Aix*Marseille université
Initiative d'excellence
Année 2018-19

Domaine Sciences et Technologies
Licence informatique

Programmation 2 : Un raytracer
Code UE : SIN3U02

Projet Programmation 2, pour ceux en avance

1 Introduction

1.1 Explication

Ce projet est un projet pour les étudiants en avance sur les TP. Il est totalement optionnel.

1.2 Qu'est-ce qu'un raytracer ?

Le raytracing est une technique de rendu d'images d'environnement 3D, produisant des images de très bonne qualité. Les raytracers les plus aboutis sont capables de créer des images photoréalistes de scènes complexes, et sont utilisés notamment pour créer des effets spéciaux dans les films ou des films d'animation. Nous nous proposons de construire un raytracer plus modeste mais néanmoins capable de créer des images gérant des éclairages sophistiqués, de la réflection et peut-être même de la réfraction. Le principe est heureusement relativement simple, il nous demandera d'utiliser un peu d'algèbre et de géométrie, et surtout une bonne compréhension de la programmation orientée objet.

L'œil perçoit une image grâce à la détection des photons arrivant sur la rétine. Ces photons arrivent en ligne droite depuis la source de leur émission, typiquement un objet. Le photon a été émis par l'objet :

— soit car l'objet est éclairé par une ou plusieurs sources de lumières,
— soit car l'objet est réfléchissant (un miroir), le photon provient d'un autre objet situé dans une position symétrique de l'œil,
— soit car l'objet est transparent (une vitre), le photon est passé au travers depuis une source plus lointaine.

En traçant un ligne droite depuis l'œil, on peut retrouver l'objet perçu par la rétine. En traçant une ligne droite depuis cet objet vers les sources de lumières, on détermine s'il est éclairé, en traçant une ligne droite dans la direction de réflection on peut déterminer la couleur du reflet, et en traçant une ligne droite à travers l'objet transparent on peut déterminer ce qui est vu à travers l'objet. Autrement dit, on essaie de suivre le chemin du photon en sens inverse. Dans tous les cas, la primitive du calcul est de déterminer quel est le premier objet se situant sur une demi-droite issue d'un point. De là vient le terme *raytracer* ou *lancer de rayon*.

1.3 Organisation du projet

Nous allons réaliser ce raytracer en faisant du développement incrémental. Nous allons partir d'une application minimale mais fonctionnelle (typiquement l'affichage d'une image noire), puis ajouter des fonctionnalités une par une. Chaque fonctionnalité ajoutée doit être modeste ; ainsi :

— les étapes du développement sont courtes,
— les potentielles erreurs de programmation se cachent dans la petite partie d'implémentation venant d'être réalisée et sont donc plus facile à trouver,
— on réalise rapidement les progrès effectués, ce qui constitue une bonne source de motivation.

Chaque fonctionnalité doit être testée avant de passer à la suivante. Ceci permet ensuite de se concentrer sur la fonctionnalité d'après, avec la confiance que ce qui a été produit avant est fiable (la plupart du temps). Nous apprendrons à mettre en place des tests pour nous assurer que le programme est correct et n'est jamais dégradé par l'ajout d'une fonctionnalité.

Il existe de nombreuses ressources concernant le raytracing sur Internet. N'hésitez pas à les consulter pour mieux comprendre les aspects mathématiques ou techniques du projet, et approfondir le sujet.

1.4 Versioning

Ce projet est l'occasion d'apprendre à utiliser des outils de développement professionnel. En plus de l'IDE IntelliJ, vous aurez à utiliser un outil de gestion de versions, en l'occurrence `git`. Un tel outil permet d'enregistrer à distance votre projet et de permettre à plusieurs personnes de travailler simultanément, sans devoir échanger les fichiers par courriel, et sans se marcher sur les pieds (par exemple modification sans le savoir du même fichier).

Le principe de `git` est d'avoir un *dépot* distant : une version de votre projet stockée sur un serveur sur Internet (en l'occurence par `github`). Vous disposez en plus de dépots locaux sur les ordinateurs sur lesquels vous travaillez. Vous faites vos modifications sur votre ordinateur, et lorsque vous avez accompli une amélioration qui fonctionne bien, vous pouvez la faire enregistrer (`commit`) par `git`. Ces enregistrements sont locaux à votre ordinateur, et vous pouvez en faire autant que vous le souhaitez. Si vous voulez partager votre travail avec votre équipe, il vous faut l'envoyer vers le dépot distant (`push`). À l'inverse, si vous souhaitez récupérer le travail fait par vos co-équipiers, il faut ramener ces modifications depuis le dépot distant (`pull`). IntelliJ est capable de gérer `git` ; vous trouverez dans le menu `VCS` l'option `Commit`, et l'option `Git` qui contient `push` et `pull`.

Vous travaillerez de la façon suivante :

— Si vous ne possédez pas de compte Github, aller sur https ://github.com et créez-vous un compte gratuit.
— Allez sur ce lien https ://classroom.github.com/g/MmYUxqO9, créer une équipe ou joignez une équipe existante (2 personnes maximum par équipes). Mettez vous d'accord avant !
— Au tout début, vous lancerez IntelliJ, puis commencerez un nouveau projet depuis un dépot `git` (menu `File`, `New`, `Project from version control`, `Git`). Renseignez l'adresse du dépot `git` fournie par `github`, vous la trouverez en cliquant sur le bouton vert `clone or download` sur la page `github` du projet. IntelliJ va initialiser votre dépot local, en vous demandant dans quel répertoire le placer.
— Modifiez le fichier `README.md`. Mettez votre nom. Faites un `commit` avec pour message "inscription d'un membre de l'équipe", puis un `push`. Votre équipier fait un `pull` puis fait de même.
— À chaque question répondue, faites un `commit` en sélectionnant les fichiers modifiés. Chaque `commit` doit contenir un message précisant la nature des modifications effectuées.
— À chaque incrément terminé, faites un `push` de votre travail.
— Ceci est le minimum. Vous pouvez faire plus de `commit` et plus de `push`, ainsi que des `pull` pour récupérer le travail de votre co-équipier.
— Si vous avez un problème et souhaitez l'aide de votre instructeur en dehors des séances, un `push` lui permet de voir votre programme.

2 Incrément 1 : affichage d'une image noire

Afin de rendre le projet plus ludique et de nous voir progresser à chaque incrément, on choisit de commencer par créer l'affichage de l'image, avant de travailler sur sa génération. Chaque incrément permettre d'améliorer les images obtenues.

Pour gérer l'affichage, nous utilisons une librairie permettant de gérer les interface graphique, la librairie JavaFX fournie en standard avec les librairies standards de Java. IntelliJ ou Eclipse permettent de créer rapidement une application minimale utilisant JavaFX. Nous vous fournissons un projet ainsi généré, auquel nous avons ajouté le minimum pour obtenir l'affichage d'une image, pour l'instant toute bleue. Le projet contient :

— une classe `Main`, qui s'occupe de lancer l'application. On lance l'application en cliquant sur le triangle vert dans la marge à gauche de la déclaration `public class Main`. Après un délai, une fenêtre bleue devrait s'ouvrir, c'est notre application.
— un fichier `imageViewer.fxml`, qui contient une description de ce que doit contenir la fenêtre graphique. Elle contient un canvas (le terme technique pour une zone de dessin).
— une classe `Renderer`, qui contient la partie du programme gérant la fenêtre graphique.
— une classe `Raytracer`, qui contient l'algorithme déterminant la couleur de chaque pixel.
— une classe `Helpers`, dans laquelle vous trouverez une fonction de comparaisons des `double`.

En plus de ces fichiers, contenus dans les sous-répertoires du répertoire `src`, on trouve un répertoire `test` qui contiendra les tests. Pour l'instant, il existe un seul test pour la version actuelle de la classe `Raytracer`. Pour passer le test, ouvrir la classe `RaytracerTest` et cliquez sur le triangle vert à gauche du test. Un rapport s'affichera pour dire si le test est validé ou non.

1. Assurez-vous que le test passe, en modifiant au besoin la classe `Raytracer`. Ne modifiez pas le test, car nous voulons une image noire.
2. Vérifiez que l'application affiche maintenant une image noire.
3. Comment faire pour avoir une image d'une autre dimension, par exemple en 300 par 200 ? Cherchez où les dimensions de l'image sont définies.
4. Modifions un peu l'image. Essayez d'obtenir des rayures verticales noires et blanches. Faites des rayures de largeurs 1 pixel, puis 10 pixels. Ensuite essayez d'obtenir un damier. Adaptez les tests à chaque étape.
5. Parcourez l'implémentation de chaque classe. Demandez à votre instructeur de vous expliquez les détails du programme dont vous ne comprenez pas le rôle.

3 Incrément 2 : Vecteurs

Puisque nous travaillons sur un problème de nature géométrique, nous allons avoir besoin de vecteurs et de points. Pour simplifier les futurs calculs, nous utilisons des vecteurs de dimension 4, la 4^e dimension contiendra un 0 pour les vecteurs et un 1 pour les points (cette technique permettra de gérer la translation comme une opération linéaire, et donc encodable par une matrice).

1. Créez une classe `Vector` dans le répertoire `geometry` de la même façon. Ajoutez 4 champs, pour les 4 coordonnées. Pour créer une classe, clic droit sur le répertoire `geometry` dans la barre de navigation à gauche de l'écran, puis choisir les options `nouveau` et `classe`.
2. Ajoutez un constructeur prenant les 4 coordonnées en paramètres.
3. Ajoutez deux méthodes

   ```
   public static Vector point(double x, double y, double z)
   public static Vector vector(double x, double y, double z)
   ```

   Ces deux méthodes invoquent le constructeur pour construire une instance de `Vector` et la retourner.
4. Que veut dire le mot-clé `static` ?
5. Ajoutez une méthode `public Vector add(Vector vec)` générant un nouveau vecteur par l'addition de `this` et `vec`.
6. Ajoutez une méthode de test d'égalité de deux vecteurs. Vous pouvez utiliser IntelliJ, menu `Code`, `générer`, `equals() and hashcode()`. Attention, `==` n'est pas adapté pour tester l'égalité de `double`, utilisez plutôt la fonction fournie `Helpers.compareDouble(double, double)`. De la même façon, ajoutez une méthode `toString`.
7. Créez une classe de tests pour `Vector`. Pour cela, positionnez le curseur sur la déclaration `public class Vector`, faites simultanément `Alt-Entrée` et sélectionnez l'option de créer des tests. Suivez les instructions.
8. Ajoutez des tests pour chacune des méthodes de la classe `Vector`. Assurez-vous que toutes les méthodes se comportent comme prévues.

Pour l'instant, on se contente de l'addition de vecteurs. Nous aurons certainement besoin de plus par la suite, mais nous ajouterons des méthodes au fur et à mesure que les besoins apparaîtront.

4 Incrément 3 : Création d'une caméra

L'image est construite en lançant depuis l'œil des rayons dans chaque direction du champ de vision, pour déterminer la couleur de chaque pixel. Nous utilisons un objet caméra dont le rôle est de déterminer pour chaque pixel, la direction du rayon pour ce pixel.

On considère la caméra en position $(0, 0, 0)$, regardant dans la direction $(0, 1, 0)$ (c'est un choix arbitraire, on pourra raffiner plus tard pour déplacer la caméra). Pour calculer la direction associée à un pixel, on peut imaginer un écran rectangulaire placé sur l'hyperplan $y = 1$ avec son centre en $(0, 1, 0)$. Cet écran représente le champ de vision de la caméra : ce qui apparait du point de vue de la caméra dans cet écran est l'image à afficher. Le coin inférieur droit de cet écran est en position $(-x, 1, -z)$ et le coin supérieur gauche en position $(x, 1, z)$. x est choisi de façon à voir l'angle de vue souhaité, par exemple $x = 1$ donne un angle de vue de 90°, $x = \sqrt{3}$ donne un angle de vue de 120° (quelle fonction trigonométrique utiliser ?). z est choisi pour le ratio x/z soit égal au ratio longueur sur hauteur de l'image.

1. Créez une classe `Camera` dans le répertoire `tracer`. Ajoutez un constructeur prenant l'angle de vision en degré, le rapport d'aspect (rapport longueur sur hauteur de l'image), et la longueur et la largeur de l'image en nombre de pixels.
2. Dans la classe `Vector`, ajoutez une méthode `public Vector normalize()` pour retourner le vecteur normalisé de `this`. Ajoutez des tests dans le fichier de tests de la classe `Vector`.
3. Ajoutez une méthode prenant les coordonnées entières x, y du pixel, et retournant un vecteur normalisé correspondant à la direction du rayon pour ce pixel. Les fonctions trigonométriques et de nombreuses autres fonctions mathématiques sont préfixés ainsi : `Math.sin(double)`, etc.
4. Créez une classe `Ray` dans `tracer`, regroupant le point d'origine et la direction d'un rayon.
5. Créez une méthode `public Ray ray(int x, int y)` retournant le rayon associé à un pixel. La direction de ce rayon doit être normalisée (attention, plus tard les directions des rayons ne seront pas toujours de norme 1).
6. Ajoutez des tests pour la classe `Camera`.
7. Ajoutez une méthode `public double dotProduct(vector vec)` dans la classe `Vector` calculant le produit scalaire, et ajoutez des tests pour cette méthode.
8. Ajoutez une propriété `Camera` dans `Raytracer`. Le constructeur doit recevoir la caméra en argument.
9. Pour visualiser l'effet obtenu, modifiez `Raytracer` pour que la luminosité d'un pixel soit proportionnelle au produit scalaire entre la direction du rayon pour ce pixel et le vecteur $(0, 1, 0)$. Regardez la documentation de `javafx.scene.paint.Color` pour voir comment créer des couleurs arbitraires.

5 Incrément 4 : Création d'une chose

Nous devons déterminer si chaque rayon intersecte une des choses constituant la scène à afficher. Pour commencer, les scènes ne contiendront qu'un seul objet. Le principal service rendu par un objet est de dire si un rayon l'intersecte, et à quelle distance de la source du rayon a lieu cette intersection.

1. Créez une classe `Intersection` dans `tracer`, comprenant deux propriétés : un rayon, et une distance représentant la distance entre l'origine du rayon et le point d'intersection. La distance sera signée : si elle est négative l'intersection est en arrière du rayon.
2. Ajoutez un constructeur pour `Intersection` initialisant les deux propriétés.
3. Ajoutez un package `thing` dans `src`. Pour cela, clic droit sur le répertoire `src` dans la barre de navigation, options `nouveau` puis `paquet`.
4. Créez une classe `Sphere` dans `thing`, avec des propriétés adéquates et un constructeur.
5. Créez une méthode `public Optional<Intersection> intersect(Ray ray)` dans la classe `Sphere`, pour calculer la première intersection d'un rayon avec la sphère. Les instances d'`Optional` s'obtiennent soit par `Optional.empty()`, soit par `Optional.of(something)`, selon qu'il existe ou pas une intersection.

Le calcul des intersections est expliqué ci-dessous, vous pouvez aussi essayer de trouver la méthode vous-même.

6. Créez des tests pour vérifier le bon calcul de l'intersection. C'est une partie technique, propice aux erreurs complexes qui pourront nous empoisonner plus tard, donc prenez le temps de bien tester de nombreux cas (0, 1, 2 intersections, rayon partant de l'intérieur, de l'extérieur, intersections négatives uniquement, etc.). Testez avec des sphères de différents rayons, centrées en différents points.
7. Ajoutez une classe `Scene` dans `tracer`, qui contient pour l'instant juste une propriété `thing` de type `Sphere`. Ajoutez dans `Raytracer` une propriété de type `Scene`, initialisée par le constructeur qui doit recevoir la scène en paramètre. Déplacer la propriété `Camera` de `Raytracer` dans `Scene`. Le constructeur de `Raytracer` doit recevoir uniquement un objet du type `Scene`.
8. Modifiez `Renderer` pour y définir une scène, avec une sphère positionnée en $(0, 10, 0)$, et qu'elle soit passée en argument au constructeur de `Raytracer`.
9. Modifiez `Raytracer`, pour qu'un pixel soit blanc si le rayon intersecte la sphère de la scène, et noir si le rayon n'intersecte pas la sphère. Vérifiez que l'image obtenue est bien un disque blanc sur fond noir. Faites varier le rayon de la sphère et constatez l'effet sur l'image. Vous pouvez aussi jouez avec des niveaux de gris selon la distance de l'intersection.

5.1 Calcul de l'intersection d'une sphère et d'une droite

Une sphère donnée par un centre et un rayon est décrite comme l'ensemble des points tels que :

$$\|\text{point} - \text{center}\|^2 = \text{radius}^2$$

Une droite passant par un point source et ayant un vecteur direction est décrite comme l'ensemble des points :

$$\text{source} + \lambda \cdot \text{direction}$$

pour tout réel λ. Les points d'intersections entre la sphère et la demi-droite sont décrits par les réels λ tels que :

$$\|\text{source} + \lambda \cdot \text{direction} - \text{center}\|^2 = \text{radius}^2$$

En développant on obtient l'équation de second degré pour λ :

$$\|\text{direction}\|^2 \lambda^2 - 2\langle \text{point} - \text{center} | \text{direction} \rangle \lambda + \|\text{point} - \text{center}\|^2 - \text{radius}^2 = 0$$

On utilise la méthode du discriminant pour calculer les racines. La plus petite racine positive, si elle existe, détermine notre point d'intersection (c'est précisément la distance entre le source et le point d'intersection).

5.2 Utilisation d'`Optional`

`Optional` est une classe permettant de représenter des objets facultatifs : soit un objet, soit pas d'objet. Pour l'utiliser, il faut importer la classe. IntelliJ affiche Optional en rouge tant que l'import n'est pas fait. Pour ajouter l'import, positionnez le curseur sur le terme `Optional` en rouge, puis faites `Alt-Entrée`, l'import devrait être ajouté. Vous pouvez consulter la documentation d'`Optional` en ligne. Voici les méthodes dont nous avons besoin :

```
static Optional<T> empty() // pour créer une option sans objet,
static Optional<T> of(T t) // pour créer une option contenant l'objet `t`,
boolean isPresent() // pour vérifier si l'option contient un objet.
T get() // pour récupérer l'objet présent dans l'option.
```

6 Incrément 5 : Couleurs

Jusque-là nous avons utilisé la classe `Color` de JavaFX. Pour calculer les couleurs à afficher, nous aurons besoin d'additionner des couleurs entre elles (pour les mélanger), de les atténuer, et de les multiplier entre elles (pour les filtrer). Nous allons pour cela utiliser construire notre propre classe de couleurs (une solution meilleure serait plutôt d'étendre la classe de JavaFX, nous verrons comment faire plus tard).

1. Créez la classe `Color` dans `geometry`. Une couleur possède trois propriétés, les composantes de rouge, de vert et de bleu, codés par des `double` entre 0 (pas de couleur) et 1 (couleur saturée). Créez aussi un constructeur.
2. Ajoutez une méthode `Paint toPaint()` convertissant notre couleur en la couleur de JavaFX correspondante. Utilisez ce constructeur :

```
new javafx.scene.paint.Color(red,green,blue,1);
```

Attention, lorsqu'on additionnera deux couleurs, les composantes pourront devenir supérieure à 1 (c'est normal), mais ici `red`, `green` et `blue` doivent être au plus 1. Si une composante est supérieure à 1, on passe 1 comme argument au constructeur de JavaFX.

3. Ajoutez l'addition de deux couleurs (les composantes s'additionnent), la multiplication de deux couleurs (les composantes se multiplient), l'atténuation d'une couleur par un scalaire (les composantes sont multipliés par le scalaire).

```
public Color add(Color color)
public Color multiply(Color color)
public Color scale(double lambda)
```

4. Testez les méthodes de la classe `Color`. Vous aurez besoin d'une méthode `equals` dans `Color`, procédez comme pour la classe `Vector` pour la générer.
5. Ajoutez une classe `Material` dans `geometry` qui regroupe les propriétés physiques de la chose (pour l'instant la sphère). La seule propriété pour commencer est la couleur. Prévoyez un constructeur pour cette classe.
6. Ajoutez une classe `Surface` dans `tracer`. Cette classe permet de grouper les informations concernant la surface touchée par un rayon à son intersection. Elle contient pour l'instant uniquement le matériau de la chose au point d'intersection. Mettez aussi un constructeur.
7. Ajoutez dans `Sphere` une propriété de type `Material` et dans `Intersection` une propriété de type `Surface`. Modifiez les constructeurs et les méthodes pour prendre en compte l'ajout de ces propriétés.
8. Ajoutez une méthode `public Color getColor()` à la classe `Surface`.
9. Modifiez `Raytracer` pour que la couleur du pixel soit celle de la surface intersectée. Vérifiez le bon fonctionnement en testant divers matériaux de différentes couleurs.

7 Incrément 6 : Lumières

Pour créer une impression de relief, il nous faut des couleurs plus réalistes, créant des dégradés selon la position de l'objet par rapport à la lumière. Nous allons ajouter des lumières ponctuelles : des lumières situées en un point de l'espace et éclairant depuis ce point.

1. Ajoutez une classe `Light` dans `tracer`, avec trois propriétés :
   — le lieu où se trouve la lumière (un point),
   — la couleur de cette lumière,
   — l'intensité de la lumière (de type `double`). Mettez aussi un constructeur.

2. Ajoutez dans `Scene` une propriété de type `Light`, modifiez le constructeur de `Scene`. Ajoutez une lumière (choisissez les valeurs des propriétés à votre guise) à la scène de l'image actuelle, de sorte que le programme compile. Pour l'instant la lumière n'a pas d'effet.
3. Pour calculer la luminosité d'un point à la surface d'une chose, il est important de connaître le vecteur normal à la surface, au niveau de ce point. Modifiez `Surface` pour ajouter une propriété `Vector normal`. Modifiez le constructeur de `Surface`.
4. Dans `Sphere`, lors de la construction de l'intersection, calculez la normale à la surface de la sphère au point d'intersection (simplement le vecteur entre le centre et le point d'intersection, normalisé). Testez que la normale est bien calculée.
5. Pour savoir si un point est bien éclairé, on doit déterminer si la chose et la lumière sont bien de part et d'autre du plan tangent à la surface. Une façon de le vérifier est de faire le produit scalaire entre le vecteur normal à la surface, et la direction de la source de lumière depuis le point d'intersection. Si c'est positif, la lumière éclaire bien la surface. Dans la classe `Surface`, modifiez `getColor` pour que si la surface n'est pas éclairée, le pixel correspondant soit noir. Il faudra ajouter un paramètre `Scene` à `getColor` pour que `getColor` puisse accéder à la position de la lumière.
6. Testez différentes position pour la lumière, vous devriez pouvoir créer des croissants de Lune.

8 Incrément 7 : Modèle de Phong

On raffine le calcul de la couleur d'un point. Pour cela, on distingue trois composantes.

— Une composante ambiante, la couleur à l'ombre.
— Une composante diffuse, qui dépend de l'exposition de la surface à la lumière : si le plan tangent à la surface est orienté de manière à être perpendiculaire au rayon lumineux, la lumière est plus forte. Au contraire si le rayon lumineux arrive en rasant la surface, la lumière est faible.
— Une composante spéculaire, qui crée du brillant là où les rayons lumineux se reflétent directement en direction de l'œil.

Un fichier `material.txt` contient les valeurs pour divers matériaux communs.

Chacune de ces composantes est plus ou moins forte selon la nature du matériau, et selon l'angle de la lumière au point d'intersection. Nous aurons besoin de 4 vecteurs tous normalisés :

— la direction $\vec{d}$ du rayon provenant de la caméra,
— la direction $\vec{l}$ de la source lumineuse depuis le point d'intersection,
— la normale $\vec{n}$ à la surface au point d'intersection,
— la réflexion de $\vec{l}$ par rapport au plan tangent à la surface, c'est-à-dire le vecteur $\vec{r} = \vec{l} - 2\langle\vec{n}|\vec{l}\rangle\vec{n}$.

1. Remplacez dans la classe `Materiel` la propriété de couleur par trois propriétés `ambiant`, `diffuse`, `specular` de type `Color`, et une propriété `shininess` de type `double`. Adaptez le constructeur.
2. Modifiez `getColor`. La couleur à afficher est la somme des trois couleurs (ambiante, diffuse et spéculaire). Créez donc trois méthodes privées, calculant chacune une composante. Pour l'instant, `getAmbiantColor` retourne la couleur ambiante du matériau, les deux autres retournent du noir.
3. Modifiez `getAmbiantColor`, pour qu'elle retourne le produit de l'intensité de la lumière par la couleur ambiante du matériau.
4. Modifiez `getDiffuseColor`. La couleur diffuse est obtenue en multipliant :
   — l'intensité de la lumière
   — la couleur diffuse du matériel
   — le produit scalaire de $\vec{l}$ et $\vec{n}$.

   Vérifiez que la sphère commence à avoir un effet de relief (essayez différentes valeurs pour le matériau et pour l'intensité lumineuse).
5. La composante diffuse ne doit être comptée que si la surface est tournée vers la source lumineuse. Faites en sorte que la composante diffuse soit noire si le produit scalaire précédent est négatif.

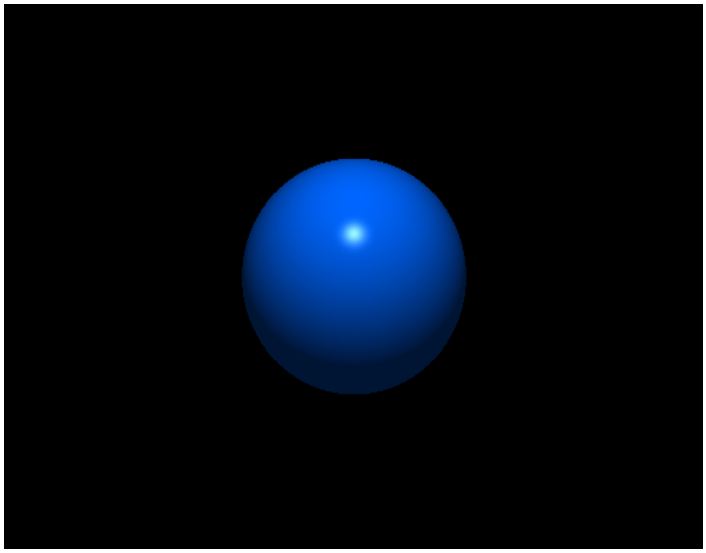

FIGURE 1 – Rendu d'une sphère

6. La composante spéculaire est elle-aussi noire si ce produit scalaire est négatif. Sinon, elle s'obtient par multiplication de :
   — l'intensité de la lumière,
   — la couleur spéculaire du matériel,
   — le produit scalaire de $\vec{d}$ et $\vec{r}$, puissance la brillance `shininess` du matériau (utilisez `Math.pow`). Si ce produit scalaire est négatif, la composante spéculaire est noire.

   Vous devriez apercevoir un point de brillance sur la sphère. Jouez aussi avec la position de la caméra pour voir l'effet de sa position sur les composantes spéculaires et diffuses. Vous pouvez aussi désactiver la composante diffuse pour tester la composante spéculaire.

9 Incrément 8 : plusieurs choses

Nous voulons maintenant construire des images avec plusieurs sphères.

1. Ajoutez à la classe `Intersection` cette méthode :

```
public static int compareByDistance(Intersection inter1,
                                    Intersection inter2) {
  return Double.compare(inter1.distance, inter2.distance);
}
```

   Quel est le rôle de cette méthode ?
2. Modifiez la classe `Scene`, pour qu'elle ait comme propriété une liste de `Sphere`. On instanciera la liste, de type `Collection<Sphere>` en utilisant un objet de la classe `ArrayList`.
3. Créez une méthode `addSphere` dans la classe `Scene`, qui ajoute une sphère dans la scène à afficher.
4. Ajoutez à `Raytracer` une méthode `List<Intersection> allIntersections(Ray ray)` calculant les intersections du rayon avec toutes les sphères.
5. Ajoutez ensuite à `Raytracer` une méthode `Optional<Intersection> findClosestIntersection(Ray ray)` déterminant la première intersection touchée par le rayon (de distance positive minimum). Vous pouvez trier la liste d'intersections avec l'instruction

```
intersections.sort(Intersection::compareByDistance);
```

6. Corrigez `Raytracer` pour prendre en compte les nouvelles méthodes.
7. Créez une scène avec plusieurs sphères et produisez une image, pour vérifier que le calcul de la plus proche intersection fonctionne. Que se passe-t-il si deux sphères s'intersectent ? Si une sphère est située devant une autre ?

10 Incrément 9 : Réflexion

On ajoute maintenant des effets de miroir. Pour cela, lors du calcul de la couleur d'un point, on lance un nouveau rayon, depuis le point dont on calcule la couleur, dans la direction symétrique à la direction de provenance du rayon initial. Si $\vec{d}$ est la direction du vecteur initial, et $\vec{n}$ la normale de la surface au point d'intersection, le rayon réfléchi a pour direction $\vec{d} - 2\langle\vec{d}|\vec{n}\rangle\vec{n}$. À partir de maintenant, le raytracer va devenir récursif : calculer la couleur d'un rayon demande de calculer la couleur du rayon réfléchi. Pour limiter la profondeur de récursion, on comptabilisera cette profondeur avec un argument `int depth`. Si la profondeur devient trop grande, on stoppe la récursion.

1. Créez une nouvelle méthode `Color getColor(Ray ray, Scene scene)` dans `Surface`. Renommez l'ancienne en `getPointColor`. Pour l'instant `getColor` appelle `getPointColor` et retourne le résultat tel quel. Assurez-vous de n'avoir rien cassé en vérifiant que le programme compile et fonctionne.
2. Ajoutez une méthode `getReflectedColor`, qui retourne du noir. Modifiez `getColor` pour additionner la couleur du point avec la couleur réfléchie.
3. Dans `Raytracer`, renommer `getColor` en `getPixelColor`, puis extrayez de `getPixelColor` une méthode `getRayColor(Ray ray)`.
4. Ajoutez dans `Raytracer` une constante `int MAX_DEPTH`, vous pouvez la mettre à 5. Ajoutez un argument `depth` à `getRayColor`, et dans `Surface` à `getColor` et `getReflectedColor`.
5. Dans `Raytracer.getRayColor`, faites en sorte que si l'argument `depth` est supérieur à la constante `MAX_DEPTH`, la couleur retournée soit noire. Assurez-vous que `depth` est incrémenté lors de l'appel à `getColor`.
6. Modifiez `Surface.getReflectedColor`. Calculez le rayon réfléchi, obtenez sa couleur récursivement, multipliez-la par la couleur spéculaire du materiel.
7. Vérifiez que vous obtenez bien une image avec de la réflexion. Vous devriez aussi observez de nombreux points noirs. Ces points noirs sont dus à des imprécisions numériques : lors du calcul du rayon réfléchi, il lui arrive de repartir de très légèrement derrière la surface, et donc d'intersecter de nouveau la surface.

   Pour éliminer les points noirs, modifiez la classe `Ray` pour déplacer le point d'origine légèrement dans la direction du rayon, par exemple d'une distance de `10 * Helpers.EPSILON`. Attention, cela modifie les distances d'intersection très légèrement, ce qui va affecter vos tests de distance d'intersection.

11 Incrément 10 : Ombres

Pour augmenter un peu plus le réalisme, nous allons ajouter des ombres au rendu. Pour cela, il faut décider pour chaque point à afficher s'il est à l'ombre ou pas. Lors du calcul de sa couleur, s'il est à l'ombre seule la composante ambiante est comptée, s'il est à la lumière on ajoute les composantes diffuse et spéculaire. Pour déterminer s'il est à l'ombre, on lance un rayon en direction de la source lumineuse : si l'intersection la plus proche est située avant le point lumineux, le point est dans l'ombre de cette intersection.

1. Ajoutez une méthode `isInShade(Light light, Raytracer raytracer)` dans `Surface`, pour déterminer si le point de la surface est dans l'ombre de la lumière. Pour cela on fabrique un rayon de ce point en direction de la source de lumière, on calcule la première intersection et on détermine si elle est plus ou moins proche que la source de lumière.

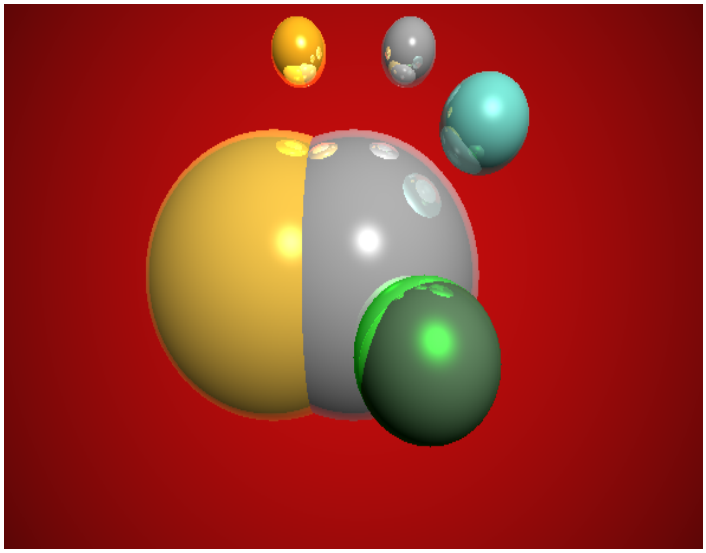

FIGURE 2 – Rendu de boules avec de la réflexion

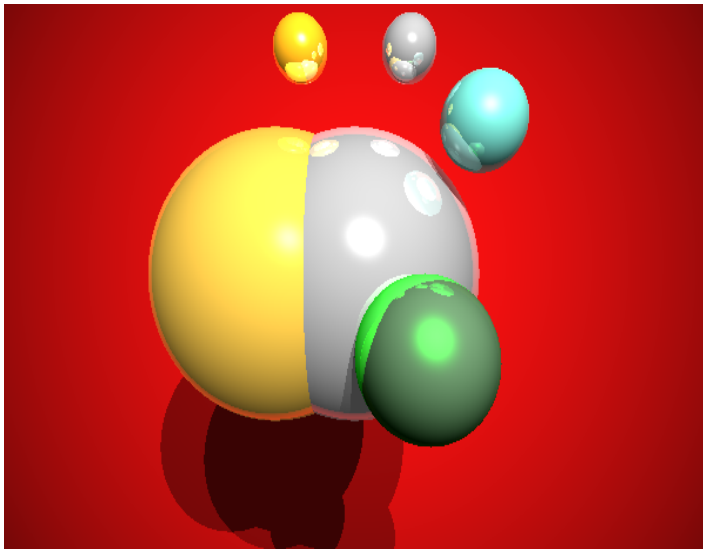

FIGURE 3 – Rendu de boules avec de l'ombre

2. Dans le calcul de la couleur de point, désactiver le calcul des couleurs diffuses et spéculaires si le point est à l'ombre.
3. Vérifiez que l'effet d'ombre est bien présent en affichant une scène de plusieurs sphères.
4. Ce sera plus impressionnant si on utilise plusieurs sources lumineuses. Modifier `Scene` pour avoir une collection de sources lumineuses (de type `Collection<Light>`) plutôt qu'une seule source lumineuse.
5. Ajoutez une méthode `addLight` dans `Scene`, pour ajouter des sources lumineuses à la scène.
6. Modifiez `Surface` : pour chaque source de lumière, on calcule la couleur du point pour cette lumière. Puis on ajoute toutes ces couleurs, ainsi que la couleur réfléchie.

12 Incrément 11 : Matrices

La prochaine étape consiste à nous permettre de transformer les sphères par des opérations linéaires, ce qui permettra de construire des ellipses par exemple. Pour cela, nous aurons besoin de calcul matriciel.

1. Créez dans `geometry` une classe matrice. Nous allons uniquement utiliser des matrices de dimension 4×4, ajoutez une constante `DIM = 4`.
2. Ajoutez une propriété, un tableau bidimensionnel de `double`.
3. Ajoutez une méthode `static Matix init(BiFunction<Integer,Integer,Double> toCoef)` créant une nouvelle matrice à partir d'un objet décrivant pour chaque paire d'indice le coefficient associé. On peut ainsi créer une matrice identité ainsi :

```
Matrix id = init ((i,j) -> i == j ? 1.0 : 0.0);
```

4. Nous aurons besoin de quelques matrices particulières. Pour chacune, implémentez une méthode statique créant une telle matrice.
   — la matrice identité,
   — la matrice de translation par le vecteur (x, y, z) :

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

   — la matrice de rotation autour de l'axe vertical (pour z), d'angle θ (donné en radian) :

$$\begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

   — la matrice de rotation autour de l'axe pour x, d'angle θ (donné en radian) :

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

   — la matrice de rotation autour de l'axe pour y, d'angle θ (donné en radian) :

$$\begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

   — l'aggrandissement/rétrécissement par un vecteur (x, y, z) :

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

   — la transvection, qui dépend de 6 paramètres :

$$\begin{pmatrix} 1 & x_y & x_z & 0 \\ y_x & 1 & y_z & 0 \\ z_x & z_y & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

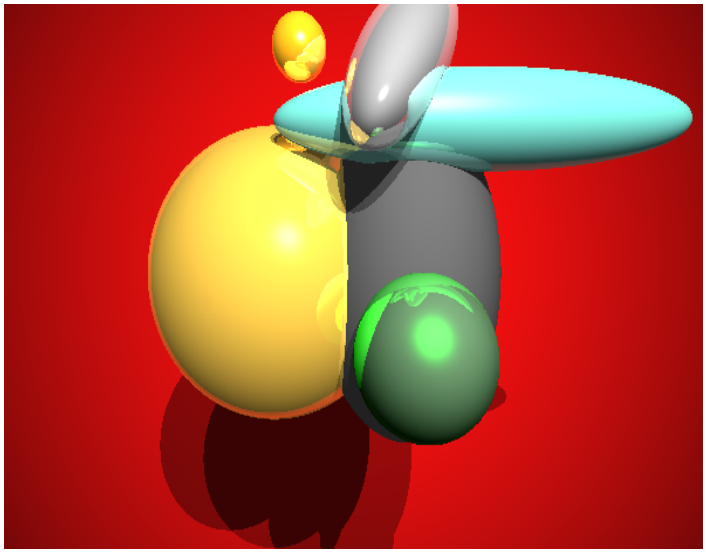

FIGURE 4 – Ellipses

5. Ajoutez une méthode `multiply` pour la multiplication de matrices retournant une nouvelle matrice, et une méthode `apply` qui multiplie une matrice par un vecteur (pour obtenir le vecteur image de la transformation représentée par la matrice).
6. Testez `multiply` et `apply`. Testez les matrices de transformations, leur effet sur les vecteurs.
7. Ajoutez une méthode `transpose` retournant la matrice transposée. Testez.
8. Ajoutez une méthode `inverse` calculant la matrice inverse, récupérez l'essentiel de l'algorithme sur Ametice et adaptez-le pour votre implémentation. Testez, il y a peut-être un piège.

13 Incrément 12 : Transformations

La prochaine étape consiste à proposer des transformations affines sur les sphères, pour afficher des ellipsoïdes.

1. Ajoutez une méthode `transform` dans `Ray`, qui prend une matrice et construit un nouveau rayon image de `this` par la matrice.
2. Ajoutez plusieurs propriétés à la classe `Sphere`, qui contiennent une matrice de transformation M, sa matrice inverse M^{-1}, et la transposée de son inverse $(M^{-1})^T$.
3. Ajoutez aussi un méthode pour appliquer une transformation (prise sous la forme d'une matrice N) à la sphère. La nouvelle matrice de transformation doit être NM. Recalculez les matrices inverse et transposée de l'inverse.
4. Modifiez la méthode `intersect` de `Sphere`. Simplement, le rayon utilisé pour le calcul doit être la transformation du rayon passé en paramètre par la matrice M de transformation de la sphère, et le point d'intersection obtenu doit être transformé par la matrice inverse.
5. Testez sur plusieurs exemples que la distance est correctement calculée. Essayez d'afficher une ellipse.
6. Le calcul de la normale est devenu faux (ce qui peut expliquer que les couleurs ou les réflexions ne sont pas convaincantes). Pour le corriger, il faut multiplier le vecteur normal, tel que calculé précédemment, par la transposée de la matrice inverse. Faites la modification, testez et générez des images d'ellipses. Vérifiez que les couleurs et les réflexions semblent réalistes.

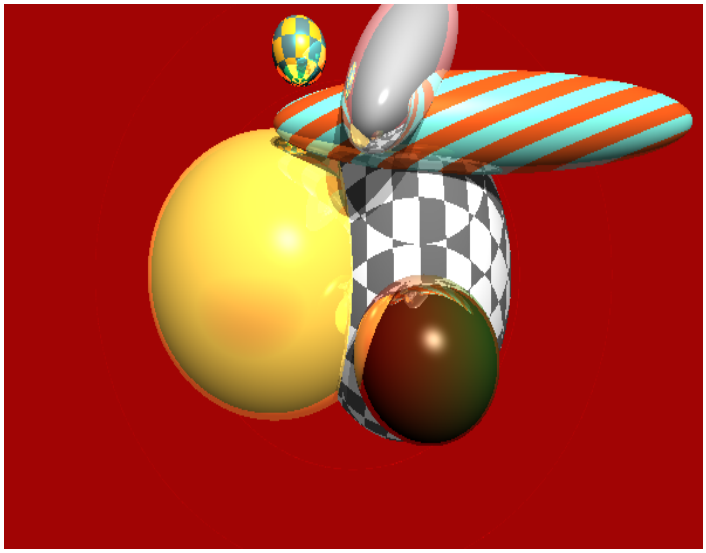

FIGURE 5 – Des textures !

14 Incrément 13 : Textures.

Ajoutons un peu de texture sur nos sphères, pour obtenir des effets intéressants.

1. Modifiez `Sphere` en ajoutant une méthode `getMaterial(Vector point)` qui renvoie un `Material` en fonction du point de la sphère. Pour l'instant, cette méthode retourne toujours le même matériau.
2. Créez une classe abstraite `Texture` dans `geometry`. Le principe de cette classe est de représenter des fonctions des points de l'espaces vers les matériaux. Elle dispose donc principalement d'une méthode `Material getMaterial(Vector point)`. On peut cependant aussi lui appliquer des transformations arbitraires, via une méthode `void transform(Function<Vector,Vector> f)` (qui n'est pas abstraite). `getMaterial` applique la transformation au point donné en argument, met à zéro la composante `z`, puis passe le vecteur ainsi obtenu à une méthode abstraite `getMaterialAtAbsolutePosition(Vector vector)`. Ainsi seule cette méthode sera implémentée dans les extensions concrètes,, sans tenir compte des transformations.
3. Ajoutez une classe `ConstantTexture`, qui correspond à avoir la même texture en tout point de l'espace.
4. Modifiez `Sphere` pour quelle possède une propriété `texture` plutôt que la propriété `material`. Ajustez la création des `Surface` dans l'algorithme d'intersection. Si la sphère est transformée, il faut que la texture suive. Pour cela, on appelle `getMaterial` de la texture avec comme argument le point d'intersection obtenu avant d'appliquer la matrice inverse.
5. Ajoutez une classe `GridTexture`, qui utilise deux matériaux différents organisés en grille de cube de dimension $1 \times 1 \times 1$. Pour cela, il faudra arrondir des `double` en `long`, à l'entier relatif inférieur le plus proche. Attention, `(long) myDouble` donne un entier plus grand si `myDouble` est négatif. Ajoutez donc pour cela une méthode statique `roundDown` dans `Helpers` et testez-là bien.
6. Dans la classe `Material`, ajoutez une méthode permettant de faire la moyenne pondérée de deux matériaux.
7. Créez une texture `LinearGradientTexture` paramétrés par deux matériaux, telle que les points avec $x < -1$ utilisent le premier matériau, les points avec $x > 1$ utilisent le deuxième matériau, et les points intermédiaires utilisent un mélange en proportion $(x+1)/2, 1-(x+1)/2$.
8. Créez aussi une texture `RadialGradientTexture` telle que l'origine possède un matériau, les points de norme inférieure à 1 utilisent un mélange, et les points au-delà utilisent un deuxième matériau.
9. N'hésitez pas à ajouter vos propres idées de textures !

15 Incrément 14 : Réfraction

La réfraction est le phénomène de transparence des objets : un rayon lumineux peut traverser un objet transparent. Il est néanmoins dévié : sa trajectoire fait un angle à l'interface entre les deux milieux traversés, selon la nature des matériaux transparents, selon les lois de Snel-Descartes.

1. La couleur d'un point d'intersection doit maintenant comporter une composante de plus : la couleur réfractée (ou transmise). Ajouter une méthode `getRefractedColor` dans `Surface`, retournant pour l'instant du noir. Faites en sorte que cette couleur soit prise en compte dans le calcul de la couleur du point.
2. Ajoutez une propriété booléenne `isOpaque` à Material. Si le matériau est opaque, il n'y a pas de composante réfracté, la couleur réfractée doit être noire.
3. Ajoutez dans `Surface` le calcul du rayon refracté, dans une méthode `getRefractedRay`. Pour l'instant, on considère que le rayon refracté part du point d'intersection dans la même direction que le rayon depuis la caméra.
4. Dans `getRefractedColor`, ajoutez un appel récursif au raytracer pour déterminer la couleur du rayon réfracté. La couleur obtenue doit être multiplié par la couleur diffuse du matériau.

   Créez une scène avec une ou plusieurs sphères transparentes pour tester l'effet obtenu. On doit pouvoir voir les objets derrière un objet transparent. Cependant, ce modèle n'est pas réaliste et ne fournit pas les effets impressionnants dont sont capables les raytracers. Pour cela, il faut tenir compte des lois optiques : les rayons lumineux changent de direction lorsqu'ils changent de milieu.
5. Ajoutez une propriété de réfractivité à la classe `Material`. C'est un nombre à virgule flottante valant au moins 1. Les indices de réfraction de divers matériaux sont facilement trouvables sur Internet, et varient entre 1 et $2,5$ en général. On se souvient que cette indice est le rapport entre la vitesse de la lumière dans le vide et dans le matériau en question.
6. Ajoutez une propriété `double refractionRatio` qui est la ratio des indices de réfraction entre le matériau dans lequel évolue le rayon depuis la caméra et le matériau dans lequel évolue le rayon réfracté. Ajoutez un mutateur pour cette propriété.
7. Le calcul du ratio de réfraction est subtil, car il faut retrouver les indices de réfraction des deux matériaux de part et d'autre de la surface. Dans la classe `Intersection`, ajoutez une propriété `Sphere thing` contenant la sphère intersectée. Faites en sorte qu'elle soit initialisée par le constructeur.
8. Dans `Sphere`, modifier la méthode de calcul des intersections. Elle doit désormais retourner la liste des tous les points d'intersections avec le rayon, y compris les points se trouvant derrière la caméra (à distance négative). Modifier les méthodes de la classe `Raytracer` : pour calculer le point d'intersection, on crée la liste des toutes les intersections avec tous les objets, on la trie, et on récupère le premier objet à distance positive.
9. Dans `Raytracer`, ajoutez une méthode pour calculer le ratio des indices de réfraction à l'intersection trouvée. Pourquoi est-ce compliqué ? si plusieurs sphères s'intersectent, l'indice de réfraction dans l'intersection n'est pas défini correctement (les sphères peuvent avoir divers matériaux). Pour cela, on considère que l'indice de réfraction est la moyenne [1] des indices des matériaux en présence. Pour calculer les matériaux en présence, on considère que chaque droite de l'espace intersecte chaque objet un nombre pair de fois : la droite rentre, sort, rentre, sort,... de l'objet. On parcoure la liste des intersections, et on stocke l'ensemble des objets ouverts après chaque intersection. Pour une intersection, si l'objet n'était pas dans l'ensemble, on l'ajoute, sinon on le retire de l'ensemble. On s'arrête dès qu'on arrive à la première intersection de distance positive. En l'absence de tout objet, l'indice de réfraction est 0.

   Une fois le ratio de réfraction obtenu, assurez-vous qu'il est mis-à-jour dans la surface de l'intersection trouvée.

1. on peut aussi prendre le minimum ou le maximum, peu importe, dans la nature les matériaux ne s'intersectent pas, ici on veut juste que le calcul donne le même résultat dans l'intersection quelque soit la droite choisie pour le rayon.

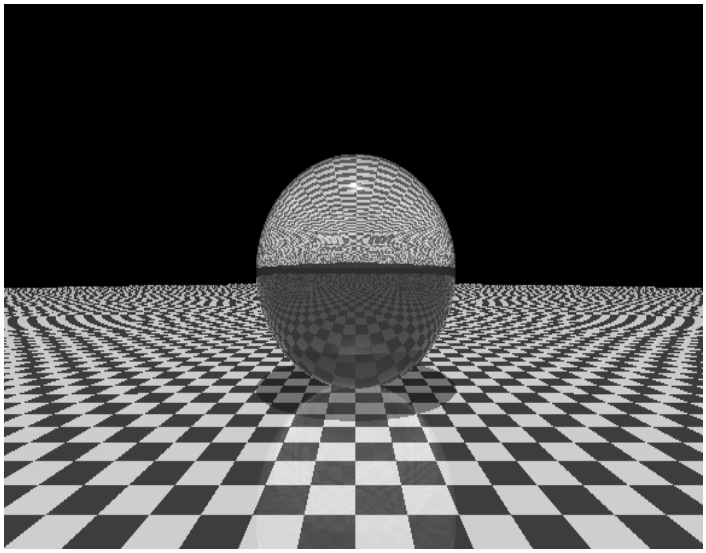

FIGURE 6 – De la réfraction

1. Modifiez le calcul du rayon refracté. Pour cela, on décompose la direction du rayon selon les composantes normales et tangentielles par rapport à la surface. Pour obtenir la direction du rayon refracté, on multiplie la composante tangentielle par le ratio de réfraction, puis on ajoute une composante normale de sorte à obtenir un vecteur unitaire. Testez !

16 Incrément 15 : d'autres formes

Nous avons dessinés suffisamment de sphères, réfléchissons à comment intégrer d'autres formes géométriques. Attention, nous allons devoir refactoriser notre programme, assurez-vous d'avoir de nombreux tests pour repérer facilement les problèmes que cela va créer.

1. Renommer `Sphere` en `Thing` en utilisant la fonctionnalité de renommage d'IntelliJ (shift-F6 sur le nom de la classe). Recréez une classe `Sphere` et déplacez-y toutes les méthodes de la classe `Thing`. Changez `Thing` en une classe abstraite, déclarez-y la méthode abstraite `List<Intersection> intersections(Ray ray)`. Faites que `Sphere` étende `Thing`. Les anciens constructeurs de `Sphere` sont devenus des constructeurs de `Thing`, ce qui est erroné car `Thing` est abstraite : remplacez donc les `new Thing` par `new Sphere`. Vérifiez maintenant que tout fonctionne (tests, affichage de scènes) avant l'étape suivante.
2. `Thing` peut faire plus de choses, par exemple gérer la matrice de transformations. Déplacer donc les matrices de transformation de `Sphere` vers `Thing`, ainsi que la méthode `transform`. Faites en sorte d'avoir dans `Thing` une méthode non-abstraite `intersection` et une méthode protégée et abstraite `basicShapeIntersection`. La première prend un rayon, lui applique la transformation, et le rayon transformée passe à la deuxième, récupère la liste d'intersections et pour chacune transforme la position du point d'intersection et la normale. Ainsi `Sphere` n'implémente que `basicShapeIntersection`. Pour cela, il faudra ajouter des méthodes dans la classe `Surface` pour y modifier les propriétés. Testez de nouveau que tout fonctionne toujours.
3. `Thing` peut aussi gérer la texture. Déplacez-y la propriété `Texture` et la méthode `getMaterial`. Il ne doit rester dans `Sphere` que le constructeur, la méthode `basicShapeIntersection`, plus des propriétés et méthodes privées.
4. Pour ajouter un nouvel objet, il suffit maintenant d'implémenter une méthode `basicShapeIntersection`. Commencer par définir un objet `Plane` représentant un hyperplan passant par l'origine, et défini par son vecteur normal. Faites en sorte que le nombre d'intersection avec le plan soit toujours pair : 0

FIGURE 7 – Avec un cube et des plans

pour un rayon parallèle au plan, et 2 si le rayon intersecte le plan, le deuxième point est pris à l'infini (`Double.POSITIVE_INFINITY` ou `Double.NEGATIVE_INFINITY`). Le principe est que l'intervalle entre les deux points définis "l'intérieur du plan", ce qui détermine les indices de réfractions de chaque côté. Assurez-vous que `Raytracer` ne considère par les points à l'infini.

5. Ajouter ensuite une classe `Cube` pour le cube dont les sommets sont les vecteurs de coordonnées $\{-1, 1\}^3$. Pour calculer les deux intersections, on calcule les intersections avec les hyperplans $x = 1$ et $x = -1$, et de même pour y et z. Pour chaque axe, on obtient une plus proche et une plus lointaine intersection. La plus lointaine des trois plus proches et la plus proche des trois plus lointaines détermine l'intersection (potentiellement vide) avec le cube (faites un dessin).