

# Informatique graphique

## Codage d'un Raytracer

Présentation du code et des résultats

Mars  
**31**  
2017



## Table des matières

---

<b>■ 1 Introduction . . . . .</b>	<b>6</b>
<b>■ 2 • Structure du code réalisé . . . . .</b>	<b>7</b>
<b>■ 2 • Établissement de la scène de test . . . . .</b>	<b>8</b>
2.1 Intersection entre un rayon et une sphère . . . . .	8
2.2 Ajout de la lumière. . . . .	10
2.3 Cas avec plusieurs sphères . . . . .	11
2.4 Ajout de la couleur. . . . .	12
<b>■ 3 • Ombres portées . . . . .</b>	<b>14</b>
3.1 Éléments théoriques . . . . .	14
3.2 Résultats obtenus . . . . .	15
3.2.1 Sans décolorer le point d'intersection . . . . .	15
<b>■ 4 • Matériaux . . . . .</b>	<b>16</b>
4.1 Matériau diffus . . . . .	16
4.2 Matériaux spéculaires. . . . .	16
4.2.1 Relation de réflexion. . . . .	16
4.2.2 Code associé. . . . .	16
4.2.3 Exemples graphiques . . . . .	18
4.3 Matériaux transparents. . . . .	20
4.3.1 Relation de réflexion. . . . .	20
4.3.2 Code associé. . . . .	21
4.3.3 Résultats obtenus. . . . .	22
4.4 Matériaux spéculaires et transparents . . . . .	23
4.4.1 Motivations et relations théoriques. . . . .	23
4.4.2 Code associé. . . . .	23
4.4.3 Résultats obtenus. . . . .	24
<b>■ 5 • Diverses améliorations . . . . .</b>	<b>26</b>
5.1 Correction gamma . . . . .	26
5.2 Ajout de l'antialiasing . . . . .	26
5.2.1 Problème posé. . . . .	26
5.2.2 Solution utilisée. . . . .	27
5.3 Diverses autres améliorations. . . . .	28
5.3.1 Couleur par défaut pour une scène . . . . .	28
5.3.2 Gestion des sphères lumineuses. . . . .	29
<b>■ 6 • Éclairage diffus . . . . .</b>	<b>30</b>
6.1 Aspects théoriques. . . . .	30
6.1.1 Motivations. . . . .	30
6.1.2 Généralités. . . . .	30
6.1.3 Les BRDFs . . . . .	30
6.1.4 Équation du rendu . . . . .	31
6.1.5 Méthode de Monte-Carlo . . . . .	31
6.1.6 Mettre tout ensemble . . . . .	32

6.2	Code réalisé . . . . .	32
6.2.1	Génération de la direction aléatoire . . . . .	33
6.2.2	Adaptation de la méthode <i>getColor</i> . . . . .	33
6.2.3	Envoyer plusieurs rayons par pixels . . . . .	34
6.3	Résultats obtenus . . . . .	34
<b>7</b>	<b>• Source étendue . . . . .</b>	<b>36</b>
7.1	Aspects théoriques . . . . .	36
7.1.1	Motivations . . . . .	36
7.1.2	Modélisation . . . . .	36
7.2	Code réalisé . . . . .	36
7.2.1	Modification de <i>Material</i> . . . . .	36
7.2.2	Regénérer un nouveau rayon . . . . .	37
7.3	Résultats obtenus . . . . .	38
<b>8</b>	<b>• CSG . . . . .</b>	<b>39</b>
8.1	Objectifs . . . . .	39
8.2	Structure du code . . . . .	39
8.3	Réalisation de l'union . . . . .	40
8.3.1	Présentation de la classe <i>Union</i> . . . . .	40
8.3.2	Fonction pour tester l'appartenance d'un point . . . . .	40
8.3.3	Fonction d'intersection . . . . .	40
8.3.4	Mise en pratique . . . . .	41
8.4	Réalisation de l'intersection . . . . .	41
8.4.1	Présentation de la classe <i>Intersection</i> . . . . .	41
8.4.2	Fonction pour tester l'appartenance d'un point . . . . .	41
8.4.3	Fonction d'intersection à un rayon . . . . .	42
8.4.4	Mise en pratique . . . . .	42
8.5	Réalisation de la différence . . . . .	42
8.5.1	Présentation de la classe <i>Substraction</i> . . . . .	42
8.5.2	Fonction pour tester l'appartenance d'un point . . . . .	43
8.5.3	Fonction d'intersection à un rayon . . . . .	43
8.5.4	Mise en pratique . . . . .	43
8.6	Résumé des opérations . . . . .	44
8.7	Travail avec des plans . . . . .	44
8.7.1	Généralités . . . . .	44
8.7.2	Modélisation du plan . . . . .	44
8.7.3	Savoir si un point est de le plan . . . . .	45
8.7.4	Trouver les points d'intersection . . . . .	45
8.8	Travail avec des sphères . . . . .	46
8.8.1	Généralités . . . . .	46
8.8.2	Modélisation d'une sphère . . . . .	46
8.8.3	Savoir si un point est de la sphère . . . . .	46
8.8.4	Trouver les points d'intersection avec la sphère . . . . .	46
8.9	Travail avec des cylindres . . . . .	47
8.9.1	Généralités . . . . .	47
8.9.2	Modélisation d'un cylindre . . . . .	47
8.9.3	Savoir si un point est dans un cylindre . . . . .	47
8.9.4	Trouver les points d'intersection avec le cylindre . . . . .	48

<b>8.10 Travail avec des tores . . . . .</b>	<b>50</b>
8.10.1 Généralités . . . . .	50
8.10.2 Modélisation d'un tore . . . . .	50
8.10.3 Savoir si un point est dans un tore . . . . .	51
8.10.4 Trouver les points d'intersection avec un tore . . . . .	52
<b>8.11 Travail avec des boîtes . . . . .</b>	<b>54</b>
8.11.1 Généralités . . . . .	54
8.11.2 Modélisation d'une boîte . . . . .	54
8.11.3 Savoir si un point est dans une boîte . . . . .	55
8.11.4 Trouver les points d'intersection avec une boîte . . . . .	56
<b>8.12 Mise en pratique . . . . .</b>	<b>57</b>
<b>9 • Textures. . . . .</b>	<b>60</b>
9.1 Organisation du code pour les textures . . . . .	60
9.1.1 Les principales classes . . . . .	60
9.1.2 Modifications génériques . . . . .	60
9.2 Texture carrelée . . . . .	60
9.2.1 Configurations possibles . . . . .	60
9.2.2 Appliquer la texture . . . . .	61
9.2.3 Résultat obtenu . . . . .	61
9.3 Textures utilisant un bruit de Perlin . . . . .	62
9.3.1 Structure du code . . . . .	62
9.3.2 La classe <i>Perlin</i> . . . . .	62
9.3.3 La classe <i>PerlinTexture</i> . . . . .	63
9.3.4 La classe <i>MarbleTexture</i> . . . . .	64
9.3.5 La classe <i>WoodTexture</i> . . . . .	65
9.3.6 Combiner les deux . . . . .	67
<b>10• Les maillages . . . . .</b>	<b>68</b>
10.1 Mise en place des maillages . . . . .	68
10.1.1 Comment définir un maillage ? . . . . .	68
10.1.2 Classes utilisées dans le code . . . . .	68
10.1.3 La classe <i>BoundingBox</i> . . . . .	68
10.2 Le fonctionnement de la classe <i>Geometry</i> . . . . .	69
10.2.1 Les différents attributs . . . . .	69
10.2.2 Chargement du fichier . . . . .	69
10.2.3 Construction de la boîte englobante . . . . .	70
10.2.4 Chercher les points d'intersection avec le maillage . . . . .	70
10.3 Exemple de rendu . . . . .	72
<b>Conclusion . . . . .</b>	<b>74</b>

## Liste des figures

---

1	Intersection d'un sphère avec un rayon . . . . .	9
2	Deux sphères de couleurs différentes . . . . .	10
3	Grandeurs permettant de calculer l'intensité lumineuse . . . . .	10
4	Ajout de la lumière pour notre sphère . . . . .	11
5	Ajout de murs dans la scène . . . . .	12
6	Ajout de couleurs à la scène . . . . .	13
7	Visualisation de la stratégie pour déterminer les ombres portées . . . . .	14
8	Correction de l'imprécision numérique sur les ombres portées . . . . .	15
9	Notations pour les matériaux spéculaire . . . . .	17
10	Différentes versions de sphères spéculaires . . . . .	18
11	Vérification de l'effet du nombre maximal de récurrences dans une scène avec des miroirs . . . . .	19
12	Quelques résultats « jouet » avec les surfaces spéculaires . . . . .	19
13	Situation de transparence et réfraction d'un rayon . . . . .	20
14	Réfraction sur divers matériaux . . . . .	22
15	Utilisation des coefficients de Fresnel pour différents matériaux . . . . .	24
16	Visualisation de l'intérêt de l'utilisation de la correction gamma . . . . .	26
17	Zoom sur une zone de transition de l'image sans antialiasing . . . . .	27
18	Comparaison d'images avec et sans antialiasing . . . . .	28
19	Tests de l'éclairage diffus avec plusieurs valeurs de rayons par pixels . . . . .	35
20	Génération d'un nouveau rayon aléatoire dont la loi est centrée sur la source de lumière . . . . .	37
21	Une scène témoin et les résultats en utilisant une source étendue . . . . .	38
22	Illustration du principe de la CSG . . . . .	39
23	Test d'union entre une sphère et un cylindre . . . . .	41
24	Test d'intersection entre une sphère et un cube . . . . .	42
25	Test de différences entre une sphère et un cube . . . . .	43
26	Objets utilisés pour comparer les différentes actions dans notre CSG . . . . .	44
27	Toutes les opérations possibles avec notre système de CSG . . . . .	44
28	Intersection entre un rayon et un plan . . . . .	45
29	Grandeur permettant de définir un cylindre . . . . .	47
30	Grandeur permettant de définir un torus . . . . .	51
31	Schéma pour illustrer la manière de déterminer si un point est dans un tore . . . . .	52
32	Grandeur permettant de définir une boîte . . . . .	54
33	Union de trois cylindres . . . . .	57
34	Réalisation de l'image test trouvée sur Wikipédia et illustration de l'ordre dans la différence . . . . .	58
35	Différents essais avec l'objet créé en CSG . . . . .	58
36	Textures de carrelage avec différents pas sur un plan parallèle à $(Oxz)$ . . . . .	61
37	Scène avec des murs entièrement couverts par cette texture . . . . .	62
38	Tests de la texture de carrelage sur différents objets . . . . .	62
39	Représentation des cinq premières octaves pour le bruit de Perlin . . . . .	64
40	Fonction à bruiter pour réaliser du marbre . . . . .	64
41	Essais de différents coefficients pour obtenir un effet marbré . . . . .	65
42	Application de la texture de marbre sur différents solides . . . . .	65
43	Essais de différents coefficients pour obtenir un effet bois . . . . .	66
44	Un bon compromis de valeur pour le bois . . . . .	66
45	Application de la texture de marbre sur différents solides . . . . .	66
46	Scène utilisant deux textures définies par bruit de Perlin et de la CSG . . . . .	67
47	Explication de la stratégie pour déterminer s'il y a intersection sur une boîte . . . . .	68
48	Condition sur les coordonnées de $P$ pour que ce dernier appartienne au triangle $ABC$ . . . . .	71
49	Exemple de rendu de maillage . . . . .	73

## Liste des tables

---

1	Indices optiques de quelques matériaux courants . . . . .	22
2	Différents temps de calculs pour une image de taille 350 × 350 avec 5 rebonds maximum . . . . .	34

## Introduction

---

Ce rapport présente les différentes réalisations obtenues dans le cadre de ce cours d'informatique graphique. Pour cela, nous détaillerons pour chaque réalisation quelques éléments théoriques justifiant les opérations réalisées, puis nous présenterons le code associé ainsi que des résultats.

Toutes les fonctionnalités proposées n'ont pas été réalisées, cependant un spectre large a tout de même été abordé. Voici la liste des opérations proposées :

- **Ombres portées** Pour la prise en compte des ombres dans le cas où un objet se situe entre la lumière et le point testé.
- **Matériaux** Nous avons réalisé les quatre principaux types de matériaux demandés, à savoir le diffus, le spéculaire, le transparent ainsi que le mélange spéculaire-transparent avec les coefficients de Fresnel. Nous aurions aimé ajouter les matériaux *glossy*, mais n'avons pas eu le temps de le faire.
- **Antialiasing** Ce dernier a été réalisé avant de passer à l'échantillonage avec des méthodes de Monte-Carlo.
- **Éclairage diffus** Pour ce dernier, nous repréciserons également l'origine de la méthode. Ce dernier a été fait dans la suite de l'antialiasing.
- **Sources étendues** Ces dernières ont été réalisées. Cependant, nous ne les avons pas utilisé par la suite, à cause du temps de calcul accru qu'elles demandaient pour obtenir de « bonnes images ». Ainsi, même si le rendu est plus réaliste, nous avons décidé de ne pas les utiliser souvent afin de pouvoir fournir plus d'exemples en diminuant les temps de calculs.
- **Les maillages** Pour ces derniers, nous ne présenterons que la version basique qui permet de l'afficher. Les améliorations pour lisser la surface ou encore utiliser le BVH ne sont pas présentes ici.
- **CSG** Pour la CSG, nous avons essayé d'être aussi exhaustifs que possibles. En particulier, nous avons réalisé les trois opérations élémentaires (union, intersection et différence) ainsi que modélisé les solides de base (cylindre, boîte, tore, sphère et le plan dans une moindre mesure). Le solide élémentaire manquant est ici le cône.
- **Textures** Pour les textures, nous proposons deux types de textures. D'une part une texture procédurale simple en quadrillage. D'autre part, deux textures s'appuyant sur des bruits de Perlin, du marbre et du bois.

Tous ces éléments seront présentés les uns à la suite des autres. Une précision importante concernant les illustrations est que ces dernières ont en grande partie (à partir des matériaux) été effectuées à partir du code final qui a été dégradé au besoin. En particulier, à partir de la partie sur les matériaux, les murs sont de vrais plans et non des sphères géantes, ce que nous trouvions plus élégants à l'image et un peu plus rapide pour les calculs (on obtient aussi des résultats meilleurs avec les miroirs en jouant un peu comme nous le verrons).

Enfin, ce code est initialement issu d'un travail fait à deux avec Valentin DEMEUSY. En effet, sur une six séances, nous étions à tour de rôle à l'université Lyon 1 ou à Centrale Lyon pour les cours, et avons donc assisté aux cours en décalé. Nous avons ainsi travaillé sur une base commune de code. Cependant, le code qui vous est présenté ici a été totalement réécrit à partir de ce dernier. Ainsi, des variations pourront apparaître entre les deux codes. En particulier, le code initial n'inclut pas les textures et la gestion des matériaux n'était pas strictement équivalente. Le code initial est disponible à l'adresse suivante :

<https://github.com/stity/informatique-graphique-raytracer/>

Le code remanié est lui disponible à l'adresse suivante ainsi que dans l'archive de ce rapport :

[https://github.com/DDouteaux/Raytracer\\_info\\_graphique/](https://github.com/DDouteaux/Raytracer_info_graphique/)

Concernant les figures proposées, la plupart n'ont pas été réalisées en formant  $1024 \times 2014$ , car ma machine n'était pas assez performante pour toutes les calculer à ce format en un temps raisonnable. La plupart des images ont donc été générées dans des formats entre  $150 \times 150$  ou  $300 \times 300$ .

## 1 • Structure du code réalisé

Avant de détailler les différentes étapes de réalisation du code de ce Raytracer, nous allons présenter le modèle utilisé dans les grandes lignes. L'idée est ici de fournir une vue générale du modèle retenu, afin de pouvoir plus facilement se retrouver dans ces différents développements. Cette partie propose également des informations sur la machine à disposition pour les calculs. Afin de mieux représenter les différents éléments modélisés, nous avons organisé le code C++ en différents classes. Si toutes les classes seront largement couvertes dans ce rapport, nous vous présentons ici l'organisation générale de ces dernières et les méthodes les plus utiles.

**Fichier main** Le fichier *main* joue le rôle de grand arbitre. C'est dans ce dernier que son créé tous les objets et que ce situe la boucle pour chercher les valeurs d'intensités par pixels.

**La scène** Cette dernière est représentée par la classe *Scene*. Cette classe propose deux méthodes centrales, une pour chercher les points d'intersection avec tous les éléments de la scène (*intersect*) et une autre pour renvoyer la couleur du rayon qui lui est passé en paramètre.

**Les objets** La scène est composée de plusieurs objets. Si dans les premières moutures du code on ne dénombrait que les sphères, nous avons rajouté (en particulier en créant le CSG) une plus grande diversité dans ces derniers. Pour simplifier le travail, une classe générique *Object* a été créé dont hérite tous les autres types d'objets. Dans la version finale, on retrouve les objets suivants : *Sphere*, *Torus*, *Cylinder*, *Box*, *Plan*, *Geometry* (pour les maillages).

**Les matériaux** Pour plus de lisibilité dans le code, une classe pour simuler les matériaux a été créée. Ainsi, chaque *Object* disposera d'un attribut *Material* pour s'avoir s'il est diffus, transparent, de quelle couleur, avec quel indice optique ou encore de quel texture...

**Les textures** Ces dernières sont incluses dans les matériaux. Pour les représenter, nous avons ici aussi une classe générique *Texture* dont vont hériter tous les types de textures. De plus, pour les textures utilisant un bruit de Perlin, une autre classe générique héritant de *Texture* et nommée *PerlinTexture* a été créée. On retrouve ainsi trois textures dans ce code, *SquaresTexture* (un quadrillage), *MarbleTexture* (du marbre utilisant le bruit de Perlin) et *WoodTexture* (du bois utilisant le bruit de Perlin).

**Le CSG** Ce dernier réutilise toutes les classes d'objets (qui comme nous le verrons ont été adaptées pour ce besoin particulier). De plus, trois types d'opérations ont été définies, *Union*, *Intersection* et *Subtraction*. Dans les faits, ces trois types d'opérations héritent en fait d'*Object*, et implémentent leur action dans la méthode d'intersection *intersect* (qu'elles vont alors appliquer sur un sous-ensemble d'objet avant d'envoyer le résultat à la version de *Scene*).

**Classes utilitaires** Ces dernières avaient pour but de rendre le code plus transparent. On retrouve ainsi une classe pour modéliser les vecteurs en trois dimensions (*Vector*) ; un fichier de méthodes utilitaires (*helpers.h*) ; un solveur pour les équations de degré quatre (*quarticsolver*) et enfin une classe pour les boîtes englobantes des maillages (*BoundingBox*).

## 2 • Établissement de la scène de test

Dans cette première partie, nous allons nous intéresser à la mise en place des fondamentaux pour notre scène de test. L'idée est ici de réaliser une première version de notre scène classique, avec une sphère entourée de murs. Nous n'utiliserons dans cette première partie que des considérations basiques, à savoir de simple intersection sphère-rayon et un calcul simple d'intensité lumineuse. Les différents éléments relatifs aux ombres ou aux matériaux seront abordés plus tard.

### 2.1 | Intersection entre un rayon et une sphère

La première étape de ce travail a été de mettre en place l'intersection entre un rayon et une sphère, dans un cadre « 2D » initiallement.

Pour cela, nous sommes partis de l'équation d'un rayon (1).

$$C + t \cdot \vec{u} \quad \text{avec} \quad \begin{cases} C, \text{ le point de départ du rayon} \\ \vec{u}, \text{ la direction du rayon} \end{cases} \quad (1)$$

Dans le même temps, l'équation d'un sphère est donnée par (2).

$$\|P - O\|^2 = R^2 \quad \text{avec} \quad \begin{cases} O, \text{ le centre de la sphère} \\ R, \text{ le rayon de la sphère} \end{cases} \quad (2)$$

Le point d'intersection entre un rayon et une sphère peut alors se trouver en égalant les deux relations (1) et (2). En passant quelques calculs sous silence, nous obtenons la relation suivante :

$$\begin{aligned} P \in (1) \cap (2) &\Leftrightarrow \|C + t \cdot \vec{u} - O\|^2 = R^2 \\ &\Leftrightarrow t^2 + 2 \langle \vec{V}, \vec{C} - \vec{O} \rangle t + \|C - O\|^2 - R^2 = 0 \end{aligned}$$

Notre objectif est ici de déterminer les  $t$  tels qu'il y ait égalité. Ainsi, il nous faut résoudre cette équation du second degré pour obtenir nos/notre intersection s'il existe (ou non). En notant  $\Delta$  le discriminant de ce polynôme, nous distinguons alors diverses situations :

- ◎  $\Delta < 0$  Aucune solution, le rayon ne coupe pas la sphère.
- ◎  $\Delta = 0$  Une unique solution, le rayon est tangent à la sphère.
- ◎  $\Delta > 0$  On distingue alors diverses solutions :
  - ▷  $t_1, t_2 > 0$  Le rayon coupe deux fois la sphère, et les deux intersections sont face à la caméra.
  - ▷  $t_1 < 0$  et  $t_2 > 0$  Une intersection se trouve derrière la caméra ( $C$ ) et la seconde devant (la seule qui nous intéresse ici).
  - ▷  $t_1, t_2 < 0$  Les deux intersections sont derrière la caméra ( $C$ ).

On remarque en particulier que dans le cas où il y a plusieurs racines, l'idée est alors de récupérer celle la plus proche de la caméra, ie. le  $t$  le plus petit, mais tout en étant positif (pour être face à la caméra). Ce schéma d'intersection est proposé avec une première version de notre méthode `intersect` (classe `Sphere`) :

```

1 bool Sphere::intersect(const Ray& r, Vector& P, double& t, Vector& N) {
2     double a = 1;
3     double b = 2.*dot(r.u,r.C-O);
4     double c = (r.C-O).squaredNorm() - R*R;
5     double delta = b*b - 4*a*c;
6     if (delta>=0) {
7         double t1 = (-b-sqrt(delta))/(2*a);
8         double t2 = (-b+sqrt(delta))/(2*a);
9     }
  
```

```
10     if (t1 > 0) {
11         P = r.C+ t1*r.u;
12         t = t1;
13         N = P-O;
14         N.normalize();
15         return true;
16     } else if (t2 > 0) {
17         P = r.C+ t2*r.u;
18         t = t2;
19         N = P-O;
20         N.normalize();
21         return true;
22     }
23 }
24 return false;
25 }
```

Cette méthode peut alors être utilisée dans la première mouture de notre méthode `getColor`. Cette version initiale reste simple et ne rend pas d'effets en trois dimensions.

```
1 Vector Scene::getColor(const Ray &ray, int recursion){
2     // P sera le point d'intersection s'il existe et N la normale si P existe.
3     Vector P;
4     Vector N;
5     Ray ray(C, u);
6     int objectId;    // id de l'objet intersecté s'il existe
7
8     if (scene.intersect(ray, P, N, objectId)) {
9         return Vector(1,1,1);
10    } else {
11        return Vector(0,0,0);
12    }
13 }
```

Nous obtenons alors directement le résultatat de la Figure 1.

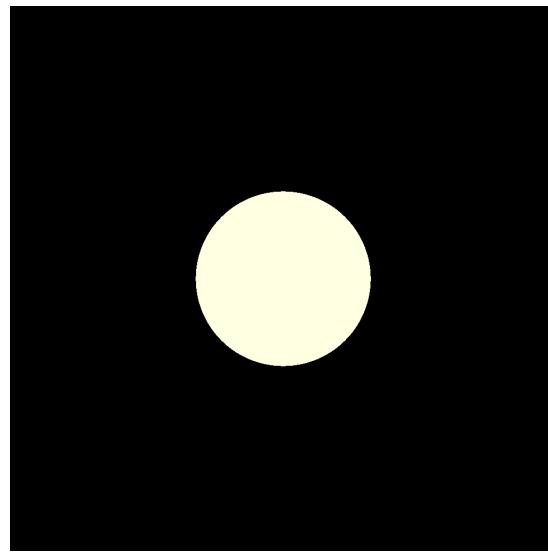
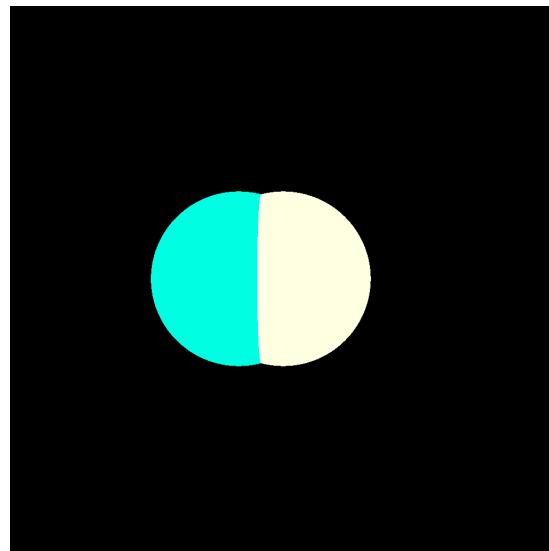


FIGURE 1 • Intersection d'un sphère avec un rayon

Nous pouvons également à la place de renvoyer `Vector(1,1,1)` (blanc) en cas d'intersection, renvoyer la couleur de la sphère qui est intersectée. Il faut pour cela stocker cette dernière dans la classe `Sphere` par exemple. On peut alors regarder la situation dans le cas où plusieurs sphères s'intersectent.

Nous obtenons alors le résultats de la Figure 2.

On remarque la séparation entre les deux sphères qui n'est pas linéaire, dans la mesure où ces dernières étaient à des profondeurs différentes. Pour pouvoir distinguer effectivement quelle sphère est devant l'autre,



**FIGURE 2** • Deux sphères de couleurs différentes

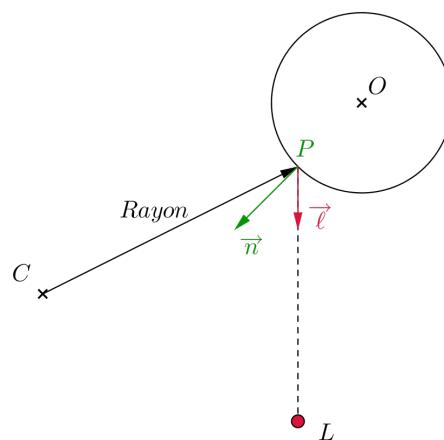
la classe *Scene* dispose d'une méthode d'intersection qui va garder l'objet dont le point d'intersection est le plus proche de la caméra (le fonctionnement est similaire à la méthode *intersect* vue précédemment, le détail est visible dans *scene.cpp/intersect*). Pour plus de détails, se référer à la Section 2.3.

## 2.2 | Ajout de la lumière

Le mécanisme d'intersection étant désormais mis en place, il nous reste à ajouter la lumière afin d'avoir un nouvel effet de dimension. Pour cela, il nous faut prendre en compte la distance entre notre source de lumière et le point d'intersection afin de calculer l'intensité lumineuse en ce point. Pour cela, nous utiliserons la relation (3).

$$\mathcal{I}(i, j) = \max \left( 0, \vec{\ell} \cdot \vec{n} \right) \times \frac{\ell}{d^2} \quad (3)$$

Dans cette relation (3),  $\vec{\ell}$  représente un vecteur unitaire dirigé du point d'intersection  $P$  vers la source de lumière  $L$ ; et où  $\vec{n}$  est le vecteur normal à la sphère en  $P$ . Pour mieux visualiser ces grandeurs, se référer à la Figure 3.



**FIGURE 3** • Grandeurs permettant de calculer l'intensité lumineuse

Nous pouvons alors adapter le code de notre méthode *getColor* dans le cas où il y a intersection pour ajouter cet élément :

```

1 Vector l = L-P;
2 double distLight2 = l.squaredNorm();
3 l.normalize();
4
5 return 1500*std::max(0., dot(N,l))/distLight2;

```

Dans le cas où la sphère dispose d'une couleur, il suffit de multiplier cette valeur de retour par la couleur de la sphère. On obtient alors dans notre scène à une sphère le résultat de la Figure 4.

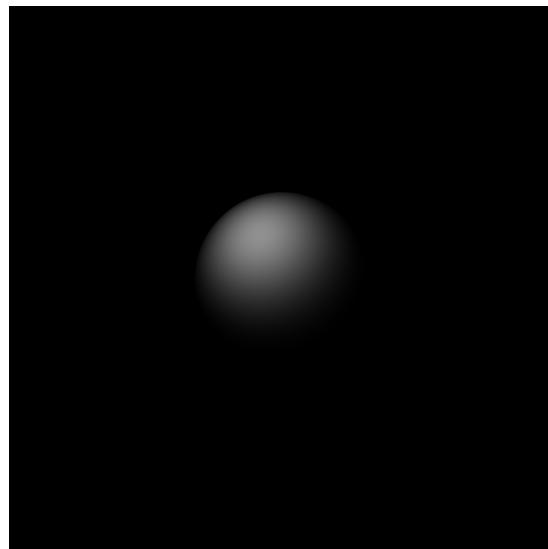


FIGURE 4 • Ajout de la lumière pour notre sphère

On remarque alors que le résultat fourni un effet de relief plus important.

## 2.3 | Cas avec plusieurs sphères

Nous avons déjà rapidement évoqué cette situation dans la Figure 2. De manière plus formelle, l'algorithme est le suivant :

- 1. Pour chaque objet de la scène, déterminer le point d'intersection (s'il existe) le plus proche de la caméra sur le rayon.
- 2. Choisir l'objet dont le point d'intersection est le plus proche.

Ainsi, cet algorithme nous permet de récupérer le point d'intersection qui est bien le premier dans la direction du rayon. Le code associé en découle alors immédiatement :

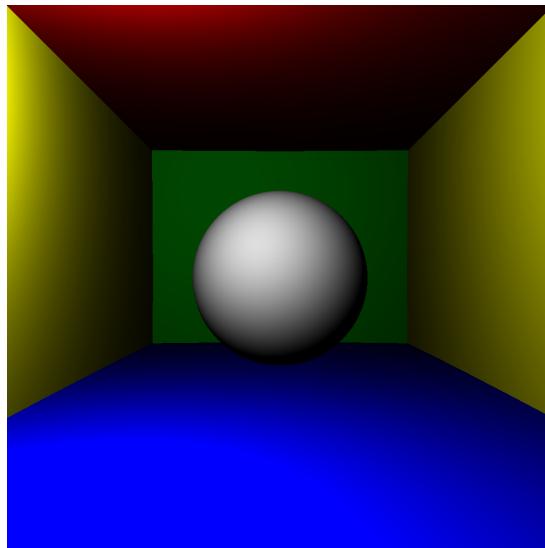
```

1 bool Scene::intersect(const Ray& r, Vector& P, Vector& N, int& id) {
2     double mint = 1E9;
3     bool result = false;
4     for(int i = 0; i<objects.size(); i++) {
5         double t;
6         Vector N1;
7         Vector P1;
8         if (objects[i]->intersect(r, P1, t, N1)) {
9             result = true;
10            if (t<mint) {
11                mint = t;
12                P = P1;
13                N = N1;
14                id = i;
15            }
16        }
17    }

```

```
18     return result;
19 }
```

On peut alors améliorer un peu notre scène initiale en ajoutant par exemple des murs pour encadrer notre sphère. Dans la mesure où le code actuel ne propose que des objets de type *Sphere*, nous allons utiliser des sphères de rayon suffisamment grands pour que leur partie visible soit « quasiment plane ». La scène utilisée varie peu de celle proposée dans le cours, et nous obtenons alors le résultat de la Figure 5.



**FIGURE 5** • Ajout de murs dans la scène

## 2.4 | Ajout de la couleur

---

Pour terminer sur cette partie introductory concernant le positionnement de notre « scène test », nous allons traiter l'ajout de couleur (représenté par un vecteur à trois composantes RGB). En anticipant sur la suite de ce travail, nous proposons directement d'utiliser une classe *Material* qui contiendra toutes les informations relatives au matériau constituant un objet, que ce soit sa couleur, sa texture, son type,...

Pour le moment, cette classe ne contiendra que la couleur, mais sera étoffée par la suite. Nous pouvons ainsi modifier le retour de la fonction *getColor*, en ajoutant la couleur des matériaux :

```
1 return shadow_coeff * (1500*std::max(0., dot(N,l))/distLight2)*objects[objectId
] ->material.color;
```

Nous obtenons alors une nouvelle version de la Figure 5, avec des couleurs en plus à la Figure 6.

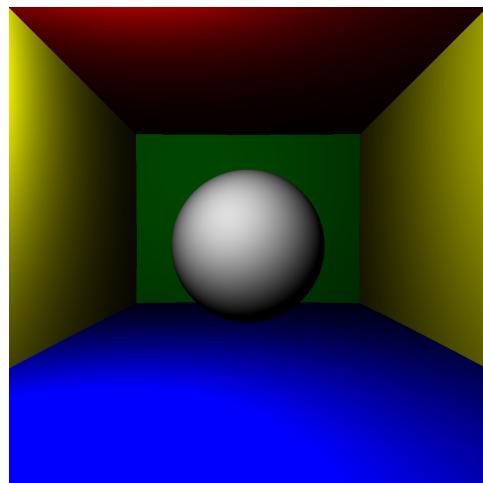


FIGURE 6 • Ajout de couleurs à la scène

### 3 • Ombres portées

Si nous disposons désormais d'une scène test, nous pouvons remarquer que cette dernière manque de réalisme. Le premier élément marquant est l'absence d'ombres (notamment sous la sphère centrale), que nous allons aborder ici.

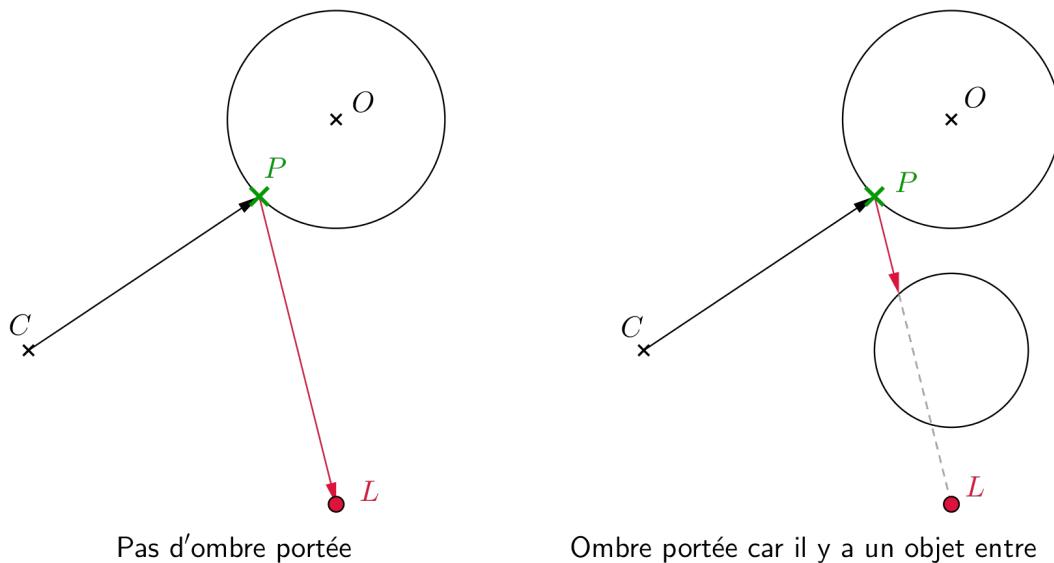
#### 3.1 | Éléments théoriques

Le problème posé avec l'ombre est que l'on ne vérifie pas actuellement s'il existe un obstacle entre une lumière et un point d'intersection. Ainsi, le calcul de l'intensité d'un rayon se préoccupe actuellement uniquement de la distance entre le point et la source de lumière.

Pour prendre en compte les ombres, il nous faut ajouter à cela la vérification qu'il n'y ait pas d'objet entre le point et la source de lumière. Pour cela, nous procédons comme suit :

- 1. On recalcule un rayon partant du point d'intersection  $P$  en direction de la lumière  $L$ .
- 2. On regarde si ce rayon intersecte un élément de la scène :
  - ▷ 2.1. Si le rayon intersecte un élément, le pixel va avoir un élément d'ombre.
  - ▷ 2.2. Si le rayon n'intersecte aucune élément, la formule donnant l'intensité du pixel est inchangée.

Ceci se comprend aisément avec la Figure 7.



**FIGURE 7 •** Visualisation de la stratégie pour déterminer les ombres portées

Nous pouvons alors légèrement adapter le code la méthode `getColor`, en ajoutant après le calcul du vecteur  $\vec{\ell}$  les éléments suivants :

```

1 double shadow_coeff = 1;
2 Vector Pp, Np, direction;
3 int idp;
4 direction = (L-P);
5 direction.normalize();
6 if(intersect(Ray(P, direction), Pp, Np, idp)){
7     if((Pp-P).squaredNorm() < distLight2){
8         shadow_coeff = .6;

```

```
9     }
10 }
11
12 return shadow_coeff * (1500*std::max(0., dot(N,1))/distLight2)*objects[objectId]
    ]->material.color;
```

On remarque l'apparition d'une variable *shadow\_coeff* qui va permettre de présenter ces ombres. S'il n'y a pas d'intersection (cas 2.1), alors ce facteur doit être neutre dans le calcul de l'intensité, sinon il doit atténuer la luminosité du pixel. Il nous suffit donc de multiplier notre valeur de retour par ce coefficient pour représenter ces ombres.

## 3.2 | Résultats obtenus

### 3.2.1 Sans décoller le point d'intersection

Comme ceci était attendu en lisant le cours, le premier essai fourni une figure extrêmement bruitée. L'origine est ici numérique, en effet, pour notre calcul d'ombres portées, nous cherchons les intersections du « rayon  $\vec{PL}$  » avec les objets de la scène. Or, les imprécisions numériques vont entraîner que dans certains cas la solution de l'équation d'intersection entre une sphère et un rayon va fournir un point légèrement en face de  $P$ , et ainsi la sphère va se faire de l'ombre elle-même...

Pour corriger ceci, une solution est de décoller légèrement le point de départ du vecteur  $\vec{PL}$  de la surface de la sphère en prenant par exemple  $P + 0,001\vec{N}$  à la place. On corrige ainsi l'imprécision numérique, le code fixé devient alors :

```
1 if(intersect(Ray(P+0.001*N, direction), Pp, Np, idp)){
```

Avec le correctif proposé, nous obtenons alors le résultat de la Figure 8, où le bruit a disparu.

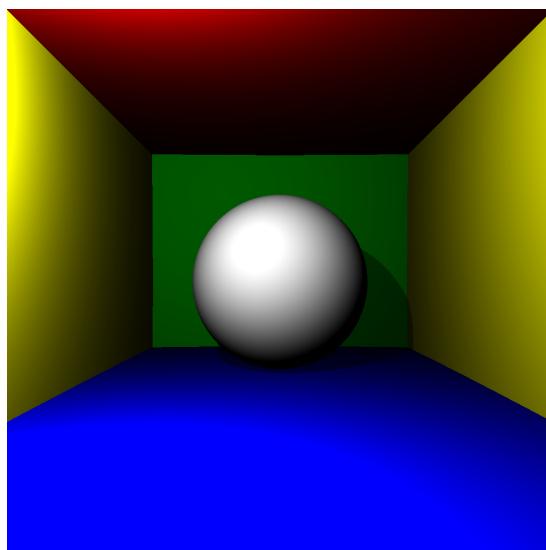


FIGURE 8 • Correction de l'imprécision numérique sur les ombres portées

La valeur de coefficient d'ombre permet de déterminer de combien l'ombre va influer sur la couleur finale du pixel. Tant que nous n'utilisons pas de technique qui moyenne les rayon (comme Monte-Carlo), prendre une valeur trop faible créera des ombres trop marquées. Ainsi, nous prendront pour le moment une valeur de 0,6. Cette valeur évoluera quand nous utiliserons Monte-Carlo, car dans ce cas là, on peut moyenner cette influence, et donc considérer une ombre de 0 si jamais un objet s'intercale.

## 4 • Matériaux

### 4.1 | Matériaux diffus

Nous commençons la revue des matériaux mis en œuvre suite aux travaux initiaux avec les matériaux diffus. Dans les faits, ces derniers correspondent simplement à l'effet déjà mis en place, à savoir des matériaux qui répartissent uniformément la lumière dans toutes les directions (typiquement du platre).

Ainsi, aucun élément de code supplémentaire n'est à ajouter pour leur prise en compte. Cependant, ces derniers sont l'occasion (avant de passer à d'autres matériaux) de remettre en forme le code pour un usage plus général.

Ainsi, nous avons ajouté à ce moment une classe *Material* permettant de représenter le matériau d'un objet, chaque objet ayant alors un attribut *Material*. De même (en anticipant toujours de futurs développements), les objets sont généralisés par une classe *Object* qui regroupe les méthodes et attributs communs (dont l'attribut du *Material*, qui est sa seule utilité dans cette première version). Le patron de cette première version est alors :

```

1 class Object{
2 public:
3     Object(Material material);
4
5     // Déclaration virtuelle pour la fonction d'intersection à réimplémenter
6     // pour chaque objet
7     virtual bool intersect(const Ray&, Vector&, double&, Vector&) {return false;}
8
9     Material material;
};
```

On remarquera qu'en plus du matériau de l'objet, cette classe demande à toute classe en héritant d'implémenter la méthode d'intersection, ceci nous permet de stocker un vecteur d'*Object* dans notre scène et d'appeler de manière transparente la méthode d'intersection. Ainsi, le code de la méthode *intersect* pour la classe *Scene* va appeler cette méthode générique, et aucune modification ne sera à prévoir quand d'autres formes seront implémentées. On remarque également une évolution dans la manière d'appeler le matériau d'un objet, comme en témoigne le code ci-dessous issue de la nouvelle version de *getColor*:

```

1 return shadow_coeff * (1500*std::max(0., dot(N,1))/distLight2)*objects[objectId
]->material.color;
```

Suite à ces modifications, nous avons créé différents matériaux, directement utilisables dans notre code (*whiteDiffuse*, *blueDiffuse*,...). D'autres matériaux créés seront proposés par la suite.

### 4.2 | Matériaux spéculaires

#### 4.2.1 Relation de réflexion

Il s'agit des matériaux ayant le comportement d'un miroir, i.e. les matériaux qui vont refléter la lumière. Pour prendre en compte ces derniers, il nous faut trouver la relation entre le rayon incident à la surface (notons le  $\vec{t}$ ) et le rayon réfléchi (notons le  $\vec{r}$ ). Les autres notations sont usuelles et rappelées à la Figure 9.

De ce graphique, nous pouvons en déduire immédiatement la relation (4).

$$\vec{r} = \vec{t} - 2 \langle \vec{t}, \vec{n} \rangle \quad (4)$$

#### 4.2.2 Code associé

**Réfléchir un rayon** Le code transcrivant cette relation est immédiat, et ajouté dans la classe *Vector*, où l'idée est alors de réfléchir le vecteur représenté (*this*) :

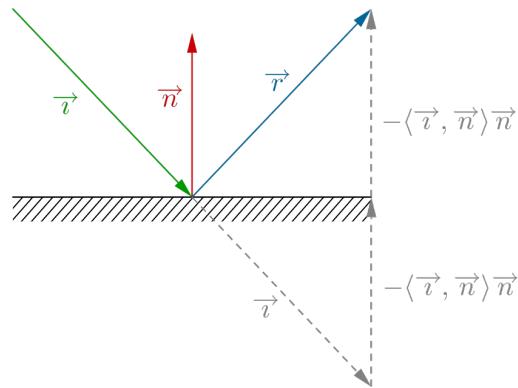


FIGURE 9 • Notations pour les matériaux spéculaire

```

1 Vector Vector::reflect(const Vector& N) const {
2     return *this - 2.*dot(N)*N;
3 }
```

En particulier, on remarque l'usage de la méthode `dot` de la classe `Vector` qui calcule directement le produit scalaire du vecteur représenté avec le vecteur passé en argument.

**Amélioration des matériaux** Désormais, nous avons deux possibilités de matériaux : spéculaire ou diffus. Nous avons donc légèrement amélioré notre classe `Material` pour qu'en plus de la couleur, cette dernière prenne également en compte le type de matériau. Nous avons donc un patron de classe qui est le suivant :

```

1 class Material {
2     public:
3         Material(Vector color=Vector(1.,1.,1.), bool isDiffuse=true, bool
4             isSpecular=false, bool isTransparent=false, double indice = 1);
5         Material(const Material& material);
6
6         Vector color;           // Couleur du matériau
7         bool isDiffuse;        // Caractère diffus (true) du matériau
8         bool isSpecular;       // Caractère spéculaire (true) du matériau
9         bool isTransparent;    // Caractère transparent (true) du matériau
10        double indice;        // Indice de réfraction du matériau
11    };
```

Nous remarquons que la matériau dispose donc d'une couleur, et d'un booléen par caractéristique (diffus, spéculaire, transparent (à venir à la section suivante)), mais aussi d'un indice de réfraction comme nous le verrons pour le cas transparent à la Section 4.3.

Ceci nous permet donc de totalement caractériser (pour le moment !) notre matériau à sa création, et ainsi d'obtenir un fonctionnement plus simple au niveau de `getColor`.

**Les matériaux dans getColor** L'ajout des booléens dans la classe `Material` nous permet de filtrer en fonction des types de matériaux pour savoir quel traitement appliquer au rayon. De plus, le cas des surfaces spéculaires implique de la récursion dans l'image. En effet, nous allons pouvoir avoir des situations où un rayon va être réfléchi, possiblement plusieurs fois s'il y a plusieurs objets spéculaires dans la scène. Ainsi, il est nécessaire de limiter le nombre de récursion pour éviter des cas de calculs « infinis ».

Ainsi, la méthode `getColor` va désormais prendre un argument supplémentaire, qui sera le nombre de récursion initial, d'où sa nouvelle description :

```

1 Vector Scene::getColor(const Ray &ray, int recursion)
```

Ainsi, à chaque fois qu'un rayon sera réfléchi, nous allons appeler de nouveau `getColor` pour le rayon réfléchi afin d'obtenir la couleur du pixel (et simuler donc cette réflexion). Cet appel se fera en diminuant le nombre de récursion, ou n'aura pas lieu si nous avons déjà atteint notre quota (`recursion <= 0`). Si jamais la réflexion ne peut avoir lieu car le nombre maximum de réflexion est atteints, nous renvoyons une couleur par défaut, ici du noir.

```

1 if(objects[objectId]->material.isDiffuse){
2 ...
3 } else if (objects[objectId]->material.isSpecular){
4     if(recursion > 0){
5         Vector refl = ray.u.reflect(N);
6         return getColor(Ray(P+0.001*N, refl), recursion-1)*objects[objectId]->
7             material.color;
8     } else {
9         return Vector(0,0,0);
10 }

```

On remarque également que la couleur du rayon réfléchi est multiplié par la couleur du miroir. Même si ce comportement n'est pas tout à fait réaliste, il signifie que la couleur du miroir aura une influence sur la couleur du pixel (des exemples seront montrés juste après).

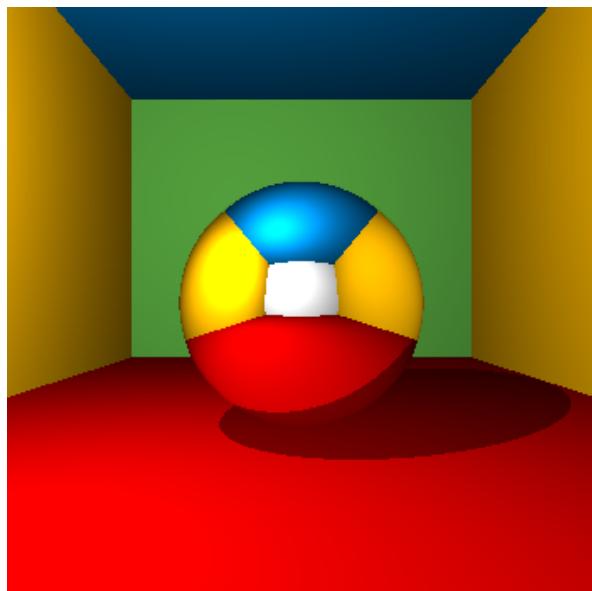
Dans la plupart des situations de ce compte-rendu, nous avons utilisé un nombre de récurrences compris entre 3 et 5.

#### 4.2.3 Exemples graphiques

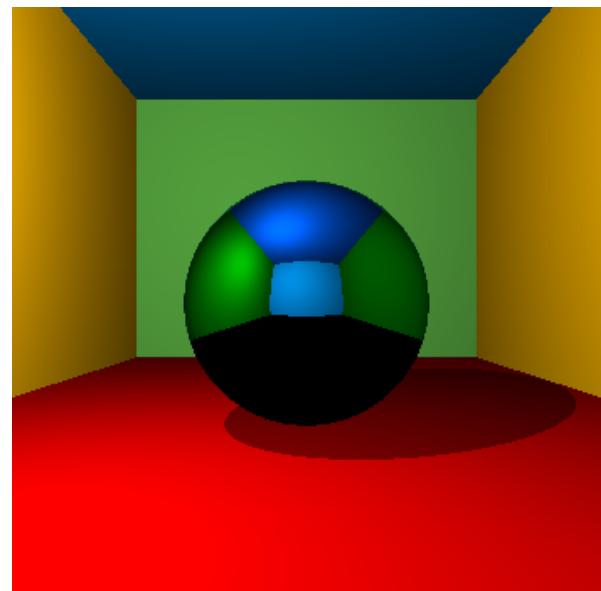
*À partir de cette section, les exemples sont réalisés à partir du code final qui sera ré-adapté pour se retrouver dans les situations proposées (retirer l'anti-aliasing, l'éclairage diffus,...). Cependant, ceci nous permet de réaliser quelques exemples en utilisant d'autres éléments que des sphères. En particulier, pour notre scène témoin les murs sont désormais représenté par de « vrais » plans, pour plus de détails sur ces derniers vous pouvez vous référer à la Section 8.7.*

**Une réflexion basique** Pour commencer, nous proposons une situation où la sphère centrale agit comme un miroir. Plus particulièrement, deux situations ont été testées :

- ◎ *Figure 10a* La sphère est un miroir « commun », ie. sans teint et qui réfléchi juste la lumière.
- ◎ *Figure 10b* La sphère est un miroir, mais propose sa propre couleur (elle est bleue).



(a) Une sphère miroir



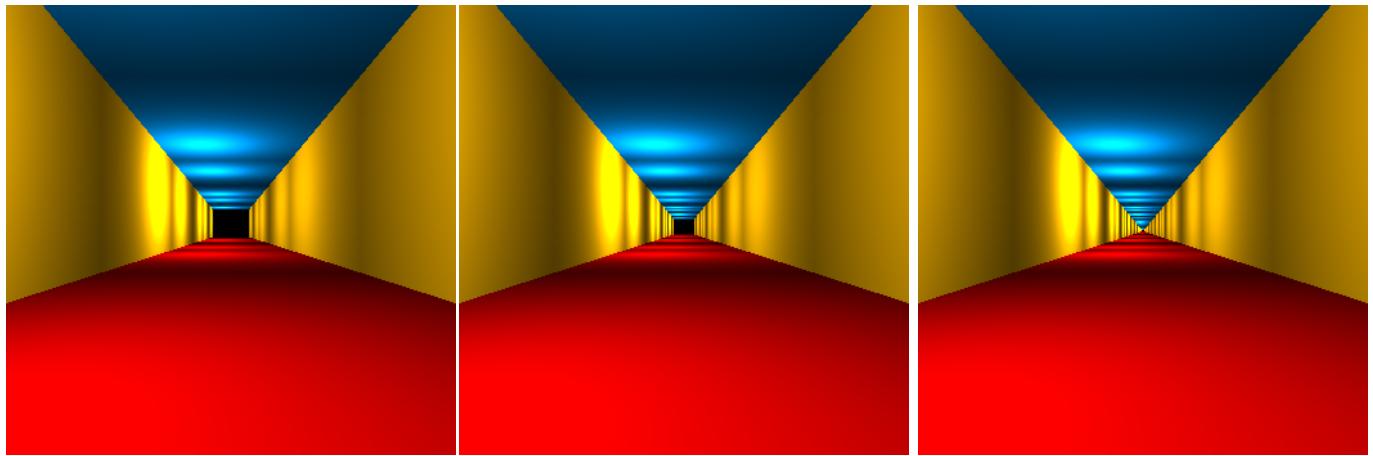
(b) Une sphère miroir colorée

**FIGURE 10** • Différentes versions de sphères spéculaires

On remarquera en particulier que lorsque l'on ajoute une couleur (autre que du blanc) à la sphère spéculaire, cette dernière va réaliser un synthèse additive des couleurs, d'où l'aspect des couleurs reflétée (par exemple le jaune et le bleu qui donnent du vert ou encore le rouge et le bleu du noir).

**Influence du nombre maximal de récursions** Comme nous l'avons vu, nous avons mis une valeur limite au nombre de renvoie récursif du rayon entre les miroirs pour obtenir notre images. Nous avons alors fait un essai en retirant la sphère centrale de notre scène, et en ne conservant que les murs de cette dernières. Ensuite, nous avons changé le mur vert en une surface spéculaire ainsi que le mur blanc (celui derrière la caméra, qui est visible à la Figure 10a).

Dans cette situation, on garantit que des rayons vont avoir faire un nombre d'aller-retour très important entre ces deux miroirs, et nous pouvons donc tester la situation où le rayon « se perd » avant la fin de sa récursion. Différents essais ont alors été réalisés avec des valeurs différentes pour le nombre de récursions autorisées et sont présentés à la Figure 11.

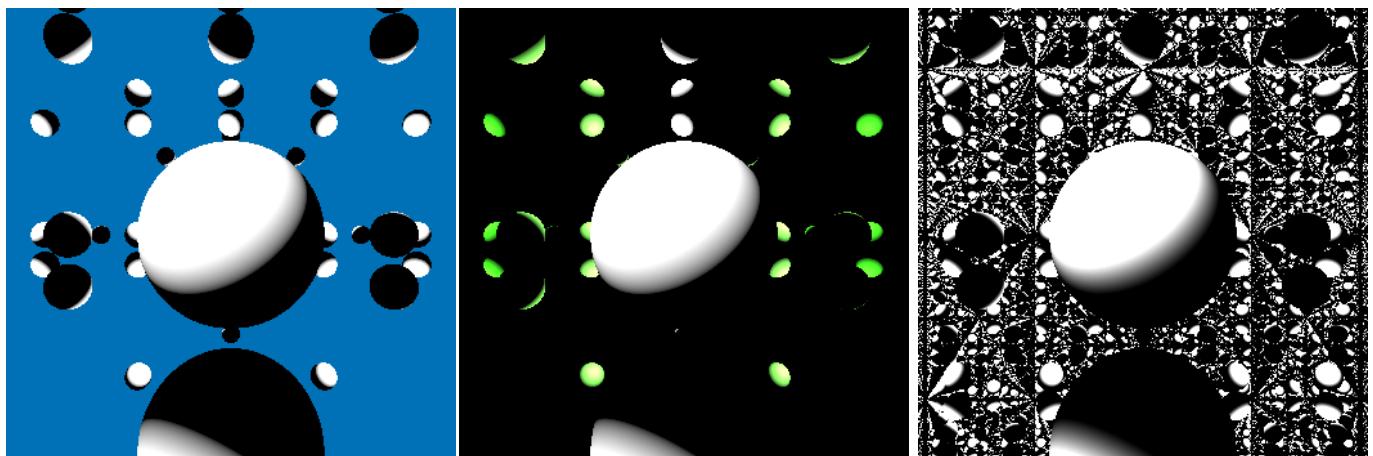


(a) Nombre de récursions fixé à 5      (b) Nombre de récursions fixé à 10      (c) Nombre de récursions fixé à 100

FIGURE 11 • Vérification de l'effet du nombre maximal de récurrences dans une scène avec des miroirs

Le nombre maximum de récursions peut être observé en regardant le nombre de « tâches lumineuses » causées par notre source ponctuelle. On retrouve ainsi en Figure 11a un total de cinq tâches, une par récursion. De même en Figure 11b où l'on retrouve dix tâches. Pour la Figure 11c le décompte n'est pas possible, cependant on remarque que la récursion permet d'aller plus loin dans les reflets...On notera que dans cette situation les rayons perdus fournissent du noir qui n'avait rien à faire enormalement dans ces images.

**Un peu de jeu avec les miroirs** Pour terminer et toujours tester notre récursion, nous avons essayé de réaliser la même image, en remettant une sphère diffuse entre nos murs, murs que nous avons rendus tous spéculaires. Nous avons alors réalisé différentes situations...pour voir le résultat !



(a) Murs spéculaires et rayons perdus      (b) Murs spéculaires blanc et verts et (c) Murs spéculaires, rayons perdus en bleu      rayons perdus en noir et récursion de 100

FIGURE 12 • Quelques résultats « jouet » avec les surfaces spéculaires

Rien de particulier à remarquer ici, si ce n'est que dans les Figure 12a et 12b une part importante de l'image est constituée de rayons qui ont été perdus par un trop grand nombre de réflexions. De plus, la Figure 12b permet de voir l'importance de plusieurs réflexions sur des miroirs colorés (les deux latéraux ici), où on remarque que certaines images refléchies sont plus vertes que d'autres.

Enfin, la Figure 12c n'a pas d'intérêt autre que de voir ce qu'un résultat théorique peut donner. On retrouvera cependant un certain nombre de motifs qui se répètent, le fait que ces motifs restent précis et peu bruités est dû ici au fait que nous ayons utilisé des plans et non de grandes sphères qui auraient un peu déformés ces derniers.

## 4.3 | Matériaux transparents

### 4.3.1 Relation de réflexion

**Établissement de la relation** Les matériaux transparents sont des matériaux qui comme le verre vont laisser passer la lumière en la faisant dévier de sa trajectoire d'incidence. Cette situation peut être représentée par la Figure 13.

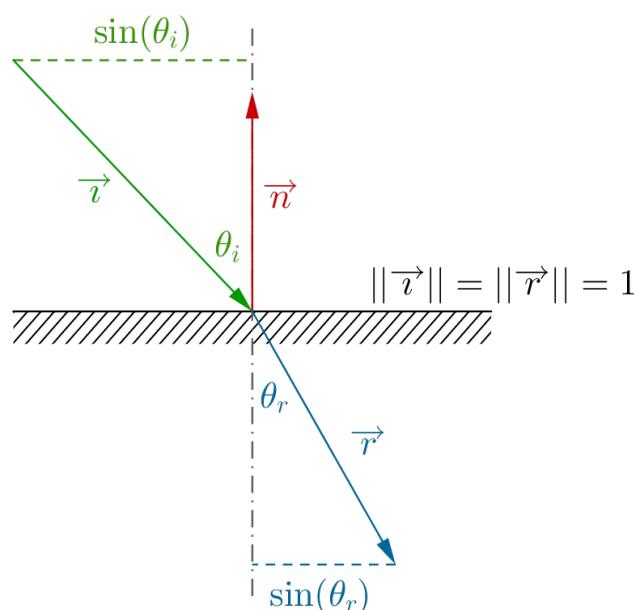


FIGURE 13 • Situation de transparence et réfraction d'un rayon

Comme pour le cas spéculaire, ce qui nous intéresse est d'obtenir l'expression de  $\vec{r}'$  en fonction des autres paramètres.

Pour cela, nous aurons besoin de la relation liant les angles d'incidence et réfracté (relation de Snell-Descartes) qui est fournie par la relation (5).

$$n_i \sin(\theta_i) = n_r \sin(\theta_r) \quad (5)$$

Dans nos situations, la plupart du temps (et tout le temps dans nos essais), nous considérerons que le rayon provient de l'air, ainsi le milieu 1 est simplement l'air. De plus, nous considérerons que les rayons incidents et réfractés sont unitaires. Nous pouvons alors exprimer les composantes tangentialles et normales du rayon réfracté :

$$\begin{aligned} \vec{r}_T &= \frac{n_1}{n_2} (\vec{i} - \langle \vec{i}, \vec{n} \rangle \vec{n}) \vec{t} \\ \vec{r}_N &= -\sqrt{1 - \frac{n_1^2}{n_2^2} (1 - \langle \vec{i}, \vec{n} \rangle^2)} \vec{n} \end{aligned}$$

Nous pouvons alors en retirer l'expression du rayon réfracté qui est un peu plus complexe que dans le cas de la réflexion (6).

$$\vec{r} = \frac{n_1}{n_2} \vec{i} - \left[ \frac{n_1}{n_2} \langle \vec{i}, \vec{n} \rangle + \sqrt{1 - \left( \frac{n_1}{n_2} \right)^2 \left( 1 - \langle \vec{i}, \vec{n} \rangle^2 \right)} \right] \cdot \vec{n} \quad (6)$$

On remarque en particulier que la racine carrée peut être négative et que dans ce cas il y a uniquement réflexion mais pas de réfraction ie. le rayon n'entre pas dans la sphère.

**Remarque sur l'utilisation de la relation (6)** À la différence de la relation de (4), cette nouvelle relation dépend du sens de la normale, en effet, le facteur  $\langle \vec{i}, \vec{n} \rangle$  y apparaît non lié directement à  $\vec{n}$ , ainsi une variation du signe de  $\vec{n}$  ne sera pas compensée par celle du produit scalaire, ce qui peut influer sur le fait que le rayon rentre ou non dans la sphère.

#### 4.3.2 Code associé

En reprenant notre structure de code précédente, les modifications à réaliser sont mineures :

- ◎ **Ajout d'un type de matériau** Ce type de matériau a été appelé *transparent* et le booléen associé ajouté dans *Material* (il était déjà visible dans le code de la Section 4.2). De plus, on stocke l'indice du matériau directement dans le cette classe afin de pouvoir tester différents verres et différents effets.
- ◎ **Modification de getColor** Pour cette fonction, une nouvelle modalité est à ajouter en plus des cas diffus et spéculaire si le matériau est transparent. Ce code découle directement de la relation (6).

```

1 if (intersect(ray, P, objectId)) {
2     if (objects[objectId]->material.isDiffuse){ ... }
3     else if (objects[objectId]->material.isDiffuse){ ... }
4     else if (objects[objectId]->material.isTransparent){
5         if(recursion > 0){
6             bool is_refracted;
7             double n1 = 1;
8             double n2 = objects[objectId]->material.indice;
9             Vector normale = N;
10
11             // Si le rayon rentre dans la sphère, il faut inverser la
12             // normale
13             if(dot(normale, ray.u)>0){
14                 std::swap(n1, n2);
15                 normale = -normale;
16             }
17
18             // On réfracte le rayon selon les normales
19             Vector refr = ray.u.refract(normale, n1, n2, is_refracted);
20             if(is_refracted){
21                 return getColor(Ray(P+0.001*refr, refr),
22                             recursion-1);
23             }
24         } else {
25             return Vector(0,0,0);
26         }
27     } else {
28         return Vector(0,0,0);
29     }
30 }
```

On retrouve les différents éléments vus, à savoir dans l'ordre :

- ◎ 1. La matériau externe est l'air, donc l'indice  $n_1$  vaut 1. L'indice du matériau est lui directement stocké dans l'attribut *material* de type *Material*.

- 2. Puisque le rayon va rentrer dans la sphère, il faut échanger les milieux et inverser la normale pour pouvoir générer correctement le rayon réfracté.
- 3. On réfracte le rayon et on renvoie le résultat obtenu.

### 4.3.3 Résultats obtenus

La Table 1 recueille les valeurs de différents indices optiques associés à des matériaux

Matériau	Diamant	Eau	Silicium	Verre	Cornée
Indice	2,4175	1,33	3,4777	$1,4 \leq . \leq 1,8$	1,37 - 1,4

TABLE 1 • Indices optiques de quelques matériaux courants

Nous avons alors pu réaliser des tests pour les valeurs remarquables de la Table 1, à savoir le diamant (Figure 14d), le silicium (Figure 14e), le verre (en prenant 1,4 et 1,8 aux Figures 14b et 14c) ou encore l'eau (Figure 14a).

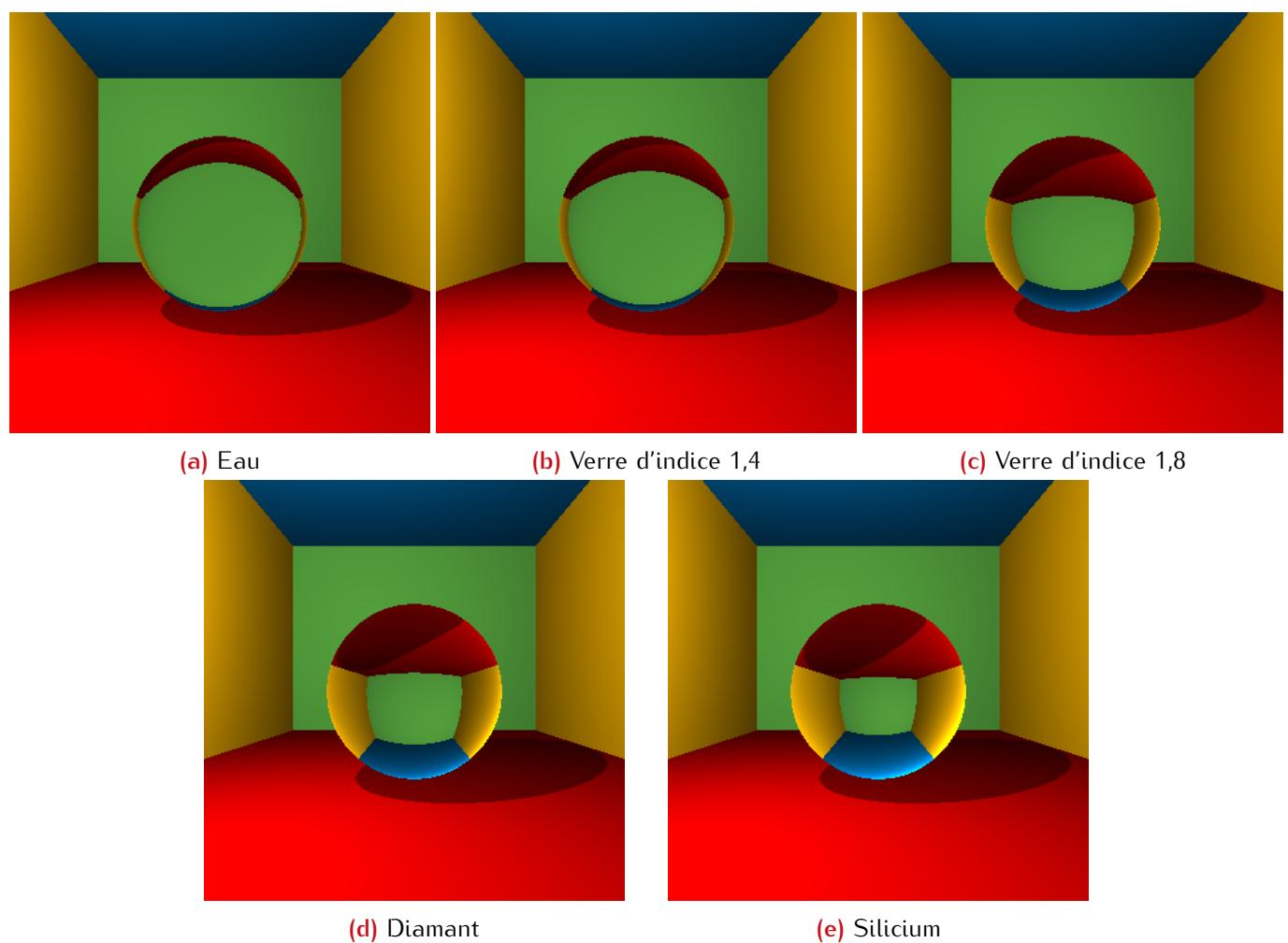


FIGURE 14 • Réfraction sur divers matériaux

On remarque le retournement effectué par la réfraction (par exemple entre le sol et le plafond). De même, on remarque que plus l'indice est élevé, moins le matériau paraît transparent au sens où les parties réfractées prennent plus d'espace en son sein.

## 4.4 Matériaux spéculaires et transparents

### 4.4.1 Motivations et relations théoriques

En réalité, un matériau transparent ne l'est jamais totalement. En effet, il va toujours partiellement refléter les rayons et ainsi séparer les rayons en une composante réfractée et une composante réfléchie.

Dans le cours, il était précisé qu'un calcul exact est très coûteux, cependant il est possible d'utiliser une approximation utilisant un coefficient  $k_0$  défini par (7).

$$k_0 = \frac{(n_1 - n_2)^2}{(n_1 + n_2)^2} \quad (7)$$

À partir de ce coefficient, il est possible de définir un coefficient de transmission en réfraction et en réflexion qui sont définis par (8) et (9).

$$T = k_0 + (1 - k_0) (1 - \langle \vec{n}, \vec{t} \rangle)^5 \quad (8)$$

$$R = 1 - T \quad (9)$$

### 4.4.2 Code associé

Pour prendre en compte les coefficients de Fresnel si ces derniers sont demandés, il est possible de faire comme pour les matériaux transparents en utilisant un nouveau type de matériau que nous appellerons *fresnel*. Concernant le coefficient et son calcul, ce dernier dépend de l'indice des deux milieux, indice qui est déjà pris en compte dans notre classe *Material*.

Ainsi, la modification majeure est encore une fois celle de la fonction *getColor* où un nouveau cas doit être ajouté. Ce dernier a été ajouté avant les cas transparent et spéculaire afin que la volonté d'utiliser les coefficients de Fresnel prime si jamais cette dernière est exprimée pour un matériau.

```

1 // Deuxième cas, l'objet est transparent et spéculaire (coeff de Fresnel)
2 if(objectMaterial.isFresnel){
3     if(recursion > 0){
4         Vector refractionComposant;
5         bool is_refracted;
6         double n1 = 1;
7         double n2 = objectMaterial.indice;
8         Vector normale = N;
9
10        // Si le rayon rentre dans la sphère, il faut inverser la normale
11        if(dot(normale, ray.u)>0){
12            std::swap(n1, n2);
13            normale = -normale;
14        }
15
16        // On réfracte le rayon selon les normales
17        Vector refr = ray.u.refract(normale, n1, n2, is_refracted);
18        if(is_refracted){
19            refractionComposant = getColor(Ray(P+0.001*refr, refr), recursion-1,
20                                            recursionMax);
21        }
22        Vector refl = ray.u.reflect(N);
23
24        // Calcul des coefficients de Fresnel
25        double k0 = pow(n1-n2,2)/pow(n1+n2,2);
26        double T = k0 + (1-k0)*(1-dot(normale,-ray.u));
27        double R = 1-T;

```

```

28     return R*refractionComposant + T*getColor(Ray(P+0.001*N, refl), recursion-1,
29         recursionMax)*objectMaterial.computeColor(P, N);
30 } else {
31     return this->envColor;
32 }

```

On remarque en particulier les points suivants :

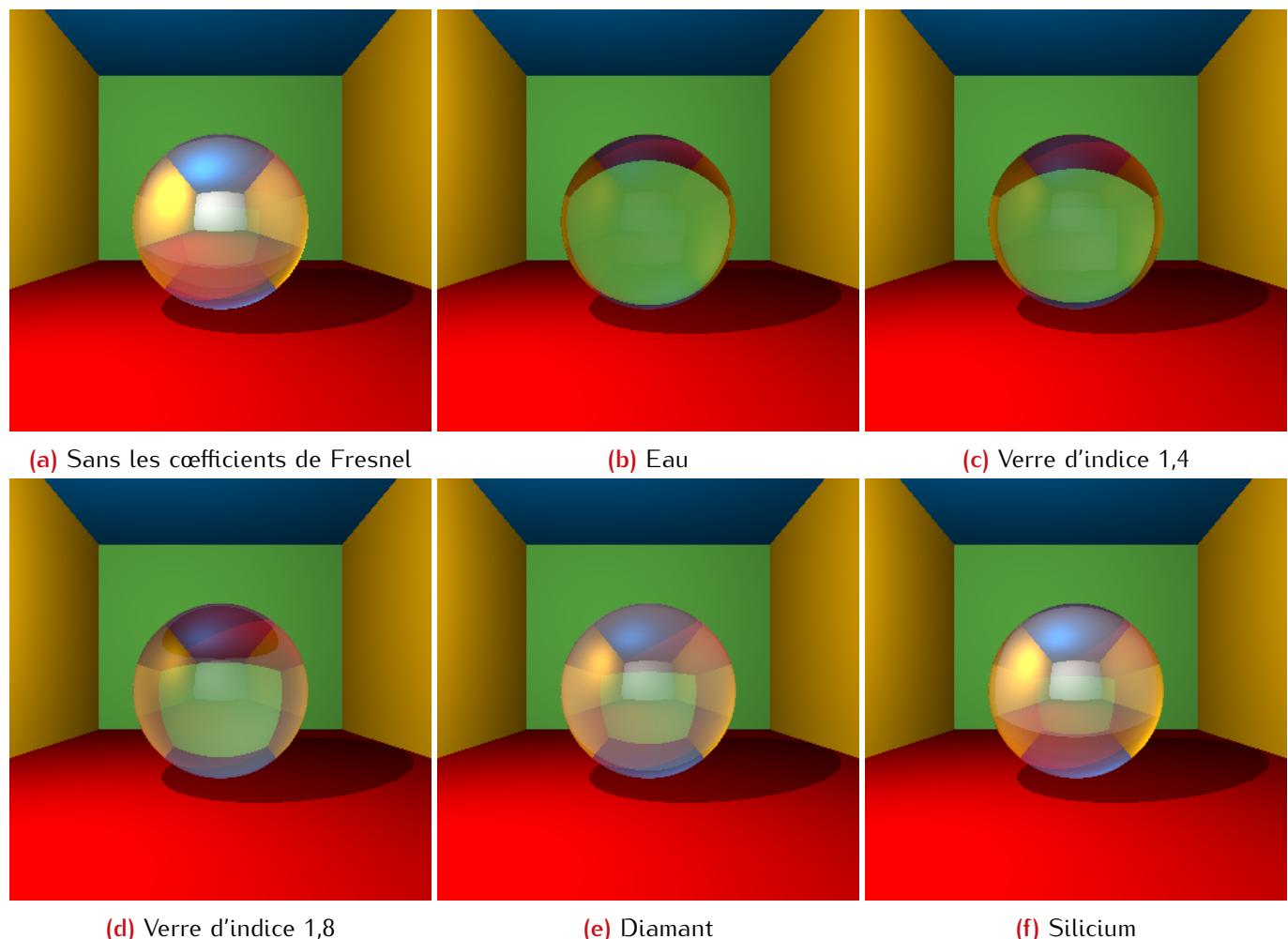
- ◎ 1. On retrouve le code du cas de la transparence à savoir une préparation du rayon pour rentrer dans la sphère.
- ◎ 2. Le rayon est réfracté selon la normale puis réfléchi selon cette dernière.
- ◎ 3. On calcul les coefficients de Fresnel avec les relations (8) et (9).
- ◎ 4. On retourne une valeur qui mêle les deux retours réfléchis et réfractés selon les proportions  $R$  et  $T$  :

$$R \cdot \text{Couleur}_{\text{réfractée}} + T \cdot \text{Couleur}_{\text{réfléchie}}$$

De plus, ce code implique de lancer deux rayons par point dans le cas où l'on touche une surface de type *fresnel*, ce qui a un impact important sur le temps de calcul de la scène en utilisant ces matériaux.

#### 4.4.3 Résultats obtenus

Nous avons utilisé alors le code suivant sur des situations similaires à celles de la Table 1. Nous avons alors obtenu les résultats de la Figure 15, en ajoutant de plus le cas où les coefficients de Fresnel ne sont pas utilisés, ie. le cas où l'on moyenne juste les composantes en réflexion et en réfraction.



**FIGURE 15** • Utilisation des coefficients de Fresnel pour différents matériaux

On remarque que plus le matériau à un indice optique élevé, plus la partie réflexion prend le dessus. Ceci est réaliste, dans la mesure où les matériaux à fort indice optique vont tendre à se rapprocher d'un matériau spéculaire. On remarque que le rendu pour le verre reste conforme à notre « perception » du verre, pour les matériaux à plus fort indice le rendu peut être soumis à caution (on ne connaît pas le domaine de validité de l'approximation utilisée).

*Ceci conclura cette partie sur les matériaux, cette dernière sera étendue un peu plus tard dans le rapport avec l'ajout de textures procédurales et utilisant un bruit de Perlin.*

## 5 • Diverses améliorations

### 5.1 Correction gamma

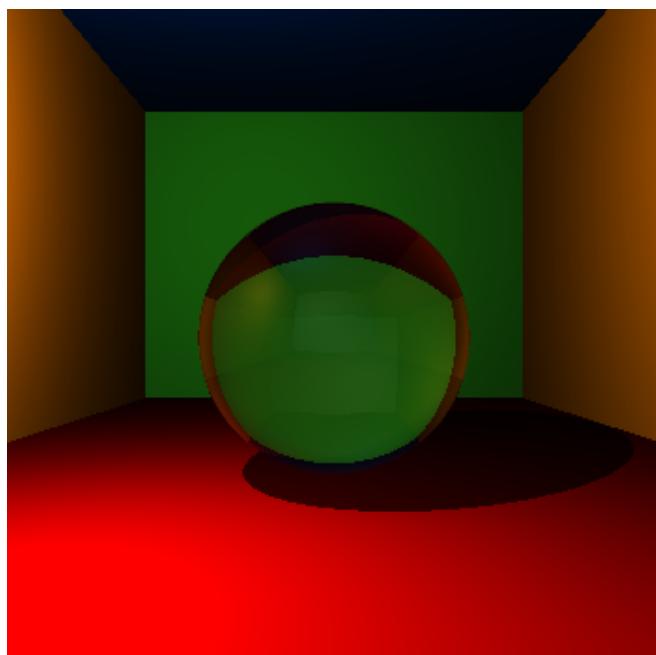
Comme expliqué, les écrans modernes disposent d'un rendu des couleurs qui est non linéaire. Il est ainsi nécessaire d'utiliser une fonction pour rectifier cet aspect, la fonction étant donnée par (10).

$$x \mapsto x^{\frac{1}{2.2}} \quad (10)$$

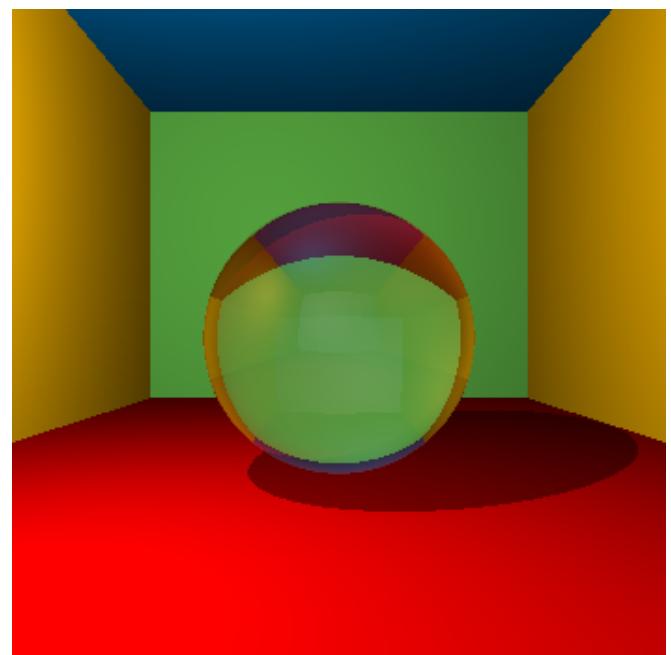
La correction est à exécuter juste avant le rendu, c'est-à-dire avant de stocker le résultat final dans le tableau de pixels représentant l'images :

```
1 img [(i*W+j)*3]=std::min(255., 255*pow(finalIntensity[0], 1./2.2));
2 img [(i*W+j)*3+1]=std::min(255., 255*pow(finalIntensity[1], 1./2.2));
3 img [(i*W+j)*3+2]=std::min(255., 255*pow(finalIntensity[2], 1./2.2));
```

Pour visualiser la différence, nous vous proposons deux images en Figure 16 l'une d'entre elle étant générée sans la correction gamma et l'autre avec.



(a) Sans correction gamma



(b) Avec correction gamma

**FIGURE 16** • Visualisation de l'intérêt de l'utilisation de la correction gamma

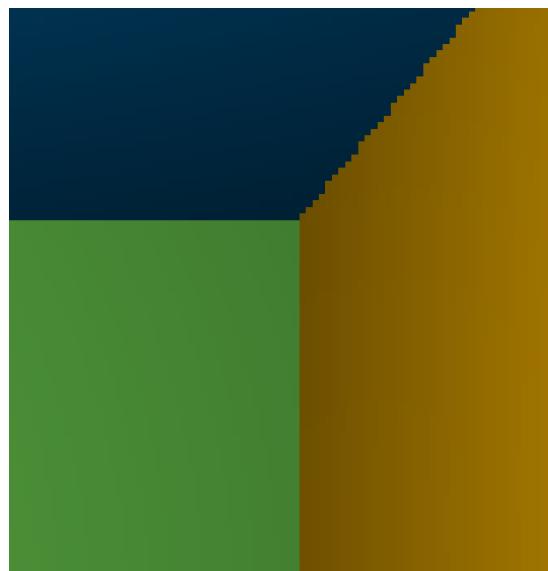
On remarque que l'image sans gamma est plus sombre et plus terne que ne l'est l'autre image. Dans ce rapport, les seules images sans correction gamma étaient celles de la première partie sur la mise en place de la scène test, les autres images (ie. à partir de la section sur les matériaux) utilisaient déjà cette correction gamma.

### 5.2 Ajout de l'antialiasing

#### 5.2.1 Problème posé

Si l'on regarde plus en détails nos précédentes images, ces dernières sont (très) pixelisées, avec en particulier des transitions entre les différents éléments qui sont assez abruptes, comme en témoigne la Figure 17.

On remarque que le résultat est soit tout vert, soit tout jaune et soit tout bleu, mais que les transitions sont très pixelisées.



**FIGURE 17** • Zoom sur une zone de transition de l'image sans antialiasing

### 5.2.2 Solution utilisée

Une solution à ce problème est l'utilisation de techniques d'antialiasing. Plusieurs solutions existent pour cela, celle retenue ici est de réaliser un échantillonage aléatoire de rayon pour chaque pixel. L'idée est ainsi de ne pas tirer un seul rayon du centre du pixel comme nous le faisons actuellement, mais plutôt de tirer  $n$  pixels depuis ce rayon, à partir de positions choisies aléatoirement par rapport au centre du pixel.

Pour cela, on désire cependant que les rayons tirés près du centre du pixel soient majoritaires (pour conserver un coérence physique dans leur signification), ainsi nous allons utiliser un choix de point de départ qui sera réalisé par une gaussienne centré au centre du pixel.

Comme suggéré dans les supports, nous avons alors utilisé la méthode de Box-Muller pour générer nos rayons. En notant *rayPerPixel* le nombre de rayons tirés par pixels, nous avons alors le code suivant :

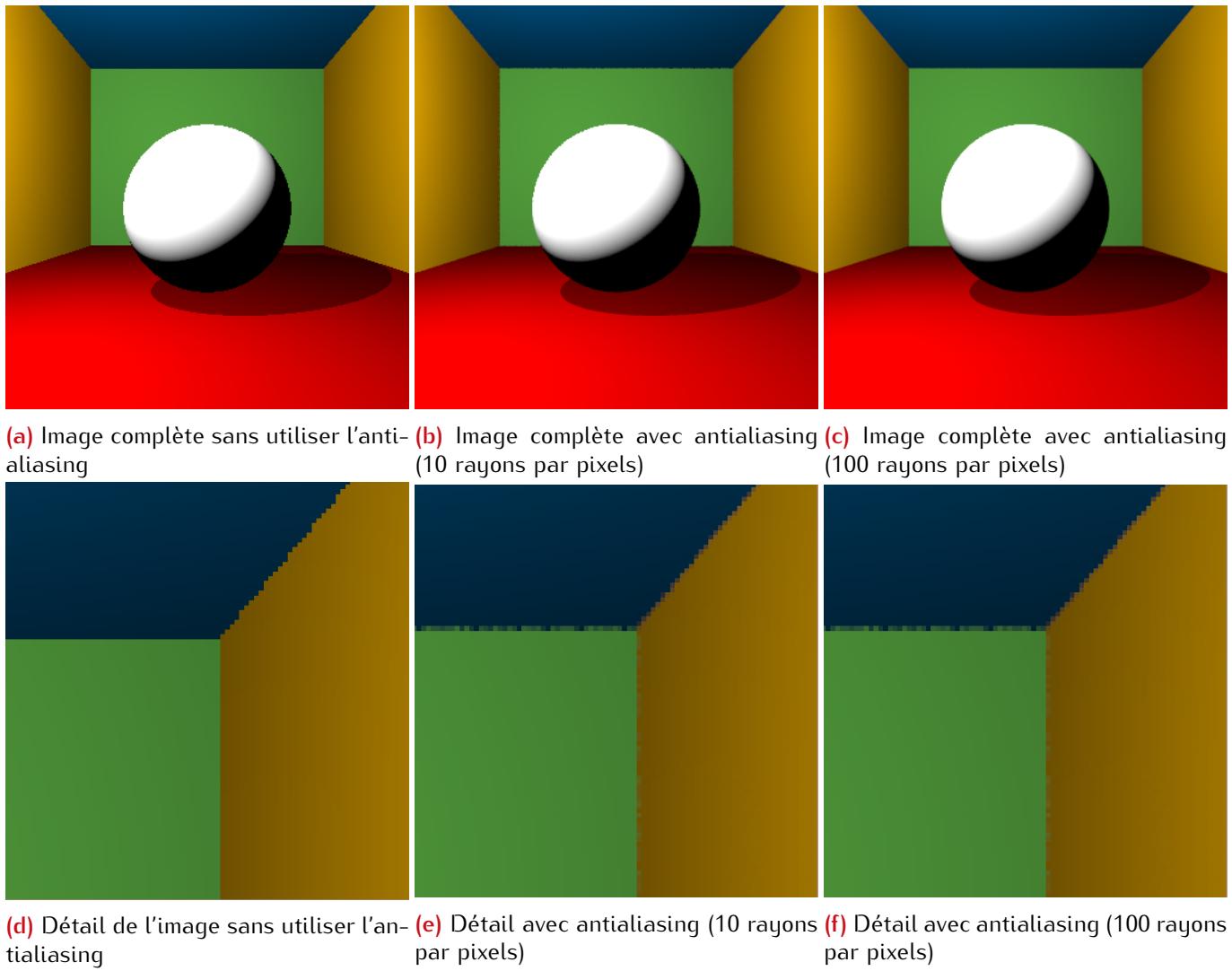
```

1 #include <random>
2
3 default_random_engine engine;
4 uniform_real_distribution<double> distrib(0,1);
5
6 Vector sumIntensities
7
8 for(int k=0; k<rayPerPixel; k++){
9     // La direction du rayon qui est lancé, on la suppose normalisée. r1 et r2
10    // permettent de
11    // légèrement perturber le point de départ du rayon afin d'obtenir des
12    // transitions moins abruptes.
13    double r1 = distrib(engine)-.5;
14    double r2 = distrib(engine)-.5;
15    Vector u (j+r1-W/2.+1/2., H-i+r2-H/2.+1/2., -W/(2.*tan(fov/2.)));
16    u.normalize();
17
18    sumIntensities = sumIntensities + scene.getColor(Ray(C, u), 100, 100);
19}
20
21 Vector finalIntensity = sumIntensities*(1./rayPerPixel);
22 img[(i*W+j)*3]=std::min(255., 255*pow(finalIntensity[0], 1./2.2));
23 img[(i*W+j)*3+1]=std::min(255., 255*pow(finalIntensity[1], 1./2.2));
24 img[(i*W+j)*3+2]=std::min(255., 255*pow(finalIntensity[2], 1./2.2));

```

Cette action nécessitant de générer plusieurs rayons par pixels, elle va cependant fortement ralentir l'exécution du code par rapport aux étapes précédentes. Le résultats est quant à lui des plus satisfaisant.

Nous comparons ainsi en Figure 18 les cas d'image avec un rayon par pixel (pas d'anti-aliasing), 10 rayons par pixels et 100 rayons par pixels.



**FIGURE 18 •** Comparaison d'images avec et sans antialiasing

On remarque bien que dans le cas sans antialiasing, les détails sont très coupés au niveau des couleurs, alors que ces derniers s'affinent en augmentant le nombre de rayons par pixels (jusqu'à obtenir un résultat très lissé avec 100 rayons par pixel). Concernant l'image global, le résultat est également présent, bien que moins visuel sur ces versions réduites pour le rapport.

## 5.3 | Diverses autres améliorations

En plus des grandes étapes qui suivent et des petites améliorations précédemment proposées, nous vous proposons ici une liste de petites améliorations réalisées à la volée afin de se simplifier la vie dans le code.

### 5.3.1 Couleur par défaut pour une scène

Comme nous l'avons évoqué, il peut arriver dans une scène qu'un rayon soit « perdu » car il n'a rencontré que des surfaces transparentes ou spéculaires pendant le nombre de récursions qui lui étaient proposées. Dans ce cas, une valeur par défaut est renvoyée pour le rayon, valeur qui était rentrée manuellement auparavant.

Ainsi, il est proposé dans notre code que cette valeur par défaut en cas de perte du rayon soit directement un attribut de la classe *Scene* afin de simplifier ses modifications et appels. Dans cette optique, tous les retours de la forme `return Vector(0,0,0)` dans `getColor` sont remplacés par :

```
1 return this->envColor;
```

### 5.3.2 Gestion des sphères lumineuses

Nous verrons par la suite que l'utilisation de sources lumineuses étendues nécessite d'utiliser un code spécial. Afin de pouvoir toujours continuer à utiliser la version avec source ponctuelle (qui même si elle est moins réaliste reste plus rapide à calculer à résultat comparable), nous avons donc ajouté un mécanisme dans la classe *Scene* pour réaliser la bascule.

Ainsi, une méthode nommée `addLuxSphere` permet d'ajouter la sphère lumineuse à la scène. Dans ce cas, un attribut de la classe scène nommé `luxSphereIndex` prendra comme valeur l'index de la sphère lumineuse dans le tableau d'objets. En particulier, ceci nous permet de simplifier deux points :

- 1. L'utilisateur peut encore utiliser l'ancien code, les deux version cohabitent donc sans soucis.
- 2. L'utilisateur peut ajouter la sphère à n'importe quel moment dans la construction de la scène, et n'est pas obligé de l'ajouter en première. Ceci fait un petit plus en flexibilité.

La méthode ayant ceci en charge est assez simple :

```
1 void Scene::addLuxSphere(Object * luxSphere){  
2     this->objects.push_back(luxSphere);  
3     this->luxSphereIndex = this->objects.size() - 1;  
4 }
```

## 6 • Éclairage diffus

### 6.1 Aspects théoriques

#### 6.1.1 Motivations

L'objectif initial est ici de prendre en compte l'éclairage indirect. Ce dernier est un éclairage induit par des objets (par exemple diffus), qui vont influer sur les couleurs de objets proches.

Par exemple, si on place une balle blanche sur une table verte, cette dernière pourra à proximité de la table avoir l'impression d'être légèrement verte.

Ceci permet donc d'augmenter légèrement le réalisme de notre scène.

#### 6.1.2 Généralités

L'idée est ici de réfléchir comme pour les surfaces diffuses ou transparentes. Pour ces dernières, nous renvoyons systématiquement un rayon dans la direction de réfraction ou de réflexion calculée.

Pour les surfaces diffuses, nous avions vu que ces dernières renvoient la lumière de manière uniforme dans toutes les directions possibles. Ainsi, l'idée va être de renvoyer le rayon dans une direction aléatoire, et de moyenne sur plusieurs directions.

Cette justification provient de l'utilisation des BRDF (présentées en Section 6.1.3), de l'équation du rendu (présentée en Section 6.1.4) et de méthodes de Monte-Carlo pour approximer des intégrales (présentées en Section 6.1.5).

#### 6.1.3 Les BRDFs

**Généralités** Il s'agit de fonction qui vont caractériser les matériaux en précisant la probabilité qu'un photon arrivant à leur surface avec un incidence  $\vec{i}$  rebondisse dans la direction  $\vec{o}$ . On note simplement cette probabilité par une fonction  $f$  sous la forme :

$$f(\vec{i}, \vec{o})$$

Ces BRDFs vérifient deux principes, le principe de réciprocité d'Helmholtz (11) et la conservation de l'énergie lumineuse (12).

$$f(\vec{i}, \vec{o}) = f(\vec{o}, \vec{i}) \quad (11)$$

Ce principe de réciprocité implique qu'il y a même probabilité pour qu'un rayon arrive en  $\vec{i}$  et soit redirigé vers  $\vec{o}$  que l'inverse. Il y a donc une certaine symétrie dans les BRDFs.

$$\int f(\vec{i}, \vec{o}) \cos(\theta_i) d\vec{i} \leq 1 \quad (12)$$

Cette relation signifie simplement que le matériau ne va pas créer de lumière supplémentaire, mais juste renvoyer au plus toute l'intensité lumineuse qu'il a reçue.

**BRDF spéculaire** Comme nous l'avons expliqué, dans le cas spéculaire, tous les photons sont redirigés vers la direction miroir. Ainsi, en utilisant la relation (4), nous en déduisons directement l'expression de la BRDF qui renvoie une incidence  $\vec{i}$  dans la direction miroir (13).

$$f_{spéculaire}(\vec{i}, \vec{o}) = \frac{1}{\cos(\theta_i)} \delta(2 \langle \vec{i}, \vec{n} \rangle \vec{n} - \vec{i} - \vec{o}) \quad (13)$$

Dans cette relation,  $\delta$  est un Dirac, ainsi elle signifie juste que la seule direction possible est la direction miroir.

Le cosinus vient de la nécessité pour cette BRDF de remplir la relation (12).

**BRDF diffuse** Dans le cas diffus, nous avons précisé que le rebond était uniforme entre les directions, ainsi aucune d'entre elle n'est favorisé. Ainsi, cette BRDF est de la forme  $f(\vec{\iota}, \vec{o}) = \alpha$  et  $\alpha$  vérifie la relation (12), donc :

$$\int \alpha \cos(\theta_i) d\vec{\iota} = \alpha \pi \leq 1$$

On en déduit donc que cette BRDF est donnée par :

$$f_{\text{diffus}}(\vec{\iota}, \vec{o}) = \frac{1}{\pi}$$

### 6.1.4 Équation du rendu

**Généralités** Cette équation permet d'exprimer de manière théorique la couleur d'un point  $x$  et dans la direction de sortie  $\vec{o}$ . Cette relation vous est proposée en (14).

$$\mathcal{L}_o(x, \vec{o}) = \mathcal{E}(x, \vec{o}) + \int f(\vec{\iota}, \vec{o}) \mathcal{L}_i(x, \vec{\iota}) \cos(\theta_i) d\vec{\iota} \quad (14)$$

Dans la relation (14), interviennent les termes suivants :

- ◎  $\mathcal{L}_o(x, \vec{o})$  L'intensité lumineuse de la surface en  $x$  dans la direction  $\vec{o}$  que l'on cherche à déterminer.
- ◎  $\mathcal{E}(x, \vec{o})$  L'émissivité de la surface en la direction et le point donné.
- ◎  $\mathcal{L}_i(x, \vec{\iota})$  L'intensité lumineuse qui arrive au point  $x$  depuis la direction  $\vec{\iota}$ .
- ◎  $\cos(\theta_i)$  Cosinus de l'angle entre une direction d'incidence  $\vec{\iota}$  et la normale à la surface.

**Cas spéculaire** On peut par exemple appliquer cette équation au cas spéculaire où une expression de  $f$  est connue (voir la relation (13), on obtient alors :

$$\mathcal{L}_o(x, \vec{o}) = \mathcal{E}(x, \vec{o}) + \int \mathcal{L}_i(x, 2\langle \vec{o}, \vec{n} \rangle \vec{n} - \vec{o}) \delta(2\langle \vec{\iota}, \vec{n} \rangle \vec{n} - \vec{\iota} - \vec{o}) d\vec{\iota}$$

Comme le matériau n'émet pas de lumière, on peut prendre  $\mathcal{E} = 0$ , et on obtient directement le résultat intuité :

$$\mathcal{L}_o(x, \vec{o}) = \mathcal{L}_i(x, 2\langle \vec{o}, \vec{n} \rangle \vec{n} - \vec{o})$$

On retrouve le fonctionnement intuité, à savoir que la seule contribution est celle de la direction miroir. Ceci nous indique également que dans notre code, aucune modification ne sera à prévoir dans le cas spéculaire, puisque nous utilisons déjà la seule direction possible pour le calcul de notre résultat.

### 6.1.5 Méthode de Monte-Carlo

**Généralités** Dans l'équation du rendu (14), la partie compliquée à calculer est celle de l'intégrale. Il nous faut donc trouver une méthode d'approximation pour obtenir sa valeur.

La solution retenue ici est d'utiliser une méthode de Monte-Carlo, à savoir une méthode qui va échantillonner la fonction sous l'intégrale et moyenner les résultats obtenus. Appliquée à l'intégrale de l'équation (14), nous obtenons la relation (15).

$$\int f(\vec{\iota}, \vec{o}) \mathcal{L}_i(x, \vec{\iota}) \cos(\theta_i) d\vec{\iota} \approx \frac{1}{n} \sum_{i=0}^n f(\vec{\iota}_i, \vec{o}) \mathcal{L}_i(x, \vec{\iota}_i) \frac{\cos(\theta_i)}{p(\vec{\iota}_i)} + O\left(\frac{1}{\sqrt{n}}\right) \quad (15)$$

Pour cette relation, le terme  $p(\vec{\iota}_i)$  correspond à une loi de probabilité qui est « proche de  $f$  ». Encore une fois, ce terme n'est pas le plus simple à obtenir dans le cas général.

**Cas diffus** Dans le cas diffus, la particularité est qu'il est simple de trouver une loi de probabilité proche de  $f$ . À partir de la BRDF diffuse, on en déduit que la probabilité est simplement :

$$p_{\text{diffus}}(\vec{i}) = \frac{\cos(\theta_i)}{\pi}$$

De plus, dans le cas diffus, la fonction  $f$  est constante et il nous suffit donc d'échantillonner le terme  $\cos(\theta_i)$ . Une approximation possible est alors de générer des coordonnées  $x$ ,  $y$  et  $z$ , un approximation possible donnée en (16).

$$\begin{cases} x = \cos(2\pi r_1)\sqrt{1 - r_2} \\ y = \sin(2\pi r_1)\sqrt{1 - r_2} \\ z = \sqrt{r_2} \end{cases} \quad r_1, r_2 \in \mathcal{A}([0; 1]) \quad (16)$$

Ces coordonnées vont nous permettre de déterminer l'échantillonage  $\vec{i}_i$  à utiliser. Il nous reste alors simplement à multiplier le tout par le coefficient de diffusion qui va préciser l'importance de la part de la diffusion dans la couleur finale :

$$\int f(\vec{i}, \vec{o}) \mathcal{L}_i(x, \vec{i}) \cos(\theta_i) d\vec{i} \approx \frac{1}{n} \sum_{i=0}^n i \frac{\rho_d}{\pi} \mathcal{L}_i(x, \vec{i}_i)$$

**Vitesse de convergence** Nous avons vu avec la relation (15) que la convergence se fait en inverse de la racine carrée, elle est donc assez lente. Du point de vue du code, il faudra donc considérer un certain nombre de rayons par pixels afin d'avoir un résultat intéressant, comme nous nous y intéresserons en Section 6.3.

De plus, on remarque que la solution proposée par Monte-Carlo est ici de lancer  $n$  rayons par intersection, ce qui demande de lancer  $n^r$  rayons dans le calcul d'un pixel... Une autre stratégie est de ne renvoyer qu'un rayon par intersection mais de manière aléatoire, et la moyenne permettra de combler les imprécisions, on s'en sort donc uniquement avec  $n$  rayons par pixel ce qui sera plus rapide.

Ce compromis est intéressant dans la mesure où il permet de calculer plus rapidement la couleur d'un pixel, cependant il faudra tirer un peu plus de rayons au global pour que la moyenne soit efficace.

### 6.1.6 Mettre tout ensemble

Si on résume les différents éléments vus, pour prendre en compte l'éclairage diffus, nous aurons à faire les modifications suivantes :

- **Matériaux diffus** Pour ces matériaux, il va désormais falloir lancer plusieurs rayons par pixels (on le faisait déjà pour l'antialiasing !), et surtout faire rebondir les rayons après leur passage sur le matériau en utilisant nos paramètres de récursion.
- **Autres types de matériaux** Nous avons vu avec l'exemple du spéculaire (mais c'est aussi plus ou moins le cas pour les matériaux transparents), que seule la direction de réflexion/réfraction compte, ainsi le code reste inchangé pour ces matériaux qui réalisent déjà le bon échantillonage.
- **Renvoyer les rayons** Ceci va demander un calcul de rayon aléatoire autours d'une normale, mais également de pouvoir exprimer les coordonnées  $x$ ,  $y$  et  $z$  de ce nouveau rayon dans un repère local à la surface du matériau. Il nous faudra donc être capable de générer ces repères locaux autours des normales.

## 6.2 | Code réalisé

Pour réaliser le code, nous reprenons les éléments listés auparavant concernant les adaptations à prévoir.

### 6.2.1 Génération de la direction aléatoire

Cette direction aléatoire va partir du point d'intersection (noté  $P$ ), et est donnée par la relation (16). Pour utiliser les coordonnées fournies par cette relation, il nous faut déterminer un repère autours de la normale  $\vec{n}$  à la surface de l'objet au point  $P$ . Nous considérons que la normale est différente de  $\vec{0}$  (ce qui poserait des problèmes dans tout le code !), ainsi une de ses composantes est non nulle.

Si on note  $\vec{n} = (n_0; n_1; n_2)$  alors les vecteurs  $(-n_1; n_0; 0)$  et  $(0; n_2; -n_1)$  sont orthogonaux à  $\vec{n}$ . De plus, comme  $n_{0..2}$  ne sont pas tous nuls, au moins un de ces vecteur est non nul, notons le  $\vec{d}_1$ .

Nous avons donc déjà deux directions pour notre repère, reste à récupérer la troisième directement par produit vectoriel :

$$\vec{d}_2 = \vec{n} \wedge \vec{d}_1$$

Nous avons ainsi un repère orthonormal autours de  $\vec{n}$  si on a normalisé  $\vec{d}_{1,2}$ .

Du point de vue du code, ceci est transparent grâce aux méthode implémentée dans la classe `Vector`:

```

1 void Vector::randomCos(const Vector &N){
2     // Les coordonnées aléatoires pour le nouveau repère.
3     double u = distrib(engine);
4     double v = distrib(engine);
5     double x = cos(2*PI*u)*sqrt(1-v);
6     double y = sin(2*PI*u)*sqrt(1-v);
7     double z = sqrt(v);
8
9     // Construction d'un repère orthonormé autours de N
10    Vector directionUn = (N[0] != 0 || N[1] != 0) ? Vector(N[1], -N[0], 0) : Vector
11        (0, N[2], -N[1]);
12    directionUn.normalize();
13    Vector directionDeux = N.cross(directionUn);
14
15    // Calcul des coordonnées du rayon à partir :
16    //     - des coordonnées des axes du nouveau repère
17    //     - des coordonnées aléatoires précédemment tirées.
18    Vector result;
19    result = x*directionUn + y*directionDeux + z*N;
20    this->xxyz[0] = result[0];
21    this->xxyz[1] = result[1];
22    this->xxyz[2] = result[2];
23 }
```

On remarquera que notre code modifie directement le vecteur sur lequel se fait l'appel pour qu'il représente la direction aléatoire.

### 6.2.2 Adaptation de la méthode `getColor`

Comme nous l'avons vu, une adaptation est nécessaire pour notre méthode `getColor`, afin que cette dernière prenne en compte les rebonds pour les matériaux diffus. Nous allons ainsi changer légèrement le code ce ce cas dans cette méthode.

Dans cette nouvelle version, nous recherchons toujours la présence d'ombres portées. Cependant au lieu de renvoyer directement le résultat comme nous le faisions, nous allons à la place faire rebondir le rayon (s'il reste encore des récurrence à faire !) et ajouter le résultat à celui obtenu précédemment. Du point de vue du code, les ajouts sont alors les suivants :

```

1 if(objects[objectId]->material.isDiffuse){
2     ...
3     // Dans le cas diffus, on prend en compte l'éclairage indirect.
4     // Pour cela on renvoie un rayon depuis la surface de manière aléatoire (cf.
4         BRDF de la surface diffuse).
5     Vector finalColor = shadow_coeff * (1500*std::max(0., dot(N,l))/distLight2)*
          objects[objectId]->material.color;
```

```

6   if(recursion > 0){
7     // Génération de la direction aléatoire.
8     Vector randomDirection;
9     randomDirection.randomCos(N);
10    Vector indirect = getColor(Ray(P+0.001*N, randomDirection), recursion-1);
11    finalColor = finalColor + objects[objectId]->material.diffusionCoeff*objects[
12      objectId]->material.color*indirect*(1./PI);
13  }
14 }

```

On remarque que le résultats précédent est désormais stocké dans une variable `finalColor` à laquelle on ajoute la contribution du rebond. Cette dernière va prendre en compte les éléments suivants :

- **Coefficient de diffusion** Le matériau ne va pas se comporter comme un miroir, ainsi la contribution du rebond doit être modulée, et elle l'est par ce coefficient. Pour prendre en compte ce coefficient, nous avons ajouté un nouvel attribut à la classe `Material`.
- **Couleur obtenue par rebond** Cette dernière est obtenue en faisant rebondir le rayon dans la direction `randomDirection` qui est calculée avec la méthode proposée à la section précédente (`randomCos`).
- **Couleur du matériau** La couleur du rebond est également modulée directement par la couleur de l'objet sur lequel le rayon a rebondi (comme nous l'avons fait pour les miroirs).

Pour les autres types de matériau, aucune modification n'est à apporter.

### 6.2.3 Envoyer plusieurs rayons par pixels

Ceci avait déjà été mis en place auparavant pour l'antialiasing. Ainsi, la structure pour envoyer plusieurs rayons par pixels est déjà disponibles dans le fichier `main.cpp`. Cependant, l'antialiasing et cet éclairage diffus augmentent considérablement le temps de calcul ( $n$  rayons par pixels et les surfaces diffuses qui renvoient les rayons), ainsi nous avons utiliser les possibilités de parallélisme avec l'instruction suivante :

```
1 #pragma omp parallel for
```

Ces quelques modifications nous permettent désormais de tester notre éclairage diffus.

## 6.3 Résultats obtenus

Avec le code présenté, nous pouvons alors commencer à tester le résultat. Le point intéressant ici est d'estimer la précision obtenue sur l'image ainsi que le temps de calcul pour cette dernière. Pour ce rapport, nous avons cherché à pouvoir voir les évolutions et avons donc eu besoin de calculer un certain nombre d'images. Comme ce processus prend du temps (et que ma machine n'est pas particulièrement performante), je me suis restreint à des images de taille  $350 \times 350$  afin de pouvoir calculer plusieurs échantillons dans un laps de temps raisonnable sur ma machine.

Les résultats obtenus sont consignés à la Table 2, nous avons pris un maximum de 5 rebonds pour les calculs.

Nombres de rayons	1	1	3	5	10	25	50	100
Parallélisme	non	oui	oui	oui	oui	oui	oui	oui
Temps d'exécution	11,966	5,356	15,198	23,832	42,945	106,929	221,369	443,161

TABLE 2 • Différents temps de calculs pour une image de taille  $350 \times 350$  avec 5 rebonds maximum

Sans surprise, on remarque que le temps est fortement diminué en utilisant la parallélisation (facteur deux à quatre selon les situations testées). De même, passé un palier initial ou l'évolution n'est pas totalement linéaire, le temps de calcul évolue ensuite linéairement avec la taille de l'image. On remarque

en particulier que sur l'ordinateur utilisé, le calcul de l'image en dimension  $1024 \times 1024$  aurait pris un peu plus d'une heure...

Regardons alors les visuels obtenus pour ces différentes situations, qui sont regroupés à la Figure 19.

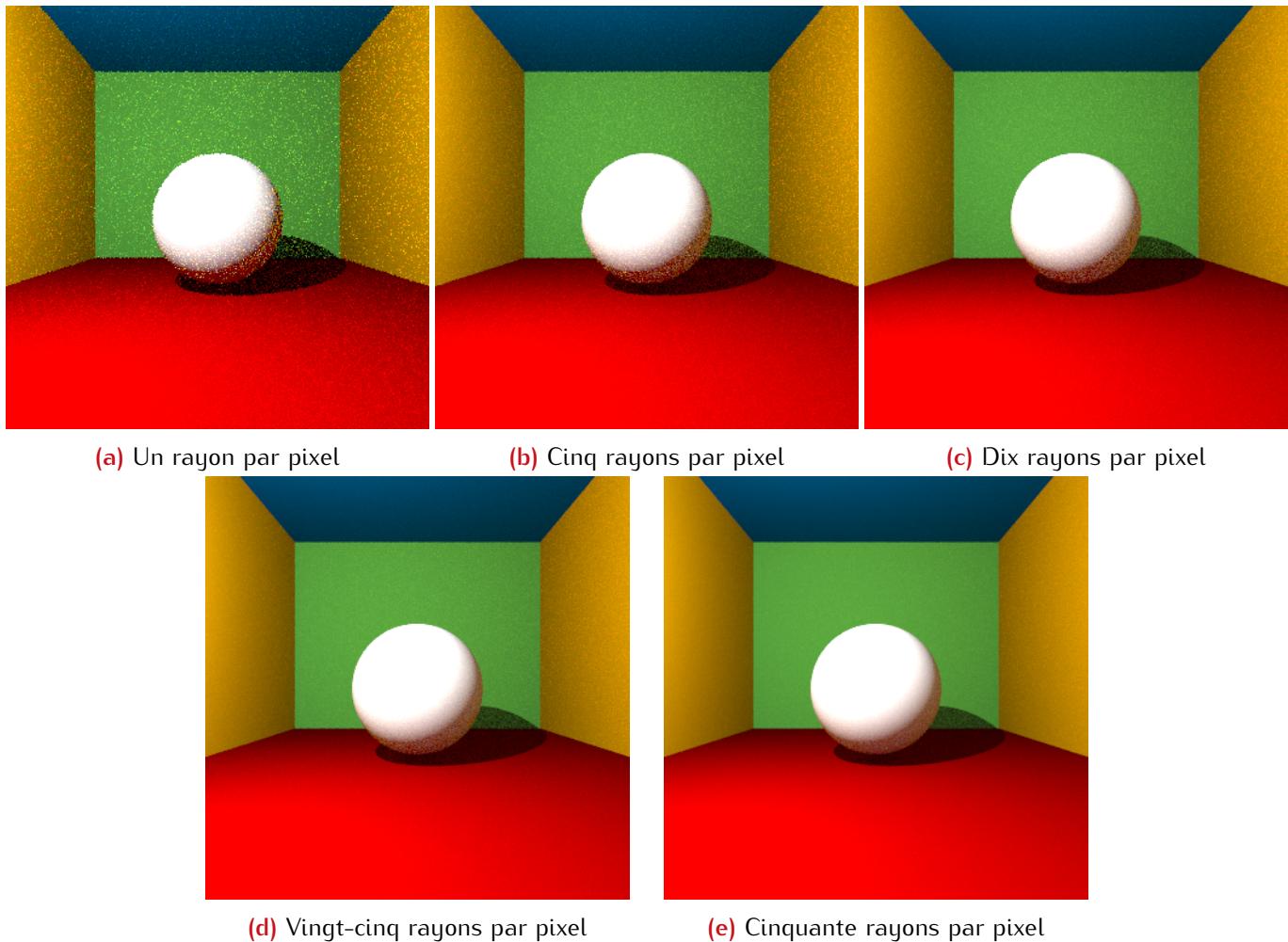


FIGURE 19 • Tests de l'éclairage diffus avec plusieurs valeurs de rayons par pixels

On retrouve bien le fait que plus il y ait de rayons par pixels, plus le résultat est net, ce qui est cohérent. Si la différence entre le cas 25 et 50 n'est pas flagrant sur ces petites version, il l'est sur l'original ou les murs sont en particulier plus unis dans l'image avec cinquante rayons par pixels.

Il faudra donc toujours chercher un équilibre entre précision attendue pour l'image et temps de calcul (et ici ma machine ne joue pas en ma faveur...).

## 7 • Source étendue

---

### 7.1 Aspects théoriques

---

#### 7.1.1 Motivations

Si le réalisme des résultats obtenus jusqu'à présent s'améliore, un remarque peut être objectée concernant les ombres. En effet, ces dernières sont actuellement un peu trop franches. Une solution pour améliorer cette situation est de changer le modèle de lumière.

Pour le moment, nous avions considéré uniquement des lumières ponctuelles, et pour déterminer les ombres portées, nous regardions si un objet était entre le point d'intersection et la source de lumière. Le cas échéant un facteur d'ombre est appliqué, autrement la couleur est rendue inchangée.

En utilisant une source étendue, nous allons alors permettre de prendre en compte le fait qu'en échantillonnant en direction de la source de lumière, des rayons peuvent être bloqués et d'autres non. On obtiendra ainsi des ombres normalement plus douces, ce qui serait un peu plus réaliste.

#### 7.1.2 Modélisation

Une première solution est d'ajouter un facteur d'émissivité dans le matériau, de retirer la source ponctuelle et de « laisser faire ». Cependant, les chances qu'un rebond atterrisse sur la lumière seront alors assez faibles, ce qui fournirait une image très bruitée.

Une autre solution possible est de considérer que l'on va échantillonner vers la source de lumière. Cette solution est celle retenue pour ce code. Ainsi, au lieu de renvoyer les rayons totalement au hasard dans la scène, nous allons essayer d'échantillonner dans la direction de la lumière.

Pour ce faire, on va chercher dans quelle direction est la source lumineuse à partir d'un point d'intersection et tirer un nouveau rayon depuis ce point vers une direction choisie aléatoirement en direction de la source lumineuse.

### 7.2 Code réalisé

---

Pour réaliser ce point, nous avons besoin de pouvoir faire trois choses :

- **Adapter la classe Scene** Afin que cette dernière traite ce nouveau cas séparément du cas où l'éclairage est ponctuel. Nous avions expliqué dans la partie sur les petites améliorations que ceci était réalisé par un indicateur de position de la sphère lumineuse dans la liste des objets. Nous ne reviendrons donc pas sur cet aspect, mais uniquement sur le renvoie du nouveau rayon.
- **Être capable de générer le rayon** Ceci demande en particulier de savoir générer un repère autours de la normale au point d'intersection. Ceci a déjà été utilisé pour l'éclairage diffus, et sera donc repris ici (un peu de refactoring a été mis en place à cette occasion).
- **Représenter des sphères lumineuses** Pour cela, il nous faut transformer un peu la classe *Material* en y ajoutant un champs d'émissivité qui permet passé une certaine valeur de savoir que l'on a affaire à une source de lumière.

Nous allons traiter ces trois aspects dans la suite de cette section.

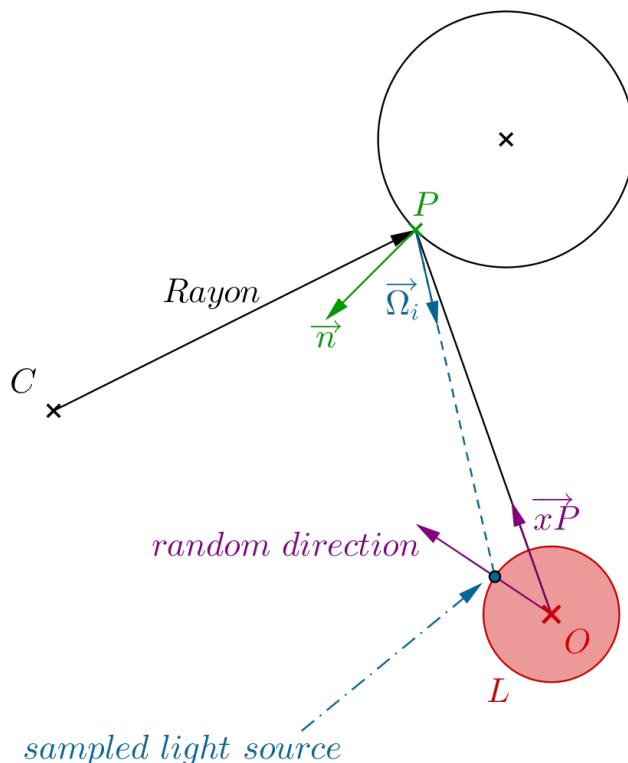
#### 7.2.1 Modification de Material

Pour prendre en compte ce cas, nous ajoutons un attribut *emissivity* de type *double* à cette classe. Le constructeur est également mis à jour pour prendre en compte ce nouveau paramètre.

La valeur par défaut est laissée à 1, ce qui correspond à un matériau neutre (ie. qui n'émet ni n'absorbe de lumière).

### 7.2.2 Regénérer un nouveau rayon

Ceci est fait conjointement entre les classes *Vector* et *Scene*. Pour regénérer ce nouveau rayon, nous nous trouvons dans la situation de la Figure 20.



**FIGURE 20** • Génération d'un nouveau rayon aléatoire dont la loi est centrée sur la source de lumière

La stratégie est en fait ici de procéder comme suit :

- 1. On échantillonne depuis la source de lumière une direction aléatoire en utilisant notre méthode *randomCos* précédemment abordée. On part pour cela du vecteur  $\vec{xP} = \vec{OP}$  et on échantillonne autours pour trouver une direction aléatoire. Cette étape figure en violet sur la Figure 20.
- 2. On définit ensuite le point de la sphère lumineuse qui intersecte  $O + t \times \vec{\text{randomDir}}$ . On obtient ainsi un point échantilloné sur la surface de la sphère lumineuse (*sampledLightSource*). Ces étapes sont en bleu sur la Figure 20.
- 3. On définit alors le rayon dirigé par  $\Omega_i$  qui part de  $P$  et va vers ce point échantilloné. C'est ce vecteur qui servira à diriger le nouveau rayon à renvoyer.

Du point de vue du code, on se retrouve alors avec les éléments suivants dans la nouvelle version de *getColor*.

```

1 // L'éclairage se fait par une source étendue.
2 // On échantillonne dans la direction de la source lumineuse.
3 Vector xP = (P-dynamic_cast<Sphere*>(objects[this->luxSphereIndex])->0) .
   getNormalize();
4 Vector randomDirection;
5 randomDirection.randomCos(xP);
6 Vector sampledLightSource = randomDirection*dynamic_cast<Sphere*>(objects[this->
   luxSphereIndex])->R + dynamic_cast<Sphere*>(objects[this->luxSphereIndex])->0;
7 double distLight = (sampledLightSource-P).squaredNorm();
8 Vector omega_i = (sampledLightSource-P).getNormalize();
9 double shadow_coeff = 1;
10 Vector PPrime, NPrime;
11 int idPrime;

```

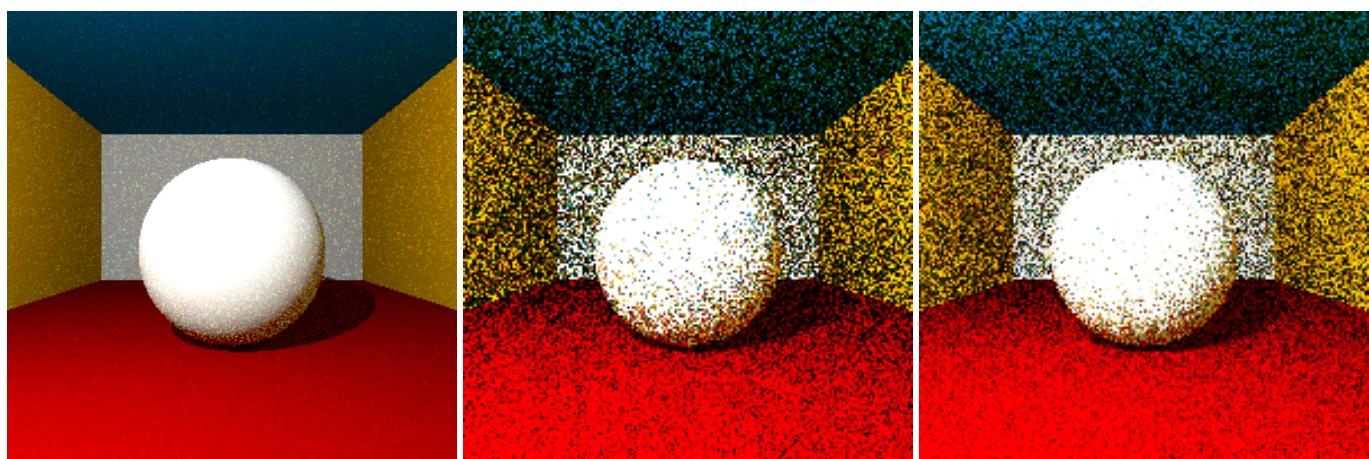
```

12 Material materialPrime;
13
14 if(intersect(Ray(P+0.001*N, omega_i), PPrime, NPrime, idPrime, materialPrime)){
15   if(idPrime != 0){
16     shadow_coeff = 0;
17   }
18 }
```

La suite est la même que celle vue pour l'éclairage diffus. Le temps de calcul sera cependant plus long, dans la mesure où il faut regénérer un nouveau repère à chaque intersection, ce qui est un peu coûteux.

## 7.3 | Résultats obtenus

Avec le code proposé, nous avons obtenu les résultats de la Figure 21.



(a) Source ponctuelle et 5 rayons par pixels      (b) Source étendue et 300 rayons par pixels      (c) Source étendue et 500 rayons par pixels

**FIGURE 21 •** Une scène témoin et les résultats en utilisant une source étendue

On remarquera que le fait d'utiliser une source étendue implique un bruit plus important dans l'image. Ainsi, pour avoir un résultat comparable à celui d'une source ponctuelle, il faut alors utiliser plus de rayons par pixels, ce qui augmente encore les temps de calcul. Comme le temps de calcul a fini par être important sur la machine de rendu, j'ai abaissé la résolution de l'image (ce qui n'aide pas à avoir un meilleur rendu!).

Quoiqu'il en soit, le résultat est tout de même extrêmement bruité, ainsi le code semble être partiellement buggué. Je n'ai cependant pas réussi à déterminer l'origine de ce problème.

Dans la mesure où les temps de calculs sont tout de même significativeemnt plus important avec cette version du code, les rendus de la suite du rapport n'utiliseront pas l'éclairage par source étendue, afin de gagner du temps de calcul des images.

## 8 • CSG

### 8.1 Objectifs

L'idée du CSG (*Construction Solid Geometry*) est une méthode permettant de modéliser des solides exotiques à partir de solides de bases. Ainsi, en utilisant des opérations ensemblistes comme l'intersection, l'union ou encore la différence, on va combiner ces différentes briques initiales pour obtenir des objets plus originaux. Un exemple est fourni à la Figure 22 explique le principe de cet outil.

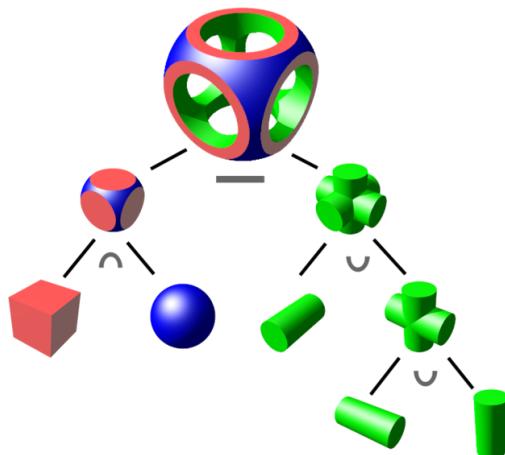


FIGURE 22 • Illustration du principe de la CSG

On remarque sur cette figure l'utilisation d'union (partie gauche en verte), d'intersection (partie rouge) et de différence (dernière étape). De plus, on peut observer que seules trois formes ont été utilisées : des cylindres, des sphères et un cube.

Nos objectifs vont donc être de mettre en place une structure capable de réaliser de telles opérations.

### 8.2 Structure du code

Pour résumer les besoins, nous devons prévoir les éléments suivants :

- *Des classes pour les opérations* Les trois opérations proposées vont fournir en sortie un objet. L'idée est alors de dire que les classes qui vont réaliser ces opérations seront directement les objets attendus. Nous prévoyons donc trois classes pour les opérations :

*Intersection      Union      Subtraction*

Ces différentes classes et les stratégies à adopter pour ces dernières seront détaillées aux Sections 8.3 à 8.5.

Ces classes doivent permettre de traiter des objets et éventuellement de les regrouper sous un même matériau. Dans la mesure où ceci n'est pas toujours souhaité, ces classes fourniront un paramètre pour savoir si on veut mutualiser ou non les matériaux.

- *Des classes pour les formes* Nous disposons déjà d'une classe pour représenter les sphères, cette dernière pourra être réutilisée. Nous avons également ajouté quatre autres formes :

*Plan      Cylinder      Torus      Box*

De plus, ces classes devront pouvoir s'interfacer entre elles pour exécuter les opérations dessus, elles devront donc partager un certain nombre de méthodes qui seront présentes dans la classe mère *Object*.

Les détails sur ces classes seront fournies dans les Sections 8.7 à 8.11.

- ➊ **Une classe pour les points d'intersection** Lors de la réalisation des opérations, il nous faudra décider parmi des points potentiels si ces derniers seront dans l'union, l'intersection,...Pour cela, nous allons regrouper les points dans un tableau de points. À ce compte, nous avons créé une classe pour les représenter, nommée *IntersectionPointCSG*.

Son utilisation sera explicitée plus en détails dans les spécifications des différentes classes.

- ➋ **Fonction d'intersection** Nous allons également mutualiser la fonction d'intersection dans la classe *Object*, qui fera les traitements communs aux différents objets et déléguera les calculs spécifiques aux classes associées.

On remarque donc que l'on essaie d'utiliser au maximum les classes déjà existantes qui vont être modifiées pour ce nouvel usage.

## 8.3 | Réalisation de l'union

### 8.3.1 Présentation de la classe *Union*

La classe associée à l'union est simplement nommée *Union*. Cette dernière va permettre de lier deux objets *a* et *b* en un nouvel objet qui aura un matériau donné (*material*).

Comme nous l'avons mentionné, cette classe va hériter de la classe générique *Object*, qui lui impose donc de définir deux méthodes :

- ➊ *isInside* Cette méthode va permettre de savoir si un point *P* est situé dans l'objet ou non.
- ➋ *intersection* Cette méthode va permettre de déterminer les points d'intersections entre l'objet de l'union et un rayon. Comme nous l'avons expliqué, nous avons pris le parti de retourner les points d'intersections sous la forme d'un tableau de points (chacun de classe *IntersectionPointCSG*) avant son utilisation par la classe *Scene*.

Dans la suite de cette partie, nous allons détailler ces deux méthodes.

### 8.3.2 Fonction pour tester l'appartenance d'un point

La méthode *isInside* va simplement prendre en paramètre un point *P* et déterminer s'il est dans l'union. La définition de l'union est ici de dire (en utilisant des opérateurs logiques) que :

$$\begin{aligned} P \in \text{Union}(a, b) &\iff P \in a \cup b \\ &\iff P \in a \text{ || } P \in b \end{aligned}$$

Ainsi, pour coder l'union il nous suffit uniquement d'utiliser un `//`, d'où le code qui est simple :

```
1 bool Union::isInside(const Vector &P) const{
2     return a->isInside(P) || b->isInside(P);
3 }
```

On remarque donc que dans les fait, cette méthode va juste déléguer aux différents types d'objets de donner leur avis et de réutiliser leurs résultats, d'où un code assez transparent.

### 8.3.3 Fonction d'intersection

Dans la mesure où l'union est définie comme le fait d'être dans *a* ou dans *b*, l'algorithme pour l'intersection vient directement :

- ➊ 1. Déterminer les intersections du rayon avec *a* et *b*.
- ➋ 2. Pour chaque point d'intersection avec *a* (représenté par un objet de type *IntersectionPointCSG*) on regarde s'il est aussi dans *b*. Si c'est le cas, on ne l'inclus pas ; sinon on l'ajoute à la liste des points d'intersection finaux.
- ➌ 3. On fait de même avec *b*, en ne gardant que les points d'intersection qui ne sont pas dans *a*.

On obtient in fine une liste de points d'intersection qui composent notre union. On remarquera que l'on a exclus les points étant dans les deux objets à la fois. La raison est qu'autrement nous aurons des doublons non nécessaires. Ainsi, on limite le nombre de points finaux pour le calcul et on définit de manière unique toute la surface de notre union.

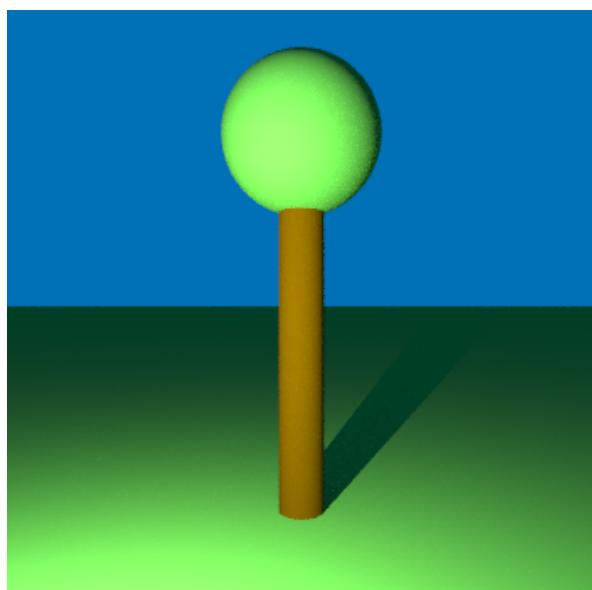
Le code vient de lui-même et peut être consulté dans le fichier `union.cpp`.

Encore une fois, une grande partie du travail est délégué à des méthodes qui sont redéfinies dans les différents types d'objets.

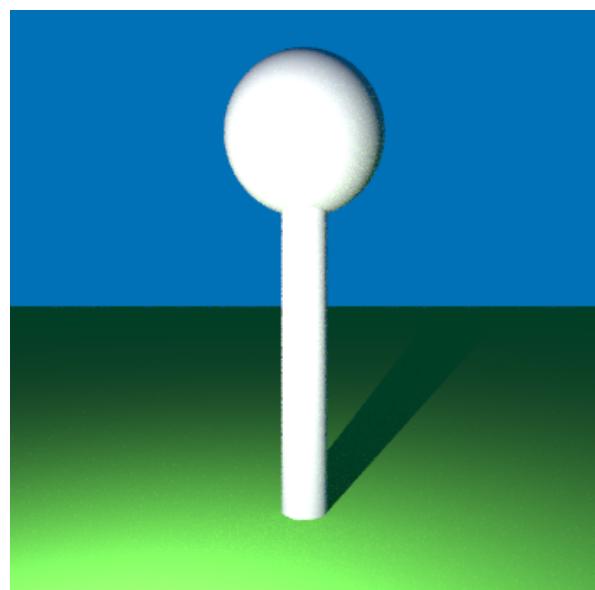
### 8.3.4 Mise en pratique

Des mises en œuvre plus avancées seront exposées un peu plus tard une fois que toutes les opérations et tous les objets seront définis. Pour commencer, nous avons essayé une situation simple unissant un cylindre et un sphère (par exemple pour faire un arbre très simple).

Deux résultats vous sont alors proposés à la Figure 23.



(a) Exemple d'union avec matériaux différents



(b) Exemple d'union en mutualisant le matériau

**FIGURE 23** • Test d'union entre une sphère et un cylindre

Pour obtenir les figures de la Figure 23, nous avons de plus utilisé un plan vert pour le sol, et mis la couleur par défaut de la scène pour les rayons perdus en bleu.

## 8.4 | Réalisation de l'intersection

### 8.4.1 Présentation de la classe `Intersection`

La structure de cette classe est similaire à celle de l'union, avec les mêmes méthodes proposées. Nous allons donc détailler ces méthodes ici.

### 8.4.2 Fonction pour tester l'appartenance d'un point

Comme pour le cas de l'union, nous allons exprimer l'intersection de deux objets en terme d'opérateurs logiques. Nous avons immédiatement la relation :

$$\begin{aligned} P \in \text{Intersection}(a, b) &\iff P \in a \cap b \\ &\iff P \in a \wedge P \in b \end{aligned}$$

Ainsi, pour coder l'union il nous suffit uniquement d'utiliser un `&&` d'où le code qui est simple :

```

1 bool Intersection::isInside(const Vector &P) const {
2     return a->isInside(P) && b->isInside(P);
3 }
```

Tout comme l'union, on délègue les calculs importants aux classes des objets.

### 8.4.3 Fonction d'intersection à un rayon

Dans la mesure où l'intersection est définie comme le fait d'être dans *a* et dans *b*, l'algorithme pour l'intersection vient directement :

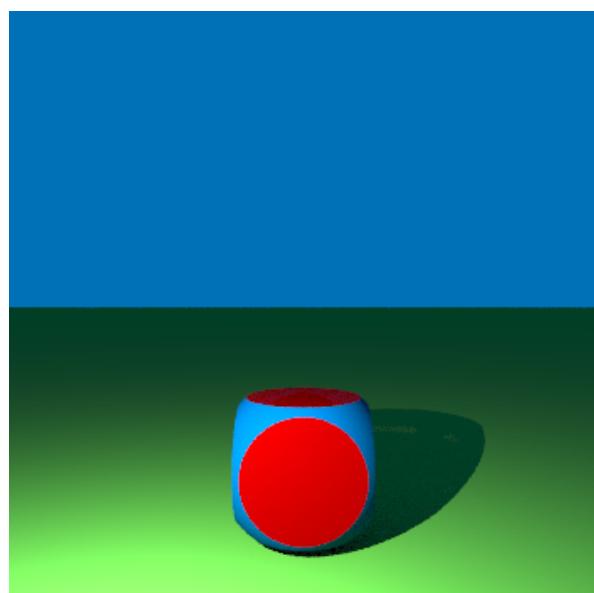
- 1. Déterminer les intersections du rayon avec *a* et *b*.
- 2. Pour chaque point d'intersection avec *a* (représenté par un objet de type *IntersectionPointCSG*) on regarde s'il est aussi dans *b*. Si c'est le cas, on l'inclus ; sinon on ne le prend pas en compte.
- 3. On fait de même avec *b*, en ne gardant que les points d'intersection qui sont également dans *a*.

Encore une fois, une grande partie du travail est délégué à des méthodes qui sont redéfinies dans les différents types d'objets. Le code est disponible dans le fichier *intersection.cpp*.

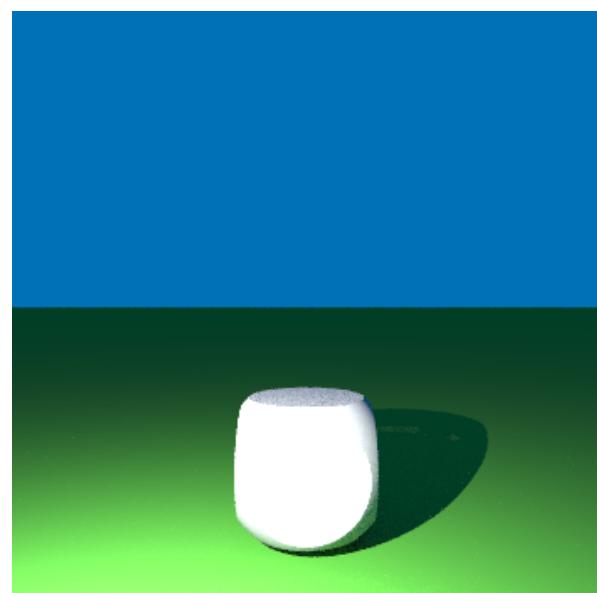
### 8.4.4 Mise en pratique

Des mises en œuvre plus avancées seront exposées un peu plus tard une fois que toutes les opérations et tous les objets seront définis. Pour commencer, nous avons essayé une situation simple avec l'intersection entre une sphère et un cube.

Deux résultats vous sont alors proposés à la Figure 24.



(a) Exemple d'intersection avec matériaux différents



(b) Exemple d'intersection en mutualisant le matériau

**FIGURE 24** • Test d'intersection entre une sphère et un cube

Pour obtenir les figures de la Figure 24, nous avons de plus utilisé un plan vert pour le sol, et mis la couleur par défaut de la scène pour les rayons perdus en bleu.

## 8.5 | Réalisation de la différence

### 8.5.1 Présentation de la classe Substraction

On retrouve la même structure que pour les intersections et les unions, nous allons donc détailler très succinctement cette classe et ses méthodes.

### 8.5.2 Fonction pour tester l'appartenance d'un point

En utilisant des opérateurs logiques, la différence s'écrit directement :

$$\begin{aligned} P \in \text{Différence}(a, b) &\iff P \in a - \{b\} \\ &\iff P \in \{x / x \in a \wedge x \notin b\} \end{aligned}$$

On remarque donc qu'il faut utiliser différents opérateurs pour pouvoir exprimer cette opération. Nous obtenons alors le code suivant :

```
1 bool Subtraction::isInside(const Vector &P) const{
2     return this->a->isInside(P) && !this->b->isInside(P);
3 }
```

À la différence des deux autres opérations, cette dernière n'est pas symétrique, l'ordre des arguments a donc ici une importance cruciale !

### 8.5.3 Fonction d'intersection à un rayon

Dans la mesure où l'intersection est définie comme le fait d'être dans *a* sans être dans *b*, l'algorithme pour l'intersection vient directement :

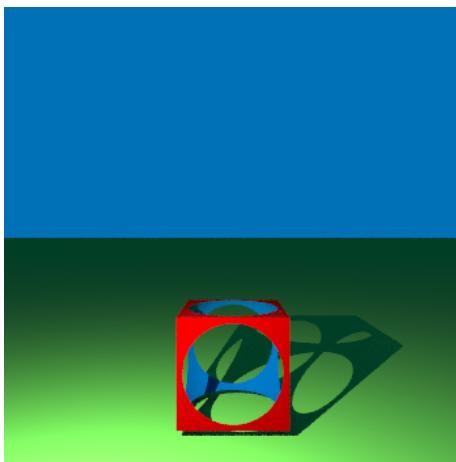
- 1. Déterminer les intersections du rayon avec *a* et *b*.
- 2. Pour chaque point d'intersection avec *a* (représenté par un objet de type *IntersectionPointCSG*) on regarde s'il est aussi dans *b*. Si c'est le cas, on ne l'inclus pas, sinon on l'ajoute.
- 3. Pour les points d'intersection avec *b*, on va garder ceux qui sont également dans *a* et qui appartiendront donc à la limite de *b* dans *a*. Il est important de garder ces points limitrophe, car ils permettront de bien délimiter le contours de *b* dans *a*. Pour ces derniers, on va cependant inverser la normale afin qu'ils représentent bien des points de la surface de *a* et pas de celle de *b*.

Encore une fois, une grande partie du travail est délégué à des méthodes qui sont redéfinies dans les différents types d'objets. Le code est disponible dans le fichier *subtraction.cpp*.

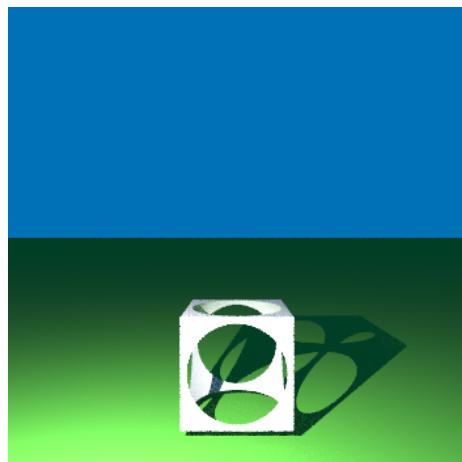
### 8.5.4 Mise en pratique

Des mises en œuvre plus avancées seront exposées un peu plus tard une fois que toutes les opérations et tous les objets seront définis. Pour commencer, nous avons essayé une situation simple avec l'intersection entre une sphère et un cube.

Deux résultats vous sont alors proposés à la Figure 25.



(a) Exemple de différence avec matériaux différents



(b) Exemple de différence en mutualisant le matériau

**FIGURE 25** • Test de différences entre une sphère et un cube

Pour obtenir les figures de la Figure 25, nous avons de plus utilisé un plan vert pour le sol, et mis la couleur par défaut de la scène pour les rayons perdus en bleu.

## 8.6 | Résumé des opérations

Pour résumer ces opérations, nous allons les appliquer toutes les trois à un même couple d'objet. Ce dernier a été déjà partiellement utilisé, il s'agit d'une sphère bleue et d'un cube rouge, représentés Figure 26.

(a) Cube rouge utilisé

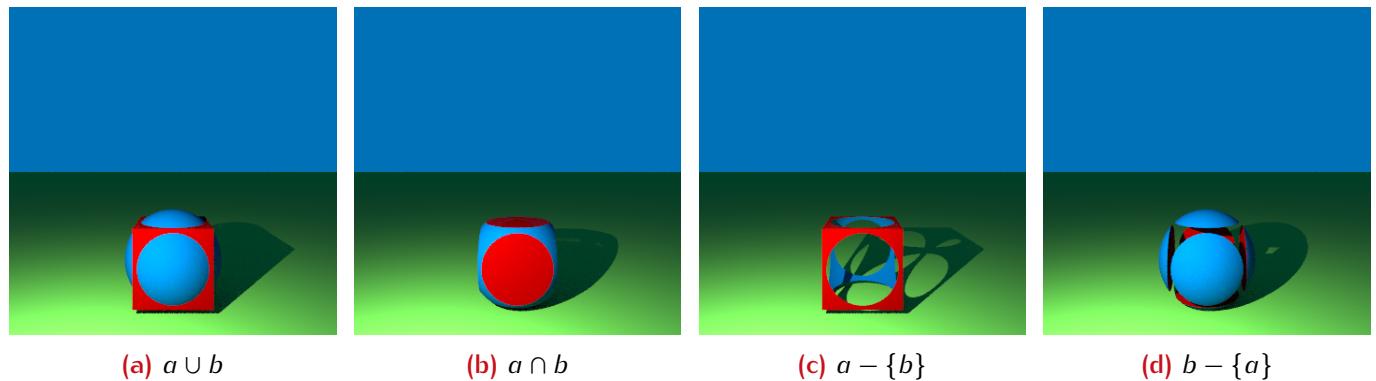
(b) Sphère bleue utilisée

**FIGURE 26** • Objets utilisés pour comparer les différentes actions dans notre CSG

Nous avons alors testé les quatre opérations possibles :

$$a \cup b \quad a \cap b \quad a - \{b\} \quad b - \{a\}$$

Les résultats sont exposés à la Figure 27.

**FIGURE 27** • Toutes les opérations possibles avec notre système de CSG

## 8.7 | Travail avec des plans

### 8.7.1 Généralités

Ce premier objet n'est pas à proprement parler un objet utilisé pour la CSG. Cependant, ce dernier a été ajouté au moment de l'ajout des autres formes, et offre une première approche simple des besoins de modélisation pour nos objets.

Pour la CSG, nous avons vu que nous avions besoin de deux méthodes :

- *isInside* Une fonction qui permet de savoir si un point donné est dans la forme concernée.
- *intersect* Une fonction qui calcule les points d'intersection entre la forme concernée et un rayon donné.

Ces deux méthodes sont les seules dont nous avons besoin, et en particulier ce sont les seules appelées sur les objets par les différentes classes modélisant les opérations et que nous venons d'aborder. Pour cela, nous allons préciser ici (et pour chaque autre type d'objet ensuite) la stratégie adoptée pour répondre à ces deux besoins.

Un autre aspect qui sera abordé est comment avons-nous défini les différents solides, ie. la signification de leurs attributs.

### 8.7.2 Modélisation du plan

Un plan a été modélisé d'une manière simple avec deux éléments : Ainsi, le plan d'équation  $z = 1$  serait tout simplement défini en prenant un point  $P_0(0, 0, 1)$  et le vecteur normal  $\vec{n} = (0, 0, \pm 1)$ . On remarque que la normale a son importance ici et peut conditionner le résultat de l'image final. En effet, une normale

orientée « dans le mauvais sens » créerait une ombre non réaliste. Ceci peut être corrigé dans le cas général, mais comme la caméra ne peut voir qu'un côté d'un plan à la fois, nous avons considéré que ce point pouvait être réglé manuellement à la création du plan pour que le rendu soit adéquat.

### 8.7.3 Savoir si un point est de le plan

Il s'agit de la missions de la méthode `isInside`. Dans le cas du plan, il s'agit juste de savoir si ce point  $P$  vérifie :

$$\langle \overrightarrow{PP_0}, \vec{n} \rangle = 0$$

On teste ainsi juste que le vecteur soit bien orthogonal au plan. Le code de `isInside` est donc immédiat :

```
1 bool Plan::isInside(const Vector& P) const{
2     return abs(dot(P->P0, N)) < 0.0001;
3 }
```

On remarquera que l'on s'autorise une certaine marge d'erreur, dans la mesure où un point ne sera jamais *exactement* sur le plan.

### 8.7.4 Trouver les points d'intersection

Ceci est réalisé par la méthode `intersect`. Dans le cas du plan, l'algorithme est assez simple :

- 1. On vérifie que le rayon n'est pas parallèle au plan, dans ce cas il n'y aura pas (ou que) des intersections, ces cas sont donc mis de côté.
- 2. On calcule la valeur de  $t$  (position sur le rayon) pour laquelle l'intersection a lieu. Pour cela, on projette sur une droite portée par  $\vec{n}$  les différentes grandeurs et on normalise par la distance de la projection de  $\vec{u}$  et sur  $\vec{n}$ . On obtient alors la relation :

$$t = \frac{\langle \overrightarrow{CP_0}, \vec{n} \rangle}{\langle \vec{u}, \vec{n} \rangle}$$

Les grandeurs sont observables sur la Figure 28.

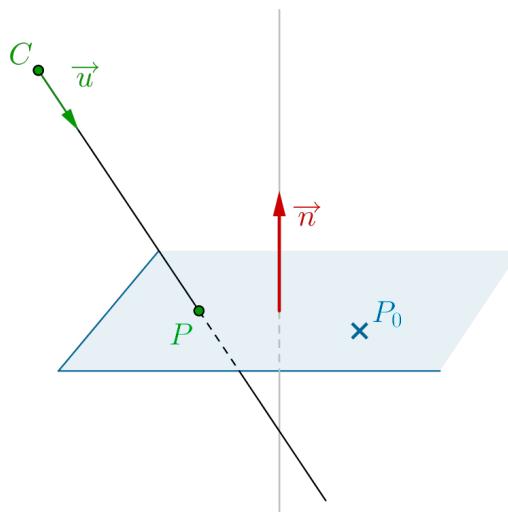


FIGURE 28 • Intersection entre un rayon et un plan

- 3. On initialise les autres grandeurs, à savoir le matériau et la normale (cette dernière est directement celle du plan), et on peut ajouter le point d'intersection à la liste de points passée en paramètre.

Le code est alors transparent en reprenant cette structure :

```

1 void Plan::intersect(const Ray& r, vector<IntersectionPointCSG *>& LI) const{
2     // On vérifie que le rayon n'est pas parallèle au plan.
3     if(abs(dot(r.u, this->N)) > 0.0001){
4         // Calcul du t où a lieu l'intersection par projections.
5         double t = dot(this->P0-r.C, this->N)/dot(r.u, this->N);
6         Vector P = r.C + t*r.u;           // Point d'intersection.
7         Vector N_ = this->N;           // Normale en ce point (N car le point
8             est sur le plan).
9         Material m = this->material;   // Matériaux de l'objet
10
11         // Ajout du point à la liste des intersections connues
12         IntersectionPointCSG* IP = new IntersectionPointCSG(P, N_, m);
13         LI.push_back(IP);
14     }
}

```

## 8.8 | Travail avec des sphères

### 8.8.1 Généralités

Le cas de la sphère a déjà été partiellement traité auparavant, puisque le code initial permettait justement de ne représenter que des sphères. Ainsi, pour inclure le CSG dans notre code, la classe liée aux sphères a juste dû changer un peu d'aspect pour s'adapter aux nouveaux besoins.

Nous allons donc redétailler les nouvelles méthodes dont nous avons besoin, *isInside* et *intersect*.

### 8.8.2 Modélisation d'une sphère

Comme nous l'avons vu, les sphères sont modélisées par deux grandeurs :

- ◎ *Un point C* Il représente le centre de la sphère.
- ◎ *Un double R* Il représente le rayon de la sphère.

Rien de nouveau ici, tous ces éléments ont déjà été vus.

### 8.8.3 Savoir si un point est de la sphère

Pour savoir si un point est une sphère, il s'agit uniquement de savoir si sa distance au centre de la sphère est inférieure au rayon, à savoir :

$$\|P - C\|_2^2 \leq R^2$$

Le code associé est donc immédiatement :

```

1 bool Sphere::isInside(const Vector &P) const{
2     return (P-0).squaredNorm() < pow(R, 2);
3 }

```

### 8.8.4 Trouver les points d'intersection avec la sphère

Dans le code initial, la méthode calculait directement les points d'intersection, et toutes les grandeurs associées. Dans l'optique d'un CSG, nous avons besoin de récupérer tous les points d'intersection avant d'en faire le tri. Ainsi, à la place de décider directement dans la classe *Sphere* quel était le point d'intersection le plus proche du départ du rayon, nous allons à la place enregistrer tous les points d'intersection trouvés (et leurs normales) dans le tableau de points d'intersections.

Le code est donc encore plus simple qu'auparavant, puisque toute solution est gardée pour un tri ultérieur. Ainsi, si on reprend le code précédemment vu, chaque cas *t1>0* et *t2>0* aura le droit d'ajouter son point d'intersection trouvé à la liste finale.

## 8.9 | Travail avec des cylindres

### 8.9.1 Généralités

Pour la modélisation du cylindre, plusieurs solutions étaient envisageables :

- ◎ Utiliser uniquement des cylindres infinis, qui sont plus simples à modéliser.
- ◎ Utiliser des cylindres orientés selon les axes.

Le parti pris est ici de considérer des cylindres quelconques, de taille finie et non nécessairement alignés par rapport aux axes. Cependant, le cas particulier des cylindres orientés le long des axes a permis de guider les calculs pour le cas général.

### 8.9.2 Modélisation d'un cylindre

Dans ce code, les cylindres sont modélisés à l'aide de trois variables :

- ◎ *Deux points A et B* Ces points sont les deux points extrêmes du cylindre. Nous entendons par là les points situés au centre des deux disques fermant le cylindre de part et d'autre. Ainsi, la hauteur du cylindre n'est pas un paramètre pour le modéliser, mais peut se retrouver en calculant la longueur  $AB$ .
- ◎ *Un double R* Il s'agit du rayon du cylindre.

Les notations utilisées sont rappelées à la Figure 29.

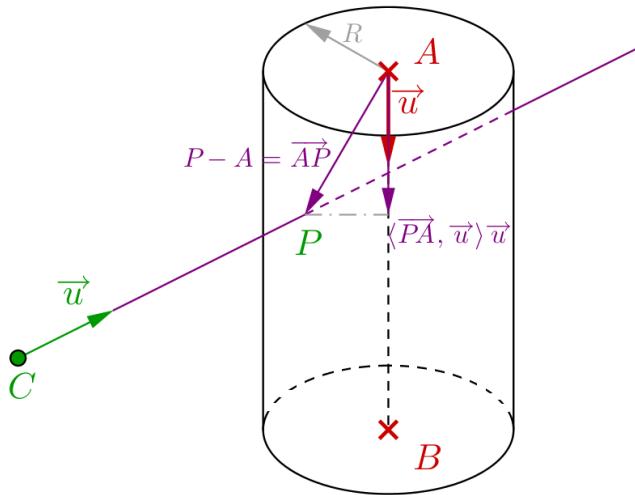


FIGURE 29 • Grandeur permettant de définir un cylindre

Cette figure présente aussi un certain nombre de notations utilisées pour déterminer si un point est dans le cylindre ou non.

### 8.9.3 Savoir si un point est dans un cylindre

Pour savoir si un point est dans un cylindre, la fonction `isInside` se concentre sur trois tests :

- ◎ *Éloignement à l'axe* Pour ce dernier, on projette le point à tester ( $P$ ), dans le disque de centre  $A$  est orthogonal à l'axe du cylindre (c'est un des disques qui ferme le cylindre). Nous regardons alors simplement si la distance entre le projeté et  $A$  est inférieur au rayon, si c'est le cas, le point est à « bonne distance » de l'axe du cylindre.
- ◎ *Dépassement de chaque côté* Pour cela, on considère le vecteur  $\vec{u} = \vec{AB}$ . On regarde alors les signes des produits scalaires  $\langle \vec{u}, \vec{AP} \rangle$  et  $\langle \vec{u}, \vec{BP} \rangle$  pour savoir si  $P$  est n'est pas situé après l'un des points terminaux du cylindre.

Ainsi, la condition globale est (en notant  $P_A$  le projeté mentionné dans le premier test) :

$$AP' < R \quad \text{et} \quad \langle \vec{u}, \vec{AP'} \rangle > 0 \quad \text{et} \quad \langle \vec{u}, \vec{BP'} \rangle < 0$$

Le fait que le signe demandé pour les produits scalaires soit différent vient de l'orientation du vecteur  $\vec{u}$ , si nous avions pris le vecteur  $\vec{BA}$  à la place, les signes seraient inversés.

Ces conditions s'expriment directement sous forme de code :

```

1 Vector u = (B-A).getNormalize();
2 // Tests dans l'ordre 1. && 2. && 3.
3 return ((P-A-(P-A).dot(u)*u).squaredNorm()<(R*R)) && ((P-A).dot(u)>0) && ((P-
4 B).dot(u) < 0);
}
```

#### 8.9.4 Trouver les points d'intersection avec le cylindre

Pour les points d'intersection du cylindre, nous procédons en trois temps :

- Intersection avec le disque de centre  $A$ .
- Intersection avec le disque de centre  $B$ .
- Intersection avec « le corps » du cylindre.

Nous allons détailler tour à tour ces différentes étapes. Nous utiliserons pour cela deux grandeurs génériques :

```

1 const Vector normale = (B-A).getNormalize();
2 const double ndotr = normale.dot(r.u);
3 Material m = this->material;
```

Le vecteur  $normal$  correspond à un vecteur orienté le long de l'axe du cylindre (pour mémoire  $A$  et  $B$  sont les deux points extrêmes de cet axe). La grandeur  $ndotr$  correspond au produit scalaire entre le vecteur directeur de l'axe et la direction du rayon. Enfin,  $m$  représente le matériau du cylindre.

**Intersection avec le disque de centre  $A$**  L'algorithme utilisé ressemble fortement à celui pour le cas du plan, auquel on ajoute un terme pour vérifier que l'on est à distance raisonnable de  $A$  :

- 1. On calcule la valeur de  $t$  pour laquelle le rayon va intersecter le plan contenant le disque de centre  $A$ .
- 2. On teste si le point obtenu est à une distance inférieure au rayon du cylindre de  $A$  et s'il est bien devant le point de départ du rayon.
- 3. Si le point rempli toutes les conditions précédentes, on crée la normale en ce point (qui est de direction opposée à  $normale$  qui était elle dirigée vers l'intérieur du cylindre) puis on insère ce point dans la liste de points d'intersection.

Du point de vue du code, on retrouve une structure similaire au cas d'un simple plan :

```

1 // Demi-plan passant par A. On calcule le point d'intersection avec le plan et on
2 // vérifie :
3 //     1. Que le point est au plus à une distance R de A.
4 //     2. Que le point est bien devant le départ du rayon (t>0).
5 double tA = normale.dot(A-r.C)/ndotr;
6 Vector PA = r.C+tA*r.u;
7 if ((PA-A-(PA-A).dot(normale)*normale).squaredNorm()<R*R && tA > 0) {
8     // Toutes les conditions sont réunies, on crée la normale et on ajoute l'
9     // intersection.
10    Vector N = (-1)*normale;
11    LI.push_back(new IntersectionPointCSG(PA, N, m));
12 }
```

**Intersection avec le disque de centre  $B$**  Pour ce disque, on procède comme avec le disque de centre  $A$  (« remplacer les  $A$  par des  $B$  dans le code précédent »). La seule différence vient de la normale, qui est ici dans le même sens que *normale*, puisque ce vecteur était orienté de  $A$  vers  $B$  et va donc dans le sens qui sort du cylindre en  $B$ .

**Intersection avec le corps du cylindre** Cette partie est celle demandant le plus de calculs. On peut aisément trouver sur Internet des relations donnant le calcul de l'intersection d'un cylindre axé autour d'un axe, mettons  $x$  (les calculs présentés proviennent initiallement de :

[http://heigeas.free.fr/laureray\\_tracing/cylindre.html](http://heigeas.free.fr/laureray_tracing/cylindre.html)

On rappelle que le rayon est donné par la relation (1), soit l'ensemble  $\{P/P = C + t\vec{u}, t > 0\}$ . L'équation d'un cylindre infini d'axe  $x$  est quant à elle donnée par (17).

$$y^2 + z^2 = R^2 \quad (17)$$

Ainsi, l'intersection entre un rayon et ce cylindre infini est donné en injectant (1) dans (17).

$$P_y^2 + P_z^2 - R^2 = 0$$

Or  $P_y = C_y + tu_y$  et  $P_z = C_z + tu_z$ , d'où en développant les expressions :

$$\begin{aligned} (C_y + tu_y)^2 + (C_z + tu_z)^2 - R^2 = 0 &\iff C_y^2 + 2C_y u_y t + u_y^2 t^2 + C_z^2 + 2C_z u_z t + u_z^2 t^2 - R^2 = 0 \\ &\iff (u_y^2 + u_z^2) t^2 + 2(C_y u_y + C_z u_z) t + (C_y^2 + C_z^2 - R^2) = 0 \end{aligned}$$

Nous obtenons donc une équation de degré deux, que l'on sait résoudre sans problèmes. Cependant, nous avions vu que nous cherchions à modéliser un cylindre quelconque, ie. qui ne soit pas nécessairement centré sur l'axe  $x$ . Il nous faut donc légèrement réécrire ces coefficients pour notre cas.

Nous avons vu que nous disposions d'un vecteur  $\vec{n} = \frac{\overrightarrow{AB}}{\|\overrightarrow{AB}\|_2}$ . Ainsi, notre cylindre sera centré sur un axe porté par ce vecteur  $\vec{n}$ . Pour adapter les coefficients, nous remarquons les points suivants :

- Ⓐ  $u_y^2 + u_z^2$  correspond à la norme de  $\vec{u}$  sans sa composante selon l'axe du cylindre. Comme le vecteur  $\vec{u}$  est unitaire, nous pouvons utiliser le fait que  $u_y^2 + u_z^2 = 1 - u_x^2$  (valable pour tout système de coordonnées).
- Ⓐ  $C_y^2 + C_z^2$  correspond à la norme de  $\overrightarrow{OC}$  sans sa composante selon l'axe du cylindre.
- Ⓐ  $C_y u_y + C_z u_z$  correspond au produit scalaire entre  $\overrightarrow{OC}$  et  $\vec{u}$  sans la composante selon l'axe du cylindre de ces vecteurs.

Ainsi, pour adapter les coefficients de l'équation, il nous suffit de retirer la composante de  $\vec{u}$  et  $C$  selon l'axe du cylindre, ie. selon  $\vec{n}$ . En d'autres termes, nous pouvons réécrire l'équation sous la forme (en considérant donc un repère dont  $\vec{n} = \overrightarrow{AB}$  est un axe) :

$$\left(1 - \left\langle \vec{u}, \overrightarrow{AB} \right\rangle^2\right) t^2 + 2 \left\langle C - \left\langle \overrightarrow{C}, \overrightarrow{AB} \right\rangle \overrightarrow{AB}, \vec{u} - \left\langle \vec{u}, \overrightarrow{AB} \right\rangle \overrightarrow{AB} \right\rangle t + \left(\left\| C - \left\langle \overrightarrow{C}, \overrightarrow{AB} \right\rangle \overrightarrow{AB} \right\|_2^2 - R^2\right) = 0$$

Une fois ce polynôme explicité, le reste de l'algorithme est simple :

- Ⓐ 1. Calculer les coefficients du polynôme et son discriminant.
- Ⓐ 2. Si le discriminant est positif, on aura des racines, sinon on s'arrête.
- Ⓐ 3. Pour chacune des racines, on regarde les points suivants :
  - ▷ Que la valeur de  $t$  est positive, ie. que l'on est du bon côté du rayon.
  - ▷ On vérifie que le point associé est bien situé « entre  $A$  et  $B$  ». Ceci peut se vérifier en regardant les signes des produits scalaires  $\langle \overrightarrow{AP}, \vec{n} \rangle$  et  $\langle \overrightarrow{BP}, \vec{n} \rangle$ .

Si ces conditions sont remplies, nous allons créer un point d'intersection, sinon on s'arrête pour cette racine.

- ◎ 4. Si on crée un point d'intersection, il faut déterminer sa normale. Comme on sait que l'on est sur le corps du cylindre et non sur les disques aux extrémités, il nous suffit de créer un vecteur partant du projeté orthogonal de  $P$  sur l'axe du cylindre et dirigé vers l'extérieur. Pour cela, il suffit de retirer la composante selon  $n = AB$  du vecteur  $AP$  :

$$\vec{nP} = \vec{AP} - \langle \vec{AP}, \vec{AB} \rangle \vec{AB}$$

Du point de vue du code, on retrouve ces différents éléments :

```

1 // La mise en équation conduit à une équation de degré deux avec pour coeff...
2 Vector deltaC = r.C-A;
3 double a = 1-ndotr*ndotr;
4 double dcdotn = normale.dot(deltaC);
5 double b = 2*(r.u-ndotr*normale).dot(deltaC-dcdotn*normale);
6 double c = (deltaC-dcdotn*normale).squaredNorm()-R*R;
7 double delta = b*b-4*a*c;
8
9 // On résout notre équation du second degré
10 if (delta > 0) {
11     double tc1 = (-b-sqrt(delta))/(2*a);           // La 1e solution de l'équation
12     Vector P1 = r.C+tc1*r.u;                      // Le point d'intersection
13     calculé
14     // On vérifie les conditions sur le point d'intersection
15     if (tc1 > 0 && (P1-A).dot(normale) > 0 && (P1-B).dot(normale) < 0) {
16         Vector N = (P1-A-(P1-A).dot(normale)*normale).getNormalize();
17         LI.push_back(new IntersectionPointCSG(P1,N,m));
18     }
19     // Même principe qu'avec la première solution.
20     double tc2 = (-b+sqrt(delta))/(2*a);
21     Vector P2 = r.C+tc2*r.u;
22     if (tc2 > 0 && (P2-A).dot(normale) > 0 && (P2-B).dot(normale) < 0) {
23         Vector N = (P2-A-(P2-A).dot(normale)*normale).getNormalize();
24         LI.push_back(new IntersectionPointCSG(P2,N,m));
25     }
26 }
```

## 8.10 | Travail avec des tores

### 8.10.1 Généralités

Comme pour les cylindres, nous avons plusieurs possibilité, qui étaient soit de nous concentrer sur des tores centrés autour des axes, soit de considérer le cas général. Comme pour les cylindres, le choix est ici de considérer le cas général, en adaptant des relations trouvées dans des cas particuliers.

### 8.10.2 Modélisation d'un tore

Pour modéliser un tore, il nous faut utiliser quatre grandeurs :

- ◎  $C$  Le centre du tore, i.e. le point autour duquel est défini le cercle sur lequel va se développer le tore.
- ◎  $\vec{u}$  La direction du tore, qui va permettre de définir dans quel plan le cercle sur lequel va se développer le tore se trouvera. Ainsi, le point  $C$  et le vecteur  $\vec{u}$  vont permettre de définir le plan médian du tore.

- $R$  Le grand rayon du tore, qui correspond au rayon entre  $C$  et les centres des cercles de la couronne.
- $r$  Le rayon des cercles générés à partir des points du cercle de centre  $C$  et de rayon  $R$ , qui seront générés dans un plan perpendiculaire au plan médian du tore.

Les notations utilisées sont rappelées à la Figure 30.

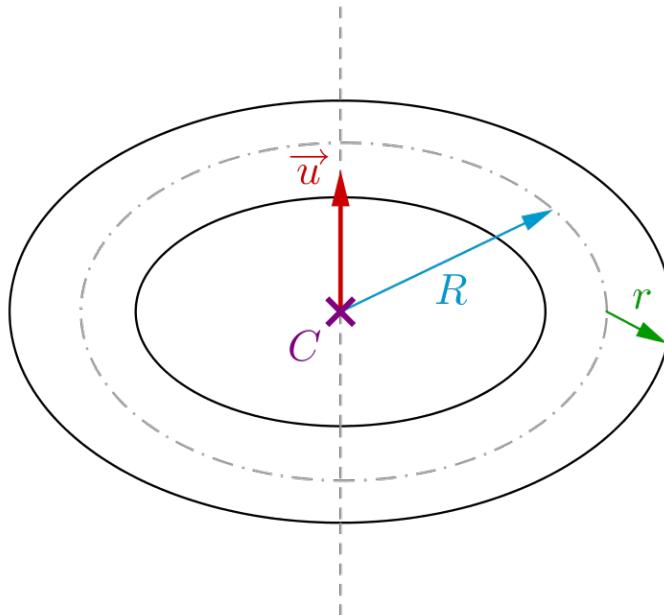


FIGURE 30 • Grandeurs permettant de définir un torus

### 8.10.3 Savoir si un point est dans un tore

Pour cette première action, réalisée par la méthode `isInside`, il suffit de savoir si le point est dans la couronne. Nous procédons pour cela en deux temps :

- *Projection sur le cercle de centre  $C$  et de rayon  $R$*  On commence par ramener le point  $P$  dans le plan médian du tore. Pour cela, on retire la composante selon l'axe  $\vec{u}$  du tore du vecteur  $\vec{CP}$ , à savoir :

$$\vec{CP} - \langle \vec{CP}, \vec{u} \rangle \vec{u}$$

À cette étape, nous avons le projeté dans le plan médian. Pour nous ramener au cercle demandé, on normalise ce résultat que l'on multiplie ensuite par le grand rayon du tore, d'où le point cherché :

$$C + \frac{\vec{CP} - \langle \vec{CP}, \vec{u} \rangle \vec{u}}{\|\vec{CP} - \langle \vec{CP}, \vec{u} \rangle \vec{u}\|} \times R$$

- *Position de  $P$  par rapport au projeté* Pour savoir si  $P$  est à bonne distance de son projeté sur le grand cercle du tore, il suffit juste de regarder si la distance entre  $P$  et ce projeté est inférieure au petit rayon  $r$  du tore.

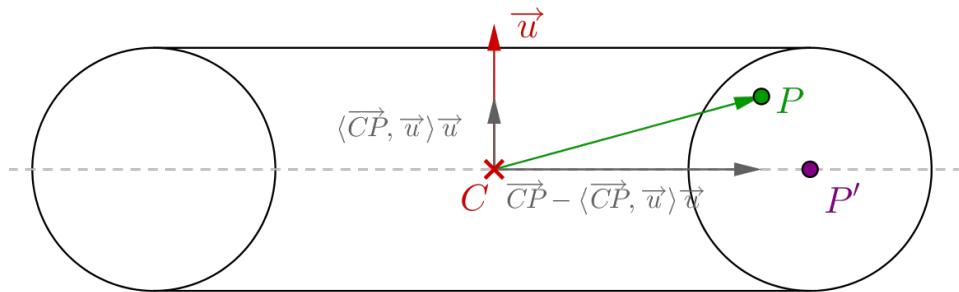
Une vision plus graphique des calculs vous est proposée en Figure 31.

Le code en découle directement :

```

1 bool Torus::isInside(const Vector &P) const{
2     // On projete P sur le cercle de centre C et de rayon R.
3     Vector deltaP = P-C;
4     Vector projection = C+(deltaP-deltaP.dot(u)*u).getNormalize()*R;
5     // Si la distance de P à sa projection est < r^2 alors P est dans le tore.

```



$$P' = C + \frac{\overrightarrow{CP} - \langle \overrightarrow{CP}, \vec{u} \rangle \vec{u}}{\|\overrightarrow{CP} - \langle \overrightarrow{CP}, \vec{u} \rangle \vec{u}\|} \times R$$

**FIGURE 31** • Schéma pour illustrer la manière de déterminer si un point est dans un tore

```

6   // Sinon P est trop loin pour être dans ce tore.
7   double distToCircle2 = (P-projection).squaredNorm();
8   return distToCircle2 < pow(r,2);
9 }
```

#### 8.10.4 Trouver les points d'intersection avec un tore

Le problème est ici que le tore est une surface de quatrième ordre, nous ne pourrons donc pas nous ramener à une solution aussi simple qu'une équation de degré deux. La modélisation étant plus complexe, la stratégie a ici été de regarder sur Internet les solutions existantes. La suite de ce travail a ainsi utilisé les résultats proposés à la page suivante :

[http://www.cosinekitty.com/raytrace/chapter13\\_torus.html](http://www.cosinekitty.com/raytrace/chapter13_torus.html)

Les principaux résultats utilisés sont alors les suivants avec nos notations usuelles (pour mémoire,  $E$  représente  $\vec{u}$  dans un repère local au solide et  $D$  représente  $P$  dans ce même repère). Le rayon est donné par :

$$\begin{cases} x = E_0 t + D_0 \\ y = E_1 t + D_1 \\ z = E_2 t + D_2 \end{cases}$$

En déroulant les calculs d'intersection, on aboutit à l'équation suivante :

$$(Jt^2 + Kt + L)^2 = Gt^2 + Ht + I \quad \text{où} \quad \begin{cases} G = 4R^2(E_0^2 + E_1^2) \\ H = 8R^2(D_0E_0 + D_1E_1) \\ I = 4R^2(D_0^2 + D_1^2) \\ J = E_0^2 + E_1^2 + E_2^2 = \|E\|_2^2 \\ K = 2(D_0E_0 + D_1E_1 + D_2E_2) = 2\langle \vec{D}, \vec{E} \rangle \\ L = D_0^2 + D_1^2 + D_2^2 + R^2 - r^2 = \|D\|_2^2 + R^2 - r^2 \end{cases}$$

On peut alors développer cette équation, et on aboutit à l'équation de degré quatre suivante :

$$J^2t^4 + 2JKt^3 + (2JL + K^2 - G)t^2 + (2KL - H)t + (L^2 - I) = 0$$

Pour la résoudre, nous utiliserons également un solveur de degré quatre trouvé sur Internet à l'adresse suivante :

<https://github.com/dxli/quarticSolver/blob/master/quarticSolver.cpp>

Le code n'est pas détaillé (car pas étudié en détails), nous faisons ici confiance à l'auteur. On remarquera cependant qu'il demande que le coefficient devant  $t^4$  soit unitaire, il faut donc diviser notre précédente équation par  $J^2$  pour être au bon format.

Avec ces différents éléments, l'algorithme devient naturel :

- ➊ 1. Se ramener dans un repère ayant le vecteur  $\vec{u}$  comme axe. Pour cela, on pourra utiliser notre routine générant un repère orthonormé à partir d'un axe donné. Une fois dans ce repère, on calcule  $E$  et  $D$ , les représentations de  $\vec{u}$  et  $C$  dans ce nouveau repère.
- ➋ 2. On calcule les coefficients précédemment proposés, et on met en forme l'équation pour l'envoyer dans le solveur.
- ➌ 3. Pour chaque solution, on l'ajoute dans l'ensemble des points d'intersection calculés, et on lui associe une normale. Cette dernière est simplement calculée en reprenant le projeté de  $P$  sur le cercle de rayon  $R$  (voir la partie précédente sur la fonction `isInside`) que l'on note  $P'$ , et en prenant alors la version normalisée  $\vec{n} = \frac{\vec{PP'}}{\|\vec{PP'}\|}$ .

Nous ne fournissons pas tout le code ici (l'expression des coefficients n'est pas très intéressantes), mais une structure générale pour ce dernier :

```

1 void Torus::intersect(const Ray & r, std::vector<IntersectionPointCSG *> & LI)
2 {
3     // On change le repère du rayon pour l'exprimer dans un repère où le tore est
4     // centré autour de l'axe vertical.
5     // Dans ce nouveau repère, l'axe vertical est u et les deux autres axes T1 et
6     // T2.
7     Vector T1, T2;
8     this->u.orthogonalSystem(T1, T2);
9     Vector E(r.u.dot(T1), r.u.dot(T2), r.u.dot(this->u)); // 
10    // Coordonnées de u dans le nouveau repère.
11    Vector D((r.C-C).dot(T1), (r.C-C).dot(T2), (r.C-C).dot(this->u)); // 
12    // Coordonnées de C dans le nouveau repère.
13
14    // Calcul des coefficients de l'équation d'intersection.
15    ...
16
17    // L'équation est de degré quatre, on utilise un solveur numérique.
18    unsigned int nroots = quarticSolver(coef, roots);
19
20    Material m = this->material;
21
22    // Pour chaque solution, on ajoute cette dernière la liste des points d'
23    // intersection déjà connus.
24    for (unsigned int i = 0; i <nroots; ++i) {
25        // Pour mémoire, on veut être en face du point de départ du rayon ie. t>0.
26        if (roots[i] > 0) {
27            Vector P = r.C+roots[i]*r.u; // 
28            // Coordonnées dans le repère général.
29            Vector Pprime = C+(P-C-(P-C).dot(u)*u).getNormalize()*R; // 
30            // Projété de P sur le cercle (cf. intersect).
31            Vector N = (P-Pprime).getNormalize(); // 
32            // Vecteur normal au point d'intersection donné par un "vecteur rayon".
33
34            // L'ajout à proprement parler
35            IntersectionPointCSG* IP = new IntersectionPointCSG(P,N,m);
36            LI.push_back(IP);
37        }
38    }

```

29      }  
30    }

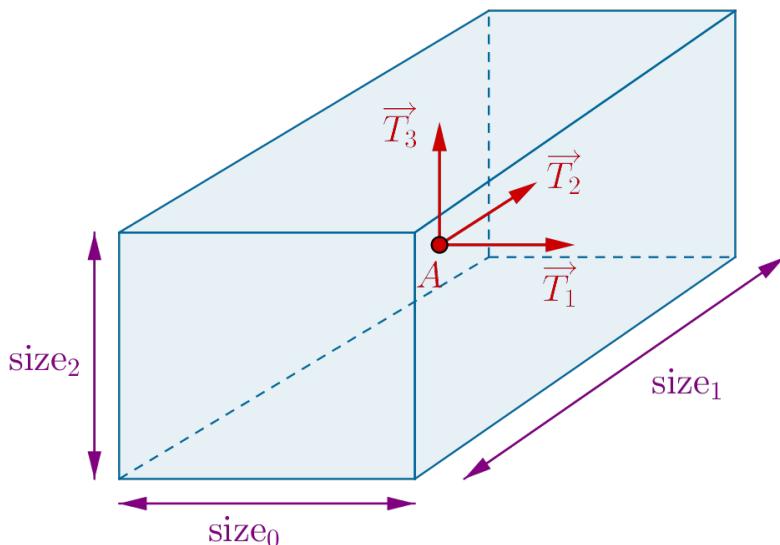
## 8.11 | Travail avec des boîtes

### 8.11.1 Généralités

Comme pour les cylindres et les tores, il existe une forme « simple » des boîtes qui serait de les considérer parallèles aux axes. Encore une fois, la stratégie ici est de considérer des boîtes dans le cas général, et de ce ramener à ce cas particulier par changement de repère le temps de faire les calculs utiles.

Nous verrons également qu'il existait plusieurs solutions pour représenter ces dernières, et que nous avons essayé de nous arrêter sur celle qui nous semblait la plus simple du point de vue des calculs.

### 8.11.2 Modélisation d'une boîte



**FIGURE 32** • Grandeur permettant de définir une boîte

**Les grandeurs nécessaires** Pour modéliser la boîte, deux grandes solutions sont possibles :

- **Deux points** Cette première version consiste à modéliser la boîte par ces deux points extrêmes. Si cette solution fonctionne bien pour des boîtes alignées sur les axes, elle est plus complexe dans le cas de boîtes dans le cas général et a donc été écartée pour ce code.
- **Un point, des vecteurs et des tailles** Cette version est un peu plus lourde pour l'utilisateur, mais plus simple à traiter dans les calculs et est celle retenue. L'idée est alors de d'utiliser quatre grandeurs :
  - ▷ **C** Un point qui servira d'origine pour le repère local de la boîte.
  - ▷  $\vec{T}_1$  La première direction du repère local de la boîte.
  - ▷  $\vec{T}_2$  La seconde direction du repère local de la boîte. La troisième direction sera déterminée en joignant à  $\vec{T}_1$ .
  - ▷ **Vecteur de tailles** Les dimensions selon les trois axes du repère de la boîte.

Dans le détail, il existe plusieurs solutions pour déterminer le point d'origine  $C$ . La première serait de considérer que  $C$  est un des angles de la boîte, et que le repère serait établi le long des arêtes (ainsi par exemple  $\vec{T}_1$  serait le long d'une arête). L'autre solution est de considérer que  $C$  sera situé au centre de la boîte.

Nous avons retenu la solution de  $C$  au centre de la boîte, dans la mesure où ceci simplifie les calculs pour l'appartenance comme nous le verrons juste après.

**Générer la boîte** Rien ne garantit que l'utilisateur va fournir deux directions  $\vec{T}_1$  et  $\vec{T}_2$  qui soient orthogonales. Ainsi, il nous faudra redresser  $\vec{T}_2$  pour que nous ayons bien  $\langle \vec{T}_1, \vec{T}_2 \rangle = 0$ . Une fois ce redressement réalisé, on peut simplement définir la troisième direction de la boîte par produit vectoriel :  $\vec{T}_3 = \langle \vec{T}_1, \vec{T}_2 \rangle$ . Nous obtenons alors bien un système orthonormal (nous normons également les vecteurs).

On se retrouve alors avec la situation de la Figure 32.

Dans la mesure où l'initialisation est un peu plus fournie que pour les autres classes, nous fournissons ci-dessous le code pour le constructeur de notre classe *Box* :

```

1 Box::Box(Vector C, Vector T1, Vector T2, Vector size, Material material) : Object
2   (material){
3     this->C = C;                                     // Origine du repère
4     local.
5     this->T1 = T1.getNormalize();                   // Normalisation de la
6     première direction.
7     this->T2 = (T2-T2.dot(T1)*T1).getNormalize(); // On retire la
8     composante selon T1 de T2 et on normalise cette seconde direction.
9     this->T3 = T1.cross(T2);                      // La troisième direction
10    s'obtient par produit vectoriel.
11    this->size = size;                            // Dimensions de la boîte
12    .
13  }
```

Nous pouvons désormais nous intéresser aux fonctions utiles pour la CSG.

### 8.11.3 Savoir si un point est dans une boîte

Dans la mesure où le point  $C$  est immédiatement au centre de la boîte, déterminer si un point  $P$  est dans cette dernière revient simplement à faire un test sur la norme un du vecteur  $\vec{CP}$  :

$$\|CP\|_1 < \frac{1}{2} \text{Vecteur de tailles}$$

Dans le fait, les imprécisions peuvent ici entraîner le fait que les points d'intersections que nous calculerons soient un peu décollés de la surface. Ainsi, nous avons dû ajouter un facteur de biais pour prendre en compte ceci, le test réellement utilisé (et qui est ici dans *isAlmostInside*) est donc :

$$\|CP\|_1 < \frac{1}{2} \text{Vecteur de tailles} + \epsilon$$

De plus, cette méthode *isAlmostInside* propose deux tests possibles, selon que l'on utilise le repère local de la boîte (cas le plus courant) ou non (et l'on doit alors projeter  $P$  sur les différents axes du repère local). On obtient ainsi le code suivant :

```

1 bool Box::isAlmostInside(const Vector &P, bool localSystem) const{
2   if(localSystem){
3     return abs(P[0]) < size[0]/2+epsilon && abs(P[1]) < size[1]/2+epsilon &&
4       abs(P[2]) < size[2]/2+epsilon;
5   } else {
6     Vector deltaP = P-C;
7     return abs(deltaP.dot(T1)) < size[0]/2+epsilon && abs(deltaP.dot(T2)) <
8       size[1]/2+epsilon && abs(deltaP.dot(T3)) < size[2]/2+epsilon;
9   }
10 }
```

On remarque ainsi que la boîte est le penchant  $\mathcal{L}_1$  de la sphère, le seul soucis est que les tests sur cette norme sont un peu plus long (et plus ouvert aux erreurs !) que les tests avec la norme  $\mathcal{L}_2$ .

De même, cette méthode illustre l'intérêt du positionnement choisi pour le centre de la boîte. En effet, si ce dernier avait été sur un sommet il aurait fallu écrire manuellement les tests de positivité des composantes, ici l'utilisation de la fonction `abs` simplifie l'écriture du code. Cependant, les performances seront les mêmes, puisque `abs` va dans tous les cas réaliser le test que nous nous économisons de notre côté.

#### 8.11.4 Trouver les points d'intersection avec une boîte

Afin de bien organiser ce code, la stratégie a été de regarder ce qui se faisait déjà sur Internet. Nous avons ainsi trouvé une structure pour une implémentation efficace à l'adresse suivante :

<http://www.cs.utah.edu/~awilliam/box/box.pdf>

**Intersection avec les plans de la boîte** Come d'habitude, le code initial est fourni pour une boîte aligné aux axes. Pour se ramener à ce cas simple, nous ferons comme pour le tore et le cylindre, en réalisant un changement de repère sur  $\vec{u}$  et  $C$  (ici le point de départ du rayon) avant les calculs.

Dans ce code, on retrouve les mêmes formules déjà utilisées pour déterminer l'intersection rayon-plan et rayon-extrémités cylindres, nous ne les redétaillerons donc pas ici. Le seul point que nous détaillerons est la manière de déterminer les points d'entrée et de sortie du rayon dans la boîte. Pour cela, on définit un  $t_{\max}$  et un  $t_{\min}$ , et on regarde composante par composante si on obtient des meilleures valeurs. Pour faire ces tests, on cherche en fait à extraire le plus petit intervalle commun entre les valeurs précédentes et l'intervalle pour la composante testée (et si l'intervalle commun est vide, c'est qu'il n'y a pas d'intersection). Ainsi, les différents tests reviennent à faire le même travail que celui qui est présenté pour les maillages en Figure 47.

À l'issue de cette étape, il est important de voir que l'on a déterminé les points d'intersection avec les plans de la boîte, mais que ces points peuvent potentiellement être hors de la boîte (les plans étant infinis dans les calculs actuels).

**Vérifier que les points sont dans la boîte** Cette partie est assurée par la méthode `addIntersectionPoint`. L'idée est désormais de vérifier si les points potentiels sont situés dans la boîte et de déterminer la normale en ces points. Pour cela, l'algorithme est :

- ① 1. On vérifie si le point est dans la boîte en utilisant `isAlmostInside` (on est dans le repère local de la boîte ici). Si le point n'est pas dans la boîte, on arrête là ; sinon on continue.
- ② 2. Il faut ensuite déterminer sur quelle face a lieu l'intersection. L'idée est ici de calculer la distance entre le point d'intersection et  $C$  (centre de la boîte) pour chaque direction. Le résultat le plus proche des dimensions de la boîte signifiera que l'on est sur cette face.
- ③ 3. Pour la face sur laquelle on est (selon la direction  $i$ ), la normale est donnée par  $\text{signe}(P_i)\vec{T}_i$ . En effet, le repère local permet directement d'avoir l'orientation de la normale en fonction des coordonnées d'un point.

On obtient ainsi un point et sa normale. Avant de les ajouter à la liste finale des points d'intersections, il nous faut refaire un changement de repère pour ramener le point  $P$  dans le repère de la scène. On utilise la formule classique :

$$\overrightarrow{OP} = \overrightarrow{OC} + P_{\text{box},0}\vec{T}_1 + P_{\text{box},1}\vec{T}_2 + P_{\text{box},2}\vec{T}_3$$

Le code associé est alors le suivant, nous avons retiré certaines parties redondantes :

```

1 void Box::addIntersectionPoint(double t, Vector &E, Vector &D, vector<
2     IntersectionPointCSG *> &LI) const{
3     Vector P = D + t*E; // Point d'intersection sur la surface du cube
4     Vector N;           // Normale en ce point d'intersection
5
5     if(this->isAlmostInside(P, true)){
6         double min = numeric_limits<double>::max();
7
7         // Le point d'intersection est sur une face orthogonale à T1.

```

```
9     double distance = abs(size[0]/2 - abs(P[0]));
10    if(distance < min){
11        min = distance;
12        N = (P[0] > 0 ? 1 : -1)*T1;
13    }
14
15    ... (idem pour T2 et T3)
16
17    if(min < epsilon){
18        // Si le point est bien dans la boîte (modulo un epsilon), on crée le
19        // point d'intersection
20        Material m = this->material;
21
22        Vector realP = C + P[0]*T1 + P[1]*T2 + P[2]*T3;
23        IntersectionPointCSG* IP = new IntersectionPointCSG(realP, N, m);
24        LI.push_back(IP);
25    }
26 }
```

## 8.12 | Mise en pratique

Nous avons déjà présenté succinctement des exemples pour les différentes opérations réalisées. Nous allons ici présenter un exemple utilisant un peu tous les concepts vus.

Nous avions vu à la Figure 22 une illustration trouvée sur Wikipédia des possibilités de la CSG. Comme cette figure n'emploie que des éléments que nous avons développé, nous pouvons essayer de la réaliser à notre tour.

À ce compte, l'intersection sphère-boîte a déjà été présentée en Figure 24. Pour la pièce utilisant trois cylindres, cette dernière vous est proposée à la Figure 33.

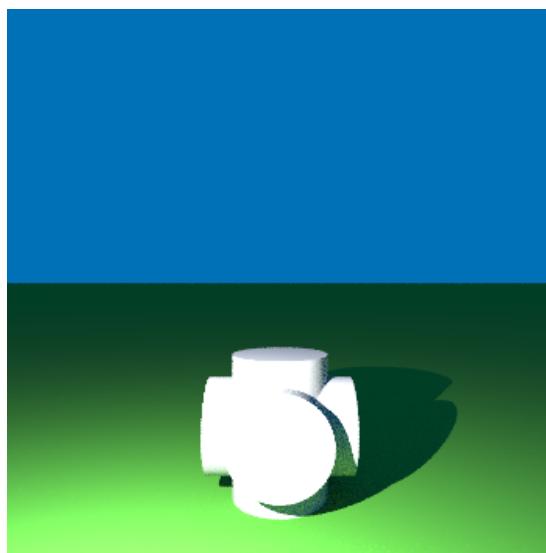
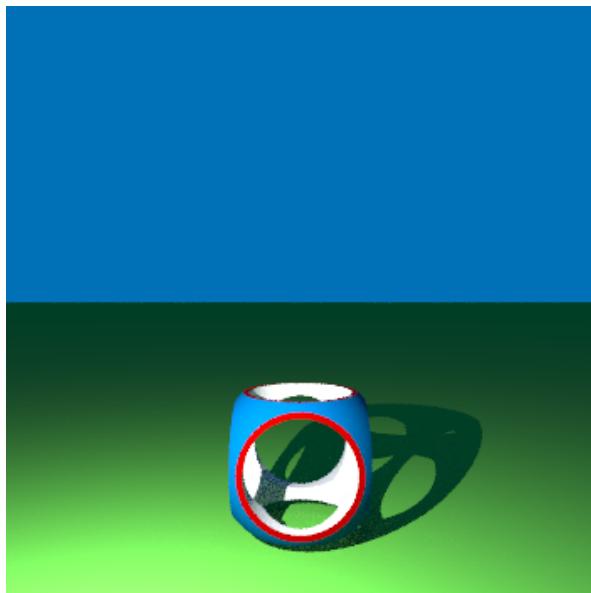


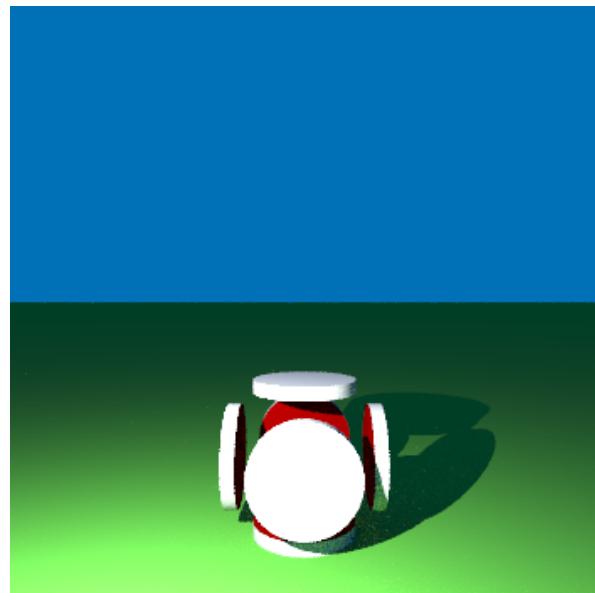
FIGURE 33 • Union de trois cylindres

Il nous reste alors à faire la différence entre le résultat de la Figure 24 et de la Figure 33. Nous obtenons alors bien le résultat attendu, qui est présenté à la Figure 34a. Nous pouvons encore une fois attirer l'attention sur le fait que la soustraction n'est pas commutative, en échangeant l'ordre pour la soustraction finale, et l'on obtient alors le résultat de la Figure 34b.

On remarquera en particulier que nous avons ici gardé les couleurs initiales, à savoir la boîte en rouge, la sphère en bleu et l'union de cylindres en blanc. On voit par exemple que dans la Figure 34b, la sphère n'intervient pas.



(a) Reconstitution de l'image de Wikipédia



(b) Images de Wikipédia en inversant la différence

**FIGURE 34** • Réalisation de l'image test trouvée sur Wikipédia et illustration de l'ordre dans la différence

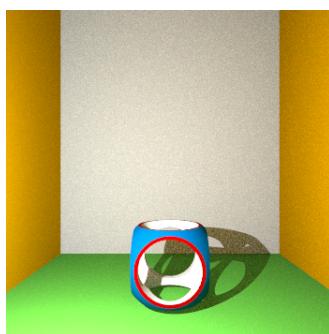
Le code associé pour créer les objets avec nos classes est le suivant :

```

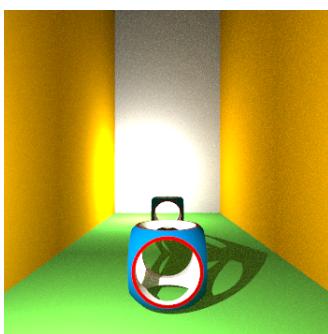
1 // Création de l'union des trois cylindres
2 Cylinder cylinder1(Vector(0, -2, 19), Vector(0, -2, 31), 4, brownDiffuse);
3 Cylinder cylinder2(Vector(0, -8, 25), Vector(0, 4, 25), 4, brownDiffuse);
4 Cylinder cylinder3(Vector(-6, -2, 25), Vector(6, -2, 25), 4, brownDiffuse);
5 Union union1(&cylinder1, &cylinder2, whiteDiffuse, true);
6 Union union2(&union1, &cylinder3, whiteDiffuse, true);
7
8 // Intersection entre la boîte et la sphère
9 Box boxUnion(Vector(0, -2, 25), Vector(0, 0, -1), Vector(0, 1, 0), Vector(1, 1, 1)*10,
   woodSurface);
10 Intersection objectIntersection(&boxUnion, &sphereUnion, whiteDiffuse, false);
11
12 // Différence entre les deux conglomérats d'éléments
13 Subtraction objectSubtraction(&objectIntersection, &union2, whiteDiffuse, false
  );

```

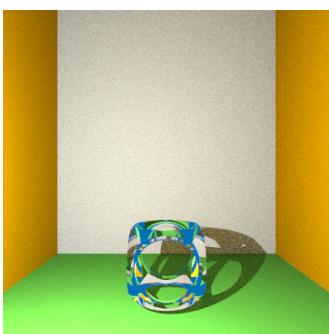
Pour terminer, nous avons réalisé quelques essais en changeant les matériaux, et en utilisant par exemple des miroirs, du verre,...



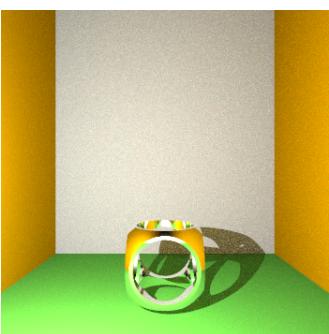
(a) Encadré avec des murs



(b) Miroir en fond de scène



(c) Utilisation de verre



(d) Matériau spéculaire

**FIGURE 35** • Différents essais avec l'objet créé en CSG

On remarque que dans le cas où l'objet CSG est transparent, ce dernier induit un nombre important de réflexion, et qu'une grande partie de ces rayons est perdue, d'où les reflets bleus sur cet objet. De

manière générale, les résultats semblent plutôt convaincant (ici 30 rayons par pixels), mais mettent un peu de temps à calculer sur ma machine.

## 9 • Textures

### 9.1 | Organisation du code pour les textures

#### 9.1.1 Les principales classes

Nous avons choisi de réaliser ici deux types de textures procédurales :

- **Texture géométrique** Une texture en forme de damier selon les plans horizontaux. L'idée était ici de tester les solutions, il aurait pu être possible de tester d'autres motifs, mais nous n'avons pas pris le temps de le faire.
- **Texture avec bruit de Perlin** Pour l'exemple, nous avons réalisé une texture de marbre et une de bois pour voir le fonctionnement de ce type de textures.

Du point de vue des classes, nous avons essayé de ne pas répéter le code au maximum, ainsi nous avons utilisé une classe générique pour toutes les textures nommée *Texture*. Puis, la classe représentant la texture carrelée (*SquaresTexture*) et la classe générique pour le bruit de Perlin (*PerlinTexture*) en ont hérité. Cette classe mère impose seulement que toutes les textures proposent une méthode nommée *applyTexture* qui applique la texture en un point donné (et sa normale au besoin).

Pour les textures utilisant un bruit de Perlin, elles héritent toutes de la classe *PerlinTexture* qui définit ce bruit avec son attribut nommé *perlin* et demande à ses héritières d'implémenter une méthode *applyPerlin* qui utilise le bruit généré.

#### 9.1.2 Modifications génériques

Sans surprise, l'association d'un texture à un objet se fait au niveau du matériau de ce dernier. On ajoute ainsi une nouvelle fois un attribut à la classe *Material* qui sera la texture de ce dernier.

Il s'agit ensuite de savoir comment appeler cette texture au niveau de la fonction calculant la couleur (*getColor*). La solution retenue est de changer l'appel à la couleur de l'objet. Auparavant, cette dernière se faisait en utilisant directement l'attribut de l'objet, par exemple :

```
1 finalColor = finalColor + objectMaterial.diffusionCoeff * objectMaterial.M->color
   * indirect * (1./PI);
```

Désormais, nous avons remplacé cet appel par un appel à une nouvelle méthode de la classe *Material* nommée *computeColor* qui va déterminer si le matériau a une texture ou non, et donc décider de la meilleure solution à renvoyer :

```
1 finalColor = finalColor + objectMaterial.diffusionCoeff * objectMaterial.
   computeColor(P, N) * indirect * (1./PI);
2
3 Vector Material::computeColor(Vector& P, Vector& N){
4     if(this->texture == NULL){
5         return this->color;
6     } else {
7         return this->texture->applyTexture(P, N);
8     }
9 }
```

Ainsi, on obtient un fonctionnement transparent, qui nécessite juste que chaque texture implémente la méthode permettant de l'appliquer en un point donné (ce que nous avions déjà vu).

### 9.2 | Texture carrelée

#### 9.2.1 Configurations possibles

Nous proposons trois paramètres à l'utilisateur :

- *color1 et color2* Les deux couleurs permettant de réaliser le carrelage demandé. Si nous avions fait un carrelage en trois dimension, il aurait fallu ajouter une couleur pour ne pas avoir deux briques de même couleur côté à côté, ce n'était cependant pas l'objectif ici.
- *pas* Le pas du carrelage.

Ces paramètres restent intuitifs, et adaptés au cas en deux dimensions.

### 9.2.2 Appliquer la texture

La texture est appliquée sur les plans parallèle à l'axe ( $Oxz$ ), nous ne travaillons donc que sur deux coordonnées. Le motif s'obtient alors en calculant des coordonnées pour un point dans un repère de résolution *pas* et en regardant si les coordonnées sont toutes les deux paires simultanément.

De plus, autours des axes le résultat sera toujours le même, il est donc nécessaire d'ajouter un terme d'offset pour les valeurs négatives afin d'obtenir un carrelage régulier.

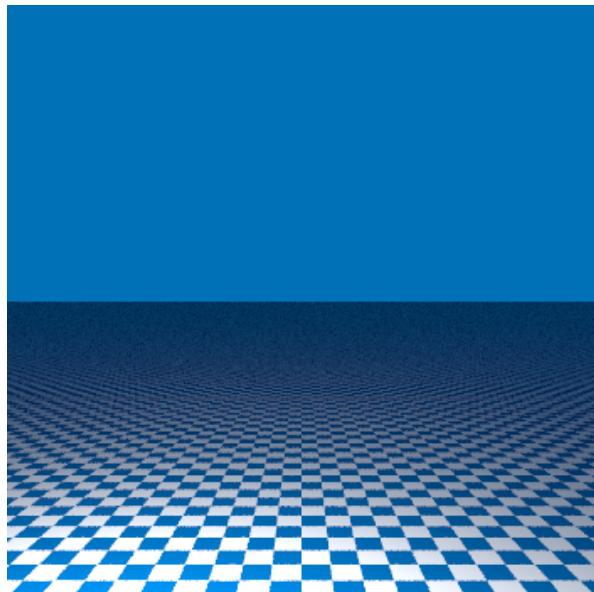
Le code associé est alors le suivant :

```

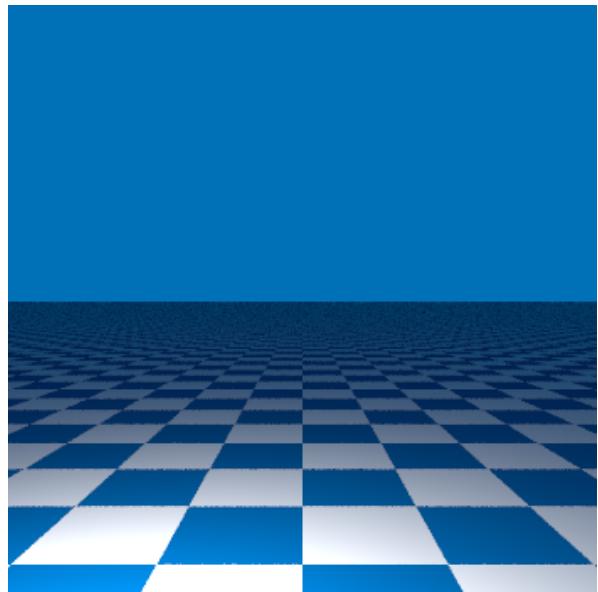
1 Vector SquaresTexture::applyTexture(const Vector& P, const Vector& N) const {
2     int newX = int(P[0])/this->pas - (P[0]<0 ? 1 : 0);
3     int newZ = int(P[2])/this->pas - (P[2]<0 ? 1 : 0);
4
5     return ((newX + newZ)%2) ? this->color1 : this->color2;
6 }
```

### 9.2.3 Résultat obtenu

Pour un plan parallèle à l'axe ( $Oxz$ ), nous obtenons le résultat de la Figure 36, en essayant différents pas possibles pour la grille.



(a) Grille avec un pas de 2



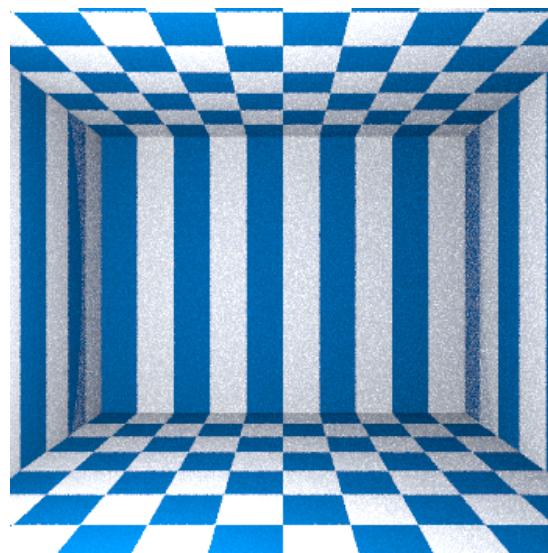
(b) Grille avec un pas de 10

**FIGURE 36** • Textures de carrelage avec différents pas sur un plan parallèle à ( $Oxz$ )

On peut alors regarder ce que donnerait une boîte entière pour la scène, et voir ainsi comme se comporte cette texture pour les autres types de plans, ceci vous est proposé à la Figure 37.

Enfin, regardons la situation en prenant un plan non parallèle aux axes, une sphère ou encore un tore, avec les cas de la Figure 38.

On remarque à travers ces exemples que la texture manque de qualité pour la gestion de la 3D (puisque n'utilise que deux coordonnées et deux couleurs), elle permet cependant d'obtenir des résultats intéressants, bien que non totalement régulier avec des formes plus complexes.



**FIGURE 37** • Scène avec des murs entièrement couverts par cette texture



(a) Avec un plan non parallèle aux axes

(b) Avec une sphère

(c) Avec un tore

**FIGURE 38** • Tests de la texture de carrelage sur différents objets

## 9.3 | Textures utilisant un bruit de Perlin

### 9.3.1 Structure du code

Pour ces textures, nous avons créé une classe représentant le bruit de Perlin nommée *Perlin*. Puis une classe commune à toutes les textures utilisant ce bruit de Perlin, nommée *PerlinTexture*, dont toutes les classes l'utilisant doivent hériter.

Nous allons donc détailler ici les méthodes globales de la classe *Perlin* et leur utilisation pour les textures (classes *MarbleTexture* et *WoodTexture*).

### 9.3.2 La classe Perlin

Le code utilisé provient d'un site ayant adapté la version initiale de Ken PERLIN, situé à l'adresse suivante :

<http://www.massal.net/article/raytrace/page3.html>

Le code proposé sur ce site a cependant été remanié ici pour s'inclure dans notre modèle objet. Nous détaillerons cependant ici les principaux composants de ce code. On rappelle également que ce bruit se répète périodiquement, puisque la fonction de hachage utilise un nombre fini d'éléments.

**Le vecteur  $p$**  Ce dernier est utilisée par fonction de hachage pour déterminer les gradients à choisir, et est défini à partir du tableau de permutations initial.

**La méthode noise** Cette dernière va générer le bruit. Pour cela, elle va ramener le point dont on cherche le bruit dans un cube unitaire sur lequel le bruit est défini. De plus, elle va lisser ces valeurs à l'aide d'une fonction de lissage *fade* dans le code, qui est la fonction suivante :

$$f(t) = 6t^5 - 15t^4 + 10t^3$$

Cette fonction permet d'avoir des transitions plus lisses entre les différentes valeurs et a été proposée initialement par Perlin. Changer cette méthode changerait en partie le résultat du bruit.

Ensuite, cette méthode va utiliser la fonction de hashage pour mapper les valeurs des coordonnées sur celles du bruit. Enfin, les résultats sont interpolés en utilisant un gradient pour obtenir des transitions réalistes entre les différentes valeurs du bruit.

**La méthode fade** La méthode de lissage initial pour la transformation des coordonnées.

**La méthode grad** La méthode pour calculer le gradient en un point avant de réaliser l'interpolation finale.

**La méthode lerp** La méthode pour interpoler les valeurs à la fin de l'algorithme.

### 9.3.3 La classe PerlinTexture

Cette classe générique va utiliser la classe *Perlin* et la mettre à disposition pour nos différentes textures. De plus, cette dernière propose de lui fournir une certain nombre de paramètres :

- **Le bruit** Ce dernier est initialisé et stocké dans l'attribut *perlin*.
- **Des couleurs** Par défaut, nous n'utilisons ici que deux couleurs qui serviront à mapper les niveau de bruit avec une couleur. Par exemple pour le marbre, les niveaux bas seront en blanc et on effectue une interpolation linéaire jusqu'au noir pour les hauts niveaux.
- **Le grain** Ce dernier sert à définir l'importance relative du bruit sur le motif initial. Plus ce dernier est élevé, plus le bruit aura de l'importance dans le motif final.
- **Le nombre d'octaves** Ce dernier sert à définir le nombre de fonctions sinusoïdales qui sont conservées pour calculer le bruit final, et définit donc la précision de la structure.

Ainsi, si la classe *Perlin* défini le hashage aléatoire nécessaire à ce bruit, c'est la classe *PerlinTexture* qui va générer le « vrai » bruit. Pour cela, on va sommer *octave* bruits de fréquences de plus en plus élevées et d'amplitude inversement proportionnelles à cette fréquence. On obtient ainsi un bruit plus naturel, et assez fin.

Le code de la fonction appliquant le bruit est alors :

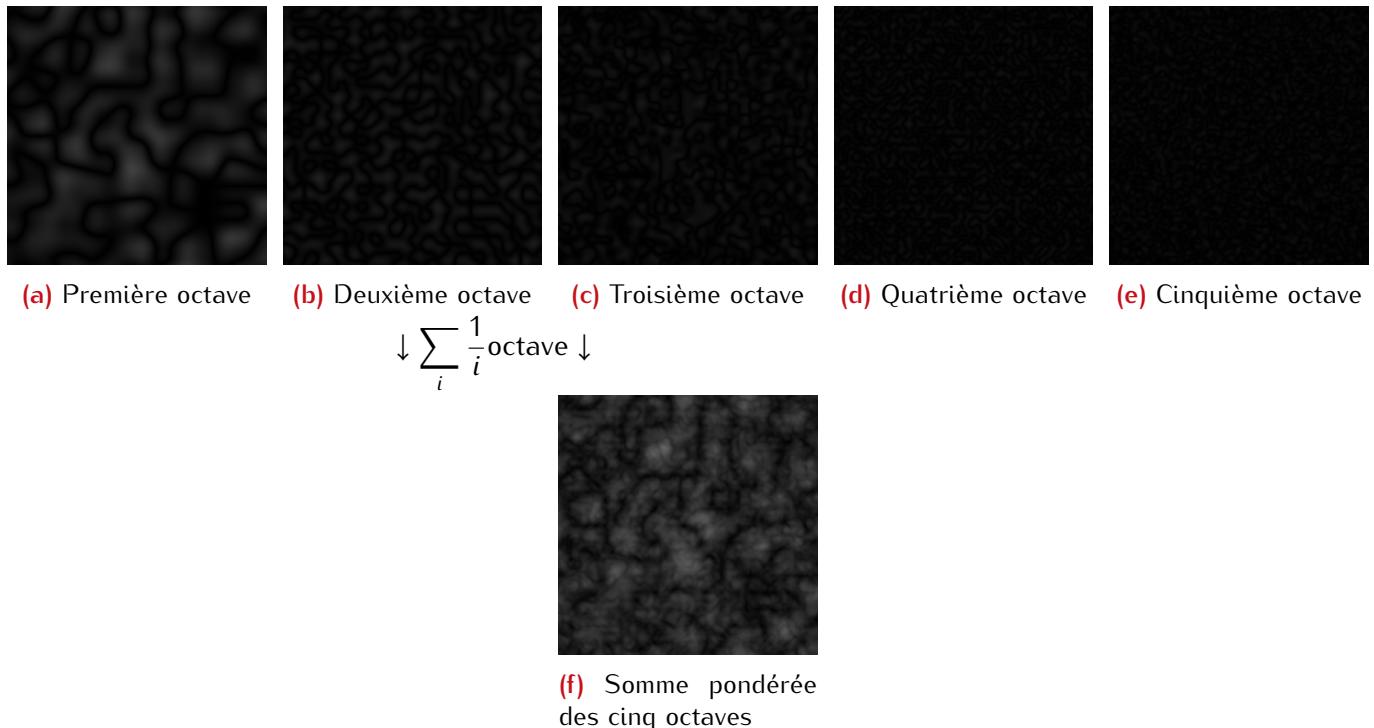
```

1 Vector PerlinTexture::applyTexture(const Vector& P, const Vector& N) const {
2     double noiseCoef;
3     for (int level = 1; level<this->octave; level++) {
4         noiseCoef += (1./level) * fabsf(float(this->perlin->noise(level*.05*P[0],
5             level*.05*P[1], level*.05*P[2])));
6     }
7     return applyPerlin(noiseCoef, P, N);
}

```

On remarque que la boucle principale calcule le bruit en utilisant le hashage proposé par l'attribut *perlin* et en faisant varier sa fréquence par la valeur *level*. De plus, chaque résultat est multiplié par *1./level* pour que les composantes avec les plus hautes fréquences ne soient pas celles qui influencent le plus le résultat final (ce qui ne serait pas réaliste). Pour comprendre ceci, nous avons représenté en Figure 39 le bruit obtenu en sommant cinq octaves, et chacune des octaves séparément.

Enfin, on remarque que l'application du bruit sur une fonction est déléguée à une autre méthode qui sera elle spécifique à chaque texture (*applyPerlin*).



**FIGURE 39** • Représentation des cinq premières octaves pour le bruit de Perlin

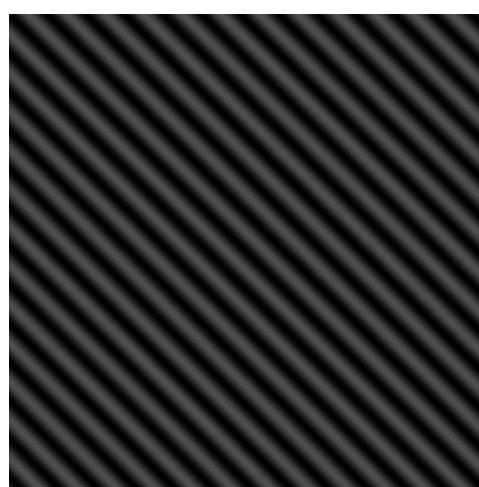
### 9.3.4 La classe *MarbleTexture*

Avec cette classe, nous allons voir la manière d'utiliser le bruit de Perlin pour générer une texture. Dans les faits, chaque texture utilisant un bruit de Perlin va utiliser ce bruit pour bruiter un motif initial et qu'il se rapproche ainsi de la réalité.

Dans le cas du marbre, la fonction à bruiter est la suivante :

$$f(x, y, z) = \sin(x + y + z) + \alpha$$

En prenant  $\alpha = 0,5$  et en mappant les niveaux faibles en blanc et les autres en noir, on obtient le visuel de la Figure 40.

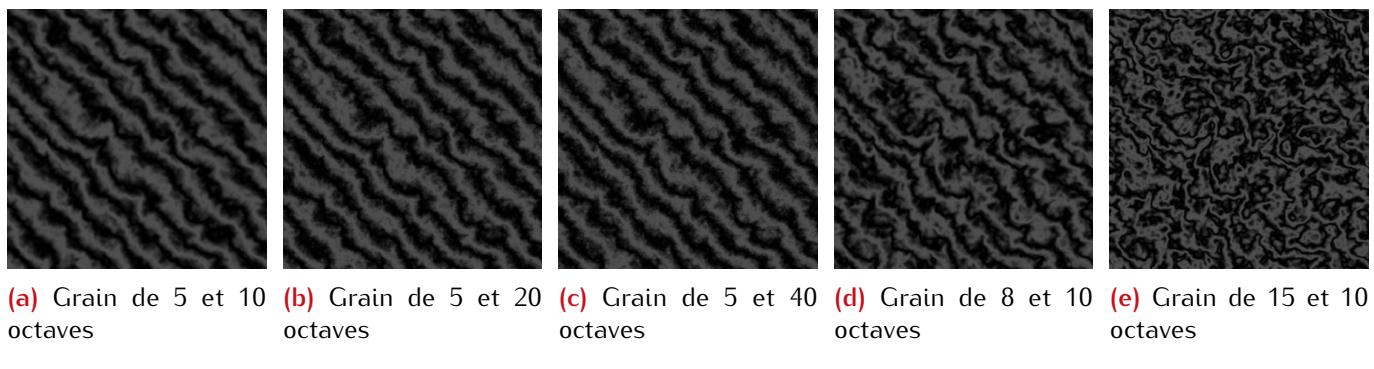


**FIGURE 40** • Fonction à bruiter pour réaliser du marbre

Pour bruiter ce résultat, on utilise alors la valeur aléatoire précédemment calculée, et l'on considère désormais la fonction :

$$f_{\text{bruítée}} = \sin(x + y + z + \text{grain} \times \text{noise}) + \alpha$$

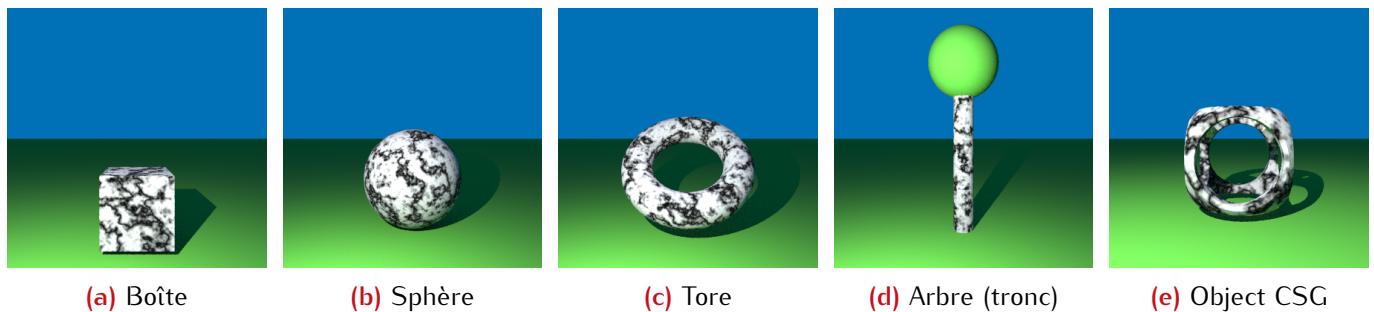
Comme nous l'avions dit, la variable de grain permet de définir l'importance relative du bruit par rapport aux vraies paramètres initiaux. Pour déterminer les valeurs optimales, ceci ne peut se faire que par expérience. Nous fournissons par exemple cinq situations testées à la Figure 41.



**FIGURE 41** • Essais de différents coefficients pour obtenir un effet marbré

Comme on s'y attendait, on remarque que plus le nombre d'octave utilisé est important, plus le résultat est lissé, et des bruits légers apparaissent à côté des marbrures. Ainsi, pour notre application, l'utilisation d'un nombre d'octaves élevé semble intéressant. Pour le choix du gain, tout dépend de l'effet recherché. Si on cherche à avoir un marbre avec des nervures très marquées et assez alignées, on préférera un gain faible. Au contraire, si on désire un marbre « plus aléatoire », on préférera des valeurs plus élevées.

En conclusion, pour le marbre, il semble que prendre des valeurs relativement élevées pour les deux paramètres soit ici intéressant. Nous prendrons par la suite un gain de 15 et 40 octaves. En utilisant ces paramètres, nous pouvons essayer d'appliquer cette texture à divers objets. Des exemples vous sont proposés à la Figure 42.



**FIGURE 42** • Application de la texture de marbre sur différents solides

On remarque que la texture s'applique bien à ces différents objets. Le cas de l'arbre a été gardé ici car la texture du marbre faisait presque penser à l'écorce d'un bouleau...

### 9.3.5 La classe *WoodTexture*

Le fonctionnement est analogue à celle de la classe *MarbleTexture*, cependant nous allons utiliser une autre fonction pour avoir ce rendu. La fonction utilisée ici est la suivante :

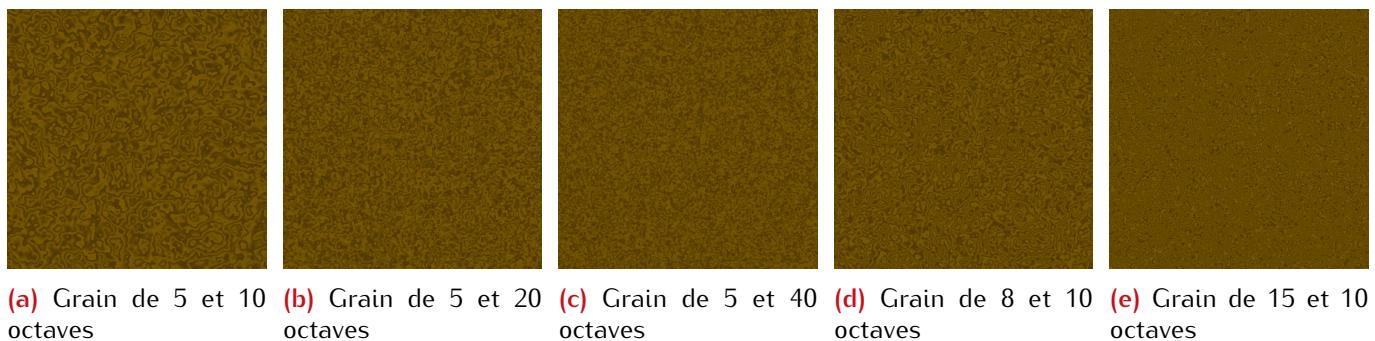
$$f(x, y, z) = \frac{1}{2} (1 - \cos(\pi(\text{bruit}(x, y, z) - E(\text{bruit}(x, y, z)))))$$

On en déduit donc immédiatement le code de la version de *applyPerlin* pour l'effet bois :

```

1 Vector WoodTexture::applyPerlin(double noiseCoef, const Vector& P, const Vector&
2     N) const{
3     double v = grain*noiseCoef;
4     double finalCoef = (1-cos(PI*(v-floor(v))))*.5;
5     return Vector((1-finalCoef)*this->color1 + finalCoef*this->color2);
}
```

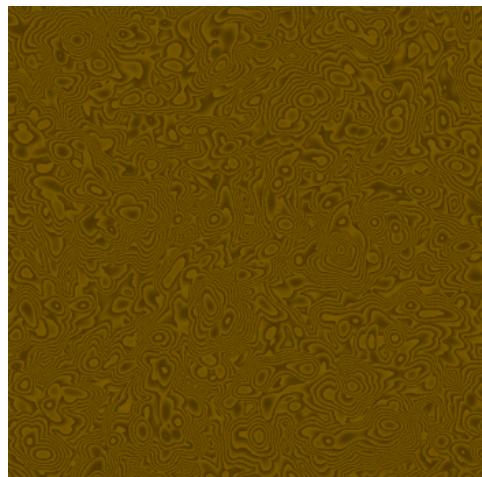
Comme pour l'effet marbre, il nous faut définir de manière empirique les valeurs optimales pour les coefficients. Différents essais sont présentés à la Figure 43.



**FIGURE 43** • Essais de différents coefficients pour obtenir un effet bois

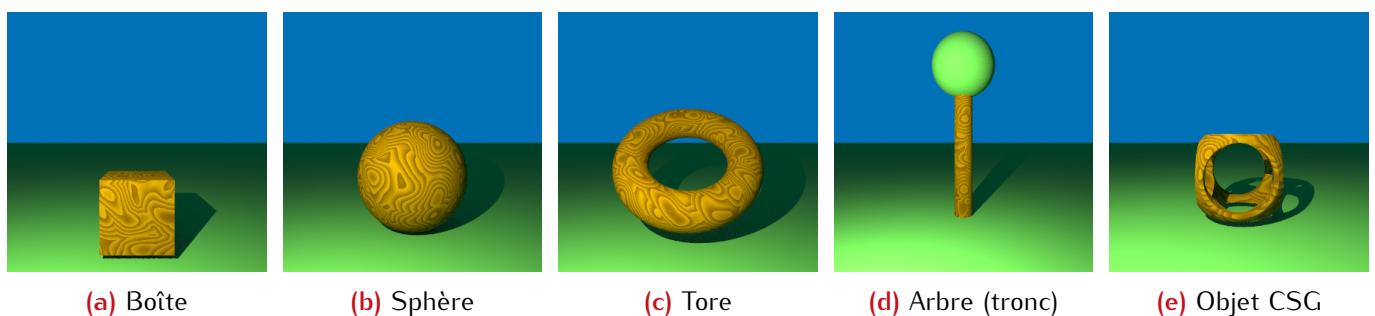
De manière générale sur les essais, nous avons remarqué que prendre un nombre d'octaves trop important donne des cernes trop rapprochés par rapport à la réalité. Nous allons ainsi conserver une valeur restreinte pour le nombre d'octave. Concernant le grain, si ce dernier est un peu plus important on obtient des cernes bien séparés, ce qui donne un effet intéressant.

Par exemple, en prenant 5 octaves et un grain de 15, on obtient un résultat plus satisfaisant pour notre cas, visible à la Figure 44.



**FIGURE 44** • Un bon compromis de valeur pour le bois

Il reste alors à tester cette texture sur différents objets. Les résultats sont proposés à la Figure 45.



**FIGURE 45** • Application de la texture de marbre sur différents solides

Le rendu semble plutôt cohérent avec les attentes. En particulier, on remarque que le grain trouvé s'applique correctement aux différents cas d'usage mis en avant.

### 9.3.6 Combiner les deux

À titre d'exemple, nous fournissons le cas de l'objet CSG avec une texture de bois qui soit disposé sur une surface en marbre à la Figure 46.

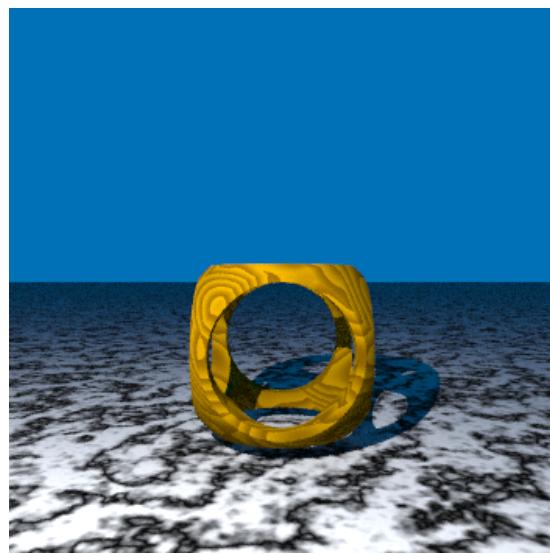


FIGURE 46 • Scène utilisant deux textures définies par bruit de Perlin et de la CSG

## 10 • Les maillages

### 10.1 Mise en place des maillages

#### 10.1.1 Comment définir un maillage ?

Un maillage est une collection de triangles qui est stockée à partir des données suivantes :

- Une liste de sommet, ie. de coordonnées en trois dimensions.
- Une liste qui précise quels sont les points qui composent les triangles.
- De plus, on peut parfois trouver la donnée de la normale face par face.

Cependant, la manière dont sont enregistrées les données dépend du fichier source et est donc à adapter au cas par cas.

#### 10.1.2 Classes utilisées dans le code

Pour notre code, nous avons créé deux classes liées aux maillages :

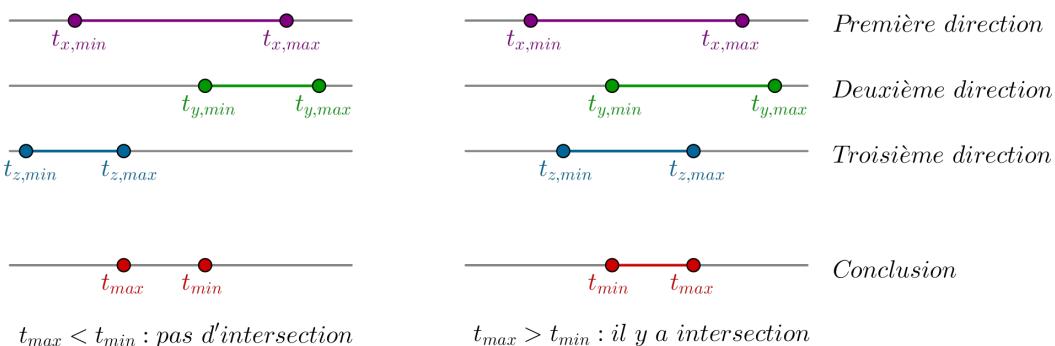
- *Geometry* La classe qui représente le maillage (les sommets, les associations, les normales,...).
- *BoundingBox* La classe qui représente les boîtes englobantes pour les différentes parties du maillage.

Dans le code présenté, le maillage est associé à une unique *BoundingBox*. En effet, nous n'avons considérer que la boîte englobante globale, mais pas la recherche des sous-parties du maillage.

#### 10.1.3 La classe *BoundingBox*

Les *BoundingBox* sont supposés parallèles aux axes. Ainsi, pour les définir nous n'avons besoin que de deux paramètres. Ces paramètres vont représenter les points extrêmes de la boîte. Le repère associé à la boîte sera lui directement le repère usuel.

Comme pour tous les objets, il faudra pouvoir vérifier si un rayon intersecte cette boîte englobante. Étant donné qu'elle est parallèle aux axes, la méthode sera plus simple que dans le cas général vu pour la CSG.



$$\text{Avec } \begin{cases} t_{min} = \max(t_{i,min}, i \in \{x; y; z\}) \\ t_{max} = \min(t_{i,max}, i \in \{x; y; z\}) \end{cases}$$

**FIGURE 47** • Explication de la stratégie pour déterminer s'il y a intersection sur une boîte

Pour chercher cette intersection, il suffit de calculer les  $t_{min}$  et  $t_{max}$  dans chacune des directions du repère pour lequel le rayon va intersecter la boîte englobante. On cherche ensuite à déterminer s'il existe

un intervalle commun aux trois domaines de  $t$  selon les trois axes. La condition à vérifier est alors que :

$$[t_{x,\min}; t_{x,\max}] \cap [t_{y,\min}; t_{y,\max}] \cap [t_{z,\min}; t_{z,\max}] \neq \emptyset$$

Si cette condition est remplie, il y a intersection, sinon il n'y a pas intersection.

Le code vient alors aisément, le calcul de  $t_{\max}$  et  $t_{\min}$  se voit bien sur un dessin (voir Figure 47).

De plus, on remarquera que le test pour voir que l'intervalle commun est vide ou non revient juste à vérifier que ses bornes soient dans le bon sens.

```

1 bool BoundingBox::intersect(const Ray &r) const{
2     double tx1 = (bmin[0] - r.C[0])/r.u[0];
3     double tx2 = (bmax[0] - r.C[0])/r.u[0];
4
5     double ty1 = (bmin[1] - r.C[1])/r.u[1];
6     double ty2 = (bmax[1] - r.C[1])/r.u[1];
7
8     double tz1 = (bmin[2] - r.C[2])/r.u[2];
9     double tz2 = (bmax[2] - r.C[2])/r.u[2];
10
11    double tmax = min(min(max(tx1, tx2), max(ty1, ty2)), max(tz1, tz2));
12    double tmin = max(max(min(tx1, tx2), min(ty1, ty2)), min(tz1, tz2));
13
14    return tmax > tmin;
15 }
```

Cette classe est utilisée par la classe *Geometry* qui sera présentée à part dans la Section précédente de par son usage plus grand.

## 10.2 | Le fonctionnement de la classe *Geometry*

---

### 10.2.1 Les différents attributs

Comme nous l'avons précisé, cette classe va regrouper les différents éléments servant à définir le maillage. Dans le code réalisé, on retrouve ainsi les points suivants :

- *faceGroup* Un vecteur pour représenter des groupes de sommets et savoir quel sommet est dans quel groupe. Ce vecteur est à lire conjointement avec *faces*, *uvIds* et *normalIds*.
- *faces* Un vecteur qui contient les sommets s'associant pour créer les faces. La solution retenue est que les éléments fonctionnent trois par trois. Ainsi *faces[0]*, *faces[1]* et *faces[2]* formeront un triplet qui définit un sommet et ainsi de suite.
- *uvIds* Même principe que pour le vecteur de sommets, mais avec les coordonnées de texture.
- *normalIds* Même principe que pour le vecteur de sommets, mais pour les normales.
- *vertices* Un tableau contenant tous les sommets du maillage.
- *normals* Un tableau contenant toutes les normales du maillage.
- *uvs* Un tableau contenant les coordonnées de texture pour ce maillage.
- *bbox* La boîte englobante du maillage. Cette dernière est calculée à l'initialisation de la classe.

### 10.2.2 Chargement du fichier

Le chargement des données se fait à l'initialisation de l'objet. Pour le cas du fichier *girl.h*, nous utilisons les informations fournies dans les supports de cours afin de savoir comment lire les différentes lignes du fichier.

Ensuite, on remplit les différents attributs présentés à la section précédente. Pour terminer, on calcule la boîte englobante, ce qui est détaillé dans la partie suivante.

### 10.2.3 Construction de la boîte englobante

Par défaut, on prend comme points minimum et maximum des points initialisés avec des valeurs très élevées. Ensuite, on parcours le tableau des sommets (*vertices*) pour mettre à jour les coordonnées de ces points de manière itérative.

Pour terminer, on imprime ces coordonnées à l'utilisateur pour information. Le code est alors immédiat :

```

1 void Geometry::build_bbox(){
2     // On initialise avec des valeurs très élevées.
3     bbox.bmin = Vector(1E9, 1E9, 1E9);
4     bbox.bmax = Vector(-1E9, -1E9, -1E9);
5     double bmin0 = 1E9, bmin1 = 1E9, bmin2 = 1E9, bmax0 = -1E9, bmax1 = -1E9,
6         bmax2 = -1E9;
7
8     // On parcourt tous les sommets et on met à jour les valeurs
9     for (unsigned int i = 0; i < vertices.size(); i++) {
10         bmin0 = std::min(bmin0, vertices[i][0]);
11         bmin1 = std::min(bmin1, vertices[i][1]);
12         bmin2 = std::min(bmin2, vertices[i][2]);
13
14         bmax0 = std::max(bmax0, vertices[i][0]);
15         bmax1 = std::max(bmax1, vertices[i][1]);
16         bmax2 = std::max(bmax2, vertices[i][2]);
17     }
18
19     // Construction des points extrêmes à partir des valeurs calculées.
20     bbox.bmin = Vector(bmin0, bmin1, bmin2);
21     bbox.bmax = Vector(bmax0, bmax1, bmax2);
22
23     // Information de l'utilisateur
24     cout << bbox.bmin[0] << " " << bbox.bmin[1] << " " << bbox.bmin[2] << " "
25         << bbox.bmax[0] << " " << bbox.bmax[1] << " " << bbox.bmax[1] << endl;
}

```

### 10.2.4 Chercher les points d'intersection avec le maillage

**Commencement de la stratégie** Pour cela, on utilise comme précédemment une fonction que l'on nomme *intersect*, qui prend en paramètre un rayon et une liste de points d'intersection à compléter.

La stratégie consiste ici à déterminer si le rayon intercepte la boîte englobante. Si ce n'est pas le cas, on s'arrête là ; sinon, on continue.

Dans le cas où il y a une intersection avec cette boîte, rien ne garantie qu'il y ait réellement une intersection (il y a du vide dans la boîte !). On doit alors chercher avec quel triangle l'intersection a lieu.

**Trouver les intersections par triangle** Pour cela, on parcourt à nouveau tous les triangles du maillage (par le tableau *vertices*). À cette fin, nous avons créé une deuxième méthode *intersect* dans *Geometry*, qui prend en paramètre le rayon, l'identifiant du triangle et ensuite la valeur de la normale et du paramètre *t* à déterminer s'il y a intersection.

L'idée est alors de procéder comme suit :

- Ⓐ 1. On récupère les trois sommets *A*, *B* et *C* du triangle.
- Ⓑ 2. On calcule une normale à la face, qui est donnée par  $\vec{n} = \overrightarrow{AC} \wedge \overrightarrow{AB}$  et que l'on normalise.
- Ⓒ 3. On peut alors réutiliser (une fois de plus) notre formule donnant l'intersection d'un rayon avec un plan. Si le *t* obtenu est négatif, il n'y a pas intersection (elle est derrière la source de lumière), sinon il y a intersection.
- Ⓓ 4. Pour terminer, il faut déterminer si le point d'intersection est dans le triangle ou non. Pour cela, nous allons expliciter le point d'intersection entre le plan et le rayon. Pour mémoire, le rayon est

décrit par  $\{P/C + t\vec{u}, t > 0\}$  et le plan a pour équation paramétrique ( $\alpha$  et  $\beta$  sont des paramètres réels) :

$$\begin{cases} x = \alpha x_{\overrightarrow{AB}} + \beta x_{\overrightarrow{AC}} + x_A \\ y = \alpha y_{\overrightarrow{AB}} + \beta y_{\overrightarrow{AC}} + y_A \\ z = \alpha z_{\overrightarrow{AB}} + \beta z_{\overrightarrow{AC}} + z_A \end{cases}$$

On peut alors remplacer dans l'expression du rayon composante par composante et on aboutit à :

$$\begin{cases} x_C + tx_{\vec{u}} = \alpha x_{\overrightarrow{AB}} + \beta x_{\overrightarrow{AC}} + x_A \\ y_C + ty_{\vec{u}} = \alpha y_{\overrightarrow{AB}} + \beta y_{\overrightarrow{AC}} + y_A \\ z_C + tz_{\vec{u}} = \alpha z_{\overrightarrow{AB}} + \beta z_{\overrightarrow{AC}} + z_A \end{cases}$$

On peut alors laisser toutes les inconnues  $\alpha$ ,  $\beta$  et  $t$  d'un côté, et on réécrit alors le système sous forme matricielle :

$$\vec{w} = \begin{bmatrix} x_C - x_A \\ y_C - y_A \\ z_C - z_A \end{bmatrix} = \begin{bmatrix} x_{\overrightarrow{AB}} & x_{\overrightarrow{AC}} & -x_{\vec{u}} \\ y_{\overrightarrow{AB}} & y_{\overrightarrow{AC}} & -y_{\vec{u}} \\ z_{\overrightarrow{AB}} & z_{\overrightarrow{AC}} & -z_{\vec{u}} \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ t \end{bmatrix}$$

En notant  $D$  le déterminant de ce système, on peut alors exprimer les valeurs  $\alpha$  et  $\beta$  des paramètres pour notre point  $P$ . On obtient alors :

$$\alpha = -\frac{1}{D} \vec{w} \wedge \vec{v} \quad \text{et} \quad \beta = -\frac{1}{D} \vec{u} \wedge \vec{w}$$

- ➊ 5. On remarque alors que  $\alpha$  et  $\beta$  correspondent aux coordonnées de  $P$  dans le système  $(A, \overrightarrow{AB}, \overrightarrow{AC})$ . Ainsi, pour que  $P$  soit dans le triangle, il faut que ces coordonnées soient positives et de somme inférieure à un. Il s'agit de la dernière condition. Ces conditions peuvent être visualisées à la Figure 48 (attention, sur cette figure, le  $C$  est celui du triangle et non pas le point de départ du rayon).

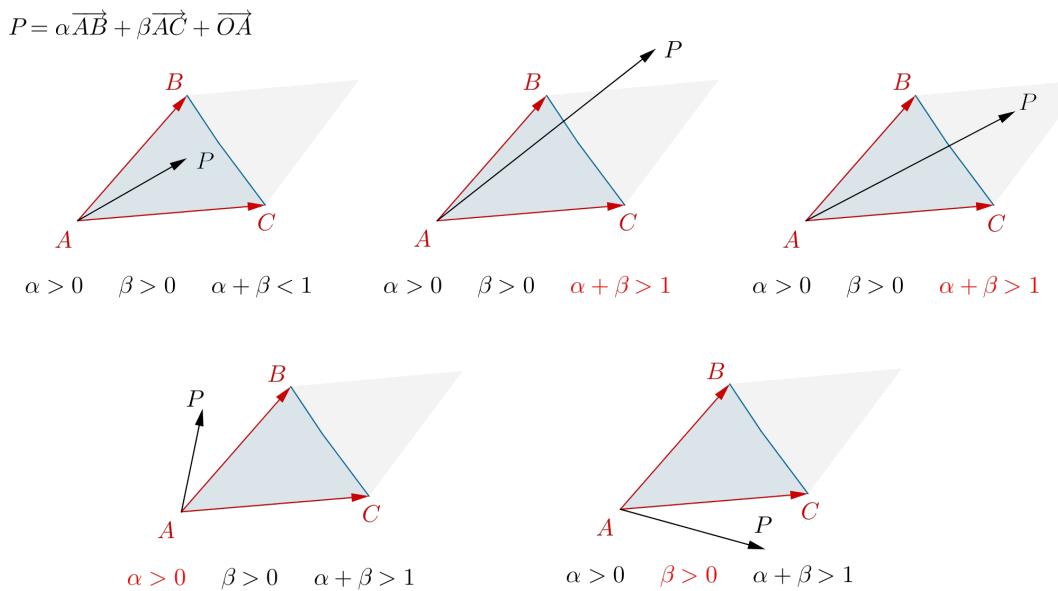


FIGURE 48 • Condition sur les coordonnées de  $P$  pour que ce dernier appartienne au triangle  $ABC$

Une fois ces aspects théoriques déterminés, il suffit juste de les coder, ce qui se fait aisément avec les méthodes que nous avons défini. On obtient alors le code suivant :

```
1 bool Geometry::intersect(const Ray r, int id, Vector &N, double &t) const {
2     // Les trois points du triangle
3     const Vector &A = vertices[faces[id*3]];
```

```

4   const Vector &B = vertices[faces[id*3+1]];
5   const Vector &C = vertices[faces[id*3+2]];
6
7   // Le rayon intersect-t-il le plan?
8   N = (C-A).cross(B-A).getNormalize();
9   t = N.dot(A-r.C) / N.dot(r.u);
10  if(t<=0){
11      return false;
12  }
13
14  // Résolution utilisant le système vu
15  Vector P = r.C + t*r.u;
16
17  Vector u = B-A;
18  Vector v = C-A;
19  Vector w = P-A;
20
21  double uu = u.dot(u);
22  double uv = u.dot(v);
23  double vv = v.dot(v);
24  double uw = u.dot(w);
25  double vw = v.dot(w);
26
27  double detM = uu*vv - uv*uv;
28  double alpha = (uw*vv - vw*uv)/detM;
29  double beta = (uu*vw - uv*uw)/detM;
30
31  // Conditions pour que P soit dans le triangle
32  if(alpha>0 && beta>0 && alpha+beta<1){
33      return true;
34  }
35
36  return false;
37 }
```

**Trouver la meilleure intersection** Pour terminer, la première méthode va donc passer en revue tous les triangles et retourner parmi tous les points d'intersection celui de plus faible  $t$  (comme nous le faisons dans la scène). Si un point est retourné, on l'ajoute à la liste des points d'intersection finale.

Le code est sensiblement identique à celui que l'on peut trouver dans la classe *Scene*.

Nous disposons donc de toutes les méthodes de base pour pouvoir rendre un maillage composé de triangles.

### 10.3 Exemple de rendu

L'ordinateur sur lequel j'effectue les rendus n'est pas extrêmement puissant. Le rendu proposé n'est donc pas en très bonne résolution, mais permet de montrer un des résultats de ce code.

Un exemple de rendu vous est ainsi proposé à la Figure 49.

*Sur cette partie, on notera que le code est moins abouti que celui proposé dans le répertoire commun. La raison est que j'ai préféré m'attarder sur le travail des textures en fin de module, et je ne me suis donc pas mis à jour sur la fin de la partie maillage de mon camarade. Ne voulant pas présenter un code auquel je n'aurai pas pris part et que je n'aurais pas compris, j'ai préféré laisser de côté la partie sur l'optimisation du calcul du maillage (BVH) et le calcul des normales en utilisant les coordonnées barycentriques pour obtenir un résultat plus lissé.*



FIGURE 49 • Exemple de rendu de maillage

## Conclusion

---

En conclusion, ce travail nous a permis d'avoir une première approche du Raytracing et de ses applications (et ses charmes !). Le code développé propose un certain nombre de fonctionnalités de base utiles, mais du travail reste encore à faire.

Ainsi, sur cette version finale, nous avons noté une erreur dans la gestion de l'éclairage indirect, qui reste trop bruité. Nous n'avons cependant pas réussi à corriger ce dernier. Dans le même temps, des comportements étranges apparaissent parfois avec les boîtes au niveau de leurs ombres, ou une bande d'ombre peut apparaître au milieu de ces dernières.

Mis à part ces deux bugs relevés, le code fonctionne et fournit déjà une bonne latitude de liberté à l'utilisateur, que ce soit sur le choix et la construction (CSG) des formes ou encore les textures.

Pour continuer ce projet, il serait intéressant de traiter les points suivants :

- *Ajouter plus de textures* Que ce soit des textures utilisant un bruit de Perlin ou totalement déterministes, nous n'avons fourni ici que des grandes lignes sur la réalisation de ces textures. Ainsi, il serait intéressant de compléter cette offre. Des possibilités seraient de créer des nuages (en utilisant Perlin), une structure d'anneau (en utilisant un sinus sur les sphères), du feu (avec Perlin)....
- *Aller plus loin avec Perlin* Les premiers exemples ont été concluant. Cependant, il serait intéressant d'aller chercher plus loin, en particulier vers le bumping en utilisant un bruit de Perlin. Cette technique revient à bruire les normales d'un objet pour lui donner un aspect irrégulier. Ceci serait intéressant pour nous sortir des objets parfaits que nous utilisons jusqu'à présent.
- *Caméra* Il serait intéressant de pouvoir coder le mouvement de la caméra, afin de pouvoir réaliser de petites vidéos ou cette dernière se déplacerait dans la scène.
- *Diversifier les sources étendues* Outre la correction du bug, il serait intéressant de proposer à l'utilisateur d'autre source que la sphère pour des sources étendues.
- *Enrichir les BRDFs* En particulier, en ajoutant les matériaux glossy dans la gamme proposée. D'autres idées seraient encore possibles, mais ces dernières semblaient les plus adaptées au regard du code qui est actuellement disponible.