IN 104 – Rendu d'images 3D par lancer de rayons

https://github.com/xoolive/raytracer

L'objectif de ce projet est de concevoir et développer un projet logiciel complet tout en découvrant plusieurs méthodes de travail couramment utilisées dans le monde industriel. Le logiciel demandé sera capable de synthétiser des images par lancer de rayon.
Le langage de programmation à utiliser est le langage C.

Attention ! Ce sujet est amené à être corrigé régulièrement. Il est recommandé de le mettre souvent à jour depuis le site.

1 Description du logiciel – modèle physique

1.1 Principe général

Le logiciel à écrire est un logiciel de synthèse d'images. Il prendra en entrée un modèle d'une scène qui est décrite par un ensemble de facettes triangulaires. En plus de ses propriétés géométriques, chaque facette a des propriétés visuelles ; nous nous limiterons à sa couleur et à sa réflectivité. En sortie, nous voulons produire l'image que pourrait voir un observateur placé à un certain point et qui regarderait la scène modélisée dans un champ de vision (une certaine ouverture) avec un éclairage particulier, que nous limiterons à une seule source (ponctuelle).

Pour déterminer l'ensemble des pixels (points) qui constituent l'image, on place sur le trajet des rayons un écran imaginaire délimité par l'ouverture sur un plan perpendiculaire à la direction d'observation. Cette surface est ensuite divisée en un nombre de lignes et de colonnes correspondant à la résolution recherchée ; chaque case ainsi formée étant la localisation d'un pixel. Pour les premiers tests, une résolution faible, de l'ordre de 50×50 pixels, est largement suffisante.

L'algorithme consiste à lancer un certain nombre de rayons depuis le point d'observation en direction du modèle (figure 1). Pour chaque pixel de l'image, on détermine sa couleur à l'aide d'un rayon. Le rayon est lancé depuis le point d'observation et passe par le pixel, puis va intersecter ou non une facette du modèle.

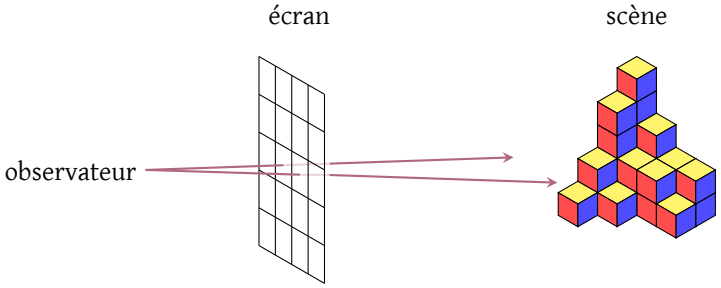

Figure 1 – Principe général de l'algorithme

1.2 Calcul de la couleur résultante

La couleur résultante est déterminée au niveau du point d'intersection entre le rayon et le modèle en fonction de divers paramètres :

- la couleur propre de la facette, qui fait partie de la description du modèle ;
- le degré d'éclairement du point d'intersection par la source lumineuse et la couleur de cette source ;
- le coefficient de réflexion sur la facette intersectée, qui s'il est maximum (égal à 1), en fait un miroir. Avec un coefficient de réflexion non nul, la couleur de la lumière qui se réflechit au point d'intersection influence donc la couleur observée ;
- une couleur de fond pour les rayons qui n'intersectent pas le modèle.

Les couleurs sont décrites suivant le modèle RVB (RGB en anglais) : chaque couleur est décomposée selon trois composantes (rouge, vert, bleu) et chaque composante est discrétisée sur une échelle de 0 (absence de cette couleur dans le mélange) à 255 (présence maximale de cette couleur dans le mélange). Ainsi, la couleur codée par (255,0,0) est le rouge le plus intense, (200,0,0) est un rouge pur mais moins intense, (200,100,100) est un mélange de rouge à 50 %, et de vert et de bleu à 25 % (le résultat est entre orange et brique). Toutes les formules relatives aux couleurs marchent par 3, une sous-formule par composante.

La formule qui donne la couleur finale d'un pixel de l'image résultat est la suivante :

$$C = k \cdot C^R + \frac{E \cdot (1-k)}{255} \left(\overrightarrow{N} \cdot \overrightarrow{S}\right) \cdot C^S \cdot C^F \tag{1}$$

Dans cette formule :

- k représente le coefficient de réflexion (entre 0 et 1) de la facette intersectée ;
- C^R est la couleur qui se réfléchit au point d'intersection ;
- $\overrightarrow{N}$ est la normale à la facette intersectée au point d'intersection ;
- $\overrightarrow{S}$ est le vecteur de direction du point d'intersection vers la source lumineuse ;
- C^S est la couleur de la source lumineuse ;
- C^F est la couleur de la facette intersectée.
- E vaut 0 ou 1 selon que le point d'intersection est ou non éclairé par la source. Il vaut 1 si la source est du bon côté de la facette intersectée (ce qui se traduit par un produit scalaire positif et s'il n'y a pas d'obstacle entre le point intersecté et la source lumineuse. Pour trancher ce dernier point, il faut savoir si une facette du modèle est dans la direction de la source et plus proche que cette-ci du point d'intersection ; on peut le déterminer en lançant un rayon du point d'intersection vers la source lumineuse.

1.3 Calcul de la couleur réfléchie

La couleur réfléchie C^R présente dans la formule (1) découle des lois de Snell-Descartes : le rayon réfléchi est dans le plan d'incidence et les deux angles sont identiques

$$\left(-\overrightarrow{I}, \overrightarrow{N}\right) = \left(\overrightarrow{N}, \overrightarrow{R}\right) \tag{2}$$

La figure 2 illustre ce phénomène.

À son tour, la couleur réfléchie se détermine en lançant un nouveau rayon, dit *secondaire*, qui peut à son tour se réfléchir. De manière à éviter tout problème de réflexion infinie, il conviendra de limiter le nombre de réflexions autorisées à partir d'un rayon initial.

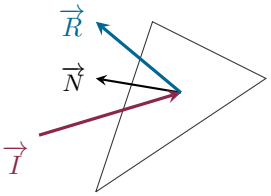

Figure 2 – Loi de la réflexion de Snell-Descartes

1.4 Format et orientation de l'écran

On définit un repère caméra par un point observateur O, un point au centre de l'écran A et un point B au milieu à droite sur l'écran, tel que $\overrightarrow{AO} \perp \overrightarrow{AB}$ [1].

Le point C permettant de définir un repère sur l'écran est alors défini tel que $\left(\overrightarrow{AB}, \overrightarrow{AC}, \overrightarrow{AO}\right)$ forme un repère orthogonal, avec $\|\overrightarrow{AB}\| = \|\overrightarrow{AC}\|$.

On découpe alors l'écran ainsi défini en une grille en fonction de la résolution voulue.

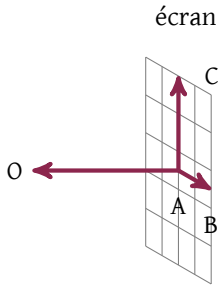

Figure 3 – Description du repère écran

1.5 Format du fichier d'entrée (.scn)

Une scène est définie par un « repère caméra », une source de lumière, une couleur de fond est un ensemble de facettes décorées. Le format des fichiers textes que votre sous-programme de lecture devra pouvoir lire est le suivant :

- toutes les lignes qui commencent par le caractère dièse # sont des lignes de commentaires, à ignorer ;
- une première ligne contient la position de l'observateur x, y, z ;
- la deuxième ligne représente l'écran : $x_A, y_A, z_A, x_B, y_B, z_B, w, h$; dans l'ordre les coordonnées du point au centre de l'écran, les coordonnées du point à droite de l'écran, et la taille de l'image en sortie (largeur, hauteur).
- une troisième ligne contient le nombre de sommets de la scène suivie des coordonnées de chacun de ses points x, y, z, chacun sur une ligne différente.

1. Dans votre programme, il conviendra de vérifier que le point B fourni vérifie $\overrightarrow{AO} \perp \overrightarrow{AB}$ et, si nécessaire, de le projeter orthogonalement sur le plan $\left(A, \overrightarrow{AO}\right)$

- à la suite, une nouvelle ligne contient le nombre de facettes, puis pour chaque facette écrite sur une ligne différente les indices des sommets concernés (en commençant par 1), le coefficient k de réflexion de la facette, et les trois composantes RGB de sa couleur propre : a, b, c, k, R, G, B.
- les deux dernières lignes contiennent respectivement les trois composantes de la couleur de fond R, G, B, puis les caractéristiques de la source de lumière x, y, z, R, G, B.

Le fichier `scenes/cubes.scn` est un exemple de fichier que votre programme doit savoir lire.

Afin de vous aider à lire le fichier, la librairie `libutil` contient une fonction `splitLine` qui découpe une chaîne de caractères en un tableau de mots, le découpage se faisant au niveau des espaces et des virgules.

1.6 Format du fichier de sortie (PPM)

Le programme doit écrire l'image rendue au format PPM (P3). Il vous appartient de rechercher ce qu'est un fichier PPM valide, d'en écrire un et de l'afficher sur votre poste de travail avec le logiciel de votre choix (Sous Linux, le logiciel `xv` est capable de lire ce type de fichiers)

Sachez qu'avant d'en arriver à l'écriture de ce fichier, vous pourrez faire des tests de votre programme grâce à la librairie `libimgsdl` fournie. [2]

2 Méthode de travail

2.1 Développement itératif

Il est quasiment impossible de réussir à coder un programme complexe du premier coup. Dans les cours d'initiation au langage C, on vous proposait des étapes intermédiaires simples pour vous permettre de coder des programmes plus complexes. Aujourd'hui, il vous appartient de définir vous-mêmes les étapes intermédiaires : à partir d'une situation complexe, commencez par identifier des cas particuliers simples (angles tous perpendiculaires, fichiers d'entrée à une seule facette).

Une fois que votre programme compile et fonctionne correctement, intéressez-vous à la généralisation. Ainsi, l'écart entre l'état courant de votre code et une version stable est toujours faible. Il est toujours plus facile de chercher les erreurs dans un programme proche d'une version stable que dans un programme de 5000 lignes qui n'a jamais été compilé.

2.2 Tests unitaires

Lors du développement et de la maintenance d'un logiciel, il est préférable d'écrire un jeu de tests unitaires. Il s'agit de petits programmes indépendants [3] qui testent une par une les fonctions implémentées dans le programme principal. En général pour chaque fonction, on teste la correction des résultats de la fonction pour des cas simples, des cas plus complexes et des cas aux limites du domaine.

Le programme de tests unitaires compare pour chaque test une valeur obtenue à une valeur attendue et affiche un résumé à la fin de son exécution. L'intérêt de tels programmes est d'assurer

2. Des élèves en difficulté pourront faire des copies d'écran des rendus `libimgsdl` pour leurs compte-rendus.
3. On entend par programme indépendant le fait qu'il contient sa propre fonction `int main()`, tout en faisant appel aux fonctions utilisées par le programme principal. Il convient donc de faire attention lors de la compilation à ce qu'une seule fonction `int main()` soit présente dans chaque exécutable.

qu'après chaque modification, on ne régresse pas dans les fonctionnalités fournies. Dans le monde du logiciel, on essaie de s'interdire de partager son code source avec les autres développeurs si celui-ci fait échouer des tests unitaires.

2.3 `Makefile`

Si un programme d'un seul fichier est en général facile à compiler (la commande `gcc fichier.c` suffit), les gros projets logiciels sont plus complexes à gérer. On compile souvent séparément chaque fichier, en faisant appel à des fichiers d'en-tête situés dans différents répertoires, en activant des options bien précises, et en procédant à une édition de lien avec des librairies statiques et dynamiques à appeler dans un ordre précis. Attendre que tout les développeurs et utilisateurs sachent compiler toutes les parties du projet tient de la gageure.

Afin de pallier ce problème, on utilise fréquemment l'outil `make` qui lit un fichier nommé `Makefile`. Ce fichier décrit comment compiler des sous-parties d'un projet (appelées *cibles*) à partir d'un ensemble de règles de dépendances. Pour chaque cible, le fichier donne une liste de dépendances et une ligne de compilation. En appelant `make <cible>` depuis un terminal, l'outil `make` vérifie si les dépendances ont besoin d'être recompilées et si besoin, réexécute la commande de compilation donnée.

Nous utiliserons dans ce projet l'outil `make` : le premier `Makefile` vous permettant de terminer la partie 1 vous est fourni. En revanche, vous devrez l'éditer pour les parties 2 et 3. Lors de la livraison de votre code source, vous fournirez un fichier `Makefile` : la commande `make all` devra permettre de compiler votre projet (il vous appartient bien sûr de vérifier que la commande fonctionne chez vous avant de soumettre votre projet) ; la commande `make run` devra lancer votre logiciel et écrire sur le terminal un message suffisamment explicite pour décrire ce que le logiciel devrait faire. Enfin `make tests`devra compiler et exécuter tous vos tests unitaires au même titre que le test fourni.

2.4 Règles et bonnes pratiques (2 points)

Pendant ce module, vous serez évalués à la fois sur vos compétences en conception de projet, en programmation et sur votre capacité à mettre en pratique un certain nombre de bonnes pratiques.

Pour la partie programmation, il vous est demandé de suivre les bonnes pratiques suivantes :
- les variables/fonctions commencent par une minuscule ; les instructions du préprocesseur sont toutes en majuscules ;
- dans chaque fichier d'en-tête (en `.h`), utilisez les instructions `#ifndef`, `#define` et `#endif` ;
- lorsque vous faites un test d'égalité, optez pour un test avec la constante à gauche : préférez `if (0 == x) { }` à `if (x == 0) { }` : cette habitude permet de faire échouer la compilation si vous mettez un seul signe = au lieu de deux ;
- compilez systématiquement avec l'option `-Wall`, et enlevez tous les avertissements ;
- faites des tests unitaires pour toutes vos fonctions et relancez régulièrement vos tests unitaires pour vérifier que l'état de votre code n'a pas régressé.

3 Objectifs

La page `https://github.com/xoolive/raytracer` décrit le contenu du dossier fourni, la manière de le télécharger et, uniquement si vous le souhaitez, d'interagir avec git. Mais pour le moment, concentrez-vous sur la partie 3.1.

Règles et bonnes pratiques		2 points
Code indenté, commenté ; bonnes pratiques		1 point
Respect d'un développement itératif	⋆	1 point
Tests unitaires		3 points
Note rendue par le programme de test fourni		2 points
Écriture de ses propres tests	⋆	1 point
Livraison de la fourniture		4 points
Code qui ne compile pas (tests ou exécutable)		-2 points
Lancer de rayon fonctionnel	⋆	2 points
Lecture de fichier d'entrée		1 point
Production de fichier de sortie		1 point
Étude d'une problématique choisie	⋆	4 points
Définition du problème étudié	⋆	1 point
Analyse du problème	⋆	3 points
Bonus difficulté	⋆	+1 point
Analyse de la pratique	⋆	4 points
Soutenance		4 points
Fond et forme seront notés à pondération égale		
Pénalité de retard		10 %

Table 1 – Barème de notation. Les items marqués du symbole ⋆ doivent faire l'objet d'une analyse dans le rapport. Ils seront évalués au moins partiellement, sinon entièrement, au regard de celui-ci.

La table 1 décrit le barème de notation du module. Notez bien que la qualité du rapport influence la note de plusieurs parties qui évaluent a priori votre code.

Comme dans l'industrie, tout retard fera l'objet de pénalités sur la note finale. Tout retard fera l'objet d'une pénalité de 10 % sur la note finale ; des retards à deux des échéances entraîneront une pénalité de 20 %, et ainsi de suite.

Si vous demandez en bonne et due forme un report de délai **au moins trois jours à l'avance**, alors, en fonction des circonstances, la pénalité pourrait ne pas être appliquée.

Les échéances à respecter sont les suivantes :
- livraison de la partie 1 le 22 avril avant 22h ;
- livraison de la partie 2 le 17 mai avant 22h ;
- livraison finale (sous la forme d'une archive zip, tar.gz ou d'un lien vers un repository github) de l'ensemble de votre espace de travail (codes, tests, rapport **au format PDF**) ; le 31 mai avant 22h.

3.1 Démarrons en douceur

Cette première partie du projet est fortement encadrée afin de vous permettre de vous familiariser avec le sujet et l'environnement de travail ; et d'éviter à tous de partir sur une mauvaise piste. Les structures de données et les objectifs sont définis. Une simple commande à exécuter sur votre répertoire de travail saura déterminer si vous avez rempli vos objectifs. Ainsi, les moins à l'aise devraient pouvoir assurer leurs arrières. Attention cependant : une grande rigueur de travail est nécessaire ; le moindre écart à la consigne fera échouer les tests et aura un impact immédiat sur la note finale.

Un squelette minimal de code vous est fourni sur le site. Il contient des noms de structures et des en-têtes de signatures de fonctions, à respecter.

Le fichier `tests/tests_notes.c` contient l'ensemble des tests sur lesquels vous serez évalués. Il fait appel aux deux fichiers d'en-tête `geometry.h` et `intersection.h` : le fichier de geométrie contient les descriptions des structures de données alors que le fichier `intersection.h` contient les en-têtes des fonctions testées : ces fonctions que vous écrirez doivent être capables de calculer l'intersection d'un rayon avec le plan induit par une facette ; et de tester l'appartenance d'un point sur un plan à une facette.

Attention : pour une compilation complète, vous devrez ajouter sur la ligne TESTS_OBJ le nom des fichiers compilés dans lesquels vous aurez codé toutes les fonctions nécessaires au bon fonctionnement du test. Vous pouvez choisir librement le nom de ces fichiers pourvu qu'il y ait une cohérence.

Contenu de la livraison La livraison devra être envoyée avant le dimanche 20 avril à 22h, sous la forme d'une archive zip ou tar.gz. L'archive sera alors décompressée pour évaluation avec la commande `make tests`. L'archive devra contenir les dossiers `include`, `src`, et le fichier `Makefile`. Le fichier de tests fourni ne doit pas être dans l'archive.

Analyse attendue dans le rapport final Vous devrez décrire vos analyse et décomposition du problème, les écueils identifiés et les choix qui en ont découlé. Décrivez les tests unitaires conçus et analysez l'apport qu'ils ont eu dans votre projet.

3.2 Un peu plus de liberté...

Vous avez réussi ? Félicitations ! Pour cette nouvelle étape, vous devez remplir les objectifs présentés au début du sujet et écrire un programme de lancer de rayons complet. Vous êtes libres dans le choix des structures de données et dans l'écriture des tests unitaires (que vous ne devez pas vous dispenser d'écrire pour autant !). Notez que vous devrez éditer le fichier Makefile afin que la commande `make tests` exécute à la fois les tests notés de la première partie et les tests que vous écrivez pour cette seconde partie.

Note sur l'utilisation des librairies `libutil.a` et `libimgsdl.a` Afin de vous faciliter la tâche, deux librairies statiques vous sont fournies. Il s'agit de code déjà compilé et archivé dans une librairie statique (d'où le `.a`). Le contenu des fonctions est décrit dans `util.h` et `imgsdl.h`. `util.h` vous propose des outils que vous n'aurez pas besoin de coder. `imgsdl.h` vous permet d'afficher une image à l'écran sans avoir à coder l'écriture du fichier PPM.

Attention lorsque vous utiliserez `libimgsdl.a` ! Cette archive contient une fonction `main`. Lors de l'édition de lien, vous ne pourrez pas lier à la fois `libimgsdl.a` et un objet qui a lui aussi une fonction `main` à l'intérieur (comme par exemple `tests/tests_notes.c`). Vous trouverez un fichier `tests/tests_imgsdl.c` qui vous donnera un exemple d'utilisation à compiler et exécuter avec la commande `make tests_sdl`.

Contenu de la livraison La livraison devra être envoyée avant le dimanche 18 mai à 22h, sous la forme d'une archive zip, tar.gz, ou si vous êtes à l'aise, d'un repository git [4] L'archive sera alors décompressée pour évaluation avec la commande `make run`. Celle-ci devra, à partir du fichier `scenes/cubes.scn`, soit générer un fichier `cubes.p3` et afficher le chemin vers ce fichier ; soit ouvrir une fenêtre (avec `libimgsdl.a`) pour afficher le rendu. L'archive devra contenir les dossiers `include`, `scenes`, `src`, `tests` et le fichier `Makefile`.

4. Cliquez ici et suivez les consignes de livraison décrites.

Analyse attendue dans le rapport final Vous devrez décrire vos analyse et décomposition du problème, les écueils identifiés et les choix qui en ont découlé. Décrivez les tests unitaires conçus et analyser l'apport qu'ils ont eu dans votre projet.

3.3 Définissez vos objectifs !

Vous pourrez attaquer cette partie quand vous aurez un logiciel fonctionnel. Une liste d'améliorations vous sont suggérées, mais vous pouvez définir les vôtres même si elles ne sont pas dans la liste. Pour les objectifs qui nécessitent un développement logiciel, vous êtes entièrement libres dans l'implémentation mais des explications sur votre approche du problème seront attendues.

Chaque objectif fera l'objet d'une présentation dans le rapport : une grande importance sera attachée à la description du problème. Une fois le problème posé, vous pourrez commencer votre analyse écrite, *puis*, le cas échéant, coder des améliorations [5]. Les résultats seront présentés dans le rapport sous la forme d'une analyse écrite accompagnée d'éléments visuels (schémas, images avant/après, courbes de performance, etc.)

Il sera préférable de se concentrer sur un seul objectif, afin de le traiter en profondeur, plutôt que de papillonner en survolant tous les domaines.

Faites vous plaisir dans le domaine qui vous intéresse le plus !

Quelques suggestions (dans le désordre des niveaux de difficulté) :
- affichage d'une scène exposée à des sources de lumière multiples ;
- transparence, réfraction (Snell-Descartes) ;
- profondeur de champ ;
- comparaison de performances entre la fonction $1/\sqrt{x}$ et `fastInvSqrt` (fournie dans `libutil.a`).
- modélisation des couleurs en trois composantes teinte, saturation, valeur (HSV) plutôt que rouge, vert, bleu (RGB) ;
- parallélisation des calculs de rendu (étude papier) ;
- génération de fichiers d'entrée du logiciel à partir de scènes décrites en utilisant des formes géométriques courantes (sphères, cylindres, ellipsoïdes, théières, etc.) ;
- affichage d'une source de lumière (halo, parhélie, flare, etc.) ;
- étude de la complexité ;
- utilisation d'une structure d'octrees pour optimiser les calculs d'intersection

3.4 Analyse de la pratique

Le rapport final devra contenir une partie d'une ou deux pages qui analyse de façon critique le déroulement du projet ainsi que les facteurs de réussite et d'échecs.

3.5 Soutenance

Le projet se concluera par une soutenance en deux parties : 10 minutes de présentation et 10 minutes de questions. La présentation comportera une démonstration sur la base du code de la livraison finale, une analyse des difficultés recontrées (y compris la problématique de la partie 3.3) et des solutions mises en œuvre. Il ne sera pas nécessaire de présenter ce qu'est le lancer de rayons.

5. Dans ce cas, il vous sera demandé de fournir le code qui aura servi à l'analyse lors de la livraison finale.

A Historique

Le lancer de rayons est la manière historique de gérer l'imagerie 3D. C'est une excellente manière de rendre compte des caractéristiques géométriques de la lumière. (réflexion, réfraction, transparence) Aujourd'hui, d'autres méthodes existent pour gérer la 3D, mais le lancer de rayons reste néanmoins un sujet d'actualité.

L'article fondateur pour le lancé de rayon récursif a été publié en juin 1980 par T. Whitted. Il s'agit de *An improved illumination model for shaded display*, publié dans *Communications of the ACM*, (vol. 23, n°6).

B Lectures recommandées

Pour ceux qui veulent en savoir plus, voici quelques références digne d'intérêt sur le site de Wikipedia (en anglais) :

- l'article sur le lancer de rayons :
  https://en.wikipedia.org/wiki/Ray_tracing_%28graphics%29
- l'article sur l'infographie :
  https://en.wikipedia.org/wiki/Computer_graphics
- l'article sur le langage de programmation Cg (C for graphics), développé par NVIDIA pour faire calculer un rendu d'images par la carte graphique (GPU) au lieu du CPU traditionnel :
  https://en.wikipedia.org/wiki/Cg_%28programming_language%29
- l'article sur le format PPM :
  https://en.wikipedia.org/wiki/Netpbm_format