TP HPC 3

Master Informatique - 1ère année

Année 2020-2021

L'objectif de ce TP est de poursuivre l'étude de la parallélisation multi-threadée et plus particulièrement, d'étudier la manière de mettre en place un équilibrage dynamique des calculs. Il s'effectuera au travers de la parallélisation de l'application de lancer de rayons, vue dans le module d'initiation à la recherche.

Préliminaires

Récupérez la version initiale des fichiers sources, disponibles dans l'archive jointe à ce TP. Celle ci-contient un sous-dossier nommé `V1` qui sera utilisé pour le développement d'une version avec équilibrage statique. Vous serez amené à créer un second sous-dossier `V2` pour la version avec équilibrage dynamique.

Partie 1 - Equilibrage statique

En lancer de rayons, les calculs effectués pour un pixel sont totalement indépendants des autres pixels. Cet algorithme est donc une cible potentiellement aisée pour une parallélisation ...

La première approche qui sera étudiée consistera à découper l'image à calculer en zones indépendantes, chaque zone étant ainsi affectée à un thread différent. La découpe sera faite par bande horizontale disjointes (voir figure 1).

FIGURE 1 – Découpage de l'image en bandes, chaque bande étant affectée à un thread différent.

Le calcul séquentiel de l'image est actuellement géré par la méthode `camera::genererImage()`. Cette méthode suppose que la caméra est positionnée aux coordonnées $(0, 0, 2)$ et que l'écran se trouve dans le plan Oxy (voir figure 2).

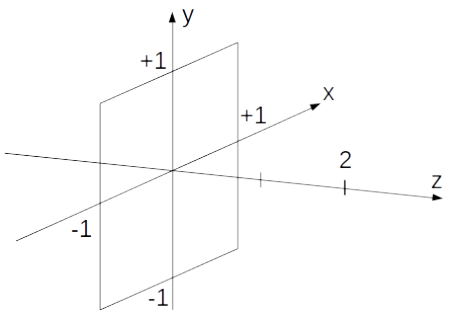


FIGURE 2 – Position de la caméra..

La méthode calcule alors les coordonnées du centre de chaque pixel, génére un rayon issu de la position de la source vers ce centre et déclenche les calculs induits par l'algorithme de lancer de rayons. La couleur du pixel de l'image est ensuite mise à jour lors de sa récupération.

Exercice 1

Ajouter la méthode suivante :

```
void genererImageParallele(const Scene& sc, Image& im,
                           int profondeur, int nbThreads)
```

à la classe `Camera`. Ses trois premiers paramètres sont les mêmes que pour la méthode `genererImage()`, tandis que le quatrième représentera le nombre de threads à utiliser pour le calcul. Dans un premier temps, cette méthode se contentera de générer les bandes nécessaires les unes après les autres et de les afficher pour vérification à l'écran. Une structure `zone` a été définie à cet effet dans le fichier `Camera.hpp`. Vous gérerez le fait que la hauteur de l'image ne sera pas forcément un multiple du nombre de bandes ...

Exercice 2

Complétez la classe `Camera` avec la méthode **statique** suivante :

```
static void calculeZone(const Scene &sc,
                        Image &im, int profondeur,
                        const zone &area, const Point &position)
```

avec :
- `sc`, `im` et `profondeur` les paramètres identiques à ceux définis dans la méthode `genererImage()` ;
- `area` les informations concernant le zone de pixels à calculer ;
- `position` les coordonnées de la position de la caméra.

Cette méthode se contentera, à ce stade, d'afficher les informations concernant la zone dont elle a la charge.

Modifiez ensuite la méthode `genererImageParallele` de telle sorte qu'elle lance les `nbThreads` threads, chacun d'entre-eux exécutant une version différente de `calculeZone`. Modifiez enfin l'appel à la fonction de calcul de l'image dans le module `lr.cpp`, en notant qu'une constante nommée `NBT` y a été définie pour spécifier le nombre de threads à utiliser.

Testez le fonctionnement de votre application avec un nombre de threads variable.

Remarque : le fichier `CMakeLists.txt` a été modifié pour pouvoir prendre en compte l'utilisation des threads. Consultez les modifications qui ont été effectuées, qui pourront vous être utiles pour des développements ultérieurs.

Exercice 3

En vous inspirant du code de la méthode `genereImage()`, complétez la méthode `calculeZone()` afin qu'elle effectue le calcul de la bande de l'image qui lui a été attribuée. A la fin de son exécution, la méthode devra afficher l'*id* du thread qui l'exécute, suivi du temps que lui a pris le calcul des pixels. Vous réfléchirez aux parties du code qui nécessitent une protection par mutex... Testez votre application avec un nombre de threads variable.

Partie 2 - Equilibrage dynamique

Comme vous avez pu le constater sur les résultats obtenus par l'approche précédente, les temps d'exécution de chaque thread sont particulièrement déséquilibrés, ce qui nuit à l'efficacité de la parallélisation. Ceci provient du fait que le coût de calcul de chaque pixel n'est pas le même et que certaines zone peuvent, par conséquent, avoir un coût de calcul beaucoup plus important que d'autres. Il n'existe cependant pas de règle claire permettant d'évaluer *a priori* le coût d'une zone, et obtenir une parallélisation plus efficace en lancer de rayons passe par une approche dynamique de la répartition des tâches de calcul.

L'approche que vous allez développer dans cette seconde partie est assez classique en synthèse d'image ; elle consiste à :
- découper l'image en petites zones, généralement carrées (voir figure 3a) ; l'ensemble de ces zones recouvrent l'image et les zones n'ont pas de recouvrement ;
- affecter dynamiquement une zone à un thread sur sa demande (voir figure 3b) :
  - au démarrage, un thread demande une zone à calculer ;
  - lorsque la zone courante a été calculée, le thread demande une nouvelle zone ;
  - le thread se termine lorsqu'il est informé qu'il n'y a plus de zones disponibles.

L'idée est alors de s'assurer que les différents threads ont en permanence une tâche à accomplir, indépendamment de la complexité de ces tâches, équilibrant ainsi leur charge au cours du temps. A noter que ce principe ne peut fonctionner que s'il y a beaucoup plus de zones à calculer que de threads disponibles.

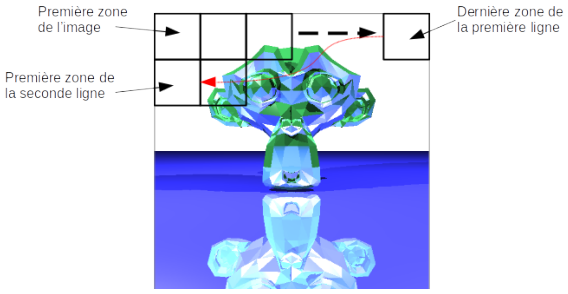
Exercice 1

Après avoir créé le dossier `V2` pour les développements liés à cette seconde version, recopiez-y les fichiers sources de la première version. Seules quelques parties du code devront être modifiées.

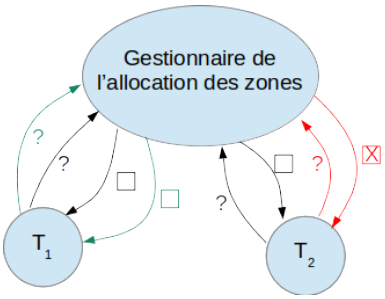
Ajoutez, à la classe `Camera`, la méthode **statique** suivante :

```
static zone zoneSuivante(const Image &im)
```

L'objectif de cette méthode est de fournir, **à chaque appel**, une zone de calcul différente de l'image passée en paramètre. Une manière de faire, illlustrée sur la figure 3a, est de commencer par la zone en haut à gauche de l'image, puis de balayer l'image par ligne horizontale de zones.



(a) Ordre et positions des zones générées.



(b) Principe de la demande d'une zone à traiter.

FIGURE 3 – Schéma général de fonctionnement de l'équilibrage dynamique. (a) Une manière simple de générer les différentes zones à traiter, en les générant par bandes horizontales successives ; (b) Mode de fonctionnement des threads pour l'équilibrage dynamique : chaque thread demande une zone à traiter à un gestionnaire, jusqu'à ce que plus aucune zone ne soit disponible.

On donne les indications suivantes :

- Pour pouvoir mémoriser l'emplacement de la prochaine zone à renvoyer lors de l'appel suivant, vous utiliserez des variables **statiques** [1] locales à cette fonction.
- La taille d'une zone sera fixée par des constantes `LARGZONE` et `HAUTZONE`, pour lesquelles vous pourrez prendre la même valeur de 32 pour cette première version. Vous prendrez cependant garde au fait que la largeur de l'image, ainsi que sa hauteur, ne sont pas forcément des multiples des valeurs définies dans ces constantes ... Il faudra donc en tenir compte lors de la génération (i) de la dernière zone d'une ligne et (ii) lors de la génération des zones de la dernière ligne.
- Lorsque toutes les zones auront été générées par la fonction, elle renverra une zone spéciale, contenant des valeurs spécifiques permettant de déterminer que plus aucune zone n'est disponible. De nombreux choix sont disponibles, parmi lesquels une valeur de coordonnée négative ou une dimension nulle.

Lorsque le code de la méthode aura été écrit, vous le testerez en faisant afficher, à la suite de chacun de ses appels, la liste des zones générées, jusqu'à ce que plus aucune zone ne soit disponible. Vérifiez bien la compatibilité de votre code avec une dimension d'image qui n'est pas un multiple de la dimension des zones ...

Exercice 2

Dans la première version parallèle, chaque thread exécute la fonction `calculeZone`, qui reçoit parmi ses paramètres la zone de calcul qui lui est dévolue. Dans cette nouvelles version, la zone n'est pas connue à l'avance, mais doit être demandée par le thread à la méthode `zoneSuivante`. Après avoir supprimé le paramètre inutile, modifiez le code de `calculeZone` de telle sorte qu'elle demande une zone à calculer et effectue le calcul correspondant jusqu'à ce que sa dernière demande débouche sur la zone indiquant la fin des calculs (voir figure 3b). Vous réfléchirez aux zones de code qui ne peuvent être exécutées qu'en exclusion mutuelle.

1. On rappelle qu'en C/C++, ces variables sont créées lors du premier appel de la fonction, qu'elles sont conservées tant que l'application n'est pas terminée et que les valeurs qu'elles avaient à la fin de l'appel n sont retrouvées au début de l'appel $n+1$.

Exercice 3

Evaluez votre application en faisant apparaître en particulier le temps de calcul total de chaque thread, ainsi que le nombre de zone qu'il aura calculé. Vous effectuerez cette évaluation pour différentes tailles d'images et un nombre de threads variables (inutile d'aller plus loin que le nombre de coeurs virtuels dont vous disposez ...)