Rapport Projet 2 : Rayracing

Paul Bastide, Alex Coudray, Julien Duron, Rémi Piau

10 mars 2019

Résumé

Lors de ce projet nous avons réalisé un programme de rendu 3d sur la base du lancer de rayon.

Table des matières

1 Introduction

1.1 Principe

Pour simuler la réalité physique, un moteur de rendu parfait devrait envoyer une infinité de rayons dans toutes les directions depuis les sources lumineuses. En étudiant ceux qui arrivent à la caméra, on pourrait savoir ce que voit cette dernière. Cependant cette méthode est inconcevable de par sa complexité : même en se limitant à un nombre fini arbitrairement grand de rayons émis, seule une petite partie de ces rayons atteindrait la caméra. Cette méthode est donc très peu efficace. Pour obtenir un rendu réaliste en effectuant le moins de calculs possibles, on peut utiliser le principe optique de retour inverse de la lumière. Le rayon ayant le même comportement, quel que soit le sens de son trajet, on peut alors considérer les rayons reçu par la camera. Ainsi en lançant un rayon par pixel de l'image à modéliser, on peut afficher cette dernière en calculant les intersections avec les formes géométriques et les propriétés des sources lumineuses et materiaux rencontrés.

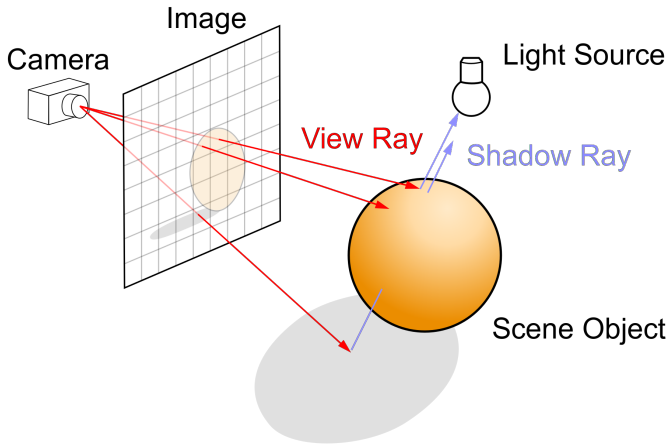


FIGURE 1 – Principe de la génération d'image par lancé de rayon (Exporté de wikipédia : Ray_trace_diagram.svg (CC-SA, wikipedia)).

1.2 Histoire et Applications

Cette méthode de rendu 3d réaliste basée sur la physique sous-jacente des phénomènes lumineux a été utilisée pour la première fois par Arthur Appel en 1968. Cependant avant 1979, les algorithmes de ray tracing se contentaient seulement de lancer des rayons puis de déterminer la couleur de l'objet pour chaque rayon sans implémentation récursive. En 1979, J. Turner Whitted proposa l'approche récursive, c'est à dire d'utiliser l'algorithme de raytracing à partir des points rencontrés pour calculer leur couleur. Cela permet notamment un rendu des reflets et de la transparence. Le raytracing connaît maintenant un grand succès et est à la base des moteurs de rendu des logiciels proffessionnels mais aussi de ceux de studios d'animation comme Pixar ou Disney.

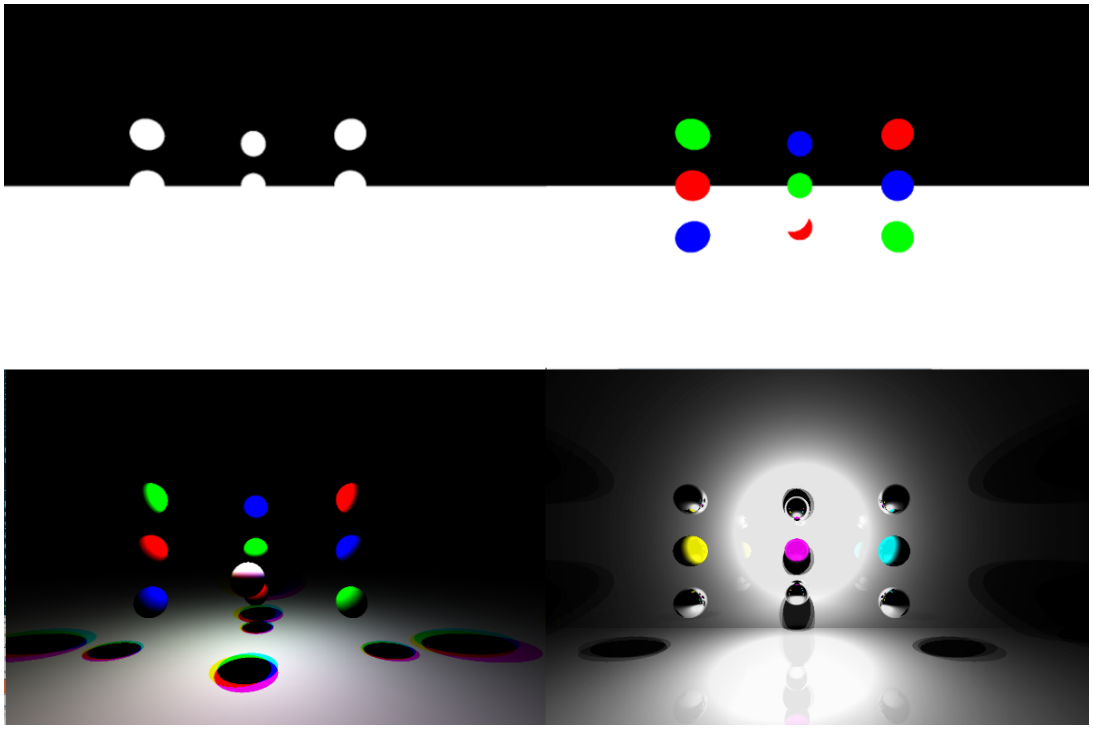


FIGURE 2 – Evolution du moteur de rendu.

2 Moteur de rendu

2.1 Evolution et conception du moteur de rendu

Nous avons réalisé le moteur de rendu par ajouts successifs. Cela nous a permis de garder une base de code robuste et claire pour les composants centraux du projet. Les différentes étapes ajoutant des fonctionnalités différentes au moteur de rendu, nous les avons gardées au sein du code. Cela permet notamment à l'utilisateur de pouvoir utiliser un rendu plus simpliste mais moins gourmand en calcul (pour verifier le placement d'un objet qui a un lourd rendu par exemple)

Nous avons d'abord réalisé un moteur de rendu simple qui affiche en blanc les pixels correspondant à l'emplacement d'un objet touché par un rayon lancé depuis la caméra (voir figure 2 en haut à gauche).

Nous avons ensuite ajouté une gestion grossière de la profondeur et des couleurs, le pixel prend maintenant la couleur de l'objet le plus proche touché par le rayon lancé depuis la caméra (voir figure 2 en haut à droite).

Nous avons ensuite rajouté une gestion de la diffusion (c.f. 2.4.2) et des sources lumineuses pour un rendu plus réaliste des scènes, nous affichant des ombres portées et des dégradés (voir figure 2 en bas à gauche).

Enfin nous avons permis au moteur de rendu de prendre en compte les phénomènes de diffusion, réflexion et transparence en même temps (voir figure 2 en bas à droite).

Le moteur de rendu dans sa version finale est un algorithme récursif dont le pseudo code est disponible en figure 2.1. On remarque que le mélange des couleurs obtenues par diffusion, réflexion et transparence est effectué en fonction des coefficients du matéraux associé à l'objet touché par le rayon.

Algorithme 1 : Rendu(rayon, n)

Entrées : rayon : rayon lumineux (demi droite),
n : nombre maximal d'appels récursifs

si $n = 0$ **alors**
  retourner NOIR
sinon
  Identifier l'objet touché par le rayon
  si *Aucun objet touché* **alors**
    retourner NOIR
  sinon
    Soit c_d le coefficient de diffusion général de l'objet touché
    Soient c_r celui de réflexion et c_t celui de transparence.
    Retourner $((c_d \times$ Couleur diffusion$) +_c (c_r \times$ Rendu(rayon réfléchi, $n-1)) +_c (c_t \times$ Rendu(rayon traversant, $n-1)))$
  fin
fin

FIGURE 3 – Algorithme du moteur de rendu

2.2 Une implémentation modulaire

Nous avons recherché une modularité maximale pour notre code afin de le rendre plus facile à mettre à jour et à utiliser. On peut voir sur la figure 4 que notre implémentation est bien étagée ce qui permet aussi de faciliter la recherche de bug en son sein.

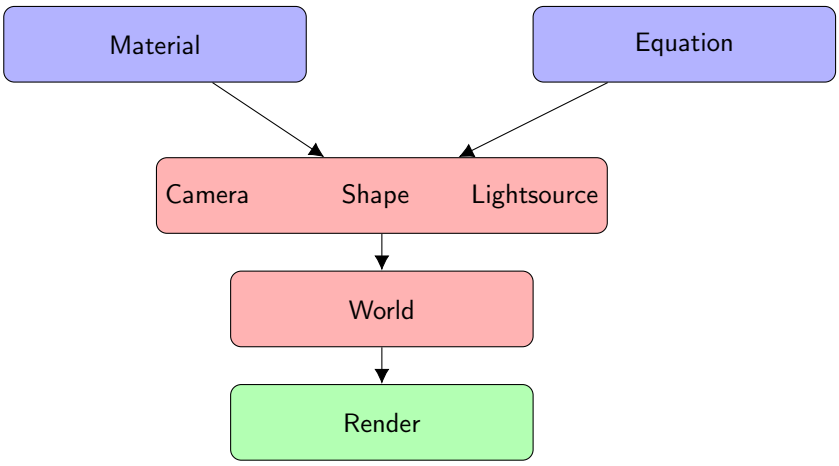


FIGURE 4 – Schéma des dépendances dans l'implémentation.

2.3 Implémentation

Nous avons souhaité donner un structure sémantique bicouche à notre logiciel :

— La couche "basse" s'occupe de la gestion des différentes formes représentables, de la caractérisation des matériaux mais aussi de la résolution des équations d'intersection. Elle comprend pour cela un module équation.
— La couche "haute" gère quant à elle le lancer de rayon et le rendu lui-même. Pour cela elle se base sur des classes `lightray` (rayon lumineux) et `Hit` (point d'impact) qui utilisent l'interface de la couche basse pour calculer la trajectoire et la couleur des rayons.

2.4 Gestion des différents phénomènes physiques

2.4.1 Colorimétrie et Synthèse de couleur

La synthèse de la couleur en un point suit plusieurs règles. Ces règles suivent les phénomènes physiques des synthèses additive et soustractive de la lumière.

La synthèse additive est utilisée lorsque l'on cherche à obtenir la somme des lumières incidentes à un point pour obtenir la lumière totale qui servira pour le rendu. En physique, c'est la somme des spectres intensité/fréquence (ou intensité/longueur d'onde) qui détermine cette synthèse. De notre côté nous discrétisons ce spectre en trois composantes : rouge, vert et bleu (décomposition RGB ou RVB en français). Nous sommons alors ces trois composantes en maximisant ces sommes à 255 (limites d'implémentation des couleurs RGB).

La synthèse soustractive est utilisée lorsque l'on cherche à obtenir la couleur perçue d'un objet illuminé par une source lumineuse colorée. En physique c'est l'utilisation du spectre d'absorption ($s_{transmis} = s_{recu} \times (1 - s_{absorption})$). Or nous avons défini une couleur à l'objet, correspondant à sa couleur en lumière blanche. Nous avons donc $s_{couleur}$ la couleur telle que $1 - s_{absorbtion} = s_{couleur}/s_{blanc}$ et donc la couleur de l'objet sous une lumière totale s_{recu} est donc : $s_{transmis} = s_{recu} \times s_{couleur}/s_{blanc}$. En discrétisant nous obtenons que chaque composante renvoyée est $C_t = C_{lumiere} \times C_{couleur}/255$ (avec C = R, G ou B). Cette méthode, bien que proche du modèle physique, donne des couleurs trop atténuées dans le rendu. Une solution donnant des rendus réalistes qui a été trouvée est de modéliser la couleur de l'objet comme un filtre max : nous obtenons ainsi pour chaque composante RGB le minimum des composantes respectives de la couleur de la lumière et de la couleur de l'objet.

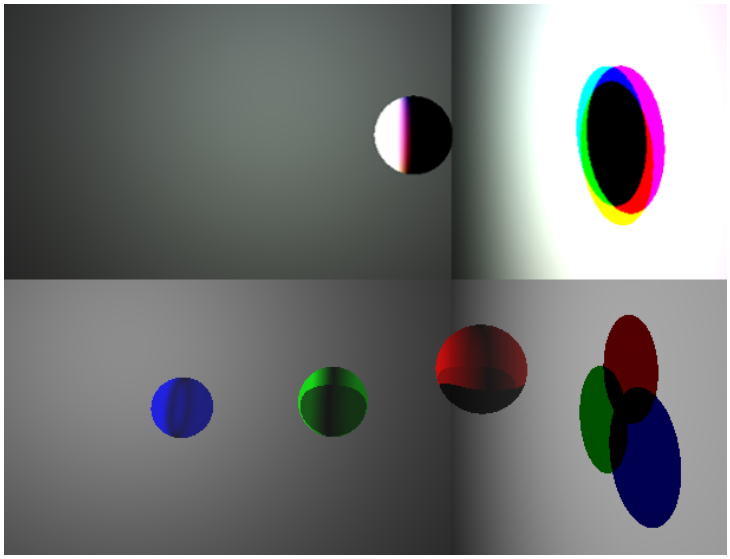


FIGURE 5 – Synthèse additive de la lumière de 3 sources dans l'ombre d'une boule blanche et soustractive d'une lumière blanche par 3 spheres transparentes colorées.

2.4.2 Diffusion

Pour avoir un rendu réaliste de la couleur des objets via la diffusion nous allons coefficienter la couleur renvoyée par des coefficients représentant divers phénomènes physiques.

Premièrement nous allons appliquer un coefficient représentant le fait que lorsqu'une surface n'est pas perpendiculaire à une source lumineuse, l'énergie reçue est diminuée d'un facteur dépendant de l'angle entre le rayon incident et la normale à la surface. On modélise cela par une diminution de l'énergie d'un coefficient $\cos(angle)$ du rayon incident.

Ensuite, nous allons appliquer un coefficient représentant l'aténuation de l'énergie de la lumière avec la distance. Comme nos sources sont considérées toutes omnidirectionnelles d'énergie E_0 et de rayon 1, on va considérer une répartition sphérique uniforme de l'énergie. L'énerie reçue par une surface d'aire A, perpendiculaire aux rayons et à une distance d si $d > 1$ est alors de $E = A \times \frac{E_0}{4 \times \pi \times d^2}$

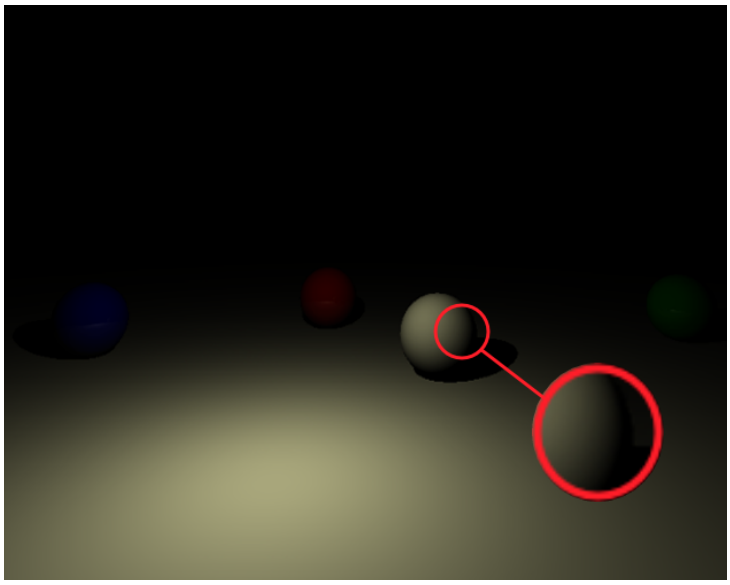


FIGURE 6 – Apparitions de dégradés (d'angle pour le zoom en rouge et distance au sol.)

2.4.3 Réflexion

Pour représenter la réflexion, nous nous basons sur la loi de Snell-Descartes. Cette loi explique que le rayon réfléchi, le rayon incident et la normale au point de contact sont dans un même plan, et que l'angle entre le rayon incident et la normale est le même qu'entre la normale et le rayon réfléchi. Pour représenter la réflexion il nous suffit juste d'ajouter à la couleur de l'objet la couleur renvoyée par le rayon réfléchi et coefficientée selon le materiau. Pour éviter d'avoir trop de calculs à faire, nous avons limité le nombre de réflexions successives qu'un rayon peut subir. Ainsi à chaque renvoi de rayon réfléchi, on réduit son compteur de durée de vie de 1, évitant des réflexions à l'infini entre deux miroirs et qui empêcherait le rendu.

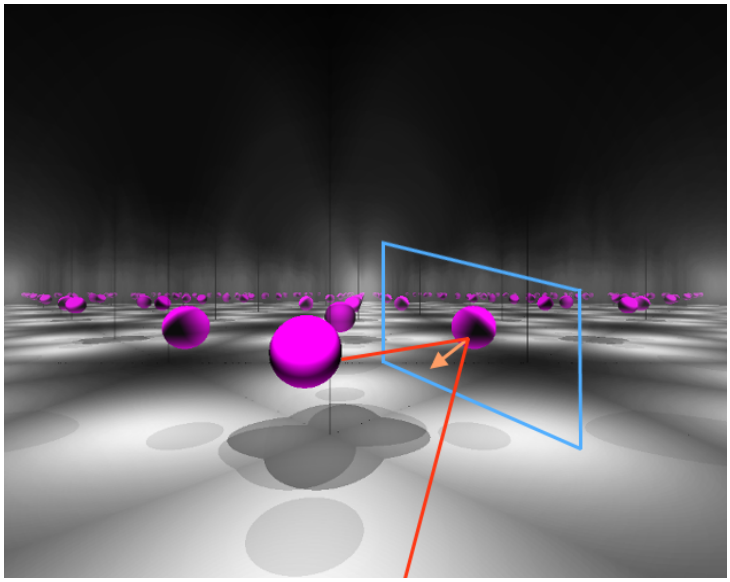


FIGURE 7 – Gestion de la réflexion (plan tangent au point de réflexion en bleu).

2.4.4 Transparence

Pour représenter la transparence, nous utilisons un mécanisme très proche de celui modélisant la réflexion. Lorsqu'un rayon incident touche un objet, nous le relançons dans la même direction et nous ajoutons à la couleur du rendu au point de contact la couleur de rendu du rayon traversant, coefficientée selon le materiau. Dans la réalité, le rayon devrait être réfracté, c'est à dire dévié selon l'indice des matériaux de l'interface de contact. Nous n'avons cependant pas eu le temps de finir d'implémenter cette réfraction.

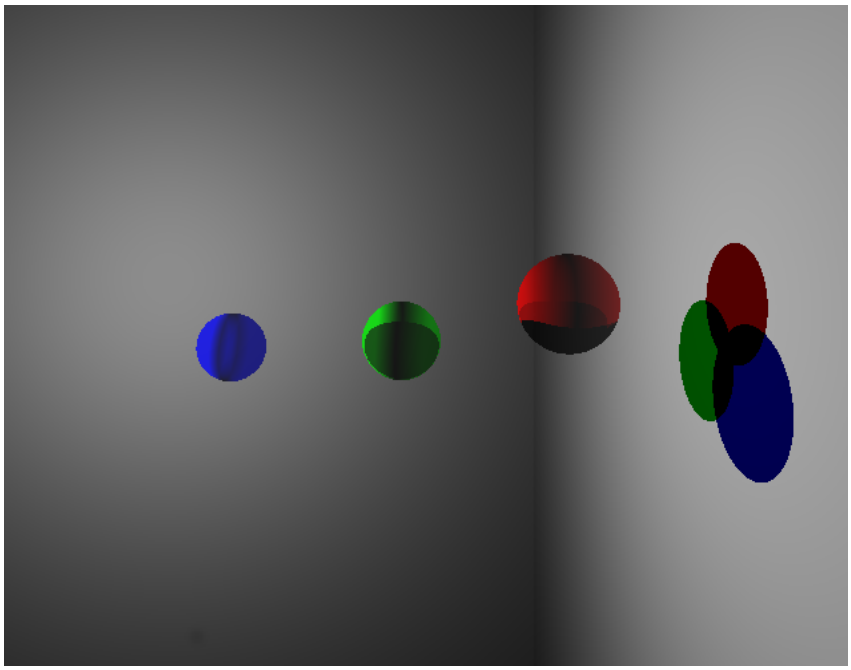


FIGURE 8 – Gestion de la transparence.

3 Gestion des formes

Nous entrons ici dans la partie calculatoire, nous précisons donc certains choix : tout d'abord, l'ensemble des calculs utilisent au plus la bibliothèque `<cmath>`, les calculs sont opérés à la précision du type `double` . Enfin, pour éviter les artéfacts nous ne considérons pas d'obstacle se situant à moins de 10^{-4} unités du départ du rayon de lumière.

Notre approche sur la gestion des formes affichables lors du rendu a été scindée en deux grandes approches : les formes décrites par une équation (polynomiale) et celles décrites par un ensemble de triangles.

3.1 Quadriques

Les Quadriques (ou surfaces quadratiques) sont des surfaces représentées par une équation cartésienne du deuxième degré. Ces équations étant résolubles avec une difficulté raisonnable, nous les avons implémentées ainsi que leurs formes dégénérées (équation de degré 2 au maximum, ce qui permet la représentation de plans).

3.1.1 Résultats

Le rendu final nous permet d'afficher, à l'aide du solveur d'équations, des quadriques quelconques (voir figure 9).

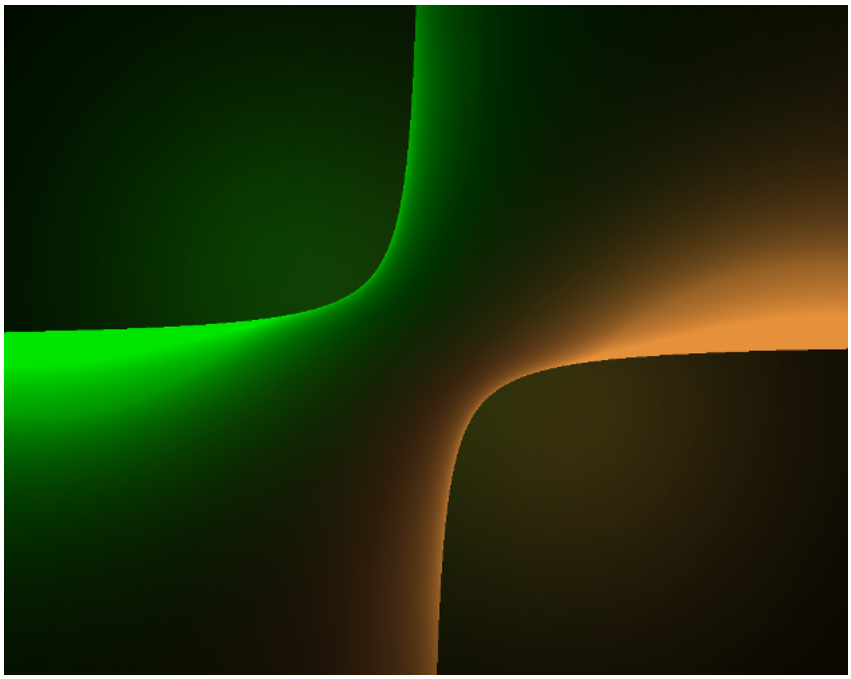


FIGURE 9 – Hyperboloïde éclairée par deux lampes.

3.1.2 Création des figures issues d'équations polynomiales

Une surface dans $\mathbb{R}^3$ est définie par une ou plusieurs équations appelées contraintes. On peut donc la voir comme l'ensemble K vérifiant :

$$K = \{x \in \mathbb{R}^3, g_1(x) = 0 \cap \cdots \cap g_k(x) = 0 \cap h_1(x) \leq 0 \cap \cdots \cap h_p(x) \leq 0\}$$

Afin de simplifier cette vision, nous considérons un cas bien moins général, mais néanmoins très intéressant : celui où $K = \{x \in \mathbb{R}^3, g(x) = 0\}$ et g est fonction polynomiale en x, autrement dit :

$$\begin{array}{rccc} g & : & \mathbb{R}^3 & \rightarrow & \mathbb{R} \\ & & (x,y,z) & \mapsto & \sum_{i=0}^{n}\sum_{j=0}^{n-i}\sum_{k=0}^{n-i-j} c_{i,j,k}x^i y^j z^k \end{array}$$

Nous noterons que, si nous voulons considérer un volume plutôt qu'une surface, nous considérerons K comme étant $\{x \in \mathbb{R}^3, g(x) \leq 0\}$. Nous représentons notre fonction g en c++ par une classe en enregistrant uniquement les coefficients $c_{i,j,k}$ et éventuellement le degré de la fonction polynomiale.

```
class FonctionPoly {
private :
    vector<vector<vector<double>>> tCoeff;
};
```

Nous avons alors un accès simple à la normale à la surface en tout point grâce à l'évaluation du gradient de g, qui se fait en $O(n^3)$ avec n le degré de g (nous calculons le gradient lors de la création de nos formes, de sorte qu'il ne reste plus qu'à l'évaluer pendant l'exécution). De même afin de récupérer le premier point d'intersection d'un rayon et d'une surface, il suffit de considérer le rayon, défini comme un point d'origine p et un vecteur directeur v et, en notant :

$$p = \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} \text{ et } v = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix}$$

Nous obtenons l'équation à résoudre en injectant les coordonnées du rayon dans l'équation $g(x,y,z) = 0$ ce qui nous donne l'équation en t :

$$\sum_{i=0}^{n}\sum_{j=0}^{n-i}\sum_{k=0}^{n-i-j} c_{i,j,k}(x_0 + tx_1)^i(y_0 + ty_1)^j(z_0 + tz_1)^k = 0$$

Dans le cadre du projet, nous avons implémenté une résolution de cette équation uniquement dans le cas $n \leq 2$, mais il est parfaitement possible d'obtenir des solutions approchées dans le cas général, par exemple grâce à la méthode de Newton. De même, il pourrait être intéressant, mais plus compliqué, de considérer des volumes définis par $K = \{x \in \mathbf{R}^3, g_1(x) \leq 0 \cap \cdots \cap g_p(x) \leq 0\}$. Bien entendu, trouver un point d'intersection avec un rayon serait beaucoup plus lourd en calculs.

Cette méthode permet la modélisation de formes continues plus complexes que de simples plans ou sphères.

3.2 Polyèdres

Un polyèdre peut être considéré comme une forme géométrique ayant des faces planes polygonales se rencontrant en des segments (arêtes). Or tout polygone est triangularisable c'est à dire décomposable en un ensemble de triangles. Ainsi tout polyèdre peut être décomposé en un ensemble de triangles.

Nous considérerons donc informatiquement les polyèdres comme une collection de triangles. Ce choix nous rapproche de la norme dans le domaine de la modélisation 3D. Cela permet par exemple d'importer des triangulations d'objets sous format .stl afin de les afficher dans le rendu.

L'obtention des points de collision entre un polyèdre et un rayon est donc la conséquence directe de l'obtention de collision entre un de ces triangles et un rayon. Afin de calculer ces points, nous appliquons la méthode suivante :

Avant tout, nous avons toujours accès aux sommets du triangle, qui serons notés p_1, p_2 et p_3 ainsi qu'à sa normale, calculée au préalable, nommée $\overrightarrow{N}$. Enfin, nous notons $\overrightarrow{v}$ le vecteur directeur du rayon, et o son origine.

L'ensemble des points susceptibles d'appartenir à la trajectoire du rayon est l'ensemble $R = \{o + t \times \overrightarrow{v} | t \in \mathbf{R}\}$

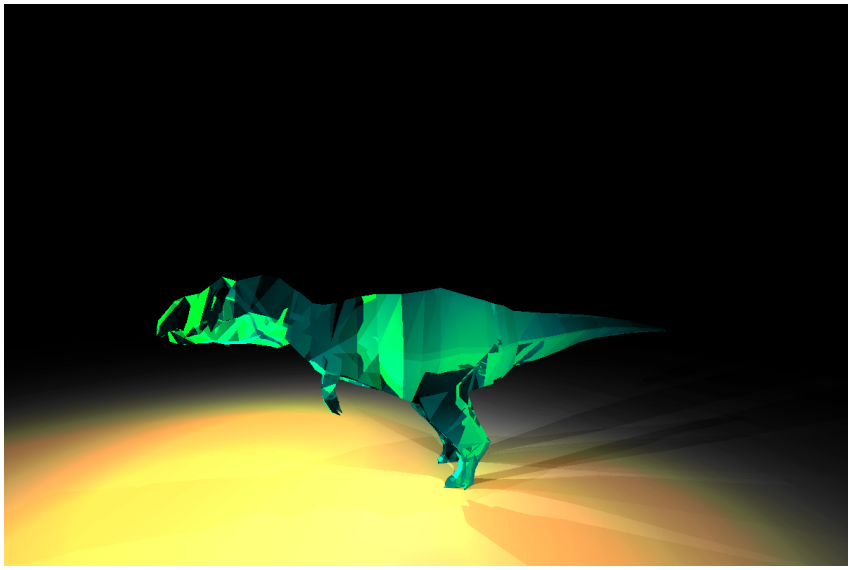
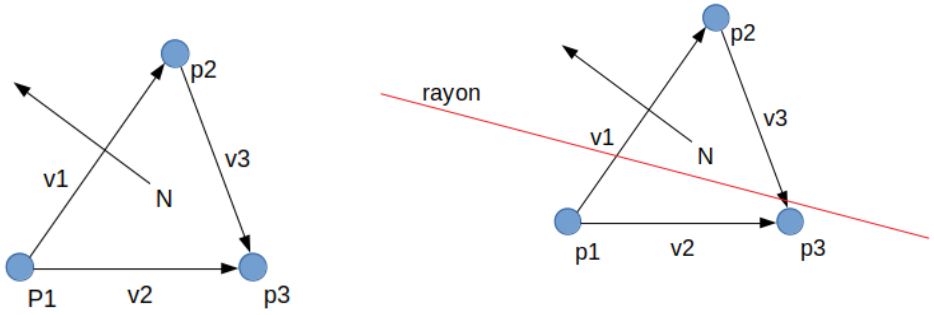


FIGURE 10 – T-Rex (polyhèdre importé par stl) sur un plan.



Nous vérifions tout d'abord que $\overrightarrow{N}$ n'est pas orthogonale à $\overrightarrow{v}$. Si jamais il s'avère que c'est le cas, nous considérons qu'il n'y a pas collision.

Sinon, nous considérons les vecteur $\overrightarrow{p_1 m}$ avec m dans R. Il existe alors un unique vecteur de cette forme orthogonal à $\overrightarrow{N}$. Le t correspondant est celui pour lequel m appartient au plan contenant le triangle. Nous notons p_0 ce point.

Il ne reste alors plus qu'à vérifier que p_0 est bien dans l'intérieur (strict) du triangle. Pour faire cela, nous considérons les vecteurs de la forme $\overrightarrow{p_i p_0}$ et vérifions qu'ils sont entre les vecteurs $\overrightarrow{p_i p_j}$ et $\overrightarrow{p_i p_k}$, avec i, j et k différents dans $[1, 2, 3]$, à l'aide de produits vectoriels.

L'intersection avec un polyèdre est définie par la première intersection avec les triangles qui le composent.

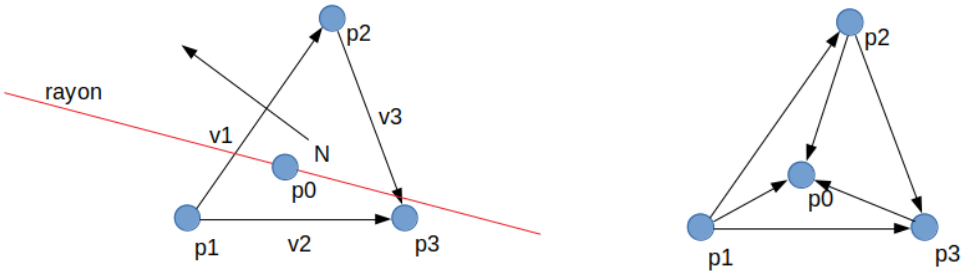


FIGURE 11 – Principe appartenance au triangle.

```
#identifier, name, parms
color, green, 0, 255, 50
material, brown_material, 0, 0, 1, 0, brown
plane, plane1, 0, 1, 0, 800, brown_materia
polyhedra, trex, models/dinosaur_model.stl, \
        700., 600., 300., green_material
lightsource, lightsource1, 600., -100.,300.,\
        200.0, white
```

FIGURE 12 – Exemple de syntaxe du fichier csv d'un monde.

4 Expérience utilisateur

4.1 Chargement du monde

Lors de notre projet nous avons mis au point un système de chargement du monde à afficher à partir d'un fichier csv (comma separated value, i.e. valeurs séparées par des virgules). Le modèle de description est simple : d'abord un identifiant donnant le type de l'objet à instancier, puis son nom (de préférence unique ce qui permet de le réutiliser dans le fichier) et enfin les différents paramètres nécessaires à l'instanciation du type en question (voir la figure 12 pour des exemples). Cette méthode nous permet de charger dynamiquement les mondes sans avoir à recompiler le programme et rend leur modification plus facile.

4.2 Gestion des arguments

Nous avons aussi réalisé une gestion du programme en ligne de commande pour permettre un réglage plus rapide et pratique pour l'utilisateur des différents aspects du rendu comme le monde à charger, la sauvegarde d'une image ou encore le type de moteur de rendu à utiliser.

5 Conclusion

La problèmatique du ray tracing étant extrêmement vaste nous avons choisi de developper ce projet non pas dans une optique d'optimisation mais plutôt dans un but de généralité. Le code posséde une base de solide qui permet théoriquement l'implémentation d'un algorithme pour une forme quelconque. Les formes réellement implémentées se veulent elles aussi les plus générales possibles. L'idée

d'une interface utilisateur aussi complète est née de cette même volonté généraliste. Ainsi nous nous sommes détournés de problématiques citées ci-dessous qui nous semblaient plus secondaires à la lumière de notre objectif mais qui sont toutes autant des enjeux majeurs pour l'implémentation d'un logiciel de raytracing complet.

Voici une liste non-exhaustive d'améliorations non implémentées dont nous développerons brièvement certains points :

— Anti-aliasing
— Accerleration de la vitesse du rendu
— Camera paramétrable
— Résolution pour des formes plus générales.

L'anti-aliasing est une méthode consistant à adoucir les changements de couleures ou de textures brutaux, qui une fois pixelisés créent des formes désagréables. Pour adoucir ces contour, le but est donc de répartir la discontinuité de couleur sur plusieurs pixels.

Afin d'accélérer la vitesse de rendu, plusieurs pistes sont intéressantes. Tout d'abord, la parallélisation des calculs, car le lancé d'un rayon n'interfère jamais (dans notre modèle bien entendu) avec celui d'un autre rayon. Il serait donc très intéressant de faire appel à la puissance des cartes GPU dans ce domaine. D'autre part, lors du lancé d'un rayon, nous faisons des calculs d'interaction avec l'ensemble des formes présentes dans le monde créé. Et ce à chaque rebond. Ce qui crée un algorithme dont la complexité est en $O(n)$, avec n le nombre d'objets sur la scène, pour chaque rebond. Nous devrions cependant pouvoir concevoir des algorithmes permettant de réduire cette complexité, en ne considérant que des calculs d'interactions "utiles", par exemple en classant les formes par position à l'aide d'un arbre.

L'anti-aliasing est une méthode consistant à adoucir les changement de couleures ou de textures brutaux, qui une fois pixelisés créent des formes désagréables. Pour adoucir ces contour, le but est donc de répartir la discontinuité de couleur sur plusieurs pixels.

Afin d'accélérer la vitesse de rendu, plusieurs pistes sont intéressantes. Tout d'abord, la parallélisation des calculs, car le lancé d'un rayon n'interfère jamais (dans notre modèle bien entendu) avec celui d'un autre rayon. Il serait donc très intéressant de faire appel à la puissance des cartes GPU dans ce domaine. D'autre part, lors du lancé d'un rayon, nous faisons des calculs d'interaction avec l'ensemble des formes présentes dans le monde créé. Et ce à chaque rebond. Ce qui crée un algorithme dont la complexité est en $O(n)$, avec n le nombre d'objets sur la scène, pour chaque rebond. Nous devrions cependant pouvoir concevoir des algorithmes permettant de réduire cette complexité, en ne considérant que des calculs d'interactions "utiles", par exemple en classant les formes par position à l'aide d'un arbre.