Engenharia de Sistemas de Computção

Paralelização de um *Ray Tracer* Simples

Eduardo Conceição
Universidade do Minho
A83870@alunos.uminho.pt

Rui Oliveira
Universidade do Minho
A83610@alunos.uminho.pt

Abstract

O objetivo deste projeto é explorar várias técnicas de otimização de performance através da exploração de paralelismo em várias vertentes aplicadas a um caso de estudo no ramo da Computação Gráfica, um *ray tracer* de implementação simples. Em concreto, esperamos explorar estruturas de aceleração de *rendering*, a utilização de tarefas assíncronas do C++, *multithreading* e ISPC.

I. Introdução

No âmbito da cadeira de Engenharia de Sistemas de Computação, foi-nos proposta a otimização do código de um *ray tracer* fornecido pela equipa docente, utilizando técnicas de paralelização estudadas nas aulas. Essas técnicas passam pela utilização de *threads*, de tarefas assíncronas, da manutenção da consistência de dados através de operações atómicas e, para o caso da nossa implementação, da utilização explícita de vetorização (SIMD) com ISPC (*Intel SPMD Program Compiler*). Mais ainda, iremos tomar partido de estruturas de aceleração de *rendering*, nomeadamente uma *BVH* de modo otmizar o processo de interseção dos raios com os objetos da cena.

Neste relatório iremos apresentar as decisões que tomamos para implementar as várias otimizações pedidas e tentaremos explicar os resultados que a sua aplicação implicou, de modo a podermos compará-los ao resultado esperado e tentar explicar a diferença.

II. Exposição do Problema

Nesta secção faremos uma breve introdução ao problema, nomeadamente quais são os pontos do código fornecido que são mais adeptos à paralelização.

i. *Ray Tracing*

Ray tracing é uma técnica de computação gráfica de geração de imagens em que é traçado o caminho dos raios de luz na imagem e é simulado o seu efeito no choque com os objetos que a constituem.

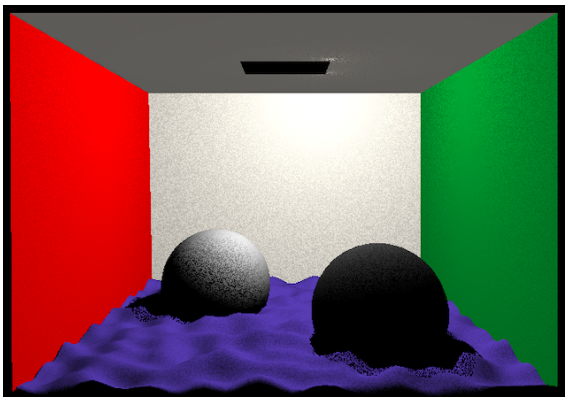

Figure 1: *Imagem gerada com um ray tracer*

Para que esta técnica possa ser implementada, em princípio, teremos um conjunto de raios na imagem, definidos pela sua origem e vetor direção, e, para cada um, testamos em que triângulos da geometria chocamos. Em primeira análise, o mais óbvio será

testar, para todos os triângulos da imagem, se existe uma colisão com o raio de luz, sendo que o triângulo mais próximo será aquele contra o qual o raio realmente colidiu. No entanto, esta abordagem tem uma intensidade computacional muito elevada. Isto deve-se ao facto que estamos a utilizar um método de força bruta que nos obriga a percorrer todos os N triângulos da geometria, apresentando este método complexidade $O(N)$. Uma primeira abordagem que podemos imediatamente implementar será uma paralelização da travessia dos vários raios da imagem, visto que, como as operações que aplicamos com cada um deles são independentes, nem temos de nos preocupar com controlo de concorrência mas, mesmo assim, não resolvemos o problema do espaço de procura ser de cardinalidade elevada.

Uma possível estratégia para contornar este problema é dividir os triângulos em vários volumes. Estes volumes, ditos *bounding volumes* são "caixas" que englobam um conjunto não vazio de triângulos. Em vez de testarmos se um raio colide com os triângulos no seu interior, podemos ver se colide com o próprio volume. Se não colidir, então também não o fará com os triângulos no seu interior. Se colidir, podemos testar os triângulos no interior. No entanto, podemos ainda adicionar recusividade a este processo, de modo a dividir os vários volumes em volumes mais pequenos. Dessa forma, vamos acabar por formar uma hierarquia de *bounding volumes*, dita *Bounding Volume Hierarchy* (BVH). Utilizando esta BVH, uma espécie particular de árvore binária de procura, passamos a ter uma procura que, no melhor dos casos, será $O(log_2(N))$, para N triângulos, num caso em que a árvore está balanceada, que é o esperado na nossa implementação.

A construção desta árvore também é de certa forma apta à paralelização, mas requer um padrão de paralelização mais sofisticado para podermos obter ganhos.

ii. *ISPC*

Outro aspeto a considerar no desenvolvimento de um programa deste calibre é a utilização de vetorização. Muitas máquinas suportam instruções *SIMD* hoje em dia, que nem sempre são devidamente aproveitadas, por uma de muitas razões.

ISPC é um compilador criado pela *Intel* que visa suportar uma extensão ao C que permite escrever programas explicitamente vetorizados, com a possibilidade de escrever variáveis que são por definição vetores de escalares que podem tomar partido das instruções das várias versões do SSE e do AVX, escrevendo programas que parecem, em primeira análise, perfeitamente sequenciais.

Uma característica interessante desta extensão ao C é que o binário gerado na compilação de um programa *ISPC* é plenamente compatível com C ou C++, característica de que tencionamos tomar partido no desenvolvimento da solução.

iii. *Locked Queue*

Uma fila bloqueante, ou *locked queue*, é uma estrutura de dados que utiliza alguma forma de mecanismo de controlo de concorrência de modo a controlar os acessos à mesma por parte de consumidores e produtores.

No contexto deste problema, essa *locked queue* irá guardar blocos de píxeis que representam uma parte da imagem a ser processada por um fio de execução, e cada consumidor irá fazer *pull* de um bloco de cada vez e processá-lo. Sendo que a ordem por que estes blocos são retirados não é relevante, a estrutura é LIFO, como uma *stack*. Mais ainda, como iremos ver mais à frente, o produtor não tem de se preocupar com controlo de concorrência.

Para além disto, no entanto, iremos também utilizar uma *locked queue* tipo FIFO para auxiliar na paralelização da construção da BVH, onde já haverá alguma preocupação em ter controlo de concorrência nos produtores.

iv. *Ambient Occlusion*

Oclusão ambiente é uma técnica suplementar ao *ray tracing* que adiciona qualidade e realismo às imagens geradas tornando as sobras menos rígidas, opacas e, por consequência, mais realistas.

O processo, de uma forma simples, consiste na criação de um número fixo de raios, normalmente dezasseis, definidos por vetores normalizados, e, para

um certo ponto, verificamos se a luz que esses raios representam é obstuída por algum objeto. Se não for, atualizamos a cor dos píxeis desse ponto, de modo a ficar mais clara.

III. Implementação da Solução

Nesta secção iremos explicar como implementamos a solução ao problema que nos foi dado face às condições impostas, bem como as possíveis consequências que as nossas decisões trarão.

i. Paralelização com Tarefas Assíncronas/ *Threads*

O primeiro passo de paralelização que fizemos foi procurar sítios no programa que pudessem ser paralelizados com a utilização de *threads* ou de tarefas assíncronas.

Em primeiro lugar, optamos por utilizar tarefas assíncronas em vez de *threads*. Esta escolha foi feita devido ao facto que tarefas assíncronas podem ser escalonadas de modo a serem atribuídas a *threads* à medida que estas estão disponíveis, em vez de criar um número de *threads* proporcional ao número de tarefas, que aconteceria se utilizássemos *std::threads*. No entanto, o escalonamento feito por defeito pelo C++ destes futuros é um de dois: ou o futuro é executado numa nova *thread* quando é criado, ou é executado por uma *thread* aleatória numa *pool* de *threads* quando invocamos o resultado do futuro (com *get*), pelo que nenhuma das duas é exatamente o que nós queríamos das tarefas assíncronas, e a implementação com *async* ou com *threads* não mostra tantas diferenças como inicialmente pensámos, pelo que, nesta secção, utilizaremos os dois termos intercambiavelmente.

O primeiro lugar onde decidimos utilizar tarefas assíncronas foi no varrimento dos vários raios que percorrem a imagem, um varrimento *row wise*. Uma vez que este varrimento é embaraçosamente paralelo, visto que não existem dependências entre as várias iterações do ciclo *for* externo (que percorre as linhas), podemos atribuir a cada uma das tarefas assíncronas a função de percorrer uma linha inteira de píxeis da imagem. Para este efeito, temos um *vector* de *tasks* onde guardamos cada uma das tarefas a ser executadas, para execução posterior.

Como dissemos anteriormente, este processo é embaraçosamente paralelo, devido ao facto que é um processo em que as variáveis que são acedidas por mais do que um *future* são *read only*, e em todo o processo de execução da tarefa, não escrevemos para variáveis partilhadas, pelo que não precisamos de implementar nenhum mecanismo de controlo de concorrência.

Em princípio, o *speedup* será proporcional ao número de *threads* que executam as várias tarefas. Este processo mais tarde no desenvolvimento do trabalho foi alterado para utilizar uma *locked queue*, que veremos mais tarde.

Inicialmente também utilizamos estes artefactos para paralelizar a construção da BVH, mas mais tarde alteramos para uma implementação mais sofisticada.

ii. Implementação de uma Estrutura de Aceleração

ii.1 Estratégia dos *Bounding Volumes*

Para criar os *bounding volumes* utilizados na BVH, utilizamos a estratégia mais simples, $AABB$ (*Axis Aligned Bounding Boxes*), em que as *boxes* são paralelipípedos alinhados com os eixos xyz. Para este propósito, temos uma estrutura auxiliar que guarda o vértice mais próximo da origem da *bounding box* e o mais afastado, que podem ser obtidos calculando o mínimo das coordenadas x, y e z de cada um dos triângulos dentro da *bounding box*, e o máximo (não necessariamente respetivamente).

Para podermos inicializar a *box*, teremos de percorrer os triângulos que estarão dentro da mesma, uma operação de complexidade $O(N)$.

ii.2 Criação da *BVH*

Para que possamos criar a BVH, temos primeiro de converter a *SceneMesh* num vetor de triângulos. Para isto utilizamos uma *struct triangle* que guarda os elementos básicos utilizados para definir um triângulo e um valor do eixo. Este valor do eixo serve para, quando a função *sort* for chamada, o comparador utilizado saiba qual é o eixo utilizado como termo de

comparação. Isto é necessário pois o eixo muda para cada nível da árvore e, como tal, é mais eficiente em termos de tempo guardarmos este valor no triângulo (mas não tanto em termos de memória, mas é um *trade-off* que podemos fazer de consciência tranquila nos dias de hoje). A conversão é feita imediatamente que a estrutrua da *SceneMesh* está preenchida, e é contabilizada no tempo de execução do programa, e consiste apenas no processo de percorrer os vértices da *SceneMesh*, agrupá-los três a três numa *struct triangle* e adicioná-los ao *vector*.

A estratégia de criação da *BVH* que utilizamos é a seguinte, assumindo que o *input* inicial do algoritmo é uma lista de triângulos:

- Caso a lista passada tenha apenas um triângulo, criamos um nodo folha com a *Bounding Box* do mesmo e o algoritmo para;
- Caso a lista tenha dois triângulos, críamos imediatamente os dois nodos filho do nodo onde nos encontramos como acontece no primeiro caso e o algoritmo para;
- Utilizando o eixo que os triângulos têm como termo de comparação, ordenamos a lista de triângulos com base na coordenada do seu centro nesse eixo por ordem crescente;
- Na estrutura do triângulo, atualizamos o eixo que serve como termo de comparação (este passo é algo demorado e poderá ser melhorado, uma vez que);
- Dividimos a lista a meio, criando duas listas de triângulos, *tri_ left* e *tri_ right*;
- Criamos uma *Bounding Box* para ambas as listas e aplicamos o algoritmo recursivamente a ambas.

No que toca ao método de divisão das listas, nós entendemos não ser necessariamente o melhor e que existem várias heurísticas bem estudadas e melhores. No entanto, tendo em conta o tempo que nos foi dado e a ambição deste projeto, acreditamos ser suficiente. Testamos também utilizando uma média da coordenada dos pontos do eixo em questão, mas esta abordagem provou ser mais custosa, o que é de esperar, pelo que a abandonamos.

Um efeito secundário bastante positivo da forma como dividimos a lista é que, como, em cada passo da criação, dividimos a lista a meio, a árvore resultante está balanceada., logo a pesquisa na mesma é uma operação $O(log_2 N)$

Em termos de *paralelização da construção da BVH*, decidimos incialmente adaptar o algoritmo anterior com a seguinte modificação: sempre que fizermos a separação da lista de triângulos em dois e chamarmos recursivamente a função *create_ bvh*, a tarefa da chamada desta função será atribuída a um *future*. Isto faz com que as chamadas recursivas à função *create_ bvh* sejam feitas em paralelo. A decisão de envergar por este caminho na paralelização adveio de dois factos: das várias tarefas que são executadas na construção da *BVH*, excluíndo a ordenação da lista de triângulos, esta é a que mais adepta se demonstra a paralelização, uma vez que as tarefas são independentes umas das outras, e do facto que este é o método normalmente aconselhado na paralelização da construção de uma árvore [4].

O algoritmo original é de complexidade $O(N)$, e, sendo que todas as execuções da chamada da função *create_ bvh* são executadas em paralelo, em teoria, passaremos a ter um algoritmo de complexidade $O(log_2 N)$, uma vez que as criações dos nodos de um dado nível em princípio poderiam ser feitas em paralelo (tenhamos em conta que estamos a simplificar a complexidade real, ignorando o facto que em cada execução da função *create_ bvh* chamamos a função *sort* da biblioteca *algorithm*, que tem complexidade $O(N \times log_2 N)$ mas, para podermos comparar, esta simplificação deverá ser suficiente). Como tal, seria de esperar que houvesse um *speedup* significativo com esta adição, mas, como veremos mais à frente, isto não se verificou.

Uma segunda tentativa de paralelização, aquela com que acabamos por seguir, utilizou uma *locked queue*, pelo que iremos explicar o funcionamento desta nova estratégia mais à frente.

Um fator adicional que poderíamos ter tido em consideração seria a utilização de um *sort* otimizado, como o *radix sort* [5], que pudesse, por sua vez, ser paralelizado, adicionando uma camada de otimização

que poderia ter efeitos bastante positivos na *performance*, mas que, devido a falta de tempo, também não pudemos implementar.

ii.3 Travessia da *BVH*

A implementação correta da travessia da *BVH* é o que torna a sua implementação interessante, uma vez que é aqui que vamos poder tomar partido das vantagens desta estrutura. Como tal, assumindo que a função recebe como *input* um raio (definido pela sua origem e direção):

- Olhando para a *Bounding Box* do nodo em que nos encontramos, verificamos se só contém um triângulo. Caso seja verdade, fazemos a interseção com o mesmo com a *intersect_ triangle*;
- Caso contrário, veremos se o raio interseta a *box* no nodo, com a função *box_ intersect*, que segue um algoritmo de interseção relativamente simples e rápido [6]. Caso intersete a caixa, verificamos ambos os nodos filhos recursivamente.

Em teoria, teremos um *speedup* considerável quando aplicarmos esta técnica, uma vez que vamos recursivamente "podando" ramos da árvore que podemos ignorar, pois, se o raio não interseta uma *bounding box*, também não irá intersetar as *boxes* que ela contém, pelo que podemos ignorar todos os nodos filhos que a ela estão associados e, por consequência, uma parte significativa dos triângulos da figura.

Mais ainda, a mesma estratégia foi utilizada para implementar a função de oclusão, sendo que nesta o algoritmo é ligeiramente diferente, e ambos os nodos filhos de um dado nodo deverão ser vistos. Visto que verificamos tanto o *left node* e o *right node*, na maioria dos casos não haverá nenhum *speedup* visto que não é feito *pruning* da árvore. Só haverá alguma vantagem numa situação em que a procura na árvore esquerda retorna a um certo ponto *true*, uma vez que torna a procura na árvore direita desnecessária visto que a primeira condição de uma cláusula ou é verificada.

No que toca ao algoritmo de interseção do raio com a *box*, é um algoritmo simples que utiliza os limites da caixa para calcular se existe alguma interseção possível com o raio [6].

iii. *Locked Queue*

A ideia de uma *locked queue* foi utilizada para implementar otimizações em duas vertentes do programa: primeiro, no processamento dos píxeis da imagem e, em segundo, na construção da *BVH*.

Em termos da implementação da *Locked Queue* para os píxeis, começamos por dividir a imagem por blocos de sessenta e quatro píxeis. A tradução disto em código é a criação de uma estrutura que guarda a definição de um bloco, sob a forma de três inteiros: o número da linha, a coluna onde o bloco começa e o bloco onde acaba. Guardamos estes valores pois a divisão é feita *row wise* e o número de colunas das imagens *input* que nos foram fornecidas são múltiplos de sessenta e quatro, pelo que o número de píxeis de uma linha podem ser divididos em blocos sem haver píxies de sobra, logo não haverão blocos divididos por duas linhas. A *queue* em si não é mais do que uma lista de blocos, que são guardados e criados sequencialmente.

Em termos do número de *threads* que vai aceder à *queue*, este é estabelecido consoante o número de *cores* da máquina da equipa (descrição em anexo). Para esse efeito utilizamos a função *hardware_ concurrency* para obter o número de *threads hardware supported* da máquina. Para a máquina de teste, esta função devolve oito, o que corresponde às especificações. No entanto, de acordo com o pedido no exercício, utilizamos apenas metade disto, de modo a corresponder ao número de *cores* da máquina. Assim, criamos um vetor de *threads* e alocamos desde já a memória necessária para quatro *threads*.

Quanto ao aspeto da utilização da *locked queue*, tentamos duas estratégias diferentes. A primeira tentativa que experimentamos implementar seguia uma filosofia *lock free*, utilizando a capacidade da classe *template atomic* do C++ de modo a implementar uma lista ligada atómica (semelhante à dos *slides* da cadeira). Para esse efeito, apenas o *pull* seria relevante ser atómico. Isto deve-se ao facto que o produtor dos elementos da *queue* é uma única *thread* sequencial, pelo que, até a *queue* estar pronta, não haverão *races* para aceder ao seu conteúdo. Como tal, não haveriam problemas de acessos paralelos para produtores (pois só há um). O importante estaria

nos consumidores, que farão acessos paralelos e descontrolados. A motivação aqui seria permitir que a atomicidade da estrutura (que seria *lock free*, visto que a estrutura caberia toda numa única linha de *cache*, nem que fosse necessário declará-la como alinhada) e da função *pull* funcionassem de forma a tornar *locks* desnecessários, com a motivação que prescindir de chamadas ao sistema operativo para fazer o *lock* melhorasse a *performance* do programa. No entanto, devido a problemas que surgiram na criação da lista ligada com aritmética de apontadores e o aviso do professor que esta estratégia seria muito mais complicada do que inicialmente pensamos e provavelmente não teríamos tempo para a concluir, decidimos abandonar esta abordagem.

A segunda estratégia que tentamos, que foi a que decidimos seguir até ao fim, utiliza um simples *mutex* e um vetor de blocos, em vez da lista ligada atómica. Cada *thread* utiliza a seguinte estratégia para aceder à *queue*:

```
/* .... */

while(locked_queue.size() != 0)
{
    // obtain lock
    mutex.lock();

    // check again if list is empty, if it is, break
    if(locked_queue.size() == 0)
    {
        // unlock so others can acess queue
        mutex.unlock();
        break;
    }
    // pull block to process
    struct block bl = locked_queue.pull();

    // unlock mutex
    mutex.unlock();

    //process block
    ...
}
/* ... */
```

Figure 2: *Algoritmo da Locked Queue*

Um aspeto que deveremos explicar é a segunda verificação do tamanho da *queue*. Existe a possibilidade de verificarmos que a fila ainda tem elementos, ou seja, a condição do *while loop* verifica-se, mas, antes de fazermos *lock*, outra *thread* terá acedido à *queue* e retirado o último elemento. Se isso acontecer, quando nós obtivermos o *lock*, ao fazer *pull* vamos ter um erro no acesso à memória e o programa vai falhar, Para resolver isso, quando nós tivermos acesso ao *lock* da *queue* outra vez, deveremos verificar se a fila realmente tem algum bloco ainda ou não e, caso não tenha, aí paramos. Este é o caso de paragem do algoritmo, que, como podemos ver, funciona de uma forma algo semelhante ao algoritmo geral da *compare_ exchange_ strong* dos *atomics*. De modo a diminuir mais ainda o número de acessos ao *lock* que o algoritmo faz poderíamos adicionar um *check* ao tamanho da fila no fim de uma iteração (guardando o tamanho da mesma quando acedemos ao *lock*), o que permite que hajam menos acessos ao *lock*. No entanto, como o número de *threads* é bastante pequeno, a diferença não seria notória, mas para uma execução num ambiente *many core*, já deveria ser tida em consideração.

A motivação para esta otimização passa pela ideia de colocar todos o *cores* da máquina a processarem uma quantidade de tarefas que, em princípio, estará mais ou menos equilibrada, assumindo que todos os núcleos da máquina estão em estado equiparável.

Uma estratégia muito semelhante foi utilizada para a contrução da *BVH*, com duas diferenças importantes:

- Esta fila é um FIFO, o que é uma necessidade pois é preciso que tarefas que são adicionadas primeiro à fila sejam as primeiras a ser executadas (a ver mais à frente a razão);

- O produtor deixa de ser um único fio de execução, pelo que é preciso controlo de concorrência no mesmo também.

O processamento dos elementos que estão na *locked queue* é feita de forma igual ao primeiro caso. Estes elementos são, neste caso, uma estrutura com os argumentos necessários para criar um novo nodo, ou seja, o apontador para o nodo, o número de triângulos que o volume vai conter e o eixo que vamos avaliar.

```
/* ... */

while(locked_queue.size() != 0)
{
    //obtain lock
    mutex.lock();

    //check again if list is empty, if it is, break
    if(locked_queue.size() == 0)
    {
        mutex.unlock();
        break;
    }

    //pull node to process (from front)
    node_to_process = locked_queue.pop_front();

    //process
    create_bvh(node_to_process);

    //unlock
    mutex.unlock();
}

/* ... */
```

Figure 3: *Algoritmo da Locked Queue para a BVH*

Como podemos ver, nesta *queue* o *lock* é mantido pela *thread* enquanto está a processar o nodo. Isto acontece devido ao facto que, dentro do processamento do nodo, se este não for uma folha, vamos adicionar elementos à *queue* (os dois nodos filhos).

Em primeira análise, poderíamos fazer *unlock* mal façamos *pop*, e depois pedimos o *lock* de novo dentro da função *create_bvh*, o que implicaria pedir o *lock* duas vezes por cada elemento da *queue*. Isto tem dois problemas: em primeiro lugar, implica cada *thread* passar mais tempo chamadas ao sistema para pedir o *lock* e segundo, se a fila ficar temporariamente vazia com o *pop* e depois serem adicionados dois elementos na sua *create_bvh*. Se as outras *threads* obtiverem todas o *lock* e verificarem que a fila está vazia antes dos dois elementos serem adicionados, podemos encontrar o problema de apenas a *thread* que inseriu os dois elementos novos vir a saber que a fila não está vazia. Como todas as outras *threads* já terminaram a execução, apenas uma *thread* ficará a executar todo o trabalho que faltar, tornando a execução exatamente igual à paralela, com a adição do *overhead* dos *locks*. Assim, a forma como fazemos *lock* e *unlock* é necessária.

iv. ISPC

Uma das otimizações que nos foi pedida foi a utilização de *SIMD* explícito através da utilização de *ISPC*, tanto uma extensão ao C como o respetivo compilador.

O primeiro passo será escolher os pontos do código onde podemos tirar partido de *SIMD*. Numa primeira análise, podemos concluir que poderemos tomar partido do mesmo na interseção com os triângulos e na oclusão.

iv.1 Estruturas de Dados

Para podermos implementar isto, temos primeiro de transformar as estruturas relevantes no seu equivalente em *ISPC*. As estruturas de que vamos precisar são as seguintes:

- *vec3*, para representar um ou mais pontos no espaço;
- *triangle*, que será utilizada para representar um grupo de triângulos;
- *ray*, uma estrutura que guardará todas as informações referentes ao raio que são necessárias para calcular a interseção;

As estruturas adaptadas têm a seguinte assinatura:

```
struct uniform_vec3
{
    uniform float v[3];
};

struct varying_vec3
{
    varying float v[3];
};

struct triangle
{
    uniform varying_vec3 vertices[3];
    varying int primID;
    varying int geomID;
};

struct ray
{
    uniform_vec3 ori;
    uniform_vec3 dir;
    varying float t;
    varying float u;
    varying float v;
    varying int geomID;
    varying int primID;
    varying bool valid;
}
```

Figure 4: *Structs ISPC*

Há alguns fatores importantes que devemos referir para explicar as nossas escolhas relativas à definição destas estruturas:

- A distinção entre *uniform_vec3* e *varying_vec3* é importante. Vão surgir situações em que queremos ter uma *struct* a representar mais do que um vetor, e outras em que isto não é verdade. Apesar de podermos fazer alterações que permitem a utilização intercambiável das duas, o *varying_vec3* permite-nos utilizar melhor a memória da *cache* pois é explícitamente uma *structure of arrays* (*SOA*), que é preferível em geral a um *array of structures*.
- A estrutura *triangle* tem, na verdade, capacidade de representação de 8 triângulos (ou qualquer que seja a *SIMD width* da máquina em causa, que, no programa, está definida com 8), e toma proveito do conceito de *SOA*s uma vez mais, sendo que o *vertices* será representado em C/C++ como um *array* bidimensional.
- Por fim, o *struct ray* é a única estrutura única do ISPC, e vai guardar os valores do *t*, *u* e *v* para cada triângulo que interseta, bem como os seus *geomID* e *primID*. A origem e direção são *uniform_vec3* pois, para um *check*, são sempre a mesma, mas é *varying* para facilitar a utilização das funções auxiliares. O booleano *valid* serve para nos dizer, para cada *lane SIMD*, se o valor de *t*, *u*, *v*, *geomID* e *primID* lá guardados são válidos, pois podem não ser, como vamos ver à frente.

Uma pequena nota que devemos adicionar, quanto à utilização inconsistente da *keyword varying*[2], esta deve-se ao facto que decidimos utilizá-la explicitamente em certos locais para nos facilitar o raciocínio, pois não estamos muito habituados a *ISPC*.

Estas estruturas, em particular o *triangle*, têm de ser inicializadas dentro da *main* do programa, o que adiciona algum *overhead*, semelhante à contrução da *BVH* ou da conversão da *mesh* de pontos para uma lista de triângulos como tínhamos na versão com *async*. Estes triângulos vão ser, mais uma vez, como no *async*, guardados num *vector* de C++ (o que nos impede de passar a estrutura toda de uma vez para o *ISPC*, mas também não seria essa a nossa intenção).

iv.2 Implementação das Funcionalidades

O *ISPC* poderia ser utilizado em vários locais do programa, no entanto nós escolhemos implementar as funções *intersect* e *occlusion*, tanto por conselho do professor como por escassez de tempo.

Para ambas as funções, é necessário ter a sub-rotina *intersect_triangle* implementada, e esta, por sua vez, precisa das funções *cross* e *dot*, para implementar alguma aritmética com vetores.

Olhando primeiro para estas funções (*cross* e *dot*) a sua implementação pouco difere daquilo que tínhamos em C++, mas passamos a ter versões das mesmas que suportam multiplicações com vários tipos de vetores e suas combinações (*varying_vec3* e *uniform_vec3* e as várias permutações possíveis entre eles), sendo

que estas não diferem em nada em relação às originais, exceto no facto que todas as versões em que um *varying_ vec3* é passado, o *float* retornado é também *varying*.

Mudanças mais interessantes podem ser observadas na função *intersect_ triangle*. Nesta função, existem vários pontos do código em que uma qualquer variável é verificada e a execução da função pode ou não finalizar aí. Como a função é implementada utlizando como argumento um *triangle* com os vértices referentes a oito triângulos (no máximo), nós temos de fazer as verificações e a devida aritmética com todos. E se, quando chegarmos a estes *returns* condicionais, alguns tiverem valores que justificam continuar e outros não? Bem, é aqui que entra o membro *valid* da *struct ray*. O *valid* vai conter um booleano para cada uma das *SIMD lanes* que nos diz se os valores *t*, *u*, *v*, *geomID* e *primID* são válidos. Numa situação em que o *check* de interseção com um triângulo é inválido, temos o *bool* dessa *lane* a *false*, e caso contrário, temos a *true*. Apesar de continuarmos a fazer os cálculos com estes triângulos inválidos mais tarde, sabemos que esses valores não nos interessam pois temos o seu *bool* como falso (nas várias alterações a esta variável, uma *lane* falsa nunca poderá ser de novo verdadeira, mas uma verdadeira pode passar a falsa). No entanto, pode acontecer que nenhum triângulo passe de um dado *check* mas todos continuam a ser testados nos próximos, apesar de todos serem inválidos. Para resolver isto, sempre que atualizamos o *valid*, retornamos se virmos que todas as suas *lanes* são *false* (verificando-as com a função *all* da *standard library* do *ISPC*). Este *check* não seria necessário se os vários valores verificados fossem *uniform*, visto que aí todas as *lanes* têm de concordar por que caminho deverão seguir, mas esse não é o caso neste programa.

Tal como na versão original, a função retorna *true* se houver interseção entre o raio passado e o triângulo e *false* caso contrário, sendo este booleano *uniform*.

Para implementar a função *intersect* no *ISPC*, alteramos os argumentos originais, de forma a que esta apenas recebe uma *struct triangle* em vez da *SceneMesh* inteira e uma *struct ray* em vez de vários parâmetros dispersos. O que a função faz é apenas aplicar a função *instersect_ triangle* ao *triangle* passado, e se retornar verdadeiro, atualiza os *geomID* e *primID* da estrutura *ray* com os da *triangle* passada e retorna se algum dos *geomID*s ou *primID*s foram atualizados (ou seja, se são diferentes de -1), com o auxílio da função *any* da *standard library* de *ISPC* (vale a pena relembrar que tanto o *geomID* como o *primID* são *varying*).

Dentro da *main*, onde a *intersect* é chamada, é necessário fazer alterações. Uma vez que apenas oito triângulos, no máximo, são passados às funções de *ISPC* de cada vez, temos de chamar a *intersect* várias vezes e analisar o resultado em cada iteração, guardando no fim apenas o *t* mais pequeno válido (validade que podemos verificar com auxílio do membor *valid* da estrtura *ray*) e o *geomID* e *primID* associado, sendo que, daí em diante, o raciocínio é o mesmo. Esta verificação do *t* válido obriga-nos a percorrer todos os oito triângulos dentro de uma *ispc::struct triangle*, o que adiciona algum *overhead* de computação e até de acesso à memória que poderá ser sentido (e, como veremos mais à frente, é mesmo) na *overall performance* do programa.

A implementação da *occlusion* é semelhante à *intersect* mas, como apenas queremos saber se existe algum triângulo em que o raio colide, apenas fazemos a verfificação da *intersect_ triangle* e devolvemos o que esta devolver. Depois, a lógica na *main* seria a mesma da *intersect*, em que teríamos de percorrer toda a lista de *struct triangles* e, se em algum ponto esta retornar *true*, paramos o ciclo, pois vimos que de facto existe oclusão. No entanto, na versão final do código não utilizamos esta função, pois tinha um erro relacionado com geometria, em que a função de *occluion* estava a assumir que havia interseção com o triângulo de onde o raio parte, no teto, sendo que o resto das primitivas ficavam completamente obstruídas pela sombra da mesma e a imagem ficava completamente preta.

v. *Ambient Occlusion*

Para que nos possamos situar no código, a implementação da oclusão ambiente encontra-se imediatamente a seguir à verificação da oclusão em relação a cada uma das fontes de luz (no caso dos ficheiros *input* fornecidos, uma).

A implementação do código em si é constituída por um ciclo *for* onde geramos *SAMPLE_ SIZE* vetores

aleatórios, sendo que em geral $SAMPLE\_SIZE = 16$, e para cada um deles aplicamos a função *occlusion* (ou *bvh_ occlusion*), utilizando como origem o ponto onde o raio atingiu, que é calculado anteriormente, e como direção o vetor aleatório que criamos. Se retornar *true*, ignoramos (ou seja, o raio é obstruído por alguma coisa na imagem e não atualizamos as cores). Caso contrário, adicionamos às cores uma contribuição por parte desse raio, que é calculada com base no material do objeto em que batemos.

Um fator que é problemático neste código é gerar um vetor aleatório que se conforme às condições necessárias para fazermos o cálculo da oclusão, sendo que a condição de que falamos é:

$$N \cdot P \geqslant 0 \wedge \|P\| = 1$$

Sendo N a normal da superfície em que o raio bateu.

A primeira abordagem a este cálculo passou, como muitos algoritmos relacionados com geração pseudo-aleatória de números, pela utilização de um ciclo *while* que, enquanto que $N \cdot P \geqslant 0$, calculamos um novo P e testamos de novo. Esta abordagem é terrível em termos de distribuição de carga. Se nós assumirmos que vão existir t *threads* a executar este código, o tempo de execução do mesmo nem é comparável entre *threads*, pois algumas passarão muito mais tempo presas neste ciclo do que outras. Isto também se verificou em testes, e degradou a *performance* consideravelmente. A nossa solução para este problema é simples, mas baseou-se na solução de outro *ray tracer*[1], e consiste apenas em transformar o vetor P no seu simétrico.

Esta solução funciona pois, se o produto interno entre a norma da superfície e o novo raio for negativo quer dizer que o raio bate por trás da superfície, em vez de na frente. Se sabemos isto, então podemos conluir que um vetor com a mesma direção e norma, mas sentido oposto baterá na frente, daí utilizarmos o simétrico.

Dessa forma, podemos contornar este ciclo, e o trabalho feito por cada um dos fios de execução vai ser não só semelhante, mas também será muito menos por não haver um ciclo *while* com um número não determinístico de iterações a ser executado.

Quanto aos valores que utilizamos para P, estes podem ser quaisquer vetores de números reais, pois a função de normalização do vetor permite-nos transformar P numa versão normalizada de si mesmo, ultrapassando quaisquer problemas que pudessemos ter na satisafação da segunda condição. É importante notar que isto funciona no contexto deste problema, mas tem implicações que poderiam afetar os cálculos negativamente no contexto de outros problemas.

Na seguinte imagem podemos comparar uma imagem *output* com e sem *ambient occlusion*, notando que as sombras são muito mais leves com *ambient occlusion*:

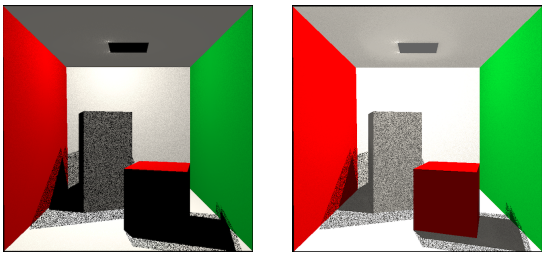

Figure 5: *Sem Ambient Occlusion vs. Com Ambient Occlusion*

IV. Testes

i. Ambiente de Teste

O ambiente de teste utilizado foi uma das máquinas da equipa, um PC *OMEN by HP - 15-ce012np quad-core*[3], cuja descrição detalhada podemos encontrar em anexo.

O *cluster Search*[7] do Departamento de Informática não foi utilizado devido a problemas de incompatibilidade com algumas das ferramentas utilizadas.

ii. *Input* e Condições de Teste

Como input para os testes foram utilizados os quatro ficheiros de imagens fornecidos: *CornellBox-Original*, *CornellBox-Mirror*, *CornellBox-Sphere* e *CornellBox-Water*, em crescente número de triângulos.

Para cada versão diferente do programa, o mesmo input foi testado dez vezes, e o resultado aceite segue uma regra *k-best*, com $k = 3$. A única exceção a isto é o ficheiro *Water*, que, para a execução sequencial,

apenas foi testado cinco vezes, devido ao enorme peso computacional que tinha e pressão que causava sobre a máquina.

Para podermos comparar as diferentes versões do programa que implementamos, testamos as seguintes combinações:

- Versão Sequencial
- Paralelização com *Locked Queue*
- Paralelização com Tarefas Assíncronas
- Tarefas Assíncronas + BVH
- Tarefas Assíncronas + BVH + Construção Paralela
- *Locked Queue* + BVH + Construção Paralela

Experimentamos as duas versões da construção paralela da árvore (identificando a versão sem *locked queue* como v.1 e com *locked queue* como v.2).

É de sublinhar que nenhum dos testes comporta a utilização de *ambient occlusion*. Isto deve-se ao facto que a sua utilização deverá apenas acrescentar uma quantidade proporcional de trabalho a cada um dos fios de execução, devido à forma como implementamos. No entanto, iremos referir algumas das conclusões a que chegamos com os testes que fomos realizando à medida que fomos implementando as melhorias à mesma que referimos anteriormente.

iii. Análise de Resultados

iii.1 Otimizações

Olhando primeiro para os resultados dos testes com a versão 1 da paralelização da *locked queue*:

	Seq.	Async	BVH	Async + BVH	Async + BVH + Const. Paralela v.1	Locked Queue + BVH + Const. Paralela v.1
Original	9867.0	2271.0	4043.3	1034.0	1047.7	1244.3
Mirror	9865.0	2271.3	4037.0	1030.7	1052.7	1243.3
Sphere	463005.0	116098.0	3708.0	1078.0	1131.0	1182.7
Water	1482647.0	373939.0	4489.0	1528.3	1747.7	1638.0

Figure 6: *Resultados (ms)*

Daqui podemos observar alguns fenómenos interessantes.

Em primeiro lugar, podemos ver que a paralelização demonstrou para todos os casos ganhos bastante positivos. Em particular, as imagens maiores foram as que demonstraram mudanças mais significativas.

Olhando primeiro para a paralelização com *async*, podemos ver que os ganhos rondom todos 4x. Em teoria, os ganhos deveriam ser lineares com o número de tarefas em execução paralela, pois cada uma das linhas de píxeis é tratada em paralelo. No entanto, isto não se verifica. Isso deve-se ao facto que, para esse comportamento se verificar, teríamos de assumir que a criação e manutenção de dezenas e dezenas de *threads* para tratar de cada uma das tarefas não tem um efeito negativo sobre a *performance* do programa, que obviamente terá. Certos comportamentos que mais tarde verificamos revelam que o *speedup* de 4x faz sentido quando temos em conta que estamos a utilizar 4 *cores*, logo vamos ter 4 unidades de processamento completamente independentes umas das outras. Dentro dessas, se o número de *threads* que pedimos for maior do que 1 vai haver utilização de *hyperthreading* e provavelmente o escalonamento justifica as perdas em relação ao teórico.

Quando adicionamos a BVH, também existe uma diferença muito notória de *performance*. O espaços de procura para a interseção e para a oclusão passarem a ser $O(log_2 N)$ fazem uma diferença enorme. É interessante vermos que o tempo é mais ou menos constante (rondando os quatro mil milissegundos). Tendo em conta que as imagens maiores têm um número muito elevado de triângulos, isto leva-nos à conclusão que realmente a diferença aqui é proporcional ao logarítmo de base dois do número de triângulos, sendo que ainda é adicionado o *overhead* de construção da própria BVH.

Como seria de esperar, os ganhos quando juntamos este dois últimos são aproximadamente um quarto do tempo com apenas a BVH.

A construção paralela da BVH, como podemos ver, não produz nenhum efeito positivo da forma como a versão 1 está implementada. Isto deve-se ao facto que na verdade, o que acontece quando fazemos a paralelização desta forma é que deixamos apenas que o nodo esquerdo seja criado depois do direito (sendo que, normalmente, é criado primeiro o esquerdo), não fazendo realmente diferença em níveis abaixo. Isto

obviamente impediria a obtenção de ganhos, sendo que, no máximo, adiciona algum *overhead* pela criação das tarefas assíncronas/*threads*, que podemos ver que é sentido (há pouca diferença, pouco mais de setenta milissegundos na *Water*, mas ela existe). Mais ainda, assumindo que a criação de um *async* gera a necessidade de criar uma nova *thread*, a criação contínua de *threads* é imensamente pesada, sendo que seriam criadas $1 + 2 + 4 + 8 + ... + N$ *threads*, sendo N o número de triângulos da geometria.

Por fim, a utilização da *locked queue* em vez de futuros completáveis para fazer o varrimento dos píxeis também não demonstrou ganhos significativos. Isto é fácil de explicar para as duas imagens mais pequenas, onde até houve *slowdown*: todas as *threads* que tentam aceder à *queue* têm de esperar por um *lock*. A obtenção do *lock* é uma operação pesada e lenta. Para uma imagem mais pequena, um bloco de píxeis irá requerer menos tempo a processar, visto que os cálculos da interseção e oclusão são mais rápidos. Quando juntamos estes factos, concluímos que o que está a acontecer é que as *threads* passam muito tempo à espera de obter o *lock* e pouco a processar um bloco, pelo que o tempo útil de trabalho é menor do que o tempo que estão *idle* à espera do *lock* quando vão à *queue* procurar trabalho. Este efeito não é tão forte com uma imagem maior, devido ao facto que as *threads* têm muito mais trabalho útil, e em princípio irão menos regularmente à fila.

Podemos comparar o efeito apenas das tarefas assíncronas e da *locked queue*, de modo a termos uma melhor ideia de como as duas implementações se comportam e qual o seu peso no *speedup*:

	Async	Locked Queue
Original	2271.0	2714.0
Mirror	2271.3	2713.0
Sphere	116098.0	129371.3
Water	373939.0	417250.3

Figure 7: *Async vs. Locked Queue (ms)*

É interessante ver que, neste caso, quando não temos uma estrutura de aceleração, a *locked queue* é sempre pior, sendo que o *slowdown* em relação ao *async* é sempre proporcional ao tamanho. Como não temos uma estrutura que nos decomponha o espaço de procura em $O(log_2 N)$, o trabalho feito para ambos os casos é sempre igual e sentimos muito mais o efeito que *locking* tem sobre a execução do programa.

Mais ainda, ainda no assunto da *locked queue*, o número de *threads* utilizado foi 4, mas o maior *speedup* que tivemos em testes foi para 8 *threads*, sendo que aí a diferença já se notava, e era um *speedup* proporcional (aproximadamente duas vezes *speedup* em relação ao de quatro *threads*). Acreditamos que isto venha do facto que a máquina que utilizamos suporta oito *threads* por *hardware*.

Virando agora a nossa atenção para a segunda versão da construção paralela:

	Seq.	Async	BVH	Async + BVH	Async + BVH + Const. Paralela v.2	Locked Queue + BVH + Const. Paralela v.2
Original	9867.0	2271.0	4043.3	1034.0	958.3	1254.3
Mirror	9865.0	2271.3	4037.0	1030.7	958.7	1253.0
Sphere	463005.0	116098.0	3708.0	1078.0	1016.3	1147.3
Water	1482647.0	373939.0	4489.0	1528.3	1607.7	1512.7

Figure 8: *Async vs. Locked Queue (ms)*

Para todos os casos em que a construção paralela é aplicada, a versão que utiliza uma *locked queue* é melhor, mas não por uma larga margem. O maior problema desta implementação passa pelo facto que, em geral, a fila vai ter duas características:

- O *lock* é mantido por cada *thread* durante bastante tempo, a *queue* está bloqueada a maior parte do tempo;
- Há pouco trabalho na *queue* a cada momento do tempo, pois só são adicionadas duas tarefas de cada vez;

Ambos problemas afetam o funcionamento da paralelização, em particular o primeiro ponto, que obriga que a maior parte das *threads* estejam *idle* muito tempo, e leva a que o trabalho seja feito quase sequencialmente.

iii.2 *ISPC*

Para cada uma das várias figuras, os resultados que obtivemos com a adição de *ISPC*:

	ISPC
Original	4476
Mirror	4334
Sphere	150822,3
Water	N/A

Figure 9: T_{exec} *ISPC (ms)*

Os resultados referentes ao ficheiro *Water* não aparecem aqui pois tivemos problemas na sua execução e não tivemos tempo para os corrigir.

Em teoria, assumindo que nós temos oito *lanes SIMD* a tratar dos triângulos, o expectável seria que o *speedup* fosse linear, sendo portanto, de oito vezes, de acordo com a Lei de Amdahl. No entanto, isto não se verificou, por vários motivos:

- A funcionalidade da oclusão não foi implementada, o que significa que ainda existe um ponto do programa em que toda a *mesh* é verificada com um método de força bruta não vetorizado, o que se torna num *bottleneck* de *performance* relevante. O que nós esperamos seria que, se tivéssemos implementado oclusão, os ganhos obtidos estariam mais próximos dos teóricos;

- Apesar da complexidade da procura ser diminuída, a introdução do *ISPC* obrigou a mais trabalho de inicilização, verificação dos dados e escolha de resultados que não estava no código original. Quando olhámos para estes aspetos em separado não são muito preocupantes, mas quando os conjugamos o seu efeito é sentido;

- Na função *intersect_ triangle*, certos *inputs* na versão original levariam a uma terminação sem que toda a função fosse executada, uma vez que certos triângulos não passam do primeiro ou do segundo *check*. No entanto, como estamos a passar, para todos os efeitos, um vetor de oito triângulos, vamos ter de levar a execução da função até ao fim, uma vez que, mesmo que apenas um dos oito triângulos seja válido, todos os cálculos serão aplicados até aos triângulos inválidos até ao fim.

iii.3 *Ambient Occlusion*

Como dissemos anteriormente, não utilizamos *ambient occlusion* nos testes. Esta decisão veio do facto que todos os testes que fizemos enquanto desenvolvíamos o programa demonstraram que a inclusão de *ambient occlusion* adicionava *slowdown* consistente, no sentido que sim, como é óbvio, o tempo de execução aumenta, mas sempre de uma forma proporcional, uma vez que o trabalho que é adicionado, como nós vimos anteriormente, é determinístico e, em geral, igualmente distribuído pelos fios de execução.

iv. Observações

Globalmente, todas as otimizações que foram implementadas originaram algum *speedup*, proporcional ao esperado. No entanto, algumas destas implementações produziram resultados não tão bons, nomeadamente devido aos mecanismos de controlo de concorrência que tiveram de ser adicionados. Mais ainda, podemos ver que o excesso de criação de *threads*, tanto para executar *asyncs* como para *std::threads*, diminui severamente o potencial que uma implementação tenha para originar algum *speedup*.

V. Conclusão

Em geral, acreditamos ter conseguido alcançar a maior parte dos objetivos. Fomos capazes de fazer vários tipos de implementações e comparar os resultados que elas criaram em várias vertentes, explicando também a maior parte das consequências.

Alguns aspetos que nos falharam foi na implementação do *ISPC*, em que nos deparámos com vários problemas algébricos que nos impediram de proguerdir, e algumas das implementações que fizémos podiam ser um pouco mais limpas. Se tivéssemos tido mais tempo, teríamos gostado de fazer uma implementação mais completa do *ISPC*, especialmente de forma a combiná-lo com outras otimizações, como a *Locked Queue* ou as tarefas assíncronas.

VI. Referências

[1] *3D C++ Tutorials - CPU Ray Tracer*. URL: http://www.3dcpptutorials.sk/index.php?id=42.

[2] *Intel®ISPC User's Guide*. URL: https://ispc.github.io/ispc.html.

[3] *OMEN by HP 15-ce012np specifications*. URL: https://www.cpu-world.com/CPUs/Core%5C_i7/Intel-Core%5C%20i7%5C%20i7-7700HQ.html.

[4] *Parallel BVH Building*. URL: http://www.aaronperley.com/parallel-downsampling/.

[5] *Parallel Sorting Algorithms*. URL: https://www.cpp.edu/~gsyoung/CS370/14Sp/Parallel%5C%20Sorting%5C%20Algorithms%5C%20Matthew.pdf.

[6] *Ray-Box Intersection*. URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-box-intersection.

[7] *SeARCH 6 Cluster hardware description*. URL: http://search6.di.uminho.pt/wordpress/?page%5C_id=55.

VII. Anexo

i. Máquina Utilizada

Table 1: *Especificações*

Intel Core i7-7700HQ	
Performance	
Number of cores	4
Number of threads	8
Processor Base Frequency	2.8 GHz
Max Turbo Frequency	3.6 GHz
Memory Specifications	
L1 cache size	4 x 32 KB 8-way associative instruction caches
	4 x 32 KB 8-way associative data caches
L2 cache size	4 x 256 KB 4-way set associative caches
L3 cache size	6 MB 12-way set associative cache
Max Memory Size	64 GB
Max Memory Bandwidth	34.1 GB/s
Supported memory	DDR3L-1600, LPDDR3-2133, DDR4-2400
Memory Channels	2
Advanced Technologies	
Extensions	SSE4.1 + SSE4.2 / Streaming SIMD Extensions AVX2 (Advanced Vector Extensions 2.0)