Projet personnel en Informatique
Lyon Ray Tracer

Maxime Gaudin

maxime.gaudin@polymtl.ca
Institue National des Sciences Appliquées de Lyon (INSA)
École Polytechnique de Montréal

décembre 2011

Résumé

Le projet personnel nous permet de réaliser, sous la direction d'un professeur du département, un programme de recherche dans le domaine qui nous convient. C'était donc pour moi une opportunité de me lancer dans le développement d'un algorithme aussi puissant que mystérieux : Le *raytracing*. Et c'est en l'honneur de ma ville natale, Lyon (France), j'ai nommé ce programme *Lyon Ray Tracer*.

Ce rapport vous permettra d'avoir une vue d'ensemble sur le travail que j'ai effectué.

Chapitre 1

Introduction

Dans un monde où l'informatique et la virtualisation des environnements de travail est omniprésent, la visualisation des données est devenu un enjeux primordiale dans des domaines aussi variés que la médecine, l'architecture, l'industrie du jeux vidéo ou encore les interfaces utilisateur.

1.0.1 La *rasterisation*

Encore aujourd'hui, la méthode privilégiée de visualisation 3D est la *rasterisation*. Cette technique consiste à modéliser l'*univers*[1] par un ensemble de triangles. Ces triangles sont transformés (*e.g.* translatés, mis à l'échelle, *etc.*) puis projetés dans un monde plan : *notre écran*.

La *fig.1.1* représente grossièrement le cheminement des données jusqu'à l'image finale.

L'avantage de cette technique vient du fait que les calculs sont très aisément parallélisables. En effet, la même opération est répétée sur chaque sommet à tracer. Nous assistons d'ailleurs à un développement de la technologie dans ce sens : les Graphics Processing Units (GPUs) sont devenus de vraies collections de processeurs !

Hélas, cette technique n'offre pas que des avantages. En effet, la complexification des scènes et surtout la demande du publique pour des rendus encore plus réalistes poussent matériel et logiciel dans leur retranchement. Les ingénieurs doivent désormais rivaliser d'ingéniosité en créer des shaders de plus en plus compliqués et gourmands.

Heureusement, depuis plus de 30 ans sous l'impulsion de [Whi80], chercheurs et ingénieurs développent une technique de rendu conceptuellement simple mais très puissante : le *raytracing* !

1. Au sens mathématique, *i.e.* tout ce qui est contenu dans un système en n dimensions.

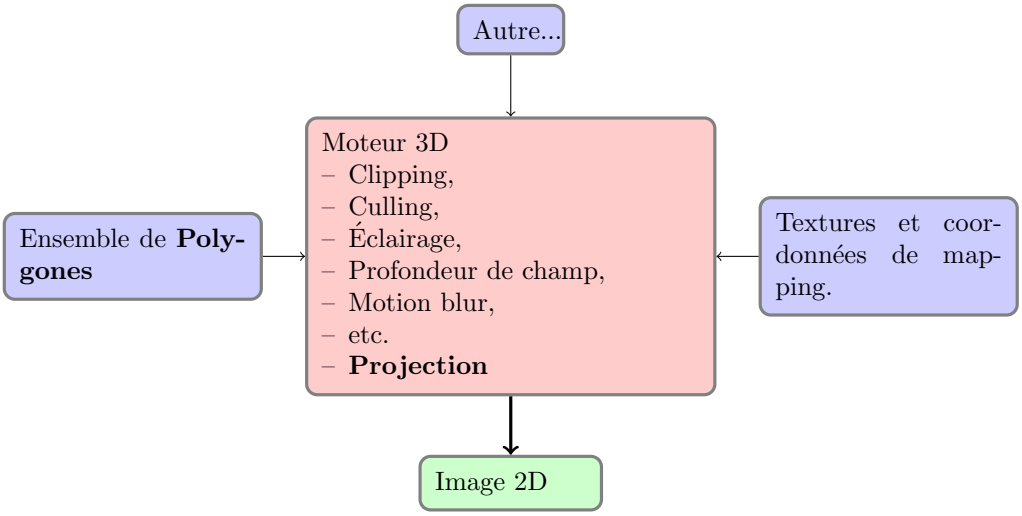


FIGURE 1.1 – Représentation grossière du pipeline de rendu via la rasterisation

1.1 Le *raytracing* ?

Le *raytracing* est une technique de rendu qui, à la manière des réseaux de neurones, tente de s'approcher au plus prêt de la réalité physique des phénomènes dans l'espoir d'obtenir un résultat.

À défaut de décrire le *pipeline* complet d'un *ray tracer* (ce n'est pas l'objectif du rapport), il m'a tout de même semblé important d'écrire une brève introduction sur ce qui pourrait bien être *la principale méthode de rendu dans un futur (très) proche.* [2]

1.1.1 Le modèle de propagation de la lumière

Comme son nom l'indique, l'idée du *raytracing* est de calculer le chemin que parcourent les rayons lumineux au travers d'une scène virtuelle.

Dans la réalité, l'œil humain perçoit les objets grâce à la lumière qu'ils réfléchissent. Nous pourrions donc imaginer calculer l'ensemble des interactions lumineuses d'une scène en partant de toutes les sources de lumière.

Un tel modèle supposerai cependant une puissance de calcul infinie et pour que les calculs restent raisonnables, les simplifications suivantes ont dû être faites :

1. **La lumière se propage en ligne droite :** Nous abandonnons donc le modèle ondulatoire de la lumière pour nous restreindre à l'optique

2. À ce sujet, je vous invite à consulter les travaux de NVidia et Intel sur le ray tracing.

géométrique. [3]

2. **La lumière que nous recevons est la même que si la source était notre œil :** En effet, plutôt que de lancer la majorité des rayons dans le vide (*i.e.* des rayons qui ne parviendraient pas jusqu'à notre œil), il est plus judicieux de lancer les rayons depuis la caméra jusqu'à notre scène — *C'est le Principe du retour inverse de la lumière de Fermat.*
3. **Le nombre de rebonds que fait la lumière est fini :** En optique géométrique, il n'y a pas de perte d'énergie et considérer tous les rebonds mènerai à des récursions infinies. [4]

1.1.2 L'algorithme simplifié :

Cette courte introduction nous amène à l'algorithme de rendu suivant :

Algorithme 1: getPixel

Entrée : Ray, rayon incident.
Entrée : L, le niveau de récursion, *i.e.* le nombre de fois que le rayon a été réfléchi ou réfracté.
Sortie : La couleur du pixel demandé.

```
1
2  FinalColor ← BLACK;
3  if L > MAX_RECURSION_LEVEL then
4      return FinalColor;
5
6  Record ← getClosestHit ( Ray ) ;
7
8  if Hit then
9      if La surface est réfléchissante then
10         getPixel ( Rayon réfléchi, L + 1 ) ;
11     if La surface est transparente then
12         getPixel ( Rayon transmis, L + 1 ) ;
13
14     FinalColor ← Couleur ambiente + Contribution Lumineuse ×
       (Couleur de l'objet + Couleur du rayon transmis + Couleur du rayon
       réfléchi) ;
15
16 return FinalColor ;
```

Il est impressionnant de constater la simplicité d'un algorithme pourtant si

3. Nous verrons tout de même dans la *sec. 3* qu'il est possible d'intégrer des phénomènes ondulatoires au rendu.

4. Comme pour les phénomène ondulatoire, nous verrons qu'il est possible de raffiner le modèle pour intégrer des pertes d'énergies (*Cf. sec. 3.3.1*).

puissant. En quelques lignes, nous venons de décrire la propagation de la lumière aux travers d'objets transparents et/ou réfléchissant !

Nous verrons cependant que, pour rester ouvert à l'extension et garder un couplage faible, une architecture robuste est nécessaire.

1.2 Contenu

Ce rapport est découpé en 4 parties.

D'abord, nous étudierons les objectifs du projet afin de bien cerner le domaine de définition du programme.

Puis, je tenterai de dresser l'état de l'art d'un domaine déjà très mature.

Aussi, nous étudierons l'architecture du programme ainsi que la logique de communication des différents modules.

Enfin, nous présenterons et analyserons les résultats obtenus.

Chapitre 2

Objectifs

Les objectifs du projet sont nombreux, aussi bien sur le plan de l'architecture logiciel, que de la gestion de projet, de la rédaction ou encore du développement.

Architecture logiciel - Un des principaux buts du projet (si ce n'est le but principal du projet) est de créer un programme ouvert à l'extension et pouvant servir de base à la création d'un *raytracer* plus évolué, plus rapide, ou plus facile à utiliser.

L'architecture du programme est défini à la section 4.

Portabilité - Afin d'être le plus portable possible, j'ai décidé d'utiliser un ensemble d'outils standards et libres :

- *Le langage C++ : Open-Spec* et compilable sur presque toutes les plateformes.
- *Les autotools*[1] Système de build standard et ne nécessitant aucun autre programme que make et bash. De plus, il assure une portabilité maximale en vérifiant la présence de toutes les bibliothèques nécessaire à la compilation et au bon fonctionnement du programme.
- *Boost :* La bibliothèque C++ de référence.
- *libpng/libjpg :* Pour l'import texture et l'export des résultats.
- *lib3ds :* Pour l'importation des modèles 3D.
- *LaTeX :* Pour la rédaction du rapport.

- *Tim Horton :* Pour le café et les beignets.

Gestion de projet - J'ai réalisé ce projet seul du 24 Septembre au 1 Décembre en utilisant un style de développement Extrem Programming (XP)[2]. Le gestionnaire de version utilisé est *git* et le projet ainsi que son site web sont hébergés

1. http://sources.redhat.com/autobook/ :
2. http://www.extremeprogramming.org/

sur GitHub.

> ***Remarque :*** *Parce que la charge de travail aurait été trop grande, j'ai décidé de ne pas rédiger les documents de références du type Dossier d'initialiation, Dossier de jalonnage, Planning prévisionnel, etc.J'ai cependant pris soin de rédiger une documentation exhaustive de chacune des classes du projet.*

Implémentation - Il était pour moi primordial de respecter tout au long de ce projet des *guidelines* afin d'assurer une cohérence de tout le code (plusieurs milliers de lignes tout de même). J'ai pour cela suivit principalement le guide de style définit par Herb Sutter dans [Sut05]. Je me suis aussi inspiré de [Mey94].

Enfin, la documentation fait aussi partie intégrante du processus de développement et la génération de celle-ci fait partie du *pipeline* de compilation.

Fonctionnalité du ray tracer - Reprenons les objectifs fixés dans le sujet :

1. ✓ Un moteur de *raytracing* "classique".
2. ✓ La gestion de primitives (sphères, plan, boite, triangles).
3. ✓ La gestion de plusieurs types de caméras : Orthographique, perspective, *etc.*
4. ✓ La gestion de plusieurs types de lumières : Point, plane, directionnelle, *etc.*
5. ✓ La gestion d'au moins un format de représentation polygonale.
6. ✓ La mise en place des structures accélératrices.
7. ✗ La gestion du *photon mapping* : Comme je l'avais pressentie dans mon sujet, le temps alloué au projet était insuffisant pour l'implémentation d'une telle fonctionnalité.

J'ai cependant pris le temps d'implémenter en plus :

1. Un chargeur de scènes XML.
2. La lecture et l'écriture d'images au format PNG ou JPG (avec la possibilité d'ajouter d'autre formats).
3. La gestion des textures.
4. Le supersampling.
5. La gestion des ombres douces.
6. La gestion de la profondeur de champ.

2.0.1 Non-objectifs

Comme dans tout projet, celui-ci possède des non-objectifs, *i.e.* des objectifs que j'ai volontairement choisi de ne pas atteindre, ou en tout cas de ne pas viser.

D'abord, concernant la vitesse d'exécution. Les *raytracers* commerciaux sont de réelles prouesses d'ingénierie et d'optimisation et j'ai considéré que l'optimisation de mon programme sortait du contexte d'un projet personnel.

Il ne s'agissait pas non plus d'en faire un produit fini mais plutôt un *proof of concept*. Il reste donc de nombreuses améliorations tant au niveaux des fonctionnalités existantes qu'au niveau de celles à ajouter.

Chapitre 3

État de l'art

Il est hélas impossible de réaliser un état de l'art complet sur le *raytracing* car cela nécessiterais des mois de recherches et mériterait un papier complet [1]. Nous pouvons cependant retracer les grandes dates de cette technologie et tenter d'en apercevoir les perspectives.

> ***Remarque :*** *Cette étude est principalement basé sur [Car03] et [Cha06].*

3.1 Découverte

C'est en 1968 (!!!) que Arthur et LaFortune publient [LW93] dans lequel ils abordent la possibilité d'effectuer un rendu en calculant l'intersection des objets avec la lumière — en l'occurrence un ensemble de rayons virtuels.

Le *ray casting* est né mais il faudra attendre un dizaine d'années avant que soit publié un algorithme capable de surpasser la *rasterization* (*Cf.* 1.0.1) en terme de qualité. En effet, le *ray casting* se contente de lancer un rayon primaire afin de déterminer s'il y a intersection ou non sans pour autant pousser la calcul plus loin.

3.2 La révolution

Le *raytracing* tel que nous le connaissons aujourd'hui fait ses premières armes dans [Whi80]. Les auteurs y introduisent la notion de rayons secondaires crées à chaque intersection avec un objet. Ainsi apparait la possibilité d'effectuer le rendu de phénomènes tels que la réflexion et la réfraction. Whitted souligne par ailleurs que la gestion de l'anti-aliasing est totalement intégré au pipeline de rendu. C'est une caractéristique importante car aujourd'hui encore, les techniques d'anti-aliasing "coûtent chère en calcul".

1. À ce sujet, je vous invite à consulter [WS01]

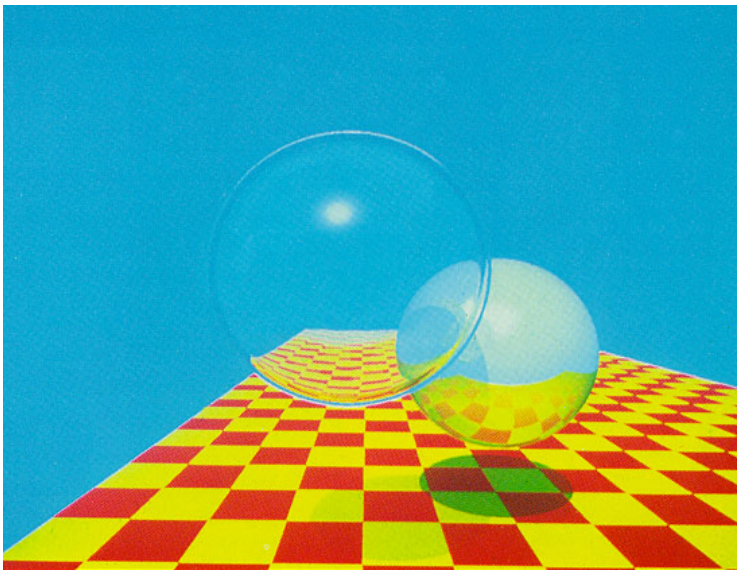


FIGURE 3.1 – Un rendu de [Whi80]

La *fig.* 3.1 présente un des rendus de [Whi80] où nous pouvons constater que déjà à l'époque, la qualité du *raytracing* était incomparable à celle de la rasterisation.

Très vite après cette publication, de nombreuses améliorations ont été apportées à l'algorithme. On peut citer notamment les ombres floues (ou douces), le jittering, le motion blur ou encore l'adaptative sampling.

3.3 Bi-directional path tracing

La seconde révolution du domaine tire ses origine dans l'avènement du *bi-directionnal path tracing*. Le terme fait référence au fait qu'au lieux de simplement considérer les rayons partant de la caméra pour aller jusqu'à la scène (*backward rays*), l'algorithme considère une équation intégrant les rayons partant des différentes sources de lumière (qui, nous le verrons, peuvent être tous les objets de la scène). C'est aujourd'hui la plus grande piste de recherche pour améliorer le rendu de scènes complexes et si la théorie est déjà très évoluée, de gros travaux d'optimisation restent à effectuer.

Étudions brièvement 2 des approches les plus utilisées.

3.3.1 Radiosity

La radiosité est une méthode de résolution de l'équation 3.2 (à titre d'illustration...).

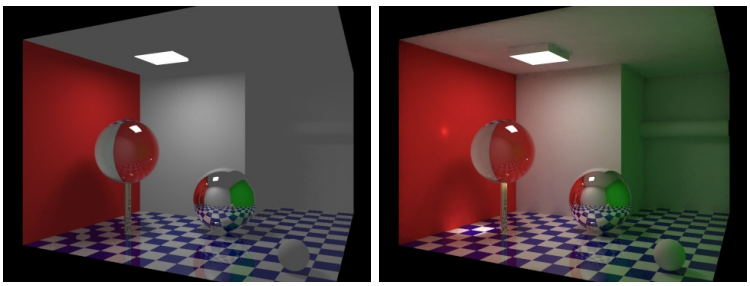
$$B(x)\,dA = E(x)\,dA + \rho(x)\,dA \int_S B(x')\frac{1}{\pi r^2}\cos\theta_x \cos\theta_{x'} \cdot \mathrm{Vis}(x,x')\,\mathrm{d}A'$$

où
– $B(x)dA$ est l'énergie totale de la zone en question,
– $E(x)$ son énergie émise,
– $\rho(x)$ sa réflectivité,
– r la distance entre x et x',
– Vis une fonction binaire valant 1 si x et visible de x' et 0 sinon.

FIGURE 3.2 – Équation de transfert de lumière.[Wik11]

Cette équation, tirée des transfert de chaleurs, s'applique très bien aux transferts de lumière d'une surface à une autre. Les *fig.3.3(a) et 3.3(b)* montrent bien la différence que l'illumination globale apporte au réalisme de la scène.

Alors que sans illumination globale, les murs gardent leur teinte d'origine, l'ajout de l'intégration de l'algorithme de radiosité transfert de la couleur du mur vert (invisible sur l'image) sur le mur du fond.



(a) CornelBox sans illumination globale.

(b) CornelBox avec illumination globale.

Plus concrètement, l'algorithme se contente de lancer des rayons depuis les différentes sources lumineuses et calcule les rebonds de ceux-ci. Évidemment, l'émission des rayons n'est pas totalement aléatoire et suit des lois de distributions. À chaque nouvelle intersection, la contribution de la source est ajoutée à l'image.

> ***Remarque :*** *Cette méthode à l'avantage d'être relativement simple à implémenter.*

3.3.2 Photon mapping

Une autre méthode très efficace d'illumination globale est le *photon mapping*. Contrairement à la radiosité qui ajoute la contribution des rayons par itération,

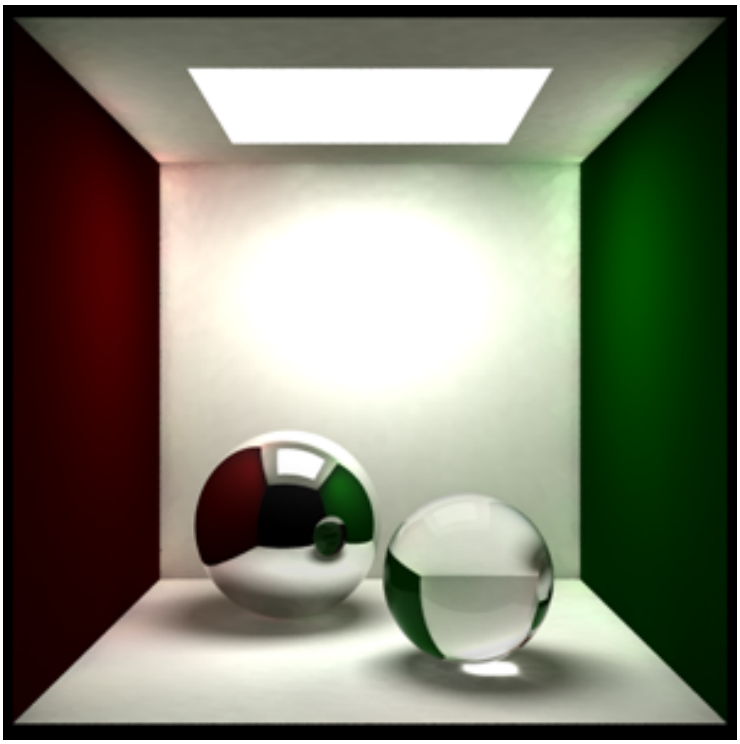


FIGURE 3.3 – Un exemple de caustique (sous la sphère de droite).

le *photon mapping* crée une cartographie des photons lancés depuis la source lumineuse jusqu'aux objets de la scène. À chaque collision, le photon est soit absorbé, soit réfléchi, soit les deux.

Cette technique permet de reproduire le phénomène de caustique, qui est une concentration de photons dans une zone de l'espace comme le montre la *fig.3.3*.

Cette méthode est cependant plus difficile à implémenter puisqu'elle nécessite des structures accélératrice comme les Kd-Tree afin de pouvoir "compter" le nombre de photons dans une certaine zone de l'espace de manière efficace.

> ***Remarque :*** *Ces deux approches forment la base des techniques utilisées mais évidemment, il existe de nombreuses variantes, que ce soit au niveau des structures de donnée, des distribution statistique, des équations de rendu, etc.dont la plus important est le final gathering.*

3.3.3 Demain ?

Si l'avenir du *raytracing* dans l'informatique de tous les jours est certain, il faudra attendre que les CPU puissent calculer efficacement les images pour voir cette technique remplacer la rasterisation.. Cela nécessite donc du nouveau matériel et des algorithmes encore plus performant.

Il est cependant déjà possible de faire du *raytracing* temps réel sur des GPUs "de bureau" grâce à des projets comme *OpenRT* ou *Optix*.

Chapitre 4

Architecture

Après avoir fait un tour d'horizon des différents modules, je vous propose d'étudier l'architecture complète du programme.

> ***Remarque :*** *Bien qu'il ne s'agisse pas vraiment de modules (ce concept n'existe pas en C++), il existe tout de même des groupements de classes qu'il soit sémantique ou fonctionnel. C'est ce que j'assimilerai comme définition pour "module".*

4.1 Camera

L'ensemble des caméras doivent respecter l'interface de la *fig. 4.1*. Concrètement, chaque caméra doit pouvoir, à partir des coordonnées de l'espace écran (coordonnées UV), fournir l'ensemble des rayons nécessaires au dessin de la scène.

4.1.1 Spécialisations

[**Implémentée**] **Caméra perspective** Une caméra simple avec gestion de la perspective.

[**Implémentée**] **Caméra perspective DOF** Une caméra avec gestion de la perspective et rendu de l'effet de profondeur de champs (*Cf. section 5.9*).

[**Non implémentée**] **Caméra orthographique** Caméra avec projection orthographique (utiles pour les rendu de pièce en 3D et le métrage).

[**Non implémentée**] **Caméra fish eye** Cette caméra permet d'utiliser des matrices de projection grand angle.

```cpp
/**
  * @file Camera.hpp
  * @author Maxime Gaudin
  * @date 2011
  *
  * Cette classe est l'interface de toutes les caméras.
  * Les coordonnées utilisées dans cette classe (et donc toutes les classes qui
  * en héritent) doivent utiliser le système de coordonnées UV qui est définit
  * comme suit :
  * (0,1)----------(1,1)
  *   |              |
  *   |              |
  *   |              |
  *   |              |
  *   |              |
  * (0,0)----------(1,0)
  */
#ifndef CAMERA_H_
#define CAMERA_H_
#include <Buildable.hpp>

#include <vector>
#include <Ray.hpp>

class Camera : public Buildable {
  public:
    /**
      * @return L'ensemble des rayons associés à la coordonnée (u,v)
      */
    virtual std::vector<Ray> getRay ( double u, double v ) const = 0;
};
#endif // CAMERA_H_
```

FIGURE 4.1 – Code de l'interface commune à toutes les caméras

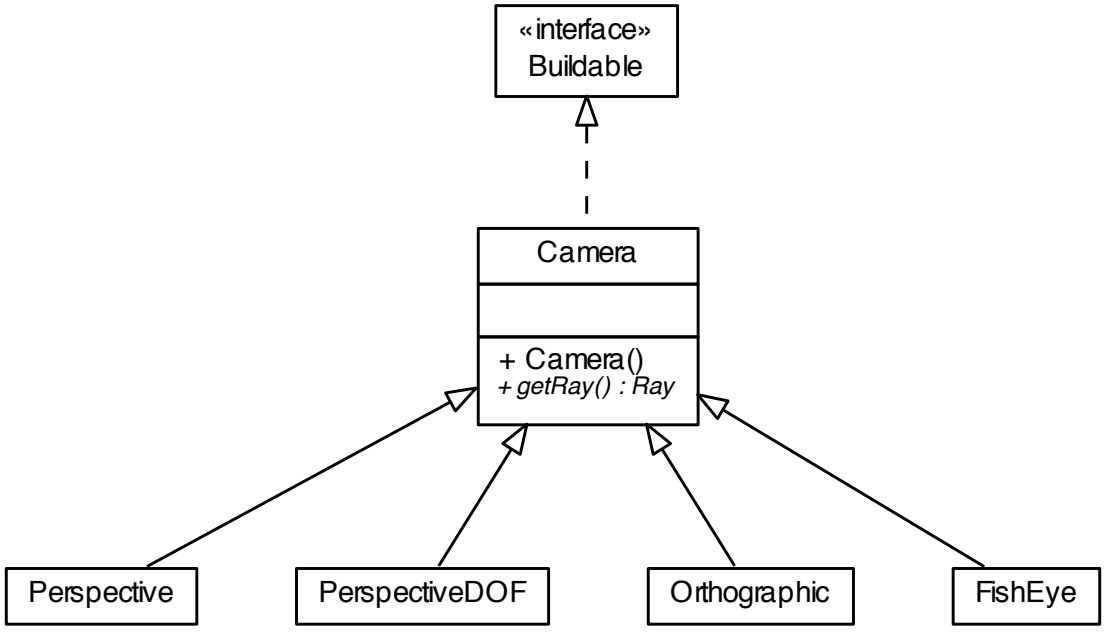


FIGURE 4.2 – Diagramme de classe du module Caméra

4.2 Sampler

Les *samplers* ont pour objectif d'échantillonner le nombre de rayons lancés pour chaque pixel. Pour cela, il est nécessaire de leur fournir toute la scène car ils nécessitent au moins la caméra mais peuvent avoir besoin de beaucoup d'autre éléments (par exemple, l'ensemble des géométries). Il est par exemple possible de *supersampler*, *i.e.* lancer plus de rayons qu'il ne semble nécessaire afin d'éviter les phénomènes de crénelage.

Encore une fois, les *samplers* doivent respecter la même interface définit à la *fig.* **??**.

```
/**
  * @file Sampler.hpp
  * @author Maxime Gaudin
  * @date 2011
  *
  * Les samplers ont pour objectif d'échantillonner le nombre de rayons
  * lancés pour chaque pixel.
  */
#ifndef SAMPLER_H_
#define SAMPLER_H_
#include <vector>

#include <Ray.hpp>
#include <Scene.hpp>

class Sampler {
  public:
    /**
      * @return Ensemble des rayons à lancer pour la scène et les
      * coordonnées passés en paramètres.
      */
    virtual std::vector<Ray> getRays ( Scene const& scene,
        unsigned int X, unsigned int Y ) = 0;
};
#endif
```

FIGURE 4.3 – Code de l'interface commune à tous les samplers

4.2.1 Spécialisations

[**Implémentée**] **DefaultSampler** Ce *sampler* lance un seul rayon par pixel.

[**Implémentée**] **SuperSampler** Ce *sampler* lance N rayons par pixel où N est passé en paramètre.

[**Non implémentée**] **Adaptative sampling** Il existe un *sampler* que je n'ai pas eu le temps d'implémenter et qui pourtant définit la stratégie la plus intelligente d'échantillonnage. Il s'agit de l'échantillonnage adaptatif qui, tant que la moyenne des pixels environnant est supérieure à un certain seuil de différence, lance des rayons selon une distribution statistique.

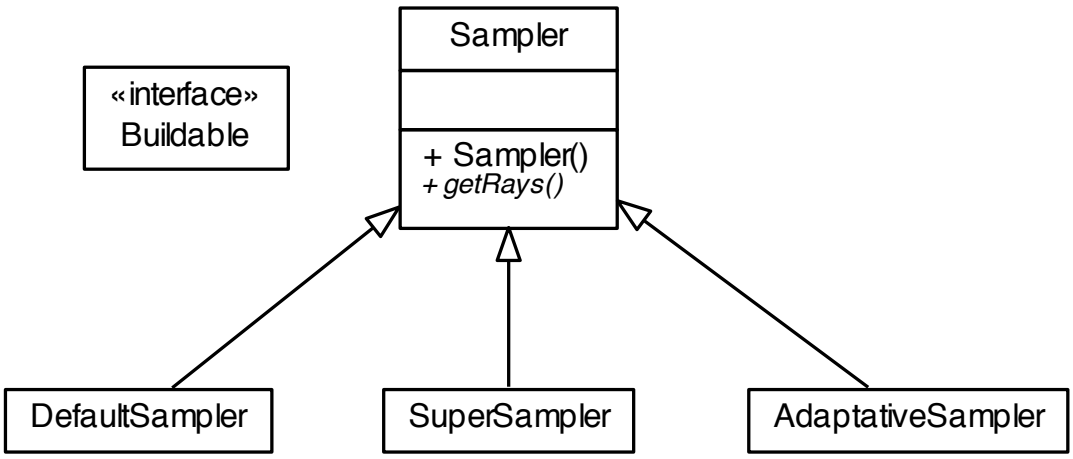


FIGURE 4.4 – Diagramme de classe du module Sampler

4.3 Lumières

L'ensemble des sources de lumières doivent respecter l'interface de la *fig.4.5*.

4.3.1 Spécialisations

[**Implémentée**] **Lumière directionnelle** Cette source éclaire tout selon une direction passée en paramètre. Elle simule une source très lointaine dont tous les rayons sont parallèles.

[**Implémentée**] **Lumière ponctuelle** Cette source définie par sa position, rayonne autour d'elle comme une point lumineuse.

[**Implémentée**] **Lumière surfacique plane** Cette source a la particularité d'avoir une composante surfacique qui permet de générer des ombres douces (*Cf. section 5.8*).

[**Non implémentée**] **Cone de lumière**

[**Non implémentée**] **Autre lumières surfaciques** Il est possible de rajouter tout les types de lumières surfaciques comme des sphères lumineuses et même des *mesh* lumineux.

```cpp
/**
  * @file Light.hpp
  * @author Maxime Gaudin
  * @date 2011
  *
  * Interface de toutes les lumières.
  */
#ifndef LIGHT_H_
#define LIGHT_H_
#include <Buildable.hpp>
#include <common.hpp>

#include <vector>

#include <HitRecord.hpp>
#include <Color.hpp>

class Camera;
class Material;
class Geometry;

class Light : public Buildable {
  public:
    Light () {}

    /**
      * @param material Matériau à associer à la lumière.
      */
    Light (Material* material) : material_(material) {}

  public:
    /**
      * @return La contribution, c'est à dire l'ajout de lumière de la
      * source à la lumière totale du point considéré.
      *
      * @param camera La caméra à considérer.
      * @param geometries Listes des géométries de la scène.
      * @param record Enregistrement de l'intersection.
      */
    virtual Color_d getContribution (
        Camera* camera,
        std::vector<Geometry*> const& geometries,
        HitRecord const& record ) const = 0;

  protected:
    Material* material_;
};
#endif // LIGHT_H_
```


FIGURE 4.5 – Code de l'interface commune à toutes les sources de lumières

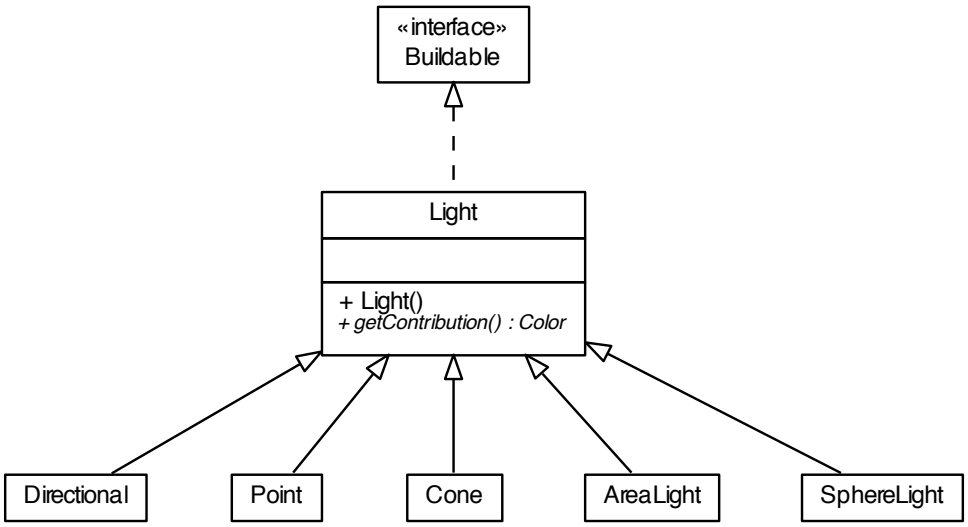


FIGURE 4.6 – Diagramme de classe du module Light

4.4 Geometry

Ce module regroupe l'ensemble des géométries que le programme peut calculer. Chacune de ces classes doivent respecter l'interface suivante :

```
/**
  * @file Geometry.hpp
  * @author Maxime Gaudin.
  * @date 2011
  *
  * Cette classe est l'interface de tous les objets dont on peut calculer
  * l'intersection avec un ray
  */
#ifndef GEOMETRY_H_
#define GEOMETRY_H_
#include <Buildable.hpp>

#include <HitRecord.hpp>
#include <Ray.hpp>
#include <Material.hpp>
#include <Maths.hpp>

class Geometry : public Buildable {
  public:
    Geometry ( );

    /**
      * @param material Matériaux lié à la géométrie
      */
    Geometry ( Material* material );

    /**
      * @param material Matériaux lié à la géométrie.
      * @param translation Translation à appliquer à la géométrie.
      * @param rotation Rotation à appliquer à la géométrie.
      * @param scale Mise à l'échelle à appliquer à la géométrie.
      */
    Geometry (
        Material* material,
        Vector3d const& translation,
        Vector3d const& rotation,
        Vector3d const& scale );

  public:
    /**
      * @return L'enregistrement lié à la collision du rayon @a ray et
```

```
     * la géométrie.
     *
     * @param ray Rayon à intersecter.
     */
    virtual HitRecord getRecord ( Ray const& ray ) const = 0 ;

    /**
     * @return La coordonnées de texture correspondant à l'intersection
     * enregistrée
     * @param record Enregistrement de l'intersection.
     */
    virtual Vector<double, 2> getUVFromHit ( HitRecord const& record ) const;

  public:
    /**
     * @return Le matériau de la géométrie.
     */
    Material* material() const { return material_; }

  protected:
    Material* material_;

    Vector3d translation_;
    Vector3d rotation_;
    Vector3d scale_;
    Matrix<double, 4, 4> transformation_;
};
#endif // GEOMETRY_H_
```

4.4.1 Spécialisations

[Implémentée] Sphère

[Implémentée] Plane

[Implémentée] Box Cette géométrie est uniquement utilisée pour la structure d'Octree.

[Implémentée] Triangle

[Implémentée] Mesh Qui est en réalité un agrégat de triangles stockés dans un *octree*.

[Non implémentée] Quadric Les *quadrics* sont un ensemble de géométries paramétrés par une équation de la forme

$$xQx^T + Px^T + R = 0$$

.

Cette équation permet de décrire toute une gamme de formes allant de l'ellipse à la sphère en passant par plusieurs types de cônes. [1].

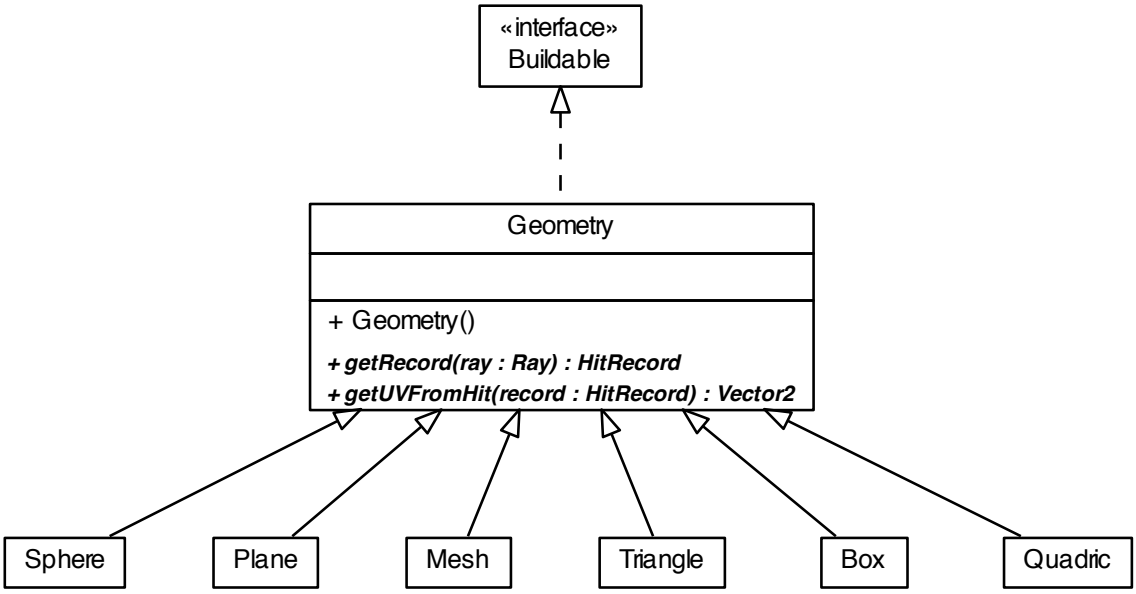


FIGURE 4.7 – Diagramme de classe du module Geometry

1. Pour la liste complète, je vous invite à consulter http://en.wikipedia.org/wiki/Quadric.

4.5 Builder

Chaque objet constructible doit aussi définir un *builder*, *i.e.* une classe permettant à partir de la description textuelle de l'objet (*Cf.* section **??**), de construire sa représentation mémoire.

L'interface à respecter est celle présentée à la *fig.* 4.8.

4.5.1 Spécialisations

Un pour chaque objet constructible. On peut citer `MaterialBuilder`, `SphereBuilder`, `DirectionalBuilder`, *etc.*

```
/**
  * @file Builder.hpp
  * @author Maxime Gaudin
  * @date 2011
  *
  * Ce fichier déclare la classe abstraite de tous les constructeurs d'objets
  * (d'interface Buildable).
  * C'est ces builders qui seronts passés au SceneReader.
  */
#ifndef BUILDER_H_
#define BUILDER_H_
#include <string>
#include <boost/property_tree/ptree.hpp>

class Buildable;
class Material;

class Builder {
  public:

    /**
      * @param ID Identifiant du type d'objet construit.
      */
    Builder ( std::string const& ID );

  public:
    /**
      * @param pt L'arbre DOM contenant les paramètres du fichier de
      * configuration de la scène,
      * @param material Matériau associé.
      *
      * @return Un pointeur sur l'objet construit.
      */
    virtual Buildable* getObject(
        boost::property_tree::ptree pt,
        Material* material ) const = 0;

  public:
    std::string getID () const;

  protected:
    std::string ID_;
};
#endif // BUILDER_H_
```

FIGURE 4.8 – Code de l'interface commune à tous les *builders*

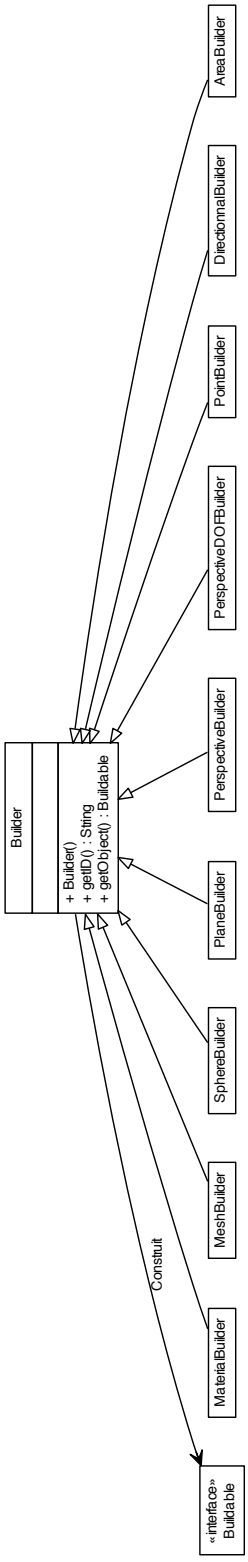


FIGURE 4.9 – Diagramme de classe du module Sampler

4.6 Scene

Pour être construite, la scène doit lire un fichier au format XML (*Cf. section* ??).

Une scène contient les champs suivant :
- Une caméra
- Une liste de matériaux
- Une liste de géométries
- Une liste de lumière.
- Une couleur ambiante,*i.e.* la “couleur du vide”.
- L’image où sera écrite le rendu.

Lors de la création, le programme lit le fichier XML et appelle successivement tous les *builders* disponible afin d’en trouver un capable de construire la représentation mémoire de l’objet.

4.7 Image

La manipulation des images est évidemment au cœur du programme. Il doit être capable de lire (pour les textures) et d'écrire (pour le rendu) plusieurs formats d'image.

Pour cela, l'architecture du module est basé autours d'un *Builder* (*Cf.* [GHJV94], *p. 97*) dont une seule instance est autorisée (*design pattern Singleton, p. 127*). Ainsi, il est possible de créer une collection de classes capables de manipuler différents formats d'images. Par la suite, le *Builder* pourra choisir la bonne implémentation des méthodes de chargement et de sauvegarde de l'image correspondant au format demandé.

4.7.1 Spécialisations

[**Implémentée**] **PNGHandler** Permet de manipuler le format PNG.

[**Implémentée**] **JPGHandler** Permet de manipuler le format JPG.

[**Non implémentée**] **Autre** Il existe plusieurs dizaines de formats qu'il faudrait supporter.

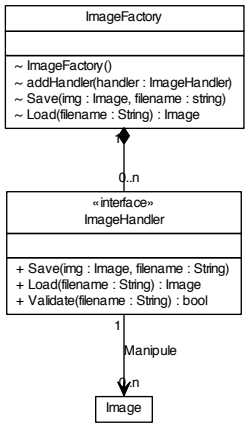
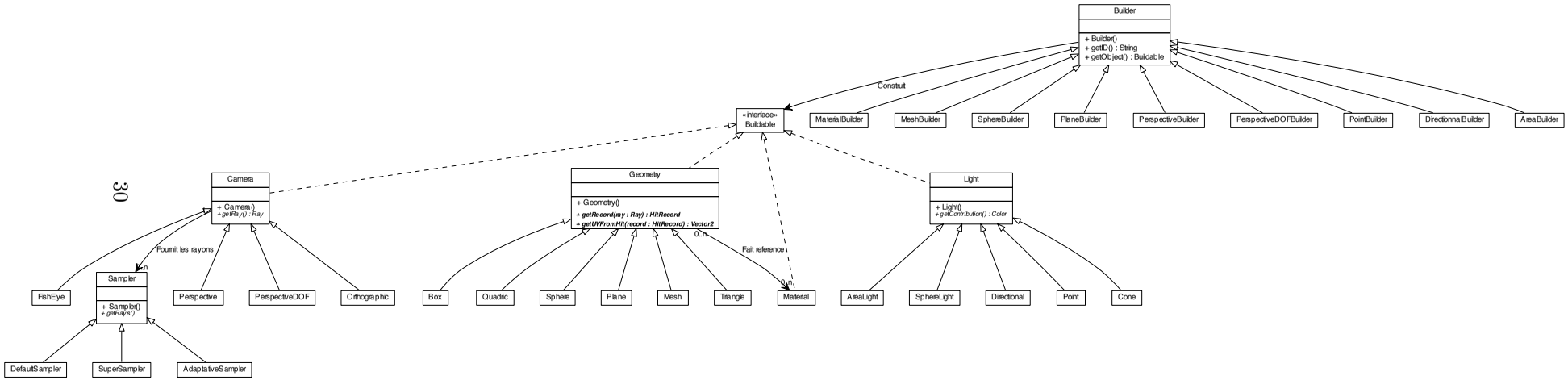


FIGURE 4.10 – Diagramme de classe du module Image

4.8 Architecture complète

La figure **??** présente l'architecture des principaux modules du programme. Malheureusement, il est impossible de présenter sous forme intelligible l'ensemble des interactions et pour cette raison, je vous invite à consulter la documentation *Doxygen* qui parvient bien mieux que moi à mettre en valeurs les relations entres les classes.


Builder
+ Builder()
+ getID() : String
+ getObject() : Buildable
Construit
«interface»
Buildable
MaterialBuilder
MeshBuilder
SphereBuilder
PlaneBuilder
PerspectiveBuilder
PerspectiveDOFBuilder
PointBuilder
DirectionnalBuilder
AreaBuilder
Camera
+ Camera()
+ getRay() : Ray
Fournit les rayons
1..n
Sampler
+ Sampler()
+ getRays()
FishEye
Perspective
PerspectiveDOF
Orthographic
DefaultSampler
SuperSampler
AdaptativeSampler
Geometry
+ Geometry()
+ getRecord(ray : Ray) : HitRecord
+ getUVFromHit(record : HitRecord) : Vector2
0..n
Fait reference
1
Box
Quadric
Sphere
Plane
Mesh
Triangle
Material
Light
+ Light()
+ getContribution() : Color
AreaLight
SphereLight
Directional
Point
Cone

Chapitre 5

Résultats & Analyse

Afin que le rapport reste dans des proportions raisonnables, je n'aborderai pas tous les résultats que j'ai obtenu. Je vous invite donc à "jouer" avec le programme et essayer les quelques scènes d'exemples.

5.1 Gestionnaire de scènes

Un des élément principaux, il permet grâce à une description textuelle, de générer la représentation mémoire de la scène. C'est par conséquent l'interface de communication entre l'utilisateur et le client.

La démocratisation et la facilité du langage XML m'ont poussé à le choisir comme métalangage de description. De plus, la disponibilité d'outils pour le *parser* rend son utilisation beaucoup plus simple [1].

Enfin, il augmente l'interopérabilité avec d'autre logiciel comme les modeleurs par exemple.

5.1.1 Exemple

La code suivant est un exemple simple de scène mettant en jeu 3 sphères, 1 plan et une caméra avec gestion de la profondeur de champ :

```
<scene width="600" height="600">

  <camera type="perspectiveDOF">
    <eye><vector3 X="0" Y="0" Z="8" /></eye>
    <lookat><vector3 X="0" Y="0" Z="0" /></lookat>
    <up><vector3 X="0" Y="1" Z="0" /></up>
    <sampling>128</sampling>
    <aperture>0.9</aperture>
    <focusPoint><vector3 X="0" Y="0" Z="0" /></focusPoint>
```

1. Dans mon cas, la *parsing* est assuré par *Boost* ce qui le rend très solide.

```
</camera>

<materials>
  <material name="red">
    <diffuse><color R="1" G="0" B="0" /></diffuse>
  </material>

  <material name="orange">
    <diffuse><color R="1" G="1" B="0" /></diffuse>
  </material>

  <material name="green">
    <diffuse><color R="0" G="1" B="0" /></diffuse>
  </material>

  <material name="blue">
    <diffuse><color R="0" G="0" B="1" /></diffuse>
    <reflexivity>0.9</reflexivity>
  </material>

</materials>

<lights>
  <light type="point">
    <position><vector3 X="-20" Y="20" Z="20" /></position>
  </light>
</lights>

<geometries>
 <geometry type="sphere" material="red">
   <centre><vector3 X="0" Y="0" Z="0" /></centre>
   <radius>1</radius>
 </geometry>

 <geometry type="sphere" material="green">
   <centre><vector3 X="-1" Y="0" Z="4" /></centre>
   <radius>1</radius>
 </geometry>

 <geometry type="sphere" material="orange">
   <centre><vector3 X="2" Y="0" Z="-4" /></centre>
   <radius>1</radius>
 </geometry>

 <geometry type="plane" material="blue">
   <point><vector3 X="0" Y="-1.2" Z="0" /></point>
```

```
    <normal><vector3 X="0" Y="1" Z="0" /></normal>
   </geometry>

  </geometries>


</scene>
```

Comme ci-dessus, une description de scène est composée de 4 blocs :

1. Un nœud `camera` : C'est ici qu'est déclaré l'*unique* caméra de la scène.
2. Un nœud `materials` : C'est ici que doivent être déclaré l'ensemble des matériaux identifiés par une chaine de caractère *unique*.
3. Un nœud `lights`.
4. Un nœud `geometries`.

Remarque : *L'ordre n'est pas imposé.*

Comme précisé dans l'architecture, c'est chaque *builder* qui décrit les paramètres dont il a besoin pour fonctionner. Certains sont obligatoires, d'autres optionnels.

5.1.2 Améliorations

Gestion de la casse Pour l'instant, la détection des paramètres est *case-sensitive*, c'est inutile et source d'erreurs difficiles à débusquer.

Auto-description Chaque *builder* devrait pouvoir se décrire via la ligne de commande en expliquant quels sont les paramètres dont il a besoin et leurs domaines de définitions.

Ces deux améliorations sont relativement simples à implémenter et c'est principalement le développement d'autres fonctionnalité et le manque de temps qui m'ont empêcher de les ajouter au programme.

Il faut tout de même noter que la seconde amélioration n'avait pas été prévu et qu'il faudra par conséquent changer l'interface des *builder*, ce qui impacte l'ensemble de code existant. C'est donc un oubli très gênant de ma part.

5.1.3 Bug connu

Aucun.

5.2 Journalisation

Pour un programme d'une telle ampleur, il est important de pouvoir retracer le l'exécution sans avoir à utiliser un debugger. C'est pourquoi j'ai choisi de me tourner vers une gestion centralisée et systématique des logs.

5.2.1 Exemple :

Rendu de la scène `DOF.lrt`

```
[Core] - Builders discovering...
[SceneReader] - 'material' builder added.
[SceneReader] - 'perspective' builder added.
[SceneReader] - 'perspectiveDOF' builder added.
[SceneReader] - 'point' builder added.
[SceneReader] - 'area' builder added.
[SceneReader] - 'sphere' builder added.
[SceneReader] - 'plane' builder added.
[SceneReader] - 'mesh' builder added.
[Core] - Building scene...
[SceneReader] - Reading scene : scenes/DOF.lrt ...
[SceneReader] - Image...
[Image] - Building new output frame (600, 600)... OK
[SceneReader] - Camera...
[PerspectiveDOF] - New Camera : Eye - ( 0 ,0 ,8 ), LookAt - ( 0 ,0 ,0 ),
Direction ( 0 ,0 ,-1 )
[SceneReader] - Materials...
[MaterialBuilder] - New material with diffuse = [ R=1, G=0, B=0 ].
[MaterialBuilder] - New material with diffuse = [ R=1, G=1, B=0 ].
[MaterialBuilder] - New material with diffuse = [ R=0, G=1, B=0 ].
[MaterialBuilder] - New material with diffuse = [ R=0, G=0, B=1 ].
[SceneReader] - Lights...
[SphereBuilder] - New point light at ( -20 ,20 ,20 )
[SceneReader] - Geometry...
[SphereBuilder] - New sphere at ( 0 ,0 ,0 ) (radius = 1)
[SphereBuilder] - New sphere at ( -1 ,0 ,4 ) (radius = 1)
[SphereBuilder] - New sphere at ( 2 ,0 ,-4 ) (radius = 1)
[PlaneBuilder] - New plane at ( 0 ,-1.2 ,0 ) (normal= ( 0 ,1 ,0 ))
[Core] - Rendering...

0%..........................................................
10%..........................................................
20%..........................................................
...
100%..........................................................
[Core] - Saving...
```

```
[Core] - Cleanup...
```

5.2.2 Améliorations

Gestion des niveaux d'alerte Alors que certaines informations peuvent être utiles à l'utilisateur, d'autres sont purement fonctionnelles et permettent un débugage plus rapide. Il serait donc judicieux de pouvoir affecter des niveaux aux événements de journalisation dans le but de pouvoir les filtrer.

5.2.3 Bug connu

Aucun.

5.3 Texture

Le plaquage des textures est intimement lié à la géométrie en question. Le moteur ne fait donc que très peu de travail comparé à la géométrie qui doit projeter elle même la texture et donner, en fonction des informations de collisions, la couleur au point demandé.

> ***Remarque :*** *Comme nous pouvons le voir dans le description de l'architecture, la gestion des différents formats d'image n'est pas à la charge de la géométrie.*

5.3.1 Exemple

La *fig. 5.1* présente un exemple de plaquage de texture (ici, une grille en metal) sur une sphère.

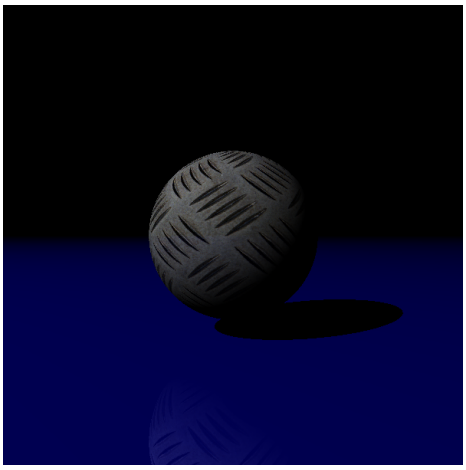


FIGURE 5.1 – Un exemple de rendu avec plaquage de texture sur une sphère.

5.3.2 Amélioration & bug connu

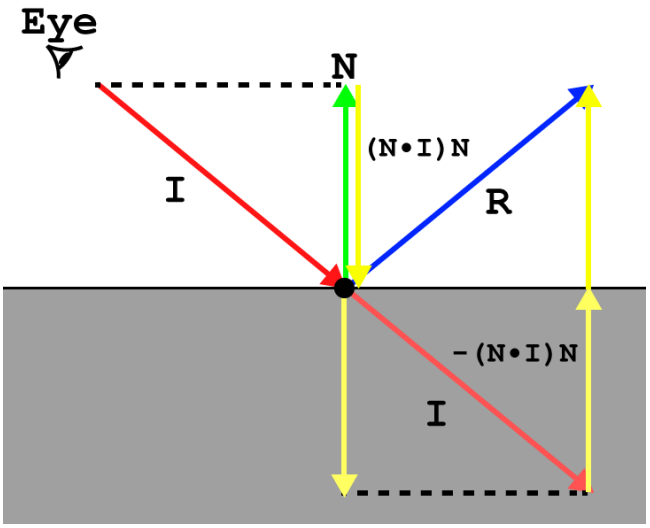
Aucun.

5.4 Réflexion

Fonctionnalité de base, le calcul du rayon réfléchi est relativement simple :

```
Ray getReflectedRay ( Ray const& ray, HitRecord const& record ) {
  double d = Vector3d::Dot ( record.normal, -ray.direction() );
  return Ray ( record.position + 0.01 * record.normal,
      ( 2 * record.normal * d ) + ray.direction() );
}
```

Le schéma de la *fig. 5.4* illustre bien ce calcul.



5.4.1 Exemple

La *fig. 5.2* présente un exemple de reflexion sur une sphère.

5.4.2 Amélioration & bug connu

Aucun.

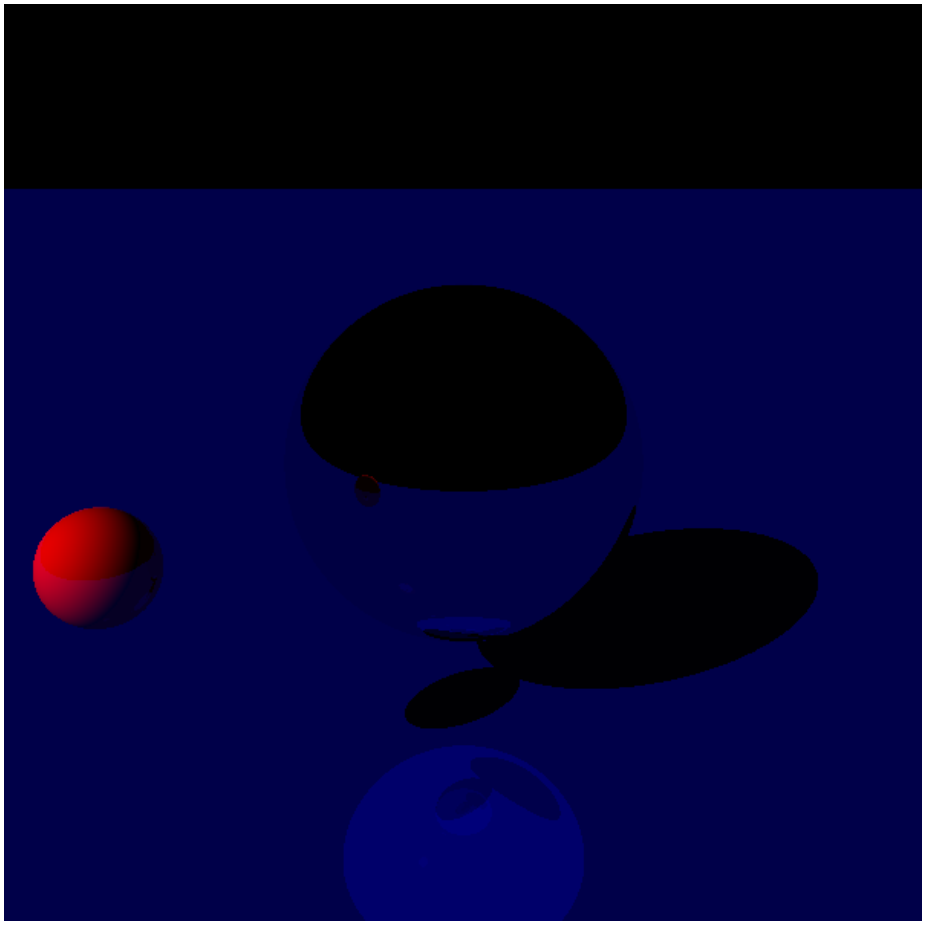


FIGURE 5.2 – Un exemple du rendu de la réflexion de l'environnement sur une sphere

5.5 Réfraction

Le calcul du rayon réfracté est un peu plus compliqué :

```
Ray getRefractedRay (
    Ray const& ray,
    double IOR,
    HitRecord const& record )
{
  double n1 = IOR;
  double n2 = record.hitGeometry->material()->IOR;
  double n = n1 / n2;

  double d = Vector3d::Dot ( record.normal, -ray.direction() );
  double c = sqrt( 1 - n * n * (1 - d * d) );

  double thetaMax = asin ( n2 / n1 );
  if ( acos(d) > thetaMax )
    return getReflectedRay ( ray, record );

  bool input = ( d < M_PI / 2.0 );

  Vector3d bias = ( input )
    ?  -0.01 * record.normal
    : 0.01 * record.normal;

  Ray newRay ( record.position + bias
      , ( n * ray.direction() ) + ( n * d - c ) * record.normal );

  return newRay;
}
```

D'un part, il faut déterminer si le rayon est entrant ou sortant. Pour cela, je compare la direction de la normale avec le rayon incident. Si les deux rayons sont de même sens alors le rayon est entrant. [2] Sinon il est sortant.

Avant de calculer le rayon réfracté, nous devons vérifier que l'angle entre la normal et le rayon incident ne dépasse pas l'angle maximale de réfraction. Autrement dit, si le rayon est rasant, il y a réflexion totale.

Enfin, le calcul du rayon absorbé est fait selon la loi de Snell-Descartes :

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

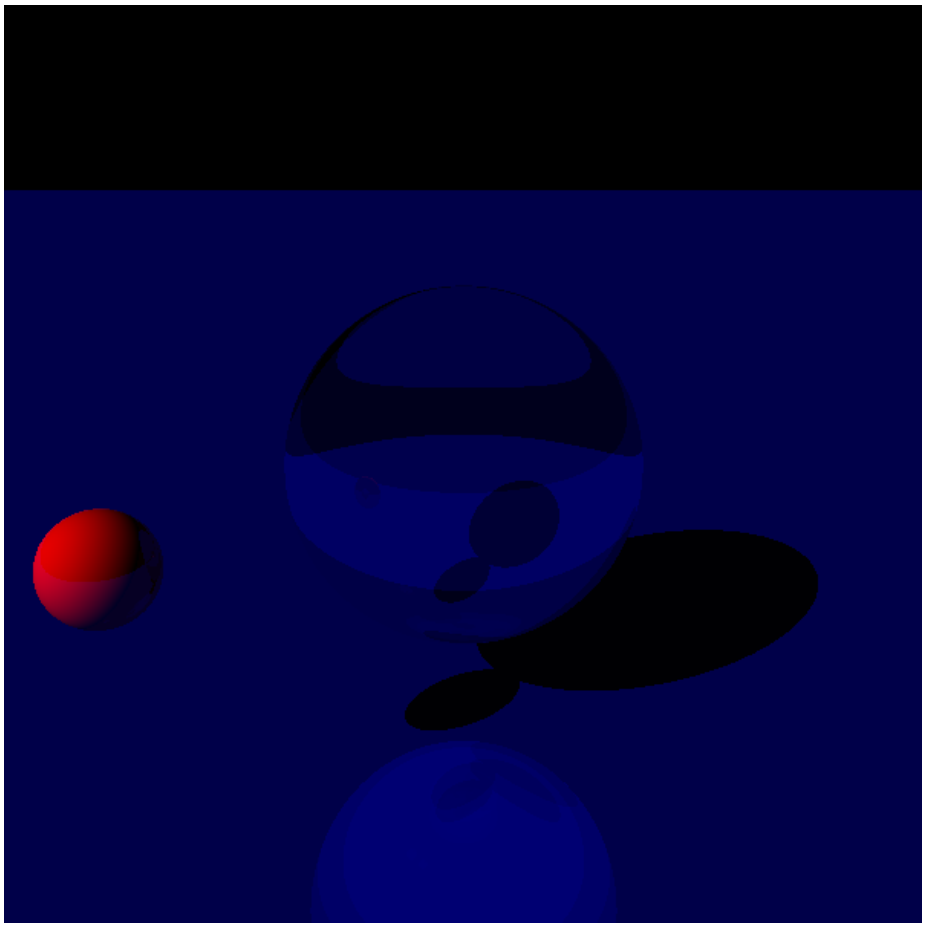


FIGURE 5.3 – Un exemple du rendu de la réfraction de l'environnement sur une sphere

5.5.1 Exemple

5.5.2 Amélioration & bug connu

Aucun.

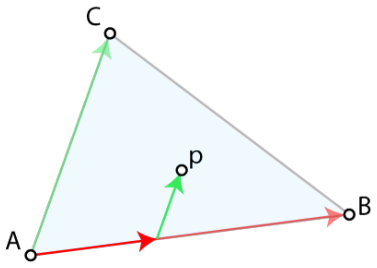
2. Pour se convaincre du calcul de `input`, un simple dessin suffit.

5.6 Interpolation des normales

Comme nous pouvons le voir sur la *fig.5.6*, l'utilisation brute des normales fournit par le modeleur ne nous permet pas d'obtenir un résultat très esthétique. En effet, chaque face possède une normale et la transition entre deux triangles est très marquée.

La solution consiste à assigner non pas une normale par face, mais 3 : une pour chaque sommet. Ainsi, en moyennant la normale de chaque sommet en fonction des faces auxquelles elle appartient, nous pourrons interpoler la normale au point d'intersection en prenant maintenant en considération les faces alentours. La *fig. 5.4* montre le résultat obtenu.

Les coordonnées barycentriques Toute la difficulté de l'interpolation des normales repose donc sur la pondération des différentes normales de la face. Pour cela, il est nécessaire de passer en coordonnées barycentriques. Dans ce système, les coordonnées du point d'intersection sont exprimées en fonction de deux côtés du triangle comme le montre la *fig. 5.6*.



Il est alors évident de calculer le poids de chaque normal à la normal au point d'intersection :

$$\vec{N} = N_a + a * (N_b - N_a) + b * (N_c - N_a)$$

où $a = |AB|$, $b = |AC|$ et N_a (*resp.* N_b et N_c) est la normale en A (*resp.* B et C).

5.6.1 Exemple

5.6.2 Amélioration

Génération des normales Afin de réduire la complexité de l'importeur, j'ai laissé le calcul des normals par sommet à un programme externe. Il serait judicieux (et pas vraiment compliqué) d'intégrer le lissage au processus de chargement normal.

5.6.3 Bug connu

Aucun.

FIGURE 5.4 – Un exemple de rendu avec interpolation des normales

5.7 Mesh

Un *mesh* n'est en réalité qu'un ensemble de triangle, de texture et de coordonnées de textures. Ce n'est donc pas dans l'agrégation que se situe la difficulté.

Le vrai problème se trouve dans le nombre de triangles qu'un mesh peut contenir. Sans structure accélératrice, le calcul de chaque pixel devrait tester l'intersection avec tout les triangles. C'est inimaginable et le temps de la calcul serait multiplié par le nombre de triangles de la scène (certains de mes modèles possèdent plus de 80 000 faces).

5.7.1 L'*octree*

Pour optimiser les calculs d'intersection, j'ai décidé d'utiliser une structure appelée *Octree*. Cette structure permet de diviser l'espace et de le représenter par un arbre. Son fonctionnement est relativement simple :

Initialisation Lors de la création, trouver le cube englobant tous les triangles du *mesh*. Puis, récursivement, diviser ce cube en 8 cubes (au maximum), chacun moitié du cube existant. Continuer tant que le contenu du cube courant est supérieur au contenu maximal. Après cette étape, nous nous retrouvons avec une subdivision du type de celle représenté à la *fig.* **??**.

Calcul d'intersection D'abord, vérifier que le rayon intersecte le cube englobant. Puis, pour chacun de ses cubes enfant, établir la collision. [3] Prendre le plus prêt et descendre comme ceci jusqu'à un cube feuille. Enfin tester l'intersection avec les triangle de la feuille.

Il devient alors évident que le nombre de tests d'intersection est considérablement réduit.

En effet, quelques calculs d'intersections très simples (avec des boites alignées aux axes — AABB [4]) suffisent à atteindre une feuille et par conséquent à tester aux maximum une centaine d'intersections avec des triangles.

5.7.2 Exemple

5.7.3 Amélioration

Aucune.

5.7.4 Bug connu

Problème d'appartenance des triangles : Après un débugage pourtant poussé, je n'ai pas réussi à comprendre pourquoi certains triangles n'était jamais

3. Il est possible d'accélérer le résultat en tenant compte de la direction du rayon sortant lors de l'intersection avec un cube.

4. http://www.cgal.org/Manual/latest/doc_html/cgal_manual/AABB_tree/Chapter_main.html

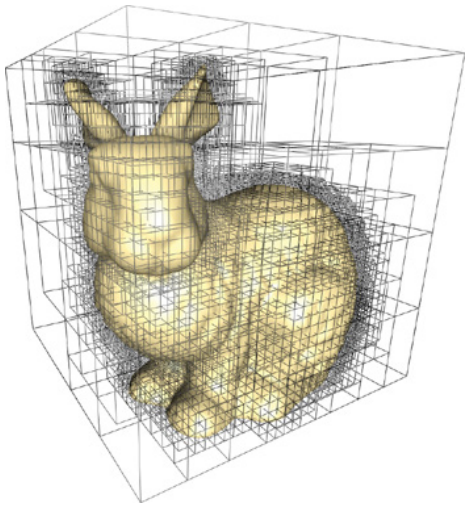


FIGURE 5.5 – Un *octree* construit sur le lapin de Standford. On observe que les zones denses en triangles sont plus subdivisées que les autres zones contenant de moins de polygones.

ajouté à la structure. Ce n'est pourtant pas des triangles proches de la *bounding box*. Ce problème est observable sur la *fig.* 5.6.

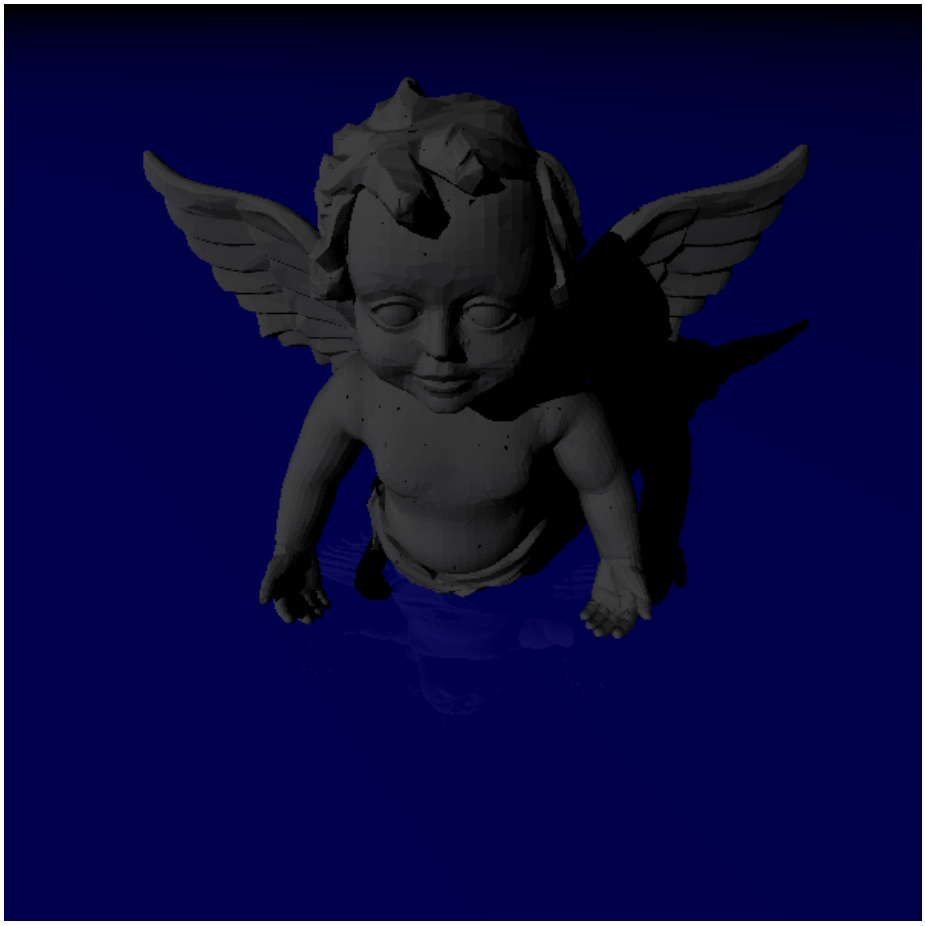


FIGURE 5.6 – Un exemple de rendu de mesh (+80 000 faces, < 1 minutes)

5.8 Ombres douces

Le rendu des ombres douces (*i.e.* de la transition entre ombre et pénombre) n'est pas immédiate avec le *ray tracing*.

En effet, l'approche trivial consiste à considérer les états d'éclairement suivant :

- **Dans l'ombre**, *i.e.* qu'il existe au moins un objet entre la source de lumière et le point considéré.
- **Pas dans l'ombre**, *i.e.* il n'existe aucun objet entre la source et le point.

Cette approche conduit à des résultats peu convaincants comme le montre la *fig.5.7*.

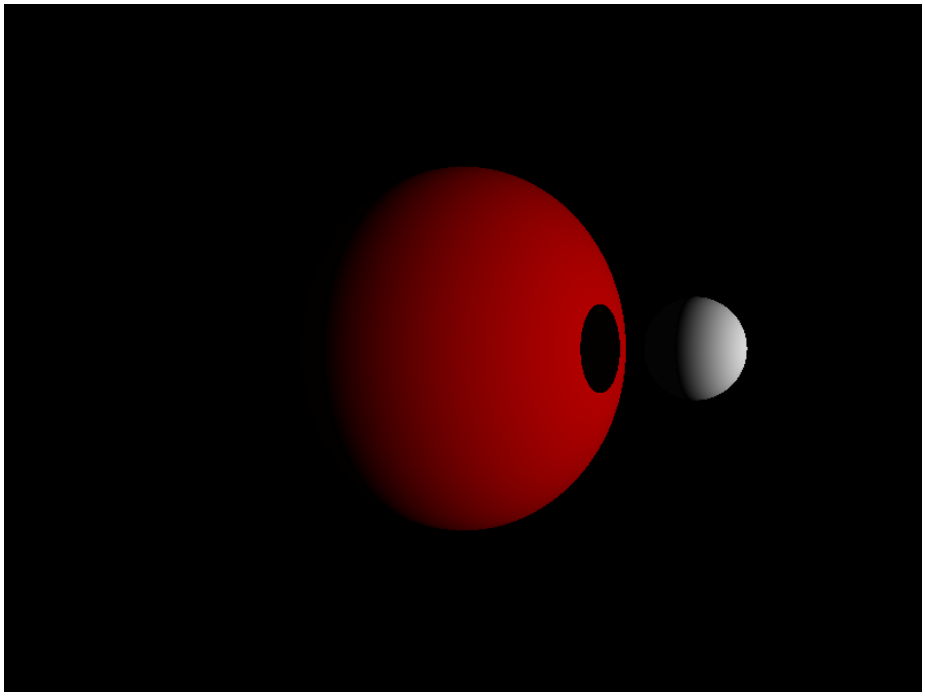


FIGURE 5.7 – Projection de l'ombre *dure* de la sphère grise sur la sphère rouge.

Pour éviter cet effet, il s'agit d'utiliser des lumières non plus ponctuelles et discrètes, mais surfacique et continue.

Ainsi, plutôt que d'établir l'appartenance d'un point à une zone d'ombre en lançant un seul rayon vers la source de lumière nous répétons cette opération sur toute la surface de la lumière (en utilisant une distribution statistique de notre choix) et en faisant la moyenne.

5.8.1 Exemple

Une telle technique, permet, au prix d'une perte de performance, d'obtenir des résultats du type de la *fig.* 5.8.

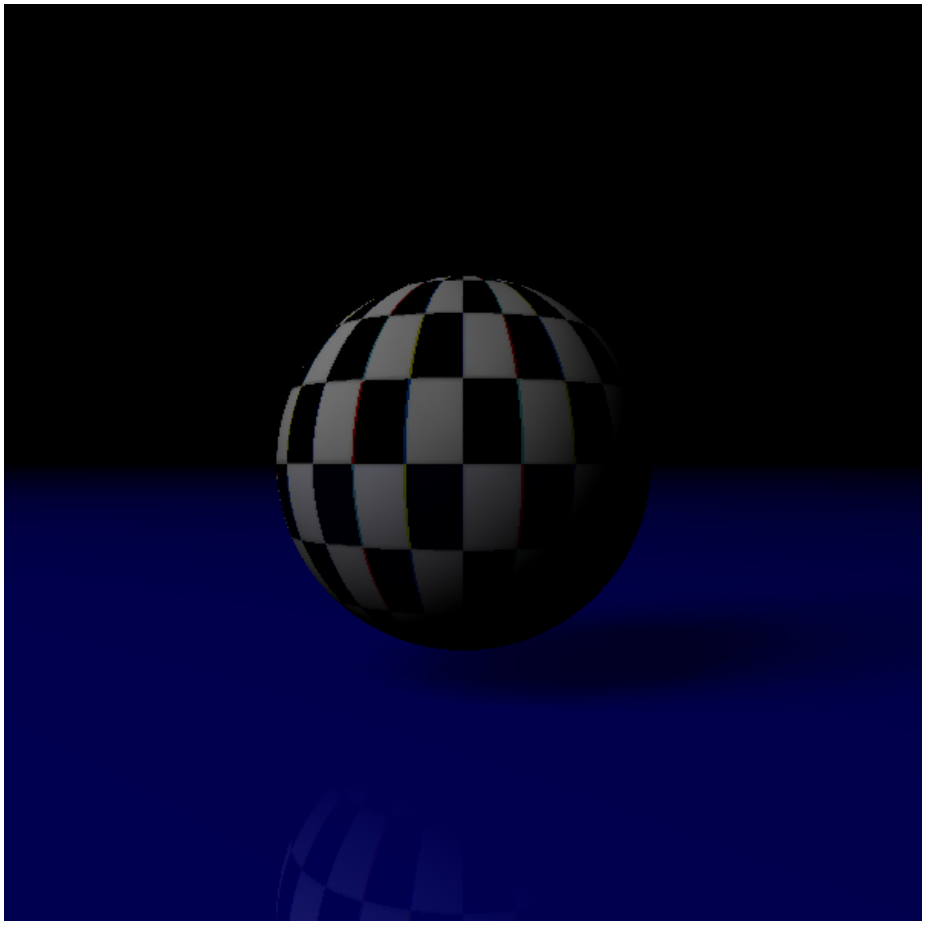


FIGURE 5.8 – Un exemple de rendu des ombres douces

5.8.2 Amélioration

Ajout de lumières surfaciques Pour le moment, la seule lumière surfacique qui est implémentée est la lumière plane. Il pourrait être pratique de disposer de lumière sphérique par exemple.

5.8.3 Bug connu

Aucun.

5.9 *Depth of Field*

Cet effet correspond à la déviation que subissent les rayons lumineux lorsqu'il traversent l'obturateur d'un appareil photo.

Le résultat est que seul le plan de focus est totalement net alors que les objets le précédent ou le suivant deviennent de plus en plus flou proportionnellement à la distance qui les séparent du plan net.

Deux implémentations de cette technique sont possibles :

1. *L'implémentation physique*, prenant en compte la forme de l'obturateur pour calculer la déviation des rayons. Elle est complexe et mal documentée.
2. *L'implémentation symptomatique*, simulant simplement le phénomène physique par des déviations aléatoires et proportionnelles à l'ouverture de l'objectif. Elle est très simple à implémenter.

Inutile de vous donner la méthode que j'ai choisi d'implémenter !

Le gros défaut de cette technique (quelque soit la méthode utilisée) est qu'elle nécessite de lancer de nombreux rayons en plus (de l'ordre de 124 fois plus).

5.9.1 Exemple

Voici deux exemples avec deux points de focus différents :

5.9.2 Amélioration

Meilleur cohérence de l'ouverture Par analogie avec la photographie, le facteur de déviation des rayons lumineux par rapport au centre optique est réglé grâce au paramètre `aperture`. Hélas, ce paramètre devra changer en fonction de la taille de la scène car une déviation d'une unité dans un monde de 100 unité et un monde de 0.1 unité ne provoque pas du tout les même résultat. Il faudrait donc trouver un meilleur paramètre pour ces quantité et le définir de manière absolue.

5.9.3 Bug connu

La gestion du focus sur un point situé dans les Z négatifs ne fonctionne pas.

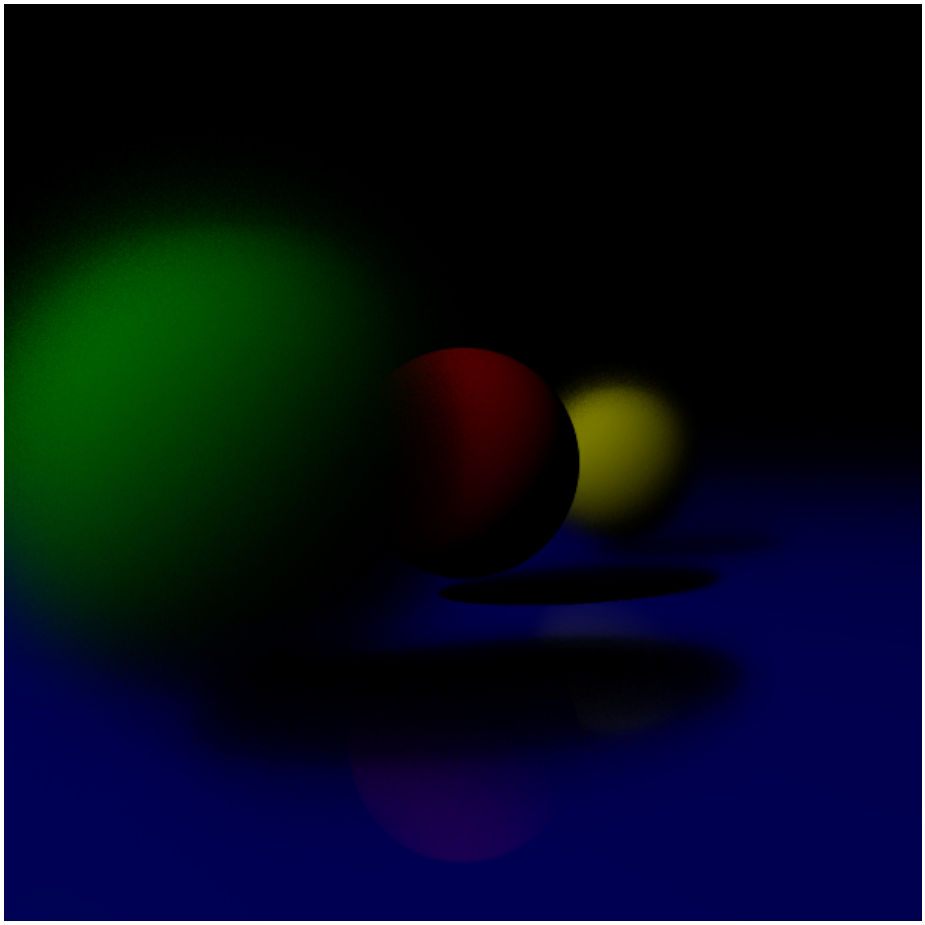


FIGURE 5.9 – Un exemple de rendu de la profondeur de champ avec point de focus sur la sphère rouge.

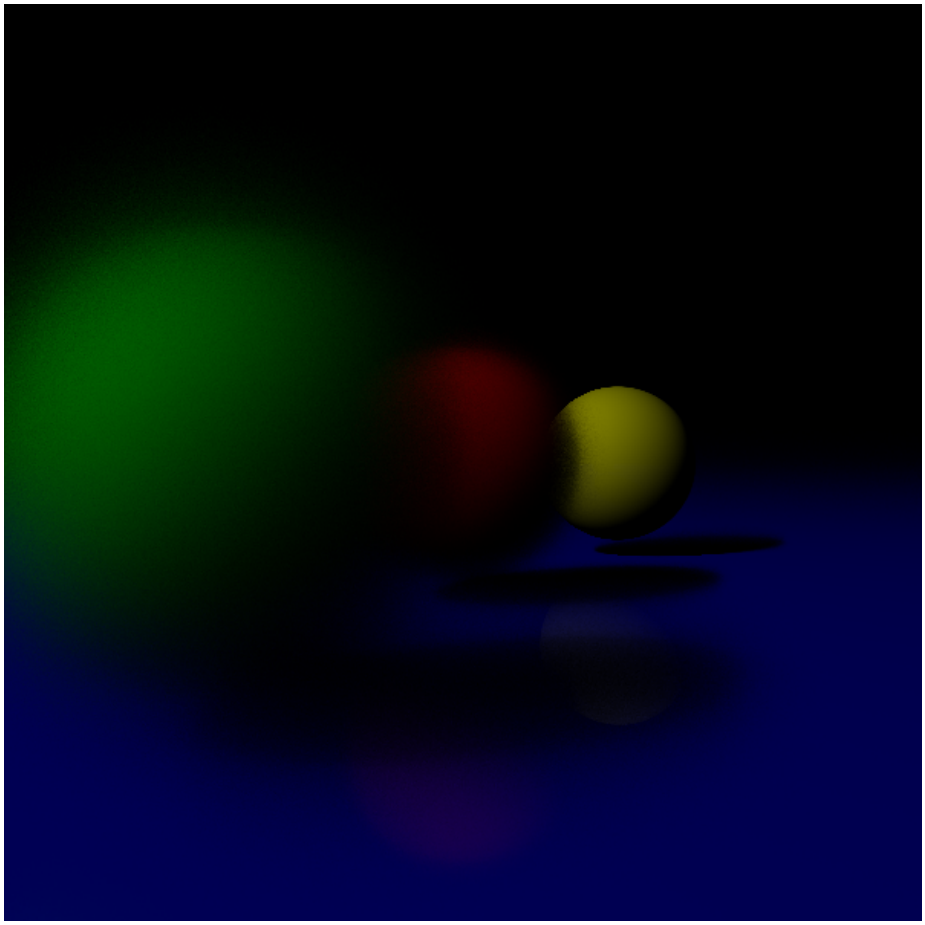


FIGURE 5.10 – Un exemple de rendu de la profondeur de champ avec point de focus sur la sphère jaune.

Chapitre 6

Manuel utilisateur

6.1 Compilation

Le *pipeline* de compilation est tout ce qu'il y a de plus classique :

```
./configure
make
```

6.1.1 Dépendances

Afin de pouvoir compiler ce programme, le script `configure` va s'assurer que vous posséder :

- boost @1.47.0, Revision 2 (>=)
- lib3ds @20080909 (>=)
- libpng @1.4.8 (==)

6.2 Licence

C'est la première fois que j'ai à faire au problème des license et avec seulement 3 bibliothèque, cela semble déjà être un casse tête. Je crois cependant que la license la plus restrictive est la *Boost licence version 1.0* et que par conséquent, ce programme doit être distribué sous cette license.

6.3 Utilisation

L'utilisation du programme est très simple :

```
./lrt SCENE.lrt [fichierDeSortie.png]
```

Remarque : *Par défaut, le fichier de sortie est* `result.png`.

Chapitre 7

Conclusion

En tant qu'élève de dernière année, j'ai souhaité que ce projet mette en œuvre l'ensemble des compétences que j'ai pu acquérir au long de mes études d'ingénieur. C'est pour cette raison que j'ai choisi un projet plutôt conséquent. J'ai pu y développer :

- Mes compétences générales en compilation, rédaction, *versionning* de code source, des dépendances, des licences...
- Ma capacité à travailler seul sur un projet complexe.
- Mes compétences en programmation C++.
- Mes compétences à traquer et résoudre les problèmes.
- Mes compétence de gestion de projet.
- Ma motivation à continuer lorsque rien ne marche comme il le faudrait (car c'est arrivé).
- Encore bien d'autres choses qui ne me viennent pas à l'esprit.

Enfin, si mon programme n'est qu'un *proof of concept*, j'espère avoir rempli les objectifs du projet personnel.

Merci de votre temps, Maxime Gaudin.

Glossaire

adaptative sampling C'est une forme de supersampling qui n'est réalisée que si un certain seuil d'aliasing est détecté.. 9, 52

aliasing Le crénelage sur une image.. 52

backface culling (Abattage ?) Appelé aussi simplement *culling*, c'est l'opération consistant à éliminer toutes les faces faisant dos à la caméra et ne pouvant par conséquent pas être vues.. 52

clipping C'est l'élimination des face hors du champ de la caméra.. 52

jittering C'est la perturbation que les rayons lumineux subissent lorsqu'une surface mal polie les réfléchi (par exemple du verre sablé).. 9, 52

motion blur (Flou de mouvement) C'est le flou généré par une vitesse d'obturation de la caméra trop grande.. 9, 52

shader Un shader est un programme appliqué à l'ensemble des pixel ou des sommets des polygones de la scène. Ils sont aujourd'hui incorporés dans le pipeline de rendu des cartes graphiques.. 1, 52

supersampling Littéralement, sur-échantillonnage, cette technique permet d'éviter des effets d'aliasing.. 6, 52

Acronyms

GPU Graphics Processing Unit. 1, 12, 52

XP Extrem Programming. 5, 52

Bibliographie

[Car03] Wayne E. Carlson. Cgi historical timeline. https://design.osu.edu/carlson/history/timeline.html, 2003.

[Cha06] Oscar Xavier Chavarro. Cgi historical timeline - reloaded. http://sophia.javeriana.edu.co/~ochavarr/computer_graphics_history/historia/, 2006.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[LW93] Eric P. Lafortune and Yves D. Willems. Bi-directional path tracing. In *Proceedings of third international conference on computational graphics and visualization techniques (COMPUGRAPHICS)*, pages 145–153, 1993.

[Mey94] Scott Meyers. *Effective C++*. Addison-Wesley, Reading, Massachusetts, USA, 1994.

[Sut05] Herb Sutter. *Exceptional C++ Style : 40 New Engineering Puzzles, Programming Problems, and Solutions. Exceptional C++ Style. Forty New Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley, Boston, MA, 2005.

[Whi80] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23 :343–349, June 1980.

[Wik11] Wikipedia. Radiosity. http://en.wikipedia.org/wiki/Radiosity_(3D_computer_graphics)#Mathematical_formulation, 2011.

[WS01] Ingo Wald and Philipp Slusallek. State of the art in interactive ray tracing, 2001.

Table des matières

Table des figures

”

S'il est plus près du programme de Whitted *que de Renderman*[a]*, LRT reste très prometteur !*

Donald Knuth ☺

a. Pixar