GUITTENY Fabrice
LE GOFF Erwan
LAHUEC Morgan

LANCER DE RAYON

IN020- Images numériques, synthèse et traitement Master 1 Informatique 2005

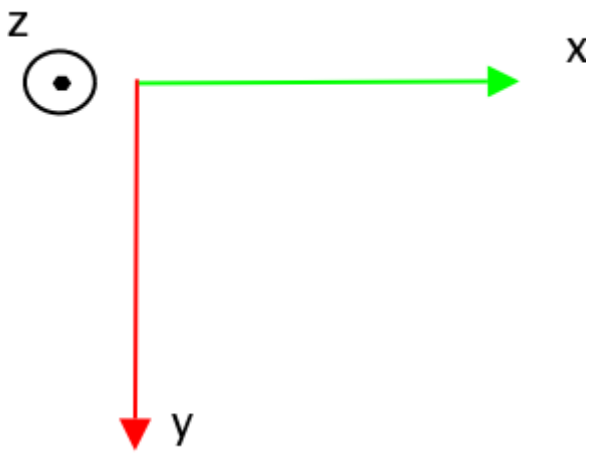
Introduction :

On souhaite implémenter en C++ un algorithme de lancer de rayon. Le lancer de rayon (ra*y tracing* en anglais) est une technique de rendu en synthèse d'image simulant le parcours inverse de la lumière de la scène vers l'observateur à travers l'écran. Cette technique simple permet de reproduire les phénomènes physiques tel que la réflexion.
Nos scènes seront composées uniquement de sphères mais celles-ci pourront être représentées de trois façons différentes. En effet, le lancer de rayon permet de définir mathématiquement les objets à représenter mais également par une multitude de facettes.

I - DESCRIPTION D'UNE SCENE:

La scène est l'élément central de notre programme. Elle contient en effet tous les données sur le nombre et la nature des sphères et des sources qui la compose. Elle fixe également la taille et la position de la fenêtre et de l'observateur. Pour pouvoir la décrire efficacement, une syntaxe spécifique au fichier de description a été choisie : le formalisme XML. Ce choix ce justifie par les nombreux avantages que représente ce langage comme la facilité d'adaptation aux données, une syntaxe rigoureuse et précise qui offre une bonne visibilité de l'ensemble et la également sa portabilité vers d'autres applications.

Le repère choisi est le suivant:



Exemple simple de hiérarchie d'un fichier XML, l'exemple ne contient qu'une seule sphère et une seule source de lumière mais il est bien évidemment possible d'en ajouter autant que souhaitées en utilisant d'autres balises <sphere> et <srclum>.

exemple.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- Exemple de description d'une scene/>
<scene>
   <spheres>
         <sphere>
         <centre>
            <x>0</x>
            <y>0</y>
            <z>0</z>
         </centre>
         <couleursph>
            <r>255</r>
            <g>0</g>
            <b>0</b>
         </couleursph>
         <rayon>50</rayon>
         <indmiroir>0</indmiroir>
         <inddiffusion>0.4</inddiffusion>
          <indspeculaire>0.3</indspeculaire>
          <indrugosite>50</indrugosite>
          <triangulee>1</triangulee>
          <indtriangulee>20</indtriangulee>
      </sphere>
   </spheres>
   <srclums>
      <srclum>
         <point>
            <x>500</x>
            <y>-500</y>
            <z>500</z>
         </point>
         <couleur>
            <r>255</r>
            <g>255</g>
            <b>255</b>
         </couleur>
         <intensite>1</intensite>
      </srclum>
   </srclums>
   <obs>
      <prof>1000</prof>
   </obs>
   <fenetre>
      <chg>
         <xchg>-100</xchg>
         <ychg>-100</ychg>
         <zchg>200</zchg>
      </chg>
      <larg>200</larg>
      <haut>200</haut>
   </fenetre>
</scene>
```

coordonnées du centre de la sphère

Couleur RVB : ici la sphere sera rouge

indices utilisés dans le modèle d'éclairement (compris entre 0 et 1 sauf pour la rugosité entre 0 et 100)

Mode et indice de triangulation :
triangulee = 0 • > sphère parfaite
triangulee = 1 • > sphère triangulée avec la méthode du TD
triangulee = 2 • > sphère triangulée avec la triangulation de l'équateur

L'indice de triangulation représente le nombre de triangles souhaités. Il n'est pas pris en compte pour les sphères parfaites.

coordonnées du centre de la source

Couleur RVB de la source. Ici, elle est blanche.

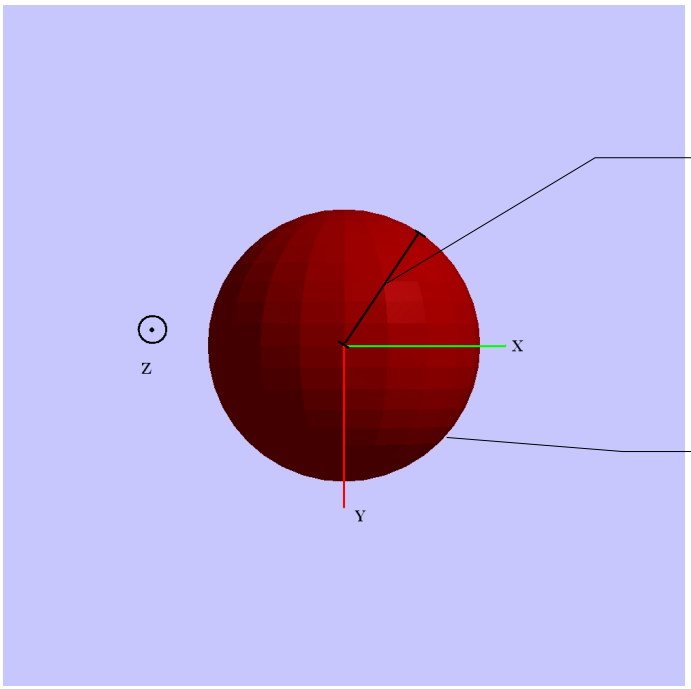
L'intensité de la source. Ici elle est maximum (1)

Position sur l'axe z de l'observateur. Cette valeur ne doit pas être execessive pour garantir une bonne projection en perspective

coordonnées du coin haut gauche de la fenetre.

Largeur et hauteur de la fenetre. En les multipliant par le pas utilisé on obtient la résolution de l'image finale.

Rendu : exemple.ppm



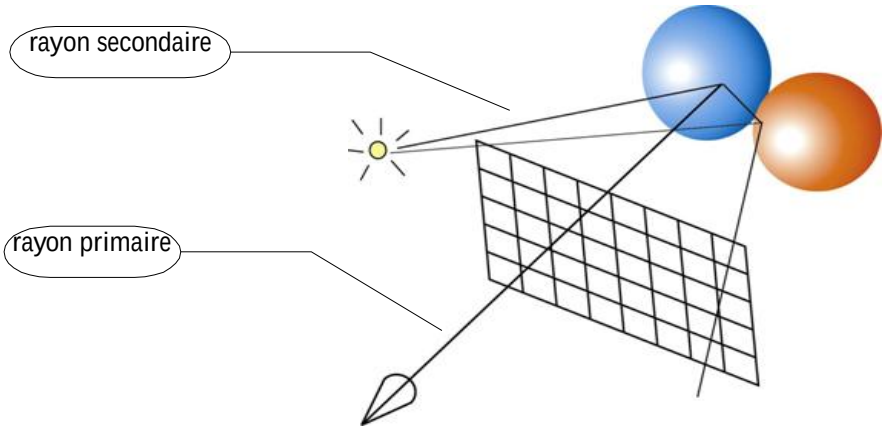
Rayon = 50

La sphère présente bien 20 “tranches” du Nord au Sud et d'Est en Ouest

II- LANCER DE RAYON:

1. *Principe du lancer du rayon :*

Pour cela, on place une fenêtre virtuelle dans la scène devant l'observateur (c'est cette image qui s'affichera sur l'écran). Pour chacun des pixels qui constituent l'image, on lance un rayon partant du point de vue (l'observateur) passant par le centre du pixel. La couleur du pixel traversé va être déterminée en suivant le cheminement du rayon lancé jusqu'aux sources lumineuses de la scène 3D.



Stéréotype: *Couleur lancerRayon(Scene * s,Rayon rayon,int niveau, int indice_sphere)*

Scene : la scène contient toutes les informations sur les sphères, les sources lumineuses, la fenêtre et l'observateur.
Rayon : un rayon est un vecteur directeur associé à un point d'orgine.
Niveau : défini le niveau de profondeur de réflexion des sphères miroirs.
indice_sphere : lors de l'appel récursif avec un rayon réfléchi, la sphère courante ne doit pas être prise en compte.

2. *Cheminement du rayon :*

- Pour chaque pixel de l'écran, on lance un rayon, une droite en fait, passant par l'oeil de l'observateur et le pixel considéré. Dans notre algorithme on utilise une fenêtre et un pas.

```
// balayage de la fenêtre
for ( y = depart_Y; y < arrivee_Y; y+=pas) {
  for ( x = depart_X; x < arrivee_X; x+=pas ) {
    vect = Vecteur3D(x,y,z_vect);
    ray.setVectDir(vect.Normalise());
    coul_pixel = lancerRayon(s,ray,0,-1);
```

- On trouve ensuite la première intersection de ce rayon avec la sphère la plus proche (fonctions *hitSphere* et *hitSphereTriangule*) ce qui implique un parcours de l'ensemble des sphères de la scène. On en déduit la couleur de cet objet en ce point.

```
// on cherche l'intersection la plus proche avec toutes les sphères de la scène
for ( int i = 0; i < s->getListeSpheres().size(); i++ ) {
  sph = s->getListeSpheres()[i];                    // sphère courante
  intersection_temp = rayon.hitSphere(sph);         // calcul de l'intersection
  if((indice_sphere==-1)||(indice_sphere!=i)){       // nécessaire pour le traitement des sphères miroirs
     if (intersection_temp!=NULL) {                  // si l'intersection n'est pas nulle,
        if (intersection==NULL) {                     // traite la première intersection
           intersection=intersection_temp;            // sauvegarde du point
                 couleur_sphere = sph->getCouleur();  // sauvegarde de la couleur
           indice_sph=i;
           }
        else {                                          // si le rayon intersecte d'autre sphère
                if (intersection_temp->getZ()>=intersection->getZ()) {
                                                                  // ou si le z de cette intersection est supérieur
                   couleur_sphere = sph->getCouleur();      // au z de l'ancienne intersection
                   indice_sph=i;                              // (le point est plus proche de l'observateur)
                   intersection=intersection_temp;
                }
              }
     }
    }
 }
```

- Si il n'y a pas d'intersection primaire alors il suffit d'afficher la couleur du fond

```
else {
   c_pixel = Couleur(c_fond);
 }
```

- Calculer un rayon à partir du point d'intersection du rayon avec l'objet et la source lumineuse. Trouver la première intersection de ce rayon avec un objet de la scène : si cette intersection existe et si l'objet intersecté est opaque, alors le point considéré est à l'ombre (cf *3.Ombres projetées)*. Si aucune intersection n'est trouvée vers la source lumineuse, alors l'objet est éclairé. Procéder au calcul d'éclairement en utilisant le modèle décrit dans la partie IV.
- Si la surface de l'objet est réfléchissante, calculer un nouveau rayon réfléchi (cf II-*4.Sphères miroirs*) à partir du point d'intersection. Rappeler récursivement la même procédure pour ce nouveau rayon. Ajouter la couleur trouvée pour le rayon réfléchi à la couleur du pixel déjà calculée.
- On borne alors cette couleur à 255 pour ses trois composantes RVB :

```
if (c_pixel.getRouge() >255) c_pixel.setRouge(255);
if (c_pixel.getVert() >255) c_pixel.setVert(255);
if (c_pixel.getBleu() >255) c_pixel.setBleu(255);
return c_pixel;
```

- La fonction *lancerRayon* retourne alors la couleur du pixel dont chaque composant est ensuite écrite dans le fichier de sortie au format ppm.

```
    fic_PPM<<coul_pixel.getRouge()<<" "<<coul_pixel.getVert()<<" "<<coul_pixel.getBleu()<<endl;
   }
}
fic_PPM.close();
```

3. Ombres projetées :

3.1 descripton de l'algorithme:

Le principe de calcul d'une ombre est simple : si le rayon secondaire (issue de l'intersection (i) avec la sphère la plus proche) vers une source donnée rencontre une autre sphère alors cette source ne contribue pas à éclairer ce point.
C'est donc une fois (i) calculée, qu'on effectue un parcours de l'ensemble des sources présentes dans la scène.

```
    for ( int j = 0; j < s->getListeSrcLumiere().size(); j++) {
          Vecteur3D vect_rsecondaire = Vecteur3D(*intersection,s->getListeSrcLumiere()[ j ].getPSource());
          // création du rayon secondaire
          r_secondaire.setVectDir(vect_rsecondaire);
```

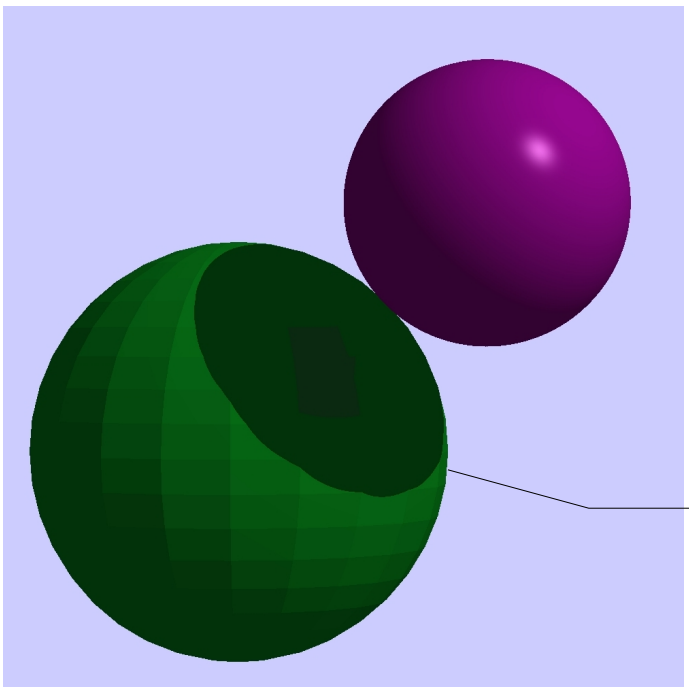
Pour déterminer si une sphère cache le point de la sphère courante, on réutilise les fonctions *hitSphere* et *hitSphereTriangulee* en ne considérant cette fois que la valeur de l'intersection. Si elle est nulle alors le point est dans l'ombre (pour cette source) sinon on procède au calcul du modèle d'éclairement.

```
 While ((!cache) && (k < s->getListeSpheres().size())) {
  sphere = s->getListeSpheres()[k];
  if (k!=indice_sph) {                          // la sphere n'est pas la sphere courante
    if (sphere->estTriangulee()) {         // si elle est triangulée on utilise hitSphereTriangulee
      Rayon ray_tria = r_reflechi.hitSphereTriangulee(sphere);
      if((ray_tria.getVectDir() == Vecteur3D(0.f,0.f,0.f)) && (ray_tria.getOrigine()==Point3D(0.f,0.f,0.f)))
          inter = NULL;
      else inter = new Point3D(ray_tria.getOrigine());
    }
   else { inter = r_reflechi.hitSphere(sphere); } // sinon on utilise hitSphere
   if (inter!=NULL)    cache = true;
 }
 k++;
}
```

3.2 différences suivant la nature de la sphere:

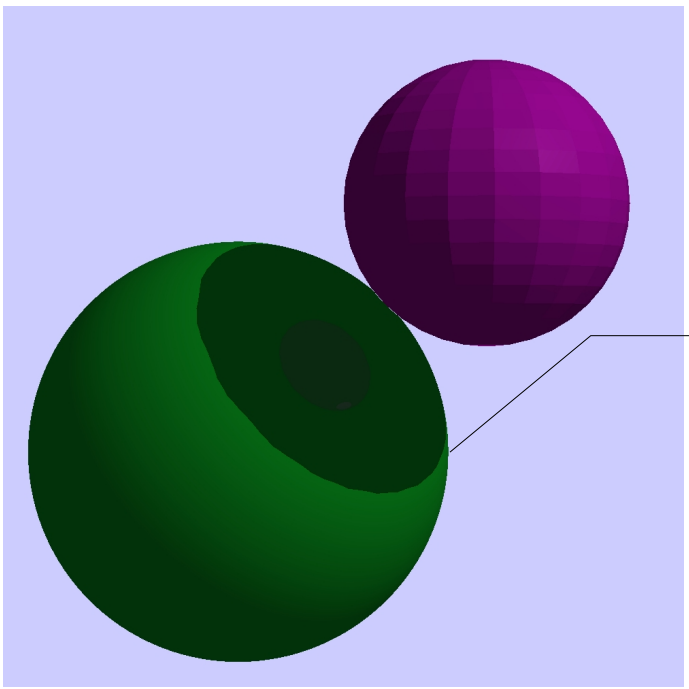
L'ombre projetée d'une sphère triangulée sur une sphère parfaite ne doit pas être la même que celle d'une sphère parfaite et vice-versa.
Notre algorithme permet de tenir compte de ces nuances et d'afficher des résultats corrects.

Ombre d'une sphère parfaite sur une sphère triangulée :



Sur une sphère triangulée, l'ombre d'une sphère parfaite s'adapte aux différentes facettes de la sphère verte.

Ombre d'une sphère triangulée sur une sphère parfaite :



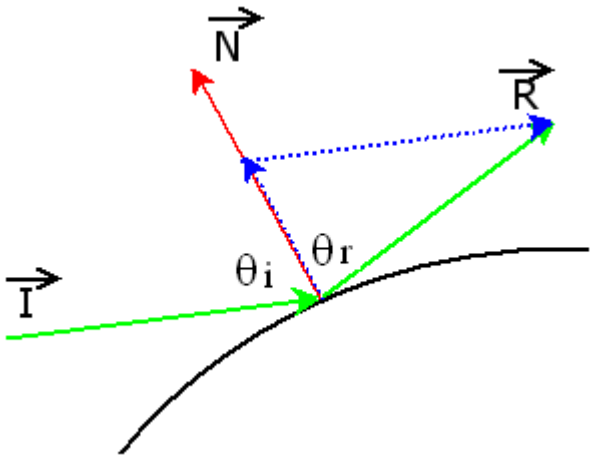
Ici le contour de l'ombre projetée correspond bien au contour des différentes facettes de la sphère violette et non à sa sphère englobante.

4. *Sphères miroirs :*

4.1 calcul du rayon réfléchi

Soient:

- I : le vecteur normé directeur du rayon primaire (observateur - pixel) de la source lumineuse,
- R : le vecteur normé réfléchi
- N : la normale à la surface de la sphère au point d'intersection



Sachant que abs(θi) = abs(θr), on établit la formule

$$\vec{R} = -2\vec{N}(\vec{N}.\vec{I}) + \vec{I}$$

4.2 explication de l'alogorithme :

Pour éviter le phénomène de réflexion à l'infini quand, par exemple, deux sphères miroirs se font face, il faut fixer arbitrairement un niveau de réflexion. On choisi donc de positionner à 5 le nombre maximum d'appel récursif pour un rayon réfléchi.

```
If((sph->getIndMiroir()>0)&&(niveau<5)) {
niveau++;
```

Si la sphère courante est miroir et que le niveau maximum n'est pas atteint, on calcule le nouveau rayon réfléchi.

```
float i_scalaire_n = rayon.getVectDir().Normalise() * vect_normal.Normalise();
Vecteur3D rayM = (vect_normal.Normalise()*(-2*i_scalaire_n)) + rayon.getVectDir().Normalise();
Rayon reflet = Rayon(*intersection,rayM.Normalise());
```

On peut alors lancer le rayon réfléchi de la même façon qu'un rayon primaire:

```
Couleur c_pixel_int = lancerRayon(s,reflet,niveau,indice_sph);
```

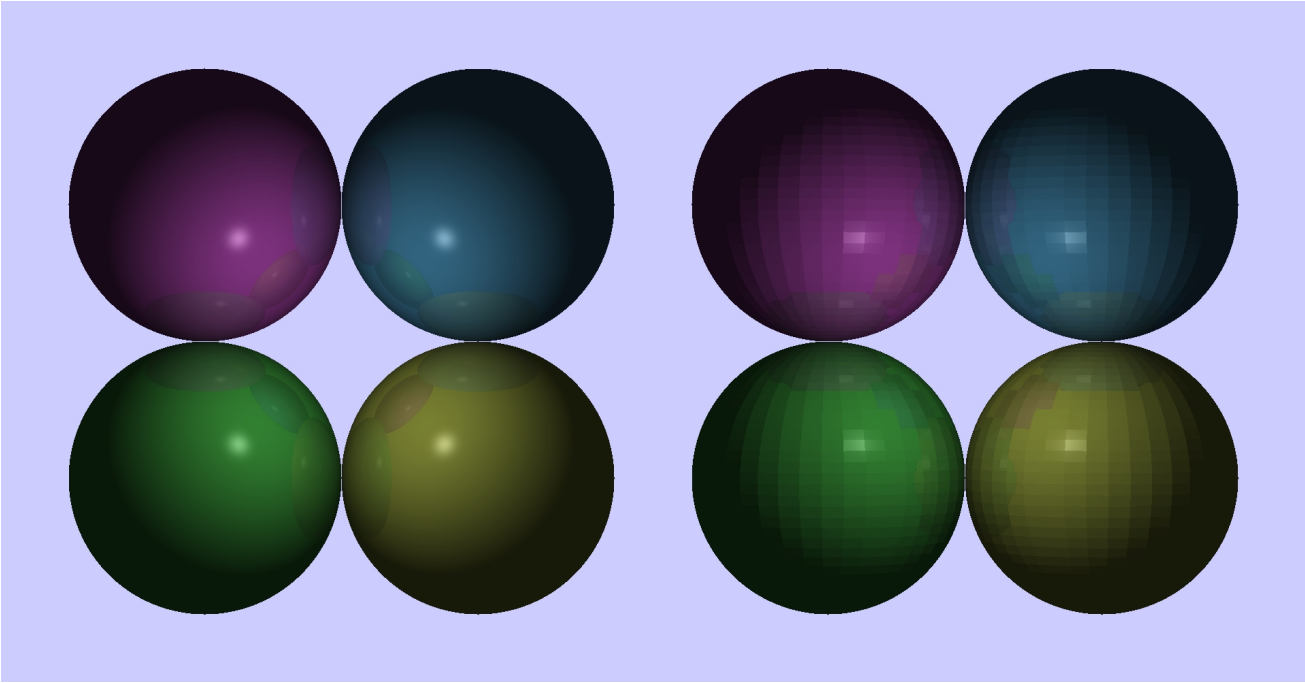
On récupère ainsi la couleur du pixel issue de la réflexion seule. Le parametre *indice_sphere* permet de ne pas considérer la sphère courante dans le calcul d'intersection entre le rayon réfléchi et une éventuelle sphère.

Si la couleur est différente de celle du fond alors le rayon réfléchi a rencontré une sphère. On calcule alors la couleur finale du pixel en ajoutant la couleur du reflet à celle de la sphère tout en tenant compte de l'indice miroir de la sphère.

```
if (!(c_pixel_int == c_fond)) {
 int r_pix =(int)(c_pixel_int.getRouge()*sph->getIndMiroir()+(c_pixel.getRouge()*(1-sph->getIndMiroir())));
 int v_pix = (int)(c_pixel_int.getVert() * sph->getIndMiroir()+(c_pixel.getVert()*(1-sph->getIndMiroir())));
 int b_pix = (int)(c_pixel_int.getBleu() * sph->getIndMiroir()+(c_pixel.getBleu()*(1-sph->getIndMiroir())));

 c_pixel = Couleur(r_pix,v_pix,b_pix);
```

4.3 nature de la sphère:

L'algorithme de calcul du rayon réfléchi ne dépend pas de la nature de la sphère : parfaite ou triangulée.



4 sphères miroirs parfaites

4 sphères miroirs triangulées

III- CALCUL DES INTERSECTIONS

Une des parties cruciale du lancer de rayon est le calcul de l'intersection entre une sphère et un rayon.
Comme nous avons 2 types de sphères (facétisées et non-facétisées), il nous a fallu créer 2 fonctions d'intersections.

La première permet d'obtenir un point intersection entre une sphère parfaite et un rayon. Elle s'inspire de la méthode du polycopié du cours.
Elle consiste en la résolution d'une équation du second degré obtenue grâce aux équations de la sphère et de la droite porteuse du rayon.
Cette équation est la suivante :

$$t^2*(V_x^2+V_y^2+V_z^2) + t*2*(V_xF_x+V_yF_y+V_zF_z-V_x.x_0-V_y.y_0-V_z.z_0) + F_x^2+F_y^2+F_z^2+x_0^2+y_0^2+z_0^2-2F_x.x_0-2F_y.y_0-2F_z.z_0-R^2 = 0$$

où : t est le paramètre
(V_x,V_y,V_z) sont les coordonnées du vecteur directeur du rayon
(F_x,F_y,F_z) sont les coordonnées du point origine du rayon
(x_0,y_0,z_0) sont les coordonnées du centre de la sphère
R est le rayon de la sphère

Cette équation n'est pas tout à fait celle du cours car cette dernière était fausse.
Après avoir calculé le déterminant •, la résolution est simple :

a. si •<0, il n'y a pas d'intersection
b. si •=0, un seul point d'intersection
c. si •>0, 2 points d'intersection : on prend le plus proche

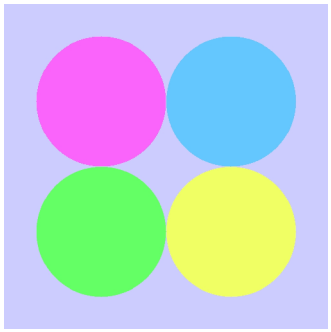
On vérifie naturellement que t est positif dans chacun des cas afin de ne pas avoir des intersections qui ne sont pas dans le champ de vision (t<0 —> intersection derrière l'oeil).

On renvoie donc le point correspondant au paramètre t trouvé si ce dernier est valide, NULL dans tous les autres cas.

La seconde fonction cherche l'intersection entre une sphère facétisée et un rayon.
Elle renvoie, quant à elle, un objet rayon qui n'est pas à considérer comme un rayon à proprement parler mais comme une « structure » permettant de connaître le point d'intersection (origine du rayon) et, du même coup, le vecteur normal à la facette intersectée (vecteur directeur du rayon).
Le fait de renvoyer en même temps le vecteur normal permet de ne pas le recalculer ultérieurement quand on en aura réellement besoin.
L'algorithme consiste en la résolution d'équations obtenues grâce à l'équation du plan d'un triangle et celle de la droite porteuse du rayon. Ces équations ont été trouvées sur Internet.
Après résolution, on obtient le paramètre t correspondant. On en déduit donc le point d'intersection.
Le vecteur normal est obtenu par un produit scalaire judicieux entre 2 vecteurs de la facette afin qu'il ne soit pas orienté vers l'intérieur.
On renvoie le rayon dont l'origine est (0,0,0) et le vecteur directeur est (0,0,0) si on ne touve pas d'intersection.

IV- DESCRIPTION DU MODELE D'ECLAIREMENT :

Voici un exemple de scène sans éclairement :



Pour donner plus de réalisme à la scène, il nous faut calculer la couleur de ses points en prenant en considération à la fois l'ensemble des sources lumineuses et les caractéristiques des surfaces des sphères éclairées.

Le modèle d'éclairement de Phong, selon lequel l'intensité lumineuse d'un point P est fonction de l'énergie réémise par ce point après qu'il ait été touché par la lumière, est le suivant :

$$I_p = I_a + I_d + I_s$$

On constate l'addition de trois énergies lumineuses :

- I_a : l'énergie due à la lumière ambiante
- I_d : l'énergie due à la réflexion diffuse
- I_s : l'énergie due à la réflexion spéculaire

D'une manière plus détaillée, on peut écrire :

$$I_p = k_a \cdot C_a + k_d \cdot \sum_{i=1}^{n} C_{d_j} + k_s \cdot \sum_{i=1}^{n} C_{s_j}$$

où n est le nombre de sources lumineuses ;

k_a, k_d et k_s sont les indices d'atténuation de la lumière par les surfaces (indAmbiant, indDiffusion et indSpeculaire) ;

C_a, C_d et C_s sont les coefficients dus à la lumière ambiante, diffuse et spéculaire.

C_a : Coefficient dû à la lumière ambiante

La lumière ambiante est assimilable à une lumière diffuse, indépendante de la position de la source lumineuse. On a :

$$C_a = coul_d(p) \cdot I_A$$

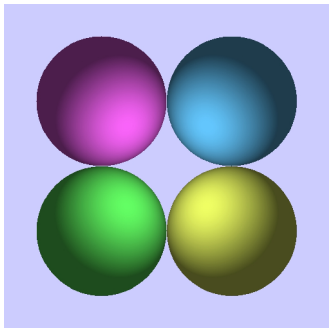
I_A est l'intensité ambiante, comprise entre 0 et 1. Elle est définie par :
Indice ambiant (indAmbiant) = 1 – indice miroir – indice diffusion – indice spéculaire

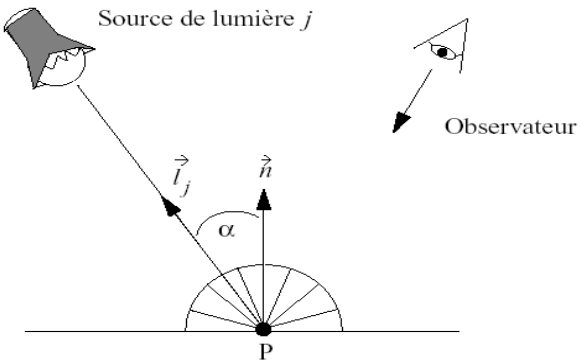
Les trois composantes de ce coefficient sont donc calculées ainsi :

```
float intensiteAmbiante = 1.f; //intensité ambiante, fixée dans le programme

float Ca_r = (float)couleur_sphere.getRouge() * intensiteAmbiante;
float Ca_v = (float)couleur_sphere.getVert()  * intensiteAmbiante;
float Ca_b = (float)couleur_sphere.getBleu()  * intensiteAmbiante;
```

C_d : Coefficient dû à la lumière diffuse

La réflexion diffuse donne un aspect mat à la sphère. La diffusion est indépendante de la position de l'observateur. Cette scène est la même que la précédente en prenant en compte la diffusion :





En considérant la figure précédente, on a :

$$C_{d_j} = coul_d(p) \cdot I_j \cdot (\vec{n} \cdot \vec{l}_j)$$

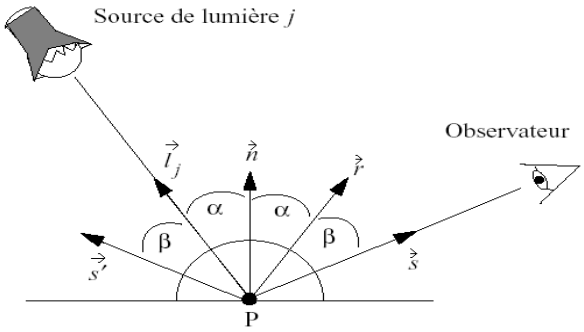
I_j est l'intensité de la source lumineuse j, comprise entre 0 et 1.
Le produit scalaire entre la normale à la surface et la direction de la source lumineuse est égale au cosinus de l'angle • entre ces deux droites.

On peut calculer les composantes du coefficient de lumière diffuse :

```
 if (produit_scalaire>=0) {
 somme_contributions +=
              produit_scalaire * s->getListeSrcLumiere()[j].getIntensite();
 (…)
}
float Cd_r = (float)couleur_sphere.getRouge() * somme_contributions;
float Cd_v = (float)couleur_sphere.getVert()  * somme_contributions;
float Cd_b = (float)couleur_sphere.getBleu()  * somme_contributions;
```

C_s : Coefficient dû à la lumière spéculaire

La réflexion spéculaire est la lumière réfléchie sans pénétrer la surface. Elle donne un aspect brillant à la sphère et des reflets de la couleur de la source lumineuse, dont les emplacements dépendent de la position de l'observateur.



$$C_{s_j} = coul_s(p) \cdot I_j \cdot (\vec{r} \cdot \vec{s})^m$$

Le produit scalaire r . s correspond au cosinus de l'angle •.

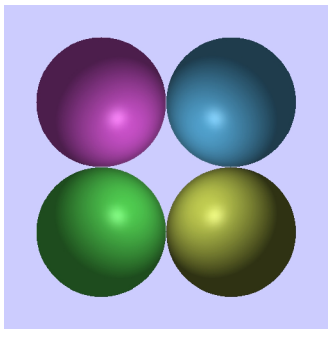
```
//si le produit scalaire est > 0, la source contribue à l'éclairement du point
if (produit_scalaire>=0) {
Vecteur3D vect_spec = (2* vect_normal * produit_scalaire) - vect_reflechi; *

Vecteur3D vect_s = Vecteur3D(*intersection,rayon.getOrigine());
Couleur coul_src = s->getListeSrcLumiere()[j].getCouleur();
int rugosite = sph->getIndRugosite();
somme_spec = pow(s->getListeSrcLumiere()[j].getIntensite()
                * (vect_spec * vect_s), rugosite);

Cs_r += coul_src.getRouge() * somme_spec;
Cs_v += coul_src.getVert()  * somme_spec;
Cs_b += coul_src.getBleu()  * somme_spec;
}
 //les vecteurs utilisés sont normalisés
```

* Le produit scalaire r . s peut s'écrire :

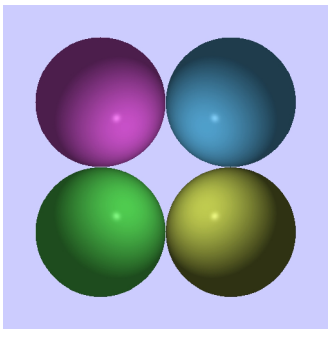
$$\vec{r} \cdot \vec{s} = (2\vec{n}(\vec{n} \cdot \vec{l}_j) - \vec{l}_j) \cdot \vec{s}$$

r . s = cos • décroît lorsque • croît, d'autant plus vite que m est grand. m est appelé indice de rugosité.

m < 10 entraîne des reflets très étalés



m = 100 entraîne des reflets très localisés.



Calcul de la couleur finale :

On peut maintenant calculer les composantes finales de la couleur en chaque point :

```
lum_diffuse_rouge = indAmbiant * Ca_r
                + sph->getIndDiffusion()  * Cd_r
                     + sph->getIndSpeculaire() * Cs_r;
lum_diffuse_verte = indAmbiant * Ca_v
                + sph->getIndDiffusion()  * Cd_v
                + sph->getIndSpeculaire() * Cs_v;
lum_diffuse_bleue = indAmbiant * Ca_b
                + sph->getIndDiffusion()  * Cd_b
                + sph->getIndSpeculaire() * Cs_b;
c_pixel =
 Couleur((int)lum_diffuse_rouge,(int)lum_diffuse_verte,(int)lum_diffuse_bleue);
```

MIROIR

```
 Couleur c_pixel_int = lancerRayon(s,reflet,niveau,indice_sph);

if (!(c_pixel_int == c_fond)) {
 int r_pix = (int)(c_pixel_int.getRouge() * sph->getIndMiroir()
                +(c_pixel.getRouge() *(1-sph->getIndMiroir())));
 int v_pix = (int)(c_pixel_int.getVert() * sph->getIndMiroir()
                +(c_pixel.getVert()*(1-sph->getIndMiroir())));
 int b_pix = (int)(c_pixel_int.getBleu() * sph->getIndMiroir()
                +(c_pixel.getBleu()*(1-sph->getIndMiroir())));

 c_pixel = Couleur(r_pix,v_pix,b_pix);
 }
```

Le détail des calculs effectués pour une sphère miroir est décrit dans la partie II - 4

V- TRIANGULATION

Cette partie est dédiée à l'explication de la fonction de triangulation des sphères.
Notre programme permet de spécifier 2 types de triangulations : celle basée sur le TD et sur une approche de triangulation à l'aide de méridiens et de parallèles, l'autre basée sur la triangulation du disque de l'équateur puis de la projection sur la sphère des triangles obtenus.

A. Triangulation à l'aide de méridiens et de parallèles

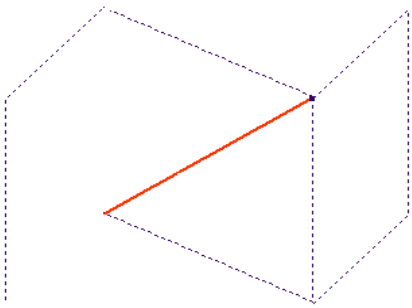
Cet algorithme se base sur celui vu en TD, nous ne l'expliquerons donc pas en détail mais nous rappellerons cependant les grandes lignes.

1. équation paramétrique d'un cercle

Tout point M d'un cercle peut être défini à l'aide de l'équation paramétrique de ce cercle.

On a :

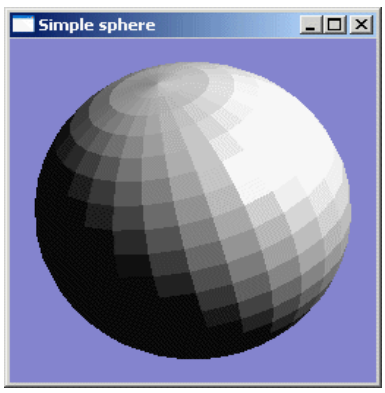
$$x_M = r * cos\bullet * cos\bullet + x_O$$
$$y_M = r * sin\bullet + y_O$$
$$z_M = r * cos\bullet * sin\bullet + z_O$$



où • et • sont les angles paramètres
r est le rayon de la sphère
(x_O,y_O,z_O) sont les coordonnées du centre de la sphère

- ***idée générale***

L'idée est de faire varier • et • avec des incréments fixes afin d'obtenir des « tranches » de sphère.
On voit rapidement que • • [0,2• [et que • • [-• /2,• /2].
Pour chaque pas de • , on fait varier • entre [-• /2,• /2] et on génère les triangles correspondants.
Ce que l'on doit obtenir est de la forme :



Les triangles du pôles sont générés différemment que les autres.
Seul un hémisphère est traité : le second est généré par symétrie (y devient -y et • devient – •).

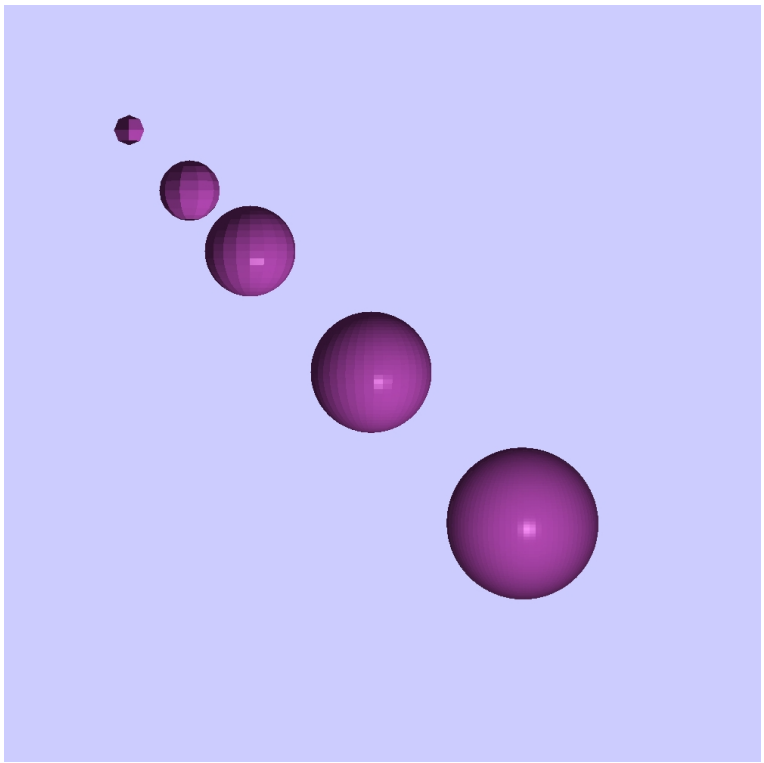
2. algorithme

```
vector<Triangle*> Sphere::Triangule() {
	vector<Triangle*> res;
	float pi = 3.1415926535;
	float ecart_OE = indTriangulee;			// écart entre 2 lignes Ouest - Est
	float ecart_NS = ecart_OE/2;				// écart entre 2 lignes Nord - Sud
	float delta_teta = (float) (2 * pi / ecart_OE);		// différence entre 2 angles Ouest - Est
	float delta_phi = (float) (pi / (2 * ecart_NS));	// différence entre 2 angles Nord - Sud
	float teta, phi;
	float phi_pole = (float) ((ecart_NS-1) * pi / (2 * ecart_NS));
	float x,y,z;
	Point3D p1,p2,p3;
	int i,j;
	for (i=0;i<ecart_OE;i++) {
	teta = (float) (i * delta_teta);
	for (j=0;j<ecart_NS-1;j++) {
		phi = (float) (j * delta_phi);
		// création du 1er triangle  (hémisphère nord)
		p1 = p1.engendrerPoint(teta,phi,rayon,centre);
		p2 = p2.engendrerPoint(teta,phi+delta_phi,rayon,centre);
		p3 = p3.engendrerPoint(teta+delta_teta,phi+delta_phi,rayon,centre);
		res.push_back(new Triangle(p1,p2,p3,couleur));
		// création du 2ème triangle (hémisphère nord)
		p2 = p2.engendrerPoint(teta+delta_teta,phi+delta_phi,rayon,centre);
		p3 = p3.engendrerPoint(teta+delta_teta,phi,rayon,centre);
		res.push_back(new Triangle(p1,p2,p3,couleur));
		// création du 3ème triangle (hémisphère sud)
		p1 = p1.engendrerPoint(teta,-phi,rayon,centre);
		p2 = p2.engendrerPoint(teta,-phi-delta_phi,rayon,centre);
		p3 = p3.engendrerPoint(teta+delta_teta,-phi-delta_phi,rayon,centre);
		res.push_back(new Triangle(p1,p3,p2,couleur));
		// création du 4ème triangle (hémisphère sud)
		p2 = p2.engendrerPoint(teta+delta_teta,-phi-delta_phi,rayon,centre);
		p3 = p3.engendrerPoint(teta+delta_teta,-phi,rayon,centre);
		res.push_back(new Triangle(p1,p3,p2,couleur));
	}
	// création du triangle du pôle nord
	p1 = p1.engendrerPoint(teta,phi_pole,rayon,centre);
	p2 = p2.engendrerPoint(teta+delta_teta,phi_pole,rayon,centre);
	p3 = Point3D(centre.getX(),rayon+centre.getY(),centre.getZ());
	res.push_back(new Triangle(p1,p2,p3,couleur));
	// création du triangle du pôle sud
	p1 = p1.engendrerPoint(teta,-phi_pole,rayon,centre);
	p2 = p2.engendrerPoint(teta+delta_teta,-phi_pole,rayon,centre);
	p3 = Point3D(centre.getX(),-rayon+centre.getY(),centre.getZ());
	res.push_back(new Triangle(p1,p2,p3,couleur));
	}
	return res;
};
```

3. problèmes rencontrés

Nous n'avons pas eu de réels problèmes de codage de cet algorithme, le seul ayant été de générer les points des triangles dans le bon ordre afin que les vecteurs normaux des facettes soient bien orientés vers l'extérieur.

4. exemple de sphères facétisées obtenues



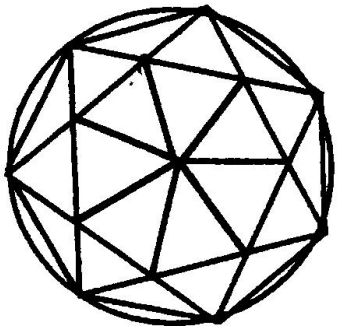
Cette image permet de montrer l'influence de l'indice de triangulation.
Plus cet indice est élevé, plus le nombre de facettes est important et plus la sphère apparaît lisse.
Indices de l'image de la sphère la plus éloignée à la sphère la plus proche : 4 – 10 – 20 – 40 – 80

B. Triangulation de l'équateur

Une autre méthode de triangulation d'une sphère consiste à trianguler le disque de l'équateur puis de projeter les triangles obtenus sur la sphère.

La première question est : comment trianguler le plan de l'équateur ?

Une solution assez intuitive peut se représenter de la manière suivante :



Pour ce faire, nous avons pensé à une découpe de ce disque en plusieurs cercles de rayons croissants.

Le nombre de cercles est donné par l'indice de triangulation de la sphère.

Entre chaque cercle interne, nous créons des triangles qui permettent de rejoindre ces cercles.

Nous découpons donc en premier temps notre disque en plusieurs cercles dont l'écart est variable (appelé ecart_alpha) afin de ne pas avoir des triangles plus grands près de l'équateur lors de la projection. Grâce à cet écart variable, les triangles apparaissent tous de la même taille.

Le premier cercle est géré différemment car il représente les futurs pôles. Nous le divisons donc simplement en 7 triangles.

Sur chacun des cercles suivants, on engendre 14 triangles.
La méthode est la suivante :

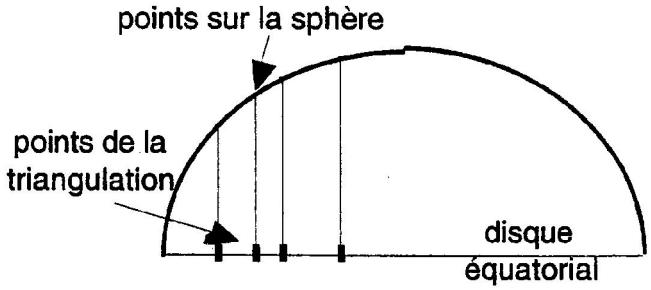
- Sur chaque cercle interne, il y a 7 points.
- A partir d'un point du cercle précédant, on engendre 2 triangles permettant de lier ces deux triangles :
  - l'un partant de ce point et ayant 2 points sur le cercle suivant
  - l'autre ayant deux points sur le cercle et l'autre sur le cercle suivant

Les détails des calculs sont visibles dans l'algorithme.

Ceci nous donne donc le code suivant :

```
// on engendre des triangles pour chaque cercle interne
 for(int i=0;i<nb_cercles;i++) {
    alpha -= ecart_alpha;                    // l'angle alpha est décrémenté de ecart_alpha
    if (i>1)
     angle_depart += ecart_angles / 2;          // l'angle de départ est incrémenté de écart_angles/2
                                                 // dès que l'on est sur le 2ème cercle interne
   angle = angle_depart;                        // l'angle en cours est initialisé à l'angle de départ
   rayon_cercle_prec = rayon_cercle;            // la valeur du rayon du cercle précédent est rayon_cercle
   rayon_cercle = (double) (cos(alpha)*rayon);        // rayon_cercle est ensuite modifié

   if (i!=0)                        // si on n'est pas sur le 1er cercle interne, le nombre de triangles est de 14
     nb_triangles = 14;

   for(int j=0;j<nb_triangles;) {      // on crée nb_triangles

              if (i==0) {                    // 1er cercle interne --> correspond aux pôles donc géré différemment
                                             // on génère un triangle par un triangle (7 en tout)
              p1 = centre;
              p2 = Point3D(x_centre+rayon_cercle*cos(angle),y_equateur,z_centre+rayon_cercle*sin(angle));
             p3=Point3D(x_centre+rayon_cercle*cos(angle+ecart_angles),y_equateur,z_centre+rayon_cercle*
sin(angle+ecart_angles));
              tri = new Triangle(p1,p2,p3,couleur);
              res_int.push_back(tri);          // ajout du triangle dans la liste des triangles de l'équateur
              angle += ecart_angles;          // l'angle est incrémenté de ecart_angles
              j++;                             // j est incrémenté de 1 (nombre de triangles créés)
     }

     else {
              // les autres cercles internes --> on génère 2 triangles par 2 triangles
             p1=Point3D(x_centre+rayon_cercle_prec*cos(angle),y_equateur,z_centre+rayon_cercle_prec*sin
(angle));
           p2=Point3D(x_centre+rayon_cercle*cos(angle-
ecart_angles/2),y_equateur,z_centre+rayon_cercle*sin(angle-ecart_angles/2));

p3=Point3D(x_centre+rayon_cercle*cos(angle+ecart_angles/2),y_equateur,z_centre+rayon_cercle*sin(angl
e+ecart_angles/2));
              tri = new Triangle(p1,p2,p3,couleur);
              res_int.push_back(tri);          // ajout du triangle dans la liste des triangles de l'équateur

              // on ne change qu'un seul point et l'ordre de construction du triangle

p2=Point3D(x_centre+rayon_cercle_prec*cos(angle+ecart_angles),y_equateur,z_centre+rayon_cercle_pre
c*sin(angle+ecart_angles));
              tri = new Triangle(p1,p3,p2,couleur);
              res_int.push_back(tri);          // ajout du triangle dans la liste des triangles de l'équateur

              angle += ecart_angles;          // l'angle est incrémenté de ecart_angles
              j+=2;                          // j est incrémenté de 2 (nombre de triangles créés)
     }
   }
 }
```

La triangulation de l'équateur étant effectuée, il faut ensuite projeter les triangles obtenus sur la sphère parfaite.



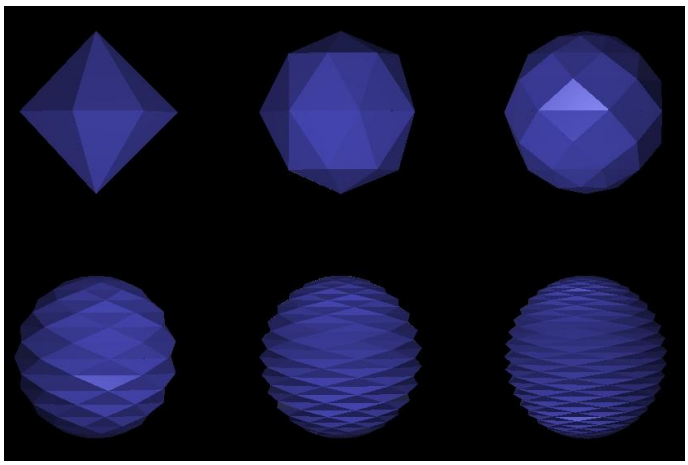
Pour ce faire, on applique la même méthode que l'intersection entre un rayon et une sphère.

On récupère dans un premier temps les points du triangle de l'équateur traité.
On crée 3 rayons dont les origines sont ces points et dont le vecteur directeur est le vecteur (0,1,0) qui est le vecteur normal au plan de l'équateur.
On calcule l'intersection entre ces rayons et la sphère à trianguler. Les points obtenus permettent de créer un nouveau triangle qui est ajouté à la liste des triangles de la sphère.
Cette méthode permet de ne générer que le pôle nord car le vecteur est (0,1,0).
Afin de générer le pôle sud, on crée des rayons de la même manière, le vecteur directeur étant modifié à (0,-1,0).
On obtient donc un nouveau triangle à ajouter à la liste.

On vérifie bien à chaque fois que les intersections trouvées ne sont pas NULL. Dans le cas contraire, cela signifie que le point dont on cherche le projeté se trouve sur le bord du disque de l'équateur et donc qu'il n'était pas à modifier.

Voici un exemple de sphères obtenues :

On remarque bien, là aussi, l'influance de l'indice triangulation qui devient de plus en plus élevé :
1 puis 2, 4, 8, 14, 20.

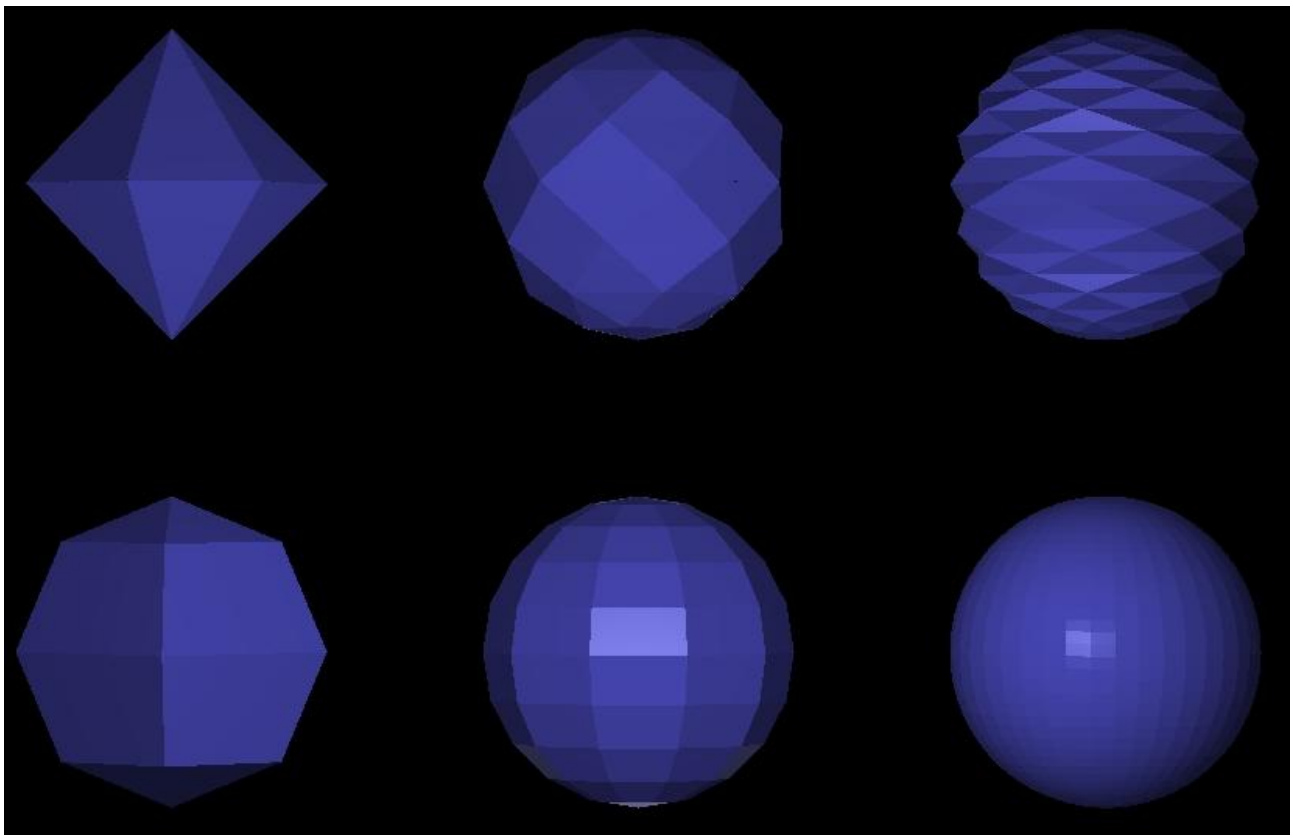


Pour résumer sur cette méthode, on peut dire qu'il existe évidemment bien d'autres solutions suivant le nombre de triangles voulus aux pôles.
Celle-ci, comparée à d'autres, permet de représenter une sphère avec un nombre relativement faible de triangles (7 triangles sur le premier cercle interne puis 14 sur les autres).
Nous avions eu comme première idée de découper en 6 le premier cercle mais cela nous faisait multiplier par 3 le nombre de triangles d'un cercle à l'autre.
Pour des raisons de place, nous avons préféré utiliser la triangulation décrite.

On remarque également que cette méthode génère beaucoup moins de triangles que la triangulation précédente vue en TD.

C'est donc tout normalement que nous avons remarqué une génération beaucoup plus rapide des images avec l'utilisation de cette méthode.

On peut remarquer sur l'image qui suit la différence entre les sphères obtenues avec ces 2 méthodes de triangulations:



VI- EXPLICATION DES CLASSES

Voici une explication des différentes classes composant notre programme.
On explicitera notamment les attributs de chaque classe et les fonctions principales (hors affichage, accesseurs et modificateurs).

classe Couleur :

Une couleur est définie par un triplet RVB.
Chaque composante est un entier compris entre 0 et 255.
On aurait pu également considérer chaque composante comme un flottant compris entre 0 et 1. Il aurait fallu, dans ce cas, multiplier à chaque fois ces valeurs par 255 afin de retrouver le triplet RVB de base.
Nous avons donc décidé de gérer directement des entiers pour éviter trop de conversion.
La comparaison de couleurs est gérée par l'opérateur == que nous avons redéfini.

classe Point3D :

Un point est défini par un triplet de coordonnées (x, y, z) plus une coordonnée homogène qui vaut 1 par défaut.
Cette dernière n'est pas utilisée dans le programme mais est présente dans le cadre de futures utilisations de cette classe.
Comme pour la couleur, l'égalité de 2 points est gérée par l'opérateur ==.
Un point peut être créé également à partir de l'équation paramétrique d'une sphère. Cette création est utilisée lors de la triangulation d'une sphère.

classe Triangle :

Un triangle est défini par 3 points et une couleur.
Pas de fonction particulière dans cette classe. Elle ne sert que lors de la triangulation.

classe SrcLumiere :

Une source de lumière est définie par un point d'origine (*pSource*), une intensité lumineuse et une couleur.
Là non plus pas de fonction particulière si ce n'est plusieurs constructeurs permettant de créer des sources lumineuses avec des valeurs par défaut (intensité et couleur).

classe Sphere :

Une sphère est définie par :

- un rayon
- un centre
- une couleur
- un indice miroir (compris entre 0 et 1)
- un indice spéculaire (compris entre 0 et 1)
- un indice de diffusion (compris entre 0 et 1)
- un indice de rugosité (compris entre 0 et 100)
- un indice de triangulation (nombre de méridiens, 0 si non triangulée)
- une liste de triangles (pour les sphères dont l'indice de triangulation est > 0)

Une sphère n'est triangulée que si son indice de triangulation est > 0.
L'importance de ces indices sera précisée dans la partie *description du modèle*.
De même, la fonction de triangulation sera explicitée ultérieurement lors de la partie *triangulation*.

Au départ, nous avions pensé avoir 2 classes sphères : une classe *SphereParfaite* et une classe *SphereTriangulee* héritant de *SphereParfaite*.
Une sphère triangulée aurait eu juste une liste de triangles et un indice de triangulation en plus.
Pour des questions de simplification au niveau du codage et au niveau du parser, nous avons tout concentré dans une même classe *Sphere*.

classe Fenetre :

Une fenêtre est définie par un point de "départ" (correspond au coin haut gauche), par une largeur (taille sur (Ox)) et une hauteur (taille sur (Oy)).
Une fenêtre est donc forcément parallèle au plan (Oxy) ce qui simplifie le problème.
On aurait pu définir une fenêtre par 2 points correspondants respectivement au coin haut gauche et au coin bas droit de la fenêtre. Dans ce cas, la fenêtre aurait pu être dans n'importe quel plan de l'espace.

classe Scene :

Une scène est définie par une liste de sphères, une liste de sources lumineuses, une fenêtre et une position de l'observateur.
L'observateur est supposé fixe sur l'axe (Oz). Sa position est donc ramenée à un seul flottant correspondant à sa coordonnée sur cet axe.
Les listes sont gérées par le type *vector<K>* ce qui nous permet de ne pas créer un type liste à part entière.
Dans cette classe, sont présentes des fonctions d'ajout d'éléments dans la scène qui sont utilisées par le parser XML.

classe Vecteur3D :

Un vecteur de l'espace est, comme pour un point, défini par 3 coordonnées (x,y,z).
Nous avons quand même consacré une classe pour les vecteurs pour des questions de logique (un vecteur n'est pas un point) et pour avoir des fonctions supplémentaires (*Norme()*, *Normalise()*, opérateurs de comparaison,...) qui ne peuvent pas toutes s'appliquer sur des objets de type *Point3D*.
Un vecteur peut être construit à partir de ses coordonnées mais aussi à partir de 2 points, le premier étant le point de départ du vecteur et le second son point d'arrivée.

classe Rayon :

Un rayon est défini par un point d'origine et un vecteur directeur ce qui s'apparente à une demi-droite.
C'est dans cette classe que sont codées les fonctions d'intersection entre un rayon et une sphère.
Il y a 2 fonctions d'intersection : une pour les sphères parfaites et l'autre pour les sphères triangulées.
L'explication de ces fonctions se fera dans la partie *calcul des intersections*.

Conclusion:

Cette mise en œuvre du lancer de rayon ne peut rendre compte de tous les phénomènes optiques tels que la réfraction, la dispersion lumineuse.
Cette technique permet la génération d'images très réalistes mais peut requérir un temps de calcul colossal en fonction de la complexité de la scène 3D et surtout du nombre de facettes utlisées pour représenter les sphères triangulées.