Université de Limoges
Faculté des Sciences & Techniques
Faculté des Sciences et Techniques de Limoges

ISIR - TP1 - Ray casting

Maxime MARIA

2022-2023

La base de code qui vous est proposée a été faite pour vous permettre de faire évoluer le moteur au cours du semestre. Bien entendu, elle n'est pas complète, donc en fonction de ce que vous voudrez faire, il sera sûrement nécessaire de modifier l'architecture.

Dans ce premier TP, il n'est pas nécessaire de comprendre complètement l'architecture de la base de moteur fourni. Les exercices sont faits pour que vous puissiez prendre en main le code au fur et à mesure, le plus facilement possible (tout du moins, j'ai essayé).

N.B. : Pour les maths 3D, nous utilisons la bibliothèque `glm`[1]. Des alias sont présents dans le fichier `defines.hpp` pour les types les plus utilisés (*e.g.* `Vec3f` pour `glm::vec3` ou `Mat4f` pour `glm::mat4`).

1 Première image

Dans cet exercice, vous allez simplement remplir une image en fonction des coordonnées de ses pixels. Pour cela, rendez-vous dans la méthode `Renderer::renderImage`. Cette méthode prend en paramètres une scène, une caméra et une texture correspondant à l'image à remplir. Pour le moment, ne vous souciez pas des deux premiers paramètres.

1. Parcourez les pixels de la texture et utiliser la méthode `Texture::setPixel` pour leur attribuer une couleur telle que :
   — la composante rouge soit la valeur de l'abscisse du pixel interpolée sur $[0, 1]$;
   — la composante verte soit la valeur de l'ordonnée du pixel interpolée sur $[0, 1]$;
   — la composante bleue soit à zéro.
   Vous devriez obtenir la Figure 1(a) (l'image est sauvegardée dans le dossier `results` de votre programme).
   N.B. : les couleurs sont représentées par des `Vec3f`.

(a) Exercice 1 (b) Exercice 2

Figure 1 – Résultats des exercices 1 et 2.

1. https://glm.g-truc.net/0.9.4/api/index.html

2 Caméra, rayons et projection perspective

Pour avoir une image de synthèse par lancer de rayons, il faut... lancer des rayons ! Les rayons primaires servent à déterminer ce qui est visible depuis le point de vue. Dans ce moteur, c'est la caméra qui se charge de générer ces rayons. Pour le moment, nous ne traiterons qu'une caméra perspective (classe `PerspectiveCamera`). Cependant, le moteur est prévu pour pouvoir ajouter d'autres types de caméras (en héritant de la classe abstraite `BaseCamera`).

1. Pour calculer la direction des rayons primaires, vous devez tout d'abord calculer la position et les dimensions de la fenêtre virtuelle (viewport) à travers laquelle ils vont passer. Implémentez la méthode `PerspectiveCamera::_updateViewport` qui met à jour le viewport en fonction de la configuration de la caméra. Vous pouvez vous aider de la Figure 2 qui reprend les mêmes notations que dans le code. Pour rappel, la hauteur du viewport se calcule à partir de l'angle de vue vertical (`_fovy`) et de la distance focale (`_focalDistance`) ; sa largeur se trouve avec le rapport largeur/hauteur de l'image (`_aspectRatio`).

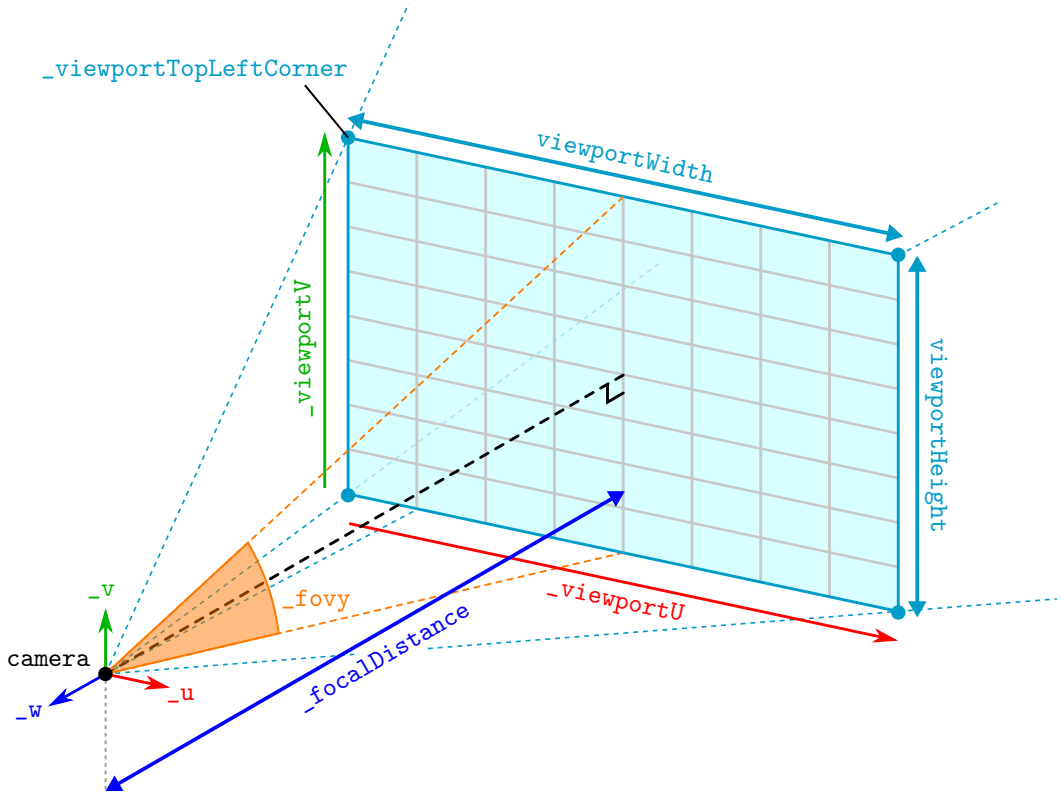


FIGURE 2 – Aide pour l'exercice 2.

2. Maintenant que le viewport est calculé, il faut générer les rayons primaires. Pour cela, la méthode `PerspectiveCamera::generateRay` prend en paramètres deux `float` (`p_sx` et `p_sy`) correspondant à une position sur le viewport (dont les coordonnées sont normalisées donc comprises entre 0 et 1) et retourne un rayon (`Ray`) dont la direction est **normalisée**. Implémentez-la.
3. Pour vérifier le fonctionnement de votre code, modifiez la méthode `Renderer::renderImage` pour générer un rayon passant par **le centre** de chaque pixel. Vous utiliserez sa direction pour appliquer une couleur au pixel tel que : `(ray.getDirection() + 1.f) * 0.5f`. Vous devriez obtenir la Figure 1(b).

3 Votre premier objet : une sphère

Avant de faire le rendu de votre premier objet, il va falloir étudier un peu la structure du moteur.

Les objets géométriques : classes `BaseObject`, `BaseGeometry`

Les objets géométriques sont représentés par une hiérarchie de classes dont la classe mère est abstraite (`BaseObject`). Elle contient deux attributs : un nom (permettant d'identifier l'objet) et un matériau (classe `BaseMaterial`, détaillée plus bas). Tous les objets héritent de cette classe et doivent donc implémenter la méthode `intersect`. Cette méthode cherche l'intersection la plus proche entre un rayon et l'objet. Si une intersection est trouvée, elle remplit la structure `HitRecord` passée en paramètre. La géométrie des objets est décrite par une hiérarchie de classes dont la classe mère est abstraite (`BaseGeometry`).

Prenons l'exemple d'un objet « sphère » (seul objet présent dans le moteur pour le moment). La classe `Sphere` hérite de `BaseObject` et représente l'objet « sphère ». Elle contient un seul attribut, `SphereGeometry _geometry` qui décrit sa géométrie.

Les matériaux : classe `BaseMaterial`

La classe abstraite `BaseMaterial` sert de base pour la définition des matériaux. Elle ne possède qu'un nom comme attribut.

Dans l'état, il n'existe qu'un seul matériau (`ColorMaterial`) dont la méthode `shade` retourne simplement sa couleur, indépendemment des directions d'observation et d'éclairage.

La scène : classe `Scene`

La classe `Scene` possède trois attributs : un ensemble d'objets (`BaseObject`), un ensemble de matériaux (`BaseMaterial`) et un ensemble de sources lumineuses (`BaseLight`). Pour l'instant, ne vous souciez pas des lumières...

La méthode `init()` permet d'initialiser la scène en dur, en ajoutant des objets (via `_addObject`), des matériaux (via `_addMaterial`), *etc.*

La méthode `_attachMaterialToObject` permet d'associer un matériau et un objet à partir de leurs noms (ce n'est pas forcément la meilleure façon de faire mais bon...).

Enfin, la méthode `intersect` parcourt les objets et cherche l'intersection la plus proche.

Le cœur du moteur : classes `Renderer` et `BaseIntegrator`

La méthode :

```
renderImage(const Scene &p_scene, const BaseCamera *p_camera, Texture &p_texture)
```

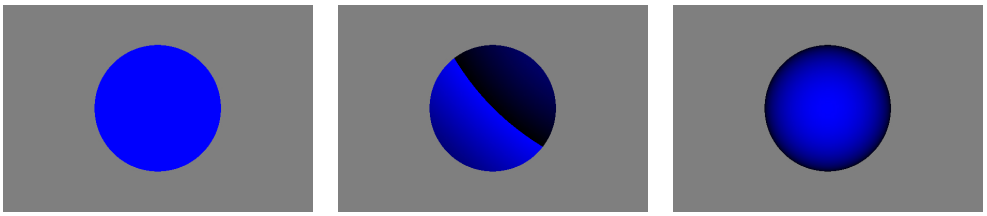
fait le rendu de la scène `p_scene` vue depuis la caméra `p_camera` et remplit la texture `p_texture`. Elle parcourt donc la texture, génère les rayons primaires et détermine la couleur des pixels.

Pour déterminer la couleur des pixels, il faut calculer la luminance qui arrive à la caméra. Cette quantité est calculée par un « intégrateur » (`BaseIntegrator _integrator`) via la méthode `Li`. Cette méthode va donc déterminer ce qui est visible depuis la caméra et calculer l'éclairage.

Dans l'état, le seul intégrateur présent dans le moteur est `RayCastIntegrator`. Il détermine simplement ce qui est visible depuis la caméra, sans prendre en comptes les éventuelles sources lumineuses.

L'exercice

1. Implémentez la méthode `SphereGeometry::intersect` qui retourne un booléen signalant s'il y a une intersection entre le rayon passé en paramètre et la géométrie. Si une intersection est trouvée, la méthode doit remplir les distances d'intersection `p_t1` et `p_t2` tel que `p_t1` $\leq$ `p_t2`. Exécutez votre code, vous devriez obtenir la Figure 3(a).
2. Le rendu est un peu « plat »... En effet, il faut prendre en compte l'angle entre la normale et la direction du rayon (le $\cos\theta$). Modifiez la méthode `RayCastIntegrator::Li` pour prendre en compte cet angle. Faites un max entre le cosinus et 0 pour gérer les cas limites. Exécutez votre code, vous devriez obtenir la Figure 3(b).
3. Normalement, vous devez trouver que le rendu semble bizarre... C'est parce que la normale de la sphère n'est pas correctement calculée. Implémentez la méthode `SphereGeometry::computeNormal` qui calcule et retourne la normale de la sphère au point `p_point`. Exécutez votre code, vous devriez obtenir la Figure 3(c). Voilà qui est mieux !



(a) (b) (c)

FIGURE 3 – Résultats de l'exercice 3.

4 Une caméra positionnable

Dans cet exercice, vous allez devoir modifier la classe `PerspectiveCamera` pour pouvoir positionner et orienter la caméra comme bon vous semble. Pour cela, il faut calculer le repère local à la caméra (`_u`, `_v`, `_w`) en fonction de la position et de la direction de la caméra (*cf.* Figure 2).

1. Implémentez le deuxième constructeur de `PerspectiveCamera` qui initialise `_u`, `_v` et `_w` avant d'appeler `_updateViewport`. Le paramètre `p_lookAt` représente le point que regarde la caméra. La direction d'observation est donc `p_lookAt - p_position`. Le paramètre `p_up` est un vecteur servant à spécifier la rotation de la caméra autour de la direction d'observation. Logiquement, il doit être situé dans le plan orthogonal à la direction d'observation (définit par `_u` et `_v`). Or au moment de l'appel du constructeur, on ne connaît pas la direction d'observation. Nous allons donc donner un vecteur `p_up` « approximatif » (*cf.* Figure 4). Généralement, on utilise le vecteur up de l'espace monde (ici y).
2. Modifiez la construction de la caméra dans la fonction `main`. Utilisez la Figure 5 pour vérifier le bon fonctionnement du constructeur. Avec 60° de FOV, vous devriez obtenir les résultats suivants :
   — Figure 5(a) : `p_position`$= (0, 0, -2)$; `p_lookAt` $= (0, 0, 79)$
   — Figure 5(b) : `p_position`$= (1, 0, 0)$; `p_lookAt` $= (1, 0, 1)$
   — Figure 5(c) : `p_position`$= (0, 1, 0)$; `p_lookAt` $= (0, 1, 1)$
   — Figure 5(d) : `p_position`$= (4, -1, 0)$; `p_lookAt` $= (-1, -1, 2)$

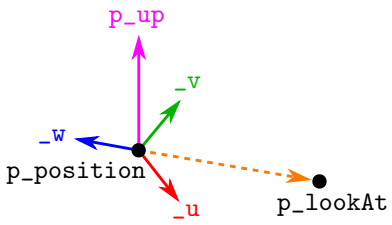
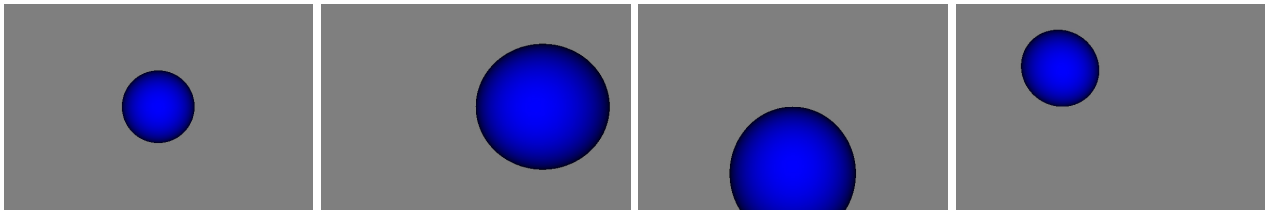


FIGURE 4 – Aide pour l'exercice 4.



(a) (b) (c) (d)

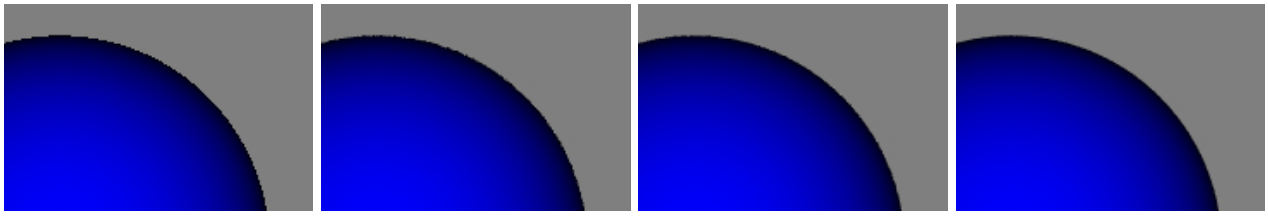
FIGURE 5 – Résultats de l'exercice 4.

5 Antialiasing

Comme vous avez dû le voir sur les images, le bord de la sphère souffre du phénomène de crénelage [2] (aliasing) dû à la nature discrète de notre image (*cf.* Figure 6(a)).

Une solution pour atténuer ce problème est de lancer plusieurs rayons par pixel et de faire la moyenne des couleurs obtenues (*supersampling*). Cela produit ainsi une sorte de flou sur les bords des objets et atténue l'aliasing. Dans cet exercice, nous allons échantillonner (pseudo-)aléatoirement notre pixel pour calculer les différentes directions des rayons. Cette méthode n'est pas la meilleure mais elle est suffisante dans notre contexte et a pour avantage d'être rapide à coder. Rien ne vous empêche de tester d'autres méthodes plus tard (*e.g. Stratified sampling*) !

1. Modifiez la méthode `Renderer::renderImage` pour lancer `_nbPixelSamples` rayons par pixel. Le nombre d'échantillons par pixel peut être modifié dans le `main` via la méthode `Renderer::setNbPixelsSamples`.



(a) Pas d'antialiasing, un rayon au centre de chaque pixel

(b) 4 rayons par pixel

(c) 8 rayons par pixel

(d) 32 rayons par pixel

FIGURE 6 – Antialiasing par échantillonage aléatoire.

2. https://fr.wikipedia.org/wiki/Cr%C3%A9nelage