

IFT 3355 - TP2: Lancer de rayons - 20%

Date disponible: Vendredi 4 Octobre avant 23h59

Date de remise: Jeudi 7 Novembre à 23h59

Équipe de deux au maximum

Contents

1 Introduction

- 1.1 Description du squelette de code
- 1.2 Installation
 - 1.2.1 Windows
 - 1.2.2 Linux
- 1.3 Configuration

2 Devoir

- 2.1 Premiers pas [10%]
- 2.2 Test d'intersection [20%]
- 2.3 Structure accélératrice [20%]
 - 2.3.1 AABB (Axis-Aligned Bounding Box) [10%]
 - 2.3.2 Naive [5%]
 - 2.3.3 BVH (Bounding Volume Hierarchies) (5%)
- 2.4 Shading [45%]
 - 2.4.1 Éclairage local [10%]
 - 2.4.2 Ombres [5%]
 - 2.4.3 Pénombre [5%]
 - 2.4.4 Réflexion miroir [5%]
 - 2.4.5 Réfraction [5%]
 - 2.4.6 Texture [15%]
- 2.5 Profondeur de champ [5%]

3 Informations Importantes

- 3.1 Commentaires
- 3.2 Remise
- 3.3 Efficacité

4 Appendices

- 4.1 Calcul vectoriel
- 4.2 C++

1 Introduction

Dans ce TP vous implémenterez un algorithme “simple” de lancer de rayons (un ou plusieurs rayons par pixel) qui calcule l'éclairage local, les ombres, la réflexion et la réfraction à l'aide d'une structure accélératrice.

Vous devriez en profiter pour subdiviser le travail ainsi que de progresser pas à pas. Commencez par implémenter les algorithmes en ordre de complexité. Ensuite, vous comprendrez beaucoup mieux les problèmes soulevés et vous serez plus aptes à résoudre le problème en entier. Jadis, on appelait cette approche en anglais “throw-away code”. Un avantage de cette approche est lié au fait que si la partie complexe de votre code ne fonctionne pas, vous pouvez revenir en arrière et la partie de base fonctionnera encore.



Prenez-vous d'avance!!! Ce projet peut prendre énormément de temps.

1.1 Description du squelette de code

L'entrée du programme est définie dans `main.cpp`. Le programme utilise `Parser` (défini dans `parser.h` et `parser.cpp`) pour analyser les fichiers de la scène (dans le répertoire `data/scene/`). Ce dernier s'occupe d'initialiser la scène (définie dans `scene.h`) avec les paramètres et objets spécifiés.

Ensuite, `Raytracer` (défini dans `raytracer.h` et `raytracer.cpp`) est utilisé pour *rendre* la scène dans un objet `Frame` qui produit deux fichiers `.bmp` en sortie (via `frame.h`). La première image sera le shading tandis que le deuxième sera votre buffer de profondeur. La scène peut contenir des objets géométriques (sphères, quads, etc.), des lumières ainsi que quelques paramètres globaux tels que la lumière ambiante.

Il y a trois répertoires dans `data/` : `scene/`, `ref/` et `out/`. Le répertoire `scene/` contient les fichiers de scène au format `.ray` décrivant les paramètres de la scène suivants : `*container`, `dimension`, `samples_per_pixel`, `max_ray_recursion`, `ambient_light`, `Perspective`, `LookAt`, `DOF`, `Material`, `PushMatrix`, `PopMatrix`, `Translate`, `Rotate`, `Scale`, `Sphere`, `Plane`, `Cylindre`, `Mesh` et `Light`. Le fichier `all_at_once.ray` est commenté afin d'expliquer le format. Dans le répertoire `ref/` se trouvent des résultats de référence afin que vous puissiez comparer vos résultats. Enfin, dans le répertoire `out/`, vous trouverez les images générées suite à l'exécution.

On ne s'attend pas à ce que vos résultats soient identiques mais ils devraient bien entendu “ressembler” fortement aux images de référence.

Votre travail sera restreint aux fichiers `object.cpp`, `raytracer.cpp`, `container.cpp` et `aabb.cpp`. Vous êtes **vivement** encouragés à regarder les fichiers `*.h` afin de bien comprendre les classes C++ ce qui vous aidera avec votre code. Les fichiers `frame`, `parser` et `resource_manager` sont moins importants, mais éducatifs.

Les outils mathématiques de base tels que les vecteurs et les matrices sont fournis par la librairie `linalg.h`. (Voir [Appendices])

1.2 Installation

La première étape est de télécharger Visual Studio Code pour votre OS. Plusieurs ressources existent en ligne pour vous aider durant l'installation.

1.2.1 Windows

Compilateur C++: Télécharger un compilateur C++ avec VS 2022, WSL ou MinGW.

CMake: Installer CMake sur le site web. Lors de l'installation, bien cocher l'ajout de l'exécutable CMake dans les variables d'environnement.

1.2.2 Linux

Compilateur C++: Utiliser votre package manager pour installer gdb.

CMake: Installer CMake grâce à votre package manager également.

1.3 Configuration

Lancez Visual Studio Code. Installer les extensions “C/C++” et “C/C++ Extension Pack”.

Ensuite, vous pourrez ouvrir le dossier contenant le projet CMake, en faisant **Open Folder...**. À l'ouverture du dossier par Visual Studio Code, vous devriez voir apparaître une petite fenêtre de configuration CMake. Veuillez sélectionner un des compilateurs dans la liste proposée.

Vous devriez voir la version du compilateur en bas à gauche de votre écran. Cliquez sur **CMake: [BUILD_TYPE]: STATUS** pour configurer le mode de compilation: **Debug** compile votre code sans optimisation, mais permet d'être couplé avec un Debugger. **Release** optimise l'exécution du code, mais enlève tous commentaires pouvant aider le développeur. **RelWithDebInfo** est un entre-deux.

Vous pouvez compiler le projet en cliquant sur “build”. Si tout se passe bien, vous devriez voir le code de sortie 0.

Pour lancer votre programme, ouvrez votre terminal et écrivez l'exécutable avec le nom de la scène.

```
#Linux  
./RAY [scene].ray
```

```
#Windows  
RAY.exe [scene].ray
```



Windows : Si vous obtenez une erreur avec *cmake*, veuillez essayer en mode administrateur.



Utilisez RelWithDebInfo lorsque vous avez une version stable afin de progresser plus rapidement.

2 Devoir

Nous vous recommandons fortement d'implémenter les algorithmes en ordre de complexité. L'algorithme doit lancer des rayons primaires dans la scène qui généreront des rayons d'ombre

et d'autres rayons secondaires. Notez que le rendu de scènes complexes composées de plusieurs primitives peut prendre du temps!

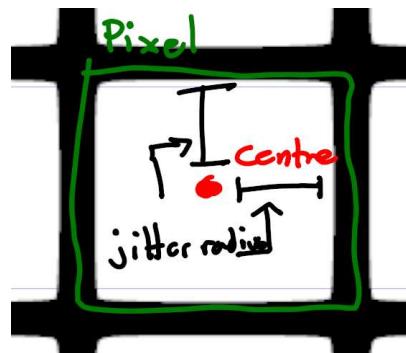


Ne changez aucun des fichiers de scène. Nous comparerons vos résultats avec les résultats sur nos propres scènes de référence pour évaluer votre travail.

2.1 Premiers pas [10%]

Pré-requis: Aucun

1. `raytracer.cpp` - Raytracer::render 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 en vérifiant que vous obtenez les bonnes coordonnées.



Définition du Jitter Radius. Les unités sont relatives à la taille du pixel. 0.5 correspond à un échantillonnage uniforme.



`tan()` prend des radians plutôt que des degrés. Utilisez `deg2rad()` pour la conversion.



Tous les rayons dont les coordonnées sont exprimées dans le repère monde doivent avoir une direction normalisée due à la convention adoptée dans notre moteur de rendu.



Utilisez les fonctions dans `basic.h` - `rand*()` pour y arriver.

2. `container.cpp` - Naive::intersect Implémentez la fonction d'intersection pour le contenant "Naive". Mettez à jour la profondeur en fonction de l'intersection la plus

- rapprochée. (N'implémentez pas l'intersection AABB pour le moment)
3. `raytracer.cpp` - `Raytracer::trace` Mettez à jour les couleurs et le buffer de sortie s'il y a intersection en fonction de `raytracer.cpp` - `Raytracer::shade`. (Ignorez le lancer de rayons secondaires pour la réflexion et la réfraction pour le moment. Laissez `raytracer.cpp` - `Raytracer::shade` vide pour le moment).

2.2 Test d'intersection [20%]

Pré-requis: Premiers pas

Vous devez implémenter les algorithmes d'intersections pour les quatre géométries: Quad, Sphere, Cylinder, Mesh dans le système de coordonnées locales.



Prenez soin d'inverser les normales. On suit la convention suivante: Les normales doivent pointer dans le sens opposé du rayon incident.

- `object.h` - `Quad::local_intersect`: Soyez sûrs de couvrir tous les scénarios (aucun ou le point d'intersection le plus rapproché de l'origine du rayon). Respectez les contraintes associées à la géométrie. Mettez à jour le point d'intersection avec la normale correspondante dans le repère local.



Faites attention au cas lorsque le rayon est parallèle au plan de support.

- `object.h` - `Sphere::local_intersect`: Soyez sûrs de couvrir tous les scénarios d'intersections possibles (aucun ou le point d'intersection le plus rapproché de l'origine du rayon). Respectez les contraintes associées à la géométrie. Mettez à jour le point d'intersection avec la normale correspondante dans le repère local.



Faites attention lorsque l'origine du rayon est à l'intérieur de la sphère.

- `object.h` - `Cylinder::local_intersect`: Soyez sûrs de couvrir tous les scénarios d'intersections possibles (aucun ou le point d'intersection le plus rapproché de l'origine du rayon). Le cylindre est considéré "vide" (i.e., un tube) et sans base ni cap. Respectez les contraintes associées à la géométrie. Mettez à jour le point d'intersection avec la normale correspondante dans le repère local.



Faites attention au cas où le rayon est parallèle à la surface du cylindre.

- `object.h` - `Maillage::local_intersect`: Cette fonction calcule le point d'intersection d'un rayon avec un maillage. Dans notre cas, tous les maillages sont formés de triangles. Pour chaque triangle, vous devez déterminer si le point est à l'intérieur à l'aide de la fonction `Mesh::intersect_triangle`. Si tel est le cas, mettez à jour le point d'intersection avec la normale correspondante en interpolant les normales associées à chaque sommet.



Faites l'interpolation des normales et coordonnées de texture en fonction de la position sur le triangle.

La profondeur d'intersection est affichée dans les fichiers `depth.bmp`. Comparez vos images de profondeur avec celles de référence.



Nous vous fournissons un code pour une caméra orthographique très simple. Cette caméra peut être utilisée pour tester l'intersection avec la sphère. Vous devez utiliser la scène de test `portho.ray` pour utiliser cette caméra.

Notez que votre code de caméra ne doit pas être basé sur le code de la caméra orthographique. Ce code n'est là que pour prendre en compte le développement initial du test d'intersection. Vous trouverez des instructions sur la façon d'utiliser le code dans `raytracer.cpp` - `Raytracer::render`.

2.3 Structure accélératrice [20%]

Pré-requis: Test d'intersection (au moins un algorithme implémenté)

2.3.1 AABB (Axis-Aligned Bounding Box) [10%]

Implémentez les algorithmes suivants:

- `aabb.cpp` - `retrieve_corners`: Retrouvez les huit coins associés au AABB.
- `aabb.cpp` - `construct_aabb`: Construisez un AABB avec le plus petit volume englobant la liste de points fournis.
- `object.h` - `*:compute_aabb`: Calculez la boîte englobante dans le repère **global** associé à chaque géométrie. Construisez un AABB avec le plus petit volume possible pour chaque géométrie. Utilisez les fonctions précédentes pour y arriver.



Cylindre/Sphère: Pour y arriver, nous allons passer par une approximation puisque la solution exacte est un peu plus compliquée. Donc, en premier lieu, vous devez calculer le AABB dans l'espace local. Ensuite, à partir du AABB, retransformez les coins dans le repère global. Construisez le AABB final à partir des points transformés.



Quad: Assurez-vous que le AABB ait un volume. Ajoutez un petit epsilon dans le bon sens au besoin.

- aabb.cpp - `AABB::intersect`: Implémentez l'algorithme d'intersection entre le rayon et le AABB. Cette fonction ne retourne que vrai ou faux en fonction de l'intersection. Faites attention parce que cette fonction pourrait permettre de grandement réduire le temps de calcul.



Assurez-vous que cette fonction soit optimisée. Évitez les opérations inutiles et les conditions en trop. Cette fonction est essentielle afin d'accélérer le raytracer.

2.3.2 Naive [5%]

Pré-requis: **AABB**

container.cpp - `Naive::intersect`: Implémentez la fonction d'intersection pour le contenant "Naive". Mettez à jour la profondeur en fonction de l'intersection la plus rapprochée. Faites une intersection avec le AABB avant d'effectuer l'intersection plus couteuse avec la géométrie.

Vous devriez observer une accélération notable si vous avez implémenté les algorithmes AABB ci-haut dans la plupart des cas. Certains cas sont dégénérés... Pensez-y!

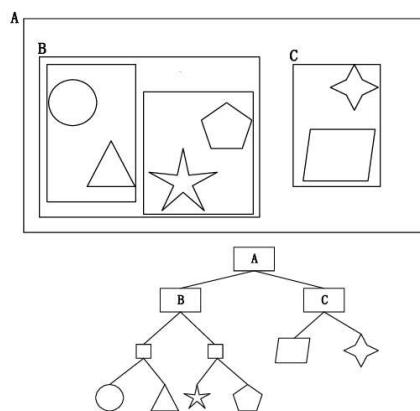
2.3.3 BVH (Bounding Volume Hierarchies) (5%)

Pré-requis: **AABB**

container.cpp - `BVH::intersect`: Implémentez la fonction d'intersection pour le contenant "BVH". Ce contenant peut avoir plusieurs représentations, mais nous allons le représenter à l'aide d'un arbre binaire. Parcourez l'arbre en profondeur. S'il s'agit d'une feuille, effectuez

l'intersection avec la géométrie. Sinon, parcourez les enfants. Si le rayon intersecte l'un de leurs AABBs, visitez leurs enfants également et répétez les opérations.

Les structures BVH sont l'une des structures les plus utilisées en lancer de rayons pour accélérer les temps de calculs. (ex: Les nouvelles cartes Nvidia RTX implémentent ces algorithmes directement sur les cartes.) Il existe plusieurs variantes avec plusieurs compromis. La version que vous avez programmée n'est pas du tout optimale dans l'état. En conséquence, le temps d'accélération n'est pas incroyable. Par contre, vous avez déjà une base pour comprendre toutes les dérivations.



Structure BVH avec Arbre Binaire

2.4 Shading [45%]

Pré-requis: **Test d'intersection (au moins un algorithme implémenté)**

2.4.1 Éclairage local [10%]

Implémentez la partie manquante de `raytracer.cpp` - `Raytracer::shade` qui réalise le calcul d'éclairage permettant de trouver la couleur à un point. Vous devez 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. (Ignorez les ombres pour le moment.) Renseignez-vous sur les paramètres en lisant le code fourni.

Lorsque cette partie est complétée, vous devriez avoir une image colorée avec de l'éclairage local. Vérifiez votre résultat avec les images de référence dans `color.bmp`.

2.4.2 Ombres [5%]

Pré-requis: Éclairage local

Implémentez le calcul des rayons d'ombre dans `raytracer.cpp` - `Raytracer::shade` et mettez à jour le calcul d'éclairage en fonction. Les rayons d'ombre doivent être émis à partir d'un point pour lequel vous calculez l'éclairage local, afin de déterminer quelles sources lumineuses contribuent à l'éclairage de ce point. Prenez bien soin d'exclure l'origine du rayon des points d'intersection (problème d'acné de surface) et les intersections plus loin que la source. Souvenez-vous que le point d'intersection pourrait être sur le même objet si celui-ci n'est pas convexe. Si le point s'avère être dans l'ombre, le terme d'éclairage devrait être de 0 pour cette source de lumière.



Faire attention au problème d'acné.

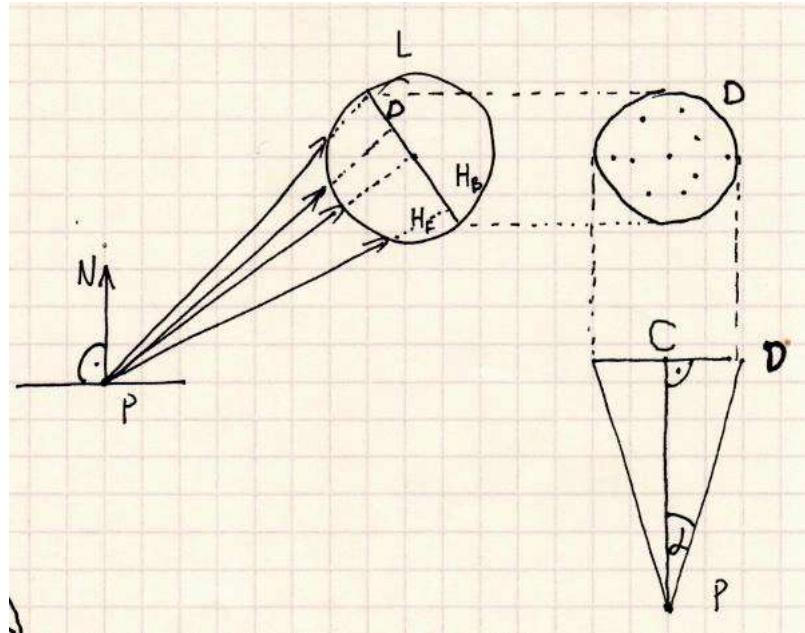
2.4.3 Pénombre [5%]

Pré-requis: Ombres

Implémentez la pénombre pour les lumières sphériques entre le point d'intersection et la lumière. Calculez l'ombre comme précédemment, mais inclure un facteur d'occlusion en fonction du nombre de rayons qui atteignent la lumière. Pour y arriver, prenez uniformément une direction à l'intérieur du cône entre le point d'intersection et la lumière sphérique. Au lieu de simplement mettre la contribution à 0, remplacez ce dernier avec le facteur d'occlusion.

Pour échantillonner uniformément des directions dans le cône, nous fournissons une routine d'échantillonnage d'un disque unitaire dans `basic.h` - `random_in_unit_disk()`. Il vous faudra trouver la bonne position, orientation et échelle de ce disque.

Enfin, moyennez la couleur récoltée en fonction du nombre de rayons envoyés.



Cône d'intersection avec la lumière sphérique.



Utilisez les fonctions dans `basic.h` - `rand*()` pour y arriver.

2.4.4 Réflexion miroir [5%]

Pré-requis: Éclairage local

Implémentez les rayons secondaires de réflexion miroir dans `raytracer.cpp` - `Raytracer::trace`. Utilisez vos notes de cours pour déterminer la nouvelle direction du rayon. Une fois que vous avez votre résultat, mettez à jour le shading pour le point d'intersection en pondérant selon la couleur réfléchie.

2.4.5 Réfraction [5%]

Pré-requis: Éclairage local

Implémentez les rayons secondaires de réfraction dans `raytracer.cpp` - `Raytracer::trace`. Utilisez vos notes de cours pour déterminer la nouvelle direction du rayon. Prenez note que toutes nos géométries sont des surfaces et non des volumes. Une fois que vous avez votre

résultat, mettez à jour le shading pour le point d'intersection en pondérant selon la couleur transmise.



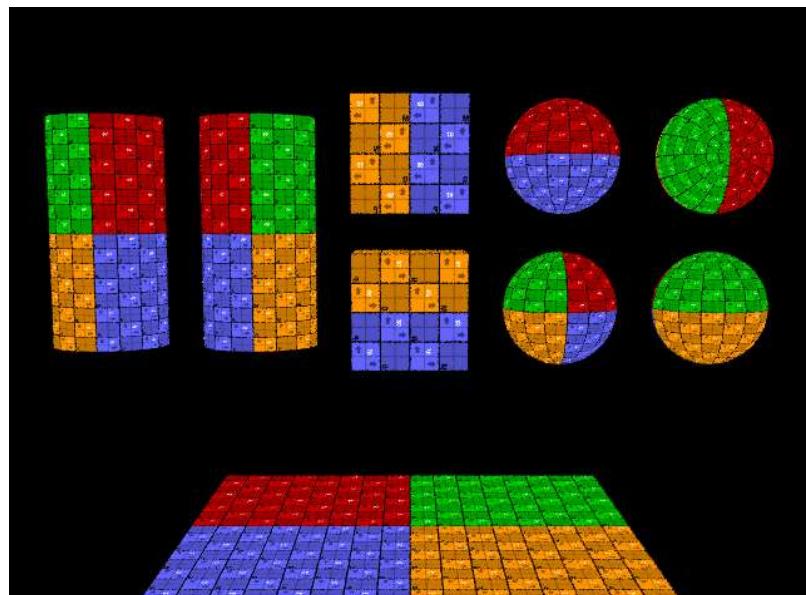
Les géométries sont remplies d'air à l'intérieur ce qui veut dire que la réfraction se fait uniquement à la surface. On fait l'approximation qu'on passe toujours de l'air au matériel en question lors de l'intersection.

2.4.6 Texture [15%]

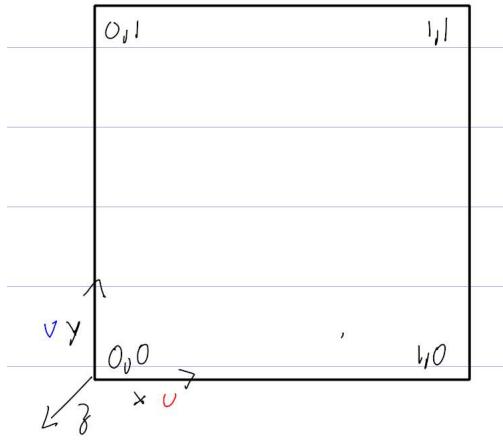
Pré-requis: Éclairage local

2.4.6.1 UV - Mapping [10%]

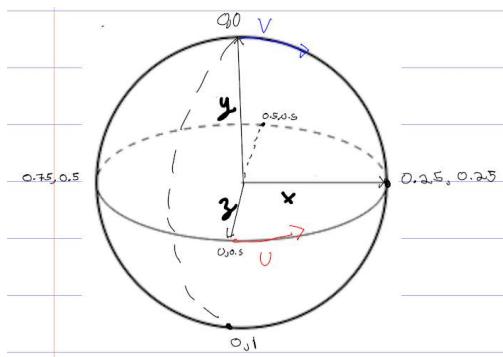
Modifiez les fonctions d'intersection pour inclure les coordonnées UV lors de l'intersection.



Vous devriez le faire pour les quatre différentes géométries.

*Coordonnées UV - Quad*

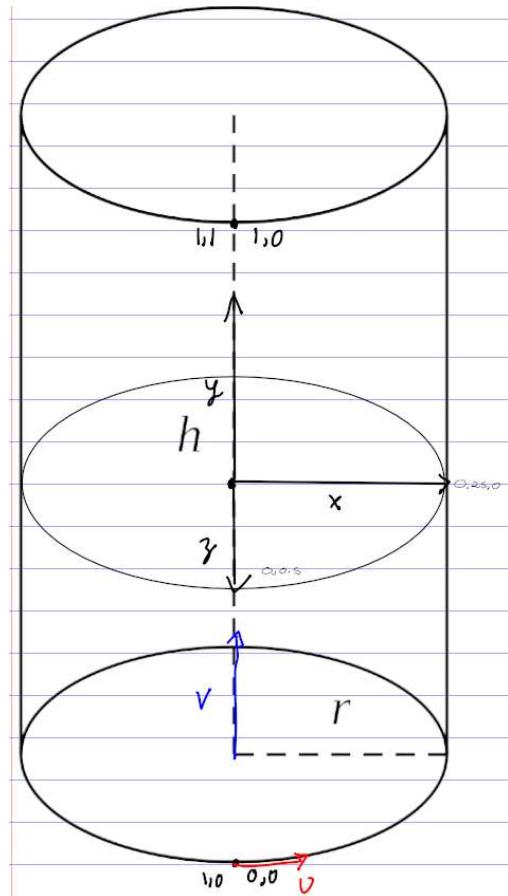
- `object.h` - `Quad::local_intersect`: Lorsqu'il y a intersection, il faut mettre à jour les coordonnées UV en fonction de la surface de la géométrie. Utilisez les coordonnées dans le plan XY afin de paramétriser la surface.

*Coordonnées UV - Sphere*

- `object.h` - `Sphere::local_intersect`: Lorsqu'il y a intersection, il faut mettre à jour les coordonnées UV en fonction de la surface de la géométrie. Utilisez les coordonnées sphériques pour y arriver.

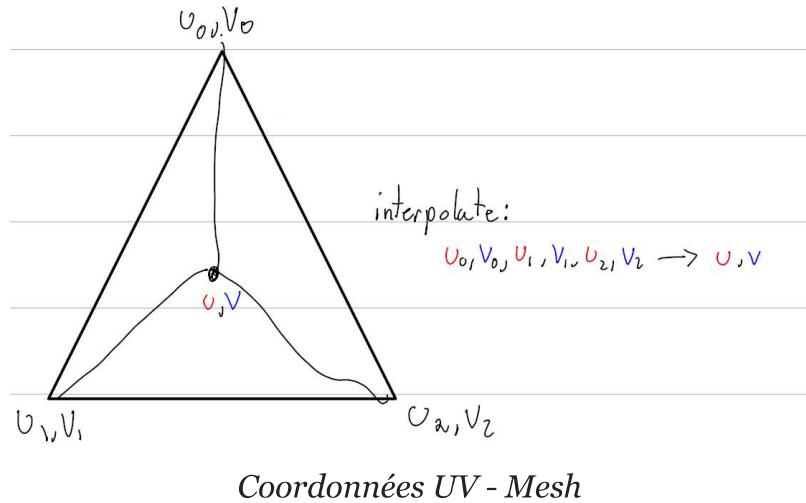


Attention aux pôles...



Coordonnées UV - Cylindre

- `object.h - Cylinder::local_intersect`: Lorsqu'il y a intersection, il faut mettre à jour les coordonnées UV en fonction de la surface de la géométrie. Utilisez les coordonnées cylindriques pour y arriver.



- `object.h` - `Mesh::local_intersect`: Lorsqu'il y a intersection, il faut mettre à jour les coordonnées UV en fonction de la surface de la géométrie. Faites l'interpolation des coordonnées de texture pour y arriver tel que vous l'avez fait pour les normales.

2.4.6.2 Couleur [5%]

Modifier `raytracer.cpp` - `Raytracer::shade` Calculez la couleur aux coordonnées UV dans la texture si cette dernière est présente. Faites attention! Les couleurs de la texture ne sont pas normalisées [0..1], mais entre [0..255] dû à la librairie `bitmap.image.h`. Modifiez le reste de l'algorithme d'éclairage en fonction du point de couleur. Si la texture est absente, prenez la couleur de `color_albedo`.

2.5 Profondeur de champ [5%]

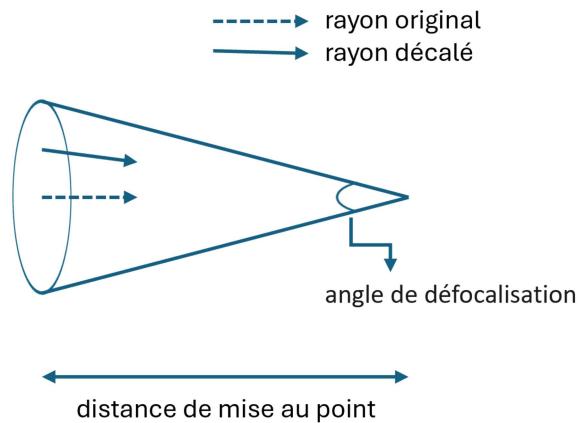
Modifiez le modèle de caméra (dans `raytracer.cpp` - `Raytracer::render`) pour prendre en compte la profondeur de champ.

Dans une caméra perspective (pinhole), tout objet est au focus (image nette), peu importe sa distance au centre de projection (origine). Une caméra réelle avec une lentille garde tout objet au focus à une certaine distance focale, mais pour tout objet qui est plus loin ou plus proche de cette distance, l'image de cet objet sera plus ou moins floue (hors focus).



Résultat sans (gauche) et avec (droite) profondeur de champ.

La distance focale (focus_distance) le long de l'axe central d'une caméra perspective définit le point focal. Si vous placez un cône le long de cet axe central avec sa pointe au point focal, l'angle de défocalisation (defocus_angle) se situe à la pointe du cône. La base du cône est un disque dont son centre est l'origine de la caméra, et son rayon (radius) est défini par l'angle de défocalisation. La normale de ce disque en 3D est orientée le long de l'axe central du cône.



Cône formé par l'angle de défocalisation et la distance de mise au point.

Pour simplifier les calculs de profondeur de champ, nous ne simulerons pas une vraie lentille. Pour un rayon donné, peu importe le pixel qu'il traverse, nous ne perturberons que la position (l'origine) de la caméra par un vecteur aléatoire à l'intérieur du disque défini ci-haut.

Dans la classe caméra, vous trouverez les paramètres nécessaires au calcul des rayons sortant de la caméra : l'angle de défocalisation (defocus_angle) et la distance de mise au point (focus_distance). Notez que la nouvelle distance focale est la distance de mise au point (focus) et vous devrez perturber l'origine du rayon par un vecteur aléatoire en utilisant la fonction random_in_unit_disk et rayon (radius) de la base du cône calculé.

Le test de scène exigera plus d'échantillons par pixel que les autres tests; le calcul devrait donc prendre plus de temps.



A noter, l'angle à la pointe est l'angle complet, en non l'angle avec l'axe central.



Si l'angle de défocalisation est égal à 0, les rayons de la caméra doivent être calculés comme s'il n'y avait pas de profondeur de champ.

3 Informations Importantes

3.1 Commentaires

Expliquez brièvement votre code à l'aide de commentaires. Nous nous attendons à ce que les commentaires soient complets et que vous fournissez une explication pour toutes les équations que vous utilisez. Par exemple, expliquez en quelques lignes les dérivations afin d'obtenir les différentes équations. Faites que votre code se maintienne sans ressource extérieure.



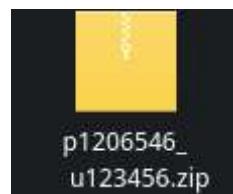
Si les commentaires ne sont pas complets ou sont excessivement longs, nous serons plus sévères lors de la correction. NE SOUS-ESTIMEZ PAS CETTE PARTIE.

3.2 Remise



Ajouter vos noms ainsi que vos matricules dans le README.

Mettez votre code dans un fichier zip nommé en fonction de vos codes d'accès (exemple pour équipe de deux: codeacces1_codeacces2.zip). Ce code d'accès est le même que vous utilisez pour vous connecter à Studium.



Exemple

Voici les fichiers à inclure dans le fichier zip.

Name	Original Size	Compressed Size	Mode	CRC checksum	Method	Date
> data	30,2 MiB		drwxr-xr-x		Store	2022-09-20 18 h 10
> extern	313,7 KiB		drwxr-xr-x		Store	2022-09-20 18 h 10
> src	73,7 KiB		drwxr-xr-x		Store	2022-09-20 18 h 10
CMakeLists.txt	1,6 KiB	479 B	-rw-r--r--	2392E20F	Deflate	2022-09-18 15 h 58
README	24 B	24 B	-rw-r--r--	8EDC9559	Store	2022-09-20 18 h 09

Fichiers à remettre

3.3 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 ou des calculs inutiles qui pourraient causer une perte de points. Pour référence, pour **all_at_once.ray**, on s'attend à ce que le temps d'exécution ne dépasse pas 30 secondes.

4 Appendices

4.1 Calcul vectoriel

La librairie **linalg.h** est particulièrement utile parce qu'elle implémente plusieurs fonctions pratiques en rendu. Voici un bref exemple des fonctions qui vous seront pertinentes.

```
#include "linalg.h"
using namespace linalg::aliases

double3 a;
double3 b = {0,0,0};
double3 c{0};
double scalar = 2;
double3 d = a + b; //Addition traditionnelle de vecteurs ignorant la 4e composante
//fonctionne aussi avec la soustraction)
double e = dot(a,b); //Produit scalaire de a et b
```

```

double3 f = cross(a,b) //Produit vectoriel de a et b
double3 g = scalar * a; //Multiplication du vecteur a par un scalaire (fonctionne aussi
avec la division)
g += a; //Auto addition
a[0]; //La premiere composante du vecteur
double l = length(a); //la norme du vecteur
double l2 = length2(a); //la norme au carre du vecteur
Vector h = normalize(a); //le vecteur a avec une longueur unitaire (ne modifie pas le
vecteur a)

```

Pour les autres fonctions, il est conseillé de jeter un oeil au code source (sous `extern/` ainsi que <https://github.com/sgorsten/linalg>).

4.2 C++

Bien que le devoir soit en C++, ne vous inquiétez pas trop. Comme la grande majorité du code est déjà écrite pour vous, les notions de C++ nécessaires sont assez limitées. 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 s'il est modifié dans la fonction appelée. Vous utilisez ce principe lorsque vous implémenterez l'intersection des rayons avec des surfaces. Soit le code suivant :

```

void Raytracer::trace(const Scene& scene,
                      Ray ray, int ray_depth,
                      double3* out_color, double* out_z_depth)
{
    //double3* out_color,out_z_depth sont des pointers. (Ils pointent vers des endroits
    dans la memoire.)

    Intersection hit;
    if(scene.container->intersect(ray, EPSILON, *out_z_depth, &hit)) {

        //& Reference: Permet d'obtenir le pointer de la variable hit.

        /* Deference: Permet d'obtenir la valeur auquelle le pointer pointe en memoire.
        ...
    }
}

```

Notez que certaines variables sont accompagnées du mot clé **const** qui indique que ces variables ne peuvent pas être modifiées. Prenez bien soin 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.