Rapport Informatique Graphique

Valentin BONNET

Ce rapport a pour but de présenter le travail réalisé sur l'implémentation d'un algorithme de raytracing pour la création d'images. Les différents éléments de l'algorithme seront mis en place au fur et à mesure, afin de constater à chaque ajout, de quelle manière il est implémenté, et la différence qu'il apporte à la création d'images. Tous les détails du code ne sont pas donnés, pour se concentrer sur les résultats intermédiaires. De même, les principes implémentés sont rappelés seulement à titre indicatif pour introduire les différentes parties.
Le code final de ce travail peut être trouvé dans le fichier *raytracing.cpp*.

Principe du lancer de rayons et mise en place d'une première scène

Lancer de rayons

La génération d'image par raytracing repose sur un principe de lancer de rayon pour chaque pixel de l'image à générer. Le principe est le suivant : l'image à représenter correspond à une scène, contenant des objets, une ou plusieurs sources de lumières, et une caméra virtuelle. Depuis la caméra virtuelle, on lance un rayon pour chaque pixel de l'image, et pour chacun de ces rayons, on détermine la contribution des sources de lumière pour ce rayon, et sur quels objets ce rayon rebondit. Ainsi, à chaque rayon, on récupère la couleur de l'objet que l'on a rencontré et qui est le plus proche de la caméra, et la contribution de la lumière à ce pixel. Ce principe permet de simuler le comportement de la lumière, simplement en prenant le trajet inverse des rayons (de la caméra vers la lumière). Le principe de réciprocité d'Helmholtz justifie qu'une telle méthode soit efficace pour représenter des scènes.
Ainsi, pour chaque pixel, on lance un rayon, représenté par le centre de la caméra, et la direction de ce rayon, qui passe au centre du pixel. La direction du rayon lancé est calculée de la manière suivante : les coordonnées en **x** et **y** sont calculées directement avec la position du pixel dans la grille : **j - W / 2** et **i - H / 2**, avec H et W respectivement la hauteur et la largeur de la grille. Ce calcul simplifié est possible car la caméra est positionnée en (0,0,55). La composante en Z est calculée avec le FOV selon la formule : **-W/(2*tan(fov/2))**.

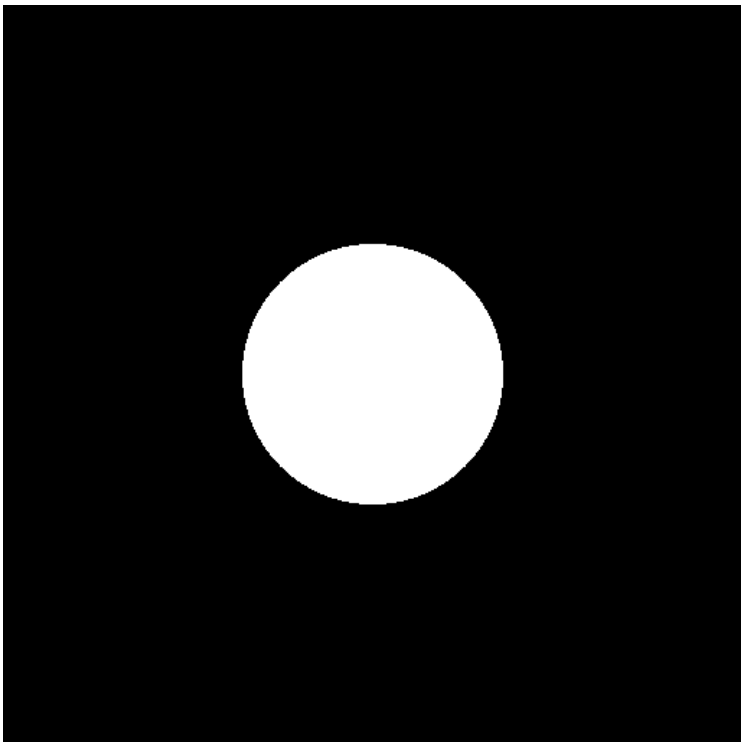
Création des objets : sphères

Une fois le lancer de rayon créé, il faut des objets à représenter, caractérisés par leur position et leur couleur au moins, pour qu'il soit possible de dire si le rayon intercepte ses objets et pour récupérer la couleur au point d'intersection. Le premier type d'objet créé est la sphère, car l'équation à résoudre pour savoir si le rayon intercepte la sphère est relativement simple. Il suffit en effet de résoudre le système suivant :

$$|| P - O ||^2 = R^2$$
$$P = C + t.u$$

où P est le point d'intersection, O le centre de la sphère, R son rayon, u le rayon et C le point de lancer du rayon. L'inconnue est t distance entre P et C.
On crée ainsi une classe *Sphere*, caractérisée par la position du centre de la sphère, son rayon et sa couleur. Cette classe est dotée d'une méthode *intersection* prenant en argument un rayon, et résolvant le système donné précédemment. Si il y a intersection, la méthode renvoie les coordonnées du point d'intersection, la distance entre ce point et le point de lancer du rayon, et la couleur du point. Toutes ces valeurs sont utiles pour représenter le pixel.

Pour tester cet ajout, on crée donc une sphère de couleur blanche que l'on place au centre (0,0,0) de la scène. On lance l'algorithme de lancer de rayons avec cette sphère, et on obtient le résultat suivant :



Création d'une classe Scene

Dans l'état, le rendu ne prend en compte qu'un seul objet. Pour pouvoir en prendre en compte plusieurs, il faut pouvoir lancer la méthode d'intersection sur tous les objets que l'on souhaite afficher, et garder pour chaque pixel la couleur du pixel le plus proche. Cette méthode sera contenue dans une classe *Scene* qui contiendra tous les objets que l'on

souhaite représenter, et la méthode permettant de calculer l'intersection sur tous ces objets. Cette méthode itère sur tous les objets de la scène et lance leur méthode d'intersection, et garde les caractéristiques de l'objet intersecté le plus proche du point (couleur, distance …).

Cette classe sera utilisée par la suite pour représenter plusieurs objets. Avant cela, nous allons ajouter un effet de lumière, pour donner de la perspective au rendu.

Source de lumières et rendus de plusieurs sphères

Lumière

Pour ajouter de la profondeur aux objets, nous allons modifier le calcul de la couleur des pixels pour ne plus simplement prendre en compte la couleur de l'objet, mais calculer la couleur du point d'intersection en fonction de sa position par rapport à une source de lumière. On fait ce calcul dans le cas où la lumière est ponctuelle et omnidirectionnelle, et la surface des objets est diffuse dans toutes les directions. Le calcul est le suivant :

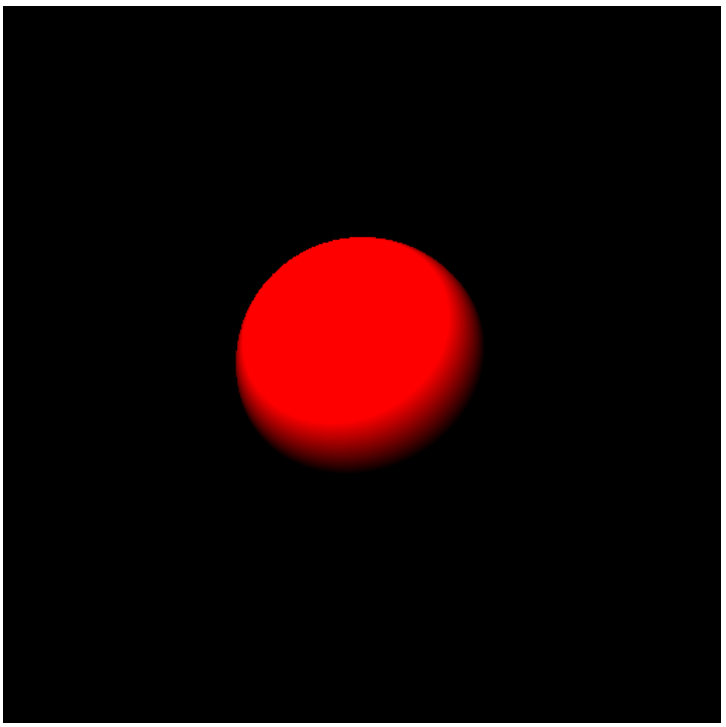
$$L = \frac{I}{4\pi d^2}\frac{\rho}{\pi} < N, \omega_i >$$

où I est l'intensité de la lumière, d la distance entre le point d'intersection et le centre de la lumière, ρ la couleur de l'objet, N la normale à la sphère au point d'intersection, et ω_i le vecteur allant du point d'intersection à la lumière.
Le calcul de la couleur est donc maintenant implémenté comme ci-dessous :

```
bool inter = S.intersect(rayon, P, N);
Vector color(0,0,0);
if (inter){
    Vector PL = L - P;
    double d = sqrt(PL.Norme());
    color = I/(4*M_PI*M_PI) * rho/M_PI * std::max(0., dot(N, PL/d)) / (d*d);
}
```

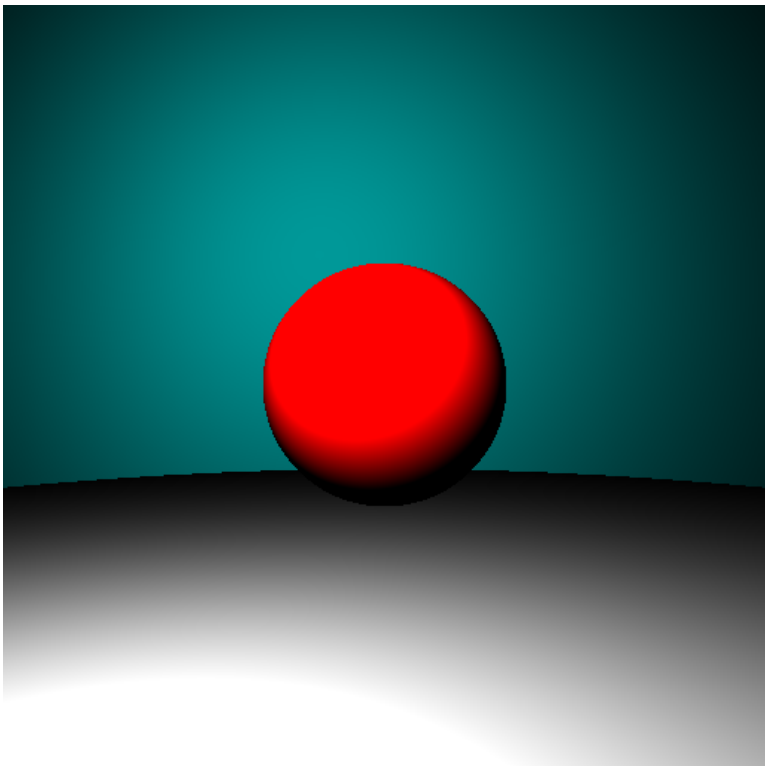
où P est le point d'intersection et L les coordonnées de la source de lumière.

On applique cette nouvelle méthode à la sphère précédente (devenue rouge) et on obtient le résultat suivant :



Ajout de “murs” à la scène

Maintenant que la lumière est implémentée, on souhaite rajouter des “murs” à la scène. Pour ce faire, on rajoute des sphères éloignées de l’origine avec des rayons très grands pour donner l’illusion de parois plates. Cet ajout permet par ailleurs de tester la classe *Scene* implémentée précédemment. Le résultat est le suivant :



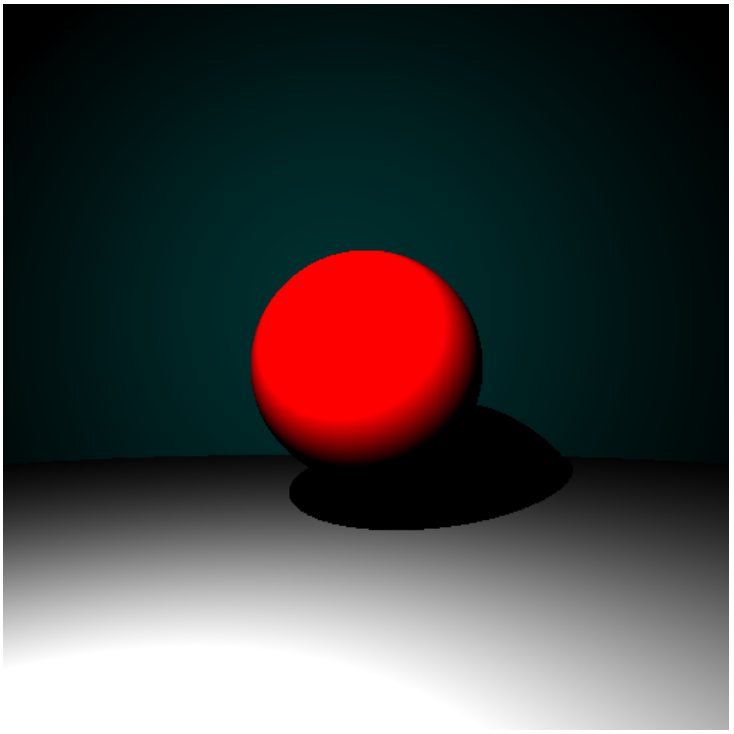
Ombres portées

Comme le montre l'image ci-dessus, pour un rendu réaliste, il devrait y avoir une ombre sur la sphère représentant le sol, générée par la sphère rouge principale (la source de lumière se situe "en haut à gauche" de la scène. Pour implémenter cette fonctionnalité, on ajoute une étape après le calcul d'intersection, qui consiste en vérifier s'il y a un obstacle entre le point et la lumière. Pour cela, on tire un rayon depuis le point vers la source de lumière, et on appelle la méthode d'intersection de la scène. Si cette méthode renvoie effectivement une intersection, dont la distance est inférieure à la distance entre le point et la lumière, alors le point se situe dans une ombre, donc le pixel associé sera noir. Cette mécanique est implémentée dans la fonction principale de la manière suivante :

```
bool inter = scene.intersect(rayon, P, N, albedo, t);
Vector color(0,0,0);
if (inter){
    Vector PL = L - P;
    double d = sqrt(PL.Norme());
    Vector shadowP, shadowN, shadowAlbedo;
    double shadowt;
    Ray shadowRay(P+0.001*N,PL/d);
    bool shadowInter = scene.intersect(shadowRay,shadowP, shadowN, shadowAlbedo, shadowt);

    if(shadowInter && shadowt < t) {
        color = Vector(0.,0.,0.);
    } else {
        color = I/(4*M_PI*M_PI) * albedo/M_PI * std::max(0., dot(N, PL/d)) / (d*d);
    }
}
```

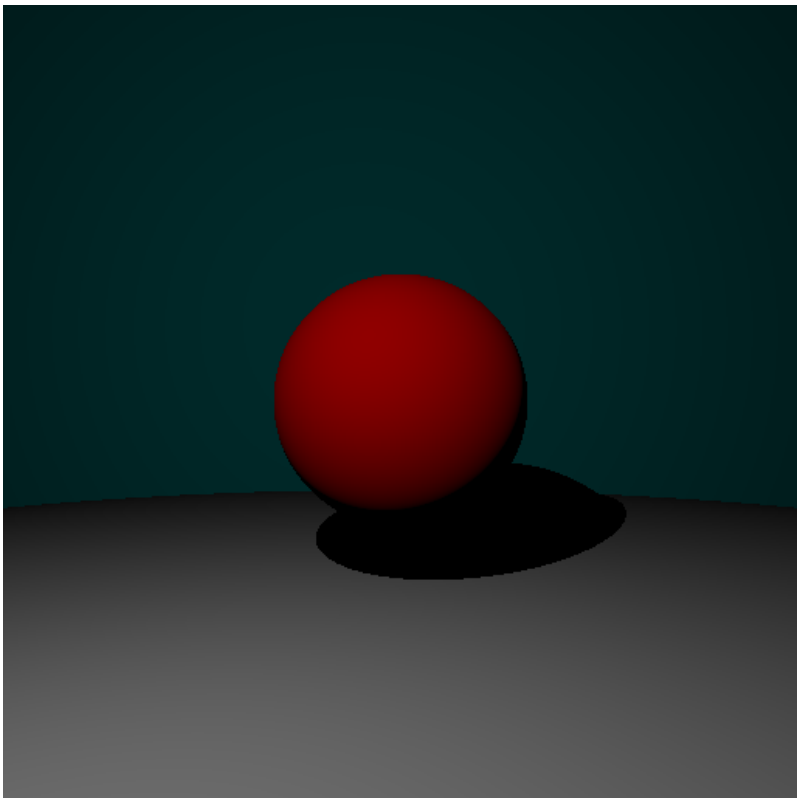
La même scène que précédemment nous donne le résultat suivant :



On observe bien l'ombre créée par la sphère principale.

Correction gamma

On ajoute à ce stade, une amélioration indépendante de ce qui a été fait jusqu'à présent. On ajoute au rendu la correction gamme, c'est-à-dire appliquer un facteur à la couleur des pixels pour compenser la correction qu'effectuent automatiquement les écrans. Pour appliquer la correction gamma, il faut mettre la couleur du pixel à la puissance 1/2.2, soit environ 0.45. Cela permet d'avoir des couleurs plus équilibrées. L'image suivante montre l'application de la correction gamma à la scène précédente.

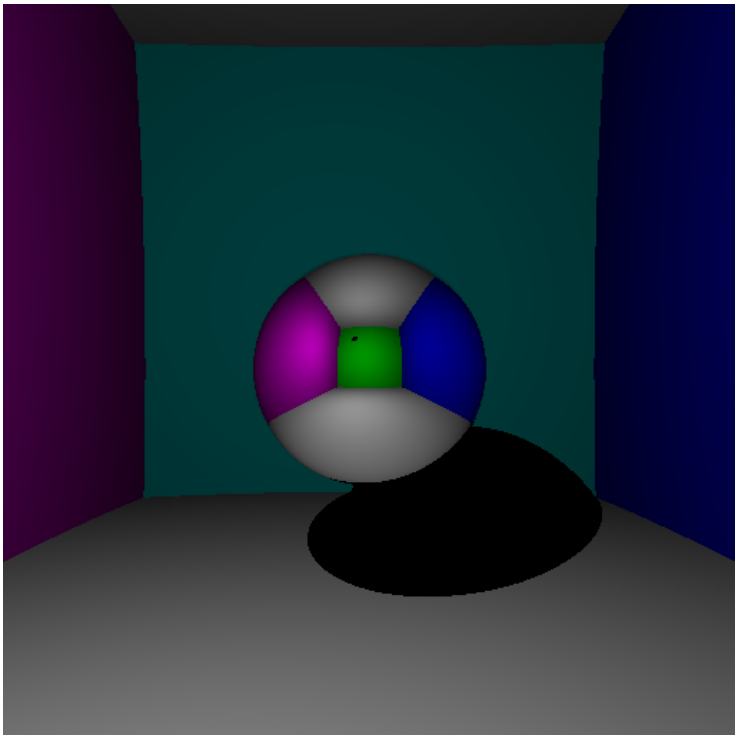


Différentes surfaces : spéculaires et transparentes

Surfaces spéculaires

Par la suite, nous implémentons différents types de surfaces pour les sphères, à commencer par les surfaces spéculaires, c'est-à-dire miroir. Le principe est que si la surface est miroir, plutôt que de prendre la couleur de l'objet, on fait rebondir le rayon, et on prend la couleur du premier objet rencontré à la place. Cela revient à faire la même routine d'intersection qu'avant, mais à partir du premier point d'intersection, et avec pour rayon le rayon réfléchi (calculé avec la normale). Le changement dans le code est que la fonction effectuant la routine de récupération de couleur et déportée de la fonction main vers une méthode de la classe *Scene*, et est appelée récursivement à chaque fois que le rayon atteint une surface spéculaire. On ajoute à la classe *Sphere* un attribut *isMirror* pour déterminer si sa surface est miroir ou diffuse. Cet attribut est renvoyé comme résultat de la fonction *intersect*, permettant à la fonction de calcul de la couleur de s'adapter en fonction de la surface

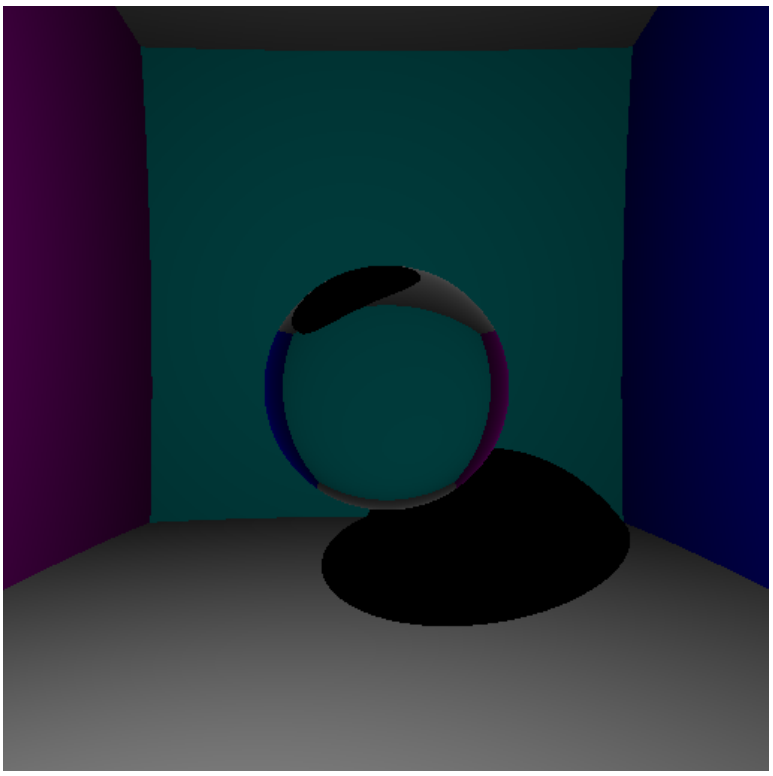
rencontrée. Le rendu pour une sphère miroir est le suivant (d'autres sphères ont été ajouté à la scène pour voir plus de choses dans la sphère miroir) :



NB : le point noir correspond à la position de la lumière dans la scène.

Surface transparente

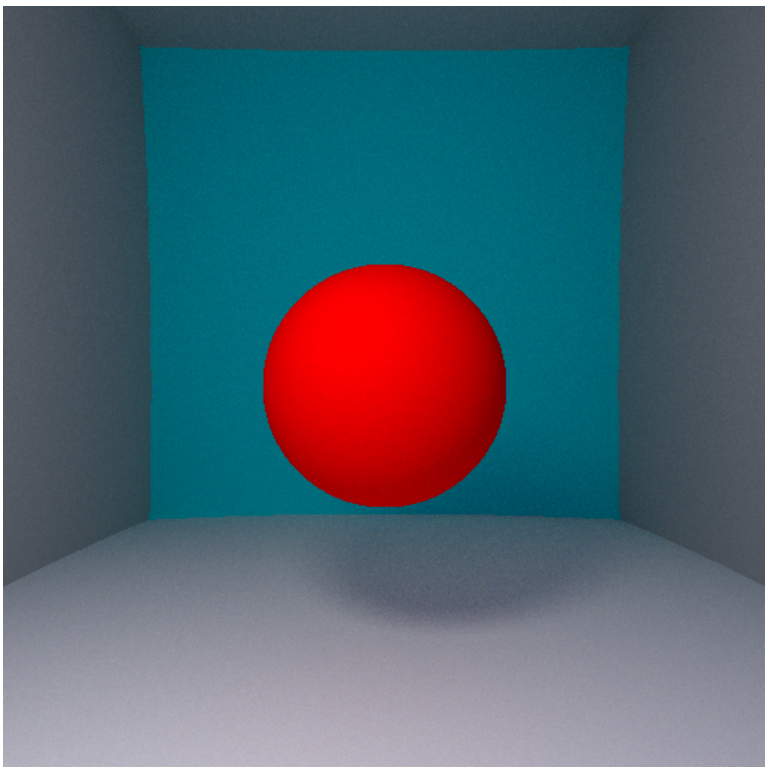
Un autre type de surface implémenté est les surfaces (ou objet) transparentes. De la même façon que pour les surfaces miroirs, quand un rayon rencontre une surface transparente, il est redirigé, et la fonction pour obtenir la couleur est appelée récursivement. La différence est que le rayon redirigé est calculé à l'aide de la loi de Descartes et dans indices de réfraction des deux milieux. Ainsi, on rajoute à la classe *Sphere*, un attribut *transparent*. On peut rajouter également un attribut donnant l'indice de réfraction, mais on se contente pour ce travail, de considérer les sphères transparentes comme ayant l'indice de réfraction du verre. Le résultat est le suivant :



Eclairage indirect

Eclairage indirect et lumière étendue

Grâce à l'implémentation de code permettant de gérer les rebonds de rayon, on peut maintenant améliorer l'éclairage de la scène en implémentant l'éclairage indirect. Le principe est de rajouter, à la contribution de l'éclairage direct, les contributions des autres sources de lumières, à savoir la diffusion des autres objets. Il faut donc, pour avoir la couleur du point, calculer la totalité des contributions extérieures, en fonction de la BRDF du matériau. Le calcul à réaliser est un calcul intégral, on utilise la méthode de Monte Carlo pour approximer ce calcul. On tire donc plusieurs rayons depuis le point d'intersection trouvé, tirés aléatoirement, qui vont rebondir récursivement dans la scène pour récupérer la contribution indirecte de la lumière. Plus on tire de rayons, plus l'éclairage indirect récupéré est de qualité.
L'ajout de cette mécanique donne le résultat suivant :



Cette image a été réalisée avec 100 rayons tirés par pixel, l'image est donc peu bruitée.

NB : Au passage, le rendu est passé d'une lumière ponctuelle à une lumière étendue. La lumière est maintenant une sphère appartenant à la scène, qui possède donc un rayon. Pour obtenir la contribution de l'éclairage direct, on ne tire plus un simple rayon vers la source de lumière, mais on tire un rayon aléatoire, centré sur le vecteur reliant le point au centre de la source de lumière. De plus, la contribution de la lumière dépend du rayon de la sphère représentant la lumière. Cela a pour effet d'adoucir les ombres pour les rendre moins discontinues, ce que l'on observe sur l'image précédente.

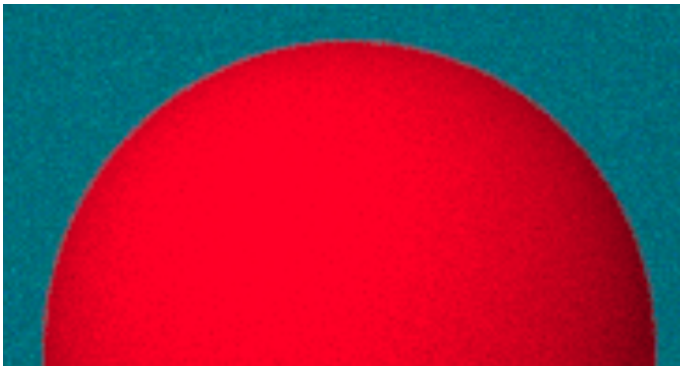
Amélioration des performances de calcul : parallélisation

L'implémentation de l'éclairage indirect rajoute beaucoup de temps de calcul au programme. Le fait de devoir tirer plusieurs rayons par pixel démultiplie directement le temps de calcul. Ainsi, si on désire une image de bonne qualité, le temps de calcul s'en trouve drastiquement augmenté. Pour accélérer le calcul du rendu, on parallélise le calcul sur les différents pixels, pour utiliser au mieux les ressources de l'ordinateur et réduire le temps de calcul. On utilise pour cela la bibliothèque openmp de c++. On place l'instruction de parallélisation sur la boucle de lancer de rayon. Pour la scène ci-dessus, pour 20 rayon par pixel, le temps de calcul devient :

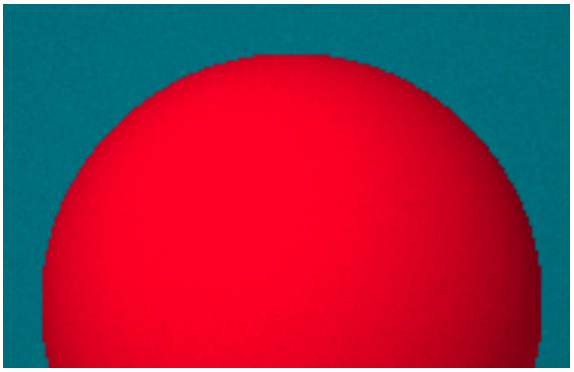
- **Sans parallélisation : 20.9s**
- **Avec parallélisation : 3.9s**

Anti-aliasing

L'implémentation de la méthode de Monte-Carlo permet par la même occasion d'implémenter l'anti-aliasing. L'anti-aliasing permet d'adoucir l'effet de crénelage, qui rend les arêtes discontinues. Sur nos images, cela est visible particulièrement sur le contour de la sphère principale. Pour palier à ce problème, plutôt que de tirer le rayon au centre de chaque pixel, on échantillonne chaque rayon aléatoirement à l'intérieur de chaque pixel, en suivant une loi normale pour favoriser les rayons proches du centre. Cela permet d'obtenir une transition plus douce entre les pixels et réduire l'effet de crénelage.
L'implémentation de l'anti-aliasing donne le résultat suivant :



Rendu avec anti-aliasing

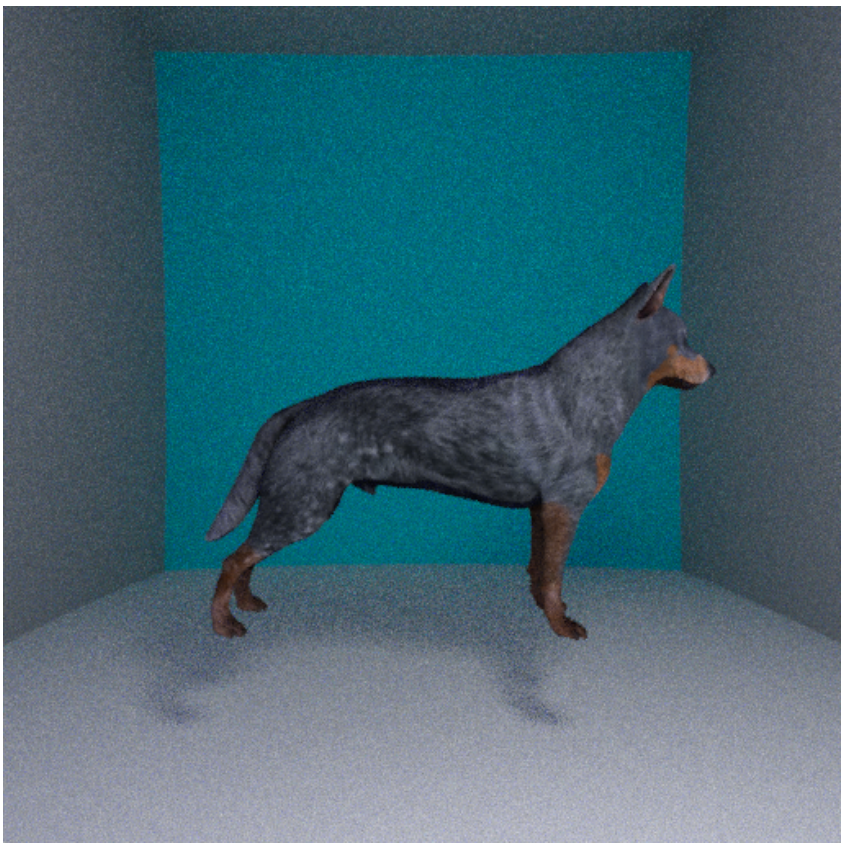


Rendu sans anti-aliasing

Maillages

Maillage et texture

Jusqu'à maintenant, les seuls objets représentés étaient des sphères. Nous allons maintenant rajouter au code la possibilité de charger et de représenter des maillages. Pour ce faire, on récupère une classe préfaite *TriangleMesh*, qui peut ouvrir un fichier *obj* et charger les sommets des triangles et les normales dans différentes listes. On rajoute également une fonction *loadTexture* qui permet d'ouvrir un fichier de texture et de stocker ces textures dans des listes.
On complète cette classe avec la méthode *intersect* permettant de l'intégrer à notre programme. Cette méthode parcourt tous les triangles du maillage, et pour chaque triangle, calcule si le rayon intersecte ce triangle (par les coordonnées barycentriques). On récupère le point d'intersection le plus proche de la caméra, et grâce aux indices du triangle concerné, récupère la couleur de la texture correspondante.
L'image ci-dessous montre la représentation d'un maillage dans la scène vu précédemment.



Optimisation : bounding box

La méthode de calcul d'intersection présentée précédemment augmente énormément le temps de calcul, proportionnellement au nombre de triangles du maillage. Pour optimiser le temps de calcul, on rajoute la mécanique de boîte englobante. Le principe est de placer le maillage dans une boîte rectangulaire, et d'effectuer d'abord un calcul d'intersection avec cette boîte (revenant à calculer l'intersection entre le rayon et plusieurs plans) avant de le faire pour tout le maillage. On crée donc une classe *Bounding_box* contenant une méthode *intersect*, puis on crée une méthode dans la classe *TriangleMesh* permettant de créer une boîte englobant tout le maillage. Cette méthode itère simplement sur tous les sommets du maillage, et garde les plus grandes coordonnées à chaque fois en tant que points extrêmes de la boîte pour s'assurer qu'elle contienne tout le maillage.
Cette méthode permet donc de gagner du temps de calcul, puisque l'on s'évite un parcours de tous les triangles si le rayon n'intersecte pas la boîte. Pour une image de 128x128, avec le maillage précédent sans texture, et 3 rayons par pixel, le temps d'exécution sur mon ordinateur était de :

- **460 secondes sans boîte englobante**
- **38 secondes avec boîte englobante**

Pour aller plus loin dans l'optimisation, on peut créer une hiérarchie de boîtes englobantes. Le principe est de créer plus de boîtes, contenues les unes dans les autres, et englobant de moins en moins de triangles. Lorsqu'un rayon intersecte une boîte, relance le test d'intersection sur les deux boîtes enfant, on garde la boîte intersectée par le rayon puis on refait le test sur les deux boîtes enfant, et ainsi de suite. L'avantage est qu'à chaque

nouveau test, on réduit grandement le nombre de triangles potentiellement intersectés, puisque tous ceux qui étaient contenues dans l'autre boîte ne sont alors plus considérés. Ainsi, plutôt que de parcourir tous les triangles, on effectue des tests de boîte en boîte, avant de parcourir les triangles contenues dans la boîte la plus petite rencontrée, ce qui réduit grandement le nombre de triangles à parcourir.

On teste ces changements avec les mêmes paramètres que précédemment, on trouve donc :

- **38 secondes avec simple boîte englobante**
- **0,3 secondes avec la hiérarchie**

Les différentes optimisations implémentées sont donc très efficaces pour réduire le temps de calcul et permettre des rendus de meilleures qualités en moins de temps.