# TME 6 : Pool de Thread

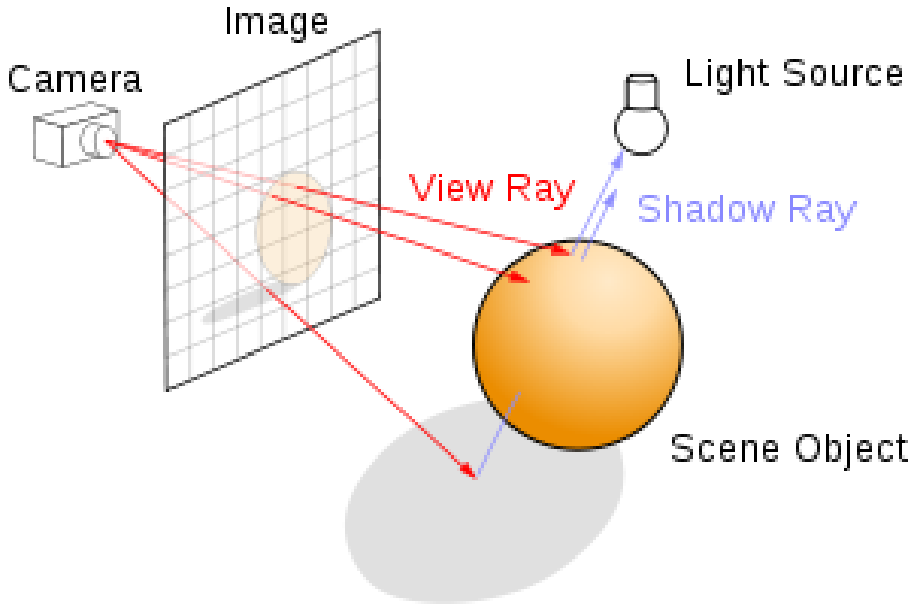
**Objectifs pédagogiques :**

- **accélération d'une application**
- **pool de thread**

## 1.1 Objectif

On vous fournit le code d'un Ray tracer très basique, qui sait dessiner des scenes représentant des Spheres colorées avec un éclairage.

Pour cela, le code calcule la couleur de chaque pixel de l'image à l'aide d'un rayon tiré de l'observateur (la caméra) vers les points de l'écran. On a pour cela actuellement une double boucle imbriquée qui calcule la couleur des pixels pour chaque position dans l'écran.




Votre mission est de paralléliser ce code. A priori la tâche est assez facile : la couleur de chaque pixel peut tout à fait être calculée en parallèle. Le calcul de la couleur nécessite un accès en lecture seule sur l'état de la scene qui contient les sphères.

## 1.2 Mise en place

Faire un "fork" ou un import du projet (comme décrit sur moodle): `https://github.com/yanntm/pure-java-raytracer` pour obtenir une copie du dépot de sources contenant le Ray Tracer à améliorer. Ce ray tracer est un exemple relativement pédagogique, développé par Carl von Bonin. Il le décrit également sur sa chaine Youtube si vous souhaitez en savoir plus.

**Question 1.** Cloner ensuite votre fork du projet dans votre espace de travail. Lancez le programme principal. Admirez le rendu. Pour la suite certaines questions nécessitent du code et d'autres des réponses en texte ou des mesures; vous rédigerez vos réponses dans "rapport.txt" (ou .md si vous préférez) que vous placerez dans votre dépot.

L'objectif est de paralléliser l'algorithme faisant le rendu de l'image. On utilisera la version sans post-processing, plus simple. C'est à dire la méthode
`public static void renderScene(Scene scene, Graphics gfx, int width, int height, float resolution)`
de la classe `carlvbn.raytracing.rendering.Renderer`.

```java
/** Renders the scene to a java.awt.Graphics object
 * @param scene The scene to Render
 * @param width The width of the desired output
 * @param height The height of the desired output
 * @param resolution (Floating point greater than 0 and lower or equal to 1) Controls the
      number of rays traced. (1 = Every pixel is ray-traced)
 */
public static void renderScene(Scene scene, Graphics gfx, int width, int height, float
    resolution) {
   int blockSize = (int) (1 / resolution);
   long start = System.currentTimeMillis();


   for (int x = 0; x<width; x+=blockSize) {
      for (int y = 0; y<height; y+=blockSize) {
         float[] uv = getNormalizedScreenCoordinates(x, y, width, height);
         PixelData pixelData = computePixelInfo(scene, uv[0], uv[1]);

         gfx.setColor(pixelData.getColor().toAWTColor());
         gfx.fillRect(x, y, blockSize, blockSize);
      }
   }

   System.out.println("Rendered in " + (System.currentTimeMillis() - start) + "ms");
}
```

**Question 2.** Identifiez les principaux éléments de cette fonction. Jetez un coup d'oeil à l'API de Graphics, une classe du JDK, ainsi que celle de Scene qui est fournie avec le ray tracer. En quoi consiste la double boucle qui est le corps de cette fonction ?

**Question 3.** On admet que dans la tâche de ray tracing, la couleur de chaque pixel de l'image peut-etre calculée indépendamment (donc en concurrence). On souhaite donc étudier diverses façon de paralléliser ce code. En première approximation on va créer un Runnable par pixel de l'image. Modifiez le code de `renderScene` pour instancier un Runnable par pixel de l'image, puis un thread pour contenir ce Runnable et le start. On attendra que tous les threads soient finis avant de revenir de la fonction. On pourra écrire cette version dans une copie "renderScene2" de la fonction, et renommer pour tester la version souhaitée.

NB: Créer trop de threads va écrouler le système; on recommande dans cette version de commencer par baisser la résolution, e.g. poser `resolution = 0.1f;` (ou même moins) dès le début de la fonction.

**Question 4.** Quel problème se pose dans l'accès à la variable nouvellement partagé "gfx" ? Comment l'avez vous résolu ?

**Question 5.** Quel est le problème posé par le nombre de threads engendrés ? Construisez une version qui n'instancie qu'un thread par colonne au lieu d'un par pixel (i.e. embarque dans son "run" la boucle interne). Comparez le temps de rendu de ces trois approches (sans thread, un par pixel, un par ligne).

**Question 6.** Créez la classe ThreadPool et son opération "submit(Runnable r)" en suivant les indications du TD. On pourra s'appuyer sur une `java.util.concurrent.ArrayBlockingQueue<E>` plutôt que notre file bloquante "maison". Ajoutez une occurrence `static` du pool de thread dans la classe Renderer; on testera plusieurs dimensionnements (taille de la file, nombre de threads instanciés).

**Question 7.** Ajoutez l'utilisation d'une barrière ou d'un latch pour permettre au thread "principal" d'attendre la fin du rendu. Donc il faut instancier le Latch avant les threads, les threads signalent le latch, le thread principal attend que tout soit fini. De nouveau, on s'appuiera plutôt sur la version du JDK `java.util.concurrent.CountDownLatch` que sur une version "maison". Mesurez les performances, avec une file relativement grande e.g. au moins 1000, et un nombre de threads variant entre un seul; environ le nombre de core de votre machine; deux fois plus de thread que de core ;

200 threads.

**Question 8.** Il y a une forte contention actuellement, à cause du traitement que l'on a été obligé de faire autour la manipulation de Graphics. La classe Renderer contient une version sans synchronisations d'une fonction permettant de peindre un rectangle avec une couleur au sein d'une image : `public static void fillColorRect(BufferedImage image, int x, int y, int width, int height, Color color)`. Elle prend une image en paramètre, pas un Graphics; pour s'adapter à cette API on va faire :

```
BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);
..boucles/pool etc mais en utilisant l'image..
..fin du dessin/latch..
gfx.drawImage(image, 0, 0, null);
```

Réecrivez le code de votre fonction render pour utiliser ce mécanisme et (re)mesurez les performances.

**Question 9.** Au lieu d'utilisez votre pool de threads "maison", utilisez un `private static ExecutorService exec = Executors.newFixedThreadPool(4);`. Il possède également une méthode "submit" donc la substitution devrait être relativement aisée. Mesurez les performances.

**Question 10.** Essayez d'initialiser votre Executor avec d'autres éxecuteurs fournis par Executors. On essaiera `newSingleThreadExecutor`, `newFixedThreadPool`, `newThreadPerTaskExecutor`, `newVirtualThreadPerTaskExecutor` qui respectivement correspondent à "un seul thread fait tout", "pool de thread, taille passée à la construction", "un thread par tâche soumise" (Ouch !), "un thread *virtuel* par tâche" (projet Loom, threads légers). Vous pouvez en essayer d'autres si le temps le permet. Concluez sur la meilleure configuration que vous avez trouvé pour améliorer le nombre de frames par seconde.