Raytracing - TD 01

Introduction au Raytracing

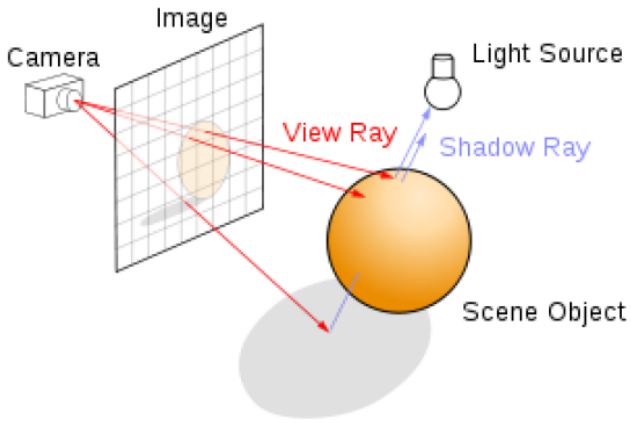
L'objectif de ce TD est de comprendre les fondamentaux du lancer de rayon (i.e. Raytracing) et de développer une petite libraire mathématique qui sera utilisé pour écrire un raytracer.

Introduction

De manière générale, le lancer de rayon est une technique qui consiste à suivre les rayons de lumière qui se propagent dans la scène et qui arrivent sur la caméra.
Ce sont ces rayons de lumière qui transportent la couleur des objets vus par la caméra.

Il existe différents algorithmes de lancer de rayon :
- backward raytracing
- forward raytracing
- bidirectional raytracing

Dans cette série de TDs, nous implémenterons le backward raytracing.



Le principe du backward raytracing est de retracer le chemin de la lumière depuis la caméra. Pour chaque pixel de l'image à rendre, un (ou plusieurs) rayons sont lancés jusqu'à intersecter un objet de la scène. Lorsqu'une intersection est détectée, le rayon rebondit dans une autre direction, qui dépend des propriétés données à la surface de l'objet.

A chaque rebond, le rayon enregistre la quantité et la couleur de la lumière retransmise par la surface au point d'intersection. Le rayon s'arrête après un nombre maximal de rebonds.
(Où bien entendu, si il entre en contact direct avec une source de lumière dans la scène)

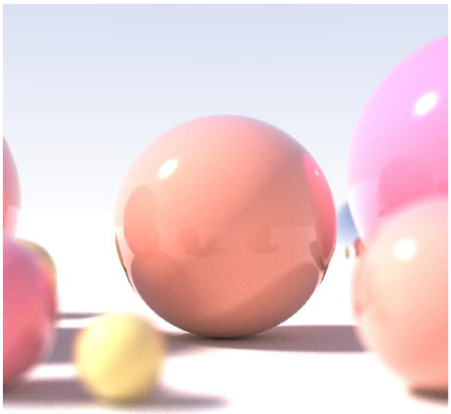
Le lancer de rayon est un algorithme récursif : on lance un rayon primaire à travers chaque pixel, et ces rayons primaires provoquent des lancer de rayon additionnels (e.x. shadow rays) lorsqu'ils entrent en contact avec des objets dans la scène.
Les couleurs seront accumulées et transportées le long de rayons se pliant aux phénomènes de réfraction et réflexion, ainsi qu'au propriétés de surface des matériaux rencontrés.

Les algorithmes de raytracing sont souvent mis en opposition avec que les techniques dites de Rasterisation, mises à disposition via les APIs de rendu OpenGL, DirectX ou Vulkan.

Les méthodes de Rasterisation sont utilisées dans le but de calculer des images en temps réel, en tirant parti de l'architecture des cartes graphiques pour calculer en parallèle les couleurs des pixels à afficher. Obtenir un résultat proche d'un rendu réaliste demandera cependant la mise en œuvre de solutions de contournement souvent approximatives. (e.x. shadow maps)
La rasterisation est principalement utilisée pour des applications en 3D interactives, incluant notamment les jeux vidéos, les outils de conceptions et aussi certaines applications web.

Les techniques de lancer de rayon sont plus coûteuses en temps de calcul et s'avèrent délicates à mettre en œuvre en parallèle. Par conséquent, leur usage à jusqu'ici été réservé au rendu d'images pré calculées, telles que les frames d'un film d'animation ou des images fixes.

La contrepartie de ce coût est qu'il permet le rendu d'images photo réalistes en se basant sur les lois de la physique. Voici des exemples d'images pouvant être obtenues via le raytracing (et un temps de calcul suffisamment long) :



Notice – Organisation d'un répertoire de TD

Un répertoire de TD sera divisé en quatre sous-répertoires :
- *bin/* : les exécutables crées via **make** seront placés ici
- *doc/* : contient divers fichiers de référence, notamment *minimal.c* et l'énoncé du TD
- *include/* : contient les fichiers d'en tête de votre programme
- *src/ :* contient les fichiers sources du raytracer et l'implémentation de vos fonctions
- *obj/* : contient les fichiers **.o* générés lors de la compilation

Vous pouvez en principe effectuer cette série de TDs au sein d'un unique répertoire de TD, cependant je vous conseille de continuer à créer un nouveau répertoire pour chaque TD. (Faites une copie du répertoire du précédent TD, vous devrez continuer sur cette base)

A faire :

01. Créez le fichier src/main.c , et copiez y le code suivant :

```
#include <stdlib.h>
#include <stdio.h>

int main (int argc, char** argv)
{
    return EXIT_SUCCESS;
}
```

Exercice 01 – Vecteurs et Points

Pour développer votre raytracer, vous aurez besoin de recourir à des objets mathématiques, notamment des points et des vecteurs.
Vous devrez créer les structures et fonction associées afin de manipuler ces entités.

A faire :

01. Créez le fichier */include/geometry.h* .
Pensez à ajouter les lignes nécessaires pour que ce fichier ne soit importé qu'une seule fois.

02a. Créez une structure `Vec3f` contenant trois champs de type `float` appelés `x` , `y` et `z` .

02b. Déclarez deux nouveaux types de variables : `Point3D` et `Vector3D` ,
en tant qu'alias du type `struct Vec3f` .
Ces alias serviront à différencier les points et les vecteurs, et à éviter les ambiguïtés.

03. Toujours dans le fichier *geometry.h* , ajoutez les prototypes des fonctions suivantes :

```
// Construit le point (x, y, z)
Point3D createPoint(float x, float y, float z);

// Construit le vecteur (x, y, z)
Vector3D createVector(float x, float y, float z);

// Construit le vecteur reliant les points P1 et P2
Vector3D createVectorFromPoints(Point3D p1, Point3D p2);

// Construit le point P + V (i.e. translation de P par V)
Point3D pointPlusVector(Point3D p, Vector3D v);

// Addition et soustraction des vecteurs V1 et V2
Vector3D addVectors(Vector3D v1, Vector3D v2);
Vector3D subVectors(Vector3D v1, Vector3D v2);

// Multiplication et division d'un vecteur V par un scalaire a
Vector3D multVector(Vector3D v, float a);
Vector3D divVector(Vector3D v, float a);

// Produit scalaire des vecteurs V1 et V2
float dot(Vector3D a, Vector3D b);

// Norme d'un vecteur V
float norm(Vector3D v);

// Construit le vecteur normalisé du vecteur V
Vector3D normalize(Vector3D v);
```

04. Créez le fichier */src/geometry.c* , et implémentez y les fonctions déclarées ci dessus.

05. Rappelez ce qu'est un produit scalaire entre deux vecteurs.
(i.e. ce qu'il représente d'un point de vue mathématique)

Exercice 02 – Couleurs

Les rayons représentés dans notre raytracer devront transporter et accumuler de la couleur.
Vous devrez créer les structures et fonction associées afin de manipuler des couleurs.

A faire :

01. Créez le fichier */include/colors.h* .
Pensez à ajouter les lignes nécessaires pour que ce fichier ne soit importé qu'une seule fois.

02a. Créez une structure `col3f` contenant trois champs de type `float` appelés `r` , `g` et `b` .

02b. Déclarez le type de variable `ColorRGB` , en tant qu'alias du type `struct col3f` .

03. Toujours dans le fichier colors.*h* , ajoutez les prototypes des fonctions suivantes :

```
// Construit la couleur (r, g, b)
ColorRGB createColor(float r, float g, float b);

// Addition, soustraction et multiplication des couleurs C1 et C2
ColorRGB addColors(ColorRGB c1, ColorRGB c2);
ColorRGB subColors(ColorRGB c1, ColorRGB c2);
ColorRGB multColors(ColorRGB c1, ColorRGB c2);

// Multiplication et division d une couleur C par un scalaire a
ColorRGB multColor(ColorRGB c, float a);
ColorRGB divColor(ColorRGB c, float a);
```

04. Créez le fichier */src/colors.c* , et implémentez y les fonctions déclarées ci dessus.

Exercice 03 – Tests pour les points et vecteurs

Étant donné que les point et vecteurs vont être utilisés au long de cette série de TDs,
il convient de tester les fonctions associées avant de continuer.

A faire :

01. Dans geometry.h , créez les fonctions pour afficher le contenu d'un point où d'un vecteur :

```
void printPoint3D(Point3D p)
void printVector3D(Vector3D v)
```

02. Dans le fichier main.c , utilisez les fonctions définies dans les exercices 01 et 02
pour vérifier les égalités suivantes :

`pointPlusVector` :	(0.0, 0.0, 0.0) + (1.0, 2.0, 0.0)	= (1.0, 2.0, 0.0)
`addVectors` :	(0.5, 1.0, -2.0) + (0.2, -1.0, 0.0)	= (0.7, 0.0, -2.0)
`subVectors` :	(0.5, 1.0, -2.0) – (0.2, -1.0, 0.0)	= (0.3, 2.0, -2.0)
`multVector` :	(0.5, 1.0, -2.0) * 2.0	= (1.0, 1.0, -4.0)
`multVector` :	(0.5, 1.0, -2.0) * 0.0	= (0.0, 0.0, 0.0)
`divVector` :	(0.5, 1.0, -2.0) / 2.0	= (0.25, 0.5, -1.0)
`divVector` :	(0.5, 1.0, -2.0) / 0.0	= ?
`dot` :	(1,0, 0.0, 0.0) . (2.0, 0.0, 0.0)	= 2.0
`dot` :	(1,0, 0.0, 0.0) . (0.0, 1.0, 0.0)	= 0.0
`norm` :	‖(2.0, 0.0, 0.0)‖	= 2.0
`norm` :	‖(1.0, 1.0, 1.0)‖	= 1.732051
`normalize` :	(1.0, 1.0, 1.0)	→ (0.57735, 0.57735, 0.57735)
`normalize` :	(0.0, 0.0, 0.0)	→ ?

Exercice 04 – Intersections (A faire à la maison)

A faire :

01. Trouvez l'équation d'intersection entre un rayon et une sphère

02. Écrivez le pseudo code pour résoudre cette équation, sans l'implémenter formellement (Cela sera à faire dans le prochain TP)

Notes :

Un rayon se définit par un point O (origine) et un vecteur D (direction).
Une sphère se définit par un point S (centre) et un rayon r

Vous aurez besoin de connaître les équations paramétriques de ces objets.
En combinant ces dernières, vous devez aboutir à une équation du second ordre.