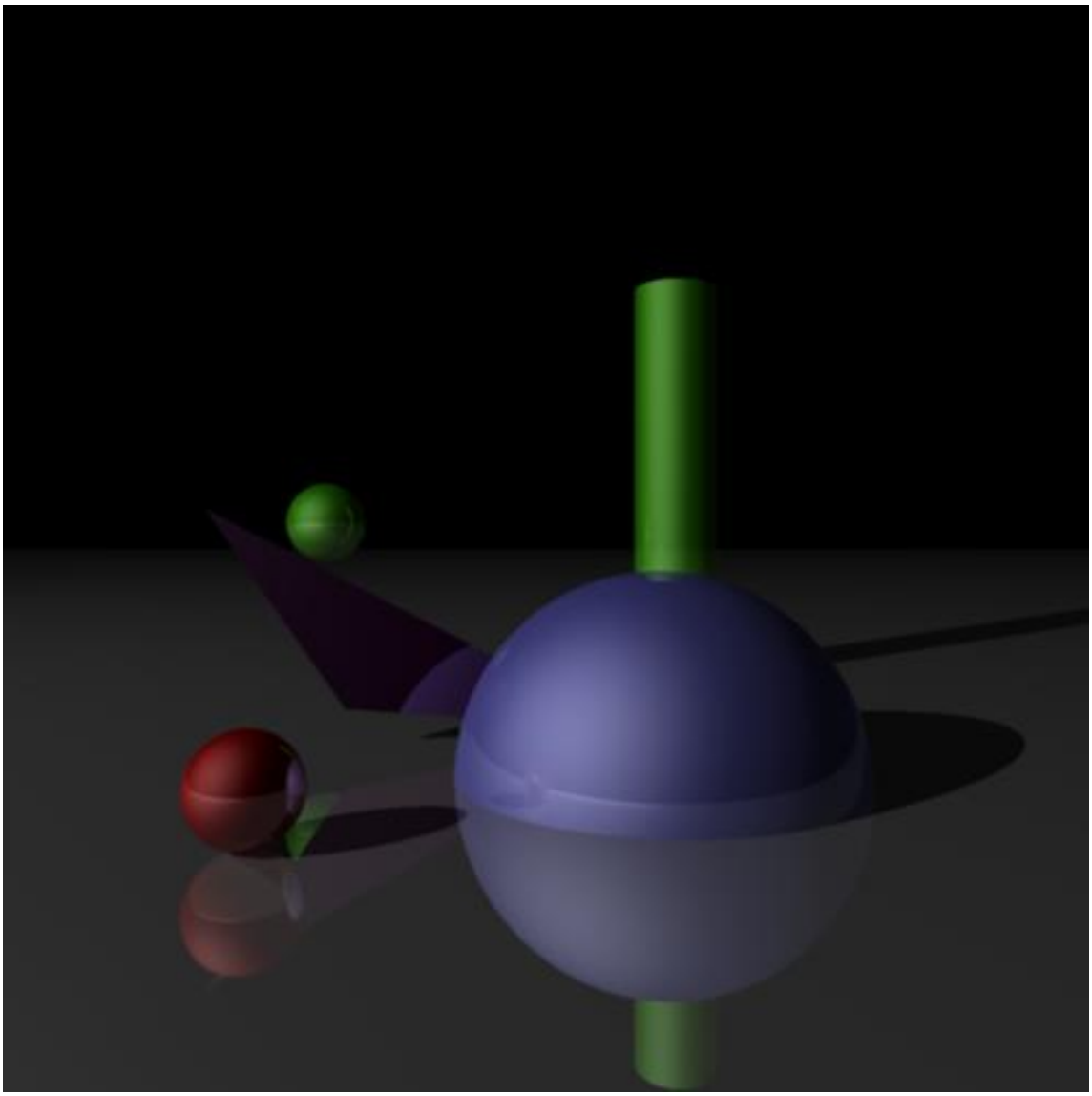
BADOUX Mathieu

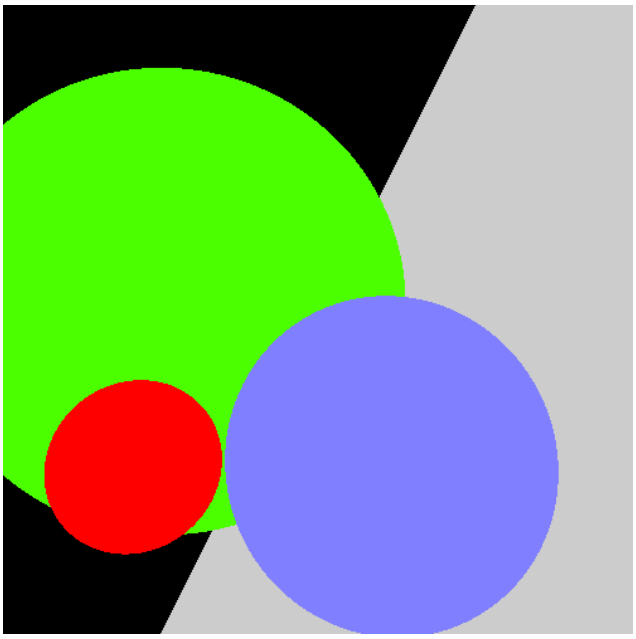
LEGAULT Bastien

TP RAY-TRACING



Introduction

On a compilé et exécuté le projet d'origine et on a obtenu le résultat suivant :



On crée une image de taille 500x500 et on initialise chaque pixel de l'image avec la couleur noire. Cette image est la représentation d'une scène, sur laquelle nous fixons une caméra en paramétrant sa position, sa distance à l'écran et sa focale

On ajoute ensuite plusieurs primitives, sphères et plan, à cette scène à l'aide de la fonction push_back() de la classe scène dans laquelle nous précisons la classe de notre primitive ainsi que le matériel de notre objet. Nous ajoutons également dans notre scène une source de lumière afin d'éclairer notre scène.

On appelle ensuite l'algorithme du tracé de rayon avec la fonction render puis nous sauvegardons notre image en format ppm.

Principe de base du ray tracing

Dans cette fonction, nous récupérons tout d'abord la position de notre caméra afin d'avoir le point d'origine des rayons que nous calculerons juste après. Pour chaque pixel de notre image, nous générons un rayon à l'aide de la fonction ray_generator() qui calcule la position de notre pixel en 3D et retourne le rayon entre le pixel courant et cette position.

La couleur de ce pixel est trouvée avec la fonction ray_trace() qui va vérifier si le rayon courant rentre en collision avec une primitive. Si c'est le cas, on cherche la couleur de la surface rencontrée. Cette couleur dépend aussi de l'illumination de la primitive entre autres. Pour l'instant, la couleur est calculée qu'avec la première primitive rentrée et non pas la primitive la plus proche de notre caméra.

Le rayon est donc défini dans la classe ray, il possède un point de départ et une direction. La classe ray possède également différentes méthodes telles que le calcul d'un point de ce rayon à une distance donnée, d'affecter un offset selon la direction de notre rayon ainsi de calculer la réflexion de notre rayon par rapport à une normale.

Chaque primitive possède une méthode intersect() propre à elle qui permet de savoir si le rayon possède une intersection avec elle.

On a donc une classe permettant de gérer les intersections qui a comme paramètre la position de l'intersection dans l'espace, la normale de la primitive en ce point et la distance relative entre le point d'intersection et la source du rayon. On stocke la normale du point d'intersection pour pouvoir par la suite réaliser des réflexions entre les différentes primitives.

Le paramètre « relative » peut être négatif si l'intersection de fait de l'autre côté de la caméra, soit dans la direction opposé du rayon. Il nous permet donc de vérifier si l'intersection détectée est devant la caméra ou derrière.

Les différentes primitives

Le plan

Chaque primitive possède donc sa propre méthode intersect() pour savoir si le rayon possède une intersection avec la primitive.

Un plan est défini par un point et une normale. L'intersection de celui-ci avec rayon (une droite passant par le point x_s avec un vecteur directeur u) est donnée par :

$$\begin{cases} < x - x_p, n_p > = 0 \\ x = x_s + tu \end{cases}$$

On regarde tout d'abord que le rayon n'est pas parallèle au plan. On fait donc le produit scalaire entre la normale du plan et la direction du rayon, si c'est nul, le rayon est parallèle au plan et il n'y a donc pas d'intersection.

Si ce n'est pas nul, on calcul la distance entre un point du plan (celui par lequel il a été défini) et l'origine du rayon à l'aide de la fonction suivante :

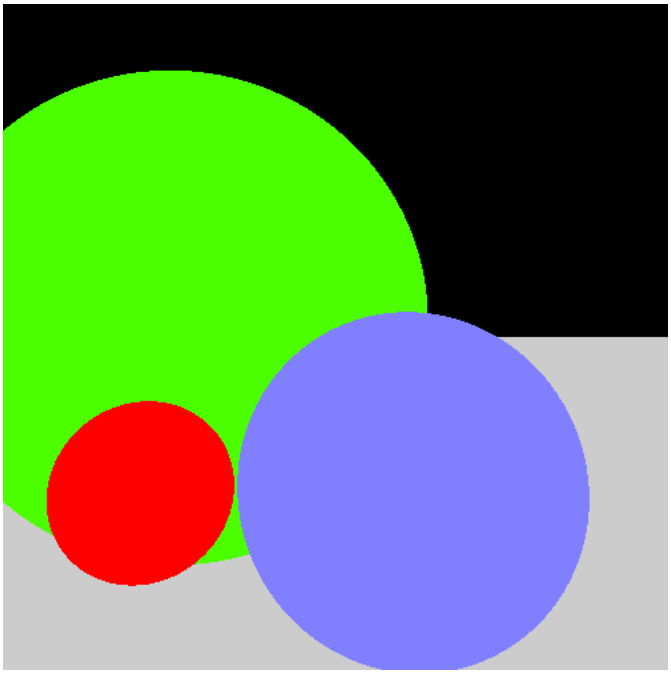
$$dist = \frac{< x_p - x_s, n_p >}{< u, n_p >}$$

Si cette distance est positive alors il y a une intersection dans le champ de la caméra au point d'intersection de coordonnée suivante :

$$Position_{inter} = Position_{origine\ rayon} + dist * u$$

L'intersection que l'on retourne possède donc le point d'intersection, la normale de la primitive en ce point qui pour un point du plan est celle du plan et la distance entre ce point et l'origine du rayon.

Une fois cette fonction codée, on obtient le résultat suivant :



La sphère

Une sphère possède un centre et un rayon, l'intersection de celle-ci avec un rayon est donnée par :

$$\begin{cases} \|x - x_0\|^2 = r^2 \\ x = x_s + tu \end{cases}$$

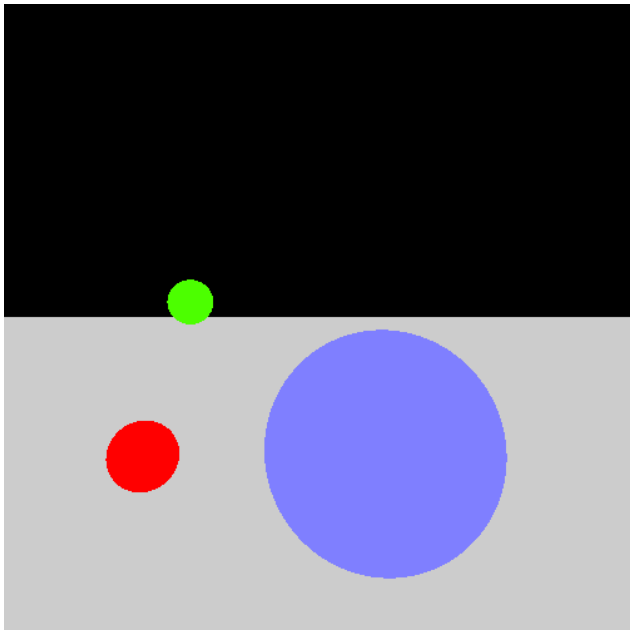
Un rayon peut couper une sphère de trois façons différentes. Soit il n'y a pas d'intersection, soit il y en a une seule et le rayon est tangent à la sphère, soit il y en a deux, une à l'entrée du rayon dans la sphère et une à la sortie. Cela correspond donc à une équation du second degré.

On calcule donc le delta de cette équation qui est le suivant :

$$\Delta = 4 * \left(< Pos_{origine\ rayon}, u > \right)^2 - 4 * < u, u > * < Pos_{origine\ rayon}, Pos_{origine\ rayon} - r^2 >$$

Un delta négatif signifie qu'il n'y a pas d'intersection. Un delta nul signifie qu'il y a une seule intersection et la distance entre l'origine du rayon et le point d'intersection est la solution de l'équation. Un delta positif correspond à deux intersections, on vérifie donc que la distance à ces intersections soient positifs sinon l'intersection a lieu derrière la caméra et donc n'est pas affichée. On prend ensuite la distance positive la plus petite car la seconde correspond à la partie cachée de la sphère.

On obtient le résultat le résultat suivant après codage de cette intersection :



Le stockage des primitives

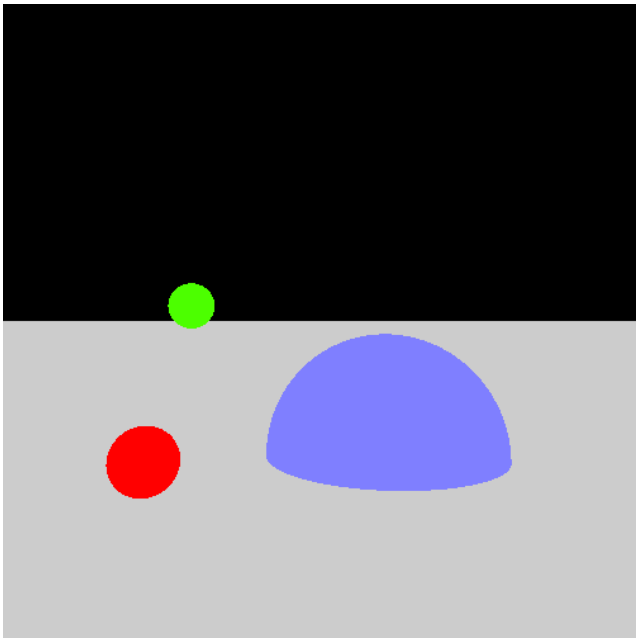
Pour stocker ces primitives et leurs caractéristiques (matériaux et illuminations), on utilise la classe scene_parameter. Dans cette classe on stocke les primitives sous forme de pointeur. Cela nous permet d'utiliser le polymorphisme.

En effet, la méthode intersect est définie dans la classe primitive_basic, les classes plane et sphere dérivant de cette dernière possèdent elles aussi, par héritage, la méthode intersect. Cependant cette méthode n'est pas la même pour chacune des primitives comme expliqué précédemment. Lors de l'utilisation de cette méthode, pour que l'ordinateur comprenne quelle méthode intersect choisir il faut donc que les primitives soient stockées en tant que pointeur et, comme c'est le cas ici, écrire override lors de la redéfinition de la méthode intersect dans les fichiers .h de chaque primitive.

La gestion de l'intersection avec les primitives

Pour chaque rayon, on utilise la fonction compute_intersection pour savoir s'il y a une intersection. Dans cette fonction, on parcourt toutes les primitives de la scène. Pour chaque primitive, on regarde le résultat de la méthode intersect, d'où l'intérêt du polymorphisme. S'il y a une seule intersection, on sauvegarde les informations liées à cette intersection, notamment de quelle primitive il s'agit. S'il y a plusieurs intersections, on ne retient que celle avec le paramètre « relative » le plus petit, mais positif, c'est-à-dire celle avec la primitive la plus proche de notre caméra.

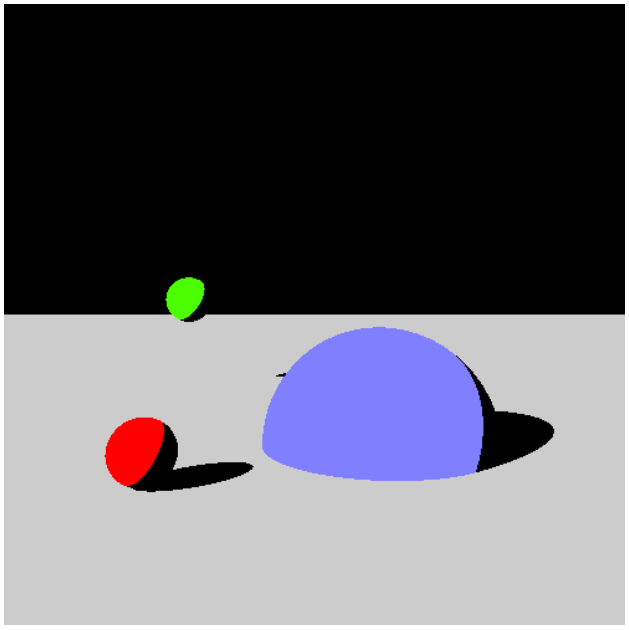
On obtient pour notre scène le résultat suivant :



On cherche ensuite à savoir si chacun des points d'intersection que nous allons afficher sont dans la lumière ou à l'ombre.

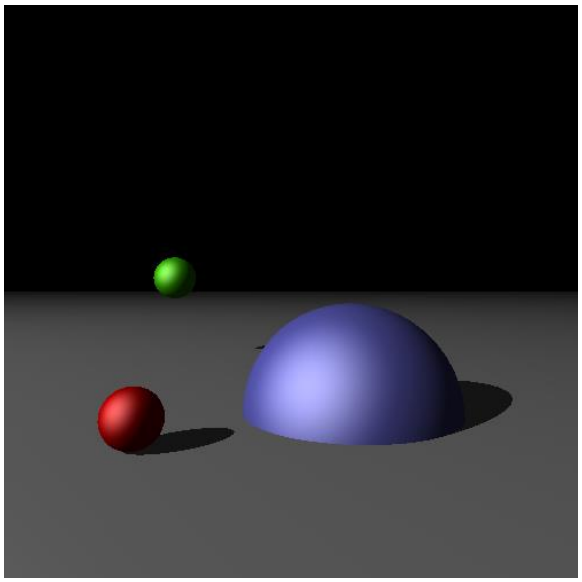
Dans la fonction is_in_shadow on crée un rayon correspondant à la lumière, il a donc pour origine le point d'intersection résultant de la fonction interesect et se dirige vers l'origine de la lumière de la scène, supposé ponctuel. Cependant on utilise un léger offset dans la direction du rayon car en partant d'un point d'intersection, on risque de détecter une intersection immédiatement avec ce même point et donc créer une ombre sur une surface qui était peut-être éclairée. On regarde ensuite s'il existe au moins une intersection de la même manière que précédemment. S'il y en a une, le point d'intersection est donc dans l'ombre, sinon il est éclairé directement.

En associant à chaque pixel qui est à l'ombre la couleur noire, on obtient le résultat suivant :

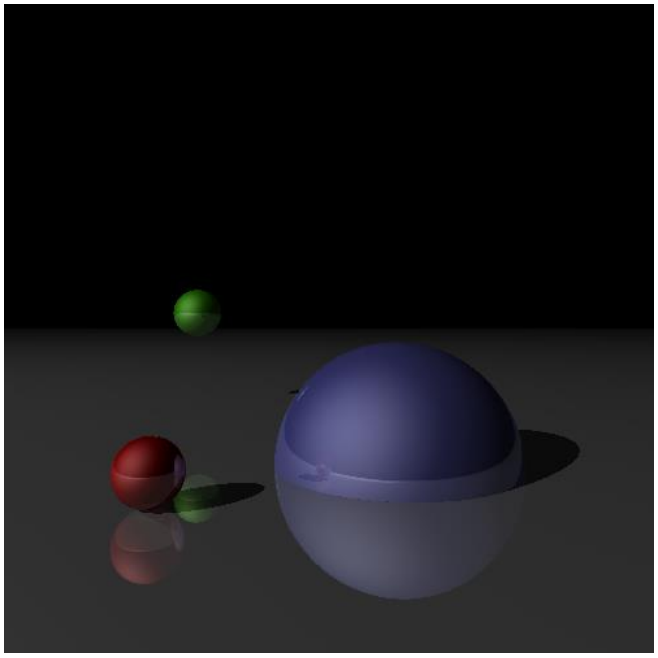


Cette information est ensuite utilisée dans la fonction compute_illumination. Pour chacune des intersections, on somme les différents ombrages créés en fonction notamment de la puissance de chaque source lumineuse. Ces ombrages sont calculés dans la fonction compute_shading.

On utilise une illumination de type phong qui permet d'avoir un éclairage plus réaliste et on obtient la scène suivante :

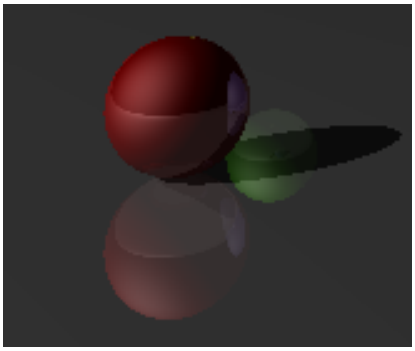


De plus, certaines primitives peuvent réfléchir la lumière. Dans la fonction ray_trace, on choisit arbitrairement le nombre de réflexion que l'on veut. Pour chaque réflexion, on trace un nouveau rayon partant du point d'intersection courant et ayant une direction symétrique par rapport à la normal, trouvé grâce à la fonction relflected. Ce nouveau rayon peut potentiellement avoir une intersection avec une autre primitive. La couleur finale est une pondération de chaque réflexion, cette pondération est obtenue grâce au paramètre reflection de la classe material associé à chaque primitive.

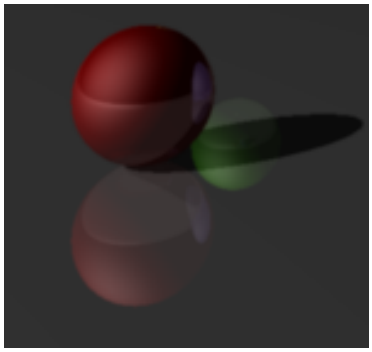


On remarque que sur les contours francs de changement de couleur, notamment dus aux ombres, il peut y avoir des coupes pixélisées, du crénelage. Pour remédier à cela, on peut utiliser le mécanisme d'anti-aliasing qui consiste à échelonner la couleur des pixels dans les zones de contour. Les contours sont donc plus doux mais peuvent apparaître plus flous.

On va donc lancer plusieurs rayons légèrement décalés pour obtenir la couleur du même pixel, ces différents rayons ont un poids suivant s'ils sont loin du rayon d'origine. La pondération totale de ces rayons nous permet d'obtenir la couleur finale.



Avant anti-aliasing



après anti-aliasing

Pour aller plus loin

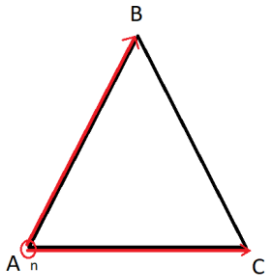
On a décidé de mettre en place le calcul de deux autres primitives.

Le triangle

Un triangle est défini par ses trois points, sa normale est calculée en fonction de ses points par le produit vectoriel de deux côtés. On a eu plusieurs approches pour implémenter la fonction intersect propre au triangle.

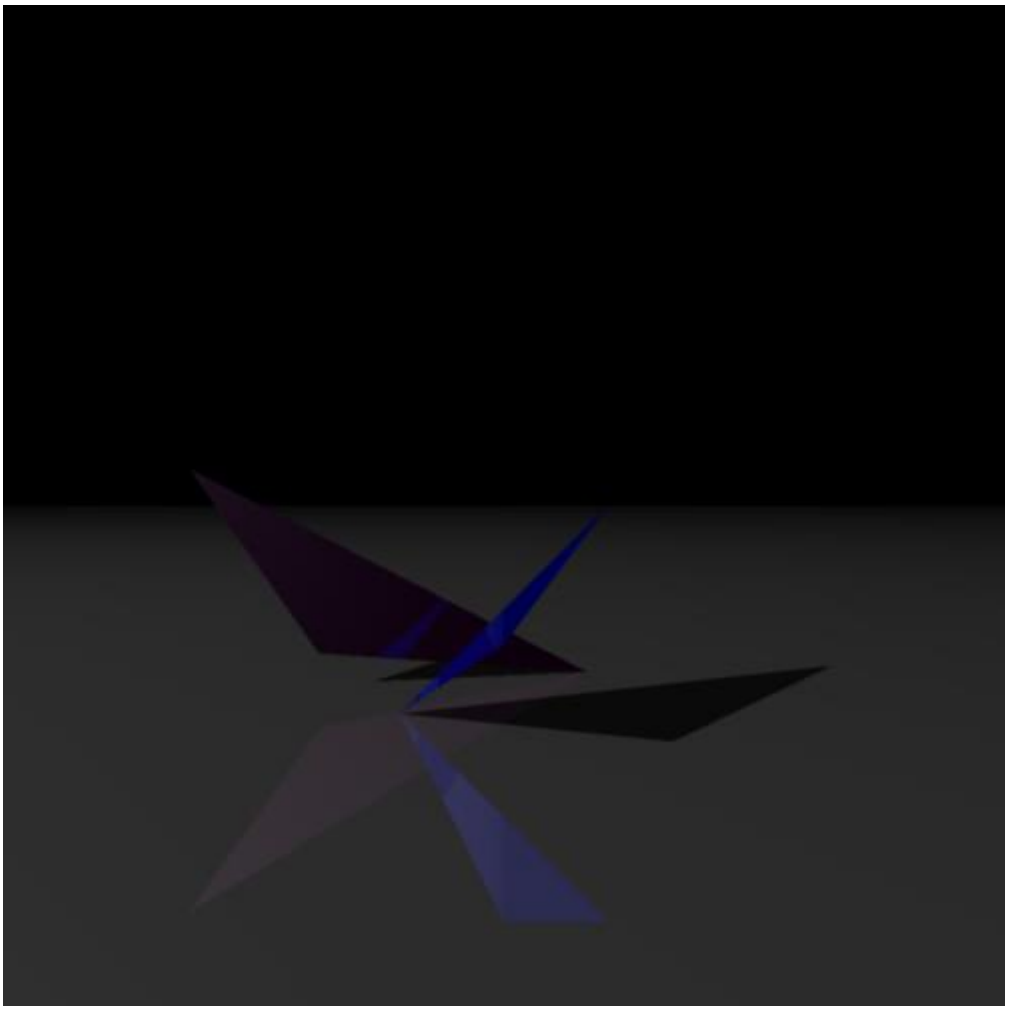
Dans chacune des méthodes, on commence par vérifier que le rayon courant est en intersection avec le plan du triangle. Si c'est le cas, on vérifie ensuite si le rayon traverse le triangle.

La première méthode consistait à convertir les coordonnées dans une base propre au triangle. Pour cela on crée la matrice de passage entre les deux bases, la base orthonormée $(\vec{x}, \vec{y}, \vec{z})$ et la base du triangle formé par deux côtés du triangle et la normale $(\overrightarrow{AB}, \overrightarrow{AC}, \vec{n})$.



Cette matrice de passage P est créée en exprimant la base du triangle dans la base orthonormée de l'espace. On test ensuite si les coordonnées converties en AB et AC sont comprises entre 0 et 1 puis que leur somme est inférieure ou égale à 1. Normalement la coordonnée en n est nul car le point est situé sur le triangle. Cependant, en utilisant cette méthode, certains triangles n'étaient pas positionnés au bon endroit, malheureusement on a pas trouvé pourquoi.

On a donc utilisé une autre méthode. Soit M le point d'intersection, on calcul les trois produits vectoriels suivants : $<\overrightarrow{AB}, \overrightarrow{AM}>$, $<\overrightarrow{BC}, \overrightarrow{BM}>$ et $<\overrightarrow{CA}, \overrightarrow{CM}>$. Ces trois produits vectoriels doivent normalement créer trois vecteurs aillant la même direction que la normale si le point d'intersection est dans le triangle. On test donc si chacun des produits scalaires entre ces vecteurs résultants et la normale du triangle est positif. Cette méthode donne des résultats satisfaisants.



Le cylindre

On crée ici un cylindre particulier, son axe est vertical. Un cylindre est défini par deux centres, les centres de ses deux bases et un rayon, l'intersection de celui-ci avec un rayon est donnée par :

$$\begin{cases} x(t)_x^2 + x(t)_y^2 = r^2 \\ x = x_s + tu \end{cases}$$

Un rayon peut couper un cylindre de trois façons différentes. Soit il n'y a pas d'intersection, soit il y en a une seule et le rayon est tangent au cylindre, soit il y en a deux, une à l'entrée du rayon dans le cylindre et une à la sortie. Cela correspond donc à une équation du second degré.

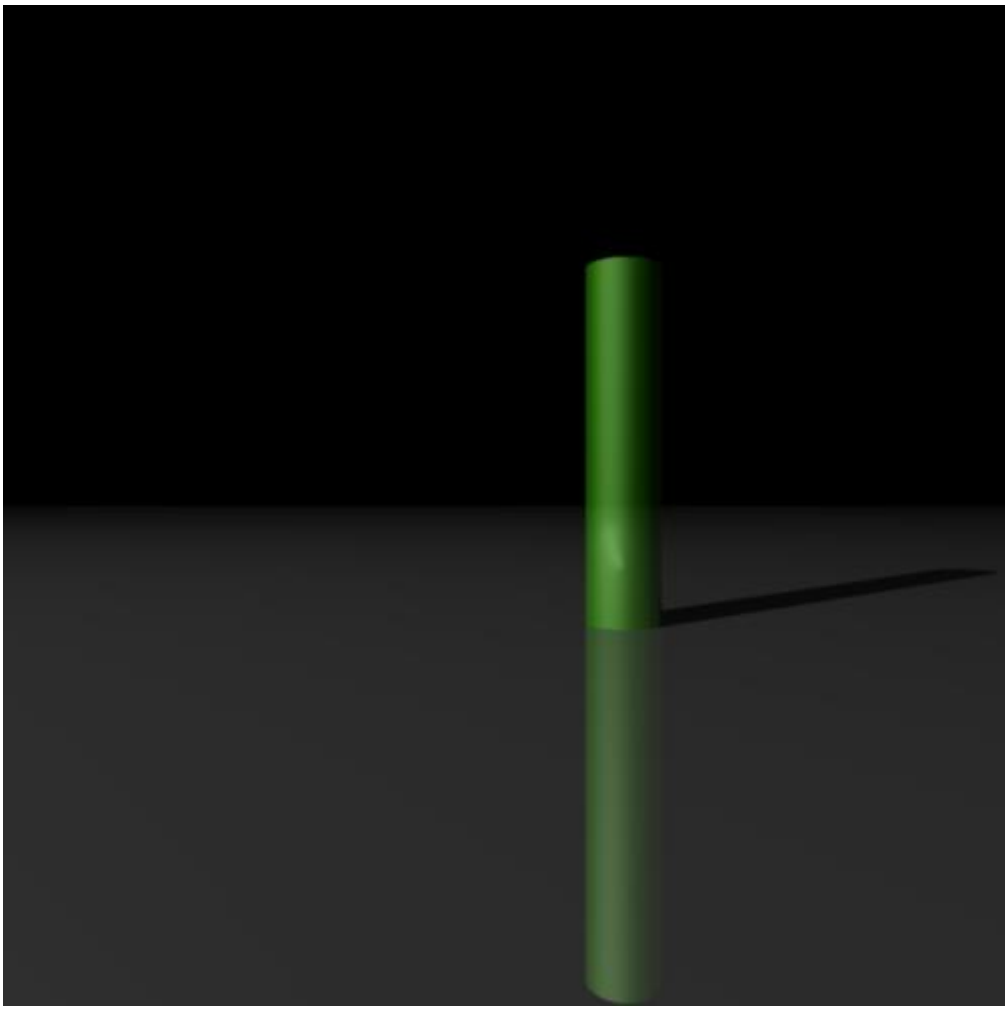
On calcule donc les coefficients de cette équation qui sont les suivants :

$$A = u_x^2 + u_z^2$$

$$B = 2 * \left[\left(X_{s_x} - centre1_x\right) * u_x + \left(X_{s_z} - centre1_z\right) * u_z \right]$$

$$C = \left(X_{s_x} - centre1_x\right)^2 + \left(X_{s_z} - centre1_z\right)^2 - rayon$$

On calcule ensuite le delta. Un delta négatif signifie qu'il n'y a pas d'intersection. Un delta nul signifie qu'il y a une seule intersection et la distance entre l'origine du rayon et le point d'intersection est la solution de l'équation. Un delta positif correspond à deux intersections, on vérifie donc que la distance à ces intersections soient positifs sinon l'intersection a lieu derrière la caméra et donc n'est pas affichée. On prend ensuite la distance positive la plus petite car la seconde correspond à la partie cachée de la sphère. Cependant, notre cylindre ayant une certaine hauteur, il faut vérifier que le point soit compris entre les deux valeurs en y des centres, notre cylindre étant vertical.



Multithreading utilisant open MP

Lors de l'affichage, nous avons vu que l'exécution de notre prenait du temps. Afin d'accélérer le temps de calcul de celui-ci, nous avons mis en place la parallélisation de notre code. Pour ce faire, nous avons utilisé la bibliothèque OpenMP. Celle-ci permet de mettre en place facilement la parallélisation de notre fonction render(). Pour cela, au début de notre fonction, nous rajoutons la ligne suivante afin de paralléliser le calcul de couleur sur les différents pixels de notre image par un nombre nb_threads que l'on définit précédemment :

```
#pragma omp parallel for num_threads(nb_threads) private(kx, ky)
```

Nous avons alors calculé le temps de calcul de la fonction render avec l'utilisation de 5 threads en parallèle et nous allons le comparer avec le temps de calcul sans multithreading.

Sans le multithreading, le temps d'exécution de notre fonction render est de 5.67s alors que l'utilisation de 5 threads en parallèle nous permet de diminuer ce temps à 1.60s. Nous avons donc diminué notre temps de calcul par 3.5.

Une autre possibilité est de créer soi-même n différents threads qui vont calculer alors les différentes couleurs de l'image pour chacun des pixels. Nous pouvons créer ceci grâce à la classe thread en leur appliquant une fonction à chacun d'entre eux. Il faut alors, avant d'enregistre l'image attendre que chacun des threads aient fini en utilisant la commande std ::thread.join() qui permet d'attendre la fin d'exécution d'un thread. Afin que les threads n'écrivent pas en même temps sur notre image, nous devons placer un verrou sur celle-ci, pour cela nous pouvons utiliser la classe mutex et appliquer le verrou juste avant l'écriture sur l'image en utilisant mutex.lock() puis mutex.unlock() une fois l'écriture fini.