

# Projet Raytracer

César Charbey

14 décembre 2025

## Table des matières

<b>1 Les Fondations du Lancer de Rayons</b>	<b>2</b>
1.1 Principe de Base . . . . .	2
1.2 Intersection Rayon-Sphère . . . . .	2
1.3 Intersection Rayon-Carré . . . . .	2
<b>2 Éclairage et Ombres</b>	<b>3</b>
2.1 Modèle d'Illumination de Phong . . . . .	3
2.2 Ombres Portées . . . . .	3
2.3 Ombres Douces (Soft Shadows) et Area Lights . . . . .	4
<b>3 Géométrie Complexé et Réflexion</b>	<b>5</b>
3.1 Gestion des Maillages . . . . .	5
3.1.1 Algorithme d'Intersection de Möller–Trumbore . . . . .	5
3.1.2 Justification : Möller–Trumbore vs Méthode Géométrique . . . . .	6
3.2 Simulation Avancée : Réflexion et Réfraction . . . . .	7
3.2.1 Le Principe du Miroir Parfait (Réflexion) . . . . .	7
3.2.2 La Transparence et la Réfraction (Loi de Snell-Descartes) . . . . .	7
3.2.3 Le Réalisme du Verre : L'Approximation de Fresnel . . . . .	8
<b>4 Structure d'Accélération</b>	<b>9</b>
4.1 Architecture d'Accélération : Le KdTree à Deux Niveaux . . . . .	9
4.1.1 Niveau 1 : Le KdTree Global (Top-Level Acceleration) . . . . .	9
4.1.2 Niveau 2 : Le KdTree Local (Bottom-Level Acceleration) . . . . .	9
4.1.3 Justification de l'Architecture à Deux Niveaux . . . . .	10
4.1.4 Parallèle avec le Raytracing Temps Réel (GPU) . . . . .	11
<b>5 Améliorations Visuelles</b>	<b>12</b>
5.1 Modèle d'Illumination Blinn-Phong . . . . .	12
5.2 Photon Mapping et Caustiques . . . . .	12
5.3 Scène Sous-Marine et Brouillard . . . . .	13
5.4 Tone Mapping ACES . . . . .	14
5.5 Interface Utilisateur (ImGui) . . . . .	15
<b>6 Rendu "Artistique"</b>	<b>16</b>
6.1 Fails . . . . .	16
6.2 Rendus Finaux . . . . .	17

# 1 Les Fondations du Lancer de Rayons

## 1.1 Principe de Base

**Github** : [Lien Github](#)

Ce projet a pour but de développer un moteur de lancer de rayons (*Raytracer*) complet en C++ / OpenGL. Partant d'une structure minimale, j'ai implémenté progressivement les concepts fondamentaux pour aboutir à un moteur capable de simuler des phénomènes physiques complexes comme la réfraction, les caustiques et l'illumination globale approximée. Ce rapport détaille chronologiquement les 5 grandes parties de développement.

La première étape consistait à mettre en place le mécanisme fondamental du *Raytracing*. Contrairement à la lumière réelle, nous lançons les rayons depuis l'oeil (la caméra) à travers chaque pixel de l'écran virtuel pour déterminer ce qu'ils touchent.

## 1.2 Intersection Rayon-Sphère

La première primitive implémentée fut la sphère. L'intersection se résout analytiquement. En considérant une sphère de centre  $C$  et de rayon  $R$ , et un rayon  $P(t) = O + tD$ , l'équation  $\|P(t) - C\|^2 = R^2$  mène à une équation du second degré  $at^2 + bt + c = 0$ .

- Si le discriminant  $\Delta < 0$ , le rayon manque la sphère.
- Si  $\Delta > 0$ , il y a deux solutions  $t_1$  et  $t_2$ . Je retiens la plus petite valeur positive (le point d'entrée le plus proche).

## 1.3 Intersection Rayon-Carré

Pour représenter les murs de la scène (Cornell Box), j'ai implémenté l'intersection avec un carré (plan fini). L'algorithme se déroule en deux temps : 1. Calculer l'intersection avec le plan infini supportant le carré. 2. Vérifier si le point d'impact se trouve à l'intérieur des limites du carré en projetant le point sur les axes locaux du carré.

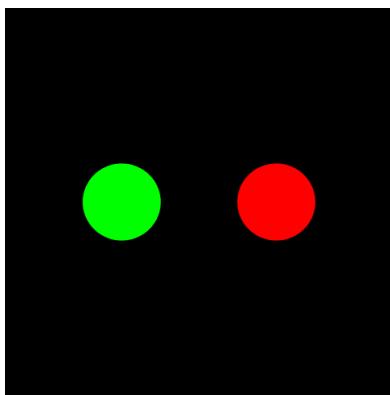


FIGURE 1 – Sphère.

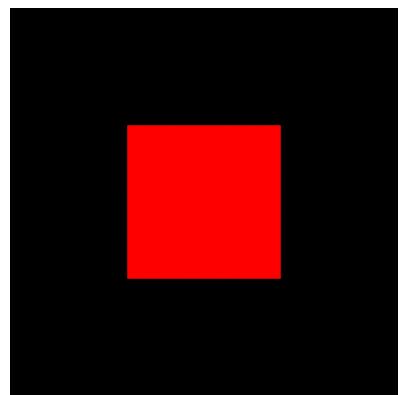


FIGURE 2 – Carré.

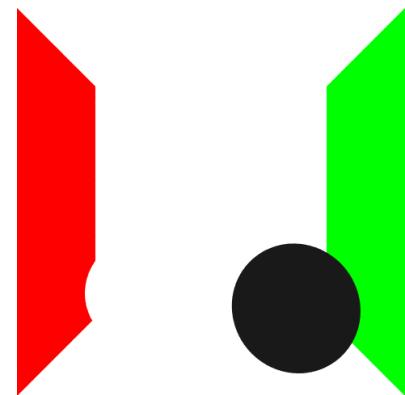


FIGURE 3 – Assemblage.

FIGURE 4 – Primitives de bases implémentées et fonctionnelle

## 2 Éclairage et Ombres

### 2.1 Modèle d'Illumination de Phong

Une fois l'intersection trouvée, il faut déterminer la couleur du pixel. J'ai implémenté le modèle local de Phong qui décompose la lumière en trois composantes :

- **Ambiante** ( $I_a$ ) : Une lumière constante qui simule l'éclairage indirect minimal.
- **Diffuse** ( $I_d$ ) : La lumière réfléchie dans toutes les directions (loi de Lambert), proportionnelle à  $\cos(\theta) = N \cdot L$ .
- **Spéculaire** ( $I_s$ ) : La tache brillante (reflet de la source) visible sur les objets brillants, proportionnelle à  $(R \cdot V)^\alpha$ .

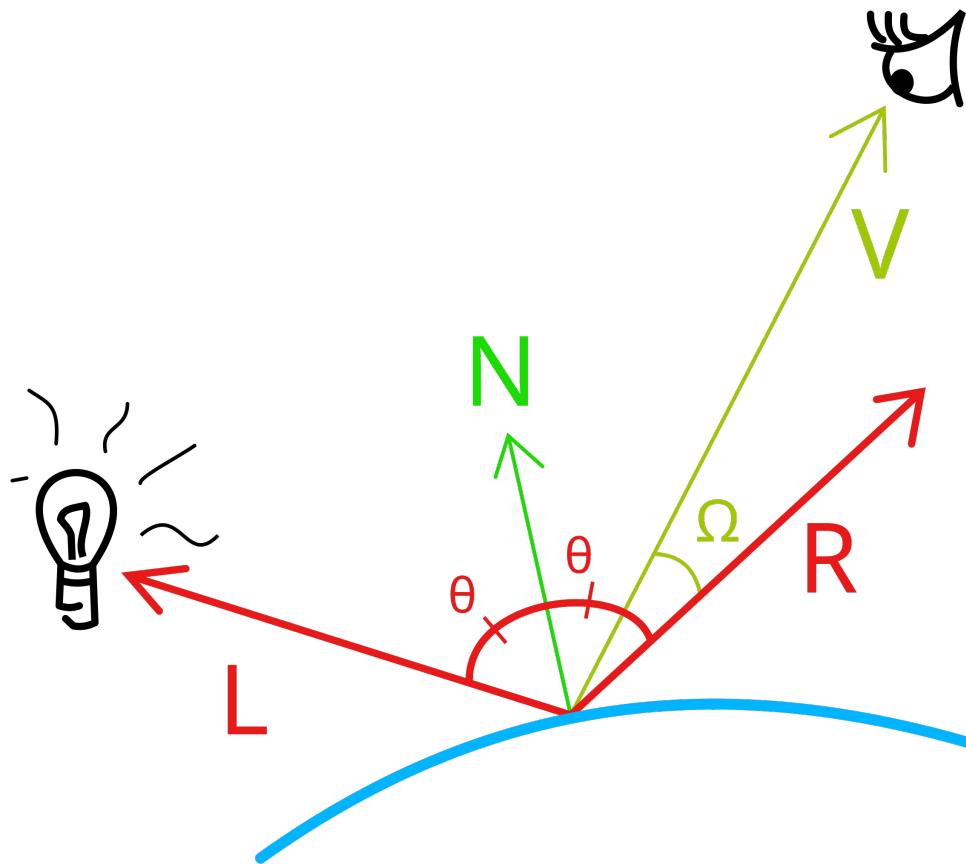


FIGURE 5 – Décomposition visuelle du modèle de Phong.

### 2.2 Ombres Portées

Pour ajouter du réalisme, j'ai implémenté les ombres. Pour chaque point d'intersection, je lance un "rayon d'ombre" vers la source lumineuse. Si ce rayon rencontre un objet opaque avant d'atteindre la lumière (distance  $t < d_{lumière}$ ), le point est à l'ombre. Dans ce cas, seules les composantes diffuse et spéculaire sont annulées (l'ambiante reste).

### 2.3 Ombres Douces (Soft Shadows) et Area Lights

Pour pallier le manque de réalisme des ombres dures (créées par des sources ponctuelles), j'ai implémenté des sources de lumière surfaciques (*Area Lights*).

J'ai enrichi la structure `Light` avec deux vecteurs  $\vec{u}$  et  $\vec{v}$  définissant un plan rectangulaire autour de la position de la lumière. L'algorithme repose sur une méthode de Monte-Carlo : au lieu de lancer un unique rayon d'ombre vers le centre, je lance  $N$  rayons (par exemple 32) vers des points aléatoires répartis sur la surface de la lumière.

Pour chaque point d'intersection dans la scène, je calcule la moyenne de la visibilité vers ces  $N$  points. Cela crée naturellement une zone de pénombre : plus l'objet occlusif est proche de la surface réceptrice, plus l'ombre est nette ; plus il s'éloigne, plus l'angle solide de la source varie, créant une ombre floue réaliste.

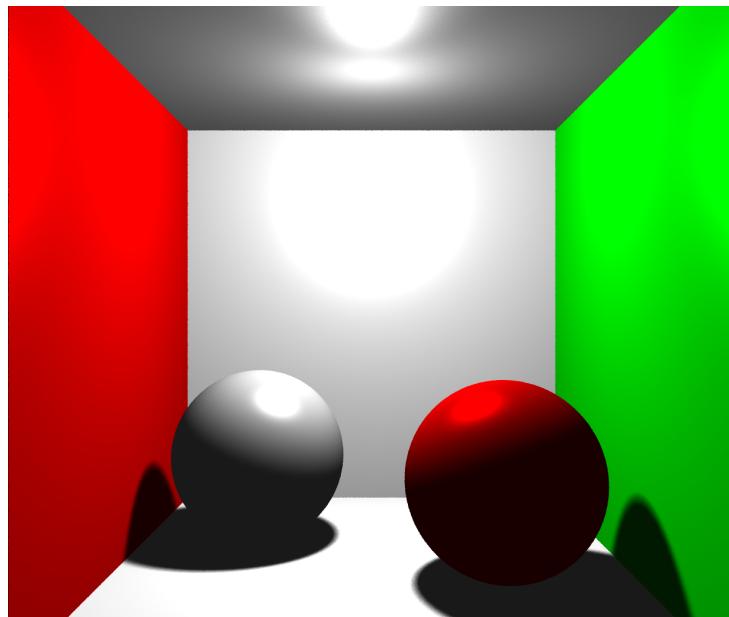


FIGURE 6 – Résultat de la Phase 2 : Sphères avec éclairage de Phong et ombres douces.

### 3 Géométrie Complex et Réflexion

#### 3.1 Gestion des Maillages

Pour dépasser les primitives simples, j'ai implémenté le chargement de fichiers .OFF. Ma classe `Mesh` stocke les sommets et les triangles. J'ai accordé une importance particulière au **lissage des surfaces** (*Smooth Shading*). Lors du chargement, je pré-calcule la normale moyenne de chaque sommet.

Lors du rendu, j'utilise les coordonnées barycentriques  $(u, v, w)$  fournies par l'algorithme d'intersection de **Möller–Trumbore** pour interpoler la normale au point d'impact :

$$N_{\text{interpolée}} = (1 - u - v) \cdot N_{v0} + u \cdot N_{v1} + v \cdot N_{v2}$$

Cela permet à des objets comme le dragon d'apparaître parfaitement courbes malgré leur nature polygonale.

##### 3.1.1 Algorithme d'Intersection de Möller–Trumbore

Pour calculer l'intersection entre un rayon et un triangle, j'ai choisi d'implémenter l'algorithme de **Möller–Trumbore**. C'est une méthode standard en synthèse d'image car elle est rapide et efficace en mémoire : elle ne nécessite pas de pré-calculer ni de stocker l'équation du plan du triangle.

Le principe repose sur un changement de base. Au lieu de chercher le point d'intersection en coordonnées cartésiennes  $(x, y, z)$ , nous cherchons à exprimer le rayon directement dans le système de coordonnées barycentriques du triangle.

Soit un rayon défini par son origine  $O$  et sa direction  $D$ , et un triangle défini par ses sommets  $V_0, V_1, V_2$ . Tout point  $T(u, v)$  sur le triangle peut être écrit comme :

$$T(u, v) = (1 - u - v)V_0 + uV_1 + vV_2$$

où  $(u, v)$  sont les coordonnées barycentriques.

Pour trouver l'intersection, nous égalons l'équation du rayon  $R(t) = O + tD$  avec celle du triangle :

$$O + tD = (1 - u - v)V_0 + uV_1 + vV_2$$

En réarrangeant les termes et en posant les vecteurs des arêtes  $E_1 = V_1 - V_0$ ,  $E_2 = V_2 - V_0$  et le vecteur distance  $T = O - V_0$ , nous obtenons un système linéaire à trois inconnues  $(t, u, v)$  :

$$\begin{bmatrix} -D & E_1 & E_2 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = T$$

Mon implémentation résout ce système en utilisant la **méthode de Cramer**. Cela permet de déterminer si le rayon coupe le triangle en effectuant plusieurs tests de rejet rapides :

1. Si le déterminant du système est proche de zéro, le rayon est parallèle au triangle (pas d'intersection).
2. Si la coordonnée calculée  $u$  est en dehors de l'intervalle  $[0, 1]$ , le point est hors du triangle.
3. Si la coordonnée  $v$  est négative ou si  $u + v > 1$ , le point est également hors du triangle.

Si tous ces tests passent, nous obtenons la distance  $t$  exacte et les coordonnées  $(u, v)$  qui seront cruciales pour l'étape suivante : l'interpolation des normales.

### 3.1.2 Justification : Möller–Trumbore vs Méthode Géométrique

J'aurais pu opter pour l'approche plus "intuitive", souvent appelée **Méthode Géométrique**. Celle-ci se déroule classiquement en deux étapes distinctes :

1. Calculer l'intersection du rayon avec le **plan infini** supportant le triangle (ce qui nécessite de calculer et stocker la normale de la face).
2. Vérifier si le point d'impact se situe bien à l'intérieur du triangle en réalisant des produits vectoriels (ou déterminants) pour tester la position du point par rapport aux trois arêtes.

Cependant, j'ai privilégié l'algorithme de Möller–Trumbore pour deux raisons techniques décisives :

- **Coût Mémoire et Pré-calcul** : Möller–Trumbore ne nécessite pas de stocker l'équation du plan ( $Ax + By + Cz + D = 0$ ). Il travaille uniquement avec les sommets bruts, ce qui réduit le poids mémoire, un atout majeur lorsque la scène contient beaucoup de triangles.
- **Coordonnées Barycentriques "Gratuites"** : Pour réaliser le lissage des normales, j'ai impérativement besoin des coordonnées barycentriques  $(u, v)$ . La méthode géométrique me donnerait la position  $P$  dans l'espace monde, m'obligeant à effectuer un calcul supplémentaire coûteux (projection d'aires) pour retrouver  $(u, v)$ . Avec Möller–Trumbore, ces coordonnées sont le résultat direct de la résolution du système, obtenues sans surcoût.

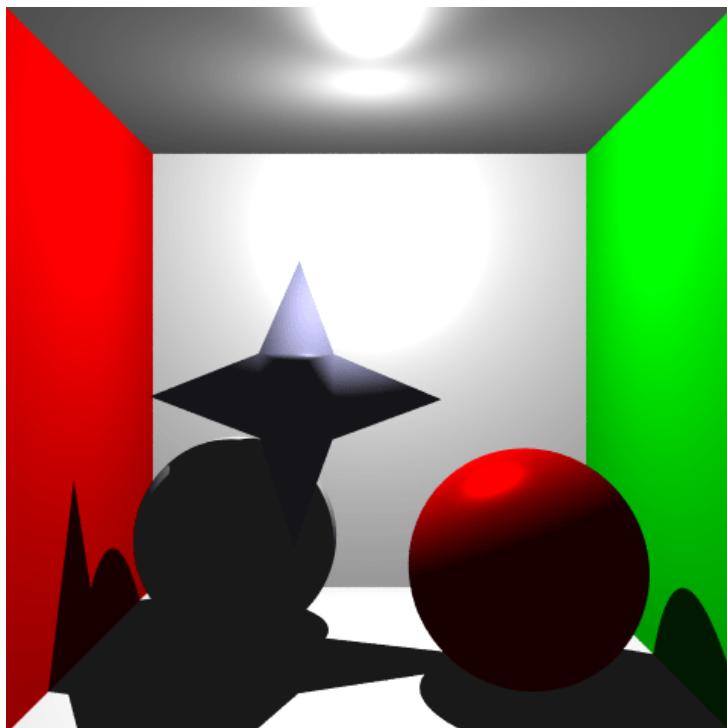


FIGURE 7 – Rendu Boîte de Cornell avec Maillage (étoile).

## 3.2 Simulation Avancée : Réflexion et Réfraction

Pour obtenir un rendu photoréaliste, il est indispensable de simuler le comportement de la lumière lorsqu'elle interagit avec des surfaces lisses (miroirs) ou transparentes (verre, eau). J'ai pour cela introduit la récursivité dans ma fonction de lancer de rayon : la couleur d'un pixel dépend désormais de la couleur rapportée par des rayons secondaires.

### 3.2.1 Le Principe du Miroir Parfait (Réflexion)

La simulation d'un miroir repose sur la loi de la réflexion spéculaire. Lorsqu'un rayon incident  $I$  frappe une surface, il rebondit de l'autre côté de la normale  $N$  avec le même angle.

J'ai implémenté le calcul du vecteur réfléchi  $R$  selon la formule vectorielle suivante :

$$R = I - 2(N \cdot I)N$$

L'algorithme procède ainsi :

1. Je calcule  $R$  et je déplace légèrement l'origine du nouveau rayon ( $P + \epsilon R$ ) pour éviter l'auto-intersection (problème d'acné).
2. Je lance un appel récursif à `rayTraceRecursive` avec ce nouveau rayon.
3. Je décrémente la profondeur de récursion. Si elle atteint 0, je retourne du noir pour arrêter le calcul et éviter une boucle infinie (ex : miroirs face à face).
4. La couleur finale est une combinaison de la couleur intrinsèque de l'objet et de la couleur retournée par le rayon réfléchi.

### 3.2.2 La Transparence et la Réfraction (Loi de Snell-Descartes)

La gestion des objets transparents est plus complexe car la lumière ne se contente pas de traverser l'objet : elle est déviée. C'est le phénomène de réfraction, régi par la loi de Snell-Descartes :

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

où  $n_1$  et  $n_2$  sont les indices de réfraction des milieux (ex : 1.0 pour l'air, 1.5 pour le verre).

Pour implémenter cela, j'ai dû résoudre deux problèmes majeurs :

**1. Gestion des Interfaces (Entrée / Sortie) :** Un rayon peut entrer dans une sphère de verre (Air → Verre) ou en sortir (Verre → Air). Le moteur doit savoir dans quel cas il se trouve pour appliquer le bon ratio d'indices  $\eta$ .

- Je calcule le produit scalaire  $c = I \cdot N$ .
- Si  $c < 0$ , le rayon frappe la face extérieure. Je suis dans l'air, j'entre dans le verre. ( $\eta = \frac{n_{air}}{n_{verre}}$ ).
- Si  $c > 0$ , le rayon frappe la face intérieure. Je suis dans le verre, je sors vers l'air. Dans ce cas, **j'inverse la normale** ( $N = -N$ ) et j'inverse le ratio d'indices ( $\eta = \frac{n_{verre}}{n_{air}}$ ).

**2. Réflexion Totale Interne :** Lorsque la lumière tente de passer d'un milieu plus réfringent (verre) à un milieu moins réfringent (air) avec un angle trop rasant, elle ne peut pas sortir. Elle est entièrement piégée à l'intérieur. Mathématiquement, cela se produit lorsque le sinus de l'angle réfracté  $> 1$ . Dans ce cas, je force une réflexion spéculaire classique.

### 3.2.3 Le Réalisme du Verre : L'Approximation de Fresnel

Dans la réalité, le verre n'est jamais 100% transparent. La quantité de lumière réfléchie par rapport à la lumière réfractée dépend de l'angle de vue.

- Si je regarde une vitre de face, elle est très transparente (peu de réflexion).
- Si je la regarde en biais (angle rasant), elle devient presque un miroir parfait.

Pour simuler ce phénomène physique crucial, j'utilise l'approximation de **Fresnel-Schlick**. Cette fonction calcule un coefficient  $R_f$  (entre 0 et 1) basé sur l'angle d'incidence et les indices de réfraction.

La couleur finale du pixel est alors un mélange pondéré (Lerp) :

$$\text{Couleur}_{Finale} = R_f \times \text{Couleur}_{Reflet} + (1 - R_f) \times \text{Couleur}_{Refraction}$$

C'est cette combinaison de réfraction physique, de gestion des sorties de milieu et d'effet Fresnel qui donne à la sphère de verre son aspect "solide" et réaliste, avec des bords plus brillants et réfléchissants.

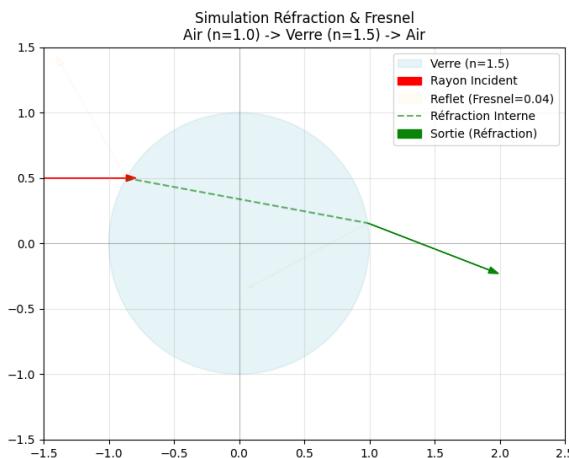


FIGURE 8 – Comportement standard : Le rayon est réfracté (dévié) en entrant et en sortant de l'objet.

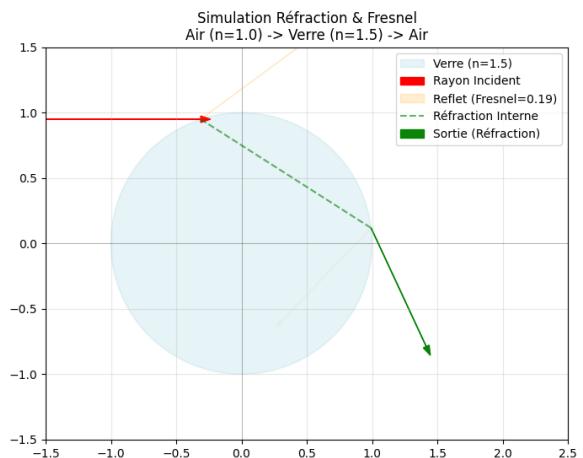


FIGURE 9 – Angle rasant : Le rayon est fortement ou totalement réfléchi + comportement standard.

FIGURE 10 – Illustration des interactions lumineuses (Réfractions & Fresnel).

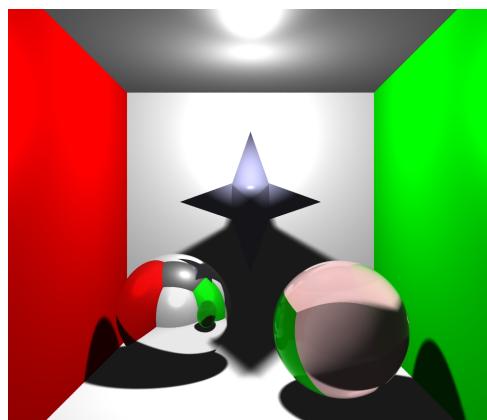


FIGURE 11 – Rendu Boîte de Cornell avec maillage, miroir et verre.

## 4 Structure d'Accélération

### 4.1 Architecture d'Accélération : Le KdTree à Deux Niveaux

L'ajout de maillages complexes rend le rendu naïf impossible. Tester chaque rayon contre chaque triangle donnerait une complexité linéaire  $O(N)$ . Pour une image HD, cela représenterait des milliards de tests inutiles.

Pour résoudre ce problème, j'ai implémenté une structure d'accélération spatiale hiérarchique : un **KdTree**, qui fonctionne ici comme une hiérarchie de volumes englobants (BVH). J'ai choisi une architecture à deux niveaux pour séparer la gestion de la scène de la gestion de la géométrie pure.

#### 4.1.1 Niveau 1 : Le KdTree Global (Top-Level Acceleration)

Ce premier niveau structure la scène dans son ensemble. Il ne "voit" pas les triangles, mais uniquement les **objets** (Sphères, Meshs entiers, Plans).

**Construction de l'arbre :** L'algorithme procède de manière récursive. Pour chaque nœud de l'arbre, je calcule la boîte englobante (AABB - Axis Aligned Bounding Box) qui contient tous les objets du nœud.

1. **Choix de l'axe** : Je détermine l'axe le plus long de la boîte englobante (X, Y ou Z).
2. **Tri Médian** : Je trie les objets selon la position de leur centre de gravité (centroïde) le long de cet axe.
3. **Division** : Je coupe la liste d'objets en deux : ceux à gauche de la médiane vont dans le fils gauche, ceux à droite dans le fils droit.
4. **Arrêt** : La récursion s'arrête lorsque le nombre d'objets dans un nœud est suffisamment petit (feuille).

**Traversée (Intersection)** : Lorsqu'un rayon est lancé, je traverse l'arbre de la racine vers les feuilles. Si le rayon ne touche pas la boîte englobante d'un nœud, je sais instantanément que je peux ignorer tous les objets contenus dans ce nœud (et ses enfants). C'est le principe du **Culling**.

#### 4.1.2 Niveau 2 : Le KdTree Local (Bottom-Level Acceleration)

Le deuxième niveau est interne à la classe **Mesh**. Il gère les milliers de triangles qui composent un objet unique. Sans ce niveau, toucher la boîte englobante d'un maillage nous obligerait à tester tout ses triangles un par un.

**Structure des données** : Pour optimiser l'accès mémoire et le cache processeur, j'ai créé une structure légère **TriAccel** stockée dans un vecteur contigu. Elle pré-calcule certaines données géométriques pour éviter de les refaire à chaque intersection.

**Algorithme de traversée optimisée** : Pour la traversée, j'ai évité la récursivité (qui peut être lente à cause des appels de fonction) au profit d'une **pile itérative (Stack)**.

1. Je teste l'intersection rayon-boîte du nœud courant.

2. Si intersection il y a, je détermine quel enfant est le plus proche de l'origine du rayon ("Near child") et lequel est le plus loin ("Far child").
3. Je visite le "Near child" en priorité et j'empile le "Far child" pour plus tard.
4. Si j'atteins une feuille, je teste l'intersection réelle (cf. Möller–Trumbore) avec les quelques triangles qu'elle contient.

Cette approche permet de trouver l'intersection la plus proche très rapidement en explorant l'espace de manière ordonnée. La complexité de recherche passe ainsi de  $O(N)$  à  $O(\log N)$ , rendant le rendu de scènes lourdes possible en quelques secondes.

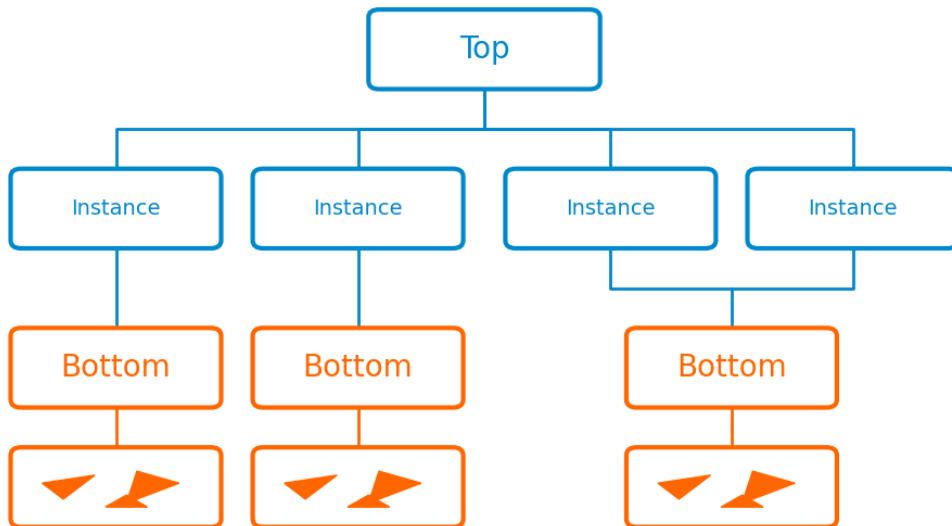


FIGURE 12 – Explications Visuel de la structure à deux niveaux de KdTree.

#### 4.1.3 Justification de l'Architecture à Deux Niveaux

J'aurais pu choisir de placer tous les triangles et tous les objets dans un unique KdTree géant. Cependant, j'ai opté pour cette approche hiérarchique pour trois raisons majeures :

1. **Gestion de l'Hétérogénéité** : Ma scène contient à la fois des primitives mathématiques pures (Sphères, Plans) et des maillages discrets (Triangles). Un arbre unique m'aurait obligé à complexifier les feuilles pour qu'elles acceptent n'importe quel type de primitive. Avec deux niveaux, le niveau global gère des pointeurs génériques vers des **Objets**, et chaque objet gère son intersection spécifique.
2. **Culling Efficace (Élagage)** : Cette architecture agit comme un filtre macroscopique. Si un rayon passe à côté de la boîte englobante du dragon, le niveau global l'élimine immédiatement (un seul test AABB). Dans un arbre unique "soupe de triangles", le rayon aurait pu traverser inutilement plusieurs nœuds spatiaux vides avant de déterminer qu'il ne touche rien.
3. **Modularité et Mémoire** : Chaque **Mesh** construit son propre arbre interne une seule fois lors du chargement. Cela sépare proprement la logique de la géométrie (le fichier .OFF) de la logique de la scène (le placement des objets). Si je devais instancier 50 maillages identiques, ils pourraient partager la même structure accélératrice en mémoire, le niveau global se contentant de gérer leurs positions.

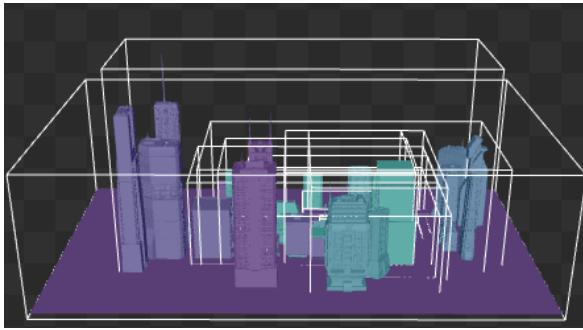


FIGURE 13 – KdTree à 1 niveau.

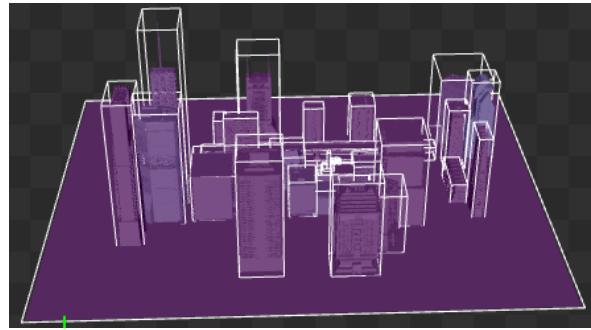


FIGURE 14 – KdTree à 2 niveaux.

FIGURE 15 – Différence entre un KdTree "simple" et "complexe".

On remarque que sur la version à deux niveaux il y a beaucoup moins de zones vides, ce qui accélère encore plus le calcul d'intersection.

#### 4.1.4 Parallèle avec le Raytracing Temps Réel (GPU)

Bien que mon implémentation sur CPU reste limitée par la nature séquentielle du processeur, l'architecture à deux niveaux correspond au standard utilisé par les cartes graphiques modernes dans le rendu temps réel.

En rendu temps réel, cette hiérarchie est nommée :

- **BLAS (Bottom-Level Acceleration Structure)** : Correspond à mon niveau local (Mesh). Il contient la géométrie brute et lourde.
- **TLAS (Top-Level Acceleration Structure)** : Correspond à mon niveau global (Scène). Il contient des "instances" légères pointant vers les BLAS.

Cette séparation est ce qui rend le raytracing possible dans les jeux vidéo actuels. Lorsqu'un personnage s'anime ou se déplace, le GPU n'a pas besoin de reconstruire l'arbre des millions de triangles qui le composent (le BLAS reste statique en mémoire). Il lui suffit de mettre à jour le TLAS, une opération extrêmement rapide car elle ne concerne que quelques boîtes englobantes. Mon moteur repose donc sur les mêmes fondements algorithmiques que ceux utilisés pour le rendu photoréaliste interactif.

Voici les références que j'ai utilisées pour comprendre et implémenter ces principes :

- **ARM Developer** : Ray Tracing - Acceleration Structures
- **Vulkan Tutorial** : Vulkan Ray Tracing - Acceleration Structures
- Ainsi que diverses vidéos YouTube sur le sujet.

## 5 Améliorations Visuelles

J'ai poussé le réalisme plus loin en intégrant plusieurs techniques avancées pour obtenir un rendu photoréaliste.

### 5.1 Modèle d'Illumination Blinn-Phong

J'ai remplacé le modèle de Phong classique par **Blinn-Phong**. Au lieu de calculer l'angle entre le vecteur réfléchi et la vue, j'utilise le vecteur médian  $H$  (Half-Vector) :

$$H = \frac{L + V}{\|L + V\|}$$

Théoriquement, cela produit des reflets spéculaires plus naturels et physiquement plausibles, notamment sur les surfaces d'eau agitées où le Phong classique peut créer des artefacts de découpe brutale.

En réalité, comme le montre la comparaison ci-dessous, la différence visuelle reste très subtile sur une image statique, influençant principalement le point de lumière.

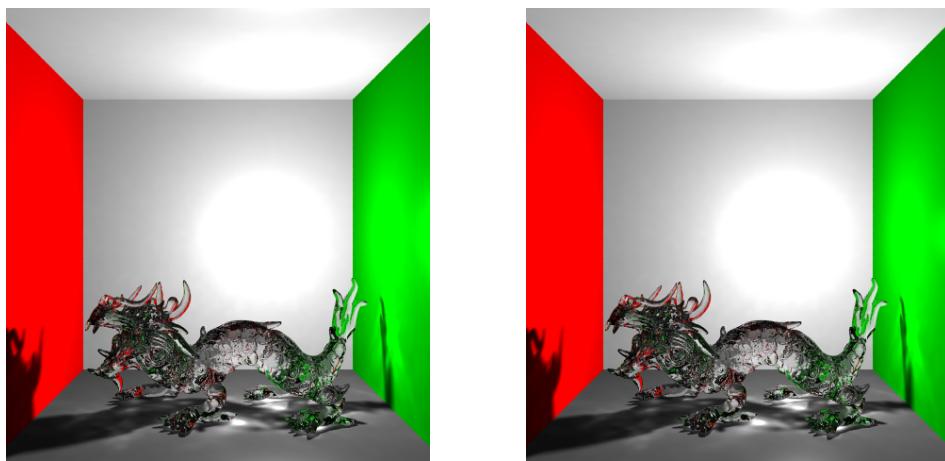


FIGURE 16 – Rendu avec  
Phong Classique.

FIGURE 17 – Rendu avec  
Blinn-Phong.

FIGURE 18 – Comparaison des modèles d'illumination : l'impact visuel est mineur.

### 5.2 Photon Mapping et Caustiques

Le raytracing (implémenté jusqu'à présent) ne peut pas calculer efficacement les caustiques (lumière focalisée par le verre). J'ai donc implémenté une première passe de **Lancer de Photons** :

1. **Émission** : Je lance des millions de photons depuis la surface de la lumière vers la scène.
2. **Stockage** : Si un photon traverse une surface spéculaire (verre) et frappe une surface diffuse, je le stocke dans une **Photon Map**. Je stocke également les photons réfléchis par le verre pour un réalisme accru.
3. **Optimisation** : J'ai implémenté un **KdTree spécifique pour les photons** (stockage de points) pour accélérer la recherche des voisins ( $k$ -NN) lors du rendu.

4. **Rendu** : Lors du lancer de rayon final, j'estime la densité de lumière en utilisant un **Filtre Conique** sur les photons proches du point d'impact, ce qui lisse le rendu des caustiques.

Cela me permet de visualiser les motifs de lumière complexes au fond de l'eau et sous la sphère de verre.



FIGURE 19 – Rendu simple : FIGURE 20 – 1M Photons : FIGURE 21 – Nuage de Ombres noires sous le verre. Apparition des caustiques. points des photons.

FIGURE 22 – Comparaison de l'apport du Photon Mapping sur le réalisme de la scène.

Voici les références que j'ai utilisées pour comprendre et implémenter ce principe :

- **Wikipédia** : Photon mapping
- **Alchetron** : Photon mapping - Jensen algorithm

### 5.3 Scène Sous-Marine et Brouillard

Pour une scène supplémentaire, j'ai créé un environnement aquatique complet :

- **Eau Procédurale** : J'ai généré un maillage de surface en combinant plusieurs fonctions sinusoïdales de fréquences différentes pour simuler la houle.
- **Brouillard** : J'applique une fonction exponentielle basée sur la distance pour simuler le trouble de l'eau et donner de la profondeur à la scène.

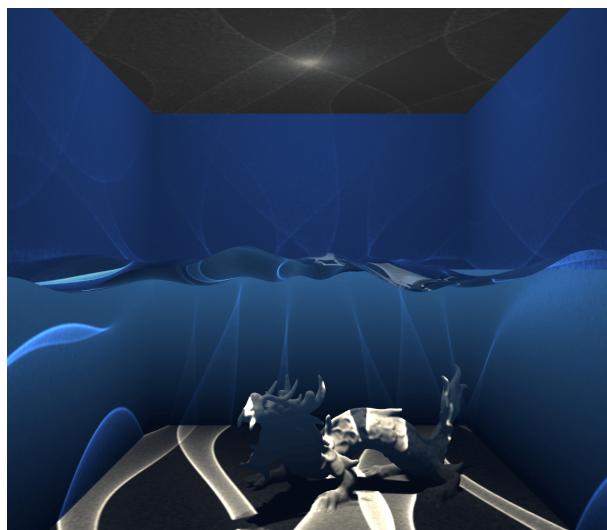


FIGURE 23 – Scène sous-marine + brouillard sous l'eau. (50M de photons)

## 5.4 Tone Mapping ACES

Enfin, pour gérer la grande plage dynamique de lumière (HDR) créée par les caustiques intenses, j'applique une courbe de **Tone Mapping ACES** à la fin du pipeline. Cela évite la saturation brutale des blancs et préserve les couleurs dans les zones très éclairées.

Sans traitement, les valeurs de pixels dépassant 1.0 (le blanc pur) sont simplement tronquées, ce qui crée des zones blanches uniformes et fait perdre les détails des caustiques. Le Tone Mapping applique une courbe non-linéaire qui compresse ces hautes lumières pour les faire rentrer dans l'espace affichable, préservant ainsi les dégradés et la saturation des couleurs.

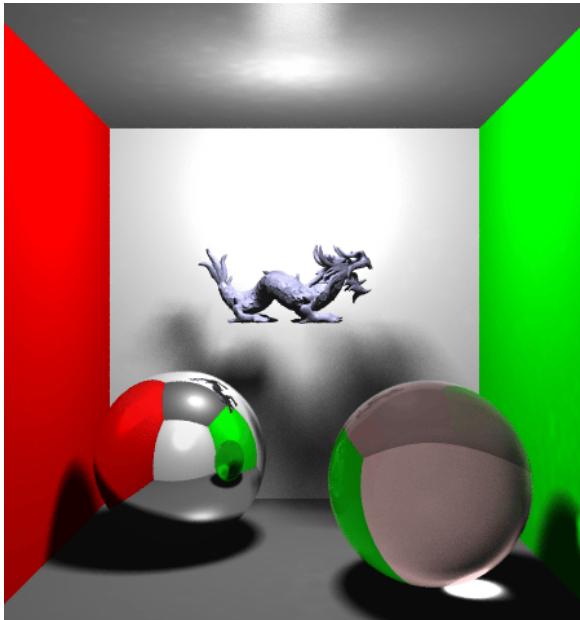


FIGURE 24 – Rendu brut : Les caustiques saturent et perdent leurs détails.

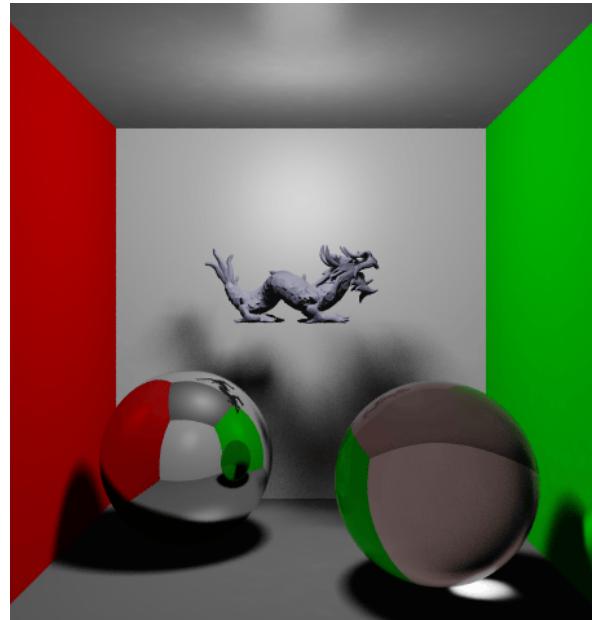


FIGURE 25 – Tone Mapping ACES : Dynamique préservée + Caustiques détaillées.

FIGURE 26 – Comparaison de l'impact du Tone Mapping sur la gestion des hautes lumières.

## 5.5 Interface Utilisateur (ImGui)

Pour faciliter l'exploration et le débogage, j'ai intégré une interface graphique interactive (ImGui) permettant de modifier les paramètres du moteur en temps réel sans avoir à recompiler le projet.

Cette interface offre un contrôle complet sur la scène :

- **Gestion de la Scène** : Sélection de la scène active (Cornell Box, etc..).
- **Éclairage** : Modification dynamique de la position de la source lumineuse ( $X, Y, Z$ ) pour observer en direct l'impact sur les photons.
- **Qualité du Rendu** : Ajustement du nombre d'échantillons par pixel (*Samples*) pour réduire le bruit de l'échantillonnage, et contrôle du nombre de photons émis pour le Photon Mapping.
- **Outils de Débogage** : Visualisation en temps réel de la Photon Map (nuage de points (touche 'p' du clavier)) et des boîtes englobantes du KdTree (touche 'k' du clavier).

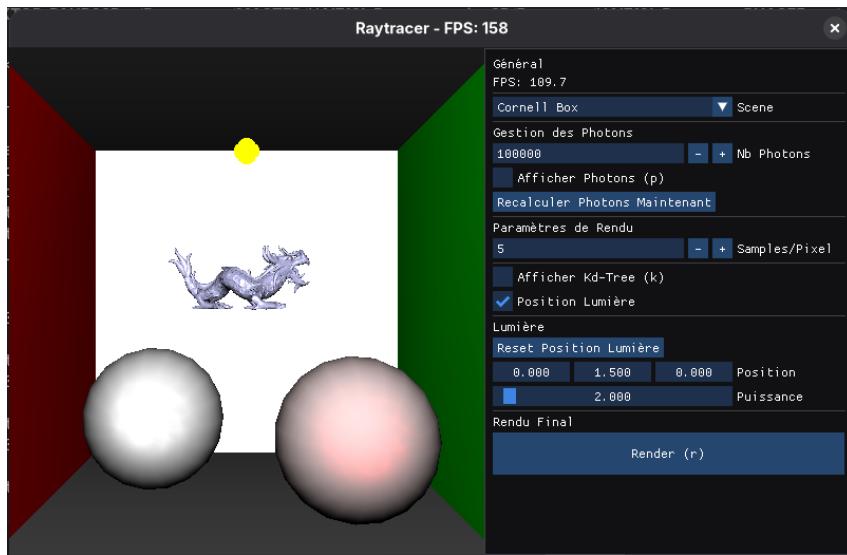


FIGURE 27 – Interface utilisateur ImGui permettant le contrôle des paramètres.

## 6 Rendu "Artistique"

Le raytracing est un domaine où la moindre erreur mathématique se traduit immédiatement par un résultat visuel, souvent chaotique, parfois surprenant. Cette section présente une sélection des meilleurs bugs ainsi que les résultats finaux une fois le moteur stabilisé.

### 6.1 Fails

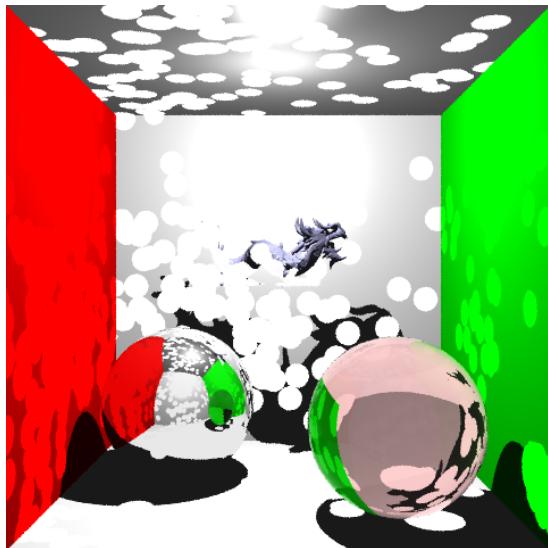


FIGURE 28 – Effet de varicelle produit par une taille de photon trop grandes.

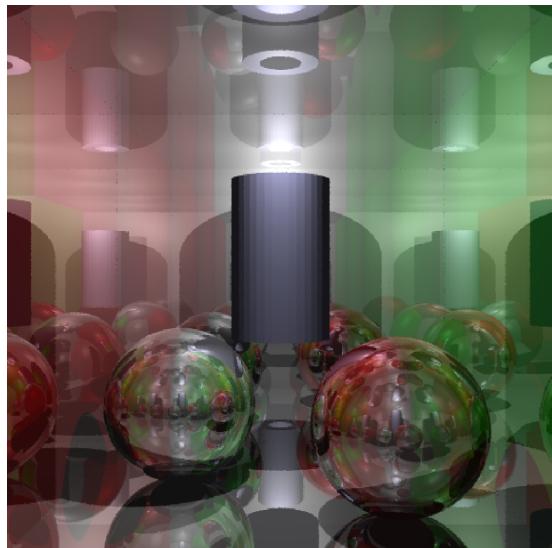


FIGURE 29 – Tous les objets étaient des miroirs sauf le maillage.

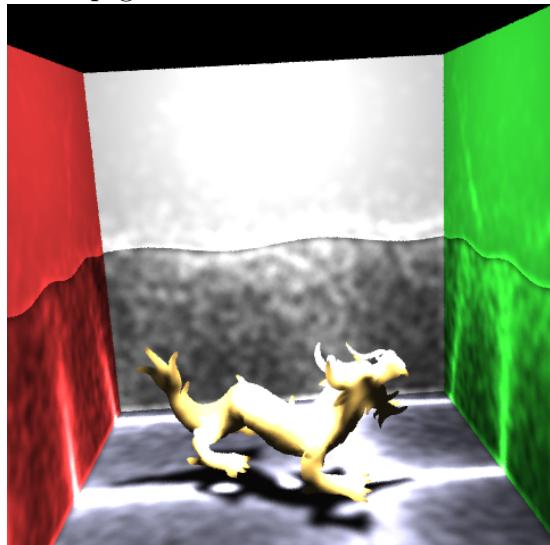


FIGURE 30 – Crédit de l'eau.

FIGURE 31 – Exemples d'artefacts visuels rencontrés lors du débogage.

## 6.2 Rendus Finaux

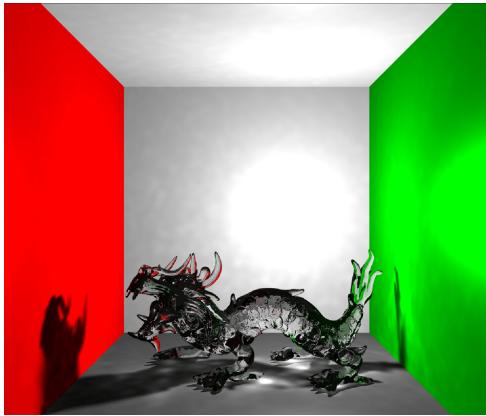


FIGURE 32 – Cornell Box :  
1 million de photons / 20 samples  
/ HD / 747 secondes

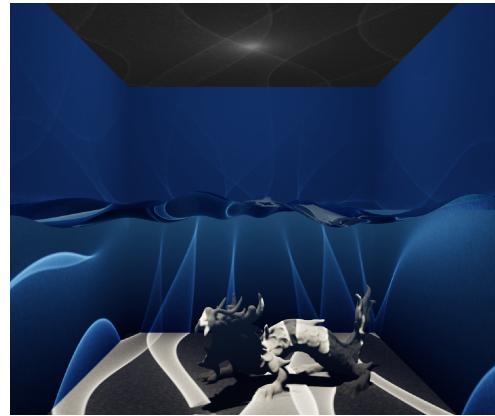


FIGURE 33 – Scène sous-marine :  
100 million de photons / 20  
samples / HD / 449 secondes

FIGURE 34 – Rendus finaux générés par le moteur.

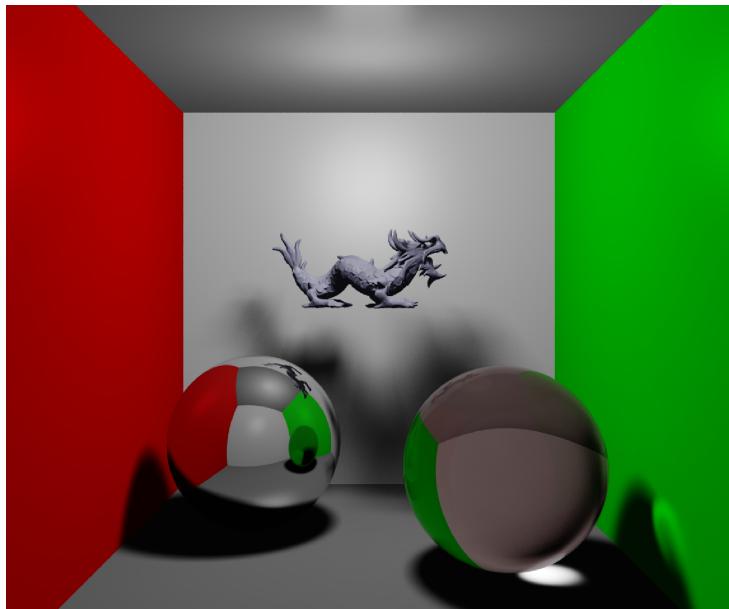


FIGURE 35 – Rendu haute qualité du Dragon (Maillage complexe + Matériau diélectrique).

10 millions de photons / 20 samples / Hd / 389 secondes

Le reste à retrouver sur la page github du projet dans la section 'Render' : Lien direct