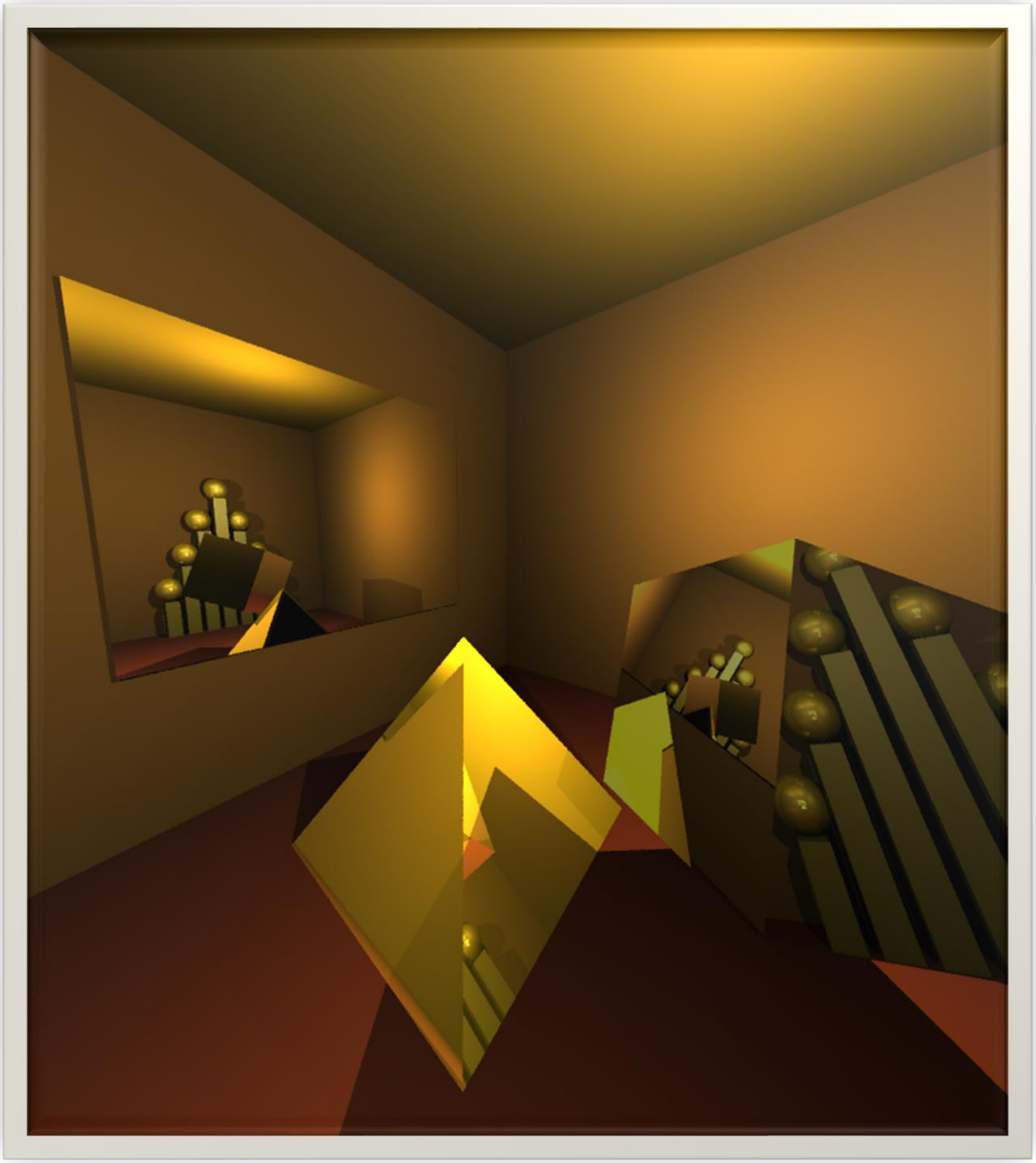
Projet de synthèse d'image

LANCER DE RAYONS

-

RAPPORT



Amany MILED
Samy WADAN

Université Gustave Eiffel
2019 – 2020

Master 1 Informatique
Groupe Apprentis

Table des matières

Description

L'application consiste à synthétiser une image, en lançant des rayons depuis un point de vue vers la scène, en passant chacun par un point différent, un point définissant un pixel de l'image finale.
Si un rayon intersecte un objet de la scène, alors la couleur du pixel associé devient la couleur de l'objet illuminé au point d'intersection.

Fonctionnalités

Les fonctionnalités implémentées dans l'application sont celles demandées par le niveau 1, 2 et 3 du sujet à l'exception de « l'extension du format de sortie », seules des images au format PPM peuvent être générées.

Voici ce qui a été rajouté en plus :
- Les formes **Plan**, **Ellipsoïde**, **Cube** et **Pavé droit** (utilisation des matrices en coordonnées homogènes pour passer vers une forme canonique puis calcul des intersections).
- Le passage en **multithreading** qui permet de partager le calcul de l'image en plusieurs threads.
- Possibilité d'**activer/désactiver** l'éclairage, la réflexion et la transparence pendant le rendu interactif.
- Ajout d'un **seuil minimal pour l'éclairage**, ce qui permet d'éviter que les zones non illuminées soient complètement noires.

L'ensemble des paramètres a une valeur par défaut qui peut être modifié dans la fonction *initParametres()* dans le fichier *main.c*.

Format et chargement de la scène

La construction de la scène se fait via un fichier de configuration **JSON**.
Il doit obligatoirement comporter les clés suivantes :

// Position de la caméra
camera_position : Tableau de 3 décimaux

// Orientation de la caméra
camera_orientation : Tableau de 3 décimaux

// Distance entre la caméra et le centre de la grille
distance_focale : Décimal

// Taille en largeur de la grille
definition_largeur : Décimal

// Taille en hauteur de la grille
definition_hauteur : Décimal

// Nombre de pixels de l'image en largeur
resolution_largeur : Entier

// Nombre de pixels de l'image en hauteur
resolution_hauteur : Entier

// Vecteur d'inclinaison de la grille
inclinaison_grille : Tableau de 3 décimaux

// Description des formes présentes dans la scène. Une entrée dans le tableau ⇔ une forme.
formes : Tableau d'objets JSON (pas de limite)

// Description des lumières présentes dans la scène. Une entrée dans le tableau ⇔ une lumière.
lumieres : Tableau d'objets JSON (pas de limite)

⚠ **Les valeurs données à *camera_orientation* et à *grille_inclinaison* doivent correspondre à des vecteurs [x, y, z] unitaires.**

Format des formes

Une forme est décrite par un objet JSON, il possède au moins la clé obligatoire type, dont la valeur correspond au nom de la forme.
Cette valeur peut prendre :
sphere – rectangle – triangle – cylindre – plan – ellipsoide – cube – pave droit

Selon le type de la forme, d'autres clés doivent être obligatoirement renseignées :

Format spécifique à chaque forme

Sphere

// Centre de la sphère
centre : Tableau de 3 décimaux

// Rayon de la sphère
rayon : Décimal

Rectangle

// Centre du rectangle
centre : Tableau de 3 décimaux

// Normale du rectangle (comme pour le plan)
normale : Tableau de 3 décimaux

// Largeur du rectangle
largeur : Décimal

// Longueur du rectangle
longueur : Décimal

Triangle

// 1er point du triangle
pointA : Tableau de 3 décimaux

// 2ème point du triangle
pointB : Tableau de 3 décimaux

// 3ème point du triangle
pointC : Tableau de 3 décimaux

Cylindre

// Point central de la base du cylindre
centre : Tableau de 3 décimaux

// hauteur du cylindre, en partant de la base
hauteur : Tableau de 3 décimaux

// Rayon du cylindre
rayon : Décimal

Plan

// « Centre » du plan, le point par lequel passe le plan
centre : Tableau de 3 décimaux

// Vecteur normal au plan
normale : Tableau de 3 décimaux

Ellipsoïde

// Centre de l'ellipsoïde
centre : Tableau de 3 décimaux

// Rayon sur l'axe x
rayonA : Décimal

// Rayon sur l'axe y
rayonB : Décimal

// Rayon sur l'axe z
rayonC : Décimal

Cube

// Centre du cube
centre : Tableau de 3 décimaux

// Taille de chaque face du cube
taille : Décimal

Pavé droit

// Centre du pavé droit
centre : Tableau de 3 décimaux

// Largeur du pavé droit
largeur : Décimal

// Hauteur du pavé droit
hauteur : Décimal

// Profondeur du pavé droit
profondeur : Décimal

Matériau

Une forme peut posséder plusieurs caractéristiques optionnelles (c'est-à-dire ayant une valeur par défaut si non renseigné).
Chacune d'entre elle peut être modifié en renseignant leur valeur :

// Couleur de la forme, décrit par un code RGB hexadécimal. Exemple FF0000 pour du rouge. (Par défaut 000000)
couleur : Chaîne de caractères

// Couleur spéculaire de la forme, décrit par un code RGB hexadécimal. Exemple 0000FF pour du bleu. (Par défaut 000000)
couleur : Chaîne de caractères

// Brillance (Par défaut 0)
brillance : Entier

// Coefficient de diffusion de la couleur de l'objet. Entre 0 et 1. (Par défaut 1)
coeff_diffusion : Décimal

// Coefficient de réflexion de l'objet. Entre 0 et 1. (Par défaut 0)
reflexion : Décimal

// Coefficient de réfraction (transparence) de la couleur de l'objet. Entre 0 et 1. (Par défaut 0)
coeff_refraction : Décimal

// Indice de réfraction. (Par défaut 1)
indice_refraction : Décimal

Rotation

Chaque forme peut également tourner autour de chaque axe.
Il faut pour cela ajouter une clé « rotation » dont la valeur est un tableau de 3 décimaux. Il représente le vecteur de rotation.

Par exemple :
rotation : [π/2, 0, -π/6]

A pour effet d'effectuer une rotation de π/2 radians autour de l'axe X et de -π/6 radians autour de l'axe Z.
La rotation vaut par défaut (0, 0, 0) si la clé n'est pas renseignée.

Format des lumières

Une lumière est décrite par un objet JSON. Il n'existe qu'un type de lumière : la lumière ponctuelle qui éclaire dans toutes les directions.
Il y a 2 clés à spécifier obligatoirement :

// Définis la position de la source de lumière
position : Tableau de 3 décimaux

// Intensité lumineuse
intensite : Décimal

Rendu interactif

Voici les touches qui permettent d'effectuer une action pendant le rendu interactif :

Contrôle de la caméra

Z : Avancer la caméra dans sa direction d'orientation (Zoomer).
S : Reculer la caméra dans sa direction inverse d'orientation (Dézoomer).
Q – **D** : Déplacer la caméra à gauche et à droite. Relativement à l'orientation de la caméra.

LEFT CONTROL : Faire descendre la caméra.
ESPACE : Faire monter la caméra.

TOUCHES DIRECTIONNELLES : Orienter la caméra.

Réglages

E : Activer/désactiver l'**E**clairage.
R : Activer/désactiver les rayon **R**éfléchis
T : Activer/désactiver la **T**ransparence.
O : Activer/désactiver l'**O**mbrage.

Autre

ENTRER : Enregistrer l'image actuelle dans un fichier PPM dont le nom a été donné à l'exécution du programme.

Compilation et exécution du code

Compilation

Le programme utilise les bibliothèques suivantes :

- OpenGL – GLEW
- SDL2

S'assurer de les posséder avant de compiler.
Note : Je ne suis pas expert en cmake, mais pour compiler sur Windows j'ai dû remplacer la ligne
target_link_libraries(lray PRIVATE ${GLEW_LIBRARIES} ${SDL2_LIBRARIES} ${OPENGL_LIBRARIES})
par
target_link_libraries(lray PRIVATE GLEW::GLEW SDL2::SDL2 SDL2::SDL2main)
et retirer la ligne d'inclusion du package OpenGL.

Du code externe a également été utilisé : (header-only, pas besoin d'installation)
- **rapidjson** : Outil permettant de parser les fichiers JSON
- **cxxopts** : Outil permettant de parser les lignes de commandes

Se placer dans le répertoire *build* initialement vide et saisir la commande `cmake ../`
Ceci aura pour effet de générer le makefile dans le répertoire courant.
Compiler avec `make`

Exécution

La compilation génère un exécutable **lray** dans le dossier *bin*.

Les arguments sont ceux spécifiés dans sujet, à l'exception de l'argument « -ps » qui s'écrit « -p » ou « --ps ».
Il est également possible d'utiliser l'option –force ou -f pour forcer le rendu interactif (par défaut uniquement pour le niveau 2).

L'argument –help ou -h est également disponible.

Architecture

L'application est codée en C++ et possède les modules suivants :

Scène

Regroupe tous les objets de la scène : les formes, les lumières, la caméra ainsi que sa grille.

Chaque forme hérite de la classe abstraite Forme qui impose de créer une méthode d'intersection avec un rayon, d'initialiser les matrices de transformations directes et indirectes et possède un objet Materiau.

Géométrie

Regroupe les objets mathématiques : vecteurs, points, matrices, rayons.

Outils

On y trouve entres autres le parseur JSON, l'objet Materiau ou la classe permettant de générer une image en sortie.

OpenGL

La classe SceneOpenGL utilise la bibliothèque SDL2 pour créer une fenêtre et afficher pixel par pixel le rendu du lancer de rayons.
Elle possède un attribut de type Input permettant d'écouter les évènements du clavier.

Application

Comporte les fonctions principales du projet, c'est dans la classe Application que l'on construit le lancer de rayons.
La méthode principale *LancerRayons()* implémente la boucle du lancer de rayons et appelle indirectement les fonctions d'éclairage, de lancer de rayons réfléchis et de lancer de rayons réfractés.

Main

Point d'entrée du programme.
Commence par parser le fichier JSON en entrée puis lance la fonction d'Application selon si l'on veut rendre l'image dans un fichier ou l'afficher interactivement.

Performances et optimisations possibles

Voici les résultats tirés du profilage du programme après exécution.
Les calculs se font sur CPU et le model de processeur utilisé est :
Intel® Core™ i5-7600K CPU @ 3.80GHz (4 CPUs), ~3.8GHz

Le test ci-dessous est effectué en mode debug pour profiler le programme, donc les performances sont un peu amoindries.

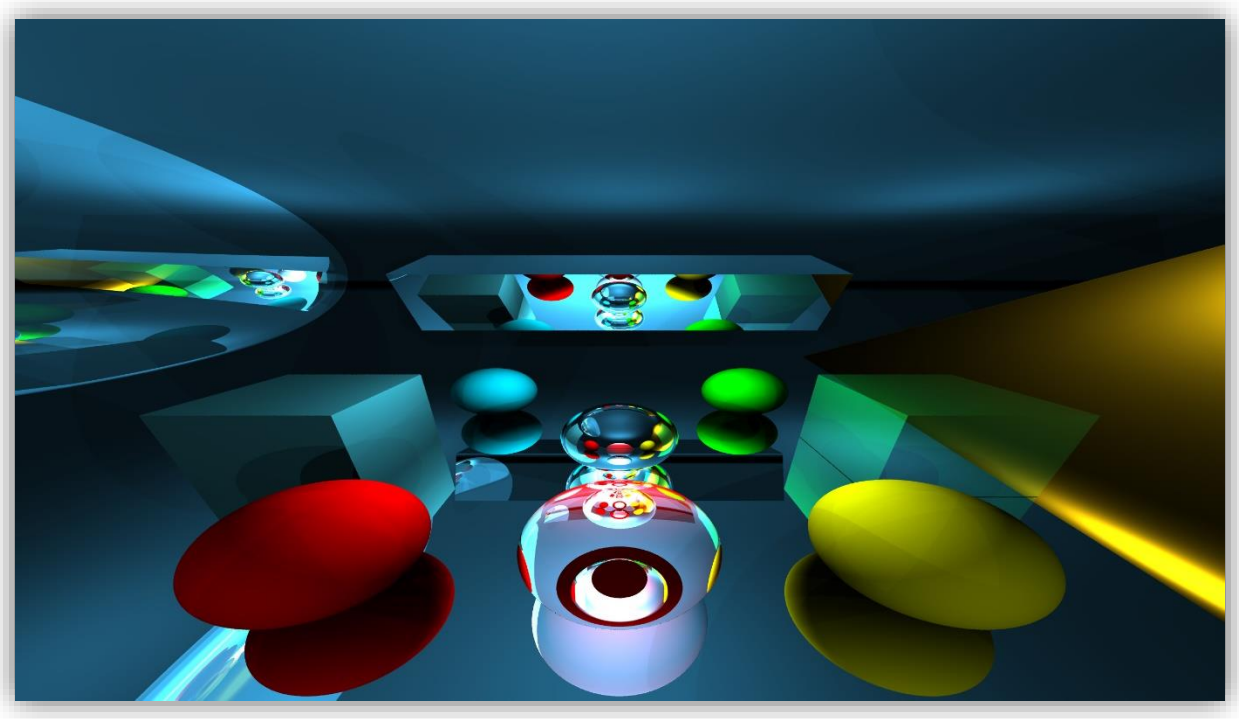
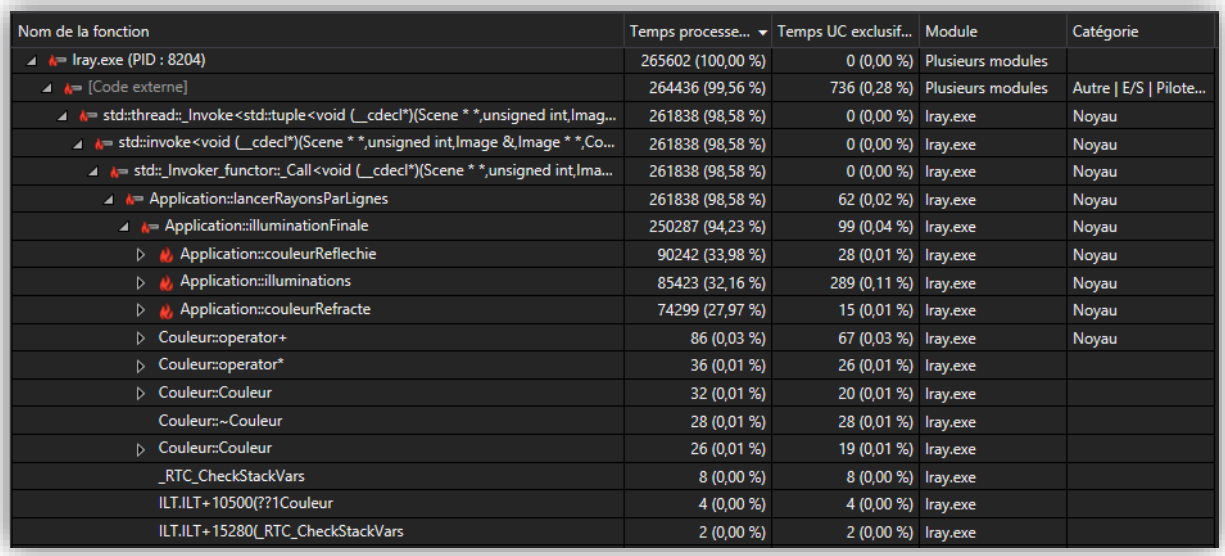


Image générée depuis la scène config2 1

La scène utilisée est celle décrite par le fichier *config2.json*. Elle possède 16 formes et 9 sources de lumières ponctuelles.
L'image précédente a une résolution de 2560*1440 pixels
Il y a également beaucoup de réflexion (estimation : ~50% de l'image) avec potentiellement une infinité de rebonds (la sphère centrale et le pavé orienté vers les sphères sur le dessus).
Il y a également de la transparence (estimation : ~20% de l'image) : l'ellipsoïde à gauche qui est transparente et reflète les rayons, pareil pour la sphère du bas au milieu, et le cube à droite est seulement transparent.
Le nombre de rebonds maximum pour la réflexion et la réfraction est fixée à 16.

Voici les résultats de performance pour générer cette image.
Résolution : 500*500
Nombre de threads : 1 (L'outil de profilage ne fonctionne apparemment pas correctement quand plusieurs threads sont actifs…)
Pixel sampling : 8



Nom de la fonction	Temps processe...	Temps UC exclusif...	Module	Catégorie
Iray.exe (PID : 8204)	265602 (100,00 %)	0 (0,00 %)	Plusieurs modules	
[Code externe]	264436 (99,56 %)	736 (0,28 %)	Plusieurs modules	Autre | E/S | Pilote...
std::thread::_Invoke<std::tuple<void (__cdecl*)(Scene * *,unsigned int,Imag...	261838 (98,58 %)	0 (0,00 %)	Iray.exe	Noyau
std::invoke<void (__cdecl*)(Scene * *,unsigned int,Image &,Image * *,Co...	261838 (98,58 %)	0 (0,00 %)	Iray.exe	Noyau
std::_Invoker_functor::_Call<void (__cdecl*)(Scene * *,unsigned int,Ima...	261838 (98,58 %)	0 (0,00 %)	Iray.exe	Noyau
Application::lancerRayonsParLignes	261838 (98,58 %)	62 (0,02 %)	Iray.exe	Noyau
Application::illuminationFinale	250287 (94,23 %)	99 (0,04 %)	Iray.exe	Noyau
Application::couleurReflechie	90242 (33,98 %)	28 (0,01 %)	Iray.exe	Noyau
Application::illuminations	85423 (32,16 %)	289 (0,11 %)	Iray.exe	Noyau
Application::couleurRefracte	74299 (27,97 %)	15 (0,01 %)	Iray.exe	Noyau
Couleur::operator+	86 (0,03 %)	67 (0,03 %)	Iray.exe	Noyau
Couleur::operator*	36 (0,01 %)	26 (0,01 %)	Iray.exe	
Couleur::Couleur	32 (0,01 %)	20 (0,01 %)	Iray.exe	
Couleur::~Couleur	28 (0,01 %)	28 (0,01 %)	Iray.exe	
Couleur::Couleur	26 (0,01 %)	19 (0,01 %)	Iray.exe	
_RTC_CheckStackVars	8 (0,00 %)	8 (0,00 %)	Iray.exe	
ILT.ILT+10500(??1Couleur	4 (0,00 %)	4 (0,00 %)	Iray.exe	
ILT.ILT+15280(_RTC_CheckStackVars	2 (0,00 %)	2 (0,00 %)	Iray.exe	

Profilage arborescence config2.json 1

Sans surprise, ce sont les fonctions d'éclairage, de réflexion et de réfraction qui représentent la plupart du temps de calcul (~92%) à parts égales.

La fonction d'illumination est découpée en 2 parties :
- Détermination si un point doit être éclairé (ombrage)
- Calcul de l'éclairage

Cette 1ère partie (calcul de l'ombrage) représente ~80% de la méthode.

Il faut prendre en compte qu'un point résultant d'un rayon réfléchis ou réfracté est calculé en réappelant la fonction *IlluminationFinale()*.
➔ Dans les 34% du calcul de la réflexion, 15% est dédié au calcul d'éclairage, le reste sont les autres rebonds du rayon, et dans ces autres calculs, la moitié également est dédié au calcul d'éclairage.
➔ Idem pour le calcul de la réfraction.

Ainsi le véritable temps de calcul du programme est défini par le calcul d'éclairage, et donc en grande partie par le calcul de l'ombrage. Voir la section d'après.

Nom de la fonction	Temps processeur total [unité,...	Temps UC exclusif...	Module
Iray.exe (PID : 8204)	265602 (100,00 %)	0 (0,00 %)	Iray.exe
[Code externe]	264738 (99,67 %)	11348 (4,27 %)	Plusieurs modules
Application::lancerRayonsParLignes	261838 (98,58 %)	62 (0,02 %)	Iray.exe
std::_Invoker_functor::_Call<void (__cdecl*)(Scene * *,unsigne...	261838 (98,58 %)	0 (0,00 %)	Iray.exe
std::invoke<void (__cdecl*)(Scene * *,unsigned int,Image &,I...	261838 (98,58 %)	0 (0,00 %)	Iray.exe
std::thread::_Invoke<std::tuple<void (__cdecl*)(Scene * *,unsi...	261838 (98,58 %)	0 (0,00 %)	Iray.exe
Application::illuminationFinale	250287 (94,23 %)	242 (0,09 %)	Iray.exe
Application::illuminations	230384 (86,74 %)	665 (0,25 %)	Iray.exe
Application::illumination	228535 (86,04 %)	1884 (0,71 %)	Iray.exe
Application::estIllumine	206407 (77,71 %)	3659 (1,38 %)	Iray.exe
Application::couleurReflechie	131272 (49,42 %)	59 (0,02 %)	Iray.exe
Application::couleurReflechieAux	131236 (49,41 %)	230 (0,09 %)	Iray.exe
Application::couleurRefracte	89890 (33,84 %)	52 (0,02 %)	Iray.exe
Application::couleurRefracteAux	89864 (33,83 %)	212 (0,08 %)	Iray.exe
Triangle::intersection	63138 (23,77 %)	2484 (0,94 %)	Iray.exe
Cube::intersection	57408 (21,61 %)	2766 (1,04 %)	Iray.exe
Vecteur::unitaire	52082 (19,61 %)	15814 (5,95 %)	Iray.exe
Vecteur::prodScalaire	42948 (16,17 %)	42898 (16,15 %)	Iray.exe
Sphere::intersection	42519 (16,01 %)	7217 (2,72 %)	Iray.exe
Triangle::intersectionCanonique	42515 (16,01 %)	11628 (4,38 %)	Iray.exe
Plan::intersection	40002 (15,06 %)	2600 (0,98 %)	Iray.exe
Cube::intersectionCanonique	39011 (14,69 %)	9329 (3,51 %)	Iray.exe
Application::interPlusProche	25966 (9,78 %)	662 (0,25 %)	Iray.exe
Rayon::Rayon	25312 (9,53 %)	11545 (4,35 %)	Iray.exe
PaveDroit::intersection	24927 (9,39 %)	1147 (0,43 %)	Iray.exe
Plan::intersectionCanonique	19252 (7,25 %)	4175 (1,57 %)	Iray.exe
Vecteur::Vecteur	16398 (6,17 %)	11431 (4,30 %)	Iray.exe
Matrice::operator*	14157 (5,33 %)	13003 (4,90 %)	Iray.exe
Vecteur::Vecteur	13591 (5,12 %)	13571 (5,11 %)	Iray.exe
Matrice::operator*	13203 (4,97 %)	12030 (4,53 %)	Iray.exe
Ellipsoide::intersection	11542 (4,35 %)	863 (0,32 %)	Iray.exe
[Appel externe] ucrtbased.dll	8609 (3,24 %)	8609 (3,24 %)	ucrtbased.dll
abs	7606 (2,86 %)	6382 (2,40 %)	Iray.exe

Profilage par fonction config2.json 1

Lorsqu'on trie les fonctions par temps d'exécution, on observe quelque chose d'inattendu :
La méthode *unitaire()* sur un Vecteur représente à elle toute seule presque **20% du temps de calcul**.
C'est-à-dire qu'un 5ème du temps du programme consiste à normaliser un vecteur. Ceci est un défaut du programme, malgré le fait qu'il faut presque toujours travailler des vecteurs normalisés, il est sûrement possible d'optimiser le code en limitant l'appel à cette méthode, car même si un vecteur est déjà unitaire, il est possible qu'on appelle quand même la méthode de normalisation. (La méthode effectue une racine carrée et 3 produits).
On pourrait par exemple normaliser la direction du rayon une fois puis calculer l'intersection avec toutes les formes plutôt que de normaliser la direction du rayon à chaque calcul d'intersection…

Maintenant que les fonctions sont triées par durée cumulée, on peut confirmer ce qui a été expliqué précédemment : la fonction d'éclairage (*illumination()*) représente 86% du temps de calcul du programme, et le calcul d'ombrage (*estIllumine()*) représente **~78% du temps de calcul**.
On rappel que le calcul d'ombrage consiste à déterminer pour chaque source de lumière, si le rayon

d'illumination est intercepté par un autre objet. Cette fonction est donc directement dépendante du nombre de formes dans la scène et du nombre de sources de lumière.
Pour ce test, il y a 9 sources de lumières et 16 formes. Il y a donc 9*16 = 144 calculs d'intersection à chaque appel à la fonction d'éclairage, ce qui est énorme pour cette scène.

Il reste une autre fonction principale : *interPlusProche()* qui consiste à déterminer la forme la plus proche de l'origine d'un rayon (9.78% du total).
Si on l'enlève du calcul du rayon réfléchis (1.65%) et du calcul du rayon réfracté (0.50%), alors en plus de la fonction d'illumination, le programme passe 9.78 – (1.65 + 0.50) = 7.63% du temps à calculer des intersections, ce qui en fait un total de 77,71 + 7.63 = **85.34%**.

Des solutions seraient d'optimiser les fonctions d'intersection elles-mêmes pour qu'elles soient plus rapides (comme éviter de normaliser un vecteur à chaque fois par exemple), et d'utiliser des structures de partitionnement de l'espace, pour ne pas à avoir à calculer les intersections de toutes les formes à chaque fois.

On remarque également que le lancer de rayons consiste à calculer la valeur de chaque pixel, indépendamment les uns des autres, ainsi l'application se prête bien au parallélisme. Utiliser plusieurs threads pour effectuer le calcul de l'image est très efficace, et c'est là qu'utiliser le GPU plutôt que le CPU a également un intérêt.

Divers

Lien du Git utilisé :
https://github.com/SaladTomatOignon/RayTracing

Problèmes connus :
- L'inclinaison de haut en bas de la caméra pendant le rendu interactif n'est pas 100% opérationnelle, la caméra tourne autour de l'axe « horizontal » qui est calculé en faisant le produit vectoriel du vecteur d'orientation de la caméra avec l'axe « absolu » (0, 1, 0) qui représente l'axe vertical. Du coup lorsque l'orientation de la caméra est colinéaire avec l'axe (0, 1, 0) le produit vectoriel vaut 0 et il n'est plus possible de tourner de « haut en bas ».
La solution serait de fixer dès le début un vecteur vertical pour la caméra, et de le mettre à jour à chaque réorientation, plutôt que de se baser à chaque fois à l'axe « absolu » (0, 1, 0).

- En réalité une forme de plus est disponible : le cône. Cependant le calcul du point d'intersection et de la normale sont parfois incorrectes. Si vous souhaitez tout de même en ajouter un dans la scène, il faut l'ajouter dans le fichier de configuration avec type : « cone » ; centre : Tableau à 3 décimaux ; hauteur : Décimal ; rayon : Décimal.

- Le cylindre a également des défauts, il est préférable de ne pas l'utiliser.

- Le calcul d'intersection du rectangle ne s'effectue que dans un cas particulier : lorsqu'il est parallèle à l'axe X, il manque les rotations pour aller vers la forme canonique. En attendant on peut toujours utiliser la rotation avec la clé « rotation ». Qui permet au final de positionner le rectangle comme on veut.