Programmation répartie. Module 4102C
TP 2 : application des threads : encore mieux que MG3D ! année universitaire 2020-21

Samuel Delepoulle – Franck Vandewiele – Joannick Gardize

février 2021

1 But du TP

Le but de ce TP est d'utiliser les threads pour optimiser l'exécution d'un lancer de rayon. Dans ce TP, le lancé de rayon n'est utilisé que comme illustration d'un processus coûteux afin de montrer comment il peut être réparti sur plusieurs threads. Les sources de base de l'algorithme sont fournies

2 L'algorithme du lancer de rayon

L'algorithme du lancer de rayon permet de calculer des images en se basant sur un parcours inverse de la lumière dans une scène : on calcule le parcours d'un rayon provenant de la caméra. Pour plus de détails se référer à : https://fr.wikipedia.org/wiki/Ray_tracing.

Le fonctionnement de l'algorithme est de calculer le trajet d'un rayon issu de la caméra et de trouver ses points d'intersection avec les objets de la scène. Pour chaque point, on calcule ensuite sa couleur qui résulte de :

- l'éclairage direct des sources de lumière (il faut calculer si la source est visible via un rayon d'ombrage) ;
- l'éclairage indirect par les autres objets (la lumière diffusée par les objets voisins).

Ce calcul peut être gourmand en ressources.

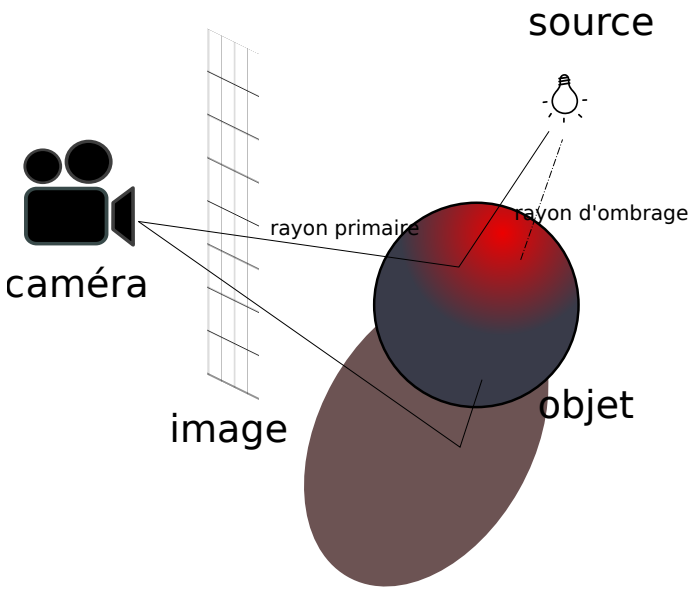

FIGURE 1 – principe du lancer de rayons

Présentation du problème de l'antiAliasage

Dans la version de base du lancer de rayon, on parcourt la grille de pixels et pour chaque pixel, on lance un rayon dont on calcule la couleur. Ceci produit un problème d'aliasage (ou de crénelage ou "pixellisation") visible dans certaines zones comme les contours des objets ou des ombres. Voir par exemple les figures 3, 4 et 5.

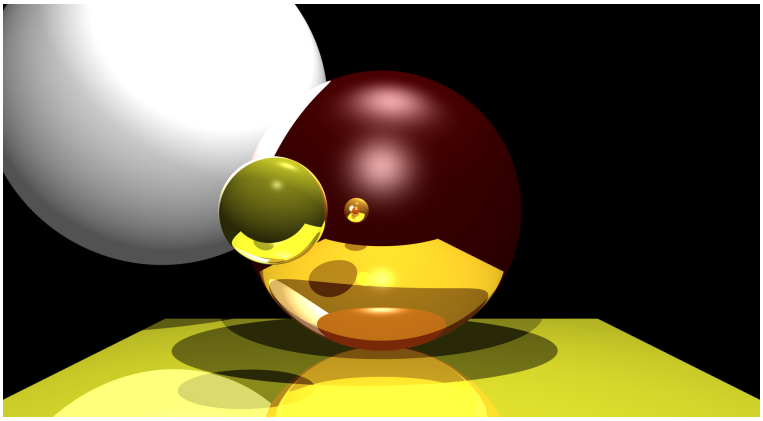

FIGURE 2 – Exemple d'image produite

Plusieurs techniques peuvent réduire ce problème. L'une d'entre elles consiste à calculer une moyenne d'intensité de n rayons lancés au hasard à travers un pixel (et non le centre du pixel).

FIGURE 3 – 1 rayon

FIGURE 4 – 4 rayons

FIGURE 5 – 100 rayons

3 LRjava

3.1 Source en java

Les sources d'un lancer de rayon écrit en java sont données, les sources sont sur github : https://github.com/delepoulle/LRjava

Pour télécharger les sources, se placer dans le dossier du TP et :

```
$ git clone https://github.com/delepoulle/LRjava
```

Créez le répertoire pour le bytecode, déplacez vous dans le répertoire des sources et compilez le lancer de rayons :

```
$ cd LRjava
$ mkdir bin
$ cd LRjava/src
$ javac lr/format/simple/*.java lr/format/*.java lr/*.java -d ../bin
```

3.2 Test d'exécution

Déplacez vous dans le répertoire d'exécution, copiez-y la scène exemple et lancez l'exécution.

```
$ cd ../bin
$ cp ../src/lr/simple.txt .
$ java lr.LR
```

Si l'exécution s'est passée normalement, vous devriez obtenir un fichier `image2.png` dans votre répertoire.

Observez l'impact du nombre de rayons sur votre image et sur le temps de rendu de l'image. Pour les tests, vous choisirez un nombre de rayons qui permet un calcul de rendu de l'ordre d'une minute (approximativement).

4 parallélisation première version : découpage statique

La classe `LR` comprend une boucle principale qui effectue le rendu de l'image ligne par ligne (méthode `renderLine`).

On souhaite pouvoir confier à un thread la tâche de rendre une partie de l'image pour, par exemple, utiliser quatre threads qui calculent chacun le quart de l'image.

Pour cela :

1. Créez une classe `ParallelRenderer` qui implémente l'interface `Runnable`. Sa méthode `run()` consistera à calculer les lignes de l'image comprises entre un minimum et un maximum (prévoyez les méthodes pour pouvoir fixer ces valeurs).
2. Modifiez la classe `LR` pour que l'image soit produite par plusieurs thread.
3. Attention : pour que l'image puisse être sauvegardée, il est impératif que tous les threads aient terminé la production de la partie d'image. Prévoyez donc ce test.
4. Comme dans le TP précédent, il est impératif d'utiliser une instance locale de générateur aléatoire faute de quoi les différents threads doivent utiliser le générateur du système qui est synchronisé. Modifiez la classe `Renderer` pour cela.
5. Recherchez le nombre optimal de threads pour cette tâche. Peut-on le trouver automatiquement ? Pourquoi ?
6. A l'aide de votre système d'exploitation, observez l'activité des différents cœurs de calcul. Les différents threads se terminent-ils en même temps ?

5 parallélisation deuxième version : découpage dynamique

Le principal problème de l'approche précédente est que la tâche confiée à chaque thread est fixée à l'avance. Dans le cas où chaque tâche ne prend pas le même temps, cette répartition n'est pas optimale. Il est possible d'optimiser la répartition en utilisant un découpage dynamique. Pour cela, on utilise des pools de threads.

Lisez la documentation sur les pools de threads et le framework `Executor` à cette adresse : https://www.jmdoudoux.fr/java/dej/chap-executor.htm

Utilisez maintenant un pool de threads pour effectuer le rendu de l'image. Quel gain constatez vous ?