Projeto Visualização e Iluminação

Catarina Costa, Fernando Alves, Marta Aguiar

Mestrado em Engenharia Informática, Universidade do Minho.

Contributing authors: pg52676@alunos.uminho.pt;
pg54470@alunos.uminho.pt; pg52694@alunos.uminho.pt;

Abstract

Este artigo resume a fase final de um projeto onde se aplicaram técnicas avançadas de renderização com o uso de *Parallel Multithreading* e a exportação de imagens em formatos JPG/PFM/OpenEXR, com o objetivo de otimizar a renderização.
Palavras-chave: Computação gráfica, renderização, *flat shading*, *path tracing*, paralelismo, multithreading

1 Introdução

Foi-nos proposta a elaboração de um projeto para a UC **Visualização e Iluminação** do perfil **Computação Gráfica** que consistiu na renderização de uma imagem tridimensional disponibilizada pelo docente, aplicando técnicas e algoritmos de renderização com o objetivo de obter uma imagem similar ao ambiente real. Este artigo é referente exclusivamente à última fase do trabalho, em que foram escolhidos os seguintes temas para serem implementados no projeto: ***Parallel multithreading*** e ***Output JPG/PFM/OpenEXR images***.

2 Métodos

2.1 *Parallel multithreading*

Relativamente à implementação de paralelismo no projeto, este processo foi aplicado no ficheiro ***StandardRenderer.cpp***, pois é onde é feita uma iteração de larga escala que percorre todos os pixels da imagem renderizada e realiza múltiplos cálculos para cada um destes. Desta forma, achamos necessário distribuir a carga de trabalho entre as várias threads, melhorando significativamente o desempenho.

A paralelização foi aplicada através do uso de diretivas OpenMP e o código seguinte é referente à aplicação das mesmas.

```
int numThreads = omp_get_max_threads();
omp_set_num_threads(numThreads);
int chunSize = 5;

#pragma omp parallel shared(W, H, perspCam) num_threads(numThreads)
{
#pragma omp for schedule(dynamic, chunkSize)
    for (int y = 0; y < H; y++) {  // loop over rows
        for (int x = 0; x < W; x++) { // loop over columns
            ...
            for (int ss = 0; ss < spp; ss++) {
                ...
            }
            ...
            #pragma omp critical
            {
                color += this_color;
            }
        }
        ...
    }
}
```

Como é possível observar no código acima apresentado, as variáveis `numThreads` e `chunkSize` são utilizadas juntamente com as diretivas da OpenMP e podem ser ajustadas com o objetivo de melhorar o desempenho do programa, neste caso, reduzir o tempo de renderização da imagem. No caso do número de threads, utilizamos a função `omp_get_max_threads()` para determinar o número máximo de threads suportados pelo sistema, permitindo que o programa se beneficie do maior grau de paralelismo possível dentro das limitações do hardware. Este valor varia conforme a máquina que é utilizada, mas os valores diferem entre as 12 e 16 threads.

Após a realização de vários testes, notamos que os melhores valores a serem utilizados para o número de threads e o valor da quantidade de *chunks* a serem utilizados, seriam, então 12 (através da função `omp_get_max_threads`) e 5, respetivamente. Esta decisão deveu-se ao facto de os valores de tempo de renderização da imagem serem mais rápidos com estes valores. Para estes testes utilizamos o render *PathTracerRender*, com um valor de *samples per pixel* (spp) de 64, e os valores das dimensões da imagem foram de 512x512px.

As diretivas OpenMP utilizadas são explicadas de seguida.

Variáveis Compartilhadas

Variáveis como `W`, `H` e `perspCam` foram declaradas como compartilhadas entre as threads, permitindo acesso simultâneo e coordenado à resolução da imagem e à câmera. Esta partilha entre as threads é possível devido aos valores constantes das variáveis,

ou seja, os valores da altura e da largura da imagem a ser renderizada não se alteram, assim como o objeto referente à câmara de perpetiva do "olho" que aponta para uma determinada direção.

Uso de `#pragma omp parallel`

A diretiva `#pragma omp parallel` cria um ambiente paralelo onde múltiplas threads são utilizadas para processar diferentes partes da imagem simultaneamente. O parâmetro `num_threads(numThreads)` oferece uma flexibilidade para a escolha da quantidade de threads pretendidas para realizar o trabalho de forma paralela, e, como referido anteriormente, o valor escolhido foi `12`.

Divisão do Loop com `#pragma omp for schedule(dynamic, chunkSize)`

O loop que itera sobre os pixels da imagem foi dividido em chunks de tamanho 10 com a diretiva `schedule(dynamic, chunkSize)`, distribuindo a carga de trabalho de forma equilibrada entre as threads. O processo *dynamic* foi escolhido devido à sua capacidade de redistribuir o trabalho de forma dinâmica, permitindo que o OpenMP ajuste a carga de trabalho conforme necessário para maximizar a eficiência, especialmente em situações onde o tempo de execução das iterações varia significativamente. Este é o caso pois os cálculos executados para cad pixel podem variar devido aleatoriedade no cálculo da cor e à soma das cores por pixel, `schedule(dynamic, chunkSize)` parece ser a melhor escolha.

Secção crítica com `#pragma omp critical`

A diretiva `#pragma omp critical` garante que a soma das cores seja feita de forma segura, de forma a evitar *race conditions*. Apenas uma thread por vez pode executar o código dentro da seção crítica, garantindo a integridade dos dados, e neste caso, garantir a soma das cores por pixel de forma correta.

Medição de Tempo com `omp_get_wtime`

No ficheiro `main.cpp`, adaptámos o código para incorporar a paralelização. Ao dividir a carga de trabalho entre várias threads, observámos que o tempo de execução calculado não coincidia com a realidade, devido à forma como era medido. O sistema estava a utilizar uma contagem de *ticks* do relógio do CPU, que representa o tempo de processador consumido pelo programa durante a renderização. Contudo, com a introdução de threads, o uso do CPU intensificou-se, fazendo com que o valor de ciclos aumentasse. Mediante disso, optámos por utilizar a função `omp_get_wtime` para medir o tempo de execução do renderizador num ambiente paralelo. As alterações implementadas estão detalhadas abaixo no ficheiro `main.cpp` em relação à paralelização.

```
start = omp_get_wtime();
myRender.Render();
end = omp_get_wtime();
cpu_time_used = end - start;
```

2.2 *Output JPG/PFM/OpenEXR images*

Nesta secção do relatório iremos explicar como guardamos a imagem em vários formatos nomeadamente JPG, PFM e OpenEXR.

2.2.1 JPG

Para guardar a imagem neste formato, recorremos ao uso do *OpenCV*, que é uma biblioteca utilizada para o desenvolvimento de aplicações na área da Visão por Computador. Esta biblioteca possui vários algoritmos e módulos de processamento de imagem.

Posto isto, adaptámos o código já existente da imagem com o formato `PPM` e alterámos para que a imagem fosse guardada em `JPG`. Criámos então um objeto *Mat* do *OpenCV* com o tipo de imagem *CV_8UC3* para guardar a imagem, uma vez que a imagem tem 3 canais de cor (RGB).

Após isto, percorremos em loop cada pixel na imagem de forma a guardá-los. Como o OpenCV usa o formato **BGR**, tivemos que trocar a ordem dos canais para que ficassem no formato RGB e a imagem fosse guardadas com as cores corretas.

2.2.2 PFM

`PFM`, ou `Portable Float Map`, é um tipo de imagem que guarda valores em ponto flutuante. Tendo isso em conta, ao contrário do formato `PPM`, onde os valores dos píxeis da imagem tinham que ser convertidos para bytes, não fizemos uso de um ToneMap para guardar a imagem.

Primeiro, inicializamos o array de píxeis e procedemos à inicialização do cabeçalho da imagem, de acordo com a estrutura PFM.

```
file << "PF\n";
file << W << " " << H << "\n";
file << "-1.0\n"; // Indica formato little-endian
```

Após isso, iteramos sobre cada pixel, começando pela última linha até à primeira, como forma de garantir que a imagem fosse guardada corretamente. Para cada pixel, foram escritos os valores RGB como floats.

Depois disso, obtemos uma imagem no formato .pfm, que pode ser visualizada utilizando um programa de visualização de imagens PFM online.

2.2.3 OpenEXR

Assim como no formato `JPG`, também utilizamos o OpenCV para auxiliar o processo de guardar a imagem no formato `OpenEXR`.

A forma de guardar a imagem foi bastante semelhante à do formato `JPG`. No entanto, inicializámos um objeto Mat do OpenCV do tipo CV_32FC3 e, em seguida, copiámos os dados dos píxeis para este objeto.

Para verificar se a imagem foi carregada corretamente, implementámos uma função Display para exibir a imagem numa janela, assim que esta fosse corretamente guardada.

3 Resultados e Discussão

Quanto à aplicação das técnicas de paralelização, obtivemos uma melhoria na redução de tempo de renderização da imagem final. Antes da aplicação destas técnicas, para valores do número de amostras por pixels (spp) de 64 e uma dimensão da imagem de 512px por 512px, o tempo de renderização da imagem era de, aproximadamente, 64 segundos. Após a aplicação destas técnicas, reduzimos este tempo para 57 segundos, com uma redução de cerca de **11%**.

Com a aplicação das técnicas de paralelização, decidimos executar o programa, que contém todos os algoritmos implementados necessários para executar uma renderização correta em termos de computação gráfica, como o *Path Tracer*. Ajustamos os valores dos parâmetros determinantes da resolução da imagem e obtivemos a seguinte imagem final. O tempo de renderização foi de **6983** segundos, aproximadamente 116 minutos, e acreditamos que seria mais elevado sem o uso de técnicas de paralelismo.

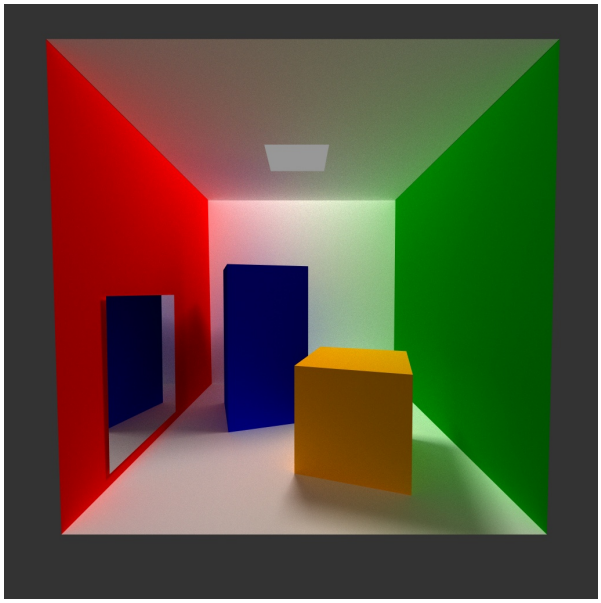

Fig. 1: Imagem renderizada com spp=2048; W=1024; H=1024. Tempo de renderização: 6983 segundos.

Relativamente aos formatos de imagem, o resultado visual de cada formato é idêntico à imagem original do formato PPM, por isso não achamos relevante demonstrar os resultados para esta parte do desenvolvimento do trabalho prático.

4 Conclusão

Neste projeto, implementamos *Parallel Multithreading* e *Output JPG/PFM/OpenEXR images*.

Aplicámos a paralelização utililizando OpenMP para distribuir a carga de trabalho da renderização entre múltiplas threads. O código foi modificado para usar diretivas `#pragma omp`, permitindo a execução simultânea de várias partes do processo de renderização. Esta abordagem resultou numa redução significativa do tempo de renderização.

Quanto à formatação das imagens, implementamos a saída de imagem nos formatos `JPG`, `PFM` e `OpenEXR`. Para o formato `JPG`, recorremos à biblioteca OpenCV para guardar a imagem com compressão adequada. O formato `PFM` foi usado para armazenar valores em pontos flutuante. E por último, mas não menos importante, para o formato `OpenEXR`, também recorremos à biblioteca OpenCV, sendo a sua forma de guardar semelhante à do JPG.

De forma a otimizar este projeto, seria ideial, para um trabalho futuro, aplicar mais diversidade e melhoramentos ao mesmo, como os temas que nos foram apresentados, tais como: estrutura de acelaração, amostragem de muitas fontes de luz, *tone mapping*, melhoria de BRDFs, luzes ambientais, camaras alternativas, mapeamento de texturas nas meshes, saída interativa da janela e rastreamento de caminho progressivo.

5 Referências

1. Acetatos da UC
2. "Physically Based Rendering: from Theory to Implementation"; Matt Pharr and Greg Humphreys; Morgan Kaufmann; 3rd Edition; 2014 Disponível em: http://www.pbr-book.org/
3. docs.opencv.org. (n.d.). OpenCV: OpenCV modules. [online] Disponível em: https://docs.opencv.org/4.10.0/index.html
4. sobral, bosco (n.d.). OpenMP Scheduling. [online] Disponível em: https://www.inf.ufsc.br/~bosco.sobral/ensino/ine5645/OpenMP_Dynamic_Scheduling.pdf
5. www.ibm.com. (n.d.). #pragma omp critical. [online] Disponível em: https://www.ibm.com/docs/en/zos/2.4.0?topic=processing-pragma-omp-critical