LifProjet Automne 2020
Thème : Ray Tracing (EG1)

Encadrant : Mr Eric Galin

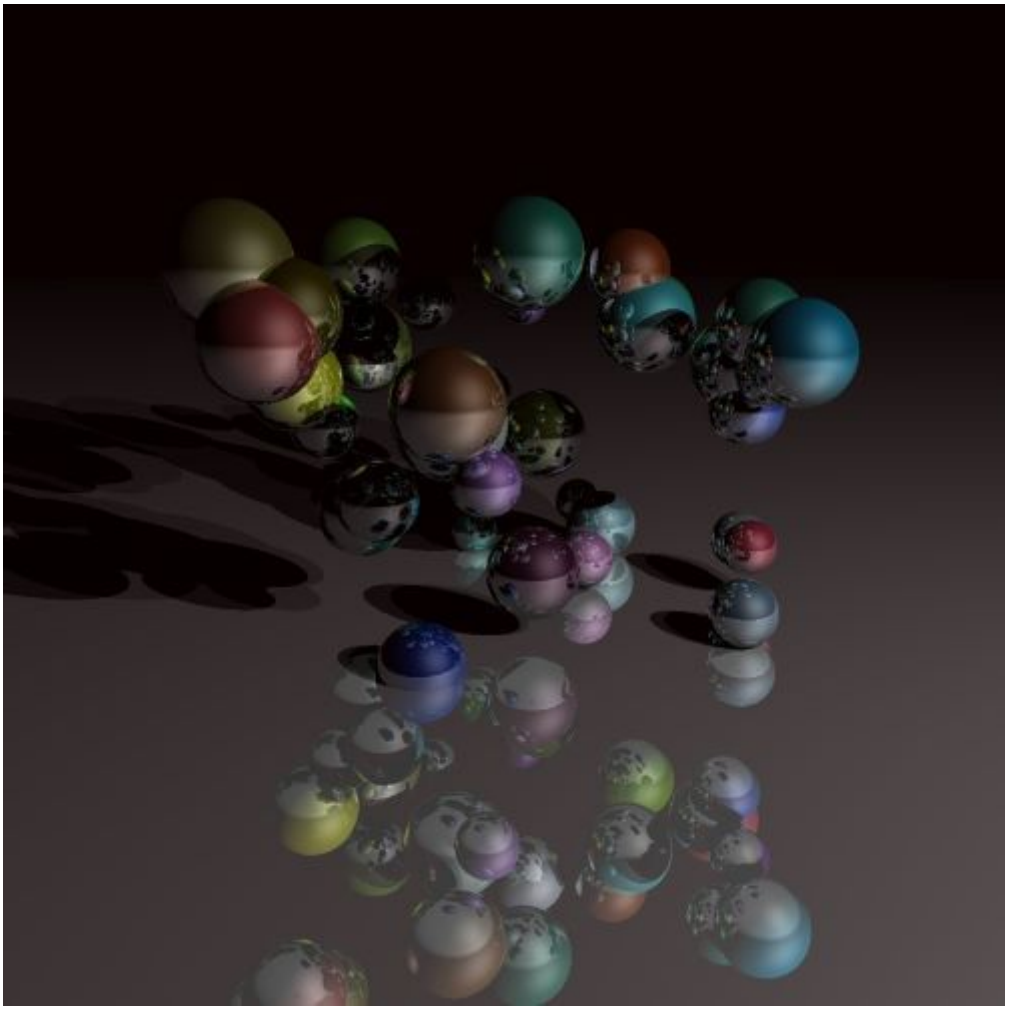


Image d'une scène de notre application où l'on peut distinguer des sphères de différentes couleurs avec réflections sur les sphères et le sol.

Groupe :

Constantin Magnin p1806593
Alban Saint-Sorny p1804792
Como Adrien p1709079

Termes utilisés :

Ray Tracing : lancer de rayon.
Illumination globale : Prendre en compte non seulement la lumière émise directement par une source lumineuse mais aussi celle ayant subi des reflections sur d'autres surfaces.
Méthode de Monte-Carlo : famille de méthodes algorithmiques visant à calculer une valeur numérique approchée en utilisant des procédés aléatoires.
Dispersion lumineuse : Certaines longueurs d'ondes de la lumière ne se propagent pas à la même vitesse dans les milieux dit "dispersifs". Donne lieu à la réfraction lumineuse.

Framework : Désigne un ensemble composants logiciels, qui sert à créer les fondations ainsi que les grandes lignes de tout ou d'une partie d'un logiciel.
Vertices : Sommet d'un angle (intersection de deux arêtes).
DepthMap : Image correspondant concernant la distance des objets au point de vue de la caméra.
Voici une DepthMap d'un objet complexe et d'un objet simple. Les parties noir corresponde aux parties proches de la caméra et claires à celles lointaines.

Pourquoi du Ray-Tracing :

La rasterization est une méthode qui consiste à convertir une image vectorielle (image 3D) en une image matricielle (image 2D) pouvant être affichée sur un écran. Elle nécessite de garder en mémoire toutes les coordonnées des vertices de la scène. C'est une méthode graphique efficace et rapide mais ne reflète pas exactement la réalité. Elle nécessite aussi d'utiliser d'autres algorithmes pour obtenir de meilleurs résultats notamment au niveau des ombres, des couleurs et des réflections.

Le ray tracing permet d'obtenir des résultats en général plus satisfaisants. De plus, les algorithmes de ray tracing peuvent s'occuper de l'illumination globale, de la transparence des objets, des réflections des réfractions et des ombres sans avoir d'autres algorithmes pour l'assister dans la tâche. Le même algorithme avec très peu de modifications peut aussi servir à simuler le son dans une scène en remplaçant le "lancer de rayons" par du "lancer virtuel de particules de sons" et de suivre leurs évolutions dans la scène.

Actuellement, le nombre de jeux, de films générés a partir de ray-tracing ou d'applications et logiciels qui supportent le raytracing ne fait qu'augmenter jour après jour. Longtemps délaissé car jugé trop gourmands en ressources pour l'exécution en temps réel, le hardware permettant d'exécuter des algorithmes du ray tracing en temps réel est de plus en plus accessible au grand public ce qui provoque beaucoup d'engouement pour cette technologie et ne fera que prendre de l'ampleur. Il est possible que la plupart des jeux remplacent peu à peu les techniques de rasterization avec le ray tracing. En effet, certains fabricants de carte graphique (GPU) en font leurs fers de lance et mise énormément sur cette technologie. Il est utile de noter que l'illumination est le facteur qui contribue le plus pour déterminer si une image est réaliste et reflète la réalité ou pas.

Introduction au projet :

Le *ray tracing* fut présenté pour la première fois par Arthur Appel en 1968. Le *ray tracing* ou lancer de rayons est une technique de calcul d'optique par ordinateur, utilisée pour le rendu en synthèse d'image. Très utilisée dans les jeux vidéos et les films par exemple, elle consiste à simuler le parcours inverse de la lumière. On calcule les éclairages de la caméra vers les objets puis vers les sources de lumières, alors que dans la réalité la lumière va de la source, traverse la scène puis rejoint l'œil. Contrairement à d'autres algorithmes (Rasterization) de synthèse d'image, elle permet de définir mathématiquement les objets à représenter et non pas seulement par une multitude de facettes (Vertices) ce qui permet un gain significatif de place en mémoire.

Notre projet à donc pour but de simuler de façon la plus exacte possible l'illumination de scènes prédéfinies en prenant en compte la réflection et réfractions de la lumière d'une scène composé d'objets simples et complexes. En revanche, le ray-tracing ne peut simuler certains aspects de la lumière comme la dispersion lumineuse sans faire appel à des techniques probabilistes comme la méthode de type Monte-Carlo que nous avons implémentée.

Organisation :

Ce projet est entièrement fait en C++ et utilise QT comme framework graphique.
Le projet est composé d'un dossier "src" contenant le code du projet (.h et .cpp), d'un dossier "doc", contenant la documentation Doxygen du projet et d'un diagramme de classe.. Nous disposons aussi d'un fichier CMakeList pour l'exécution du programme et d'un readme.md pour une présentation rapide du projet. Un dossier "data" est prévu lors de l'enregistrement des images des scènes où l'algorithme de *ray tracing* à été appliqué.

Une scène sera en générale composé d'un sol blanc et de murs de couleurs. Les murs et sols sont des objets de type "plan". Au centre il y aura d'autres objets simples et/ou complexes entremêlés les uns aux autres.

Timeline :

Tout au début, nous avons implémenté les classes Vector et Matrix. Puis nous avons choisi de faire une implémentation rapide du Ray Tracing, de la classe Renderer et du sceneManager. Nous nous sommes ensuite concentrés sur une implémentation un peu plus poussée du ray tracing (elle reste en revanche peu réaliste mais nous avions fait attention à ce que toutes les bases soient correctement coder et de ne pas se lancer sur quelque chose qui ne marche pas). Le multi-threading a ensuite été intégré à l'application afin d'améliorer les performances. Tout de suite après, nous avons commencé le développement de la fenêtre QT par précaution car personne n'avait utilisé QT et nous avions peur que cela prenne beaucoup de temps. Le développement de la fenêtre à été plus simple et plus rapide que prévu. Une meilleure implémentation des classes Vector et Matrix a été réalisée. Les réflexions et réfractions sont désormais présentes, la classe Material est donc désormais complète. Nous avons ensuite implémenté les lumières. Les formes complexes ont ensuite été ajoutées grâce à l'intersection, la soustraction et l'union de formes simples. La vue en temps réelle à été ajoutée juste avant le link entre la fenêtre QT (le front-end) et la back-end de l'application.

Documentation :

Notre algorithme de *ray tracing* peut être appliqué à une scène composée à la fois d'objet simple et plus complexe composé eux mêmes d'objets simples.

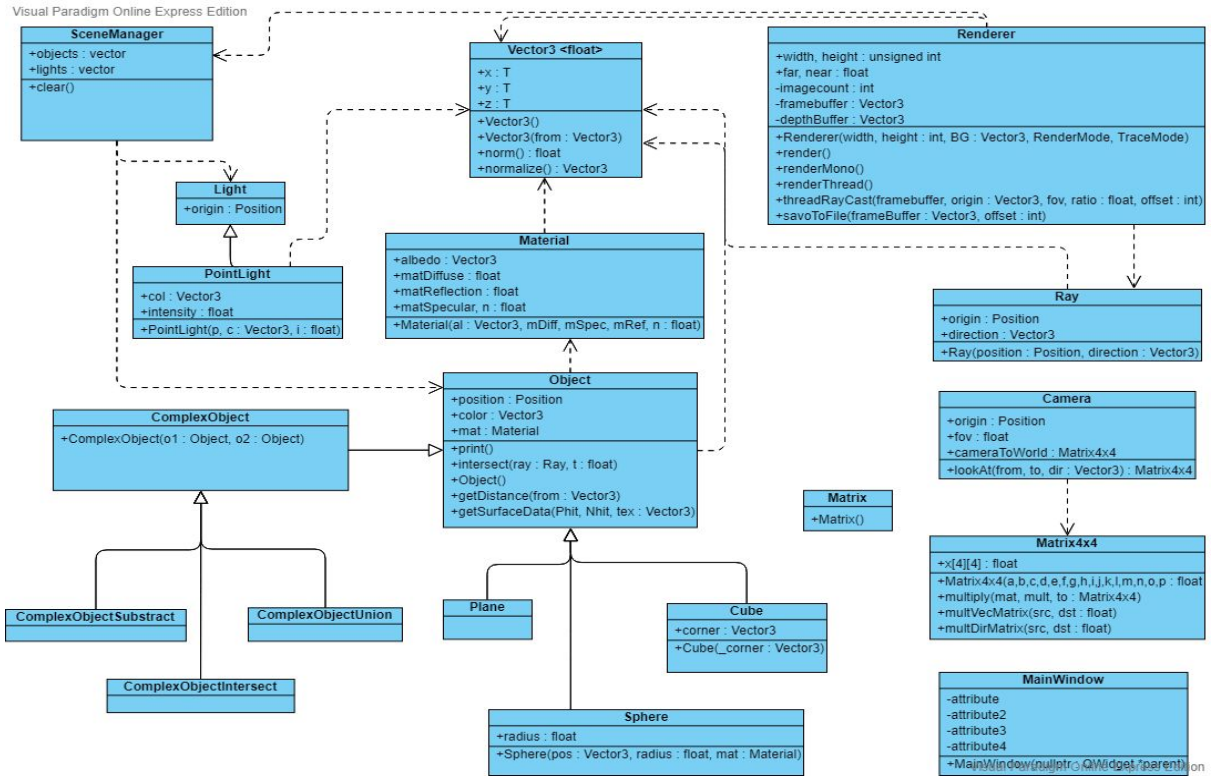
Dans un premier temps les classes “sphère” et “cube” et “plane” sont présentes pour pouvoir afficher et utiliser des objets de type simple dans les scènes.

La super classe “complexObject” est mère des classes “complexObjectSubtract”, “complexObjectUnion”, “complexObjectIntersect” et permet de composer un objet complexe à partir d'objets plus simples en utilisant leurs unions, intersections et leurs soustractions. La classe “Camera” est utile pour connaître la localisation de l'utilisateur, dans quelle direction il regarde et indique le point d'origine des rayons pour la classe “Ray”.

La classe “material” est appliquée à chaque objet et est utile pour connaître le taux de reflections, de réfractions, de transparence et la couleur de chaque objet.

Des templates Matrix et Vector pour l'algorithme de Ray-Tracing sont utilisés pour les calculs. Nous avons décidé d'utiliser nos propres templates au lieu des templates Vector proposés par QT pour être plus indépendant du framework que nous utilisons si un jour nous souhaitons changer de framework. Une architecture MVC a donc été respectée.

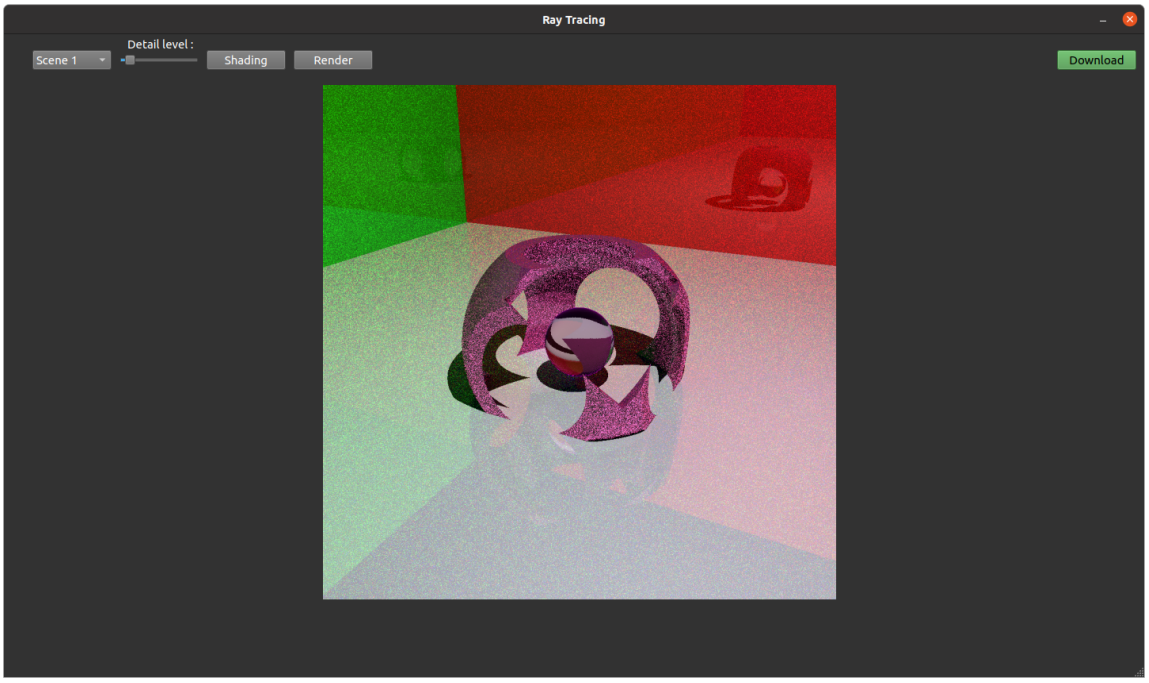
Les classes “Renderer” et “SceneManager” sont utiles pour l'exécution de l'algorithme en plus de la classe “mainWindow”, la classe de QT.



L'ensemble des classes a été documenté grâce à doxygen. Nous fournissons un diagramme des classes afin de mieux se repérer en raison du grand nombre de classes et templates et de connaître les liens entre chacune des classes.

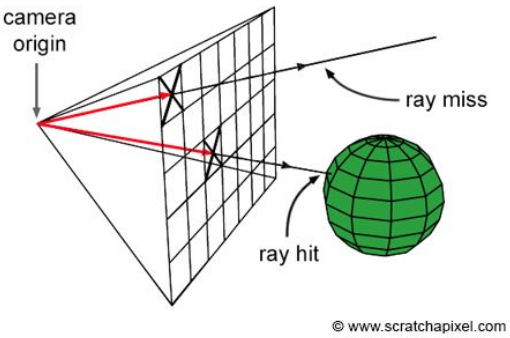
Fonctionnement de l'application:

Nous avons choisi de programmer en C++ pour des raisons de performances et d'utilisation de la mémoire. De plus, nous avons choisi d'utiliser le framework graphique QT car il est nativement en C++ et surtout parce que QT permet de créer de belles fenêtres assez simplement. Il permet aussi de créer une application cross-platform. QT existe aussi depuis plus de 20 ans et est donc une application mature et fiable qui dispose d'une grande communauté et de nombreux exemples sur internet. La gestion des événements (un appui sur un bouton par exemple, ou la gestion des sliders) est très simple et intuitive grâce au "Signaux et Slots". QT creator, l'IDE recommandé pour coder en QT est simple d'utilisation et regroupe tout ce dont nous avons besoin que ce soit au niveau logique (back-end) de l'application et front-end de l'application (gestion fenêtre, boutons). L'IDE permet de coder à la fois en C++ mais aussi en QML qui est fortement recommandé pour placer les objets sur la fenêtre. Le QML est aussi un langage ressemblant très fortement au JSON ce qui ne perturbera pas trop les développeurs, le JSON étant déjà très répandu et simple à apprendre.

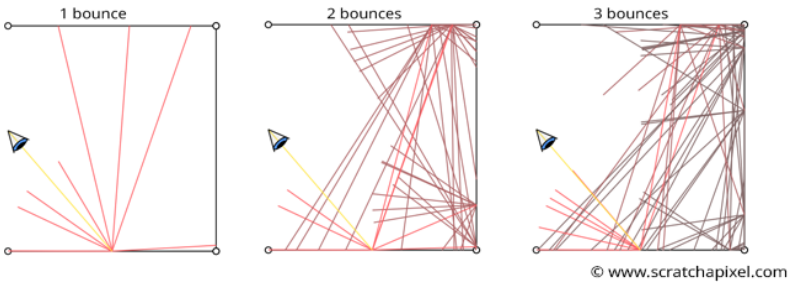


La fenêtre QT ici présentée montre la génération d'un objet simple (la sphère) en plus d'un objet complexe). Il est possible de sélectionner à l'aide du menu déroulant les scènes prédéfinies. Il faut ensuite appuyer sur le bouton "Render" pour générer la scène. On peut aussi ajuster avant la génération le niveau de détail de la scène (nombre de rayons par pixels par exemple) à l'aide du slider "Detail level". Une valeur à 0 exécutera l'algorithme mais ne permettra pas d'obtenir l'illumination globale de la scène car le nombre de rebonds sera à 0. Le niveau de détail serait donc plus faible mais le rendu obtenu plus rapidement. La valeur maximale si le slider est tout à droite permet d'obtenir 2 puissance 14, soit 16 384 rebonds. Cela donnera un bien meilleur niveau de détails mais prendra beaucoup plus de temps. En ce qui concerne les reflets, nous avons choisi d'aller jusqu'à 3 niveaux de profondeur. La fenêtre QT donne aussi la possibilité d'enregistrer des images grâce au bouton "Download" vert d'une scène dans le dossier "data". Le bouton "Shading" permet d'afficher la depthmap de la scène.

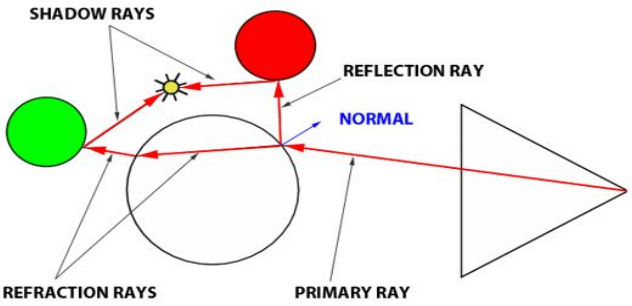
Vue d'ensemble sur le fonctionnement du Ray-Tracing dans notre application:



Rapidement, comme nous pouvons l'apercevoir dans cette image, un rayon de lumière est lancé de la caméra vers une surface composée de pixels. Nous prolongeons alors ce rayon de lumière dans la scène qui finira par croiser ou non un un objet. Si une intersection est détectée, alors on enregistre la collision et on détermine dès maintenant la couleur du pixel. Si aucune intersection n'a lieu, et que le rayon se propage à l'infini, alors le rayon de lumière est supprimé, et le pixel résultant sera éteint. On "oublie" donc les rayons qui s'éloignent à l'infini et sorte de la scène. Projeter des rayons que depuis le point de vue de la caméra (l'utilisateur) à l'avantage de nécessiter le calcul que des rayons qui seront utiles au rendu final.



Si l'on veut aller plus loin et que la lumière dans la scène soit plus réaliste on peut simuler des rebonds de lumière entre les objets. Ceci est le principe de l'illumination globale. Dans notre application nous créeront X nouveaux rayons émanant du point d'origine de la collision entre un rayon et un objet. (X étant la valeur choisie à l'aide du slider). Plus le nombre de rebonds est élevé, plus le niveau de détails de la scène est important. Les performances en pâtissent.



La direction des nouveaux rayons est aléatoire mais demeure restreinte à la demi sphère par rapport à la normale du rayon d'origine et du point d'intersection avec l'objet.

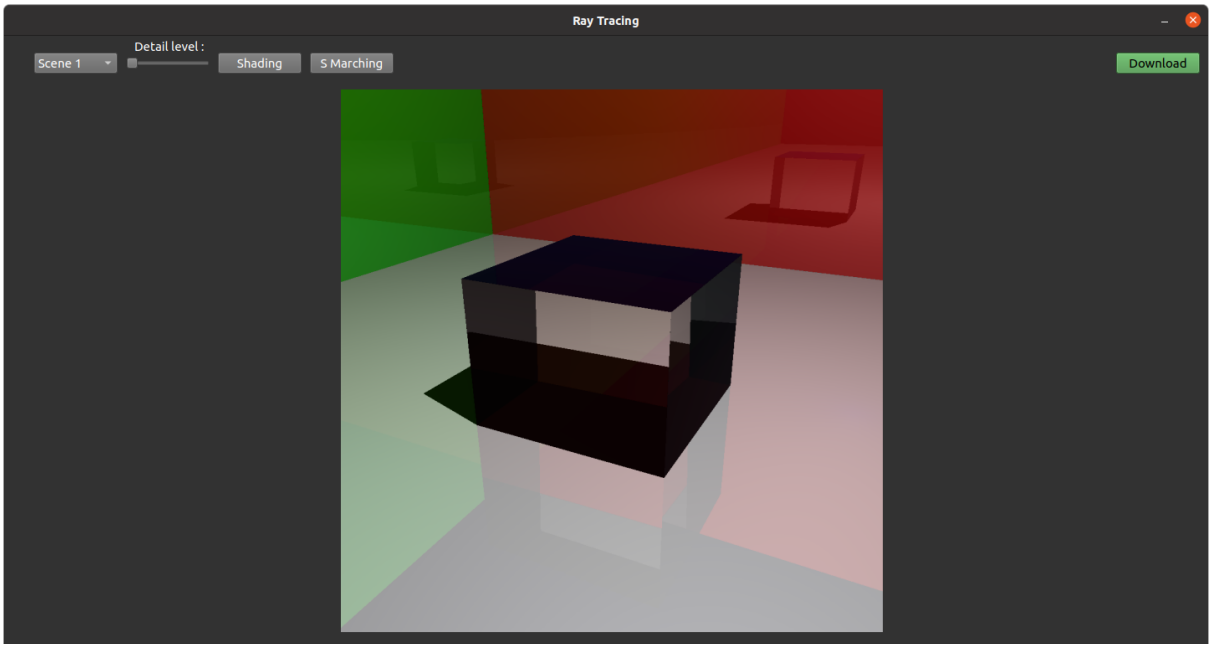
Statistiques sur les performances de l'application:

Le temps nécessaire pour générer une scène dépend de plusieurs facteurs :

- Le niveau de détails(le nombre de rebonds maximal autorisé et le nombre de nouveau rayon généré à chaque reflet).
- Le nombre d'objets présents dans la scène.
- La taille de l'image (le nombre de pixels hauteur*largeur).

Plus ces trois derniers sont élevés, plus la scène mettra du temps à être générée. Une scène très simple peut mettre une minute à être générée. Une scène bien plus évoluée peut mettre entre une demi-heure et plusieurs heures à être générée.

Une formule simple pour calculer le nombre de rayon dans une scène est : *hauteur * largeur * 2 * précision choisie par l'utilisateur*. En général, un rendu dépasse les dizaines de millions de rayons.



Exemple d'un cube transparent avec réflections, sans GI.

Problème rencontré :

Le temps nécessaire pour générer une seule image ralentit le développement et le débogage de l'application.

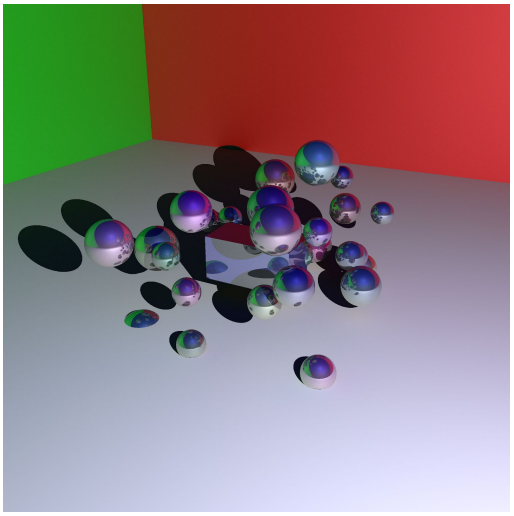
Bilan :

Avec ce projet, nous avons appris les fondements du ray tracing. Étant un projet assez consistant, cela nous a aussi permis d'avoir une première expérience proche d'un projet professionnel. Nous avons réussi à réaliser les objectifs que nous nous étions fixés. Nous sommes satisfait du résultat.

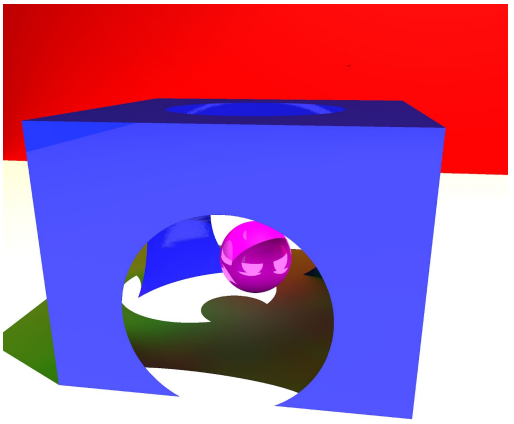
Afin de générer plus rapidement une scène, plusieurs techniques existent. Des composants plus performants. Utiliser principalement le GPU. Il est aussi possible d'approcher les objets de la scène avec la technique des *"bounding volume hierarchy"(BVH)* qui regroupe les objets par exemple proches l'un de l'autre en un seul objet afin de limiter le nombre de calculs et de recherche de collisions entre les rayons et la scène.

Annexe : Exemple d'images

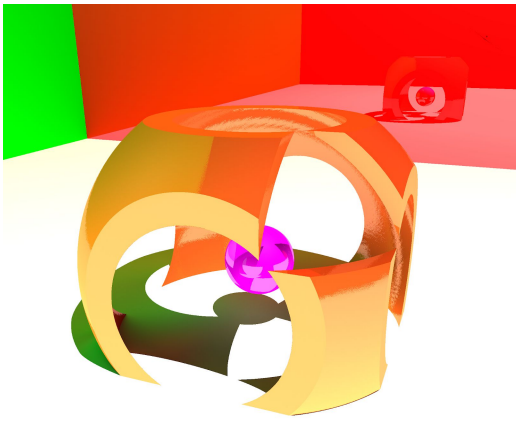
7h de rendu



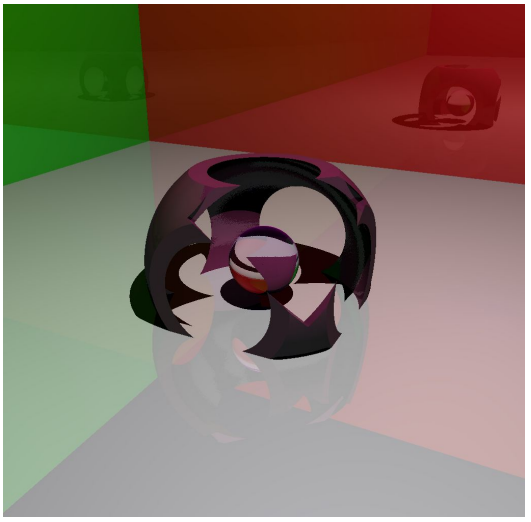
2h40 de rendu



8h de rendu



7h de rendu



4h de rendu