Lancer de rayons – TD1

Présentation

Le but de ce projet est de réaliser un programme permettant de générer des images en utilisant la technique du lancer de rayon (et ses extensions). Cette méthode permet de synthétiser des images d'une grande qualité, en prenant en compte des effets de lumière complexes.

Un affichage en 3D comme celui proposé par OpenGL n'est absolument pas nécessaire pour créer un tel programme. L'un des buts de ce projet est néanmoins de vous persuader de l'utilité et de la simplicité d'une telle visualisation pour la mise au point de l'algorithme et de vous faire reviser les bases d'OpenGL 3.x aquises dans l'UE PGHP. Ce *raytracer* sera implémenté en C++.

Ressources

- **Qt** est un toolkit graphique portable écrit en C++ permetant la création d'applications graphiques complexes. Dans le cadre de ces TDs, cette bibliothèque sera utilisée uniquement pour la création du context OpenGL, la gestion des fenêtres, et le chargement d'images. (documentation)
- **Eigen** est une bibliothèque C++ d'algèbre linéaire. Elle sera utilisée pour la représentation et manipulation de matrices, vecteurs, transformations géométriques et résolution de sysèmes d'équations. (documentation)
- **Cmake** sera utilisé pour gérer la compilation du projet. (tutorial, documentation)

Remerciements

Ce projet est basé sur ceux de Gilles Debunne et Gaël Guennebaud.

1 Prise en main

Télécharger et décompresser l'archive `sire_td1.zip`

```
> unzip sire_td1.zip
```

Plusieurs classes sont fournies afin que vous n'ayez à vous concentrer que sur les parties graphiques du TP. La plupart des classes auront à être complétées, d'autres ne seront utilisées complètement qu'à la fin du TP (`Material`). Parcourez le code rapidement pour en comprendre la structure:

- **RenderingWidget**: le visualiseur/débuggeur OpenGL. Possère un `RayTracer` et connait la `Scene`.
- **Raytracing**: chargé de lancer les rayons pour créer l'image.
- **Ray**: un rayon dans la scène.
- **Hit**: regroupe toutes les informations sur l'intersection d'un rayon avec un objet.
- **Material**: la définition complète du matériau d'un objet.
- **Scene**: regroupe tous les objets qui composent la scène.
- **Camera**: la caméra de la scène d'où on va créer l'image.
- **Shape**: classe abstraite dont dérivent les entités géométriques de la scène.
- **Object**: commun à tous les objets de la scène ; regroupe sa géométrie (classe `Shape`), son matériau (classe `Material`) et son shader OpenGL (classe `Shader`).
- **Sphere**: dérive de `Shape` et représente une sphère.
- **Mesh**: dérive de `Shape` et représente un maillage triangulaire.
- **Light**: classe abstraite dérivée en 2 types de lampes.

Première exécution

Créer un répertoire de build, configurer, et compiler :

```
> mkdir build
> cd build
> cmake ../sire_td1 -DQT_QMAKE_EXECUTABLE=/usr/bin/qmake-qt4
-DEIGEN3_INCLUDE_DIR=${HOME}/sire_td1/eigen3
> make
```

Exécuter le programme. Dans l'état actuel, le programme permet de charger des maillages (classe `Mesh`), créer des sphères (classe `Sphere`), visualiser une scène avec OpenGL 3.x (voir la classe `RenderingWidget`, fonction `paintGL`), naviguer dans la scène (classes `Camera` et `Trackball`).

Gestion de la scène

Parcourir la classe `Scene`, notamment la fonction `createScene` qui permet de créer la scène en chargeant les objets. Quel est le type et le rôle de l'attribut `mObjectList` ? Observer comment les classes `Mesh` et `Sphere` sont unifiées.

Pour faciliter les tests, les scènes peuvent être décrites par des fichiers au format XML. Ce type de fichiers permet de représenter une très grande variété de données dans un formalisme identique et rigoureux. Il permet également de facilement extraire et transformer cette information. La syntaxe des fichiers se comprend aisément en lisant un exemple minimaliste :

```
<?xml version="1.0" encoding="UTF-8"?>
<Scene>
 <Sphere radius="0.4">
  <Material>
   <DiffuseColor red="0.1" green="0.1" blue="0.9" />
  </Material>
  <Frame>
   <position x="0.1" y="0.2" z="0.0" />
   <orientation q0="0.0" q1="0.0" q2="0.0" q3="1.0" />
  </Frame>
 </Sphere>
</Scene>
```

Lorsque la touche 'l' est pressée, la méthode `loadFromFile()` du `RenderingWidget` ouvre une fenêtre pour demander un nom de fichier se terminant par l'extension '.scn' (remarquez que Qt permet de le faire en une ligne de code). Elle délègue ensuite le chargement du fichier à la `Scene` via la méthode `loadFromFile(const QString& filename)`.

Dans notre structure de fichier, les différents objets de la scène sont définis les uns à la suite des autres, comme fils du noeud racine `Scene`. À terme la scène pourra contenir des objets de différent type (lumières, maillage,...) et leur `tagName()` permettra de savoir quel type d'objet créer à partir du fichier. Pour l'instant, seul le chargement de la caméra, des sphères et des lumières est implémenté.

Voici un exemple de XML définissant une caméra :

```
<Camera fieldOfView="0.7854" xResolution="256" yResolution="256">
 <Frame>
  <position x="1.0" y="-1.0" z="0.5" />
  <orientation q0="0.0" q1="0.0" q2="0.0" q3="1.0" />
 </Frame>
</Camera>
```

Attention, cette `Camera` n'a pas de rapport avec la caméra qui affiche la scène dans le `RenderingWidget`, et qui permet de visualiser et de vérifier le bon déroulement de l'algorithme. Néanmoins la méthode `createDefaultScene`, appelée au chargement de la scène, fait

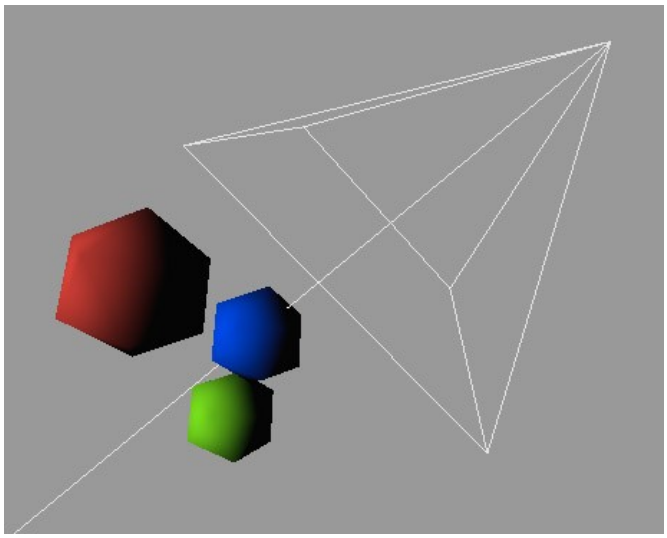
correspondre la première position de la caméra du `RenderingWidget` avec celle définie dans la scène. Vous pouvez également appuyez sur 'c' pour faire correspondre les deux caméras, ce qui vous permet d'avoir une représentation OpenGL de la scène qui va être rendue (attention, la résolution de la caméra correspondera à celle de la fenêtre).

2 Raytracing d'une sphère

Une fois que la structure du code fourni est assimilée, le premier objectif sera de visualiser une sphère par lancer de rayon. Le raytracer sera lancé lors d'un appui sur la touche 'r' du clavier (voir la fonction `RenderingWidget::pressKeyEvent`). Les illustrations suivantes seront générés à partir du fichier de scène troisSpheres.scn.

Nous allons lancer nos premiers rayons dans la scène. Pour l'instant, nous supposerons un seul rayon par pixel, et un modèle de caméra simplifié type sténopé. Le principe est simple : pour chaque pixel, lancer un rayon depuis la caméra vers la scène, chercher quel objet est le premier intersecté par ce rayon et donner au pixel la couleur correspondante.

Premier rayon

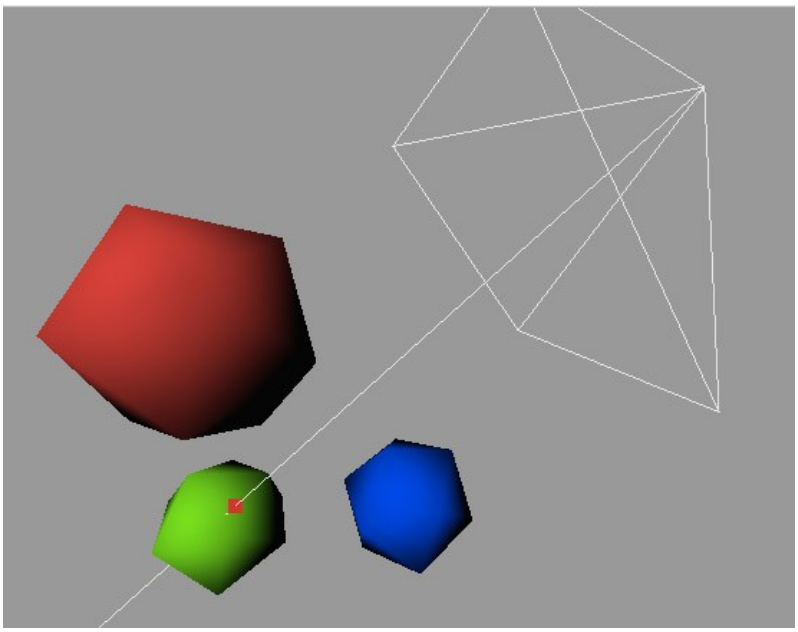


Un rayon est défini par un point de départ et une direction. La classe `Ray` représente un tel rayon. Pour simplifier les tests de votre implémentation, vous pouvez visualiser ce rayon en OpenGL à l'aide de la fonction `Line::draw` définie dans le fichier `GLPrimitives.h` et du shader `mFlatProgram`.

Dans un premier temps, on souhaite pouvoir lancer facilement un rayon dans la scène et suivre son trajet. Lorsqu'on clique avec le bouton gauche de la souris en maintenant la touche `Shift` enfoncée, le `RenderingWidget` appelle la méthode `select(const QPoint& point)`, le paramètre `point` contenant les coordonnées du point cliqué en espace écran. **Implémentez cette méthode afin qu'elle lance un rayon partant de la position courante de la caméra et passant par le pixel choisi.**

La classe `camera()` possède justement une méthode `convertClickToLine()` qui donne le bon résultat. **Ajoutez un membre `Ray` à votre `RenderingWidget`, initialisez-le dans `select` et affichez-le dans `paintGL`. Testez votre méthode : créez un rayon puis déplacez la caméra pour vérifier qu'il est bien dessiné au bon endroit (presser 'h' pour dessiner la camera dans sa position initiale).**

Intersection rayon-scène

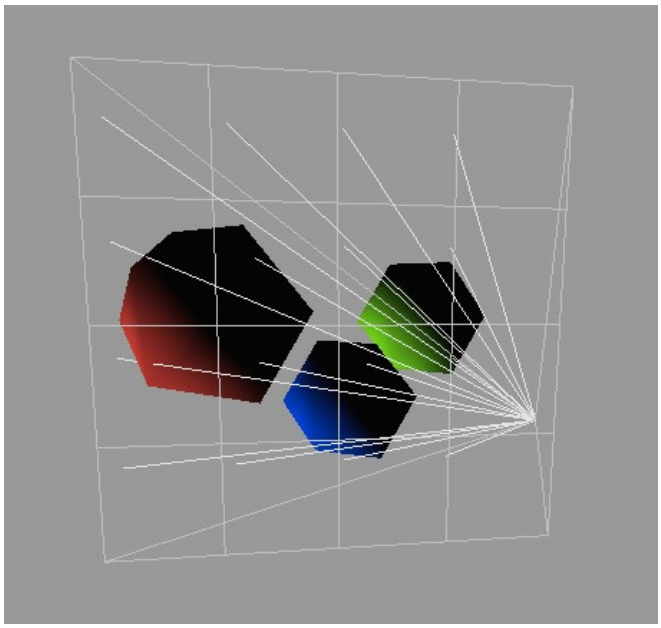


L'étape suivante consiste à trouver l'intersection (éventuelle) la plus proche du rayon avec les objets de la scène. On va pour cela utiliser une méthode `bool intersect(const Ray& ray, Hit& hit) const` dans la classe `Shape`. Elle renvera vrai ssi une intersection a été trouvée avec le rayon. La classe `Scene` possèdera la même méthode qui se contentera d'appeler successivement cette méthode sur tous les objets de la scène. **Implémentez ces deux fonctions.**
Remarque : dans l'implémentation fournie, les sphères peuvent être définies par la position de leur centre, mais aussi relativement à la matrice de transformation stockée dans la classe `Object`. Le centre est donc exprimé dans le repère local de l'objet après application de cette matrice. Ainsi, les sphères chargées depuis un fichier ont toutes pour centre (0,0,0) bien qu'elles ne soient pas placées à l'origine de la scène. Lors du lancer de rayon, il vous faut prendre en compte cette matrice de transformation, soit en l'appliquant à l'objet potentiellement intersecté (le centre et le rayon des sphères, en l'occurrence), soit en appliquant la transformation inverse au rayon (son origine et sa direction - n'appliquer que la partie linéaire de la transformation inverse à cette dernière).

Le paramètre `hit` de la méthode précédente va contenir toutes les informations d'intersection nécessaires pour la suite. La classe `Hit` stocke en particulier un "temps", correspondant à la distance de l'intersection à l'origine du rayon (d'où l'intérêt d'avoir normalisé la direction du rayon). Ce temps sera toujours positif (sinon cela représente une intersection avec un objet situé derrière l'origine du rayon) et il sera mis à jour par chaque objet qui intersecte le rayon à un temps inférieur au temps courant. Il est initialisé à une très grande valeur correspondant à une intersection à l'infini. Chaque objet teste donc s'il intersecte le rayon, et si c'est à un temps inférieur à celui actuellement stocké dans le `Hit` ; le `Hit` est modifié en conséquence.

Visualisez le résultat de votre code en appelant `intersect` lorsque vous cliquez sur un pixel avec `Shift` (i.e. dans la méthode `select`). Si une intersection est trouvée, affichez le point d'intersection (utilisez `Point::draw`). Vérifiez qu'il est bien toujours au bon endroit, en particulier dans les cas complexes (plusieurs sphères alignées intersectant le rayon).

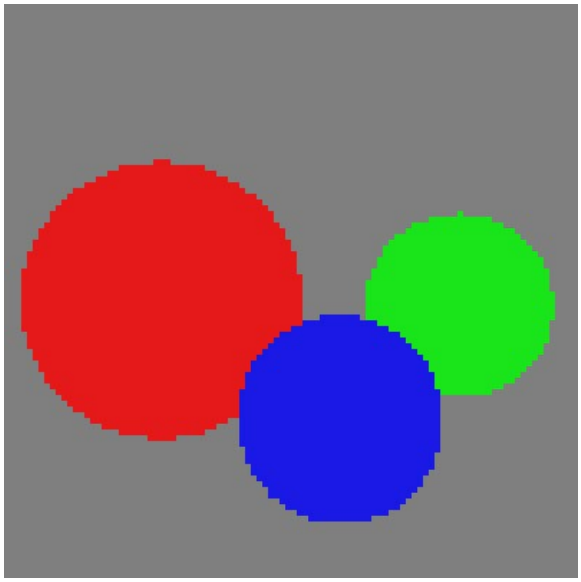
Affichage des rayons



L'étape suivante consiste à créer des rayons passant par le centre de chaque pixel. Les coordonnées de ces rayons doivent être exprimées dans le repère du monde, où se font tous les calculs. **Calculez l'origine et la direction de chacun de ces rayons dans la méthode statique `raytraceImage` de la classe `Raytracing`.**

Comme précédement, vérifiez vos résultats en affichant l'ensemble des rayons. Vérifiez en particulier que les rayons passent bien par les *centres* des pixels, et que celui correspondant à (0,0) est bien en haut à gauche.

Première image



Ajouter un attribut `mBackgroundColor` de type `Eigen::Array3f` à la classe `Scene`. Chargez-le depuis le fichier de scène s'il est présent. Il représentera la couleur de fond de la scène et donc des images.

Les images seront générées en appelant la méthode statique `raytraceImage` de la classe `Raytracing`. Cette méthode parcourt tous les pixels de l'image, calcule leur couleur et la stocke avec un `setPixel()` (voir la documentation de `QImage`).

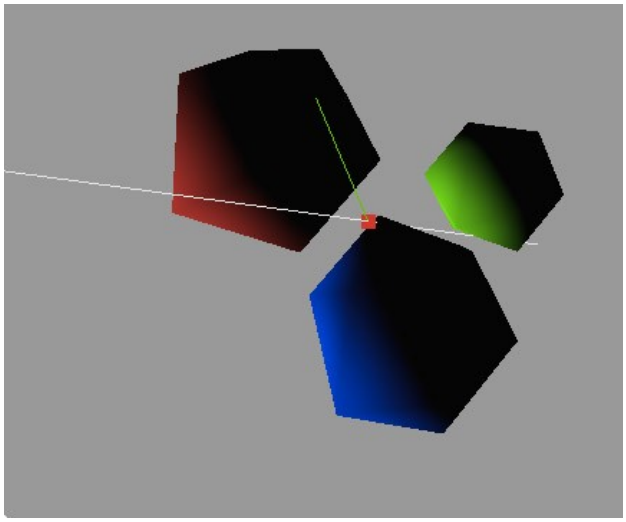
Pour calculer la couleur du pixel, `raytraceImage` utilise la méthode `rayColor`. Celle-ci se contentera pour le moment d'utiliser la méthode `intersect` de la `Scene`, en renvoyant la `diffuseColor` de l'objet intersecté (ou la couleur de fond de la scène s'il n'y a pas d'intersection). **Implémetez ces méthodes.**

Appuyer sur 'r' pour lancer les rayons et sauvegarder l'image du résultat. L'image est ici agrandie quatre fois pour voir les pixels. Vous pourrez par la suite ajouter un `QProgressDialog` à `raytraceImage` pour voir la progression des calculs longs.

3 Éclairage de base

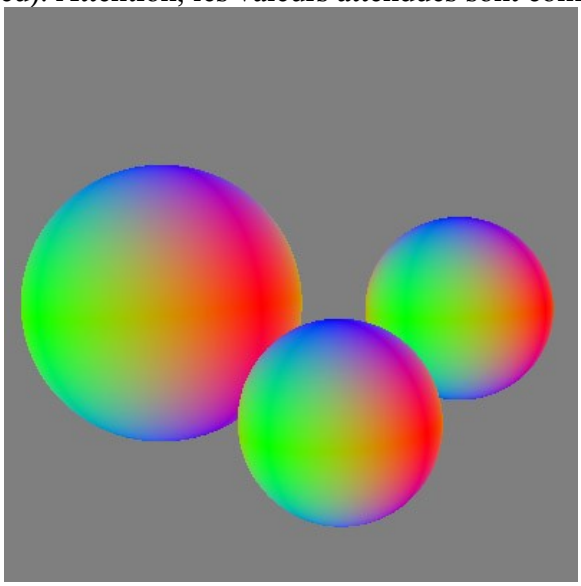
Une fois la génération des rayons primaires et le calcul d'intersection avec une sphère validés, nous allons implémenter un modèle d'éclairage simple à base du modèle de Blinn.

Normales



Lors de l'intersection d'un rayon avec un objet (les sphères dans notre cas), on va désormais également stocker dans le `Hit` la normale à la surface au point d'intersection. **Ajoutez ce code (dans `Hit` ainsi que dans la méthode `intersect`). Pour le tester, vous pouvez afficher une normale au point d'intersection (lorsque vous lancez un rayon avec `Shift+Click`)**.

Vous pouvez également affecter à chaque pixel une couleur représentant sa normale (x devient rouge, y vert et z bleu). Attention, les valeurs attendues sont comprises entre 0 et 1.

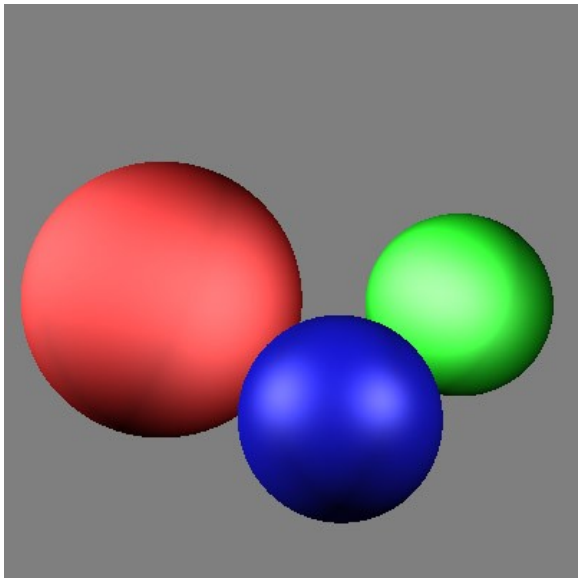


Lampes

Il faut ensuite définir dans notre scène une ou plusieurs lampes qui l'éclaireront. On considère pour l'instant deux types de sources : directionnelles et ponctuelles. La classe abstraite `Light` est donc dérivée en deux classes : `DirectionalLight` et `PointLight`.

Ajoutez un vecteur de pointeurs sur `Light` dans la scène et initialisez-le d'après le fichier de scène (complétez pour cela la méthode `loadFromFile`). Selon le `tagName()` rencontré, il faudra créer un objet de la classe adéquate.

Calculs d'éclairage



La couleur ambiante représente la couleur d'une lampe virtuelle qui éclaire toute la scène de façon uniforme. Cette lampe, également présente en OpenGL, n'a pas de signification physique et sert à "déboucher" les zones sombres. **Utilisez la couleur ambiant du matériau (en modulant son intensité) pour représenter cet éclairage.**

Les autres composantes de l'éclairage ont une intensité qui dépend du produit scalaire entre la normale à la surface et la direction de la lumière :

$$c_{pixel} = \text{SOMME}_{\text{source i}} [(\mathbf{L_i} \cdot \mathbf{N}) * c_{source\_i} * \text{BRDF}_{objet}]$$

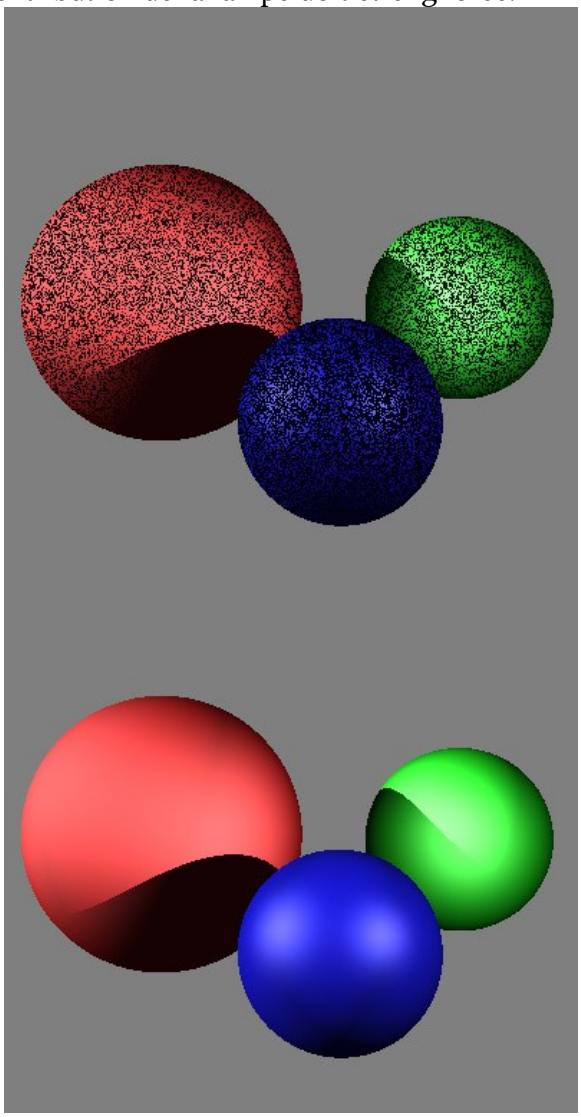
N est la normale au point intersecté sous le pixel, et $\mathbf{L_i}$ la direction depuis ce point vers la lumière. Le produit scalaire $\mathbf{L_i} \cdot \mathbf{N}$ doit être positif. Le mettre à zéro sinon pour que les surfaces qui ne font pas face à la lampe ne soient pas éclairées. La direction $\mathbf{L_i}$ est constante pour une `DirectionalLight` et dirigée vers la lampe pour une `PointLight`.

Complétez la méthode `raytrace` dans `Scene` afin qu'elle calcule la couleur au point d'intersection éclairé par les différentes lampes de la scène. Implémentez le modèle de matériau de Blinn-Phong (méthode `brdf()` de la classe `BlinnPhong`).

4 Rayons secondaires

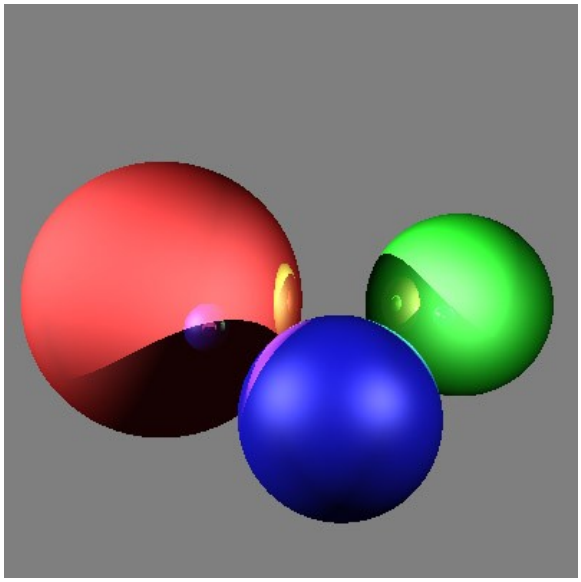
Les ombres

Pour générer des ombres, il suffit de vérifier que chaque source lumineuse est bien visible depuis le point d'intersection. Pour cela, **lancez un rayon depuis le point d'intersection en direction de la lumière, et testez s'il y a intersection**. Pour les `PointLight`, il faut de plus comparer le temps d'intersection avec la distance à la lampe : s'il est inférieur, un objet bloque la lumière. Si c'est le cas, la contribution de la lampe doit être ignorée.



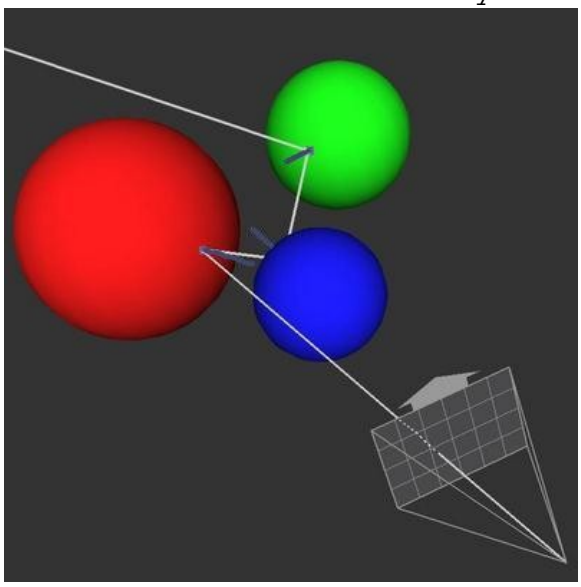
Vous allez probablement obtenir des images bruitées. En effet, les imprécisions numériques font qu'il peut exister une intersection entre un rayon partant de la surface d'un objet et l'objet lui-même. **Pour les corriger, contraignez le temps d'un `Hit` à être légèrement supérieur à zéro grâce à un epsilon (ou décaler l'origine du rayon secondaire de cet epsilon).**

Plusieurs rebonds



Un des grands intérêt du lancer de rayons est que la gestion des surfaces réflechissantes se fait via un simple appel récursif. Il suffit d'ajouter à la couleur d'un point celle d'un rayon lancé depuis ce point dans la direction mirroir à celle d'arrivée (par rapport à la normale).
Il suffit donc de rappeler `Scene::raytrace()` pour obtenir le résultat. La couleur du rayon mirroir est pondérée par la BRDF évaluée dans la direction réféchie (`brdfReflect()`). Après un certain nombre de rebonds ou lorsque le rayon n'intersecte plus d'objets, il a une contribution nulle (couleur 0,0,0).

En revanche, lorsqu'un rayon partant de la caméra n'intersecte aucun objet, on souhaite lui donner la `mBackgroundColor` définie dans la scène. Pour différencier ces deux cas, ajouter par exemple un compteur du nombre de réflexions dans la classe `Ray`.



Facultatif : Pour vérifier votre algorithme, vous pouvez ajouter au raytracer un vecteur de `Segment`, qui va représenter un chemin lumineux. Un `Segment` est une structure qui contient les deux extrémités d'un segment ainsi que la normale au point d'arrivée.
La méthode `raytrace()` concatène dans le chemin le segment correspondant au `Ray` qu'elle est en train de traiter. Une méthode `drawRayPath` affiche le chemin lumineux ainsi stocké.
Conseils : un constructeur de `Segment` pourra prendre un `Ray` et un `Hit` en paramètres. Définissez une méthode `draw()` dans `Segment`. N'oubliez pas de vider le chemin lumineux au

départ du rayon. C'est `raytrace` et non plus `intersect` qu'il faut appeler dans le `select()` du viewer.