Projet de raytracer parallèle

Xavier JUVIGNY

27 février 2017

1 But du projet

Le but du projet est d'accélérer un code effectuant le calcul d'une image de synthèse à partir d'un algorithme de raytracing. Pour simplifier le code, la scène ne peut contenir que des sphères, plus ou moins réflexives, plus ou moins transparentes.

L'accélération du code se fera à l'aide de la programmation parallèle :

1. Dans un premier temps, localement sur une machine où on essaiera d'exploiter tous les cœurs de calcul afin de minimiser le temps de restitution de la scène ;
2. Dans un deuxième temps, en distribuant le calcul de la scène sur un ensemble de machines reliées par un réseau intranet ;
3. Et enfin, pour les plus courageux d'entre nous, en essayant d'exploiter la carte graphique se trouvant sur la machine.

2 Présentation du programme de raytracing

La scène créée par le programme est statique, et définie dans la fonction principale du programme, où on tire au hasard une centaine de sphères englobées dans une sphère ciel et illuminées par une sphère de lumière.

La fonction `render` est la fonction permettant de rendre la scène initialisée auparavant dans le `main`. C'est cette fonction là qu'il faudra paralléliser et optimiser pour accélérer la restitution de la scène.

Le principe de l'algorithme est simple : on parcourt chaque pixel de l'image, et on calcul le point (ou la zone pour des raytracers un peu plus perfectionnés) correspondant qui appartient au plan de projection de la scène. On en déduit le rayon qui part de l'œil de la caméra et qui passe par ce point.

On cherche ensuite si ce rayon intersecte un des objets de la scène. Si ce n'est pas le cas, on se contente alors d'afficher la couleur du fond de l'image. Sinon, on calcul le point d'intersection et soit l'objet n'est ni transparent, ni réflexif, et dans ce cas, on calcul le rayon partant de ce point d'intersection et allant jusqu'à la source de lumière (éventuellement plusieurs rayons s'il y a plusieurs sources de lumière). Si ce rayon rencontre entre deux un obstacle, on ne l'illumine pas, sinon, on calcule sa couleur en fonction du degré d'incidence par rapport à la source lumineuse :

$$\text{col} = \text{couleur sphère} \times \text{transmission} \times \max(0, < \vec{n_h} | \vec{d} >) * \text{couleur lumière}$$

où $\vec{n_h}$ est la normale par rapport à la surface au point d'intersection du rayon lancé de la caméra, et $\vec{d}$ le vecteur directeur du rayon partant du point d'intersection et se dirigeant vers la source lumineuse.

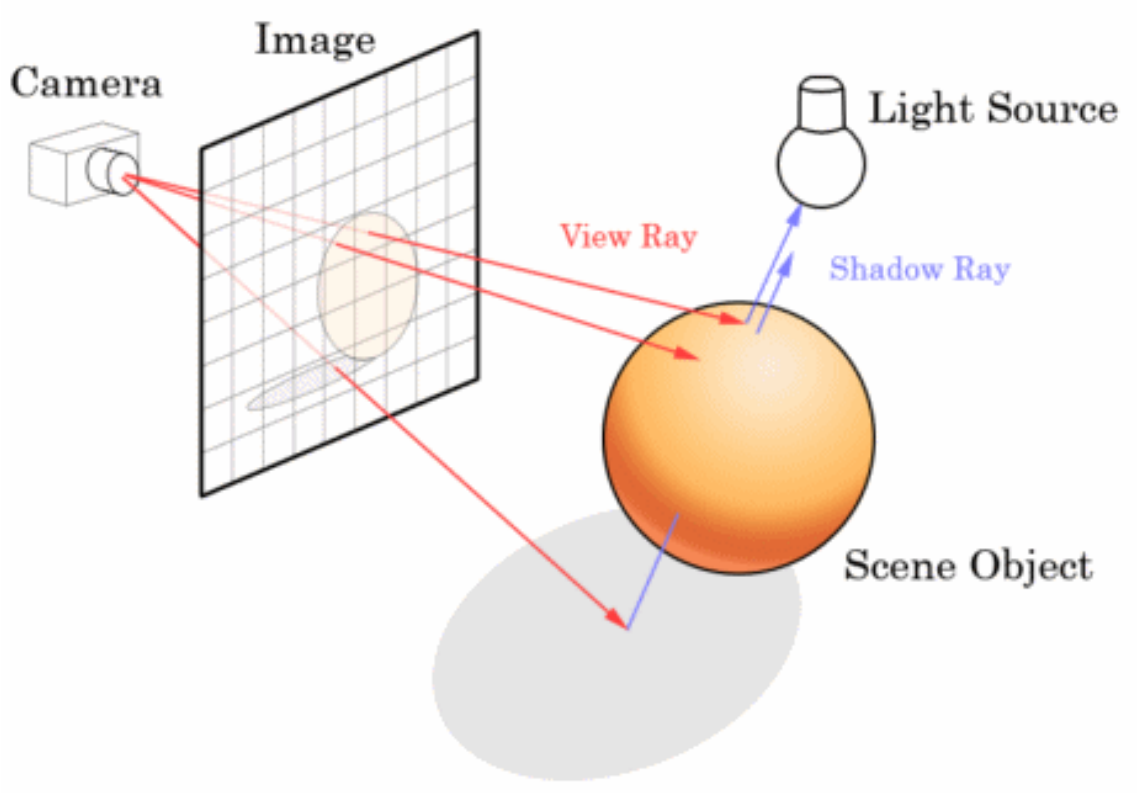

FIG. 1 : Représentation schématique du principe d'un raytracer.

Lorsqu'une sphère est transparente ou refléxive, le calcul est un peu plus compliqué, car il faut dans ce cas suivre le multiple rebond du rayon ou sa diffraction par la sphère transparente. Le calcul peut dans ce cas prendre bien plus de temps que si le rayon rencontre une sphère "mate".

3 Parallélisation

La parallélisation de ce code ne devrait pas poser trop de problème, vu que le calcul de chaque pixel de l'image est indépendante. On veillera cependant à plusieurs points :

- Veiller à ce que la granularité du parallèlisme soit suffisamment importante pour que les échanges de messages ou la gestion des threads ne soit pas pénalisante dans le temps de calcul ;
- Le temps de calcul pour chaque pixel varie grandement, sans qu'on puisse prévoir d'avance le temps pris. Il va falloir donc aussi bien en parallélisme local (avec les threads) qu'en parallélisme sur mémoire partagée (avec MPI) mettre au point une stratégie de "task farming" permettant un équilibrage dynamique des charges. Cet équilibrage pourra également être mis en pratique sur GPU pour ceux qui aimeraient le porter sur GPU.

4 MPI et les threads

Beaucoup de versions de MPI ne supportent pas les threads (et OpenMP) par défaut. Pour cela, il faut initialiser MPI avec la fonction suivante :

```
int MPI_Init_thread( int *argc, char ***argv, int required, int *provided )
```

où required peut prendre les valeurs suivantes :

1. MPI_THREAD_SINGLE : Seul le thread principal s'exécute. C'est le mode par défaut.
2. MPI_THREAD_FUNNELED : Le programme peut être multi-threadé, mais seul le thread principal pourra faire des appels à MPI.
3. MPI_THREAD_SERIALIZED : Le processus principal et ses threads peuvent appeler des fonctions MPI, mais seulement un à la fois. Les appels MPI ne peuvent se faire en parallèle et l'appel à des fonctions MPI par deux threads différents se feront l'un après l'autre.
4. MPI_THREAD_MULTIPLE : Les appels MPI peuvent être appelés par n'importe quels threads sans restrictions.

Le paramètre de sortie provided donne le niveau de support des threads le plus haut fourni par la bibliothèque utilisée.