TP MSO Synthèse d'images: Lancé de rayons
CPE

durée - 4h

2012

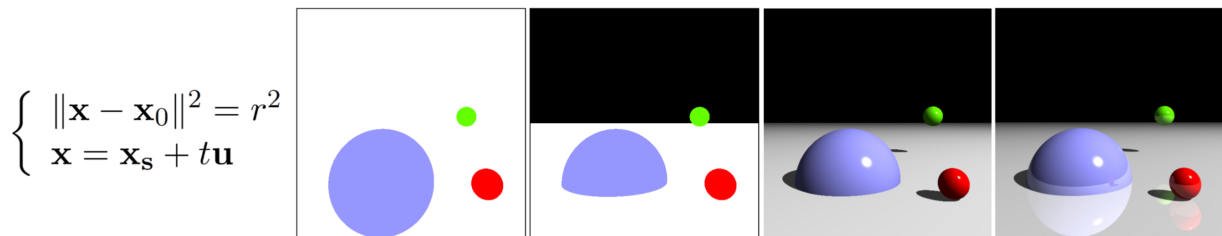

FIGURE 1 – Étapes de l'algorithme de lancé de rayons. De gauche à droite : équation du calcul d'intersection ; image des intersections ; ordonnancement des intersections suivant leur profondeur ; calcul d'illumination et d'ombrage ; réflexions.

1 But

L'objectif de ce TP est de coder un outil de rendu par lancé de rayons (ray-tracing) tel qu'on peut le trouver dans différents outils de rendu off-line (PovRay, Yafray, etc . . .).

Nous mettrons en avant les avantages et inconvénients de cette approche par rapport au rendu projectif basé sur des triangles.

- Dans un premier temps, nous mettrons en place l'intersection entre des rayons (droites) et des primitives géométriques simples.
- Dans un second temps, nous implémenterons le calcul de la couleur associé à chaque intersection.
- Enfin, nous pourrons mettre en place différents effets réalisables aisément par lancé de rayons tels que la réflexion, l'anti-aliasing, . . .

2 Prise en main de l'environnement

2.1 Programme main

La fonction *main* réalise l'appel et l'affichage d'une scène 3D.

Les appels sont, dans l'ordre d'exécution :
- Initialisation et remplissage d'une scène par des objets 3D (spheres+plan) colorés (couleur+type d'illumination).
- Création d'un buffer d'image et appel à l'algorithme de lancé de rayons dans la scène 3D sur l'image.
- Écriture de l'image dans un format ascii classique *ppm* (non compressé). L'image pouvant être manipulée par d'autres outils annexes : gimp, ...

Les différents appels sont présentés en fig. 2.

```
try
{
    //common shading parameters
    shading shad(0.2,1.0,0.5,64.0);

    //render parameters
    render_parameters render_param(3,5,true);

    //define an empty scene
    scene3d scene;

    //camera
    scene.set_camera(camera(v3(0,0,-2),v3(0,0,1),v3(0,1,0),2.0,1.2));
    //add some spheres
    scene.add(new sphere(v3(-0.5,1,2),0.9),material(color(150,150,255),shad));
    scene.add(new sphere(v3(1,1-0.2,1),0.2),material(color(255,0,0),shad));
    scene.add(new sphere(v3(+1.7,-0.2,5),0.3),material(color(100,255,0),shad));
    //add a plane
    scene.add(new plane(v3(0,+1,0),v3(0,-1,0)),material(color(255,255,255),shad));

    //add a light
    scene.add(light(v3(6,-5,0)));

    //background color
    scene.set_background_color(color(0,0,0));

    //generate an empty picture
    image pic(500);

    //run ray-tracing
    ray_tracer::trace(scene,render_param,&pic);

    //delete allocated memory
    scene.clean_memory();

    //export picture in ppm format
    pic.export_file("my_pic.ppm");
}
```


FIGURE 2 – Fonctions appelées depuis le main original.

Une fois l'ensemble des fonctions complétée, l'image de sortie doit représenter une vue de 3 sphères et d'un plan similaire à la fig. 7.

2.2 Classes utilisables

Un ensemble de classes de bases vous est fourni pour faciliter la mise en place de ce TP. L'ensemble des classes reste cependant bas-niveau et elles ne font pas appels à de librairies externes. Elles restent donc modifiables pour votre TP.

2.2.1 Classe p2d

Une classe de conteneur de base de 2 entiers. Elle sert principalement à indexer un pixel de coordonnées (kx,ky) dans une matrice.

```
p2d u(5,4);    //u=(5,4)
u.x()=8;       //u=(8,4)
u=2*u-p2d(1,0) //u=(15,8)
```

2.2.2 Classe v3

Classe de conteneur de point 3D quelconque. En interne (x,y,z) étant stockée sur 3 double. Contiens de nombreuses fonctions vectorielles.

```
v3 p(1.5,1.0,-2);         //p=(1.5,1.0,-2.0)
v3 o=p.dot(v3(1,1,0))*p;  //o=<p,(1,1,0)> p
o+=p;                     //o=o+p
v3 e=o.normalized();      //e=o / ||o||
e.z()=4;                  //e=(e.x,e.y,4)
```

2.2.3 Classe color

Classe de conteneur d'une couleur (r,g,b) où chaque canal est encodé sur un entier. Notez que la méthode static *interpolate linear* implémente l'interpolation linéaire entre 2 couleurs sur des entiers.

```
color c(255,255,0) // c <- jaune
c.b()=255;         // c <- blanc
color bleu=(0,0,255);

//magenta = (1-0.4)*rouge + 0.4*bleu
color magenta=color::interpolate_linear(color(255,0,0),bleu,0.4);

//ou pour un nombre quelconque de couleurs:
std::vector<color> vc;std::vector<double> poids;
vc.push_back(color(255,0,0));poids.push_back(1.0/3.0);
vc.push_back(color(255,255,0));poids.push_back(1.0/3.0);
vc.push_back(color(255,255,255));poids.push_back(1.0/3.0);
//final=poids[0]*vc[0]+poids[1]*vc[1]+poids[2]*vc[2];
color final=color::interpolate_linear(vc,poids);
```

2.2.4 Classe image

Classe de gestion d'une image (r,g,b). L'image est stockée en interne sous forme de vecteur<color> $= (r_0, g_0, b_0, r_1, \ldots, b_{N-1})$. La classe implémente l'initialisation, l'accès protégé aux données de couleurs et l'export d'une image dans un fichier *ppm*.

```
image pic(500); // cree une image 500x500
pic.fill(color(255,0,0)); // colore l'image en rouge

// colore pixel(10,15) en vert
pic.set_pixel(p2d(10,15),color(0,255,0));

//exporte l'image dans fichier mon_pimage.ppm
pic.export_ppm(``mon_image.ppm'');
```

2.2.5 Classe ray

Classe définissant un rayon (droite infinie orientée). La droite est définie par une position origine x_0 et un vecteur directeur unitaire $\mathbf{u}$.

```
//création d'un rayon orienté suivant l'axe x
ray r(v3(0,1,0)v3(1,0,0));

//evalue x0+5.5*u
v3 y=r(5.5);//y=(5.5,1,0)
```

2.2.6 Classe object3d

Classe virtuelle pure d'un objet 3D générique. L'objet étant défini par l'intersection entre une droite infinie (définit par la classe *ray*) et l'objet lui même. Tout objet d'une scène 3D doit dériver de cette classe générique.

On notera que les intersections avec la droite sont stockées dans un vecteur contenant des struct *intersection data* contenant la position de l'intersection, la normale et la position relative t sur la droite définie telle que $x_{\text{inter}} = x_0 + tu$.

2.2.7 Classe plane

Classe implémentant un *object3d*. Un plan est défini par une position x_0 et une normale $\mathbf{n}$. Le calcul de l'intersection est déjà complet.

2.2.8 Classe sphere

Classe implémentant un *object3d*. Une sphère est définie par un centre x_0 et un rayon r. Le calcul de l'intersection est à compléter.

2.2.9 Classe scene3d

Conteneur d'un ensemble d'objets 3D. La classe stocke un vecteur d'objets 3D (sous forme de pointeurs afin de profiter du polymorphisme), ainsi que leur *matériaux* (couleur+illumination) associé. Les lumières sont stockées dans un autre vecteur.

Si tous les objets 3D ont été alloués dynamiquement, la libération mémoire peut se réaliser directement par la classe suite à l'appel explicite à *clean memory*.

```
//cree scène vide
scene3d scène;

//ajoute une sphère dans la scène
scene.add(new sphere(...),material(...));

//ajoute une lumière dans la scène
scene.add(light(...));

//utilisation du polymorphisme pour le calcul d'intersection
//(l'intersection appelée est celui de la sphère)
const object3d *obj=scene.get_object(0);
vector<intersection_data> intersection=obj->intersect(ray(...));

//libère la mémoire allouée pour la sphère
scene.clean_memory();
```

2.2.10 Classe ray tracer

La classe *ray tracer* est un ensemble de méthodes statiques d'aide implémentant l'algorithme du lancé de rayons. La classe est à compléter dans ce TP.

3 Définition d'un objet 3D

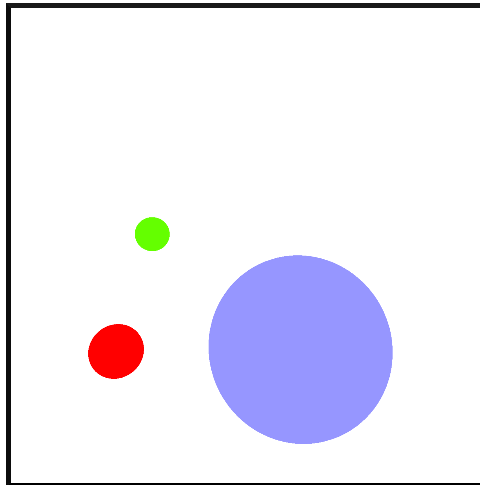

FIGURE 3 – Figure résultant de l'intersection des rayons par les 3 sphères et le plan. Notons que les couleurs des objets sont directement affectées aux pixels, et que l'ordre d'intersection par rapport à la caméra n'est pas pris en compte.

Pour la méthode de lancé de rayon, un objet 3D O est défini uniquement par ses intersections avec une droite D. Notez qu'aucune autre expression de l'objet n'est indispensable (ex. expression paramétrique, implicite, ...).

Chaque objet 3D de la scène doit être en mesure de retourner sa/ses intersections (s) avec une ligne droite. La méthode *intersect* commune à tous les objets 3D réalise ce calcul. Cette méthode renvoie un vecteur contenant toutes les intersections avec la ligne droite définie par le paramètre *ray*.

Chaque intersection contient les données des coordonnées de l'intersection, de la normale, et de la position relative de l'intersection sur le rayon.

La méthode

```
std::vector<intersection_data> sphere::intersect(const ray& seg) const
```

doit retourner l'(les) intersection(s) entre la sphère et une droite définie par le rayon passé en paramètre. On notera que le calcul d'intersection peut retourner l'ensemble des intersections, peu importe leurs positions relatives par rapport à l'écran.

La structure retournée par la méthode intersect est un vecteur d'intersection, où chaque intersection est une structure du type *intersection data*, avec :

```
struct intersection_data
{
  v3 x; //position de l'intersection
  v3 n; //normale a l'intersection
  double t; //position relative le long du rayon
};
```

L'intersection entre une sphère de centre $\mathbf{x}_0$ et de rayon r avec une droite passant par x_s et de vecteur directeur u est donné par la solution du système

$$\begin{cases} \|\mathbf{x} - \mathbf{x}_0\|^2 = r^2 \\ \mathbf{x} = \mathbf{x}_\mathbf{s} + t\mathbf{u} \end{cases}$$

Question 1 *Déterminer le paramètre t_{inter} solution du système d'équations, et donnez la position $\mathbf{x}_{inter}$, ainsi que la normale associée.*

Question 2 *En vous inspirant du calcul de l'intersection pour le plan, écrivez la méthode* sphere : :intersect.

Note : À cette étape, on devra obtenir une image telle que celle montrée en figure 3.

4 Ray-tracing

On s'intéresse maintenant à l'algorithme du ray-tracing proprement dit.

Question 3 *Observez la méthode* ray tracer : :trace(scene3d,render_param,image*). *Expliquez par un schéma ce que réalise l'appel à* ray : :generate ray from camera *avec les parametres donnés.*

4.1 Calcul du premier objet intersecté

La méthode *ray tracer : :find inter* retourne la première intersection entre un rayon orienté et les objets de la scène 3D.

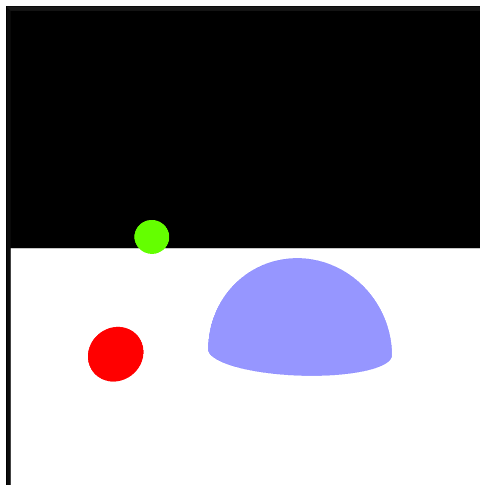

FIGURE 4 – Figure résultant de l'intersection des rayons par les 3 sphères et le plan. Les couleurs sont toujours directement affectées aux pixels, mais l'ordre des intersections est cette fois correctement prise en compte. Notons que le plan coupe bien la sphère bleue en deux, et que seule la partie du plan situé à l'avant de la caméra est affichée.

Le type de retour est un *intersection object* avec

```
struct intersection_object
{
  //donnees de l'intersection
  intersection_data inter_data;

  //identifiant de l'objet
  int id_object;
};
```

L'identifiant de l'objet correspond à sa position dans le vecteur de la *scene3d* si il y a au moins une intersection. Dans le cas où le rayon n'intersecte aucun objet, alors cet identifiant sera positionné à une valeur < 0.

Soit $(t_1, \ldots, t_n)$ les paramètres de position relative des intersections des objets de la scène avec le rayon courant. La toute première intersection t_{k_0} est caractérisée par deux contraintes

$$\begin{cases} t_{k_0} > 0 \\ \forall k \in [\![1, n]\!]/k_0 \ , \quad t_k > 0 \Rightarrow t_k \geq t_{k_0} \end{cases}$$

Question 4 *Étant donné un vecteur d'intersections, donner l'algorithme permettant de déterminer la première rencontrée ?*

Question 5 *Ecrivez la méthode* ray tracer : :find inter *afin de retourner la toute premiere intersection rencontrée lorsque celle-ci existe.*

Note : On pourra vérifier que le résultat obtenu est cohérent avec celui présenté en fig. 4.

4.2 Calcul de la couleur de l'objet

La couleur appliquée sur un pixel est calculée par la méthode *ray tracer : :find intersection color*. Sont comportement est le suivant :

- Si il existe au moins une intersection, calculer la couleur en fonction de l'objet et des lumières.
  - i. Si une lumière est directement visible, on applique un calcul d'illumination de type Phong.
  - ii. Si la lumière est cachée par un autre objet (lancé de rayon du point d'intersection vers la lumière), celui-ci est dans l'ombre, et seule la couleur ambiante de l'objet est attribuée.
- Si il n'existe aucune intersection avec le rayon courant, renvoyer la couleur de fond.

Jusqu'à présent lorsqu'une intersection existe, la couleur du pixel est directement issue de la couleur de base de l'objet. Pour obtenir une illumination de type Phong ainsi que les ombres, il est nécessaire de prendre en compte la normale de la surface au point d'intersection. Il est alors possible d'appliquer le calcul d'illumination vue précédemment ainsi que la position de la lumière.
Pour s'aider, on pourra utiliser la classe *vertex shader* qui aura été complété au TP précédent, ou que vous compléterez lors de ce TP.

4.2.1 Ombres

Nous allons compléter la méthode *ray tracer : :ray to light* qui, étant donné la position de l'intersection viens lancer des rayons en direction de la ou les source(s) de lumière.

Soit $\mathbf{p}_i$ le point d'intersection courant avec la surface. Si $\mathbf{p}_i$ n'est pas dans l'ombre d'un autre objet, alors la fonction retourne la distance entre $\mathbf{p}_i$ et la source lumineuse, sinon elle retourne une valeur négative. Notez que l'on retourne un vecteur de valeur dans le cas où la scène contient plusieurs sources lumineuse.

Question 6 *Ecrivez l'algorithme permettant de calculer la valeur de retour de cette fonction.*

Question 7 *Implémentez la méthode* ray tracer : :ray to light.

(Notez que l'on pourra utiliser la méthode de calcul d'intersection précédente).

La méthode *ray tracer : :compute color* calcule la couleur associé à une intersection étant donné les valeurs associé à celle-ci ainsi que le vecteur de distance entre l'intersection et les sources de lumière.

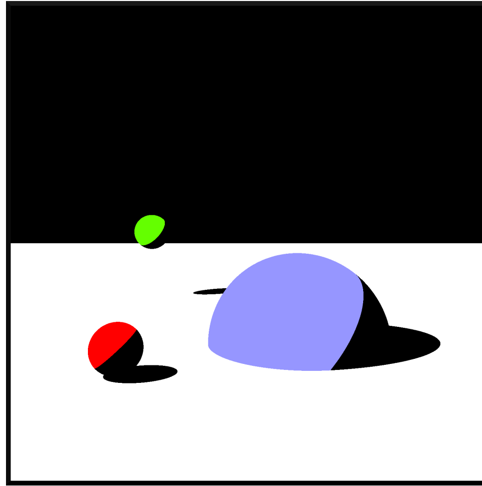

FIGURE 5 – Résultat obtenue après prise en compte des ombres (couleur mise à (0,0,0) lorsqu'une ombre est détectée).

Question 8 *Modifiez cette méthode de manière à visualiser des ombres (voir fig. 5).*

4.2.2 Illumination

Étant donné les paramètres d'illuminations, la normale à la surface, une illumination de type Phong peut être calulée par la classe *vertex shader*

Question 9 *Complétez cette classe si cela n'a pas été fait au TP précédent.*

Question 10 *Finissez l'implémentation de la méthode* ray tracer : :compute color *de manière à calculer une illumination de type Phong. Le résultat obtenu devant s'approcher de l'illustration en fig. 6.*

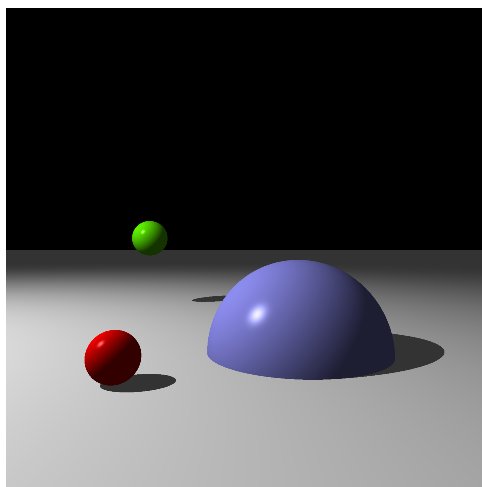

FIGURE 6 – Prise en compte d'une illumination de type Phong.

4.3 Rayons réfléchis

La méthode du ray tracing permet de prendre en compte aisément la réflexion des rayons sur des objets.

Considérons un rayon intersectant de direction unitaire $\mathbf{r}$ intersectant un objet dont la normale unitaire au point d'intersection est donnée par $\mathbf{n}$. Il est alors possible de lancer un second rayon réfléchi dans la direction $\mathbf{r}_2$ symétrique de $\mathbf{r}$ par rapport à $\mathbf{n}$. C'est à dire

$$\mathbf{r}_2 = \mathbf{r} - 2 < \mathbf{r}, \mathbf{n} > \mathbf{n} \,.$$

Ce nouveau rayon venant alors intersecter potentiellement un nouvel objet et apporter une couleur donnée.
L'algorithme de ray-tracing vu précédemment se récursifie pour un nombre quelconque de réflexions.

La couleur finale d'un pixel est alors donnée par la somme pondérée des couleurs de chaque rayon à chaque niveau de réflexion. Une amplitude de réflexion < 1 faisant diminuer l'*énergie* de chaque rayon.

Question 11 *Modifiez la méthode* ray tracer : :throw ray *afin de considérer un nombre N fixé de réflexions.*

L'application des réflexions sur l'image standard pourra être comparée à la fig. 7.

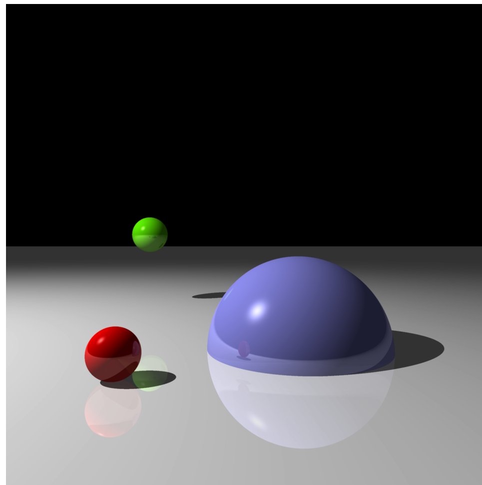

FIGURE 7 – Mise en place des reflection sur les sphères et le plan. Ici 5 niveaux de réflexions sont attribués. Chaque couleur est atténuée par un facteur de 0.2 pour chaque niveau de réflexion.

4.4 Anti-aliasing

Chaque rayon lancé est ici associé à une unique évaluation de couleur par pixel. Cet échantillonnage simple pour chaque pixel peut présenter des inconvénients visibles dans les cas suivants :
- On visualise une surface ou une texture oscillant rapidement (typiquement une texture vue de loin). Le non-respect des critères d'échantillonnages introduit un effet de recouvrement gênant.
- On s'intéresse au bords des objets qui présentent des coupes pixelisées franches.

Pour améliorer l'échantillonnage, il est possible de lancer plusieurs rayons pour chaque pixel, et de moyenner la couleur résultante en fonction de ces échantillons. Un exemple de résultat obtenu est illustré en fig. 8.

Question 12 *Modifiez la méthode* ray tracer : :trace *afin lancer plusieurs rayons pour chaque pixels, et moyenner la couleur résultante.*

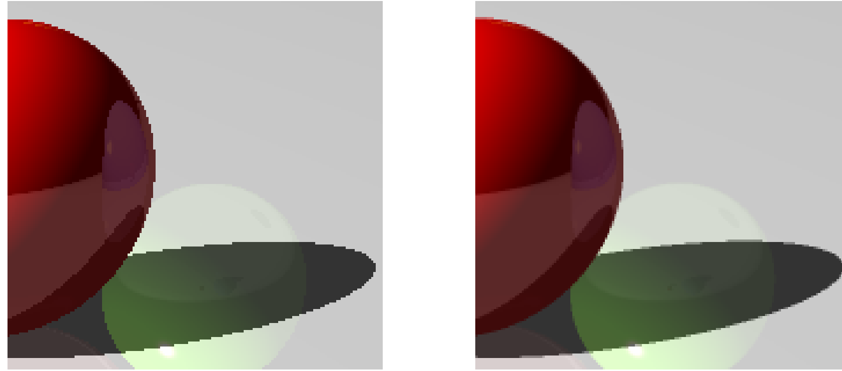

FIGURE 8 – Comparaison avant/après mise en place d'un sur-échantillannage permettant l'anti-alisasing. La figure de gauche représente un zoom sur la figure avant le sur-échantillannage, alors que la figure de droite montre le résultat après sa mise en place. Notez les transitions plus douces.

Notons que dans les moteurs actuels de ray-tracing, il est possible de mettre en place un échantillonnage adaptatif se réalisant uniquement lorsque cela est nécessaire, c'est à dire lorsque les couleurs varient localement.

5 Extensions possibles

5.1 Généralisation à d'autres primitives

Le ray tracing peut se généraliser à d'autres types de primitives. Il faut pour cela déterminer l'intersection d'une droite avec cette primitive.

Question 13 *Généralisez le calcul du ray-tracing pour d'autres primitives. En particulier, le cas du triangle qui permettra de réaliser le rendu d'une surface maillée quelconque.*
On pourra également s'interesser au cas d'une primitive cylindrique.

5.2 Parrallélisation

La méthode de ray-tracing possède l'avantage de pouvoir se paralléliser trivialement. Il est possible de prendre avantage des multi-processeurs afin d'appeler des threads permettant de calculer la couleur de différents pixels en parallèle.

Question 14 *Implémentez un tel calcul en parallèle et comparez le temps de rendu pour un calcul séquentiel total et un calcul en parallèle.*

5.3 Matériaux reflechissants

Il est possible d'étendre les types de matériaux rencontrés pour simuler d'avantages d'effets. Par exemple, on pourra prendre en compte les paramètres de l'objet lors de la reflexion afin de moduler l'amplitude de celle-ci. Par exemple, un objet de type miroir n'atténuera pas du tout le rayon réfléchi, un objet en taule réfléchiera partiellement, alors que d'autres objets seront totalement non réfléchissants.

Question 15 *Implémentez la prise en compte du type d'objet lors du calcul de la réflexion.*

5.4 Refraction

Nous avons pu mettre en place l'utilisation de rayons réfléchis par la surface. Il est également possible de considérer les réfractions. Pour cela, à chaque intersection, un rayon peut être tracé suivant les lois de Snell-Descartes tel que

$$n_1 \sin(\theta_1) = n_2 \sin(\theta_2) \ ,$$

avec n_1 et n_2 les indices optiques des milieux incidents et réfléchis, et θ_1 et θ_2 les angles incidents et réfléchis des rayons par rapport à la normale $\mathbf{n}$ à la surface.

De même que dans le cas de la réflexion, l'intensité finale est obtenue par moyenne des couleurs issue de l'ensemble des rayons.

Question 16 *Implémentez la mise en place de rayons réfractés dans votre code. Chaque objet volumique possédant alors une propriété d'indice optique.*