Projet INF 442

Quentin Bertrand & Mélanie Bru

Avril 2016

1 Basics

1.1 La classe *Vector*

La classe *Vector* est une classe représentant un vecteur en trois dimensions de l'espace. Elle comprend trois champs de type *double* correspondant aux trois cordonnées de l'espace : d1, d2 et d3.

Elle comprend les méthodes :

- *getd*1, *getd*2, *getd*3 pour accéder aux champs d'un *vector*
- *add*, pour ajouter deux *vector*
- *mult*, pour multiplier un *vecteor* par un *double*
- *print*, pour afficher les coordonnées du *vector*
- *equals*, pour tester l'égalité entre deux *vector*.

Par commodité nous avons également redéfini les opérateur $==$, $+$, $-$, $*$, pour tester l'égalité entre deux *vectors*, additionner, soustraire deux *vectors*, multiplier par un *double* et effectuer le produit scalaire.

1.2 La classe *Ray*

La classe *Ray* représente un rayon de lumière. Elle possède deux champs de type *double*

- *point*, correspondant à un point du rayon
- *direction*, correspondant à la direction du rayon

Elle comprend les méthodes *getPoint* et *getDir* pour accéder aux champs *point* et *direction*.

1.3 La classe *Sphere*

Elle comprend :

- les champs *centre* de type $Vector$ et *rayon* de type *double* correspondant au centre et au rayon de la sphère
- les champs de type double *red*, *green*, et *blue*, correspondant au niveau de chacune des couleurs dans la norme $sRGB$.
- les méthodes d'accès correspondantes $getCentre$, $getRayon$, $getRed$, $getGreen$ et $getBlue$.

1.4 La classe *Scene*

Elle comprend un champs *scene* de type $set < Sphere, ComparateurSphere >$, ainsi qu'une méthode d'accès $getScene$. Pour définir un objet de type *set* pour les *Sphere*, nous avons du définir un comparateur pour ces objets. En effet, un *set* est un ensemble muni d'une relation d'ordre, ce qui assure que l'on ne peut pas avoir le même élément deux fois dans un même *set*. Nous avons donc écrit une structure $ComparateurSphere$ qui définit une relation d'ordre que nous avons choisie arbitrairement sur les sphères.

1.5 La classe *Light*

Elle représente un point de lumière. Elle possède donc :

- les champs *point* de type $Vector$ et *red*, *green*, *blue* de type *double* correspondant à la source de lumière et à sa couleur dans le système RGB
- les méthodes d'accès orrespondantes $getPoint$, $getRed$, $getGreen$, $getBlue$.

1.6 La classe *Camera*

Elle possède :

- un champ *eye* de type $Vector$ représentant l'oeil de la caméra
- un champ *target* de type $Vector$ représentant le centre de l'écran visée par la caméra, de plus *target* est orthogonal à l'écran
- un champs *orientation* de type $Vector$ représentant la direction de l'écran
- deux champs *height* et *width* de type *double* correspondant au dimension de l'écran en terme de pixels

- les méthodes d'accès correspondantes $getEye$, $getTarget$, $getOrientation$, $getWidth$, et $getHeight$.

2 First algorithms

2.1 Ray-Sphere intersection

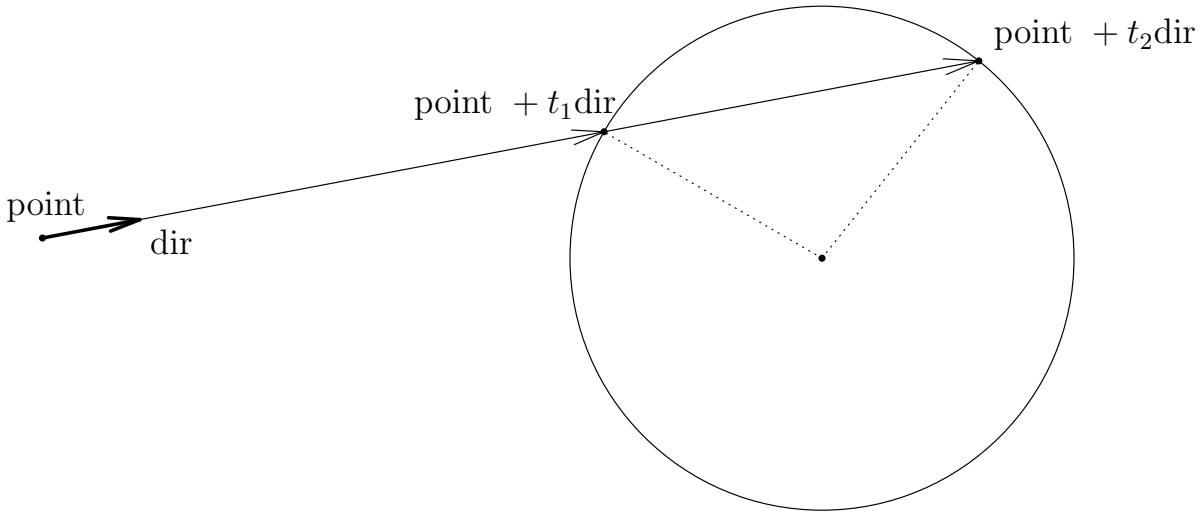

FIGURE 1 – Schéma illustrant le raisonnement utilisé pour déterminer l'intersection entre un rayon et une sphère

Le but de cette question est de déterminer si oui ou non il y a intersection entre un rayon lumineux r et une sphère s donnés.

Pour répondre à cette question nous avons créé une fonction *intersection* (dans un fichier *intersection.cpp*) qui prend en entrée un rayon lumineux r de type Ray, une sphère s de type $Sphere$ et qui renvoie un boléen, $true$ s'il y a intersection, $false$ sinon.

DÉTAILS DE L'ALGORITHME : notons *point* et *dir* respectivement un point, et la direction du rayon lumineux. On peut paramétrer l'ensemble décrit par le rayon lumineux par $\{point + dir * t | t \in \mathbb{R}\}$.

La question de l'intersection entre une sphere de centre *centre* et de rayon *rayon* est de savoir si il existe ou non un point de la droite décrite par le rayon lumineux qui passe à distance *rayon* du centre de la sphère.

Cela revient à se demander s'il existe $t \in \mathbb{R}$ tel que $\|point + dir * t - centre\| = rayon$?

Il s'agit d'une équation du second ordre en t que nous avons résolu dans le programme *intersection*.

2.2 Lighting model

Dans cette question on se fixe

- un point *point* sur une sphère
- un vecteur normal N à cette sphère en ce point
- une source de lumière *light*
- un récepteur (ici un oeil via une Camera *camera*)
- une sphere.

On calcule alors la couleur du point (donnée dans le système RGB par le modèle de Phong).

Modèle de Phong : l'ombrage de Phong est un modèle empirique d'infographie qui permet de calculer un éclairage réaliste d'une image et de donner ainsi un effet 3D.

Dans ce modèle on considère que chaque point éclairé d'une sphère émet de la couleur dans toutes les directions.

La couleur émise par le point est la somme :

- d'un terme de luminosité ambiante
- d'un terme de diffusion de la lumière
- d'un terme de réflexion de la lumière

2.3 Ray-traced image

Dans cette section on se donne :

- une source de lumière *light*
- un ensemble de sphères *scene*
- une Camera *camera*, c'est-à-dire notamment un oeil *eye* ainsi que les dimensions et la position des pixel de l'écran

Pour déterminer la couleur de chaque pixel sur l'écran, on parcourt l'ensemble de pixels et, pour chaque pixel, on détermine si le rayon qui va de l'oeil au pixel intersecte une des sphères *sphere* de la Scene.

S'il y a intersection, on calcule la couleur du point d'intersection entre la droite et la sphère (on prend le point le plus proche de l'oeil).

On stocke le résultat dans un $< vector < vector < Vector >>>$, ou chaque $Vector$ correspond à la couleur d'un pixel dans le système RGB.

On revoie la couleur des pixels de l'écran.

input : Camera camera, Scene scene, Light light
output : $< vector < vector < Vector >>>$ (ie une couleur pour chaque pixel)

```
< vector < vector < Vector >> resultat ; //on parcout chaque pixel de la
 caméra for pixel in camera do
    dist = infini ;
    sphrereVue ;
    //on regarde la sphère intersectée la plus proche
    for sphere in scene.getScene do
        if la droite oeil − pixel intersecte sphere et que la distance
         pointd'intersection − oeil est plus petite que la distance dist then
            dist = distance(pointd'intersection − oeil); //on met à jour la
             distance sphereVue = sphere; //on met à jour la sphère
        end
    end
    //on calcule la couleur du pixel
    if dist==inf then
        resultat[pixel] = black;
    end
    else
        resultat[pixel] = lighting(pointd'intersection, light, sphere);
    end
end
return resultat ;
```

Algorithme 1 : Algorithme ray-traced-image

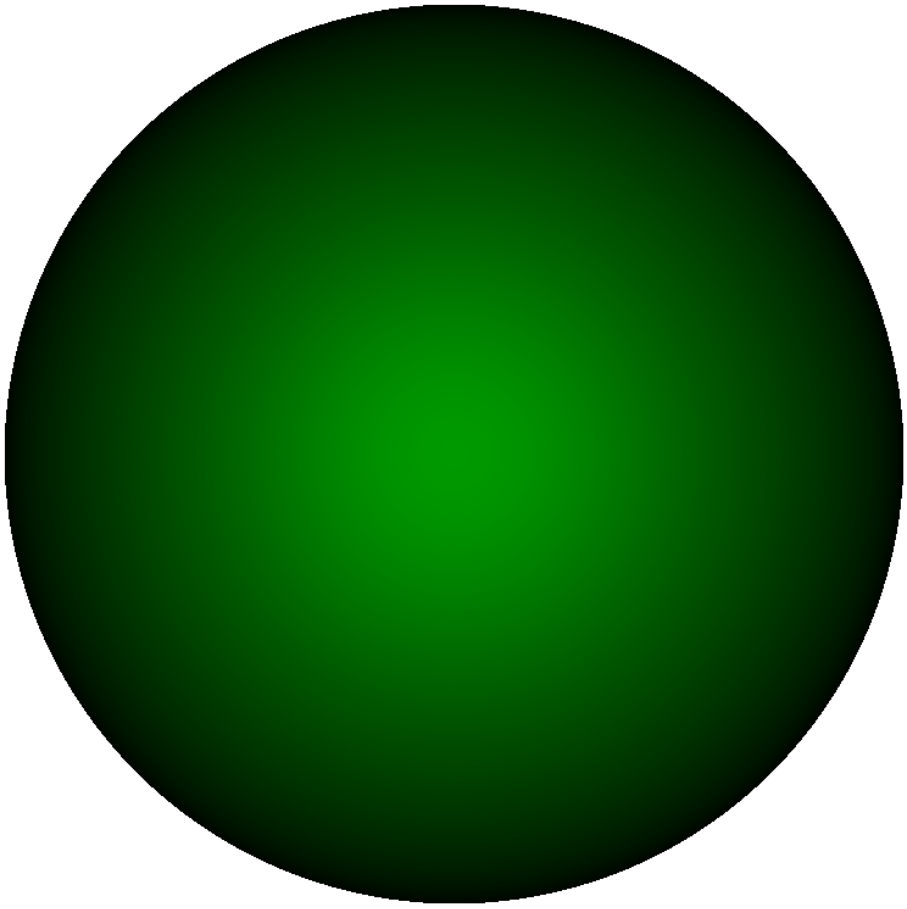

FIGURE 2 – Sphère en 3 dimensions réalisée avec notre ray-tracer

3 Shadows

L'algorithme *rayTracing* de 2.3 calcule la couleur de chaque point de l'écran, mais il ne prend pas en compte la possibilité que la sphère intersectée ne soit en fait pas éclairée, en effet il est possible d'être dans l'ombre d'une autre sphère (ou de sa propre sphère).

Pour résoudre ce problème nous avons ajouté une boucle avant l'attribution de la couleur, vérifiant qu'aucune sphère n'intercepte le rayon lumineux de la source.

S'il y a eu intersection, on regarde la droite $pointd'intersection - sourcedelumière$, et on regarde si cette droite intersecte l'une des sphères de la Scène. Si le nouveau point d'intersection est plus proche de la source on grise le pixel regardé initialement.

```
input  : Camera camera, Scene scene, Light light
output : < vector < vector < Vector >> (ie une couleur pour chaque pixel)
< vector < vector < Vector >> resultat ; //on parcout chaque pixel de la
 caméra for pixel in camera do
    dist = infini ;
    sphrereVue ;
    pointVu ;
    //on regarde la sphère intersectée la plus proche
    for sphere in scene.getScene do
        if la droite oeil − pixel intersecte sphere et que la distance
         pointd'intersection − oeil est plus petite que la distance dist then
            dist = distance(pointd'intersection − oeil) ; //on met à jour la
             distance
            sphereVue = sphere ; //on met à jour la sphère
            pointVu = pointd'intersection ;on met à jour le point d'intesection
        end
    end
    for sphere in scene.getScene do
        si le point d'intersection est caché par sphere :
        resultat[pixel] = black;
        et on sort de la boucle (break)
    end
    //on calcule la couleur du pixel
    if dist==inf then
        resultat[pixel] = black;
    end
    else
        resultat[pixel] = lighting(pointd'intersection, light, sphere);
    end
end
return resultat ;
```

Algorithme 2 : Algorithme shadows

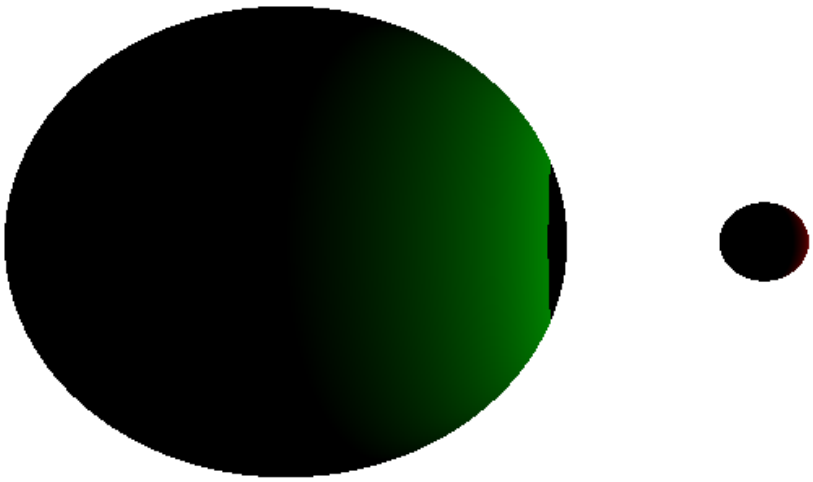

FIGURE 3 – Sphères en 3 dimensions avec ombres

4 Reflection

Dans cette section on ajoute un champ *double reflection* à l'objet *Sphere* dans le but de prendre en compte la réflection. La couleur vue par la caméra sera la somme de deux termes :

- le terme issu de *shadows* (1-coefficient de réflexion)*(couleur calculée par shadows)
- un terme nouveau issu de la réflexion il s'agit de la couleur vue par le ray-tracer avec pour rayon (le point d'intersection, la direction de réflexion des lois de DESCARTES).

Le pseudo-code est semblable à *shadows*, en plus ici on rajoute le nouveau terme à l'aide d'un appel à $ray - tracing$.

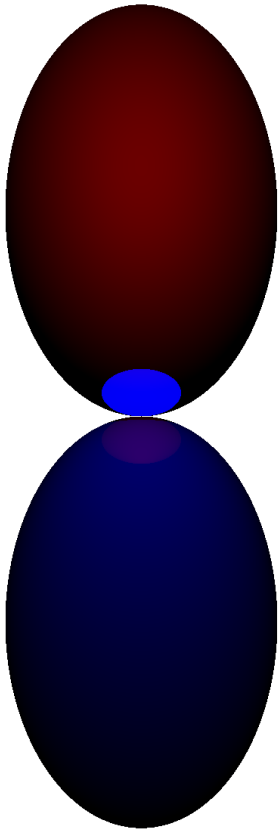

FIGURE 4 – Sphères en 3 dimensions avec ombres et réflexions

5 Distributed ray-tracing

Le calcul pixel par pixel est coûteux et long. Il est donc utile de le programmer en MPI afin de gagner en temps de calcul.

La solution que nous avons retenue pour partager la tâche entre les différents processeurs est de diviser l'image en plusieurs zones de taille égale et de donner à chaque processeur une partie de l'image à calculer. pour cela, il suffit de modifier la *Camera* dont dispose chaque processeur. Nous avons choisi de diviser la hauteur de l'image par le nombre de processeurs et de fournir à chaque processeur une bande de l'image de hauteur $heigth/numprocs$, de largeur $width$ et dont le centre $target$ est déplacé en conséquence. On peut voir un schéma de cette division de l'écran dans la figure ci-dessous.

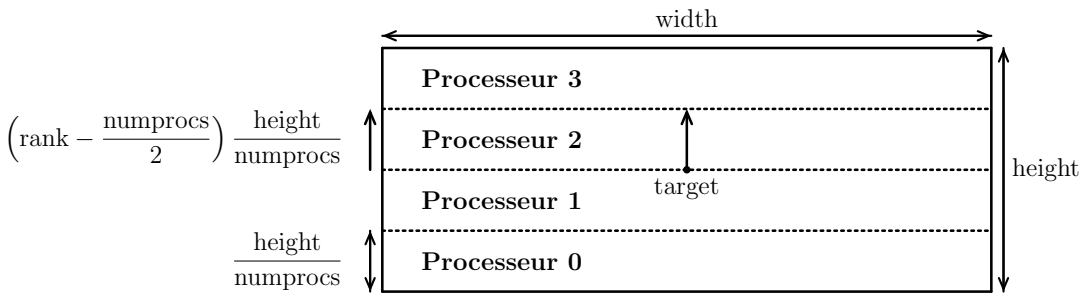

FIGURE 5 – Division de l'écran pour le MPI

On fait alors appel au même algorithme de calcul d'image que dans la question précédente. La différence vient du fait qu'il est cette fois appelé dans une autre fonction, que nous avons nommé $ReflectionMPI$ dans le code. Cette fonction initialise le MPI et le processeur 0 envoie à tous les processeurs, le nombre de sphères de *Scene*, les caractéristiques de chacune de ces sphères, de la lumière et de la caméra sous forme d'un $double*$. La modification de *camera* que nous avons décrite précédemment est réalisée.

Chaque processeur calcule alors indépendamment sa partie de l'image à l'aide de la fonction $Reflection$ vue précédemment. Les résultats sont alors convertis en $double*$ et envoyés via MPI au processeur 0.