Travaux Dirigés d'Infographie n°5

Cours de Synthèse d'Images I

—IMAC première année—

Lancer de Rayon 1

Nous allons enfin passer à la 3D ! Pour se faire nous utiliserons un algorithme appelé le lancer de rayon qui a l'avantage d'être très intuitif et facile à implémenter. Nous n'aurons pas besoin d'OpenGL pour réaliser ce TD. Tous les calculs se feront sur CPU et l'affichage sera fait grace à la SDL.

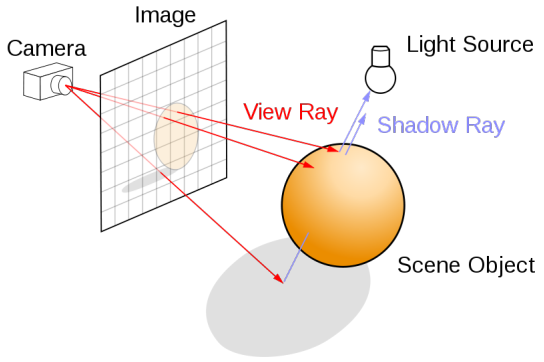
▸ Exercice 1. Un nouveau template, une nouvelle boucle d'affichage !

Téléchargez sur le site web de votre enseignant l'archive contenant le template du TD. Le répertoire a la structure suivante :

- `include` Contient les fichiers .h
  - `sdl_tools.h` Ce header contient les déclarations de fonctions utilitaires relatives à la SDL
  - `raytracer` Vous placerez dans ce répertoire les headers concernant le raytracer (voir plus loin)
- `src` Contient les fichiers .c
  - `main.c` Ce fichier contient la fonction main, c'est le point d'entrée du programme
  - `sdl_tools.c` Ce fichier contient les définitions de fonctions utilitaires relatives à la SDL
  - `raytracer` Vous placerez dans ce répertoire les fichiers source concernant le raytracer (voir plus loin)
- `Makefile` Ce makefile compile l'ensemble des fichiers sources du répertoire `src` et génère l'éxecutable `bin/raytracer`

☞ Ouvrez le fichier `main.c` et examinez la fonction `main`.
☞ Quelles sont les différences avec le template OpenGL ? Pourquoi ces différences ?
☞ Compilez et executez le programme. Que fait t-il ?
☞ Modifiez le code de dessin pour dessiner un carré au centre de la fenêtre contenant un dégradé du blanc vers noir.

▸ Exercice 2. Le principe du raytracing



De manière générale le raytracing consiste à suivre les rayons de lumières se propageant dans la scène et arrivant sur la caméra. Ce sont eux qui transporte la couleur des objets. Il existe plusieurs algorithmes de raytracing : backward raytracing, forward raytracing, bidirectional raytracing, etc. Nous allons implanter le plus simple : le backward raytracing, qui est illustré par l'image ci-dessus.

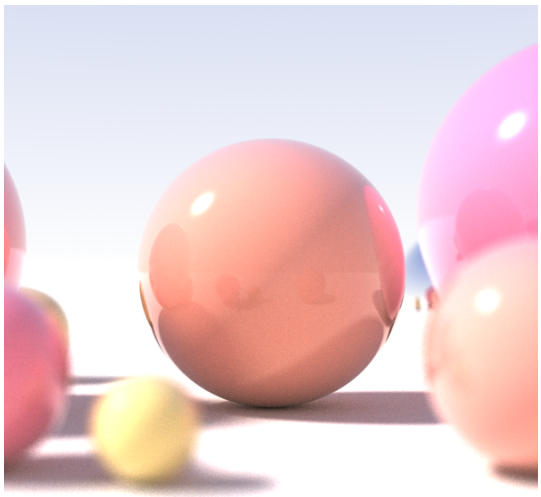
Cet algorithme à pour particularité de partir de la caméra pour remonter vers les sources de lumière. Cela peut paraitre contre intuitif : pourquoi partir de la fin du processus (l'arrivée de la lumière sur l'oeil) ? Tout simplement car partir des sources de lumière conduirait à traiter un grand nombre de rayons pour rien : parmis l'infinité de rayons envoyés par une lumière, une très faible partie (infinie également néammoins) arrive sur l'oeil. Si on part de l'oeil on est certain de ne traiter que les rayons qui interviennent dans la couleur de l'image finale.

Notre caméra est donc placée dans l'espace et possède une orientation. On place virtuellement la grille de pixels consituant notre écran sur le plan lui faisant face. Chaque pixel possède alors une position dans l'espace. L'algorithme du lancer de rayon parcourt la grille de pixel et lance un rayon à travers chaque pixel. Lancer un rayon signifie calculer l'intersection entre une demi-droite et les objets de la scène. Si l'intersection n'existe pas alors le pixel garde la couleur d'arrière plan (noir souvent, mais cela peut aussi être une image de fond). Sinon elle intersecte un objet en un point P. Ce point peut être illuminé ou être dans l'ombre. Pour savoir cela il suffit de lancer de nouveaux rayons vers les sources de lumière.

Le lancer de rayon est donc un algorithme fondamentalement récursif : on lance un rayon, qui lance lui même des rayons, qui peuvent eux même lancer des rayons, ect. Les couleurs s'accumulent le long de ces rayons selon les lois de réflection / réfraction de la physique.

Le principal avantage du raytracing est qu'il est très modulaire. On peut facilement commencer par un petit raytracer qui se contente de calculer les intersections sans s'occuper de la lumière puis ensuite le faire évoluer afin d'obtenir des images plus réalistes. C'est ce que nous allons faire. Le développement de ce TD sera donc très itératif. Il est très fortement conseillé de garder un code modulaire tout au long du TD. C'est pourquoi un template structuré vous à été fourni. Les exercices vous guideront afin de construire une hiérarchie de modules adaptée.

Voici le niveau de réalisme que l'on peut obtenir avec de bon raytracers (et beaucoup de temps de calcul !) :



☞ Ecrire sur papier l'algorithme du raytracing pour dessiner l'ensemble des pixels d'une image (en pseudo-code). On suppose que l'on a une scene `scene` et que l'on peut lancer un rayon dans cette scene via la fonction `lancer_rayon(rayon, scene)` qui nous renvoit une couleur. Notre image

est nommée `framebuffer`, a pour largeur `largeur`, pour hauteur `hauteur`. Etant donné un pixel (x, y) on se donne la fonction `rayon_pixel(x, y, camera)` qui nous renvoit un rayon en 3D pour le pixel considéré par rapport à la caméra considéré.

▸ Exercice 3. Une bibliothèque de géometrie

Une première étape dans la création d'un raytracer de A à Z est l'écriture d'un module permettant de manipuler les concepts utiles de la géometrie dans l'espace.

- ☞ Ajoutez les répertoires `include/geometry` et `src/geometry` au TD. Vous y mettrez les headers et les fichiers sources qui ont trait à la géometrie.
- ☞ Implantez le concept de vecteur 3d à travers la structure `Vector3D`. Cette ci doit contenir 3 flottant x, y et z.
- ☞ Implantez le concept de point 3d à travers la structure `Point3D` (identique à la structure Vector3D).
- ☞ Implantez les fonctions mathématiques suivantes manipulant des points et des vecteurs (voir sur internet pour celles que vous ne connaissez pas) :
  - `Point3D PointXYZ(float x, float y, float z)` : Construit le point (x, y, z)
  - `Vector3D VectorXYZ(float x, float y, float z)` : Construit le vecteur (x, y, z)
  - `Vector3D Vector(Point3D A, Point3D B)` : Construit le vecteur $\vec{AB} = B - A$
  - `Point3D PointPlusVector(Point3D P, Vector3D V)` : Construit le point $P + V$
  - Des fonctions d'addition et soustraction de vecteurs `AddVectors` et `SubVectors`
  - Des fonctions de multiplication et division d'un vecteur par un scalaire `MultVector` et `DivVector`
  - La fonction `DotProduct` calculant le produit scalaire de deux vecteurs.
  - La fonction `Norm` calculant la norme d'un vecteur.
  - La fonction `Normalize` calculant le normalisé d'un vecteur.
- ☞ Implantez le concept de rayon avec la structure `Ray3D`. Un rayon est défini par un point d'origine et un vecteur direction.
- ☞ Implantez le concept de couleur avec la structure `Color3f` contenant des champs flottant r, g et b. A noter que nous utiliserons des flottants pour représenter les couleurs car ce type est bien plus flexible pour effectuer des calculs que le type `unsigned char` utilisé par la SDL. Une composante valant 0 indiquera le noir et une composante valant 1 indiquera le blanc. Il faudra convertir les valeur flottantes en `unsigned char` avant de les passer à la SDL avec la fonction `SDL_MapRGB`. La formule de conversion est $255 * max(0, min(1, c))$.
- ☞ Implantez les fonctions suivantes manipulant des couleurs :
  - Des fonctions d'addition (`AddColors`), soustraction (`SubColors`), multiplication (`MultColors`) et division (`DivColors`) de couleurs.
  - Des fonctions de multiplication et division d'une couleur par un scalaire `MultColor` et `DivColor`
- ☞ Implantez le concept de sphère avec la structure `Sphere`. Une sphère est définie par un centre, un rayon et une couleur.
- ☞ Implantez le concept de cube avec la structure `Cube`. Un cube est défini par 2 points représentant son segment diagonal et sa couleur.

▸ Exercice 4. Calcul d'intersections

Il va encore falloir écrire quelque fonctions mathématiques avant de pouvoir passer au code de rendu. L'entitée de base d'un raytracer est le rayon, défini par sa position et sa direction. Un rayon sera « envoyé »dans la scène afin de tester s'il intersecte un objet. « Envoyer »un rayon signifie concrètement appeler une fonction (que vous coderez par la suite) qui boucle sur les objets de la scène. Elle retient l'intersection la plus proche trouvée (et la couleur de l'objet concerné !). On doit donc pouvoir tester l'intersection entre un rayon et chaque type d'objet que l'on veut pouvoir mettre dans notre scène.

☞ Ecrivez la structure `Intersection` contenant un champs pour la position et un champs pour la couleur.

☞ Implantez la fonction `int TestRaySphereIntersection(Ray3D ray, Sphere sphere, Intersection* intersection)` qui teste l'intersection entre `ray` et `sphere` (trouvez l'algo par vous même ou sur internet !). Elle renvoit 1 s'il y a intersection, 0 sinon. Si `intersection` est non `NULL`, elle place la position de l'intersection et la couleur de l'objet à l'adresse `intersection`.

☞ Même exercice pour la fonction `int TestRayCubeIntersection(Ray3D ray, Cube cube, Intersection* intersection)`.

▸ Exercice 5. La scène

On appelle « scène »l'ensemble des objets que l'on désire potentiellement voir apparaitre à l'écran (en fonction de la position de la caméra). Pour réprésenter une scène nous allons utiliser une structure `Scene` que nous placerons dans le module `raytracer`. Il faut construire cette structure de manière à pouvoir stocker des objets de différents types. Plusieurs solutions s'offrent à nous :

- Avoir un conteneur (tableau) par type d'objet. Cette solution n'est pas très souple car elle nous oblige à modifier la structure Scene à chaque fois que l'on veut ajouter un type d'objet.
- Ecrire une union `Shape` permettant de représenter tous nos types sous un même type. D'un point de vue modularité, cette solution est plus adaptée que la précédente car elle isole dans une structure dediée les changements relatifs à l'ajout d'un nouveau type d'objet.
- Ecrire une structure abstraite `Shape` contenant un pointeur sur un objet dont le type est inconnu et des pointeurs de fonctions pouvant manipuler cet objet. Cette solution est la meilleure pour la modularité du code. L'ajout d'un nouveau type correspond juste à la définition d'une nouvelle structure et des fonctions associée (ce qui doit aussi être fait pour les deux autres méthodes).

Nous allons choisir la deuxième solution qui constitue un bon compromis entre modularité et simplicité d'implantation (la troisième solution n'est pas très difficile à implanter mais il faut avoir vu les pointeurs de fonction ! Vous être libre de la coder si vous vous en sentez capable. Le temps de développement est approximativement le même que celui de la deuxième méthode).

Une union est un type dont la taille en octet est le maximum de la taille de chacun de ses champs. Par exemple si on a :

```
typedef union {
  int a;
  short b;
  char c;
} SimpleUnion;
```

Alors la relation suivante est vérifiée :
`sizeof(SimpleUnion) = max(sizeof(int), sizeof(short), sizeof(c))`.

Une union ne peut donc contenir qu'un seul de ses champs à la fois car l'espace mémoire allouée est assez grand pour le plus gros uniquement. Ainsi, à un instant donné, une variable de type `SimpleUnion` sera soit un `int`, soit un `short`, soit un `char`, à vous de choisir !

Les unions sont souvent utilisées lorsqu'il sagit d'avoir un type général possédant plusieurs sous-types. Pour ceux qui connaissent la programmation orientée objet, c'est un genre d'interface. L'exemple le plus fréquemment rencontré est celui des types évenements. On a souvent une union `Evenement` possédant un champs pour chaque type d'évenement possible.

Ici nous avons typiquement le cas de figure d'un type général : un objet peut soit être un cercle, soit un cube, soit autre chose qui n'a pas encore été défini mais que nous voudrons peut être ajouter par la suite (cone, cylindre, tore, triangle, etc.). Nous allons appeler notre union `Shape` (« Forme »en anglais). Si on ne connait pas bien les unions, on aurait tendance à vouloir faire ça :

```
typedef union {
  Sphere sphere;
  Cube cube;
} Shape;
```

Mais ce n'est pas la bonne solution ! En effet, comment savoir qu'elle forme est réellement contenue dans une variable de type `Shape` ? Vous pourriez dire : « On le sait car c'est nous qui choisissons ». Mais parfois on ne choisit pas. Par exemple si on veut écrire une fonction générique d'intersection rayon - forme, celle ci doit prendre une forme en paramètre :

```
int TestRayShapeIntersection(Ray ray, Shape shape, Intersection* intersection) {
  if(/* shape est une sphere */) {
    return TestRaySphereIntersection(ray, shape.sphere, intersection);
  } else if(/* shape est un cube */) {
    return TestRayCubeIntersection(ray, shape.cube, intersection);
  }
  /* Forme inconnue ? */
  return 0;
}
```

L'algorithme est correctement implémenté, mais que mettre à la place des commentaires dans les `if` ? On est incapable de déterminer le vrai type de la forme passée en paramètre. Une fois ça sera peut être une sphère, une autre fois un cube, une autre fois autre chose...

Il existe une technique couramment utilisée pour palier à ce problème. Toutes les unions suivent ce pattern :

```
typedef enum {
  SPHERE_SHAPE, CUBE_SHAPE
} ShapeType;

typedef union {
  ShapeType type;
  Sphere sphere;
  Cube cube;
} Shape;

typedef stuct {
  ShapeType type; /* Doit être SPHERE_SHAPE pour toutes les sphère,
    à fixer dans la fonction qui construit une sphere */
  [...]
} Sphere;

typedef stuct {
  ShapeType type; /* Doit être CUBE_SHAPE pour tous les cubes,
    à fixer dans la fonction qui construit un cube */
  [...]
} Cube;
```

On créé un nouveau type énuméré contenant une constante pour chaque type de forme possible. On ajoute à l'union ainsi qu'a toutes les structure de forme un champs type. Toutes les variable d'un même type auront la même valeur pour ce champs. Pourquoi cela marche t-il ? nous avons dit qu'une union à pour taille le maximum de ses champs. Son premier champs est un entier (les enums sont des entiers), usuellement de taille 4 octets. L'union fera donc 4 octets si aucun de ses autres champs n'est plus gros que 4 octets. Or chaque champs (sphere et cube ici) contient également un champs `type` en début de structure, qui fait donc 4 octets. Donc la taille de chacun des champs de l'union est forcément supérieur ou égale à 4. Il en résulte que les 4 premiers octets de l'union contiendront une valeur de type `ShapeType`, et ce quelque soit le type réel de l'union ! Ce champs `type` nous permet de décider quel champs utiliser vraiment dans l'union. Ca marche exactement comme pour les `SDL_Event`. Ma fonction d'intersection générique devient alors :

```
int TestRayShapeIntersection(Ray ray, Shape shape, Intersection* intersection) {
  /* Avec un switch c'est encore mieux ! */
  if(shape.type == SPHERE_SHAPE) {
    return TestRaySphereIntersection(ray, shape.sphere, intersection);
  } else if(shape.type == CUBE_SHAPE) {
    return TestRayCubeIntersection(ray, shape.cube, intersection);
  }
  /* Forme inconnue ? */
  return 0;
}
```

Maintenant, si je veux ajouter à mon raytracer une nouvelle forme, `Cone` par exemple, je dois suivre la procédure suivante :

- Ajouter la valeur `CONE_SHAPE` à l'enum `ShapeType`
- Ecrire la structure `Cone` avec pour premier champs le type de l'objet.
- Ecrire la fonction d'intersection relative au cone.
- Modifiez la fonction d'intersection générique afin d'ajouter un cas pour le cone.

A noter qu'il n'y aurait que l'étape 2 et 3 pour une solution à base de pointeurs de fonction, aucun code déja écrit à modifier !

☞ Ajouter dans le module de géometrie l'enumération `ShapeType` et l'union `Shape`.
☞ Modifiez vos structure `Sphere` et `Cube` pour qu'elles respectent le format décrit plus haut.
☞ Ecrivez la fonction `int TestRayShapeIntersection(Ray ray, Shape shape, Intersection* intersection)`

Nous sommes à présent près à écrire la structure `Scene` (dans le module `raytracer`)

☞ Ecrivez la structure `Scene`. Elle devra contenir un tableau (statique pour le moment) de `Shape` et le nombre de formes contenues dans la scène.
☞ Ecrivez la fonction `void AddSceneShape(Scene* scene, Shape shape)`. Elle doit ajouter la forme passée en paramètre à la scène pointée.
☞ Ecrivez la fonction `int ThrowRayOnScene(Ray ray, const Scene* scene, Intersection* intersection)` qui boucle sur tous les objets d'une scène afin de trouver l'intersection la plus proche avec le rayon passé en paramètre (la fonction renvoit 0 s'il n'y a pas d'intersection, sinon 1).

▸ Exercice 6. Premier raytracing simple

Nous avons enfin toutes les briques pour écrire une ébauche de notre algorithme de raytracing !

Afin de réaliser cela, il faut d'abord déterminer une position pour notre caméra. Nous allons pour l'instant placer notre caméra sur l'origine de la scene $(0, 0, 0)$, celle ci regardant dans la direction

$(0, 0, -1)$ (axe des Z, coté négatif). Le dessin se fera sur le plan $z = -1$, dans un carré de taille 2 centré en $(0, 0, -1)$. Cela implique donc que les coordonnées des coins du carré sont : $(-1, -1, -1)$, $(1, -1, -1)$, $(1, 1, -1)$, $(-1, 1, -1)$. Notre framebuffer sera donc plaqué virtuellement dans ce carré. Chaque pixel possède alors une position 3D situé dans le carré. Les rayons seront lancé à travers les pixels.

Soit (i, j) un pixel du framebuffer de dimensions $(largeur, hauteur)$. Les coordonnées de scène de ce pixel sont : $x = -1 + \frac{2i}{largeur}$, $y = 1 - \frac{2j}{hauteur}$, $z = -1$. Le rayon $R(i, j)$ passant par ce pixel à pour origine $(0, 0, 0)$ (le centre optique de la caméra) et pour direction (x, y, z).

☞ Ecrire la fonction `void SimpleRaytracing(const Scene* scene, SDL_Surface* framebuffer)`. Cette fonction itère sur chacun des pixels du framebuffer (les dimensions du framebuffer sont stockés dans `framebuffer->w` et `framebuffer->h`). Pour chaque pixel (i, j) elle calcule $R(i, j)$ selon la formule donnée précédemment puis envoie ce rayon dans la scène. Si une intersection est trouvée elle place la couleur du point d'intersection à la position (i, j) du framebuffer (avec la fonction `PutPixel` de `sdl_utils.h`). Sinon elle ne touche pas au pixel. Vous placerez cette fonction dans le module `raytracer`.

☞ Modifiez la fonction `main` pour placer à l'endroit du code de dessin un appel à cette fonction (vous aurez donc pris soin de déclarer une scène au début du main).

☞ Ajoutez à la scène une sphère à une position visible depuis la caméra (par exemple la sphère en $(0, 0, -5)$ avec pour rayon $r = 1$). Si vous avez tout bien codé, vous devriez voir apparaitre un disque de la couleur de la sphère.

Remarque : Il y a de fortes chance pour que cela ne marche pas du premier coup (en synthèse d'images rien de marche jamais du premier coup...). Ne vous découragez pas pour autant, il faut trouver d'ou provient le bug ! Pour cela afficher les position des objets, les directions des rayons, les résultat des tests d'intersection, etc.

A l'issue de cette première partie vous avez normalement un premier raytracer très simple. Il est néammoins très limité puisque :

- La caméra à une position et un champs de vision fixe.
- Il ne gère aucune lumière, et donc aucune ombre. Le dessin n'est donc pas du tout réaliste, on voit juste la projection des formes apparaitre à l'écran.

Dans le prochain sujet de TD nous verrons la gestion des transformations pour pouvoir modifier la caméra et la position des objets de manière souple. Nous étudierons également des modèles d'illumination afin d'obtenir un résultat plus crédible visuellement.