

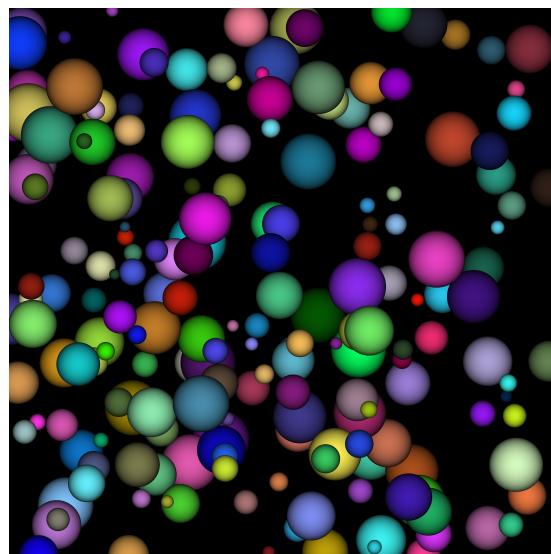
## THREADS POSIX - RAY TRACING

Le but de ce TP est d'avoir un programme parallèle qui permet de faire du *ray tracing* sur un ensemble de sphères.

### Ray Tracing

Le *ray tracing* est une technique de calcul d'optique par ordinateur, qui consiste à simuler le parcours inverse de la lumière, c'est-à-dire qu'on calcule les éclairages de la caméra vers les objets puis vers les lumières, alors que dans la réalité la lumière va de la scène vers l'œil. Plus précisément, pour chaque pixel de l'image à générer, cela consiste à lancer un rayon depuis le point de vue dans la scène 3D (la caméra). Le premier point d'impact du rayon sur un objet définit l'objet concerné par le pixel correspondant.

Dans le contexte de ce TP nous verrons une version très simplifiée du *ray tracing* qui consiste à regarder la scène du dessus et où nous ne considérons pas la réflexion. La figure suivante donne un exemple :



Dans un premier nous allons fixer la taille de la scène à  $\text{DIM} \times \text{DIM}$  :

```
1 #define DIM (4096)
```

Ensuite il faut générer de manière aléatoire un ensemble de sphères.

```
1 #define NB_SPHERE (100)
2
3 typedef struct {
4     float x, y, z;
5     float r, g, b;
6     float radius;
7 } sphere;
8
9 void generateRandomSpheres(sphere *spheres, unsigned nbSphere, unsigned dim);
```

Lors de la génération il faut bien faire attention à ce que les sphères ne s'intersectent pas. Il faut ensuite une structure pour stocker chaque pixel de la scène. La scène sera quant à elle représentée dans une matrice (en représentation en ligne par exemple).

```

1 #include <sys/types.h>
3 typedef struct {
4     u_int8_t r, g, b;
5 } pixel;
7 pixel *images;
```

Pour calculer la valeur de chaque pixel vous devrez parcourir pour chaque position la liste des sphères, et vous devrez retourner la couleur associée à la sphère heurtée si elle existe, et `-INF` sinon. Il faudra aussi appliquer un dégradé en fonction de la distance par rapport au centre de la sphère. Petit rappel :

```

1 float hit(sphere s, float ox, float oy, float *n) {
2     float dx = ox - s.x;
3     float dy = oy - s.y;
4
5     if (dx * dx + dy * dy < s.radius * s.radius) {
6         float dz = sqrtf(s.radius * s.radius - dx * dx - dy * dy);
7         *n = dz / sqrtf(s.radius * s.radius); // pour l'ombre
8         return dz + s.z;
9     }
10
11    return -INF;
} // hit
```

Une fois que vous aurez calculé la valeur de chaque pixel vous devrez créer un fichier permettant de contenir l'image. Pour cela, vous allez utiliser le codage `ppm` qui permet de stocker des images graphiques. Plus précisément, vous utiliserez le format P3 qui se décrit ainsi :

```

P3
# Le P3 signifie que les couleurs sont en ASCII, et qu'elles sont en RGB.
# Par 3 colonnes et 2 lignes :
3 2
# Ayant 255 pour valeur maximum :
255
255 0 0      0 255 0      0 0 255
255 255 0     255 255 255     0 0 0
```

Le format limite le nombre de pixel par ligne, afin de vous simplifiez la vie, écrivez un pixel par ligne.

## Les threads POSIX

Afin d'accélérer la génération de l'image nous allons utiliser la programmation parallèle et plus précisément la bibliothèque `pthread`.

```
#include <pthread.h>
```

Pour compiler votre programme vous aurez besoin de `linked` avec la librairie. Pour cela il suffit d'ajouter l'option de compilation `-pthread`. Les threads POSIX sont dits légers car ils nécessitent généralement 30 fois moins de temps que les threads générés avec la fonction `fork` pour se générer. De plus, il est beaucoup plus facile partager de la mémoire. Pour créer un thread vous utiliserez la fonction `pthread_create` (comme d'habitude allez faire un petit tour sur la page man de cette fonction).

```

1 #include <pthread.h>
3 int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

Lorsque nous créons des threads puis nous laissons continuer par exemple la fonction main, nous prenons le risque de terminer le programme complètement sans avoir pu exécuter les threads. Nous devons en effet attendre que les différents threads créés se terminent. Pour cela, il existe la fonction `pthread_join` (idem allez faire un tour sur la page man de cette fonction).

```
#include <pthread.h>
2 int pthread_join(pthread_t thread, void **retval);
```

Grace à ces deux fonctions vous pouvez maintenant "découper" votre tableau afin de le fournir à plusieurs threads (faite attention de ne pas surcharger votre processeur). Normalement vous devrez observer un gain de performance. Utilisez la commande `time` pour observer les statistiques en temps de votre programme. Qu'observez vous ?