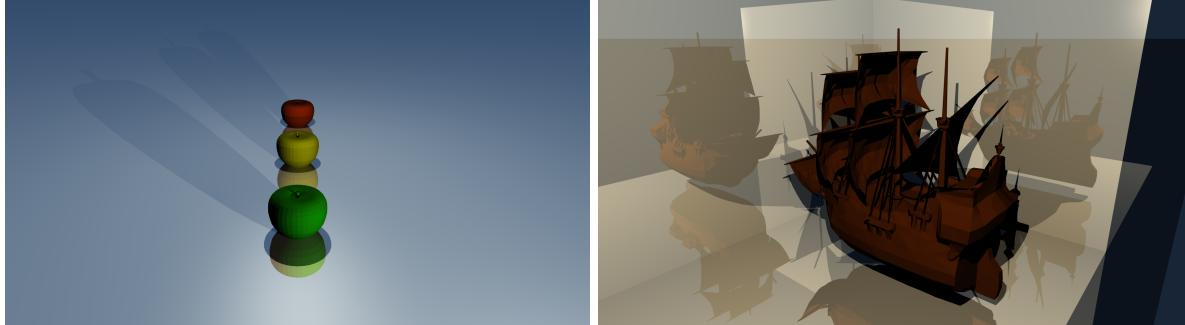


Figure 4: Dois exemplos de cenas com ficheiros PLY.

Depth of Field

Extras

Ficheiros PLY

Glossy Reflections

As *Glossy Reflections* seguem a fórmula de cálculo de uma *Area Light*. Ao invés de obter a cor de uma reflexão a partir de apenas um raio, disparamos mais *SAMPLES* raios com direções levemente alteradas de forma a apontar para uma área em volta da direção original. A cor resultante de uma reflexão é, então, a média das cores calculadas pelos raios com direções levemente alteradas e do raio principal.

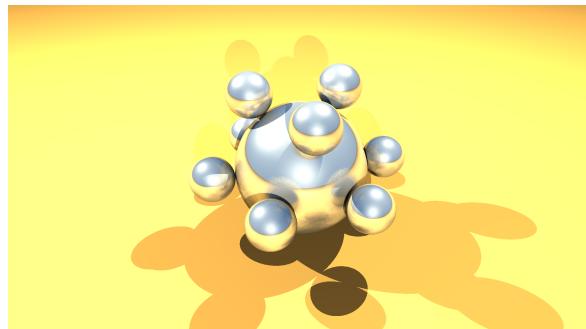


Figure 5: *Glossy Reflections* com 2 samples.