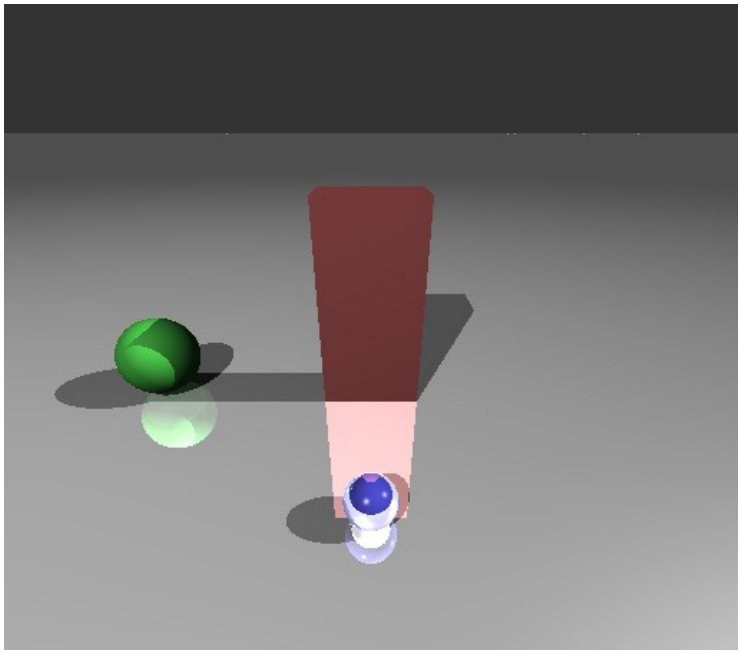
Lancer de rayons – TD2

1 Mise en route

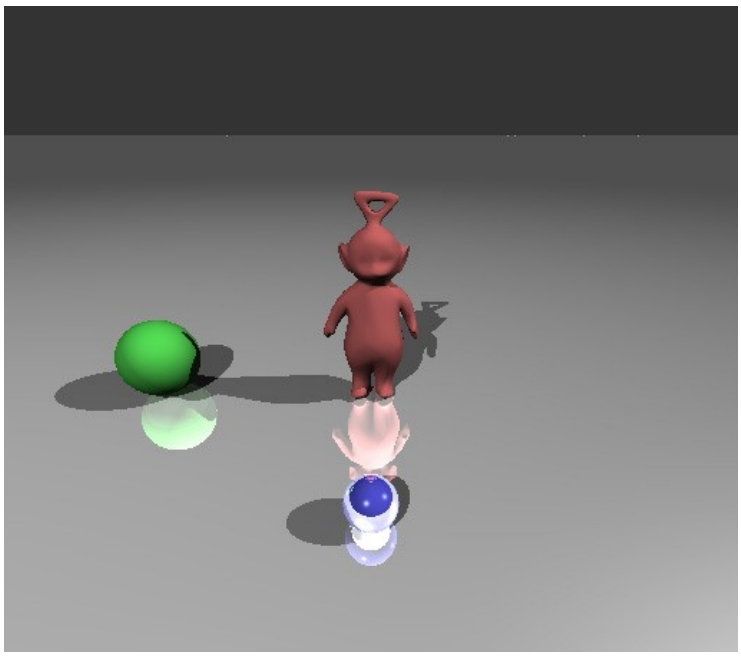
L'objectif de ce TD est de mettre en oeuvre une structure accélératrice pour le lancer de rayon. Plus précisément, vous implémenterez une hiérarchie de boîtes englobantes (BVH) pour accélérer le calcul des intersections avec un maillage.

Afin de commencer dans de bonnes conditions, télécharger l'archive sire_td2.zip qui contient une correction partielle du TD précédent, ainsi qu'un squelette de code pour la BVH (fichier "BVH.cpp"). Plus précisement, les classes `Sphere` et `Plane` (plan infini) possèdent un code d'intersection. Le code de lancer de rayon à proprement parlé est fourni dans les classes `Raytracing` et `Scene` avec gestion des transformations, des rayons d'ombres et des réfractions. La classe `PointLight` (pour représenter des sources de lumières ponctuelles) est également complétée ; les classes `Material` et `Frame` ont été modifiées pour un meilleur chargement des fichiers `.scn`. Enfin, le fichier "Ray.h" contient une fonction de calcul d'intersection avec une `Eigen::AlignedBox3d` (boîte alignée avec les axes). Le rendu de la scène construite par défaut devrait alors être :



2 Maillages triangulaires

Commencez par étudier le code d'intersection avec un plan infini. Le triangle est un peu plus complexe dans sa méthode `intersect`, mais reste simple à implémenter (référez vous au cours). Les deux faces d'un triangle jouent-elles le même rôle ? À vous de choisir.



Pour raytracer un maillage, il suffit ensuite d'utiliser les méthodes du triangle sur chacune de ses faces. La palette de scène s'en trouve alors bien élargie, au prix d'un temps de calcul croissant.

3 Bounding Volume Hierarchy (BVH)

Pour cette partie, il est utile de commencer par regarder la classe `Mesh`, et notamment d'observer la présence un attribut `mBVH` stockant une BVH. Remarquez aussi la fonction `Mesh::intersect()` qui utilise la BVH seulement si elle est construite, ainsi que la méthode publique `Mesh::buildBVH()` dont le rôle est de construire la BVH. Celle-ci sera appelée dans la fonction `Scene::createDefaultScene()` (la ligne de construction de la BVH est pour l'instant commentée).

Notez également les statistiques fournies sur la sortie standard (temps de calcul et nombre de calculs d'intersections avec des triangles). Voilà celles obtenues sur mon portable avec la scène par défaut :

```
Sans BVH:
  Raytracing time : 18.6058s  -  nb triangle intersection: 506724768

Avec BVH
  Raytracing time : 0.497817s  -  nb triangle intersection: 664163
```

Maintenant, c'est à vous de jouer. Vous avez seulement trois fonctions à implémenter : `BVH::buildNode()`, `BVH::intersect()` et `BVH::intersectNode()`. Ces fonctions sont largement commentées.

La méthode statique `Box::draw` vous permet de dessiner une boîte englobante en OpenGL (exemple d'utilisation dans `RenderingWidget::paintGL()` pour afficher celle des objets de la scènes en appuyant sur 'b'). Vous pouvez l'utiliser pour visualiser votre BVH.