IFT3355 : Infographie
Travail Pratique 2 : Lancer de rayons

Remise : 9 novembre 2020, 23:59

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), la réflexion miroir, la transparence, la réfraction, etc. Il y a beaucoup de parties à implémenter, mais elles se divisent bien en petit projets pour des coéquipiers bien synchronisés ou bien autonomes.

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 ètape. 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.

Vous constaterez aussi que la proportion de points attribuée à chaque partie est parfois très loin de l'effort que vous aurez à y consacrer. Ordonner les tâches en fonction des points est une façon intelligente d'optimiser vos efforts et vos heures de sommeil. Communiquez aussi bien à votre coéquipier vos découvertes, problèmes et subtilités d'une partie du code, car une question à l'examen intra pourrait s'inspirer de cette expérience.

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`.

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". Des référence supplémentaire seront ajoutées sur Studium au cour du mois

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 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 : `./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 :
`./TP2.exe your_scene_file_path`

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. Certaines parties se divisent bien entre coéquipiers, surtout dans les dernières étapes. Bien entendu, chaque coéquipier devrait bien comprendre le travail de l'autre... 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 (10 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 (10 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 Éclairage (20 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 Rayons secondaires (25 pts)

2.4.1 Ombrage (5 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.

2.4.2 Réflexion miroir (5 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). Mais contrairement à l'ombre, la réflexion miroir peut engendrer beaucoup plus de rayons secondaires.

2.4.3 Réfraction (10 pts)

Implémentez une seconde boucle récursive pour les rayons réfractés. Utilisez la même variable de profondeur rayDepth pour arrêter la récursion. Mettez à jour le calcul de l'illumination à chaque étape en fonction de cette autre composante.

La réfraction s'applique dans le bon ordre à chaque intersection pour un objet. On supposera que des objets réfractifs n'occupent pas le même espace 3D ; la raison principale en est une de définition ambiguë, car si une région 3D est commune entre deux sphères qui s'intersectent par exemple, l'ordre de traversée pourrait alterner l'indice de réfraction de la partie commune. Attention : Les choses peuvent se compliquer dans le cas d'objets concaves, car les intersections multiples calculées le long d'un rayon pour un objet concave pourraient ne plus être valides si une partie de ce rayon est bloquée ou réfractée par un autre objet entre les parties concaves. Un même objet pourrait alors avoir à être traité/intersecté à plusieurs reprises, ce qui crée un cas particulier car des intersections plus distantes pouraient ne plus être valides. Testez au début sans vous en soucier, et décidez par la suite si vous voulez traitez ces cas spéciaux avec une solution générale.

Nous supposerons que la caméra est dans un milieu neutre, i.e., avec un indice de réfraction à 1.0. Cela implique qu'on ne peut pas ni ne doit pas mettre la caméra dans un objet réfractif. Ceci inclut le cas qui peut sembler simple mais qui est en réalité ambigu de la réfraction des objets de type plan, car il est très facile de placer deux plans réfractifs avec un espace commun.

Dans le cas des ombres, un objet réfractif sera considéré comme un objet opaque.

2.4.4 Transparence (5 pts)

Implémentez l'effet de transparence d'une surface, i.e., qui n'affecte pas la direction incidente à la surface (autrement dit, à l'intersection). La transparence d'un objet se produit à chaque intersection avec la surface, dans le bon ordre, et ainsi, elle est cumulative, en ce sens qu'elle affecte la couleur à chaque fois que le rayon traverse une surface de cet objet. Attention : l'ordre des transparences entre plusieurs objets et surfaces est importante. Contrairement à la réfraction, il n'y a pas d'ambiguité de définition des régions 3D, car la transparence se produit à la surface, et non dans le volume. Attention : Un objet réfractif dans un objet transparent pourra affecter les intersections plus distantes de l'objet transparent.

La transparence devrait affecter l'éclairage, mais dans le cas de l'ombrage, une surface transparente sera considérée comme opaque dans le contexte suréchantillonnage, mais transparente (comme ci-haut) pour le calcul de l'illumination incidente au milieu de la sphère/point de lumière.

2.5 Texture (15 pts)

Implémentez une application (mapping) sous forme de tuiles (tiling) pour les textures, qui affecteront seulement des plans. La même texture (image) est ainsi répétée de multiples fois sur le plan. Une telle fonction de mapping se trouve dans les notes de cours sur les textures.

2.6 Suréchantillonnage au pixel (10 pts)

Implémentez un suréchantillonnage de couleur pour chaque pixel. Chaque pixel devrait avoir un nombre x de rayons qui déterminent sa couleur finale. Une pondération égale pour chaque rayon (i.e., calculer la moyenne) correspond à un filtre boîte. D'autres pondérations existent, et nous en parlerons au cours durant la partie sur l'échantillonnage.

2.7 Suréchantillonnage lumière sphérique (10 pts)

Implémentez un suréchantillonnage de lumière sphérique pour calculer la pénombre à un point donné. En gros, vous devriez lancer n rayons vers la sphère représentant le point de lumière (notée avec un rayon de longueur r). La bonne façon de procéder calcule le disque de la sphère tel que vu par le point à éclairer. n points sont générés aléatoirement sur ce disque (on pourra voir deux façons en démo), et chaque rayon correspondant est intersecté avec la scène. Le ratio de rayons d'ombre intersectant une surface opaque donnera la proportion du facteur de pénombre.

Attention au cas spécial des surfaces transparentes. Dans le cas d'une surface réfractive, vous pouvez simplement la considérer comme opaque.

Notez que créer de nouvelles scènes en écrivant vos propres fichiers de scène n'apportera pas de points supplémentaires à votre note finale, mais c'est pour sûr plus amusant de laisser

votre créativité s'exprimer ! Ne perdez pas trop de temps, mais une fois tout fini...

NB. Un objet transparent ne peut pas être réfractif, et vice versa