Paralelização e Otimização de um Ray Tracer

António Gonçalves
Universidade do Minho
a85516@alunos.uminho.pt

Gonçalo Esteves
Universidade do Minho
a85731@alunos.uminho.pt

Abstract

O principal objetivo deste trabalho passa pela otimização de uma implementação previamente elaborada de um *ray tracer*, recorrendo para isso a *threads* e *tasks* de modo a paralelizar o código, bem como à construção de estruturas auxiliares, como *Bounded Volume Hierarchies*, que nos permitem organizar de uma melhor forma os triângulos que compõe uma imagem.

I. Introdução

Um *ray tracer* ou algoritmo de *ray tracing* é uma técnica de renderização utilizada para gerar imagens com grande qualidade, recorrendo para isso à simulação do caminho efetuado pelos raios de luz e à forma como estes reagem ao interagirem com objetos.

Tendo em vista o desenvolvimento do trabalho proposto, aplicamos diferentes técnicas com o objetivo de efetuar otimizações ao código fornecido. Estas não estão apenas relacionadas com a paralelização de segmentos do programa, mas também com a reestruturação e reorganização de dados, por forma a permitir um acesso mais rápido aos mesmos.

O código produzido foi testado nas duas máquinas do grupo: uma com um processador Intel Core i7-8750H, que possui 6 *cores* e 12 *threads*, e a outra com um um Intel Core i5-6200U, que tem 2 *cores* e 4 *threads*. Enquanto que a primeira correu todos os testes, a segunda apenas correu os mais leves, uma vez que o seu menor poder computacional não permitia correr os testes mais pesados em tempo útil. Apesar disto, achamos interessante comparar, para os testes de menor exigência, os resultados obtidos em ambas, sempre que estes aparentavam levar-nos a conclusões diferentes. Deste modo, seremos também capazes de retirar algumas conclusões relativamente às melhorias de desempenho obtidas em máquinas com características significativamente diferentes.

II. Análise do Código Inicial

Por forma a analisar o desempenho do programa fornecido inicialmente, tivemos de definir uma série de testes, que viriam a ser replicados sempre que se procede-se a uma alteração no código.

Deste modo, optamos por realizar testes com os modelos guardados nos ficheiros *CornellBox-Empty-CO.obj*, *CornellBox-Original.obj*, *CornellBox-Sphere.obj* e *CornellBox-Water.obj*, uma vez que todos eles apresentam diferente número de triângulos (12, 36, 2188 e 7088, respetivamente). Para a máquina com o processador i5, apenas foram feitos testes sobre os dois primeiros modelos.

Para além disto, e por forma a diversificar mais os nossos testes, optamos por utilizar dois pontos de vista diferentes. Em ambos os casos, a imagem gerada irá estar direcionada para o ponto (0, 1, 0), no entanto, utilizamos dois pontos de observação diferentes, um em (0, 1, 3) e outro em (2, 1, 3). Como iremos verificar em seguida, o uso destes dois pontos fará com que se possam retirar algumas conclusões sobre a execução do programa em si.

Cada teste consistiu na execução do programa com as características especificadas 10 vezes, fazendo-se uma média dos resultados obtidos.

Estes foram os resultados obtidos:

	12	**36**	**2188**	**7088**
(0,1,3)	237,50	486,70	22010	77410
(2,1,3)	176,90	373,70	19159	65980

Table 1: *Tempos de execução (em ms) obtidos com diferentes pontos de vista, tendo em conta o número de triângulos (utilização do processador i7)*

	12	**36**
(0,1,3)	5964,0	14185
(2,1,3)	4586,0	11074

Table 2: *Tempos de execução (em ms) obtidos com diferentes pontos de vista, tendo em conta o número de triângulos (utilização do processador i5)*

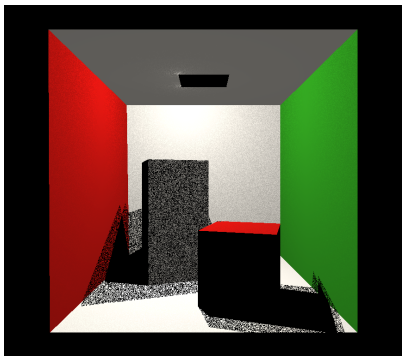

Figure 1: *Imagem obtida com o ficheiro CornellBox-Original.obj através do ponto de observação (0, 1, 3)*

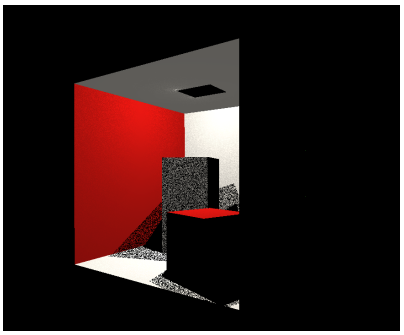

Figure 2: *Imagem obtida com o ficheiro CornellBox-Original.obj através do ponto de observação (2, 1, 3)*

Analisando os tempos de execução obtidos, torna-se fácil constatar que, tal como seria esperado, o aumento do número de triângulos implica um aumento do tempo de execução. No entanto, outra conclusão que também se pode retirar é que diferentes pontos de vista poderão implicar diferentes tempos de execução. Assim sendo, pensamos ter encontrado duas possíveis explicações para este fenómeno:

- conforme a distância entre o ponto de observação e o modelo aumenta, o tempo de execução diminui. Isto deve-se ao facto de que com o aumento da distância, o número de pixeis que intersetam o modelo diminui, e uma diminuição no número de interseções leva a menores tempos de execução (para comprovar esta teoria, efetuamos testes com pontos de observação em (0, 1, 3) e (0, 1, 4), sendo que optamos por não apresentar resultados, uma vez que consideramos que este não seja o objetivo principal do trabalho);

- para pontos de observação diferentes a uma distância semelhante, os tempos de execução poderão também variar, uma vez que apesar da distância ser a mesma, os pontos de vista deverão ser diferentes, o que pode levar a variações no número de intereseções e oclusões que ocorrem (para comprovar esta teoria, efetuamos testes com pontos de observação em (0, 1, 3) e (0, 4, 0)).

III. Estratégias Implementadas e Considerações

i. Paralelização do *tracer* recorrendo a *Tasks*

De modo a efetuar uma primeira paralelização do código, optamos, nesta fase, por uma abordagem relativamente simples. Utilizando *std::async*, inicialmente criamos uma *task* para cada pixel a ser processado. No entanto, rapidamente percebemos que esta não seria a melhor abordagem, uma vez que o número de *tasks* criado seria excessivamente grande, sendo que a máquina com menor poder computacional não era sequer capaz de correr o programa.

Posto isto, adotamos outra abordagem, que acabou por se tornar a abordagem final: criamos N *tasks*, onde N é a altura da imagem, de modo a que cada *task* seja responsável por processar uma linha de pixeis da imagem. A função responsável por esta execução paralela chama-se *executeTasks* e encontra-se no ficheiro *main.cpp*. Com isto, fomos capazes de obter os resultados que se seguem.

	12	**36**	**2188**	**7088**
(0,1,3)	5,2313	5,7057	6,3181	6,2061
(2,1,3)	4,7554	5,3693	6,1958	6,1236

Table 3: *Speedups obtidos com a paralelização do código, utilizando diferentes pontos de vista e tendo em conta o número de triângulos (utilização do processador i7)*

	12	**36**
(0,1,3)	2,2421	2,2363
(2,1,3)	2,2262	2,2201

Table 4: *Speedups obtidos com a paralelização do código, utilizando diferentes pontos de vista e tendo em conta o número de triângulos (utilização do processador i5)*

Interpretando os resultados obtidos, podemos observar que o aumento do número de triângulos leva a um ligeiro aumento do *speedup*, isto é, quanto maior a quantidade de triângulos que a imagem tem, mais benéfica será a paralelização. Uma justificação para tal poderá passar pelo facto de que com o aumento do número de triângulos, também aumenta a carga de trabalho de cada *task*, uma vez que haverão mais triângulos para verificar a interseção e a oclusão. Assim, o custo adicional da criação da *task* fica "mais mascarado" na melhoria de desempenho que estas trazem, já que há mais trabalho a ser efetuado em paralelo.

Além disto, podemos também verificar que os *speedups* obtidos em ambas as máquinas são aproximadamente iguais ao número de *cores* das mesmas. Este comportamento é o esperado tendo em conta o tipo de paralelização que foi efetuado, já que um aumento do número de *cores* deverá, em teoria, levar a uma diminuição do tempo de execução na mesma proporção. Assim, podemos concluir que a paralelização do código efetuada estará bem otimizada, já que os resultados obtidos são os esperados.

ii. Implementação de uma *Bounded Volume Hierarchy*

Após a adaptação do código para uma primeira versão paralela, viramo-nos para uma melhor estruturação dos dados. Tendo em conta a versão original do código que nos foi fornecida, sempre que se quer verificar se um raio interseta algum triângulo, devemos percorrer todos os triângulos, por forma a verificar se algum é intersetado, e em caso afirmativo, garantir que o valor retornado no fim corresponde ao do triângulo que se encontra mais longe e é intersetável. No caso das oclusões, também se irá percorrer todos os triângulos sempre que se verifique que não ocorre oclusão.

Assim sendo, implementamos uma *Bounded Volume Hierarchy* (BVH) por forma a guardar os triângulos mais organizadamente, permitindo que as travessias efetuadas sejam mais rápidas.

A BVH é construída seguindo o seguinte padrão:

- **1 -** percorrem-se os triângulos recebidos, de modo a determinar quais são os pontos mínimo e máximo que delimitam a zona onde estes se encontram. Para isto, verifica-se quais os x, y e z mínimos e máximos. Caso haja mais do que um

triângulo, passar para o ponto 2, caso contrário passar para o ponto 4;

- **2 -** os triângulos são ordenados segundo uma das coordenadas (x, y ou z). Passamos para o ponto 3;

- **3 -** os triângulos são divididos em duas partes, sendo cada parte passada a um dos nós-filho criados. Para além disso, também se passa a indicação de qual a coordenada a utilizar para ordenar os nós-filho (esta deverá ser diferente da utilizada pelo nó-pai). Cada nó-filho começa no ponto 1;

- **4 -** caso haja apenas um triângulo no nó, então encontramo-nos numa folha da BVH. Deveremos então simplesmente guardar o triângulo que recebemos.

Convém realçar que na divisão dos triângulos pelos nós-filho, sempre que estes se encontram em número ímpar, o nó-filho da esquerda irá receber mais um triângulo do que o nó-filho da direita. Optamos por esta abordagem ao invés da replicação dos triângulos uma vez que, assim, conseguimos obter melhores tempos de execução sem comprometer a qualidade da imagem final, já que as *boxes* da BVH se sobrepõe de qualquer maneira. Os tempos de execução são deteriorados com a replicação dos triângulos uma vez que se observa um aumento substancial do número dos mesmos quando se utiliza a última técnica referida (para o *model* com 36 triângulos, com replicação, a BVH gera 64 triângulos). O construtor da BVH encontra-se no ficheiro *bvh.h*.

No que toca à interseção, esta fica mais rápida se a implementarmos recorrendo à BVH, de forma semelhante ao que se observa na função *intersectBVH*, presente no ficheiro *main.cpp*. O algoritmo da mesma funciona da seguinte maneira:

- **1 -** Caso o nodo seja uma *box* passar para o ponto 2, caso seja uma folha, passar para o ponto 5;

- **2 -** Se o raio intersetar a *box*, passar para o ponto 3, caso contrário, indicar que o raio não interseta nenhum triângulo;

- **3 -** Determinar a distância entre a origem do raio e o ponto de interseção com o nó-filho da esquerda e do nó-filho da direita. Caso não intersete nenhum dos nós-filho, indicar que o raio não interseta nenhum triângulo; caso intersete apenas um dos nós filho, retornar ao ponto 1 com o nó-filho em questão; caso intersete os dois nós-filho, passar para o ponto 4;

- **4 -** Verificar qual o nó-filho que interseta a uma distância menor e retornar ao ponto 1 com esse nó-filho. Em seguida, fazer o mesmo para o outro nó-filho. Por fim, verificar se algum triângulo foi intersetado e, em caso afirmativo, indicar qual o último que foi; caso contrário, indicar que não houve interseção com nenhum triângulo;

- **5 -** No caso do nó ser uma folha, verificar se o raio interseta o triângulo guardado na folha. Em caso afirmativo, indicar qual o triângulo, caso contrário, indicar que não houve interseção.

Relativamente à oclusão, o algoritmo desta é semelhante ao da interseção, e encontra-se representado na função *occlusionBVH*:

- **1 -** Caso o nodo seja uma *box* passar para o ponto 2, caso seja uma folha, passar para o ponto 5;

- **2 -** Se o raio intersetar a *box*, passar para o ponto 3, caso contrário, indicar que o raio não interseta nenhum triângulo;

- **3 -** Verificar se o raio interseta as *boxes* do nó-filho da esquerda e do nó-filho da direita. Caso não intersete nenhum dos nós filho, indicar que o raio não interseta nenhum triângulo; caso intersete apenas um dos nós filho, retornar ao ponto 1 com o nó-filho em questão; caso intersete os dois nós-filho, passar para o ponto 4;

- **4 -** Retornar ao ponto 1 o nó-filho da esquerda. Caso o raio intersete algum triângulo deste, indicar isso mesmo, caso contrário, retornar ao ponto 1 com o nó-filho da direita;

- **5 -** No caso do nó ser uma folha, verificar se o raio interseta o triângulo guardado na folha e retornar o resultado.

Posto tudo isto, os resultados obtidos estão presentes nas seguintes tabelas:

	12	**36**	**2188**	**7088**
(0,1,3)	0,72585	1,0244	11,944	12,663
(2,1,3)	0,96878	1,4296	18,795	17,979

Table 5: *Speedups obtidos recorrendo à BVH, utilizando diferentes pontos de vista e tendo em conta o número de triângulos (utilização do processador i7)*

	12	**36**
(0,1,3)	1,1315	1,8858
(2,1,3)	1,5616	2,6775

Table 6: *Speedups obtidos recorrendo à BVH, utilizando diferentes pontos de vista e tendo em conta o número de triângulos (utilização do processador i5)*

Desta vez, os resultados obtidos nas diferentes máquinas indicam-nos diferentes coisas:

- Segundo os resultados obtidos na máquina de maior poder computacional, a geração e uso da BVH não são benéficas para os modelos com menor número de triângulos, sendo que o *speedup* obtido é muito baixo ou até mesmo um "*speed-down*". No entanto, para *models* com um maior número de triângulos, os ganhos que o uso da BVH fornece são imensos.

- Recorrendo aos resultados obtidos na máquina de menor poder computacional, o uso da BVH compensa sempre, apesar de não ser tão notório no caso do modelo com 12 triângulos.

Tendo em conta estas observações, podemos concluir que o poder computacional da máquina utilizada influencia a obtenção de ganhos ou não com a utilização da BVH (no caso da paralelização, a capacidade da máquina apenas influenciava a quantidade de ganhos obtidos). No caso da máquina com o processador i7, como esta tem maior poder computacional supõe-se que seja capaz de percorrer todos os triângulos (seguindo a versão original do código) de forma mais rápida do que a outra máquina. Assim, a utilização de uma BVH para *models* com poucos triângulos não irá compensar, já que o tempo dispendido a construir a BVH irá ser superior ao ganho com os cálculos das interseções e oclusões. Por outro lado, a máquina com o processador i5, uma vez que demora mais tempo a percorrer todos os triângulos com a versão original, já consegue aproveitar melhor a utilização da BVH, mitigando os custos da construção da mesma com ganhos suficientes nas interseções e construções, mesmo em modelos com poucos triângulos.

No entanto, ambos os conjuntos de dados permitem retirar duas conclusões importantes.

Primeiro, que o aumento do número de triângulos leva a um aumento nos ganhos obtidos. Isto deve-se ao facto de que apesar do tempo de construção da BVH aumentar cada vez mais, os ganhos obtidos nos cálculos de interseções e oclusões também compensam cada vez mais. Tal afirmação pode ser melhor compreendida quando analisamos assintoticamente o comportamento esperado das funções de interseção e oclusão.

Na versão original, é esperado que o tempo de execução das funções seja aproximadamente $O(N)$, sendo N o número de triângulos. No entanto, quando recorremos ao uso de uma BVH, o tempo de execução esperado passa a ser aproximadamente $O(log(N))$ em ambos os casos. Assim, é fácil concluir que o aumento do número de triângulos leva a um aumento diretamente proporcional do tempo de execução das funções na versão original, enquanto que utilizando a BVH, o tempo de execução aumenta de forma logarítmica, que traduz um crescimento cada vez menos acentuado para um número maior de triângulos.

Outra conclusão a retirar é que os pontos de observação utilizados também influenciam os ganhos obtidos. Quando comparamos os ganhos obtidos com o ponto (0, 1, 3) e o ponto (2, 1, 3) verificamos que obtemos maiores *speedups* para o segundo ponto. Uma possível justificação para este fenómeno poderá ser o menor número de intereseções que se obtêm utilizando o ponto (2, 1, 3). Na versão original, para determinar que não ocorre uma interseção deve-se percorrer na mesma todos os triângulos. Na versão com BVH, basta determinar que o raio não interseta as *boxes* onde os triângulos estão guardados. Isto faz com

que a determinação de uma não interseção seja extremamente mais eficiente, o que leva a uma redução significativa do tempo de execução, permitindo obter maiores ganhos consoante o número de interseções diminui.

Por fim, implementamos uma construção paralela da BVH, recorrendo para isso a *threads*. Tendo em conta a construção sequencial implementada, a principal alteração efetuada foi permitir que as chamadas recursivas da função, utilizadas para criar os dois nós-filho, sejam efetuadas por *threads* diferentes. Para além disto, garantimos que o número de *threads* criadas não ultrapassa o número de *threads* disponibilizadas pelo processador. Esta implementação encontra-se no ficheiro *bvh_par_construction.h*. Seguem-se os resultados obtidos:

	12	**36**	**2188**	**7088**
(0,1,3)	0,13540	0,17462	1,8256	1,796359
(2,1,3)	0,19925	0,25912	2,9047	2,3884

Table 7: *Speedups obtidos recorrendo à BVH construída paralelamente, utilizando diferentes pontos de vista e tendo em conta o número de triângulos (utilização do processador i7)*

	12	**36**
(0,1,3)	1,1287	1,8899
(2,1,3)	1,5738	2,6911

Table 8: *Speedups obtidos recorrendo à BVH construída paralelamente, utilizando diferentes pontos de vista e tendo em conta o número de triângulos (utilização do processador i5)*

Se utilizando a máquina com menor poder computacional os *speedups* obtidos são aproximadamente os mesmos, isso não se reflete quando utilizamos a outra máquina. Utilizando a máquina com processador i7, os ganhos obtidos descem imenso, sendo que para as versões com menor número de triângulos a construção da BVH de forma paralela é extremamente prejudicial.

Tendo por base estes resultados, podemos confirmar dois factos: primeiro, que nem sempre paralelizar o código o torna automaticamente mais eficiente, já que se deve ter em consideração outros aspetos, como os custos da replicação de recursos ou a criação de *threads*, bem como o facto de que a carga de trabalho poderá não ser grande o suficiente para compensar todo o trabalho (pensamos ser este o caso); segundo, que a paralelização de chamadas recursivas de funções é algo bastante complexo e de difícil execução.

iii. Implementação de uma *Locked Queue*

A última otimização efetuada foi a criação de uma *Locked Queue*, e a sua utilização na construção paralela de uma imagem. A implementação da *queue* pode ser observada no ficheiro *locked_queue.h* e esta é utilizada na função *executeLockedQueue*, no ficheiro *main.cpp*.

Uma *locked queue* é, na sua essência, uma espécie de vetor que controla os acessos que lhe são feitos, por forma a garantir que este não é acedido nem alterado por dois processos diferentes em simultâneo. Assim, permite lidar com problemas de concorrência de forma eficaz, caso tenha sido bem implementada, garantindo que não há acessos concorrentes à *queue* nem tempos de espera demasiado longos por parte dos processos, de modo a acederem aos dados.

A função *executeLockedQueue* tira partido de tudo o que foi referido até agora. Inicialmente, divide os pixeis em 256 blocos, de tamanho aproximadamente igual, e guarda as suas coordenadas num vetor, isto é, as alturas e larguras máximas e mínimas. Em seguida, cria uma *locked queue* com esta informação. Por fim, cria N *threads*, onde N é o número de *threads* disponibilizadas pela máquina (no caso do processador i7 são 12 *threads*, por exemplo). Cada *thread* é responsável por aceder à *queue* criada, enquanto esta tiver elementos inseridos, e retirar de lá um elemento, ou seja, as coordenadas de um bloco de pixeis, e processá-lo. Deste modo, as *threads* devem esvaziar a *queue* e processar os seus elementos, e a *queue* deve garantir que não distribui elementos repetidos e que todos são distribuídos.

Os *speedups* obtidos com esta otimização são apresentados em seguida.

	12	**36**	**2188**	**7088**
(0,1,3)	5,9974	6,4293	6,1037	5,9428
(2,1,3)	6,2953	6,2806	6,4438	5,7938

Table 9: *Speedups obtidos recorrendo à paralelização com uma locked queue, utilizando diferentes pontos de vista e tendo em conta o número de triângulos (utilização do processador i7)*

	12	**36**
(0,1,3)	2,2354	2,2452
(2,1,3)	2,2502	2,2300

Table 10: *Speedups obtidos recorrendo à paralelização com uma locked queue, utilizando diferentes pontos de vista e tendo em conta o número de triângulos (utilização do processador i5)*

De um modo geral, podemos afirmar que se continua a observar o comportamento já descrito na implementação paralela criada inicialmente: o valor do *speedup* obtido é aproximadamente igual ao número de *cores* físicos da máquina. Quando comparamos os valores de *speedup* obtidos nesta versão, e os valores obtidos com a versão paralela inicial, podemos afirmar que eles são praticamente iguais no que toca à máquina de menor poder computacional, no entanto, se olharmos para a outra máquina, verificamos que os *speedups* obtidos com esta versão são superiores para os dois *models* com menor número de triângulos, e inferiores para os *models* com maior número de triângulos. Para além disto, verificamos que esta versão providencia *speedups* mais constantes, uma vez que os valores são ainda mais próximos daquilo que se esperava.

Assim, podemos concluir que a implementação de uma *locked queue* e a restrição do número de *threads* utilizadas podem trazer alguns benefícios para a execução do programa paralelo, permitindo a obtenção de *speedups* mais constantes e até mesmo maiores quando o número de triângulos é inferior, já que, como são criados menos processos, o custo adicional é inferior e não se nota tanto em situações onde o custo total do programa é menor.

iv. Versão Final

Após a realização de todo o estudo descrito previamente, passamos à fase de construção da implementação final. Assim sendo, optamos por duas abordagens diferentes, uma utilizando a versão paralela inicial com recurso a uma BVH para guardar a informação sobre os triângulos, e a outra recorrendo à implementação com *locked queue* e utilizando também a BVH. Em ambos os casos, a construção da BVH foi feita de forma sequencial, uma vez que a construção paralela não compensa. Os resultados são apresentados em seguida.

	12	**36**	**2188**	**7088**
(0,1,3)	3,6149	5,3838	55,330	45,498
(2,1,3)	4,1332	6,7699	72,395	49,617

Table 11: *Speedups obtidos com a paralelização do código e recurso a uma BVH, utilizando diferentes pontos de vista e tendo em conta o número de triângulos (utilização do processador i7)*

	12	**36**
(0,1,3)	2,3610	3,9305
(2,1,3)	3,2319	5,5287

Table 12: *Speedups obtidos com a paralelização do código e recurso a uma BVH, utilizando diferentes pontos de vista e tendo em conta o número de triângulos (utilização do processador i5)*

	12	**36**	**2188**	**7088**
(0,1,3)	4,2870	5,9864	55,261	44,622
(2,1,3)	5,4938	8,1952	67,544	48,961

Table 13: *Speedups obtidos recorrendo à paralelização com uma locked queue e usando uma BVH, utilizando diferentes pontos de vista e tendo em conta o número de triângulos (utilização do processador i7)*

	12	36
(0,1,3)	2,3770	3,9667
(2,1,3)	3,3088	5,6616

Table 14: *Speedups obtidos recorrendo à paralelização com uma locked queue e usando uma BVH, utilizando diferentes pontos de vista e tendo em conta o número de triângulos (utilização do processador i5)*

Quando comparamos os *speedups* obtidos, podemos confirmar a persistência das características que foram descritas ao longo deste estudo:

- um aumento no número de triângulos leva a um aumento no *speedup* obtido, tornando o programa mais eficiente para maiores quantidades de triângulos (no entanto, deve-se fazer uma ressalva, pois como é visível, o *speedup* obtido para 2188 triângulos é superior ao obtido para 7088 triângulos, o que nos poderá indicar que, quando se utiliza as duas técnicas simultaneamente - paralelização do código e construção da BVH - atingimos um ponto de inflexão, onde os *speedups* deixam de aumentar para quantidades superiores de triângulos);
- diferentes pontos de observação podem implicar diferentes *speedups* finais obtidos;
- quando recorremos ao uso de BVH, pontos de observação que potenciem menos interseções irão ter melhores *speedups* (o ponto (2, 1, 3) origina menos interseções do que o ponto (0, 1, 3));
- o uso da implementação com *locked queue* é ligeiramente melhor para modelos com menos triângulos, enquanto que o contrário se verifica para modelos com mais triângulos;
- no caso da máquina com maior poder de computação, a construção da BVH prejudica os tempos de execução para modelos com poucos triângulos (para os *models* com 12 e 36 triângulos, os *speedups* obtidos foram inferiores aos obtidos com as implementações paralelas sem BVH);

IV. Utilização de *Ambient Occlusion*

Por fim, implementamos *Ambient Occlusion* no nosso *ray tracer*. Com esta funcionalidade fomos capazes de obter imagens com melhor qualidade, especialmente no que toca a sombras e concentração de cores.

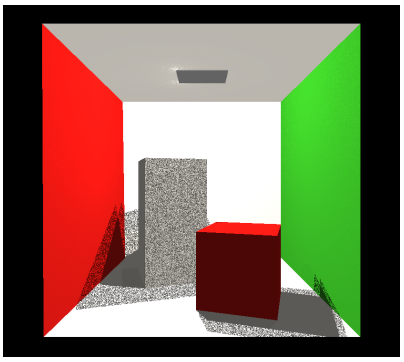

Figure 3: *Imagem com oclusão obtida com o ficheiro CornellBox-Original.obj através do ponto de observação (0, 1, 3)*

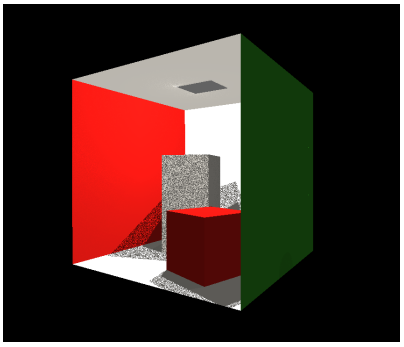

Figure 4: *Imagem com oclusão obtida com o ficheiro CornellBox-Original.obj através do ponto de observação (2, 1, 3)*

Por forma a implementar esta melhoria na imagem, começamos por criar 14 vetores diferentes, todos de comprimento máximo igual a uma unidade (os vetores correspondem aos 6 vetores unitários, contando com coordenadas positivas e negativas, e aos 8 vetores das

diagonais). Optamos por esta abordagem, ao invés da geração aleatória de vetores unitários, uma vez que a qualidade das imagens finais não é prejudicada e os tempos de execução são consideravelmente menores, já que a geração de números aleatórios é um cálculo custoso, ainda mais quando realizada em paralelo. Além disso, a geração aleatória iria tornar o programa mais instável, já que o tempo de execução de cada *thread* poderia variar imenso, consoante a velocidade com que descobrisse um vetor que cumprisse os requisitos (bastaria uma *thread* demorar mais a descobrir os vetores admissíveis para atrasar todo o programa, mesmo que as restantes descobrissem sempre os vetores à primeira).

Após a geração dos vetores, e para cada um deles, calcula-se se ocorre a oclusão, e em caso afirmativo, vai-se aumentando gradualmente a concentração de cor no pixel em questão.

De realçar que todo este processo apenas é realizado caso ocorra uma interseção, estando "ao mesmo nível" que o cálculo da oclusão com os diferentes raios de luz.

São em seguida apresentados os diferentes resultados obtidos, utilizando as técnicas referidas ao longo do estudo (apenas foi utilizada máquina com maior poder computacional, devido ao grande esforço exigido por estas implementações).

	12	**36**	**2188**	**7088**
(0,1,3)	1130,6	2721,7	124299	444655
(2,1,3)	692,5	1672,1	77701	277548

Table 15: *Tempos de execução (em ms) obtidos com diferentes pontos de vista, tendo em conta o número de triângulos*

	12	**32**	**2188**	**7088**
(0,1,3)	4,5278	5,5727	5,3693	5,4757
(2,1,3)	4,5593	5,2965	5,4314	5,4112

Table 16: *Speedups obtidos com a paralelização do código, utilizando diferentes pontos de vista e tendo em conta o número de triângulos*

	12	**32**	**2188**	**7088**
(0,1,3)	0,72905	0,88591	6,2933	6,2673
(2,1,3)	0,81289	1,0041	4,8794	5,7071

Table 17: *Speedups obtidos recorrendo à BVH, utilizando diferentes pontos de vista e tendo em conta o número de triângulos*

	12	**32**	**2188**	**7088**
(0,1,3)	6,1379	6,1773	5,3585	5,2385
(2,1,3)	6,0692	6,1452	5,2786	5,3572

Table 18: *Speedups obtidos recorrendo à paralelização com uma locked queue, utilizando diferentes pontos de vista e tendo em conta o número de triângulos*

	12	**32**	**2188**	**7088**
(0,1,3)	3,3769	4,5951	35,753	33,540
(2,1,3)	3,3357	5,1961	30,279	29,016

Table 19: *Speedups obtidos com a paralelização do código e recurso a uma BVH, utilizando diferentes pontos de vista e tendo em conta o número de triângulos*

	12	**32**	**2188**	**7088**
(0,1,3)	4,3021	5,2000	41,763	36,37676
(2,1,3)	4,8191	5,8547	32,774	37,047

Table 20: *Speedups obtidos recorrendo à paralelização com uma locked queue e usando uma BVH, utilizando diferentes pontos de vista e tendo em conta o número de triângulos*

Os resultados obtidos são semelhantes à versão sem *Ambient Occlusion*, não ao nível de tempos de execução, mas sim no que toca aos *speedups* obtidos. Estes diminuem um pouco, muito provavelmente consequência do aumento da carga computacional, mas na maior parte dos casos não são quedas muito significativas.

A única exceção é relativa ao caso em que se utiliza a BVH em modelos com maior número de triângulos. Neste caso, os *speedups* reduzem para cerca de metade. Uma possível justificação para este fenómeno pode ser o facto de que para além de aumentar o número

de vezes em que a função *occlusionBVH* é chamada, essas chamadas à função irão muitas vezes resultar em oclusões, ou seja, vai ser necessário percorrer mais vezes a BVH até uma das folhas, em vez de acabar a execução da função mais cedo ao descartar por não intersetar com as *boxes*. Isto origina ganhos, mas não tão grandes quanto seriam se as oclusões não ocorressem.

V. Considerações sobre o uso de *CUDA*

Apesar de não termos sido capazes de implementar uma implementação em *CUDA*, idealizamos duas possíveis implementações, e algumas das suas vantagens e desvantagens.

Numa primeira abordagem, pensamos em paralelizar as funções de interseção e oclusão, tornando cada *thread* responsável por tratar de um dos triângulos existentes. No entanto, esta abordagem poderá levar a dois problemas: primeiro, para pequenas quantidades de triângulos, o custo de criação de *threads* poderá não compensar os ganhos obtidos; segundo, caso os *buffers* da GPU, com a informação dos triângulos, fossem criados sempre que se aborda um pixel novo, ou então sempre que as funções eram chamadas, certamente o custo com a criação dos mesmos iria cortar imenso aos ganhos. No que toca ao primeiro problema, não existe nenhuma solução aparente, a não ser restringir o uso do programa a *models* com um número significativo de triângulos. Já o segundo caso poderia ser facilmente resolvido se os *buffers* fossem criados na GPU antes de procedermos ao processamento dos pixeis, uma vez que a informação dos triângulos não é alterada com o decurso do programa.

Outra possível abordagem consiste em criar uma *thread* para cada pixel, calculando as coordenadas do pixel a processar com base nas informações da *thread*. O maior problema com esta abordagem seria que para imagens com uma elevada quantidade de pixeis (imagens de alta definição) poderíamos necessitar de mais *threads* do que aquelas disponibilizadas pela GPU. Para além disto, mais uma vez, para pequenas quantidades de triângulos, o esforço computacional necessário para replicar as informações na GPU poderia ser superior aos ganhos. Uma forma de combater estes dois problemas seria adotar uma abordagem semelhante àquela utilizada na paralelização com *tasks*, ou seja, cada *thread* da GPU seria responsável por processar uma linha de pixeis.

Caso tivéssemos implementado a versão *CUDA*, provavelmente iríamos ter optado pela segunda abordagem, sendo que o trabalho das *threads* (isto é, se cada *thread* é responsável por um pixel ou uma linha de pixeis) iria ser escolhido tendo em conta os resultados obtidos.

VI. Conclusão

Com a realização deste trabalho fomos capazes de aprofundar o nosso conhecimento no que toca não só à criação de código paralelo, mas também à compreensão sobre o impacto que uma boa estruturação dos dados tem no programa.

Além disso, testar com diferentes *datasets*, neste caso, os *models*, bem como diferentes condicionantes ao programa, como os diferentes pontos de observação utilizados, obrigaram-nos a perceber melhor como reage o *ray tracer* a uma diferente variedade de *inputs*.

Por fim, ao realizar testes em duas máquinas diferentes, fomos capazes de ver em primeira mão os efeitos que o *hardware* tem sobre a execução de um programa, e de que forma podemos beneficiar este com as limitações impostas pela máquina.

A. Appendix

i. Gráficos de *Speed Ups* Obtidos

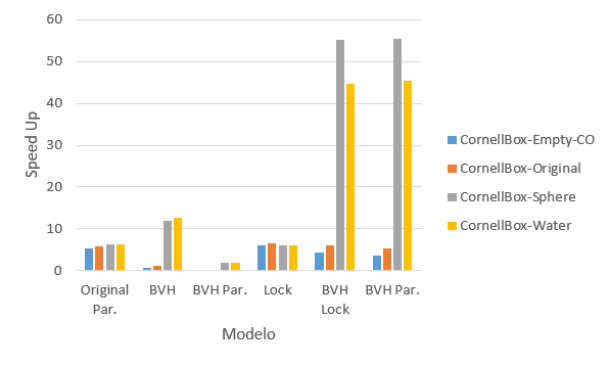

Figure 5: *Speed Ups obtidos para os vários modelos, para o ponto de observação (0, 1, 3), em relação ao algoritmo inicial*

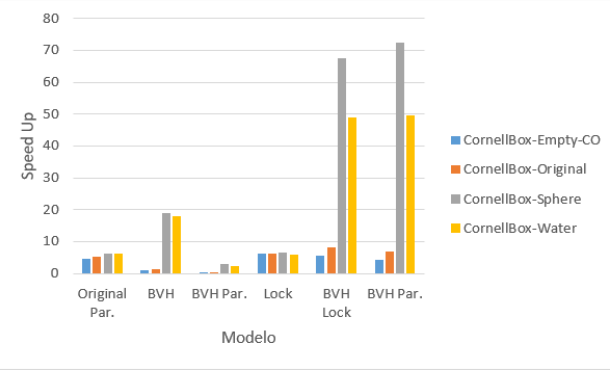

Figure 6: *Speed Ups obtidos para os vários modelos, para o ponto de observação (2, 1, 3), em relação ao algoritmo inicial*

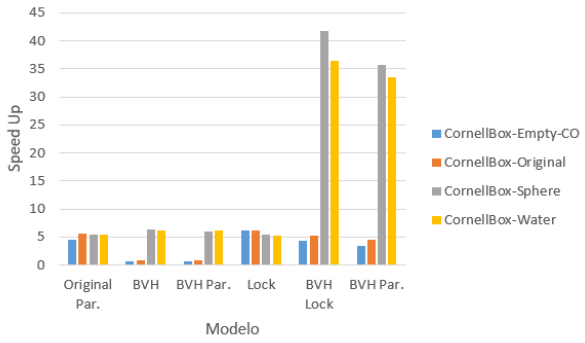

Figure 7: *Speed Ups obtidos para os vários modelos, para o ponto de observação (0, 1, 3), em relação ao algoritmo de Oclusão*

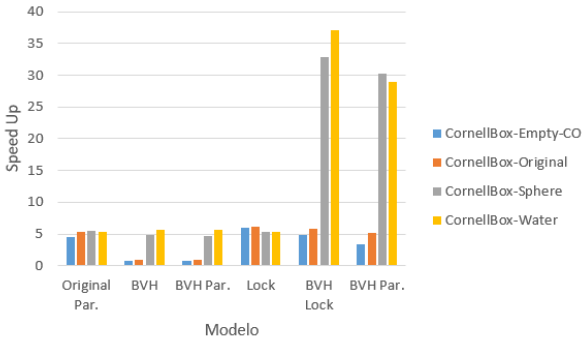

Figure 8: *Speed Ups obtidos para os vários modelos, para o ponto de observação (2, 1, 3), em relação ao algoritmo de Oclusão*

ii. Modelos

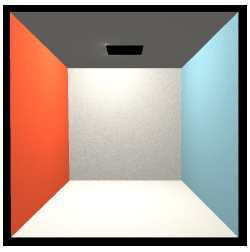

Figure 9: *Empty-CO com v = (0, 1, 3)*

Figure 10: *Empty-CO com v = (0, 1, 3) e Occlusion*

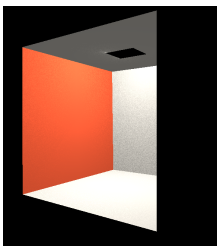

Figure 11: *Empty-CO com v = (2, 1, 3)*

Figure 12: *Empty-CO com v = (2, 1, 3) e Occlusion*

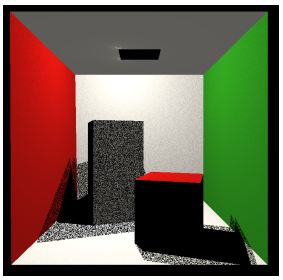

Figure 13: *Original com v = (0, 1, 3)*

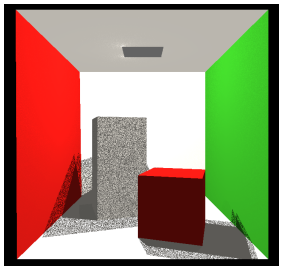

Figure 14: *Original com v = (0, 1, 3) e Occlusion*

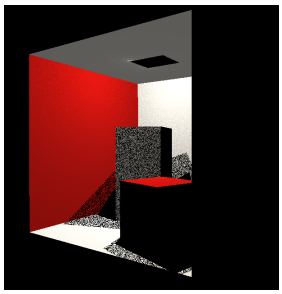

Figure 15: *Original com v = (2, 1, 3)*

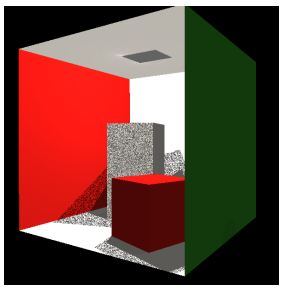

Figure 16: *Original com v = (2, 1, 3) e Occlusion*

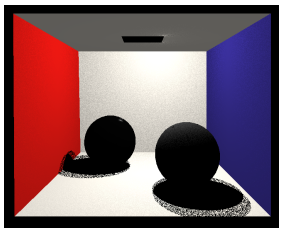

Figure 17: *Sphere com v = (0, 1, 3)*

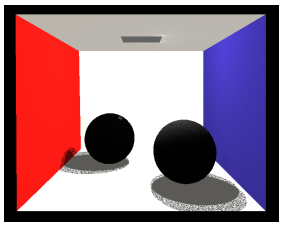

Figure 18: *Sphere com v = (0, 1, 3) e Occlusion*

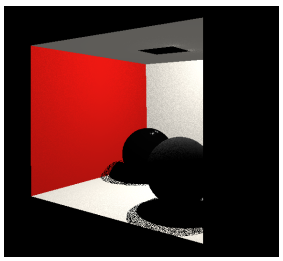

Figure 19: *Sphere com v = (2, 1, 3)*

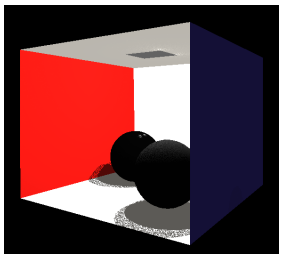

Figure 20: *Sphere com v = (2, 1, 3)*

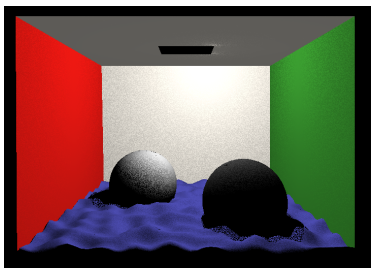

Figure 21: *Water com v = (0, 1, 3)*

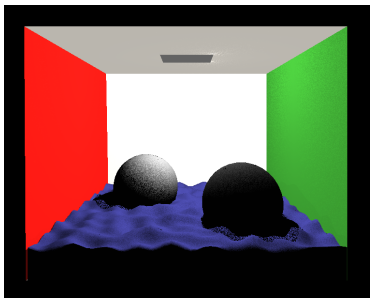

Figure 22: *Water com v = (0, 1, 3) e Occlusion*

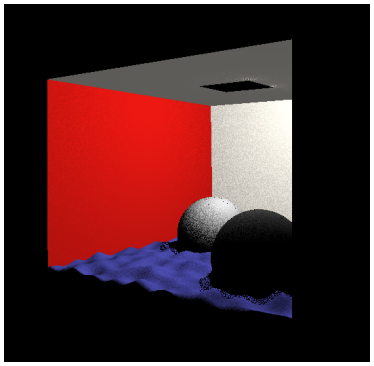

Figure 23: *Water com v = (2, 1, 3)*

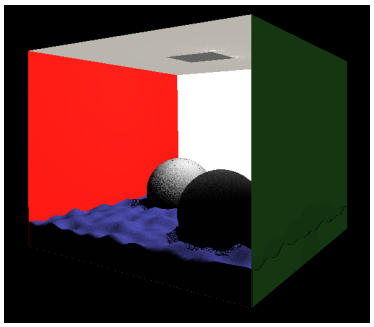

Figure 24: *Water com v = (2, 1, 3) e Occlusion*