IFT3355 : Infographie
Travail Pratique 4 : Raytracing

Remise : mercredi 8 avril 2020, 23:59

1 Introduction

Dans ce TP vous implémenterez un simple algorithme de lancer de rayons (ou un lancer de chemins si vous êtes courageux) qui produit l'illumination locale, les ombres et la réflection. Les objets géométriques supportés seront des sphères, des plans, des maillages triangulaires, et optionnellement d'autres types de surfaces.

1.1 Description du template

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 scène (dans le dossier *scenes/*, la description du format peut être trouvée dans *scenes/basic.ray*) et 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 et 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 basiques tels que les vecteurs, les matrices, les rayons et les intersections sont définis dans `basic.hpp`.

Il y a trois sous-dossier : *scenes/*, *meshes/* et *referenceResults/*. Le dossier *scenes/* contient les fichiers de scènes 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 dossier *meshes/* et la plupart des scènes dépendent d'un ou plusieurs de ces fichiers. Enfin, dans le dossier *referenceResults/* se trouvent les résultats de référence afin que vous puissiez comparer vos résultats.

Votre travail sera principalement restreint aux fichiers `object.cpp` et `raytracer.cpp`. Vous n'avez pas besoin de modifier les autres fichiers source (mais vous pouvez si vous souhaitez ajouter des fonctionnalités additionnelles optionnelles). De plus, 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.

1.2 Instructions 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 2013. 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.

Pour lancer l'exécutable : ./a4.exe or ./raytracer

Par défaut, cela va rendre une scène avec seulement un cube rouge et sauvegarder l'image de sortie *default_output.bmp* dans *scenes/* où il y aura également une image de profondeur *default_output_depth.bmp* pour laquelle le blanc est le lointain et le noir est la proximité.

Pour lancer l'exécutable et rendre une scène différentes, indiquez là comme premier argument : ./a4.exe your_scene_file_path

L'image résultat sera également sauvegardé dans le dossier *scenes/* et nommée *your_scene_file_path_output.bmp* (avec *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.

2 Devoir (Total : 100 pts)

Nous vous recommandons fortement de suivre l'ordre des objectifs et d'implémenter l'algorithme étape par étape. Celui-ci doit lancer des rayons primaires dans la scènes, qui produiront des rayons d'ombre et des rayons secondaires/réfléchis. Notez que le rendu de scènes complexes composés 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 nos résultats de référence pour noter le projet.

***Optionnellement**, vous pouvez à la place implémenter un algorithme de tracer de chemins. En implémenter le noyau est plus complexe mais la partie créative sera bien plus simple.*

2.1 Lancer des rayons (20 pts)

1. Implémentez les parties manquantes de Raytracer::render pour lancer des rayons basiquement 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 re-calculant 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 implémentez une des fonction de test d'intersection, vous devriez avoir quelques informations de profondeur sur votre image de sortie.

2.2 Tests d'intersection (15 pts)

Dans cette partie, vous implémenterez premièrement 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.

Etapes :

1. Implémentez Sphere::localIntersect. Pour cette partie, vous devez calculer si un rayon a intersecté votre sphère. Soyez sûr de couvrir toutes les scénarios d'intersection possibles (zéro, un ou deux points d'intersection). Testez votre résultat en comparaison 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 apparaîtrons 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 concentrez 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. Premièrement, vous pouvez supposer que toutes les sources de lumière sont directement visibles.

Vous devrez calculer les composantes ambiante, diffuse et spéculaire. Voyez cette partie comme l'étape déterminant la couleur à un point où le rayon intersecte la scène. Une fois finie, vous devriez avoir une image colorée avec l'illumination locale, comme dans le TP3. 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, 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, multipliez leur couleur originale par le facteur (1 - *material.shadow*).

2.5 Réflection (25 pts)

Implémentez les rayons secondaires de réflection dans Raytracer::shade. Utilisez la variable de profondeur rayDepth pour stopper la récursivité (la valeur par défaut dans la solution est 10). 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 antre 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 les traiter correctement (pour obtenir des parties de cylindres/cones/ellipsoides finies).

2. **Réfraction** : Implémentez une seconde boucle récursive pour les rayons réfractés. Utilisez la même variable de profondeur rayDepth pour stopper 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. **Brillance** : Utilisez des estimations de direction aléatoire pour prendre en compte les surfaces brillantes en plus des surfaces spéculaires.

6. **Autres** : Normal mapping, lancer de rayons GPU, occlusion ambiante, ombres 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 !