Rapport Tp1 ISIR

Vacher Jérémy

February 2024

Table des matières

1 Introduction

L'objectif de ce TP est de réaliser nos premières images en utilisant une méthode de lancer de rayons à partir d'une caméra. Nous devons aussi créer notre tout premier objet en comprenant la technique d'intersection entre un rayon et notre objet. Enfin, nous verrons le fonctionnement de la caméra et l'importance de comprendre comment la positionner pour avoir un meilleur rendu.

2 Première image

Dans le premier exercice, l'objectif est de remplir une image en fonction des cordonnées de ses pixels. Pour obtenir la première image, j'ai créé trois variables de types flottant pour nos couleurs. La variable bleue est égale zéro, la variable rouge correspond à la valeur de l'abscisse et la variable verte correspond à la valeur de l'ordonnée.

Pour avoir l'interpolation entre 0 et 1 pour la couleur rouge et verte on divise notre coordonnée de pixel par la taille - 1.

Dans la fonction renderImage() de notre renderer, nous obtenons alors ceci :

```
for ( int j = 0; j < height; j++ )
{
    for ( int i = 0; i < width; i++ )
    {
        red = (float)i / ( width - 1 );
        green = (float)j / (height - 1);
        p_texture.setPixel( i, j, Vec3f( red, green, blue ) );
    }
}
```

3 Caméra, rayons et projection perspective

3.1 Dimensions et position de la fenêtre virtuelle

Dans cette exercice nous allons effectuer des lancer de rayons à partir de la caméra. Nous allons alors commencer par calculer dans la fonction update-Viewport() de PerspectiveCamera les dimensions et la position de la fenêtre virtuelle.

Nous savons que pour calculer la hauteur de notre fenêtre virtuelle, nous pouvons faire :

$$height = tan(fovy) * focalDistance * horizontalaxes$$

Ensuite pour calculer la largeur de notre fenêtre virtuelle, nous utilisons :

$$width = height * aspectRatio * verticalaxes$$

Enfin, pour calculer la position du bord haut gauche on prend la position moins la multiplication entre l'axe de vue de la caméra et la distance focale auquel on ajoute la hauteur de notre fenêtre graphique moins la largeur.

$TopLeftCorner = position - (w * focalDistance) + (viewportV) - (viewportU);$

3.2 Génération des rayons primaires

Pour générer les rayons primaires, on part de notre positions situé en haut à gauche de la fenêtre. On y ajoute la position x multiplié par la largeur de la fenêtre moins la position y multiplié par la hauteur de la fenêtre.

On calcule la direction entre notre position sur la fenêtre et la position de la caméra et on retourne le rayon.

```
inline Ray generateRay(const float p_sx,const float p_sy)const override
{
    Vec3f posViewPort = _viewportTopLeftCorner + (_viewportU * p_sx)
    -(_viewportV * p_sy);
    Vec3f dirRay          = glm::normalize(    posViewPort - _position );
    return Ray( _position, dirRay );
}
```

3.3 Modification renderer

Enfin on modifie notre renderImage, on créé deux variables psx et psy qui vont être les positions des pixels sur notre fenêtre. Puis on génère un rayon et on utilisera la direction du rayon pour calculer la couleur du pixel.

```
float p_sx = (float)(i + randomFloat()) / ( width - 1 );
float p_sy = (float)(j + randomFloat())/ ( height - 1 );
Ray   ray   = p_camera->generateRay( p_sx, p_sy );
p_texture.setPixel( i, j, (ray.getDirection() + 1.f) * 0.5f );
```

4 Le premier objet : une sphère

4.1 Intersection Sphère/Rayon

En utilisant l'équation de l'intersection entre une sphère et un rayon on trouve a,b et c. On peut alors calculer delta, si delta est strictement supérieur à zéro on a deux intersections puisque le rayon traverse la sphère. Si delta est égal à zéro alors nous avons une intersection sinon nous n'avons pas d'intersection.

4.2 Modification renderer

Si l'on appelle la méthode LI dans notre renderImage nous obtenons l'image suivante.

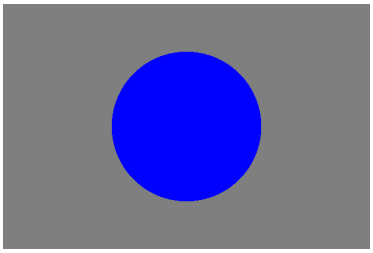

FIGURE 1 – Figure 3(a)

4.3 Modification de la fonction Li

En prenant en compte l'angle entre la normale et la direction du rayon, puis en récupérant le maximum entre notre cosinus et 0 nous trouvons le résultat ci dessous.

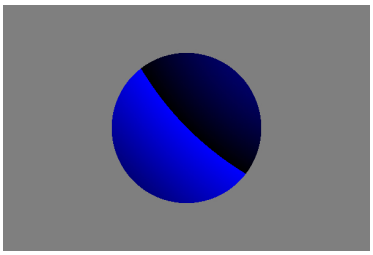

FIGURE 2 – Figure 3(b)

4.4 Calcul de la normale

Enfin, on calcule la normale entre le centre de notre sphère et le point sur la fenêtre graphique et on obtient cette image.

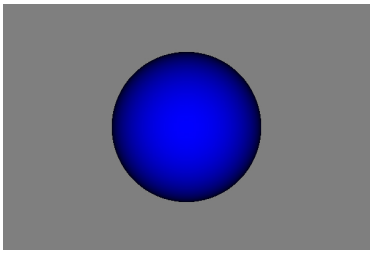

FIGURE 3 – Figure 3(c)

5 Une caméra positionnable

5.1 Implémentassions des vecteurs u, v et w

On commence par calculer w qui représente l'inverse de la direction d'observation. Ensuite, on fait le produit vectorielle entre notre vecteur w et la rotation de la caméra pour obtenir u. Enfin, on fait le produit vectorielle entre w et u pour avoir v.

```
_w = glm::normalize( -p_lookAt + p_position );
_u = glm::normalize(glm::cross(p_up, _w));
_v = glm::normalize(glm::cross(_w, _u));
```

5.2 Vérification du bon fonctionnement de la caméra

En plaçant la caméra en (0,0,-2) qui regarde en (0,0,79) on obtient l'image ci-dessous :

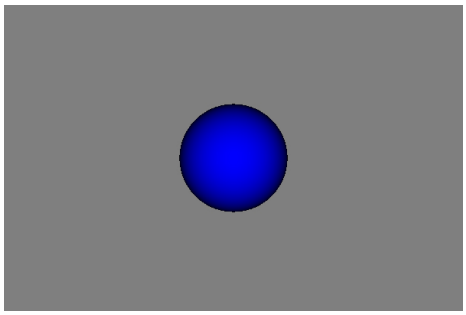

FIGURE 4 – Figure 5(a)

En plaçant la caméra en (1,0,0) qui regarde en (1,0,1) on obtient l'image ci-dessous :

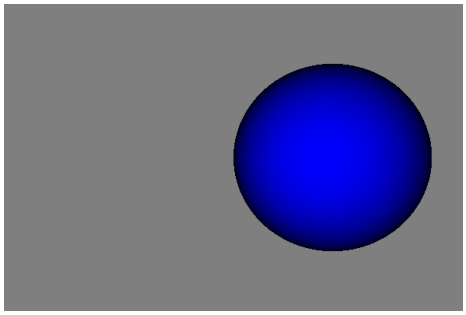

FIGURE 5 – Figure 5(b)

En plaçant la caméra en (0,1,0) qui regarde en (0,1,1) on obtient l'image ci-dessous :

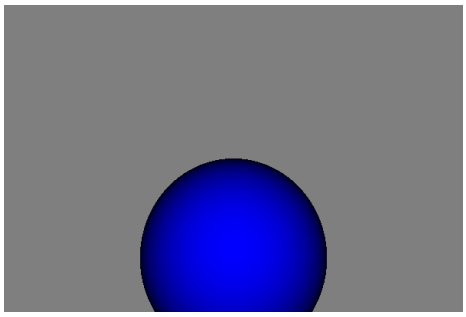

FIGURE 6 – Figure 5(c)

En plaçant la caméra en (4,-1,0) qui regarde en (-1,-1,2) on obtient l'image ci-dessous :

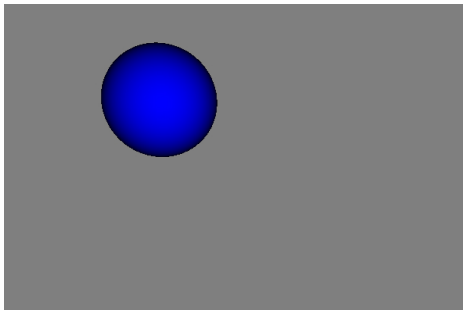

FIGURE 7 – Figure 5(d)

6 Antialiasing

Pour pouvoir lancer plusieurs rayons on fais une boucle dans renderImage sur le nombre de pixel sample dans la boucle on additionne notre couleur. Enfin, après être sortie de notre boucle on divise notre couleur par le nombre de pixel sample. On rajoute aussi dans le main la fonction setNbPixelsSamples().