Université de Strasbourg
Faculté de mathématiques et d'informatique

Printemps 2021

Géométrie pour la 3D

Projet de lancer de rayons

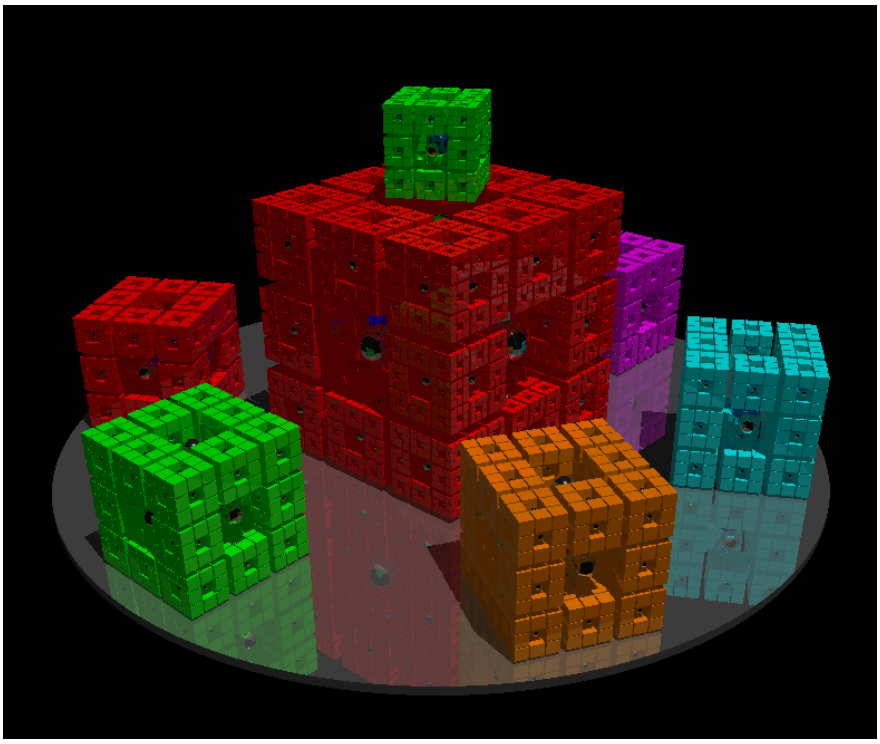


Figure 1: Rendu par lancer de rayons affichant un plateau sur lequel est posé plusieurs éponges de Menger. Les éponges de Menger contiennent des sphères très réfléchissantes en leur centre. Le plateau est également partiellement réfléchissant.

1 Introduction

Vous allez programmer un **rendu par lancer de rayons**. Le rendu par lancer de rayons est une méthode de rendu dont les images produites peuvent atteindre des degrés de photo-réalisme irréalisable via les techniques de rendu traditionnelles, au prix d'un temps de calcul drastiquement augmenté.
Dans cette technique de rendu, chaque pixel de l'écran correspond à un ou plusieurs rayons qui sont projetés à partir du foyer (ou point focal) de la caméra et qui passent par la surface occupée par le pixel dans la scène virtuelle (dans notre cas, le rayon passe uniquement par le centre de cette surface). Chaque rayon lancé peut intercepter un objet de la scène. Jusqu'à trois rayons sont alors créés (on parle de rebond) à partir du point d'intersection entre le rayon principal et l'objet : l'un va vers la source lumineuse, l'autre rebondit sur la surface par rapport à la normale de l'objet si celui-ci est spéculaire, et le dernier passe à travers l'objet si celui-ci est transparent. L'algorithme finit après un nombre de rebonds arbitraires; la couleur des objets touchés jusque-là est alors moyennée en fonction de la propriété de l'objet pour obtenir la couleur finale du pixel. Le rayon allant vers la source lumineuse est celui qui détermine la couleur de l'objet. Plus l'angle entre la direction de la lumière et la normale à la surface est proche de 180 degrés, plus le matériau est éclairé, à condition que ce rayon ne soit pas obstrué par un objet opaque, auquel cas l'objet crée un effet d'ombre.

2 Modalités

Ce projet est **à réaliser seul**. Vous devrez le déposer sur moodle dans le dépot "projet_ray-tracing" au plus tard le **vendredi 7 mai 2021 à 18h**. Merci de rendre votre travail dans **une archive au format**

TPx_prenom_nom.zip (ou autre format d'archive) avec x le numéro de votre groupe de TP.

Le travail sera principalement noté sur les points donnés dans la section suivante, mais une partie des points sera consacré à votre force de proposition pour proposer quelque chose en plus. Vous pouvez par exemple ajouter la gestion d'autres primitives (tétraèdres réguliers ...) ou encore implémenter un autre type de fractale. La qualité du code entre dans les critères d'évaluation.

Nous vous demandons également de fournir **un bref rapport (max 5 pages)** nous exposant vos choix d'implémentation et toutes autres explications que vous jugerez pertinente.

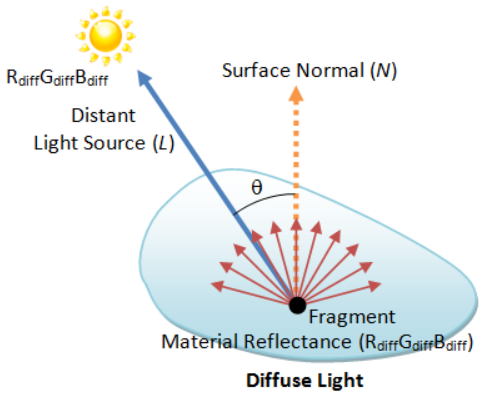
3 Travail

Vous utiliserez le même environnement de développement que pendant les séances de TP. Vous disposez d'un projet Qt de départ contenant une base de code. Lisez bien attentivement ce code et tous les commentaires (surtout les fichiers raytracing.h et raytracing.cpp) !

Aide : Un rayon est caractérisé par son **origine** et sa **direction**. Lorsque vous trouvez une intersection avec une primitive, vous allez stocker dans le champ **a_min** de la structure **Inter**, la distance entre l'origine du rayon et le point d'intersection minimum d'un objet le long du rayon. Le point d'intersection p avec cet objet le long du rayon sera lui calculé avec $p = o + dt$ avec o l'origine du rayon, d sa direction et t la distance d'intersection.

Vous devez compléter les classes suivantes du fichier raytracing.cpp :

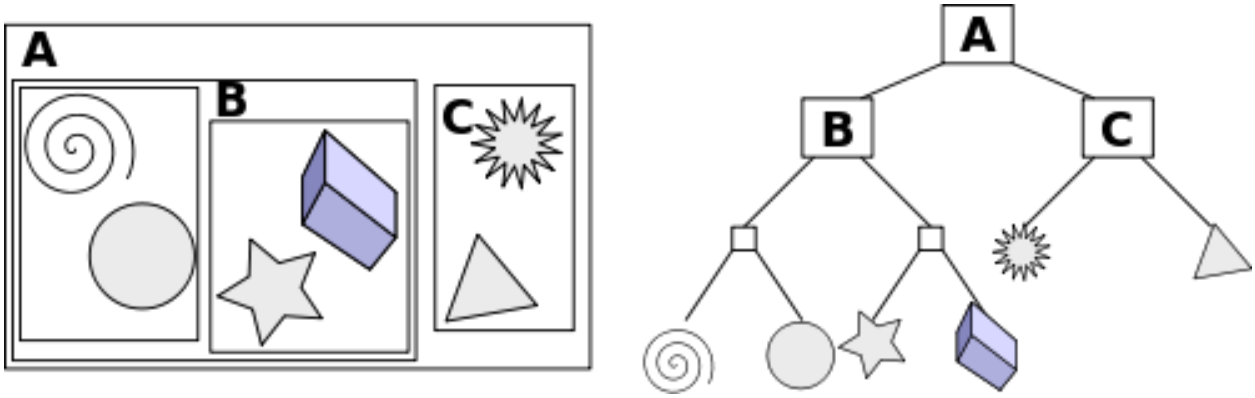
- Les classes ***Cube***, ***Sphere***, et ***Cylindre*** qui héritent de la classe ***Node*** qui représente un objet de la scène.
  Ici vous devrez implémenter les fonctions permettant de :
  1. calculer l'**intersection** entre un rayon et chacune de ces trois primitives.
  2. calculer la **normale** à la surface de l'objet au point d'intersection.
- La classe ***RTracer*** qui permet de définir et d'afficher la scène.
  La gestion des rayons et le calcul de l'image final sont déjà implémentés. Nous vous demandons ici de terminer la fonction ***ColorRayBVH()*** qui permet de déterminer la couleur d'un pixel selon l'objet le plus proche intersecté par le rayon correspondant.
  Pour simplifier, les calculs de **spécularité**, de **reflection** et de **transparence** sont déjà implémenté. Il ne vous reste plus qu'à ajouter le calcul de **la diffusion de la couleur de l'objet** au point d'intersection. Celà se fait selon l'équation suivante : $col * coeff * max(0, N \cdot L)$ avec N la normal à la surface de l'objet au point d'intersection, L la direction de la lumière, $coeff$ le coefficient de la puissance lumineuse et col, la couleur de l'objet (pour les 2 derniers, à vous de les choisir, faites des tests).
  Remarque : la position de la source lumineuse est stockée dans l'attribut **posLum** de la classe RTracer.



Ensuite, on vous demande d'ajouter à cela, la gestion d'un coefficient d'ombre sha (allant de 0.0 à 1.0). Pour cela, vous chercherez si il y a une intersection entre une primitive dans la scène et le rayon ayant pour origine, le point pour lequel vous cherchez à définir la couleur, et qui va en direction de la source lumineuse. On modifiera alors l'équation précédente pour obtenir : $(1.0 - sha) * col * coeff * max(0, N \cdot L)$

- La classe ***BVH*** (Bounding volume hierarchy), qui sert à améliorer les performances du rendu en regroupant des objets de la scène.
  Le BVH vous permettra de **décomposer votre scène en un arbre**. Ainsi, au lieu de vérifier des intersections pour les n objets d'une scène, vous vérifiez d'abord si le rayon touche la boite englobant toute la scène (la racine), puis la boite englobante de ses fils etc. Lorsque vous êtes sur une feuille, il faut alors tester l'intersection avec les primitives quelle contient. Si bien adapté à votre scène, un bon BVH peut réduire la complexité de la scène de $\theta(n)$ jusqu'à $\theta(log(n))$, réduisant ainsi les temps de calcul.
  Ici il ne vous reste plus qu'a implémenter le parcours de l'arbre avec les calculs d'intersection. Il y a deux fonctions presque similaires à implémenter (vous trouverez beaucoup plus d'info dans les commentaires du code) :

  1. **closestIntersection()**
  2. **intersecteShadow()**



De plus, il est demandé d'**implémenter une fractale appelée l'éponge de Menger** et de l'afficher dans la scène. Il y a beaucoup de façons plus ou moins harmonieuses de l'intégrer dans le code : vous avez le champ libre.

Vous proposerez une scène contenant la géométrie de votre choix (mais au moins une éponge de menger) permettant de mettre en avant toutes les fonctionnalités de votre projet.

Les items héritant de "Node" peuvent être affichés grâce à la classe "Primitives" vue en TP. Vous disposez de deux types de rendu : un rendu temps réel, qui permet d'afficher la scène et de bouger la caméra, et le rendu par lancer de rayon (non temps réel), qui est lancé grâce aux touches 0, 1, 2, 3, et 4 correspondant au nombre de rebonds (attention, bien que parallélisé ici avec la librairie OpenMP, le temps de génération d'une image peut être long, quelques secondes ou quelques minutes ...).