Étude et optimisation du lancer de rayon

1 Présentation du problème

Nous nous intéressons à la synthèse d'image à partir d'une scène modélisée en 3D. Celle-ci se divise en 2 parties : le calcul de visibilité à l'aide du lancer de rayon[2], et le calcul de couleur avec une approximation de réflectivité bidirectionnelle [5]. Ce que l'on appelera ensuite une scène [1] est la description :

- D'une ou plusieurs sources lumineuses (position, couleurs, ...), unique dans ce modèle
- D'une liste d'objets avec leurs propriétés (sphères, triangles, ...)
- D'une caméra associée à un écran (position, taille et nombre de pixels)

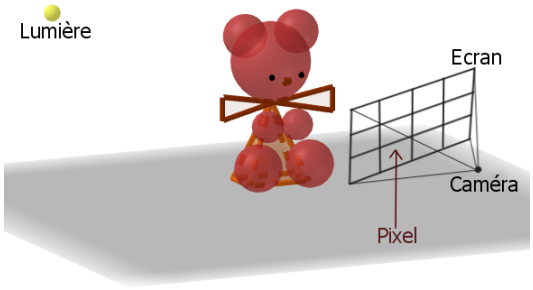


FIGURE 1 – Représentation de la scène à afficher

2 Principe du lancer de rayon

2.1 Principe physique

Le lancer de rayon se fonde sur le principe de retour inverse de la lumière : à la place de partir de la source lumineuse, l'algorithme simule des rayons partant de la caméra et rebondissant sur les objets [2].

Ainsi, pour chaque pixel de l'écran, l'algorithme s'intéresse au vecteur directeur de la caméra vers ce pixel et trouve l'objet le plus proche intersectant ce rayon. Une fois l'intersection trouvée, si le rayon partant de celle-ci et allant vers la source lumineuse n'intersecte aucun objet (c'est-à-dire que la lumière n'est pas obstruée), un calcul de couleur est effectué. Sinon, le pixel est dans l'ombre. En considérant les réflexions, la couleur totale est donc : $(1 - \text{coef}_{\text{réflexion}}) \times \text{couleur}_{\text{objet}} + \text{coef}_{\text{réflexion}} \times \text{couleur}_{\text{réflexion}}$

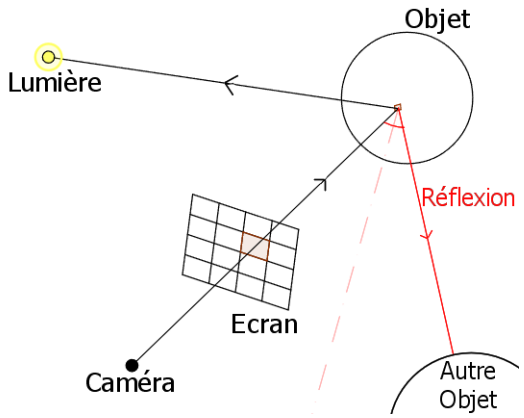


FIGURE 2 – Principe du lancer de rayon

Un rayon rebondit sur un objet avec une certaine proportion de l'intensité incidente (1 pour un miroir 0 pour un objet opaque). Au cours des réflexions cette proportion diminue, et elle sera négligée à partir d'un seuil noté ind_{ref}.

2.2 Algorithme [1]

Données : Description d'une scène, ind_{ref}
Résultat : Image de la scène $l \times c$
ecran $\leftarrow$ tableau $l \times c$ de pixels
pour *chaque pixel(i,j)* **faire**
  $prop_{intensite} \leftarrow 1$
  $\overrightarrow{vec} \leftarrow$ vecteur de la camera vers (i, j)
  tant que $prop_{intensite} > ind_{ref}$ **faire**
    $dist_{min} \leftarrow +\infty$
    $obj_{proche} \leftarrow None$
    pour *chaque objet obj* **faire**
      $dist \leftarrow$ distance de l'éventuelle intersection entre vec et obj
      si $dist_{min} > dist$ **alors**
        $dist_{min} \leftarrow dist$
        $obj_{proche} \leftarrow obj$
    $\overrightarrow{lum} \leftarrow$ rayon du point d'intersection vers la lumière
    si $obj_{proche} \neq None$ ***et*** $\overrightarrow{lum}$ *n'intersecte aucun objet* **alors**
      $\overrightarrow{vec} \leftarrow$ réflexion par rapport à la normale de la surface de l'objet
      $ecran[i][j] \leftarrow ecran[i][j] + (1 - reflection_{obj}) \times prop_{intensite} \times couleur_{obj}$
      $prop_{intensite} \leftarrow reflection_{obj} \times prop_{intensite}$
    sinon
      sortie de boucle

Algorithme 1 : Lancer de rayon

2.3 Intersections entre les rayons et les objets

L'avantage du lancer de rayon est sa capacité à trouver une intersection avec tout objet ayant une description mathématique, contrairement à d'autres méthodes nécessitant de transformer les surfaces en ensembles de triangles pour les projeter (comme la rastérisation très utilisée en synthèse d'image). Ainsi, il permet de calculer les formes et normales exactes des objets, sans approximation. Pour ce type d'objets (sphère, cylindre, spline...), cela permet d'obtenir de meilleurs résultats de forme et de couleur.

Prenons par exemple une sphère [1] de centre $\mathcal{C}$ et de rayon r et d'un rayon d'origine $\mathcal{O}$ et de direction $\vec{u}$:
Pour trouver l'intersection entre le rayon et l'objet sphère, on cherche X tel que :

$$\begin{cases} \|X - \mathcal{C}\| = r \\ X = \mathcal{O} + t \cdot \vec{u}, \quad t \in \mathbb{R} \end{cases}$$

$\Rightarrow \|\mathcal{O} + t \cdot \vec{u} - \mathcal{C}\|^2 = r^2$
$\Rightarrow \langle \mathcal{O} + t \cdot \vec{u} - \mathcal{C}, \mathcal{O} + t \cdot \vec{u} - \mathcal{C} \rangle = r^2$
$\Rightarrow \|\vec{u}\|^2 t^2 + 2t\langle \vec{u}, \mathcal{O} - \mathcal{C} \rangle + \|\mathcal{O} - \mathcal{C}\|^2 - r^2 = 0$

Ce qui correspond à un polynôme du second degré avec $\Delta = 4\langle \vec{u}, \mathcal{O} - \mathcal{C} \rangle^2 - 4\|\vec{u}\|^2 \times (\|\mathcal{O} - \mathcal{C}\|^2 - r^2)$
Ainsi, si $\Delta \geq 0$, avec t_1 et t_2 les solutions du polynôme, $t = min(t_1, t_2)$ ce qui nous permet de trouver l'intersection efficacement et exactement. La normale passant par l'intersection X est alors $\overrightarrow{\mathcal{C}X}$.

3 Calcul de couleur

3.1 BRDF

Une **BRDF**, c'est-à-dire une fonction de réflectivité bidirectionnelle, est une fonction de distribution probabiliste dépendant de l'angle solide $d\omega_i$ [3] que l'on observe, caractérisant la lumière réfléchie par rapport à la lumière incidente. On calcule cette fonction selon la description géométrique et les propriétés physiques de la surface [3]. Ces modèles sont souvent très complexes et donc souvent approximés, comme avec la méthode de Blinn-Phong, très utilisée en rendu 3D pour son réalisme et sa simplicité.

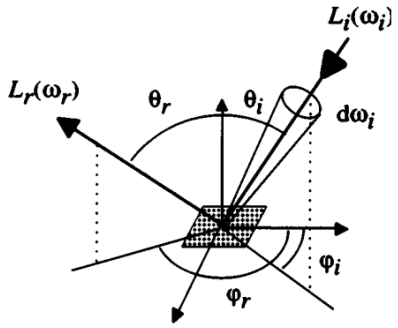


FIGURE 3 – Fonction de réflectivité bidirectionnelle, extrait de [3]

3.2 Méthode de Blinn-Phong

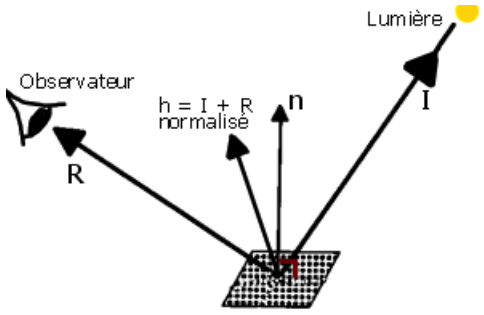


FIGURE 4 – Vecteurs impliqués dans la méthode de Blinn-Phong

La méthode de Blinn-Phong [5] calcule la couleur renvoyée par une surface en divisant la lumière émise et réfléchie en trois parties [5] : la couleur ambiante, c'est-à-dire la couleur propre de l'objet en l'absence de lumière extérieure, la couleur diffuse, plus proche de la couleur apparente de l'objet en présence de lumière et la couleur spéculaire qui est la couleur des reflets de la lumière.

Couleur ambiante : $\text{ambiante}_{\text{objet}} \times \text{ambiante}_{\text{lumiere}}$
Couleur diffuse : $\text{diffuse}_{\text{objet}} \times \text{diffuse}_{\text{lumiere}} \times \vec{I} \cdot \vec{n}$
Couleur spéculaire : $\text{speculaire}_{\text{objet}} \times \text{speculaire}_{\text{lumiere}} \times (\vec{n} \cdot \vec{h})^{\gamma}$ avec $\gamma = \text{brillance}_{\text{objet}}/4$

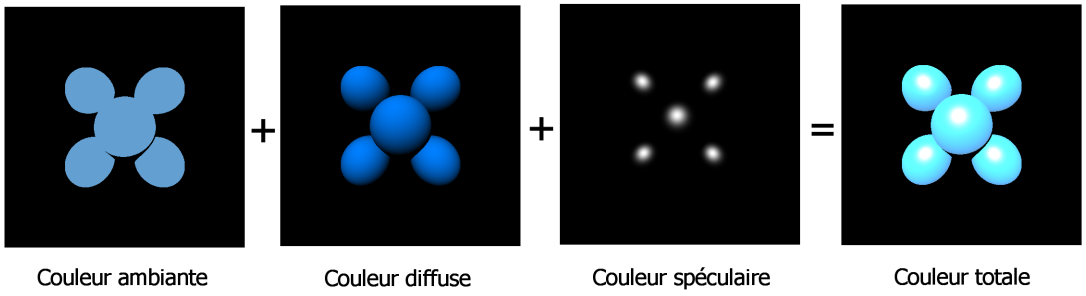


FIGURE 5 – Décomposition de la couleur pour Blinn-Phong, inspiré de Wikipédia

4 Complexité

4.1 Terminaison

Nous pouvons en premier lieu remarquer que l'algorithme décrit précédemment ne termine pas forcément, dans le cas où la scène contient des miroirs parfaits (de coefficient de réflexion 1) qui se réfléchissent

les uns dans les autres. Ainsi, nous nous limitons au problème d'une scène ne contenant aucun miroir parfait. La décroissance de $prop_{intensite}$ est alors sous-géométrique de raison $a < 1$, le coefficient de réflexion maximal d'un objet, et en négligeant les approximations sur les flottants ($\simeq 10^{-15}$ en Python), il y a au maximum $\lceil log_a(ind_{ref}) \rceil$ itérations.

4.2 Analyse de complexité

En notant $\alpha = f(ind_{ref}, scene)$ le nombre de réflexions maximal d'un rayon, l'algorithme a une complexité $\mathcal{O}(l \times c \times obj^{\alpha+1})$.

α est fortement dépendant de la scène, mais est décroissant pour ind_{ref} [6] et croissant en moyenne pour le nombre d'objets, en particulier si ceux-ci ont un fort coefficient de réflexion ou si deux objets réfléchissants sont placés l'un en face de l'autre.

(a) Réflexions, $ind_{ref} = 0.05$; 123 secondes de calcul (b) Sans réflexion, $ind_{ref} = 1$; 106 secondes de calcul

FIGURE 6 – Comparaison des images et temps de calcul pour ind_{ref}

En faisant le calcul pour la scène étudiée, avec différents nombres d'objets, on obtient le graphe 7, pour lequel une approximation linéaire semble adéquate.

Donc, dans la scène choisie, le temps de calcul est globalement une fonction linéaire du nombre d'objets. Par conséquent, α est négligeable, donc c'est l'ajout d'objets qui est le plus coûteux.

5 Améliorations

Cette dernière remarque amène à penser qu'une heuristique sur le choix des objets à intersecter peut permettre de grandement gagner en temps de calcul [4]. Pour cela, nous proposons d'utiliser une partition binaire de l'espace.

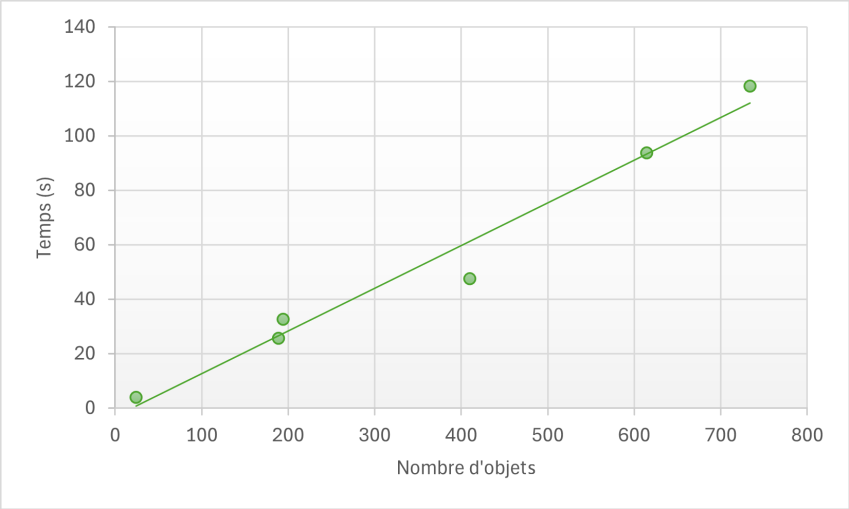


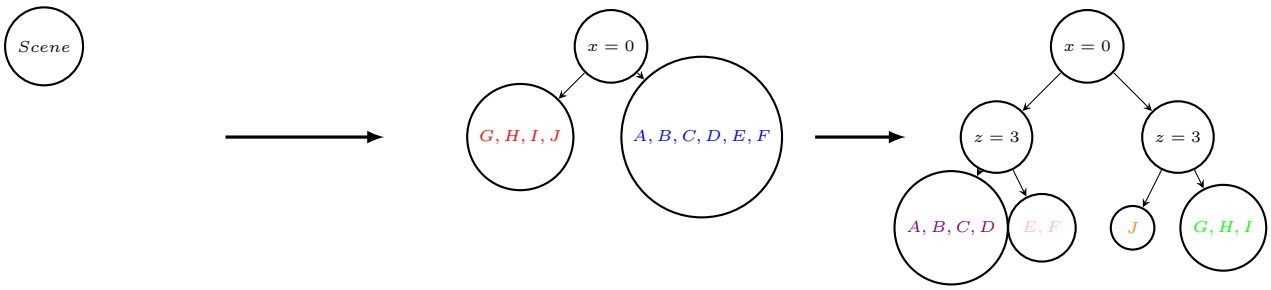
FIGURE 7 – Approximation linéaire du temps de calcul en fonction du nombre d'objets

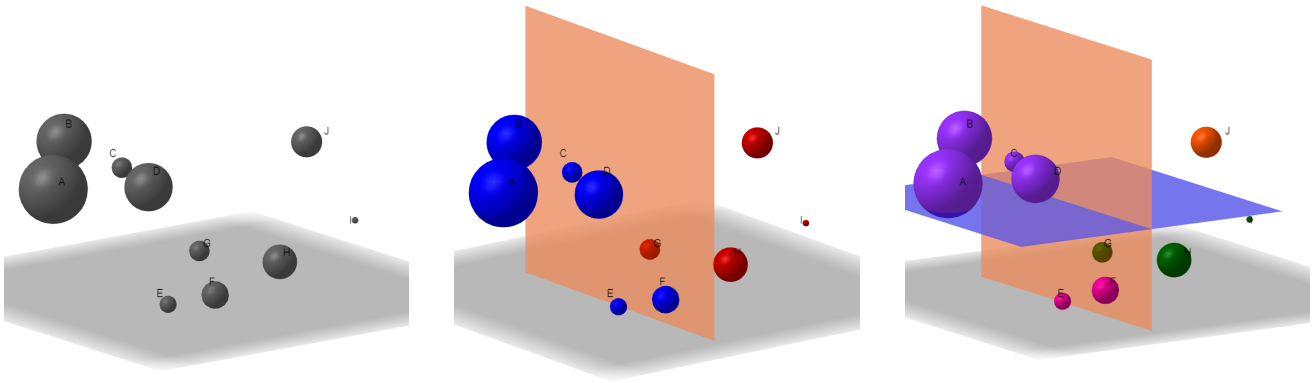
5.1 Construction de l'arbre binaire

Pour choisir les objets à tester pour l'intersection il est possible de subdiviser l'espace et de construire un arbre 3-dimensionnel à partir des subdivision en suivant l'exemple 8. L'arbre est construit récursivement à partir de la liste d'objets :

Construction
Données : liste d'objets lst, profondeur p
Résultat : arbre binaire $\mathcal{A}$
si $|lst| = 1$ **alors**
  Renvoyer Feuille(lst)
sinon
  $c \leftarrow$ médiane de la (p mod 3)ème coordonnée des objets de lst
  Renvoyer Nœud($coord_{pmod3} = c$, Construction($lst_<, p+1$), Construction($lst_>, p+1$))

Algorithme 2 : Construction de l'arbre 3-dimensionnel





(a) Étape 1 (b) Étape 2 (c) Étape 3

FIGURE 8 – Exemple de partition binaire de l'espace

5.2 Utilisation de la partition binaire de l'espace dans le lancer de rayon

Grâce à la partition binaire de l'espace, les objets sont placés dans l'arbre, ce qui permet d'éliminer rapidement un enfant si le rayon n'intersecte pas l'espace ou si un autre objet en amont du rayon a déjà été trouvé. Ainsi, il est possible de remplacer l'itération sur chaque objet par :

Parcours
Données : Arbre binaire $\mathcal{A}$, rayon $\overrightarrow{vec}, x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max}$
`// Conditions donnant le pavé dans lequel on cherche l'intersection`
Résultat : objet, distance
si $\mathcal{A}$ *est une feuille* **alors**
  Renvoyer le calcul d'intersection sur l'unique objet de $\mathcal{A}$
sinon
  Soit $\mathcal{A} = \text{Nœud}(u = c, \mathcal{L}, \mathcal{R})$
  `// plan avec u = x, y ou z`
  si $\overrightarrow{vec}$ *intersecte le pavé droit formé par les conditions* **alors**
    si $origine_{vec}.u < c$ **alors**
      $obj, dist \leftarrow \text{Parcours}(\mathcal{L}, \overrightarrow{vec}, u_{max}$ remplacé par $c)$
      si $obj \neq None$ **alors**
        Renvoyer $obj, dist$
      sinon
        Renvoyer $\text{Parcours}(\mathcal{R}, \overrightarrow{vec}, u_{min}$ remplacé par $c)$
    sinon
      $obj, dist \leftarrow \text{Parcours}(\mathcal{R}, \overrightarrow{vec}, u_{min}$ remplacé par $c)$
      si $obj \neq None$ **alors**
        Renvoyer $obj, dist$
      sinon
        Renvoyer $\text{Parcours}(\mathcal{L}, \overrightarrow{vec}, u_{max}$ remplacé par $c)$
  Renvoyer $None, +\infty$

Algorithme 3 : Parcours de l'arbre pour trouver l'intersection la plus proche

5.3 Correction

Terminaison : Parcours d'arbre fini

Correction partielle : Par induction structurelle sur l'arbre $\mathcal{A}$.

— Si $\mathcal{A}$ est une feuille, l'intersection avec l'objet contenu dans $\mathcal{A}$ est la seule intersection possible

— Soit $\mathcal{A} = \text{Nœud}(u = c, \mathcal{L}, \mathcal{R})$ avec $\mathcal{L}$ et $\mathcal{R}$ tels que Parcours est correct.

— Si $\overrightarrow{vec}$ n'intersecte pas le pavé droit formé par les conditions, il ne peut intersecter les objets contenus dans cette partie de l'espace.

— Sinon, puisque $\overrightarrow{vec}$ n'intersecte le plan $u = c$ qu'une seule fois, en supposant sans perte de généralité l'origine du rayon avec $u < c$, on observe le rayon en amont de l'intersection en parcourant $\mathcal{R}$, correspondant aux objets avec $u < c$.
Si $obj \neq None$, on a rencontré un objet en amont, donc tout objet de $\mathcal{R}$ est plus éloigné de l'origine et il n'est pas nécessaire de parcourir l'autre enfant.

5.4 Résultats 9

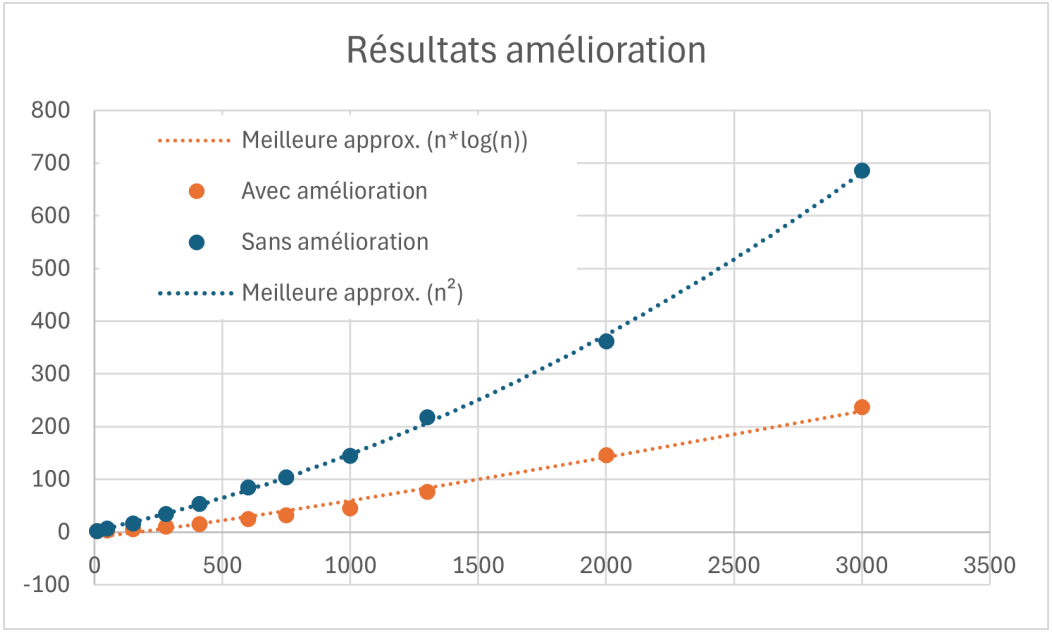


FIGURE 9 – Comparaison des temps avec et sans amélioration sur des scènes aléatoires

Références

[1] O AFLAK. Ray tracing from scratch in python.

[2] A APPEL. Some techniques for shading machine renderings of solids. pages 37–45.

[3] M. F. COHEN and J. F. WALLACE. Radiosity and realistic image synthesis.

[4] J. D. MACDONALD and K. S. BOOTH. Heuristics for ray tracing using space subdivision. pages 153–166.

[5] A. ÖZTÜRK, A. BILGILI, and M. KURT. Polynomial approximation of blinn-phong model. pages 55–61.

A Code de rendu 3D

```
class Noeud :
    """Classe correspondant a un arbre, pour implementer l'arbre k-d"""
    def __init__(self,feuille,u,plan,obj,gauche,droit):
        self.feuille = feuille
        self.axe = u
        self.val = plan
        self.obj = obj
        self.gauche = gauche
        self.droit = droit

def extrema(coord,maxi,liste_obj):
    """Trouve les coordonnees extremales d'une liste d'objets"""
    tmp = -maxi*math.inf
    for obj in liste_obj :
        if type(obj) == Sphere :
            if maxi*obj.centre[coord] + maxi*obj.rayon > maxi*tmp :
                tmp = obj.centre[coord] + obj.rayon
        if type(obj) == Triangle :
            if maxi*obj.p1[coord] > maxi*tmp :
                tmp = obj.p1[coord]
            if maxi*obj.p2[coord] > maxi*tmp :
                tmp = obj.p2[coord]
            if maxi*obj.p3[coord] > maxi*tmp :
                tmp = obj.p3[coord]
    return tmp

def construit_arbre(liste_obj,p):
    if len(liste_obj) == 0 :
        return None
    if len(liste_obj) == 1 :
        return Noeud(True,None,None,liste_obj[0],None,None)
    coord = p%3
```

```
    moy = 0
    for obj in liste_obj :
        moy += obj.centre[coord]
    moy = moy/(len(liste_obj))
    lst_gauche = []
    lst_droite = []
    for obj in liste_obj :
        if obj.centre[coord] < moy :
            lst_gauche.append(obj)
        else :
            lst_droite.append(obj)
    return Noeud(False,coord,moy,None,construit_arbre(lst_gauche,p+1),
       construit_arbre(lst_droite,p+1))

class Vecteur :
    """Description d'un vecteur avec ses coordonnees x,y et z et les
       methodes classiques associees
    Un point sera considere comme un vecteur depuis l'origine"""

class Ray :
    def __init__(self,o,v):
        self.origine = o
        self.direction = v.normalise()

class Sphere :
    def __init__(self,c,r,ca,cd,cs,s,ref):
        self.centre = c
        self.rayon = r
        self.coul_amb = ca
        self.coul_diff = cd
        self.coul_spec = cs
        self.shin = s
        self.reflection = ref

    def intersection(self,ray,scene):
        b = 2*(ray.direction.scalaire(ray.origine-self.centre))
        c = (ray.origine-self.centre).norme()**2 - self.rayon**2
        delta = b*b - 4*c
        if delta > 0 :
            r1 = (-b-math.sqrt(delta))/2
            r2 = (-b+math.sqrt(delta))/2
            if r1 > 0 and r2 > 0 :
                return min(r1,r2)
        return None

    def normale(self,point):
```

```
        return (point-self.centre).normalise()

class Triangle :
    def __init__(self,p1,p2,p3,ca,cd,cs,s,ref):
        self.p1 = p1
        self.p2 = p2
        self.p3 = p3
        self.coul_amb = ca
        self.coul_diff = cd
        self.coul_spec = cs
        self.shin = s
        self.reflection = ref
        self.n = self.normale(None)
        self.p1p2 = (self.p2-self.p1).normalise()
        self.p2p3 = (self.p3-self.p2).normalise()
        self.p1p3 = (self.p1-self.p3).normalise()
        self.centre = (p1+p2+p3)*(1/3)
        self.ang1 = self.p1p2.scalaire((self.p3-self.p1).normalise())
        self.ang2 = (self.p1-self.p2).normalise().scalaire((self.p3-self
           .p2).normalise())
        self.ang3 = (self.p1-self.p3).normalise().scalaire((self.p2-self
           .p3).normalise())

    def intersection(self,ray,scene):
        if (self.n.scalaire(scene.normale) < 0):
            self.n*=-1.0
        if ray.direction.scalaire(self.n) == 0 :
            return None

        k = self.n.x*self.p1.x + self.n.y*self.p1.y + self.n.z*self.p1.z
        somc = k - self.n.x*ray.origine.x - self.n.y*ray.origine.y -
           self.n.z*ray.origine.z
        somt = self.n.x*ray.direction.x + self.n.y*ray.direction.y +
           self.n.z*ray.direction.z
        t = somc/somt
        if t<= 0 :
            return None
        p = ray.origine+ray.direction*t
        if self.appartient(p) :
            return t
        return None

    def appartient(self,p):
        angp1 = self.p1p2.scalaire((p-self.p1).normalise())
        angp2 = self.p2p3.scalaire((p-self.p2).normalise())
        angp3 = self.p1p3.scalaire((p-self.p3).normalise())
```

```
        return angp1>=self.ang1 and angp2>=self.ang2 and angp3>=self.
            ang3

    def normale(self,_):
        return ((self.p3-self.p2).vectoriel(self.p2-self.p1)).normalise
            ()

class Scene :
    """Definition d'une scene comme on l'a definit dnas l'introduction
        """

def obj_proche(ray,scene):
    """Trouve l'objet le plus proche naivement"""
    obj_proche = None
    t = math.inf
    for obj in scene.objet :
        t_int = obj.intersection(ray,scene)
        if t_int != None and t_int<t:
            t = t_int
            obj_proche = obj
    return obj_proche,t

def touche(ray,lst_conditions):
    """Determine si un rayon touche le pave droit forme par
        lst_conditions"""
    xmi = lst_conditions[0]
    ymi = lst_conditions[1]
    zmi = lst_conditions[2]
    xma = lst_conditions[3]
    yma = lst_conditions[4]
    zma = lst_conditions[5]
    point_min = ray.origine + ray.direction*((zmi-ray.origine.z)/ray.
        direction.z)
    point_max = ray.origine + ray.direction*((zma-ray.origine.z)/ray.
        direction.z)
    condition_y1 = (point_min.y>yma and point_max.y>yma)
    condition_y2 = (point_max.y<ymi and point_min.y<ymi)
    condition_x1 = (point_min.x>xma and point_max.x>xma)
    condition_x2 = (point_max.x<xmi and point_min.x<xmi)
    return (condition_y1 or condition_y2 or condition_x1 or condition_x2
        )

def intersection(ray,scene,arbre,lst_conditions):
    """intersection avec heuristique sur l'ordre de traitement des
        objets"""
    if arbre == None :
```

```
            return None, math.inf
        if arbre.feuille :
            t = arbre.obj.intersection(ray,scene)
            if t != None :
                return arbre.obj,t
            else :
                return None,math.inf
        if touche(ray,lst_conditions) :
            if ray.origine[arbre.axe] <= arbre.val :
                lst_new = lst_conditions.copy()
                lst_new[3+arbre.axe] = arbre.val
                obj,dist = intersection(ray,scene,arbre.gauche,lst_new)
                if obj != None :
                    return obj,dist
                else :
                    lst_new = lst_conditions.copy()
                    lst_new[arbre.axe] = arbre.val
                    obj,dist = intersection(ray,scene,arbre.droit,lst_new)
                    return obj,dist
            else :
                lst_new = lst_conditions.copy()
                lst_new[arbre.axe] = arbre.val
                obj,dist = intersection(ray,scene,arbre.droit,lst_new)
                if obj != None :
                    return obj,dist
                else :
                    lst_new = lst_conditions.copy()
                    lst_new[3+arbre.axe] = arbre.val
                    obj,dist = intersection(ray,scene,arbre.gauche,lst_new)
                    return obj,dist
        else :
            return None, math.inf

    def illumination(scene,obj,L,N,V):
        amb = obj.coul_amb*scene.amb
        diff = obj.coul_diff*scene.diff*L.scalaire(N)
        H = (L + V).normalise()
        spec = obj.coul_spec*scene.spec*(N.scalaire(H))**(obj.shin/4)
        return amb+diff+spec

    def raytrace(i,j,scene,indice_reflexion_max) :
        o = scene.trouve_position(i,j)
        d = o-scene.camera
        ray = Ray(o,d)
        couleur = np.zeros((3))
        reflection = 1
```

```
    while reflection>=indice_reflexion_max :
        n_obj, t = obj_proche(ray,scene)
        if n_obj != None :
            p_int = ray.origine+ray.direction*t
            N = n_obj.normale(p_int)
            _, dist_av_lum = obj_proche(Ray(p_int+N*1e-10,scene.lumiere-
               p_int),scene)
            if dist_av_lum>=(scene.lumiere-p_int).norme() :
                L = (scene.lumiere-p_int).normalise()
                V = (scene.camera-p_int).normalise()
                ill = illumination(scene,n_obj,L,N,V)
                couleur += (1-n_obj.reflection)*reflection*ill
                reflection *= n_obj.reflection
                o = p_int+N*1e-3
                d = ray.direction.reflechi(N)
                ray = Ray(o,d)
            else :
                break
        else :
            break
    return np.clip(couleur,0,1)

def raytracing(nb_pixel_largeur,nb_scene,scene,indice_reflexion_max):
    scene.change_largeur(nb_pixel_largeur)
    screen = np.zeros((scene.height,scene.width,3))
    for i in range(scene.height):
        for j in range(scene.width):
            screen[i,j] = raytrace(i,j,scene,indice_reflexion_max)
    return screen

def new_raytrace(i,j,scene,arbre,conditions,indice_reflexion_max) :
    """Avec amelioration"""
    o = scene.trouve_position(i,j)
    d = o-scene.camera
    ray = Ray(o,d)
    couleur = np.zeros((3))
    reflection = 1
    while reflection>=indice_reflexion_max :
        n_obj, t = intersection(ray,scene,arbre,conditions)
        if n_obj != None :
            p_int = ray.origine+ray.direction*t
            N = n_obj.normale(p_int)
            _, dist_av_lum = obj_proche(Ray(p_int+N*1e-10,scene.lumiere-
               p_int),scene)
            if dist_av_lum>=(scene.lumiere-p_int).norme() :
                L = (scene.lumiere-p_int).normalise()
```

```
                V = (scene.camera-p_int).normalise()
                ill = illumination(scene,n_obj,L,N,V)
                couleur += (1-n_obj.reflection)*reflection*ill
                reflection *= n_obj.reflection
                o = p_int+N*1e-3
                d = ray.direction.reflechi(N)
                ray = Ray(o,d)
            else :
                break
        else :
            break
    return np.clip(couleur,0,1)

def new_raytracing(nb_pixel_largeur,nb_scene,scene,arbre,conditions,
   indice_reflexion_max):
    """Avec Amelioration"""
    scene.change_largeur(nb_pixel_largeur)
    screen = np.zeros((scene.height,scene.width,3))
    for i in range(scene.height):
        for j in range(scene.width):
            screen[i,j] = new_raytrace(i,j,scene,arbre,conditions,
               indice_reflexion_max)
    return screen
```