



R my Malgouyres,  
Algorithmes pour la synth se d'images  
et l'animation 3D,  
DUNOD, 2002

## TP n  14 : Projet lancer de rayon

### Objectifs :

Le but de ce TP est de programmer l'algorithme du lancer de rayons pour une sc ne 3D. La sc ne 3D est une classe d riv e de celle sur laquelle les TP pr c dents ont  t  conduits. Le TP consiste   rajouter   ce projet la visualisation de la sc ne par lancer de rayons. Dans un premier temps, nous nous limiterons aux mod les d' clairage de la r flexion diffuse et sp culaire, puis on impl mentera les mod les de transparence et r flexion sp culaire id ale.

## 1 Les nouvelles classes

### 1.1 La classe Rayon

La classe rayon est principalement compos e de son origine et de son vecteur directeur :

```
class Rayon {
    Point3D origine; // origine du rayon
    Point3D vectdirect; // vecteur directeur

    Objet3D * objetscontenant[20]; // pour la transparence
    int nbobjetscontenant; // pour la transparence

public:
    Rayon();
    Rayon (Point3D or, Point3D vd){origine=or;vectdirect=vd;}
};
```

Un tableau de pointeurs sur des `Objet3D` contient la liste des objets contenant l'origine du rayon. Nous utiliserons ces donn es lors de l'impl mentation de la transparence.

### 1.2 La classe ResultIntersect

Un algorithme d'intersection rayon-objet doit retourner un certain nombre de donn es : position du point d'intersection, normale au point d'intersection, propri t s physiques de l'objet (couleur, coefficients de r flexion et r fraction, etc...). Certaines de ces donn es, celles qui ne peuvent pas  tre retrouv es uniquement   l'aide d'un pointeur sur l'objet, seront regroup es dans une classe : la classe `ResultIntersect`.

```

class ResultIntersect{
    double t_inter; // distance de l'intersection
    Point3D normale; // vecteur normal au point d'intersection
    Couleur couleur; // coefficients RGB de la couleur (entre 0 et 1)
};

```

### 1.3 La classe Boiteenglobante

Nous utiliserons cette classe pour afficher des poly dres. Cette classe permet de m moriser un volume, qui est une boite, et qui contient un objet complexe (poly dre). Pour d terminer si un rayon intersecte un poly dre, on teste d'abord l'intersection du rayon avec la boite englobante, puis, seulement dans le cas o  le rayon frappe la boite, on teste l'intersection avec le poly dre.

```

class Boiteenglobante{
    double xmin, xmax, ymin, ymax, zmin, zmax;
public :

    Boiteenglobante(void);
    Boiteenglobante(double xmini, double xmaxi,
                    double ymini, double ymaxi,
                    double zmini, double zmaxi);
    bool testintersect(const Rayon &ray);
    bool appartient(Point3D p);
};

```

Les donn es `xmin`, `xmax`, `ymin`, `ymax`, `zmin` et `zmax` donnent les dimensions de la boite, et la fonction `testintersect` teste l'intersection d'un rayon avec la boite englobante. Pour impl menter cette fonction `testintersect`, on doit  videmment s'inspirer de l'algorithme d'intersection rayon-boite.

### 1.4 La classe Scene3DRay

La classe `Scene3DRay` h rite de la classe `Scene3D`, et contient les donn es et m thodes relatives au lancer de rayons.

```

class Scene3DRay : public Scene3D {

protected:
    // donnees relatives au rendu par lancer de rayons :

    int dim2Drendux, dim2Drenduy; // dimension des buffers pour lancer de rayon

    int profrecurmax;
    double pourcenneglig;

public:
    Scene3DRay(void):Scene3D()

```

```

// initialisation de la scene et initialisation des cameras :
Scene3DRay(int dimx, int dimy,
           double xmin, double xmax, double ymin, double ymax, double zmin, double zmax,
           int nbcamera, double hautcam, int nblumiere,
           double lumambR, double lumambG, double lumambB,
           double fR, double fG, double fB);
//
           int nbmaxsp2D, int nbbsp2D, Spline2D **psp2D);

void destruction(void);

void calculboitesenglobantes(void);
void detruitboitesenglobantes(void);

void lancer_general(imcouleur& bitmap_dib);
bool lancer_recuratif(Rayon ray, double& IR, double& IG, double& IB,
                    int prof, int prof_max, double pourcent, double pourcent_min);
};

```

## 2 Ordonnancement des tâches

Le minimum pour pouvoir tester quelque chose est d'implémenter la fonction `lancer_general`, la recherche de l'intersection la plus proche dans les fonctions `lancer_recuratif` et `calculintersectproche`, et un calcul d'intersection. Notons que l'algorithme d'intersection rayon-cylindre de révolution est déjà implémenté, on peut donc tester l'algorithme de lancer de rayons avec des cylindres sans implémenter d'algorithme d'intersection. On peut alors visualiser la silhouette d'un cylindre de révolution ou de tout autre objet dont l'algorithme d'intersection est implémenté.

Dans le cadre d'un travail à plusieurs, on pourra ensuite réaliser en parallèle le calcul d'intersection rayon-polyèdre et les modèles d'éclairément.

## 3 Lancer les rayons issus de l'observateur

Il s'agit ici d'écrire le code de la fonction

```
void Scene3D::lancer_general(imcouleur& bitmap_dib, etc...);
```

Le corps de cette fonction comporte plusieurs étapes principales :

1. *Initialisation*. Cette étape elle même se subdivise en plusieurs parties :
  - (a) En prévision de l'affichage de polyèdres (pour l'affichage de surfaces splines, etc...), il faut ramener les coordonnées des sommets des polyèdres, ainsi que les normales à ces sommets et les sources lumineuses, dans le repère de la scène;
  - (b) On initialise les buffers avec la couleur du fond;
  - (c) Pour l'affichage de polyèdres, initialisation des boites englobantes (appel de la fonction `calculboitesenglobantes()` qui est déjà écrite).
2. *Lancer des rayons*. On doit ici lancer un rayon à travers chaque pixel comme expliqué en cours. Pour lancer un rayon, on utilisera la fonction `Scene3DRay : :lancer_recuratif`

(voir plus loin pour l'implémentation de cette fonction). Il ne faut pas oublier de ramener les rayons dans le repère de la scène avant de lancer un rayon.

3. *Copier les buffers dans le bitmap.* Cette étape est semblable à une étape de la programmation de l'algorithme du *z*-buffer.
4. *Terminer* Il faut simplement ramener les coordonnées des polyèdres dans le repère de la caméra, détruire les boîtes englobantes (fonction `detroitboitesenglobantes()`)

## 4 Lancer d'un rayon

### 4.1 Prototype de la fonction `Scene3D::lancer_recuratif`

C'est dans la fonction `Scene3D::lancer_recuratif` que nous allons appeler la fonction qui calcule l'objet le plus proche intersecté par le rayon, puis la fonction qui calcule les termes d'éclairément.

```
bool Scene3DRay::lancer_recuratif(Rayon ray ,
                                double& IR, double& IG, double& IB,
                                int prof, int prof_max,
                                double pourcent, double pourcent_min);
```

Le premier paramètre de cette fonction est le rayon à lancer. On trouve ensuite les trois coefficients RGB de l'intensité à retourner, puis les paramètres de limitation de la profondeur de la récursivité de cette fonction. En effet, la fonction `lancer_recuratif` étant récursive, il faut un critère pour arrêter la récursivité à un moment donné. On peut donner une profondeur maximale, ou un pourcentage minimal de contribution au rayon initial en dessous duquel on néglige les appels récursifs.

Lors de l'appel initial de cette fonction lors du lancer des rayons primaires, le paramètre `prof` vaut 1, le paramètre `prof_max` est égal à la donnée membre `profrecurmax` de la classe `Scene3DRay`, le paramètre `pourcent` vaut 100, et le paramètre `pourcent_min` est égal à la donnée membre `pourcenneglig` de la classe `Scene3DRay`.

Dans la fonction `lancer_recuratif` on teste d'abord si la profondeur de la récursivité est supérieur à `prof_max` ou si `pourcent` est inférieur à `pourcenneglig`, auquel cas on retourne des intensités égales à 0 sans continuer les calculs.

On calculera ensuite l'intersection.

### 4.2 Calcul d'intersection

Dans la fonction `lancer_recuratif`, on calcule l'objet le plus proche intersecté par le rayon. Pour cela, on fait appel à la fonction

```
bool Scene3DRay::calculintersectproche(
    Rayon ray,
    int &numobjetproche, // numero de l'objet intersecté
    ResultIntersect &resplusproche, // donnée couleur, normale,... de l'i
    double &min_t_inter, // distance de l'objet le plus proche
    Objet3D * &objetintersecte // adresse de l'objet le plus proche à reto
);
```

Dans cette fonction, on parcourra la liste chaînée `objets` des objets de la scène (voir la classe `Scene3D`). Pour chaque objet de la scène, on appellera la fonction virtuelle `calculintersect` de la classe `Objet3D`. On mémorisera l'objet qui est à une distance  $t$  (retournée dans le paramètre de type `ResultIntersect`) la plus proche de l'origine du rayon, comme dans un calcul de minimum.

## 5 Calculer l'intersection avec un objet

### 5.1 Les tâches à réaliser

Le calcul de l'intersection d'un rayon avec un objet dépendra de l'objet en question. C'est pourquoi, au niveau de la classe `Objet3D`, on le voit sous forme de fonction virtuelle :

```
virtual bool Objet3D::calculintersect(Rayon ray_sce, // rayon à intersecter
    ResultIntersect &resinter, // résultat de l'intersection
    bool testshadow, // calcul d'ombre ou calcul exact d'intersection
    double& attenuat, // atténuation à retourner pour l'ombre
                    // d'un objet transparent
    double distance, // distance à la source lumineuse si testshadow==true
    bool phong, bool texture));
```

L'implémentation des différents algorithmes d'intersection rayon-objet se fera donc dans les fonctions `calculintersect` des classes dérivées :

- Boite,
- Sphere,
- Cylindre, (déjà implémenté)
- Cone, (déjà implémenté)
- Polyèdres, (qui permet d'afficher les surfaces splines et polyèdres).

Pour les classes de surfaces splines, on implémentera la fonction

```
bool Objet3D::calculintersectpolyedre(Rayon ray_sce, // rayon à intersecter
    ResultIntersect &resinter, // résultat de l'intersection
    bool testshadow, // calcul d'ombre ou calcul exact d'intersection
    double& attenuat, // atténuation à retourner pour l'ombre
                    // d'un objet transparent
    double distance, // distance à la source lumineuse si testshadow==true
    bool phong, bool texture));
```

de la classe `Objet3D`, qui travaille sur le polyèdre stocké dans cette classe.

Dans un premier temps, on n'implémentera que le cas où le paramètre `testshadow` est à faux.

### 5.2 L'intersection rayon-polyèdre

On fera appel à la fonction `Scene3DRay : :calculboitesenglobantes` avant de lancer les rayons primaires, dans la fonction `lancer_general` (le code de cette fonction est déjà écrit).

Cette fonction initialise les données membres suivantes de la classe `Objet3D`

```
Boiteenglobante boundingbox; // boite englobante de l'objet
Boiteenglobante *tabboundingbox; // tableau de toutes les boites englobantes
```

```

int nbboites; // nombre de boites
int * nofacesdansboites; // numéros des facettes dans chaque boite
int * indicesfinboites; // les facettes intersectant la boite k ont des numéros
                        // nofacesdansboites[i] pour
                        // indicesfinboites[k-1] <= i < indicesfinboites[k]
int racinecubiquenbboites;

```

Ainsi que la donnée membre `boundingbox` de la classe `Facette`

Pour calculer l'intersection avec le polyèdre, on teste tout d'abord l'intersection avec la boite englobante de l'objet, puis pour chaque boite du tableau `tabboundingbox`, si le rayon intersecte la boite, on teste chaque facette intersectant la boite. Pour tester une facette, on teste d'abord la boite englobante de la facette, puis le cas échéant, on calcule l'intersection de la facette avec le rayon comme vu en cours.

## 6 Modèle d'éclairage

### 6.1 Réflexions diffuse et spéculaire

Dans la fonction `lancer_recuratif`, on fera appel à la fonction

```

void Scene3DRay::computePointLight(Objet3D* objetintersecte, // objet
    Couleur couleur, // couleur, éventuellement de texture
    bool transparent, // vrai si l'un des coefficients de transparence est non nul
    bool reflexiontotale, // vrai si réflexion totale
    Point3D p3d, // point d'intersection à illuminer
    Point3D V, // direction du rayon incident
    Point3D normale, // normale à l'objet au point p3d
    Point3D R, // direction du rayon réfléchi
    Point3D T, // direction du rayon transmis
    Couleur & I // intensité à retourner
);

```

qui calcule la contribution des sources lumineuses à la couleur d'un objet au point `p3d`.

L'implémentation des termes de réflexions, diffuse et spéculaire est très semblable au cas du `z-buffer` (on pourra, en faisant attention, faire un copier-coller puis modifier le code).

### 6.2 Ombres portées

Pour les ombres portées, on doit tester si un objet se trouve sur la trajectoire entre un point d'un objet et une source lumineuse. Pour cela, on lance un rayon d'éclairage vers la source, et on teste si un objet se trouve sur la trajectoire à une distance inférieure à la distance  $t_{limit}$  de la source lumineuse à l'origine du rayon d'éclairage.

Comme le calcul d'intersection, le test de l'intersection d'un rayon avec un objet dépendra de l'objet en question. C'est pourquoi, au niveau de la classe `Objet3D`. On utilisera à nouveau la fonction virtuelle :

```

virtual bool Objet3D::calculintersect(Rayon ray_sce, // rayon à intersecter
    ResultIntersect &resinter, // résultat de l'intersection
    bool testshadow, // calcul d'ombre ou calcul exact d'intersection
);

```

```
double & attenuat, // atténuation à retourner pour l'ombre
                // d'un objet transparent
double distance, // distance à la source lumineuse si testshadow==true
bool phong, bool texture))=0;
```

Il faudra modifier le comportement de cette dernière fonction dans le cas où le booléen `testshadow` est à vrai. Pour cela, on doit déterminer si l'objet intersecté est à une distance inférieure au paramètre `distance`. On retourne vrai uniquement dans ce cas.

En utilisant la fonction `calculintersect`, on implémentera la fonction `bool Scene3DRay : :testshadow(Rayon raylum, double distance, double & attenuation)` qui retourne vrai si le rayon `raylum` est interrompu par un objet se trouvant à une distance inférieure à au paramètre `distance` (distance à la source lumineuse).

Dans la fonction `computePointLight`, on n'ajoutera le terme dû à une source lumineuse que si la fonction `testshadow` retourne faux.

### 6.3 appels récursifs, réflexion et transmission idéale

Pour le rayon réfléchi, il suffit d'appeler récursivement la fonction `Scene3D : :lancer_récuratif`, pour un rayon calculé suivant les lois de Descartes, et de multiplier par les coefficients appropriés suivant les modèles d'éclairément.

Pour le rayon transmis, la direction du rayon transmis dépend des deux indices de réfraction avant et après l'intersection (voir le cours).

### 6.4 Textures

Le plaquage de textures s'implémente au niveau de la fonction `calculintersect`, lors du calcul d'intersection rayon-objet. Il faut déterminer les coefficients RGB de la couleur de l'objet au point d'intersection à l'aide de la texture, et stocker ces couleurs dans l'élément de la classe `ResultIntersect` retourné par la fonction `calculintersect`.