IFT3355 : Infographie
Travail Pratique 4 : Lancer de rayons

Remise : 8 décembre 2021, 23:59:59

1 Introduction

Dans ce TP vous implémenterez un algorithme "simple" de lancer de rayons (un ou plusieurs rayons par pixel) qui calcule l'illumination locale, les ombres (ombres dures et pénombres douces) et la réflexion miroir.

Vous devriez profiter du temps pour planifier avancer en étapes régulières, bien ordonnées. Notez aussi qu'à quelques occasions, nous soulignons de coder la partie plus compliquée en deux étapes. Par expérience, il est facile de résoudre le problème simple, ce que vous devriez faire dans une première étape. Ensuite, vous comprendrez beaucoup mieux les problèmes soulevés, et vous serez plus aptes à résoudre le problème complet. Jadis, on appelait cette approche en anglais "throw-away code". Le gros avantage, c'est quand arrive la date d'échéance, si la partie complexe de votre code ne fonctionne pas, vous pouvez revenir en arrière et la partie de base fonctionnera encore.

Enfin, vous pouvez nous croire qu'attendre à la fin pour commencer ce TP en particulier ne sera pas agréable...

1.1 Description du squelette de code

L'entrée du programme est définie dans `main.cpp`. Le programme utilise Parser (défini dans `parser.hpp` et `parser.cpp`) pour analyser les fichiers de la scène (dans le répertoire `scenes/`, la description du format se trouve dans `scenes/basic.ray`). Il faut créer un objet Scene (défini dans `scene.hpp`) qui sera utilisé dans les différentes étapes du lancer de rayons.

Ensuite, Raytracer (défini dans `raytracer.hpp` et `raytracer.cpp`) est utilisé pour "rendre" la scène dans un objet Image qui produit un fichier `.bmp` en sortie (via `image.hpp`). La scène peut contenir des objets géométriques tels que des sphères, des plans, des maillages triangulaires ainsi que des coniques (définis dans `object.hpp` et `object.cpp`). Les outils mathématiques de base, tels que les vecteurs, les matrices, les rayons et les intersections sont définis dans `basic.hpp`. (voir section 3.3).

Il y a trois répertoires : `scenes/`, `meshes/` et `referenceResults/`. Le répertoire `scenes/` contient les fichiers de scène au format `.ray`, décrivant les paramètres de la scène suivants : Dimensions, Perspective, LookAt, Material, PushMatrix, PopMatrix, Translate, Rotate, Scale, Sphere, Plane, Mesh et PointLight. Ces fichiers sont commentés afin d'expliquer le format. Quelques maillages triangulaires au format `.obj` sont proposés dans le répertoire `meshes/` et la plupart des scènes dépendent d'un ou plusieurs de ces fichiers. Enfin, dans le répertoire `referenceResults/` se trouvent des résultats de référence afin que vous puissiez comparer vos résultats. On ne s'attend pas à ce que vos résultats soient identiques, surtout quand une partie aléatoire est utilisée dans les algorithmes, mais ils devraient bien entendu se "ressembler".

Votre travail sera principalement restreint aux fichiers `object.cpp` et `raytracer.cpp`. Vous êtes **vivement** encouragés à regarder les fichiers `basic.hpp`, `object.hpp`, `raytracer.hpp` et `scene.hpp` afin de bien comprendre les classes C++, ce qui vous aidera avec votre code. Les autres fichiers (`image.hpp`, `parser.cpp` et `parser.hpp`) sont moins importants, mais éducatifs.

1.2 Instruction pour la compilation et l'exécution

Si vous avez un IDE favori, créez un projet vide et ajoutez tous les headers (`.hpp`) et fichiers sources (`.cpp`) dans le répertoire principal, puis basez-vous sur les commandes de votre IDE. Le projet n'a pas besoin de librairies supplémentaires. Pour les utilisateurs de Windows, nous fournissons également un fichier `.sln` pour Visual Studio. Pour Linux (et donc les machines du département), nous fournissons un Makefile pour compiler le projet. Pour compiler avec celui-ci, utilisez simplement la commande `make` dans le répertoire principal. Il est aussi possible d'utiliser un WSL.

Pour lancer l'exécutable : `./TP2.exe` ou `./raytracer` Par défaut, cela fera le rendu d'une scène avec seulement un cube rouge et sauvegardera l'image de sortie par défaut `output.bmp` dans `scenes/` où il y aura également une image de profondeur par défaut `depth.bmp` pour laquelle le blanc représente le lointain et le noir représente le proche.

Pour lancer l'exécutable et rendre une scène différente, indiquez-la comme premier argument :
`./TP4.exe scenes/basic.ray`
ou
`./raytrace scenes/basic.ray`

L'image résultante sera également sauvegardée dans le répertoire `scenes/` et nommée `your_scene_file_path_output.bmp` (`your_scene_file_path_output_depth.bmp`). Pour de simple scènes, le rendu se fait en quelques secondes. Cependant, dès que vous utilisez des maillages, vous comprendrez pourquoi nous utilisons C++ pour ce TP, mais aussi pourquoi tant de recherche s'est consacrée aux méthodes de subdivisions d'espace.

2 Devoir (Total : 100 pts)

Nous vous recommandons fortement de suivre l'ordre des objectifs et d'implémenter l'algorithme étape par étape. L'algorithme doit lancer des rayons primaires dans la scène, qui génèreront des rayons d'ombre et d'autres rayons secondaires. Notez que le rendu de scènes complexes composées de beaucoup de primitives (la théière fournie a un maillage de milliers de triangles) peut prendre du temps! Ne changez aucun des fichiers de scène, nous comparerons vos résultats avec les résultats sur nos propres scènes de référence pour noter le projet.

2.1 Lancer des rayons (20 pts)

1. Implémentez les parties manquantes de `Raytracer::render` pour lancer des rayons de façon simple pour tous les pixels de l'image, en utilisant la position de la caméra et les coordonnées de chaque pixel. Vous pouvez tester votre code en recalculant le pixel comme l'intersection entre le rayon et le plan de l'image, et vérifier que vous obtenez les bonnes coordonnées.
2. Implémentez les parties manquantes de `Raytracer::trace` pour tester itérativement toutes les intersections des objets avec le rayon donné. Mettez à jour la profondeur comme la profondeur de la première intersection, puis après avoir implémenté une des fonctions de test d'intersection, vous devriez avoir quelques informations de profondeur dans votre image de sortie.

2.2 Tests d'intersection (15 pts)

Dans cette partie, vous implémenterez les fonctions d'intersection puis les fonctions pour lancer les rayons primaires. Le résultat sera l'image de profondeur, montrant les informations de profondeur d'une scène donnée. Nous noterons cette partie en fonction de l'image de profondeur générée par votre programme.

Étapes :

1. Implémentez `Sphere::localIntersect`. Pour cette partie, vous devez calculer si un rayon a intersecté votre sphère. Soyez sûrs de couvrir tous les scénarios d'intersections possibles (zéro, un ou deux points d'intersection). Testez votre résultat en comparant les images de profondeur générées par votre algorithme avec celles fournies comme résultats solution des exemples.
2. Implémentez `Plane::localIntersect`. Cette partie est très similaire à la précédente puisque vous calculez si une ligne a intersecté votre plan. Testez cette fonction de manière similaire à l'étape précédente (notez qu'en faisant ces étapes, des objets apparaîtront dans la scène).
3. Implémentez `Mesh::intersectTriangle`. Cette fonction calcule le point d'intersection d'un rayon avec un triangle. Ce test est différent de celui avec un plan puisque vous devez vérifier que le point d'intersection est à l'intérieur des limites du triangle. Repensez au cours et essayez de trouver quelles équations peuvent vous aider pour décider de quel côté des lignes de limites du triangle le rayon intersecte. Testez cette partie comme les étapes précédentes ; quand votre intersection avec un triangle fonctionne correctement, vous devriez pouvoir voir apparaître des maillages complets dans vos scènes. *Pensez également à la façon de calculer une normale au point d'intersection.*

Vous pouvez tout d'abord implémenter les tests d'intersection pour certains objets géométriques et vous concentrer sur les parties suivantes. Puis, une fois que vous aurez complété l'algorithme, vous pourrez revenir à l'implémentation des autres tests d'intersection.

2.3 Illumination locale (10 pts)

Implémentez la partie manquante de `Raytracer::shade` qui réalise le calcul d'illumination permettant de trouver la couleur à un point. Dans un premier temps, vous pouvez supposer que toutes les sources de lumière sont directement visibles. Vous devrez calculer les composantes ambiante, diffuse et spéculaire du modèle du cours (spéculaire de Blinn). Voyez cette partie comme l'étape déterminant la couleur à un point où le rayon intersecte la scène. Une fois rendue, vous devriez avoir une image colorée avec l'illumination locale. Pensez à chaque composante pour bien la tester, et en apprécier l'apport. Vérifiez votre résultat en le comparant aux références.

2.4 Ombrage (10 pts)

Implémentez le calcul des rayons d'ombre dans `Raytracer::shade` et mettez à jour le calcul d'illumination en fonction. Les rayons d'ombre doivent être émis à partir d'un point pour lequel vous calculez l'illumination locale, afin de déterminer quelles sources lumineuses contribuent à l'éclairement de ce point. Prenez bien soin d'exclure l'origine du rayon des points d'intersection (problèmes d'acné de surface), mais souvenez-vous que le point d'intersection pourrait être sur le même objet si celui-ci n'est pas convexe (par exemple, la théière). Si le point s'avère être dans l'ombre, aucune couleur (sauf le terme ambient) ne devrait être émise.

2.5 Réflexion miroir (25 pts)

Implémentez les rayons secondaires de réflexion dans `Raytracer::shade`. Utilisez la variable de profondeur rayDepth pour stopper la récursion (la valeur par défaut dans la solution est 10, mais dans des situations particulières, monter à 100 donnera des images jolies en complexité, tout en augmentant le temps de calcul). Mettez à jour le calcul de l'illumination à chaque étape pour prendre en compte cette seconde composante. Vous pouvez voir cette étape comme une extension du calcul des rayons d'ombre, déterminant la contribution de la lumière récursivement (et pondérant l'illumination nouvellement déterminée dans le pixel d'origine).

2.6 Licence créative (20 pts)

Attaquez vous à cette partie seulement une fois que vous avez fini avec les consignes précédentes. Maintenant il est temps de jouer un peu plus et d'atteindre la note maximale! Ci-dessous, nous proposons quelques suggestions que vous pourriez explorer :

1. **Géométrie conique** : Implémentez `Conic::localIntersect` pour activer les intersections entre le rayon et des surfaces coniques (https://en.wikipedia.org/wiki/Cone). Notez que ceci requiert de détecter les limites du cercle des coniques et de les traiter correctement.
2. **Réfraction** : Implémentez une seconde boucle récursive pour les rayons réfractés. Utilisez la même variable de profondeur `rayDepth` pour arrêter la récursivité. Mettez à jour le calcul de l'illumination à chaque étape en fonction de cette autre composante.
3. **Textures** : Vous pourriez avoir besoin d'utiliser une librairie annexe ici, comme `OpenCV` ou `libpng`, afin d'importer et utiliser des textures dans l'algorithme de lancer de rayon pour obtenir une couleur diffuse définie localement.
4. **Accélération** : Considérez l'amélioration de votre algorithme en utilisant une des méthodes de partitionnement de l'espace vues en classe. Comparez votre résultat à la version originale de votre lancer de rayons.
5. **Autres** : Normal mapping, lancer de rayons GPU, occlusion ambiante, ombre douces, ...

Notez que créer de nouvelles scènes en écrivant vos propres fichiers de scène ne compte pas comme une licence créative, mais c'est pour sûr amusant à faire !

3 Détails Importants

3.1 Remise

Les fichiers à remettre sont les suivants :

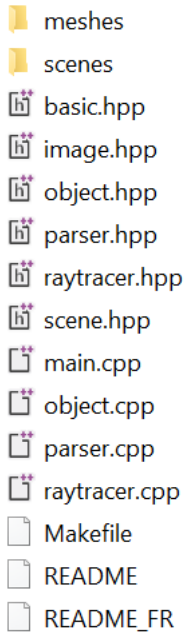

FIGURE 1 – Fichiers à remettre

Vous pourrez remplacer le contenu du fichier `README` pour y inclure votre nom, prénom, matricule ainsi que les objectifs que vous aurez réalisé dans la partie 2.6 Licence Créative.

Il est très important de remettre tout ces fichiers dans leur intégralité pour que votre solution soit testée correctement lors de la correction. Les fichiers devront être compressés dans une archive avec l'extension `.zip` ou `.rar` et avec le nom au format suivant : `TP4_NOM_PRENOM_MATRICULE`. Les fichiers doivent être compressés directement, ne compressez pas un dossier contenant les fichiers remettre, sélectionner tous les fichiers et compressez les ensuite.

3.2 Efficacité

La vitesse d'exécution de votre solution n'est pas directement évaluée. Par contre, une solution particulièrement lente pourrait indiquer une erreur d'implémentation qui pourrait causer une perte de point. Pour référence, une solution qui produit le rendu de la scène `basic.ray` a un temps d'exécution de 2.13 secondes sur une machine possédant un processeur `Intel i7-10750H`. L'execution du code est *single threaded* et est faite entièrement sur le CPU.

3.3 Calculs vectoriels

Tout comme dans le TP2, des fonctions vous permettant d'effectuer des calculs vectoriels plus proprement sont implémentées. La classe `Vector` présente dans le fichier `basic.hpp` implémente plusieurs opérateurs très utiles.

```
Vector a;
Vector b = {0,0,0,0};
Vector c(0);
double scalar = 2;
Vector d = a + b; //Addition traditionnelle de vecteurs ignorant la 4e composante (
    fonctionne aussi avec la soustraction)
double e = a.dot(b); //Produit scalaire de a et b
Vector f = a.cross(b); //Produit vectoriel de a et b
Vector g = scalar * a; //Multiplication du vecteur a par un scalaire (fonctionne aussi avec
    la division)
g += a; //Auto addition
a.print(); //Imprime les valeurs du vecteur dans la console sans nouvelle ligne
a[0]; //La premiere composante du vecteur
double l = a.length(); //la norme du vecteur
double l2 = a.length2(); //la norme au carre du vecteur
Vector h = a.normalized(); //le vecteur a avec une longueur unitaire (ne modifie pas le
    vecteur a)
a.normalize(); //Modifie le vecteur a pour le normalise
```

3.4 C++

Bien que le devoir soit en `C++`, ne vous inquiétez pas. Comme la grande majorité du code est déjà écrite pour vous, les notions de `c++` nécessaires sont très faible. Les pointeurs et les références représentent une grosse partie du langage, mais seules quelques notions vous seront nécessaires.

Lorsque l'on passe une référence à une fonction, l'objet référé sera modifié dans la fonction appelante si il est modifié dans la fonction appelé. Vous utilisez ce principe lorsque vous implémenterez l'intersection des rayons avec des surfaces. Soit le code suivant :

```
//Imaginez que les variables scene et ray sont initialisees.
Intersection hit = Intersection();
bool pass = scene.objects[i]->intersect(ray,hit);
```

La méthode `Object::intersect(Ray ray, Intersection& hit)` est appelé ici. Notez que la signature de la méthode indique que `hit` est une référence. La variable `hit` sera donc modifié si la méthode `Object::intersect` la modifie.

Les méthode `Raytracer::trace()` `Raytracer::shade()` utilisent aussi des références. Notez que certaines variables sont accompagnées du mot clé`const` qui indique que c'est variables ne peuvent pas être modifiées.

Prenez bien soins de parcourir les fichiers présents dans le devoir et de lire les commentaires pour mieux comprendre le fonctionnement du programme. Les commentaires contiennent aussi des pistes à suivre et des indices.