Projet HPC : rendu « photo-réaliste » avec illumination globale

Version du 1er mars 2019

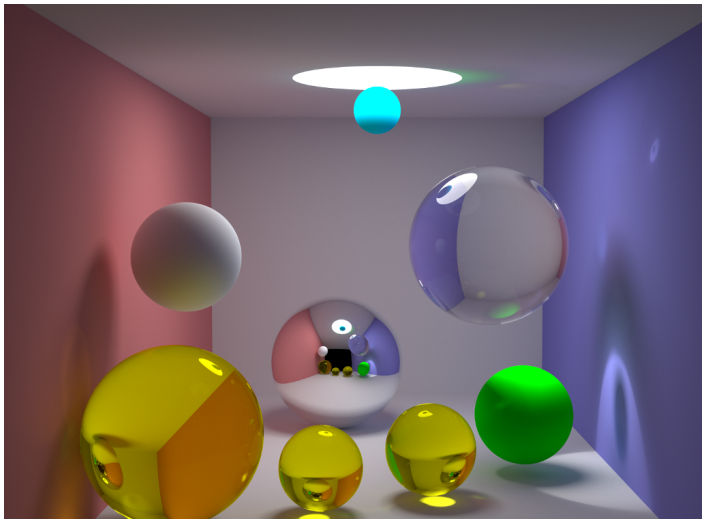

FIGURE 1 – Exemple d'image générée.

1 Présentation

Le but du projet est de paralléliser un programme séquentiel (que nous vous fournissons) qui effectue le rendu d'une image de synthèse (voir figure 1).

Dans ce programme, une *scène* fictive est décrite ; un appareil photo virtuel (*camera* en anglais) est positionné et orienté ; le code tente de reproduire la photo que prendrait l'appareil photo virtuel si la scène était vraiment présente dans le monde matériel. La scène fictive est constituée d'*objets* dont les propriétés sont décrites : ils peuvent être opaques, réfléchissants, transparents, colorés, etc. Pour que le tout soit visible, des sources de lumière fictives sont décrites.

Le rendu se veut photo-réaliste et simule quelques effets optiques non-triviaux (mieux que la plupart des jeux vidéos avec graphismes tri-dimensionnels, par exemple).

Il existe plusieurs techniques de rendu 3D. La plus simple et la plus ancienne consiste à dessiner des triangles texturés de l'arrière vers l'avant (cette technique est essentiellement utilisée par les cartes graphiques avec accélération 3D, et elle était explicitement utilisée dans de vieux jeux vidéos avant que de tels accélérateurs de calculs n'existent). Ceci ne permet pas de simuler facilement du verre, des miroirs, etc.

Le procédé du *ray-tracing* s'approche davantage d'une simulation photo-réaliste : il est possible de simuler des miroirs et des objets partiellement transparents, comme des loupes, des blocs de verre, des ombres portées, etc.

Cette technique part de l'idée que lorsqu'on prend un objet en photo, des rayons lumineux sont émis par l'objet et pénètrent dans le dispositif optique de la caméra avant de toucher un point donné du composant photosensible où il est « enregistré ». Si on voulait simuler le processus numériquement, il faudrait déterminer l'intensité lumineuse et la couleur d'une infinité de rayons entrants (on parle de la *luminance* du rayon).

Dans le *ray-tracing*, il s'agit de faire le processus dans l'autre sens : on simule des rayons qui partent de la caméra et sont dirigés vers la scène dont il faut faire le rendu. Pour chaque pixel de l'image à rendre, on « lance » un rayon vers l'extérieur et on détermine quelle luminance nous parvient le long de ce rayon (voir figure 2).

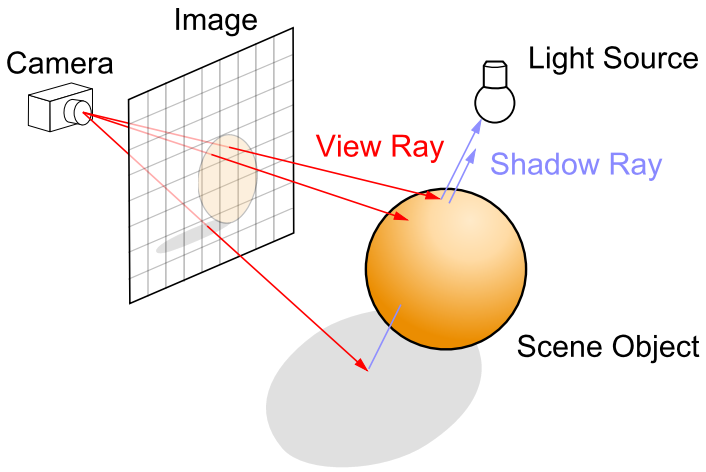

FIGURE 2 – Principe du *ray-tracing*.

Pour cela, on détermine quel objet le rayon touche en premier, et quel est le point d'intersection P. On détermine la couleur de l'objet à ce point précis (par exemple, si l'objet est texturé). A priori c'est la couleur perçue par la caméra, mais il faut prendre en comptes plusieurs phénomènes.

— Si c'est un objet réfléchissant, alors le rayon lumineux va « rebondir » dessus (phénomène physique de la réflexion) et aller toucher autre chose. On « lance » donc ce *rayon réfléchi*, qui part de P dans une nouvelle direction. Il s'agit de luminance qui vient d'ailleurs et qui éclaire le point P avant de rebondir directement vers la caméra.

— Si c'est un objet transparent, alors le rayon lumineux va pénétrer dedans (phénomène physique de la réfraction) et être dévié d'une manière qui dépend de l'indice de réfraction des milieux concernés. On lance donc le *rayon réfracté* qui part de P et traverse l'objet (soit vers l'intérieur, soit vers l'extérieur).

— Généralement, un objet transparent est aussi réfléchissant ! Donc en fait il faut tirer *deux* rayons à partir du point d'intersection et additionner les deux luminances (celle qui vient du rayon réfracté et celle qui vient du rayon réfléchi).

— Enfin, dans tous les cas, et surtout dans le cas d'un objet opaque, ni transparent ni réfléchissant, il faut prendre en compte la lumière qui arrive au point P en provenance d'ailleurs, et notamment des sources lumineuses (en effet, si on les ignore, toute la scène sera plongée dans le noir). C'est là que les choses se compliquent sensiblement : il faudrait pouvoir prendre en compte de la lumière qui peut venir de n'importe où, après avoir été réfléchie, réfractée, etc.

La technique généralement appelée le *ray-tracing* consiste à déterminer la lumière en provenance directe des sources lumineuses (en ligne droite, sans obstacle), et à ignorer tout le reste. On parle alors « d'illumination directe ». Cela permet de rendre des effets comme les ombres portées (les objets font de l'ombre), mais ne permet pas de rendre certains phénomènes physiques :

— une loupe ne va pas « concentrer la lumière » en un point ;

— un miroir ne va pas réfléchir la lumière ;

— les matériaux opaques colorés « n'irradient » pas de la lumière de leur couleur.

A contrario, les techniques dites « d'illumination globale », aussi appelées *path-tracing* consistent à tenter d'approcher la lumière qui arrive en un point en provenance du reste de la scène par un processus de Monte-Carlo.

Pour calculer la couleur d'un pixel de l'image, on lance non pas un seul, mais un grand nombre de rayons. Lorsqu'un rayon touche une surface opaque, on lance un rayon depuis le point d'intersection dans une direction *aléatoire*, et on regarde quelle quantité de lumière arrive par là. Ce rayon va peut-être heurter un autre objet, et repartir dans une autre direction aléatoire, etc. Le processus s'arrête au bout d'un nombre prédéfini de rebonds.

Ce sont ces techniques « d'illumination globale » que nous allons étudier dans ce projet.

Pour en savoir plus :

— sur le ray-tracing :
`https://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html` (en anglais).

— sur le ray-tracing :
`http://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview`

— sur le path-tracing :
`https://www.scratchapixel.com/lessons/3d-basic-rendering/global-illumination-path-tracing`

2 Description du code séquentiel

Le code séquentiel est constitué d'un seul fichier `pathtracer.c`. La scène dont il faut effectuer le rendu est constituée uniquement de sphères (!) qui sont décrites par la variable `spheres[]`. Chaque sphère est caractérisée par son rayon, les coordonnées 3D de son centre, son émissivité (émet-elle de la lumière et si oui de quelle couleur ?), la couleur de sa surface et la nature de l'objet (réfléchissant, transparent, opaque).

Un *rayon* est donné par son point d'origine O et son vecteur direction $\vec{d}$. L'ensemble des points situés sur son parcours est $O + t \cdot \vec{d}$, pour $t > 0$. Si $\|\vec{d}\| = 1$, alors t est la distance entre O et ce point.

Voici un descriptif rapide des différentes fonctions :

— `intersect()` étant donné un rayon, détermine s'il y a une sphère sur son chemin, et si oui, quelle est la première rencontrée.

— `radiance()` étant donné un rayon, détermine la luminance (c.a.d. l'intensité lumineuse en rouge/vert/bleu) en provenance de la scène jusqu'au point d'origine du rayon, dans la direction *opposée* (si le rayon pointe vers un objet, c'est la luminance en provenance de cet objet qui est renvoyée). Cette fonction est récursive et randomisée.

Un micro-BLAS de niveau 1 est fourni, et il sert exclusivement à traiter des vecteurs de taille 3 (il peut s'agir de coordonnées 3D ou bien de couleurs en RGB).

L'état interne d'un générateur pseudo-aléatoire standard est passé en argument aux fonctions qui en ont besoin.

La boucle principale effectue un sur-échantillonnage 2×2 : chaque pixel de l'image est découpé en quatre sous-pixels, dont la couleur est calculée séparément. Enfin, les quatre valeurs sont moyennées. Pour chaque sous-pixel, un nombre déterminé de rayons sont lancés et la couleur obtenue est la moyenne de toutes les luminances renvoyées.

Pour obtenir un rendu de bonne qualité, il faut beaucoup d'échantillons par sous-pixel (disons 5000). Du coup, c'est très long.

Deux cas tests vous sont donnés dans le code : un « petit » cas test pour la mise au point de vos programmes ; un « gros » cas test qui devra être utilisé par défaut pour vos tests de performance finaux. Vous pouvez au besoin utiliser d'autres cas tests pour mettre en valeur les performances de votre code, mais il faudra bien présenter ces cas tests et justifier leur utilité.

L'image finale est écrite dans le répertoire `/tmp/votre_login/`. Vous pouvez la visionner par exemple avec la commande `display` d'Image Magick.

3 Travail à effectuer

Le travail à effectuer se décompose en 2 parties.

3.1 Partie 1 : parallélisation MPI

Pourquoi un équilibrage de charge dynamique est justifié pour cette application ?

Mettez en place un équilibrage de charge dynamique de type auto-régulé en mode multi-processus avec MPI.

Vous pourrez étudier les performances de votre programme parallèle pour (au moins) les trois configurations suivantes (toujours au sein d'une même salle de TP, avec au plus 16 noeuds à chaque fois) :

— 1 processus par noeud ;

— plusieurs processus par noeud avec 1 processus par coeur physique ;

— plusieurs processus par noeud avec 1 processus par coeur logigue.

3.2 Partie 2 : parallélisation MPI+OpenMP et SIMD

3.2.1 Parallélisation MPI+OpenMP

La fonction `erand48()` vous semble-t-elle adaptée à une utilisation en mode multi-thread ?

Mettre en place une parallélisation hybride MPI+OpenMP permettant d'exploiter les processeurs multicœur dont vous disposez. On pourra alors comparer les performances d'une implémentation "MPI+OpenMP" et d'une implémentation "MPI pur" (sans multi-threading), et ce **pour un même nombre de cœurs CPU**.

3.2.2 Parallélisation SIMD

A l'aide de directives OpenMP et/ou de fonctions intrinsèques, vectorisez votre code. On pourra commencer par s'intéresser à la vectorisation de la boucle sur les échantillons (avec une directive OpenMP), puis si nécessaire à des fonctions de plus bas niveau.

Si vous avez implémenté la vectorisation avec les fonctions intrinsèques et avec les directives OpenMP, quelle est la différence de gain de performance (par rapport à une exécution scalaire) entre ces deux implémentations ?

On pourra d'abord mesurer le gain de performance offert par les unités SIMD pour un code séquentiel, avant de le mesurer pour votre meilleur code parallèle.

4 Travail à remettre

Pour chacune des deux parties, vous devrez remettre le code source, sous la forme d'une archive `tar` compressée et nommée suivant le modèle `projet_HPC_nom1_nom2.tar.gz`. L'archive ne doit contenir ni exécutable ni image générée, et les différentes versions demandées devront être localisées dans des répertoires différents. Chaque répertoire devra contenir un fichier `Makefile` : la commande `make` devra permettre de lancer la compilation, et la commande `make exec` devra lancer une exécution parallèle représentative avec des paramètres appropriés. Un fichier `Makefile` situé à la racine de votre projet devra permettre (avec la commande `make`) de lancer la compilation de chaque version.

A la fin du projet, vous devrez remettre un rapport au format `pdf` (de 5 à 10 pages, nommé suivant le modèle `rapport_HPC_nom1_nom2.pdf`) présentant vos algorithmes, vos choix d'implémentation (sans code source), vos résultats (notamment vos efficacités parallèles) et vos conclusions pour les deux parties. L'analyse du comportement de vos programmes sera particulièrement appréciée. Les machines n'étant pas strictement identiques d'une salle à l'autre, on précisera dans le rapport la salle utilisée pour les tests de performance.

5 Quelques précisions importantes

- Le projet est à réaliser par binôme.
- Pour les étudiants en master informatique, vous **devez** lire le document intitulé « Projet HPC : conditions d'utilisation d'OpenMPI ». Vous veillerez notamment à n'utiliser qu'une salle à la fois pour vos tests, et vous n'oublierez pas de tuer **tous** vos processus sur **toutes** les machines utilisées à la fin de vos tests.
- Le code de la première partie («Parallélisation MPI»), accompagné des slides de votre soutenance au format `pdf` (et donc sans animations) (prévoir une soutenance de 10 minutes, suivie de 5 minutes de questions), est à remettre au plus tard le dimanche 07 / 04 / 2019 à 23h59 (heure locale). Les soutenances de présentation de la première partie auront lieu lors de la séance de TDTP du lundi 08 / 04 / 2019 ou du mardi 09 / 04 / 2019.
  Le code de la seconde partie («Parallélisation MPI+OpenMP et SIMD») et le rapport final sont à remettre au plus tard le dimanche 12 / 05 / 2019 à 23h59 (heure locale).
- Les remises se feront par courriel à :
  - `Charles.Bouillaguet@univ-lille.fr` pour les étudiants de MAIN,
  - `cmakassikis@aneo.fr` et `smoustafa@aneo.fr` pour les étudiants du master informatique.
- En cas d'imprévu ou de problème technique commun, n'hésitez pas à nous contacter pour que nous puissions vous proposer une solution ou une alternative.