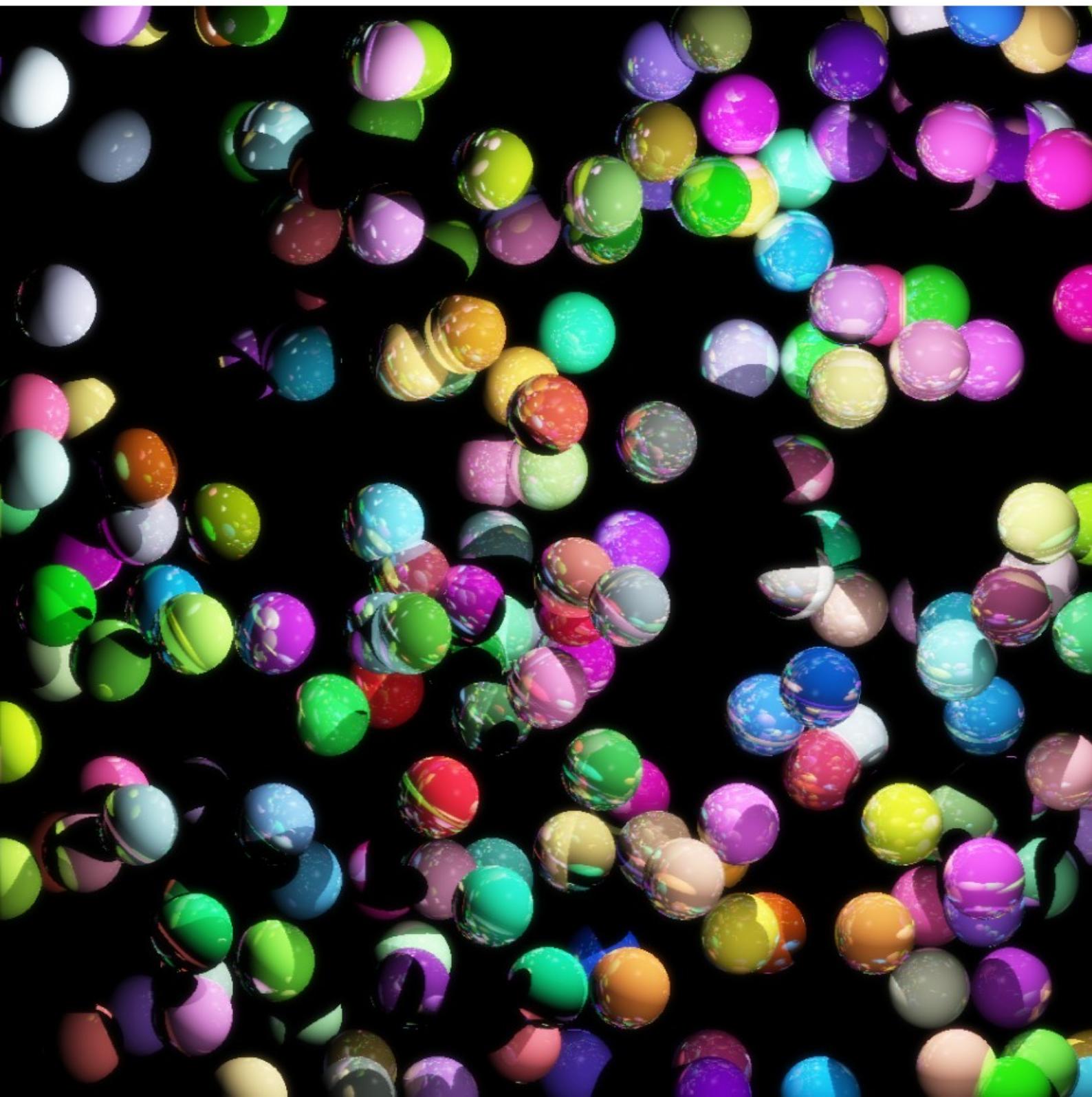


Ray Tracing

MANGNAN Valentin
CHANDEBOIS Anthony
Projet L2 Informatique



1 Sommaire

Avant d'entrer dans le vif du sujet, voici un bref petit sommaire. Nous commencerons, par rappeler quelques généralités sur le ray tracing. Dans la partie suivante vous pourrez trouver un algorithme succinct du fonctionnement de notre programme, une notice de fonctionnement, ainsi que la raison de l'utilisation de certaines librairies non-standards. La troisième partie s'intéresse en profondeur au programme. Vous y trouverez un détail sur les calculs d'intersection, l'implémentation des différents concepts du ray tracing, le calcul de la lumière ainsi que des notions sur la transparence et l'utilisations de matériaux. La partie suivante aborde quelques problèmes constatés lors de l'élaboration du projet. Suivent la bibliographie et les annexes (un schéma "UML" et un exemple simple de fichier de configuration). Bonne lecture!

2 Introduction

2.1 But du devoir et réalisation

Le but de notre devoir est de créer un moteur de lancer de rayons via le langage de programmation Python. Le lancer de rayon, en anglais le ray tracing, est une méthode permettant d'obtenir des images de synthèse photo-réalistes en reproduisant les phénomènes physiques que sont la réflexion et la réfraction. Cette méthode reconstitue le parcours inverse de la lumière depuis la caméra en direction des sources lumineuses. On dispose d'une caméra, d'un ou plusieurs objets, de lumière(s) et d'une scène. On calcule ensuite les éclairages de la caméra vers les objets, puis vers les lumières. La lumière va de la scène vers l'œil.

Nous vous invitons à garder à l'esprit - à mesure que vous lirez le rapport - que tout ceci est le fruit de près de deux semaines de programmation intensive. Chaque modification du programme ayant engendré un test, il n'est pas impossible que nous ayons procédé à plus de 500 tests. Ceci ayant été dit, revoyons rapidement le principe du ray tracing.

2.2 Principe

Le principe est simple, pour chaque pixel de l'image générée, on lance un rayon depuis la caméra dans la scène. On lance ensuite des rayons depuis le point d'impact en direction des différentes sources de lumière afin de déterminer sa luminosité, c'est-à-dire si l'objet est éclairé ou non. La couleur finale du pixel dépend de la luminosité, des propriétés de l'objet (couleur, rugosité, etc.) ainsi que d'autre informations telles que la transparence, la réflexion, etc. En voici une illustration :

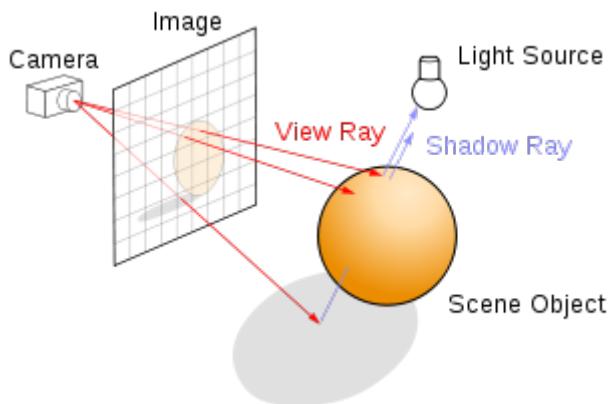


Figure 1 : Illustration du lancer de rayon

3 Le Programme

Pour mettre en place ce projet nous avons décidé de découper le programme en plusieurs parties. En effet, nous avons partagé le code en 13 fichiers .py afin de ne pas nous perdre dans le script (l'ensemble des fichiers .py ainsi que leur fonctions sont expliqués dans le tableau ci-dessous).

3.1 Algorithme

Voici un algorithme simple en pseudo-code expliquant le fonctionnement du programme :

Pour chaque pixel en largeur faire:

 Pour chaque pixel en longueur faire:

 Afficher le pourcentage

 Lancer un rayon

 Pour chaque objet dans la liste d'objets faire:

 Calculer l'intersection et la normale

 Chercher l'objet le plus proche de la caméra

 Pour chaque lumière dans la liste des lumières faire:

 Calculer les composantes spéculaires, ambiantes et diffuses de la lumière

 Évaluer l'ombre

 Lancer des rayons réfractés

 Évaluer la transparence

 Faire la somme des composantes en fonction des coefficients et des couleurs

 Remplir le pixel avec la couleur calculée

Renvoyer le tableau et l'enregistrer/l'afficher

3.2 Mode d'emploi

Le programme s'utilise de la manière suivante :

- Ouvrir le fichier config.py contenu dans l'archive du programme
- Paramétriser les matériaux en choisissant la couleur diffuse, la couleur spéculaire, la couleur ambiante et le damier.
- Définir les lumières
- Paramétriser les sphères, les plans
- Définir ou choisir une caméra.
- Donner les paramètres voulus à la scène
- Spécifier si l'on veut sauvegarder le fichier de sortie en mettant save_file=True ou False en fonction de notre choix.

3.3 Tableau récapitulatif des fichiers .py utilisés

Nom du fichier .py	Fonction du fichier .py
camera.py	Définit la classe <code>Camera</code> et ses propriétés (position, direction, taille de l'écran en pixels, vecteurs composant l'écran, la méthode <code>lancer_rayon()</code>)
chose.py	Classe abstraite servant à encapsuler un matériau et un point xc.
colors.py	Définit les couleurs selon des vecteurs. Ces couleurs serviront pour la composition des matériaux.
config.py	Configure les paramètres tels que les positions des lumières, des plans, des sphères, de la caméra, de la scène, etc.
lumier.py	Définit la classe <code>Lumiere</code> composée de la méthode <code>intersection()</code> permettant de trouver les lumières locales de créer des ombres, la réflexion, etc.
materl.py	Définit la classe <code>Material</code> et ses propriétés (telles que la couleur diffuse, la couleur ambiante, la couleur spéculaire, la couleur émise, etc.)
parall.py	Définit la classe <code>Parallelogramme</code> ainsi que ses propriétés (position, vecteurs directeurs) ainsi que la méthode <code>intersection()</code> qui détermine la position de l'intersection d'un rayon et d'un parallélogramme.
plaaan.py	Définit la classe <code>Plan</code> ainsi que ses propriétés (position, vecteurs directeurs) ainsi que la méthode <code>intersection()</code> qui détermine la position de l'intersection d'un rayon et d'un plan
sceene.py	Définit la classe <code>Scene</code> et ses propriétés (liste de caméras, de lumières, de choses, ainsi que la méthode <code>lancer_rayon()</code>)
sortie.py	Importe les module nécessaires (<code>config.py</code> , <code>matplotlib.pyplot</code> , <code>matplotlib.image</code>) et exécute le programme. Une image nous est retournée.
sphere.py	Définit la classe <code>Sphere</code> , ses propriétés (<code>xc</code> , <code>material</code> , <code>r</code>) et une méthode.
triang.py	Définit la classe <code>Triangle</code> , ses propriétés (<code>xc</code> , <code>material</code> , <code>u</code> , <code>v</code>) et une méthode.
vector.py	Définit l'objet <code>Vec</code> , ses propriétés (<code>x</code> , <code>y</code> , <code>z</code>), des méthodes d'opérations sur les vecteurs ainsi que la classe <code>Pvec</code> et ses propriétés (<code>x0</code> , <code>d</code> , <code>objet dans</code>).

3.4 Choix de programmation

Afin de réaliser ce projet nous avons favorisé certaines librairies. Pour commencer, nous avons choisi d'utiliser `matplotlib` car nous avons trouvé ce module plus simple que le module `Tkinter`. En effet, `matplotlib` est beaucoup plus pratique et nous permet d'afficher l'image dans son ensemble d'un coup. De plus, celui-ci nous permettait une meilleure gestion des vecteurs RGB.

Nous avons ensuite décidé d'utiliser le module `numpy`, qui est une librairie très puissante puisqu'elle nous permet d'utiliser des tableaux. Nous avons pris le module `imshow` qui permet de créer une image à partir de tableaux créés via la librairie `numpy`. Un autre choix nécessaire était la création d'une classe `Pvec` nous permettant de stocker des points, directions et autres à tout moment. Cela nous a permis de ne pas avoir à utiliser des tuples qui se transportent d'une fonction à l'autre.

4 Les classes

Il va sans dire qu'un programme non orienté objet aurait été inacceptable (vu l'intitulé de l'unité), il était normale que nous insérions des classes et des objets dans notre code Python. Nous aborderons en détails quelques classes à commencer par la scène.

4.1 La scène

Elle est agrégée et *elle agrège* respectivement les lumières, les caméras et les choses. En effet, la scène S possède une caméra C comme attribut et la caméra C a une scène S en attribut. De ce fait, nous pouvons passer de l'un à l'autre sans que cela ne nous pose de problème. Elle se compose évidemment de différents objets définis dans le `config.py`.

4.2 La caméra

4.2.1 Position et direction

La classe camera est le noyau du raytracing. Elle est l'interface entre le monde en 3 dimensions (défini par des équations implicites) et l'image en 2 dimensions qu'on en fera. Par conséquent, il est essentiel de maîtriser le placement de la camera, sa direction, son champ d'impact. Bien qu'au départ il nous semblait qu'une camera automatiquement réglée soit l'idéal, il est bien plus intéressant de pouvoir - et d'offrir cette possibilité à l'utilisateur - la déplacer dans l'espace librement. Sachant cela, il nous a été conseillé de paramétrier la caméra ainsi :

- par sa position dans l'espace (un vecteur représentant un point) : `x0`
- par la direction de sa focale (un vecteur unitaire) : `d0`
- par la direction du haut de la camera (sans quoi elle pouvait tourner librement sur son axe) : `up`. Précisons en même temps que `up` est bien orthogonale à la direction `d0` du fait d'un petit calcul ($up=up-(up.dot(d0))*d0$) pour enlever la composante perpendiculaire de `up`.

4.2.2 L'écran

La partie 3D de la caméra étant paramétrée, considérons l'implémentation de l'écran. Comme le suggère la plupart des figures de ce présent rapport, un écran virtuel s'interpose entre la caméra et la scène. Cet écran est représenté par différents paramètres :

- une ouverture de focale (matérialisée par les composantes `h_ouv` en hauteur et `w_ouv` en largeur)
- une résolution d'image (composantes `h_size` et `w_size`). Rappelons en effet que le but du raytracing est de produire une image. Ces paramètres étant le nombre de pixels en hauteur (resp. en largeur) de notre image.
- un array numpy associé (d'une taille de `h_size` par `w_size`) de dimension 3 : coordonnées-X, coordonnées-Y et couleur (RGB)

Après avoir paramétré l'écran, et précédemment, la caméra à proprement parler, où se situent les rayons ? Il va de soi qu'en général, les rayons viennent de la source de lumière jusqu'à l'œil. Or ici, c'est précisément le contraire qu'on effectue. Comment cela ? C'est l'œil (autrement dit la caméra) qui lance ses propres rayons vers les objets puis les sources de lumière. Comme tout lancer de rayon, une méthode `intersection` de la caméra se charge de

1. Balayer l'écran en largeur et en hauteur
2. Récupérer des informations sur l'objet le plus proche de la caméra
3. Calculer son rapport à la lumière
4. Évaluer la transparence de l'objet

Bien sûr, des questions se posent par rapport à cela. Comment fonctionne le balayage de l'écran ? Comment évalue-t-on la proximité d'un objet ? Comment la lumière est-elle évaluée ? Enfin, comment procéder pour la transparence.

4.2.3 Le balayage

Dans un premier temps, examinons succinctement le balayage. L'écran est symbolisé par deux vecteurs X et Y . Pour donner plus de choix à l'utilisateur, il lui est proposé deux types de balayage - les balayages coniques et cylindriques.

Le balayage conique consiste à envoyer des signaux légèrement décalé les uns par rapport aux autres sur l'un des côtés de l'axe optique de la caméra. Comme ceci :

```
ray=Pvec(self.x0,
          (self.d0+(h*1.0*self.h_ouv/self.h)*self.Y+
           (w*1.0*self.w_ouv/self.w)*self.X).normalize())
```

En voici un illustration :

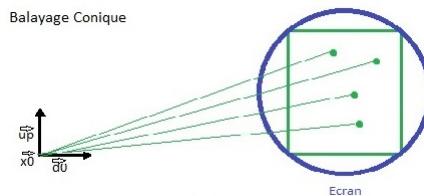


Figure 2 : Illustration du balayage conique.

Le balayage cylindrique consiste quant à lui à envoyer des signaux colinéaires au vecteur $d0$ qui est un axe de la caméra. Comme ceci :

```
ray=Pvec(self.x0+(h*1.0*self.h_ouv/self.h)*self.Y+
          (w*1.0*self.w_ouv/self.w)*self.X,
          self.d0)
```

En voici une illustration :

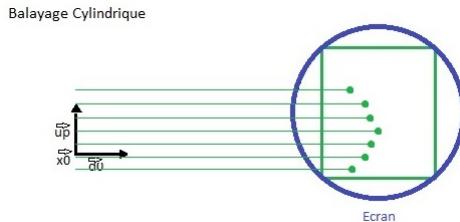


Figure 3 : Illustration du balayage cylindrique.

4.2.4 La transparence

Vu que la transparence n'est pas fonction de la lumière, il est normal qu'elle soit gérée par la caméra elle-même. On sait d'autre part que la transparence est extrêmement liée à la loi de Snell-Descartes, à savoir : $n_1 \times \sin(i_1) = n_2 \times \sin(i_2)$, où les i_k sont des angles entre le rayon et la normale, et les n_k sont les indices des milieux. Étant donné que la lumière se propage en ligne droite sur un même plan, il faut bien entendu faire la rotation du rayon sur un même plan. Ce plan est engendré par les vecteurs incidents (V - cette fois-ci, il s'agit du vecteur partant de la camera, vers l'objet) et normal à l'objet (N). L'équation de la normale de ce plan est :

```
Nplan=V.cross(N)
```

Il suffit d'effectuer la rotation de V selon $i_2 - i_1$, puis de lancer un rayon dans cette direction.

Calcul de i_1 et i_2

L'angle entre N et V (qui correspond à i_1 dans la loi de Snell-Descartes) vaut :

```
i1=acos(V.dot(N))
```

Si $|i_1| > \pi/2$, il n'y aura pas de transparence. Sinon, i_2 vaut :

```
i2=asin((n1/n2)*sin(i1))
```

Milieux

Il apparait rapidement qu'il faudra définir milieux et indices. Qu'est-ce qu'un milieu ? C'est l'intérieur d'un objet solide (donc un Plan n'est pas un milieu). Comment définir le milieu si deux solides sont l'un dans l'autre, et comment évaluer que le rayon traverse l'air (qui n'est évidemment pas un solide) ? Il y aurait bien des manières de faire, mais nous préfèrerons simplifier le code, voilà pourquoi nous nous contenterons d'utiliser une liste. Voici son fonctionnement :

- Quand on traverse un solide qui n'est pas dans la liste, on l'y rajoute.
- Quand on traverse un solide de la liste, on le supprime (on suppose pour cela que deux passages à travers un solide suffisent à le traverser entièrement).
- Quand la liste est vide, on considère que le milieu est l'air.

4.3 Les surfaces

Afin de mettre en place les différentes surfaces qu'il nous était possible de rencontrer, nous avons créé une classe **Chose** qui nous a permis d'encapsuler toutes les surfaces possibles. Chaque classe de surface a hérité de la classe **Chose**. De ce fait, voici une liste de surfaces que nous avons pu modéliser :

- le plan défini dans le fichier **plaaan.py** avec les paramètres d'initialisation **xc**, **material** et **D**.
- la sphère défini dans le fichier **sphere.py** avec les paramètres d'initialisation **xc**, **material** et **r** (rayon de la sphère).
- le parallélogramme défini dans le fichier **parall.py** avec les paramètres d'initialisation **xc** (point en haut à gauche du parallélogramme), **material**, **u** et **v** (vecteurs partant de **xc** définissant les côtés du parallélogramme).
- le triangle défini dans le fichier **triang.py** avec les paramètres d'initialisation **xc** (point en haut à gauche du triangle), **material**, **u** et **v** (vecteurs partant de **xc** définissant les côtés du triangle).

Les surfaces les plus intéressantes étant sphériques, nous parlerons de leur intersection avec un rayon.

4.3.1 Les sphères

Pour déterminer l'intersection entre une sphère et un rayon, il faut résoudre une fonction de t à l'aide de la position de la camera, de **d0**, du rayon de la sphère **r** ainsi que de la position **xc** de celle-ci. Nous commençons par calculer le discriminant de l'équation $(\text{ray}-\text{xc})=\text{r}**2$ (avec $\text{ray}=\text{x0}+\text{d}*\text{t}$) :

```
a=d**2  
b=2*(x0-xc).dot(d)  
c=(x0-xc)**2-r**2  
delta=b**2 -4*a*c
```

Il suffit ensuite d'évaluer le signe de **delta** et de calculer t de manière à garder le plus petit positif des résultats. Chaque intersection d'une figure avec un rayon renvoi un **Pvec**. C'est une encapsulation d'un vecteur point, d'un vecteur direction et de l'objet percuté en lui-même. Le procédé est le même pour toutes les surfaces.

4.4 Les matériaux

Dans notre programme, les matériaux sont aussi des points essentiels puisqu'ils nous permettent de définir les valeurs des différents types de couleurs que l'on peut rencontrer dans le lancer de rayon (couleur diffuse, couleur ambiante, couleur spéculaire, couleur émise) ainsi que les phénomènes physiques tels que la brillance qui est issue de la réflexion de la lumière, la transparence qui est due à la réfraction. Dans les paramètres se trouve la possibilité ou non de faire apparaître un damier en mettant la valeur **self.damier** à **True**. La transparence et l'indice de réfraction sont également paramétrables par l'utilisateur.

La méthode **get_difColor()** présente dans la classe **Material** est une méthode incontournable si l'utilisateur décide d'utiliser un damier. Elle va permettre de matérialiser le damier et ses différentes couleurs en utilisant la parité du point d'intersection du rayon de la caméra avec l'objet.

Parlons un peu du script :

```
def get_difColor(self, inter=None):
    if not self.damier:
        return self._difColor

    #Creation d'un damier pour les plans
    if inter<>None:
        L=4;p1=inter.x/L;p2=inter.y/L;p3=inter.z/L
        if ispair(p3):
            if (ispair(p1) and ispair(p2)) or ((not ispair(p1)) and (not ispair(p2))):
                return self._difColor
            else: return whi
        else: ...
```

Si nous regardons le bout de code ci-dessus, nous pouvons voir qu'il s'agit de modéliser le damier ainsi que ses couleurs. Comme nous l'avons déjà dit précédemment dans le rapport, nous utilisons la parité du point afin de déterminer la couleur des cases du damier. La partie du script allant de `if ispair(p3):...` jusqu'à la fin de la méthode `get_difColor()` va tester la parité des différentes composantes `p1`, `p2` et `p3` du point, et déterminer si la case est à colorier en blanc ou avec une autre couleur. Si les conditions contenues dans la partie `if ispair(p3):...` du code ne satisfont pas la suite du script alors la couleur de la case en cours de traitement mise à blanc par défaut. Voyez le schéma ci-dessous :

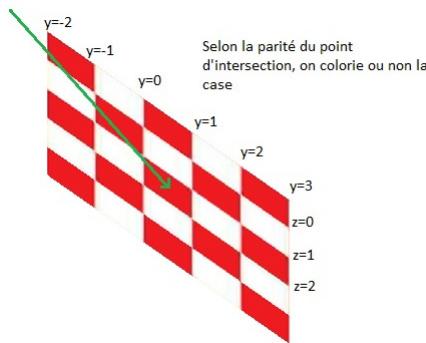


Figure 4 : Le fonctionnement du damier

4.5 La lumière

La lumière. L'essentiel de l'effet 3D. Abordons désormais quelques points concernant la lumière. Toutes les composantes suivantes sont ajoutées dans la mesure où leur somme est comprise entre 0 et 1. On procède ainsi pour chaque lumière, puis on fait la moyenne entre les lumières pour harmoniser l'éclairage. Avant d'entrer plus dans les détails, il est bon d'avoir à l'esprit ce petit schéma :

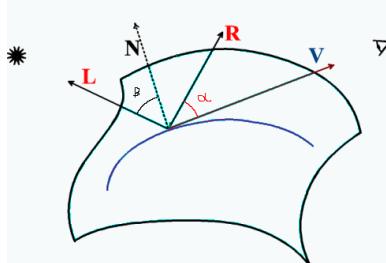


Figure 5 : Les différents vecteurs que nous utilisons

4.5.1 Lumière diffuse

Qu'est-ce que la lumière diffuse ? C'est une composante de la lumière qui est fonction de l'angle entre :

- la normale à l'objet, au point d'intersection avec le rayon
 - le vecteur normalisé, du point d'intersection, vers la source ponctuelle de lumière
- En pratique, on utilise le cosinus de cet angle ainsi :

```
cosalpha=N.dot(L)
if cosalpha>0:
    difColor=material.get_difColor(pvec.x0)
    I_local=difColor*cosalpha*self.kd
```

La première composante de cette lumière locale est donc le produit, du coefficient de lumière diffuse kd propre à la lumière, du cosinus, et de la couleur diffuse propre au matériau en surface. Les vecteurs étant normalisés, le produit vectoriel se prête bien au cosinus. Comme l'illustre la figure ci-dessous, la lumière diffuse joue un grand rôle dans l'effet 3D et l'ombrage.

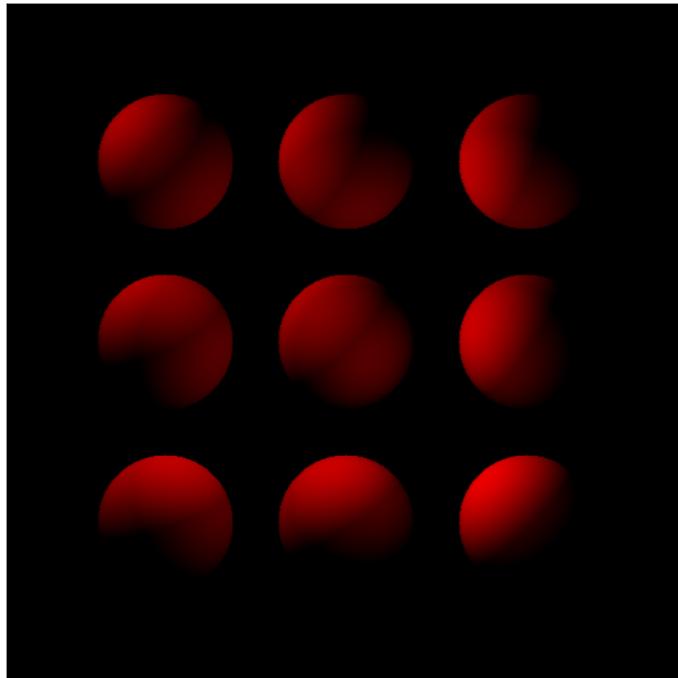


Figure 6 : Une illustration de la lumière diffuse

4.5.2 Lumière spéculaire

La lumière spéculaire, est bien souvent visible par les petites tâches lumineuses sur les surfaces éclairées. Comme la lumière diffuse, celle-ci est fonction d'un angle (entre le rayon réfléchi, et le vecteur partant de l'intersection allant vers la caméra). Celui-ci est élevé à la puissance k , la brillance de la surface (un nombre propre au matériau, compris entre 10 et 10000). On calcul cet angle ainsi :

```
cosbeta=R.dot(V)
...
speColor=material.get_speColor()
I_local+=self.ks*speColor*cosbeta**k
```

L'illustration ci-dessous montre clairement les petites tâches d'un matériau brillant. ks est la composante interne à la lumière qui définit sa tendance à créer la lumière spéculaire.

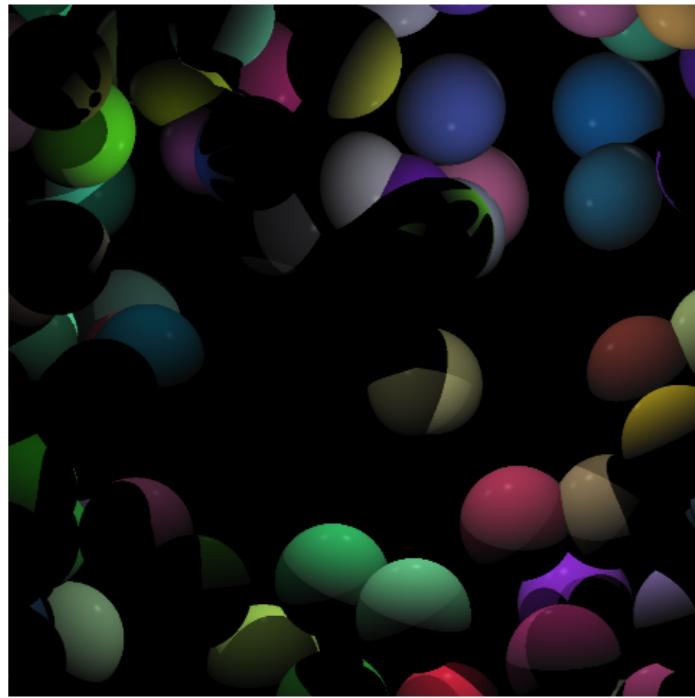


Figure 7 : Un exemple de l'effet de la composante spéculaire de la lumière

4.5.3 Lumière ambiante

C'est une simple petite contribution ajoutée à tous les objets éclairés pour être un peu plus lumineux. En général, elle n'excédera pas 0.1.

```
ambColor=material.get_ambColor()
I_local+=self.ka*ambColor
```

4.5.4 Ombres

Une image réaliste comprenant des lumières est forcément tempérée par des zones d'ombres. Le principe est très simple, une fois de plus. Il suffit de lancer un rayon vers la lumière elle-même, depuis l'intersection de l'objet (avec le rayon de la caméra). Si un objet se trouve entre la lumière et notre objet, alors il y a de l'ombre. Ce phénomène est bien illustré par la figure 7. Observez l'ombre et déduisez-en les sources ponctuelles de lumière.

4.5.5 Lumière réfléchie

Comme chacun le sait, les matériaux ont des propriétés de brillance et de réflexivité (comme l'indique l'existence d'une composante spéculaire à la lumière). Dès lors, comment concrétiser le réfléchissement d'une surface dans une autre ? Pour utiliser la réflexivité nous avons décidé de lancer un rayon (depuis l'intersection entre l'objet et le rayon de la caméra) dans la direction du rayon réfléchi. Ainsi, comme l'illustre la figure ci-dessous, on crée l'effet miroir.

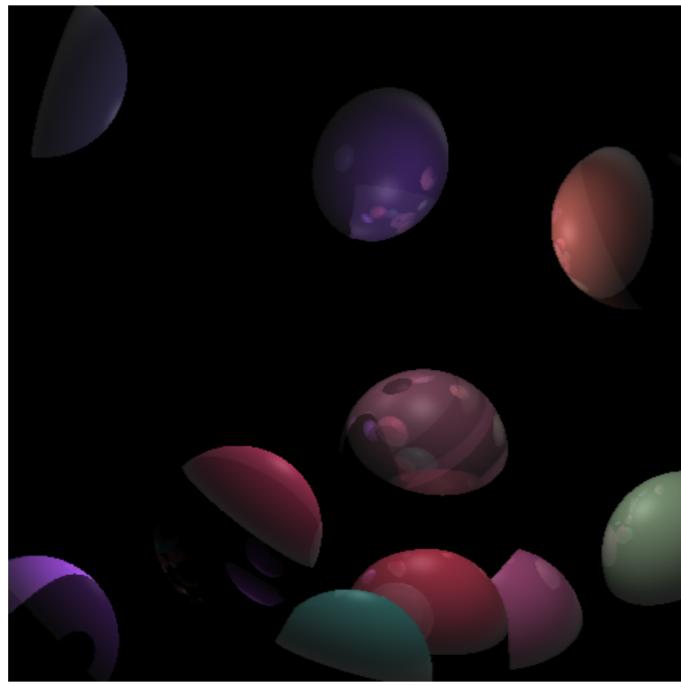


Figure 8 : Illustration de la réflexion d'une onde

Comme l'indique le code suivant, il existe un procédé récursif. En quel sens ? Si C se réfléchit dans B et que B se réfléchit dans A, alors on doit pouvoir voir C dans A. Ce constat est un peu près le suivant. Lorsqu'on dresse deux miroirs l'un face à l'autre, l'image est répétée de façon infinie par réflexion. D'où l'intérêt de pouvoir contrôler cette récursivité. Nous avons choisi d'imposer une limite à celle-ci dans le fichier config.py . Il suffit alors à chaque recherche de lumière de réduire d'1 la limite de récursivité.

```
Rpvec=transperce(self,ray)
if Rpvec<>None:
    I_reflex=cherchlumiere(self,Rpvec,recursivite-1)
```

Le petit schéma ci-dessous illustre bien cette notion.

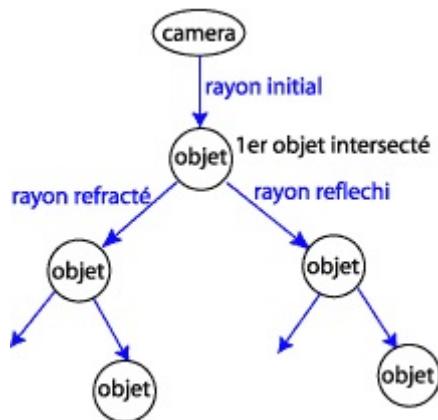


Figure 9 : Illustration de la réflexion d'une onde

5 Problèmes rencontrés

5.1 Le balayage

Comme tout projet qui se respecte, la rencontre de problèmes lors de la confection du programme est inévitable. Nous en avons rencontré quelques uns. Le premier était de savoir quel type de balayage choisir lors du lancement du programme car le type de balayage influence fortement le résultat. Nous avons donc décidé de laisser ce choix de balayage à l'utilisateur. Le choix de balayage a en effet très important car si l'on regarde les différentes sorties le rendu est différent.

5.2 Déformation en balayage cylindrique

Le second problème rencontré concernait la déformation de l'image lorsque l'on tourne la caméra lors d'un balayage cylindrique. L'image ci-dessous nous montre des boules de plusieurs couleurs lancées au hasard dans un sous-espace de l'Espace. Le résultat vu de face (image de droite) est réaliste tandis que celui vu de biais (image de gauche) ne l'est plus. Le problème de manque de réalisme n'est pas présent si la caméra est franchement à gauche ou à droite, ou à l'arrière. Ce problème est propre à la caméra et au tableau XY.

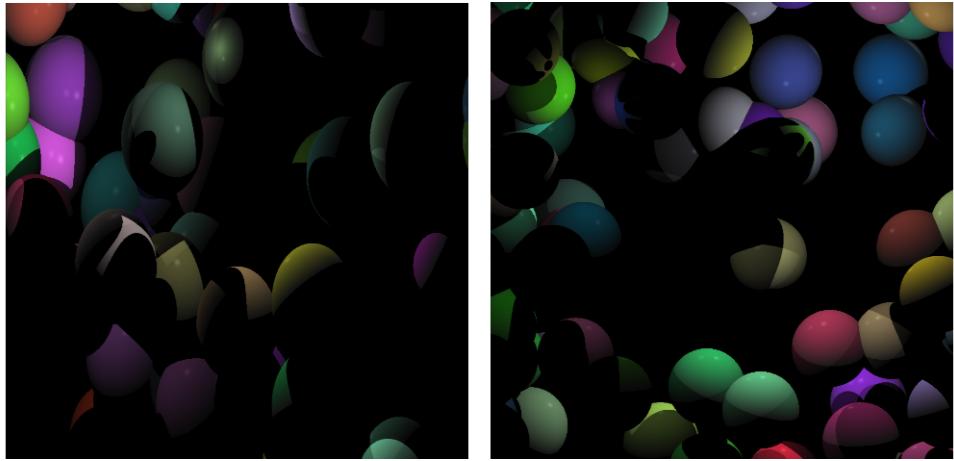


Figure 10 : Résultat vu de face et de biais

5.3 Manque de réalisme pour le balayage conique

Le troisième problème concernait la déformation de l'image vue de face avec une caméra utilisant un balayage cylindrique. L'image ci-dessous représente 9 boules jaunes (voir figure) formant un carré sur un fond en dégradé de violet. L'ouverture focale y est très large et la caméra est orientée selon l'axe x. Le problème apparent ici concerne la déformation des sphères représentant les côtés du cube. En effet, celles-ci sont anormalement étirées.

Un autre effet de ce problème est aussi le mauvais éclairage des plan en conique. Sur l'image ci-dessous, la caméra est orientée selon x. Le plan est orienté selon les axes YZ et XZ. L'objet représenté est une sphère rouge sur un fond en damier noir et blanc. L'image présente des problèmes de réalisme. La sphère rouge est très bien faite mais le damier derrière présente lui un défaut de réalisme et de netteté. Le problème de netteté du damier peut être dû à un mauvais traitement de celui-ci puisque le fond est normal à la caméra.

Le problème étant dû à un mauvais positionnement de la caméra aucune solution n'a été donnée ci ce n'est le changement de position de celle-ci.

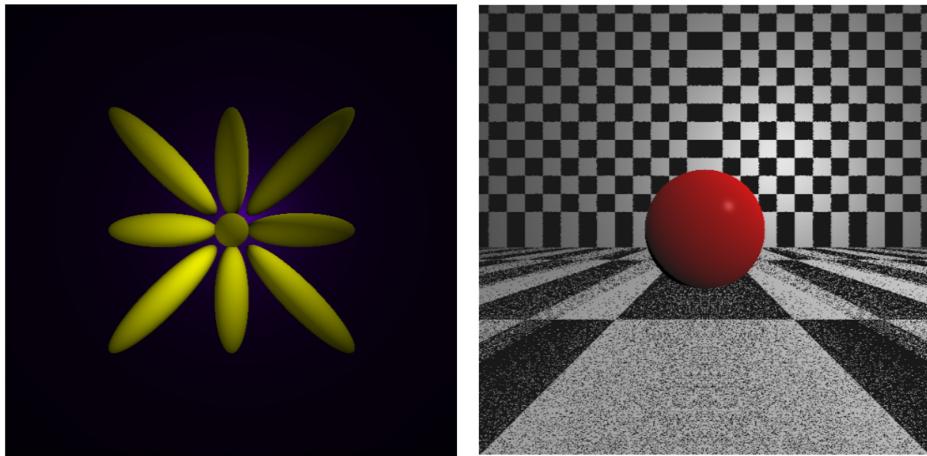


Figure 10 : Illustration du résultat précédemment évoqué vu de face

5.4 Anti-aliasing

Un problème d'anti-aliasing a également été rencontré. Pour rappel, l'anti-aliasing ou antirénelage est une méthode permettant d'éviter le crénelage d'une image, un phénomène qui survient lorsqu'on visualise certaines images dans certaines résolutions. L'image ci-dessous représente 6 sphères lancées au hasard dans un sous-espace de l'Espace. Nous pouvons directement voir que l'image présente des problèmes de netteté dû à l'antirénelage puisque si l'on zoom sur l'image, un effet d'escalier est visible sur les sphères. L'anti-aliasing n'est donc pas présent dans notre projet.

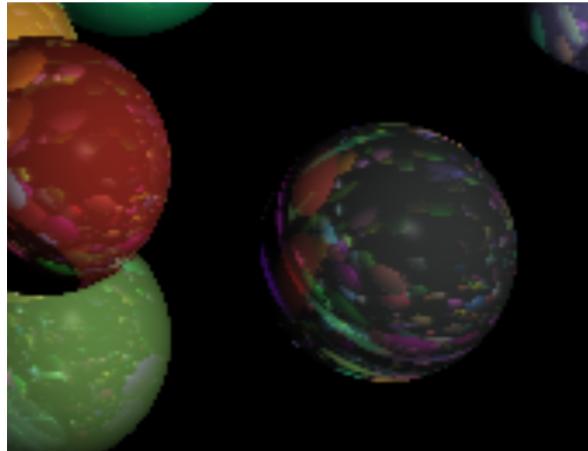


Figure 12 : Illustration représentant l'absence d'anti-aliasing

Une solution proposée serait de lancer plusieurs rayons pour chaque pixel. Dans ce cas, le résultat serait plus net mais le temps de traitement de l'image serait beaucoup plus long.

5.5 Problèmes non résolus

Le problème rencontré le plus important a été la gestion de la transparence qui pour des raisons encore obscure n'est pas utilisable. Bien sûr nous n'espérerions pas obtenir des résultats parfaits mais comme on dit souvent : C'est beau de rêver !

Enfin, un problème d'ombre est toujours présent dans notre programme. Nous avons réussi à matérialiser les ombres des objets les uns sur les autres mais nous n'avons pas réussi à les flouter comme nous pouvons le voir sur les images que l'on retrouve dans l'ensemble du rapport.

6 Réalisations

Lorsque l'on développe un projet nous rencontrons toujours des problèmes mais nous pouvons et nous devons rencontrer des succès ! C'est de nos succès que traitera cette partie du rapport. Nous vous y montrerons les meilleurs résultats obtenus et les commenterons.

Notre plus longue réalisation a mis plus de 6h à être générée. Pour cause, nous lui avons demandé de générer une centaine de sphères de couleurs différentes au hasard dans le sous-espace. Nous avons réalisé cette image afin de montrer que notre programme prend bien en compte la réflexion. En effet, sur une sphère nous pouvons voir les reflets d'autres sphères. Ici la caméra est placée de façon à nous procurer une vue de face. Comme nous l'avons dit précédemment, les ombres ne sont pas floues et provoque la disparition totale de certaines sphères qui sont derrière d'autres. Voici le résultat :

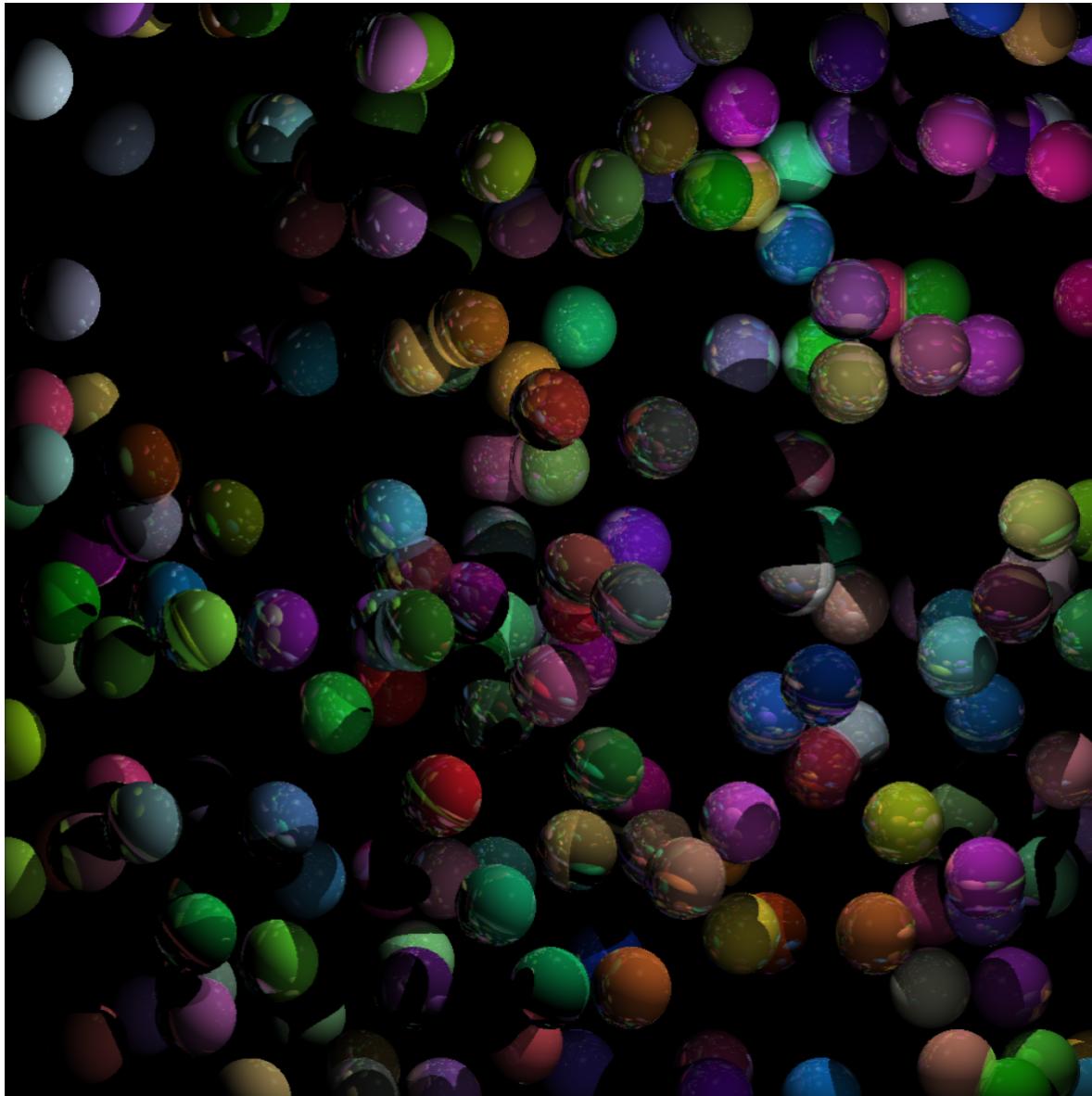


Figure 13 : Illustration du résultat obtenu après 6h de traitement

La seconde image que nous allons vous présenter représente une sphère bleue sur un fond mauve. Deux sources de lumières ont été paramétrées comme nous pouvons le constater sur l'image ci-dessous. La caméra a également été placée de face. Ce choix de placement de la caméra nous confère un très bon rendu aussi bien en ce qui concerne la netteté mais aussi le réalisme de la superposition de la sphère et du fond. La sphère étant unique dans ce placé, les seuls reflets que nous pouvons voir sont ceux de la lumière sur la sphère. Voici l'image en question :

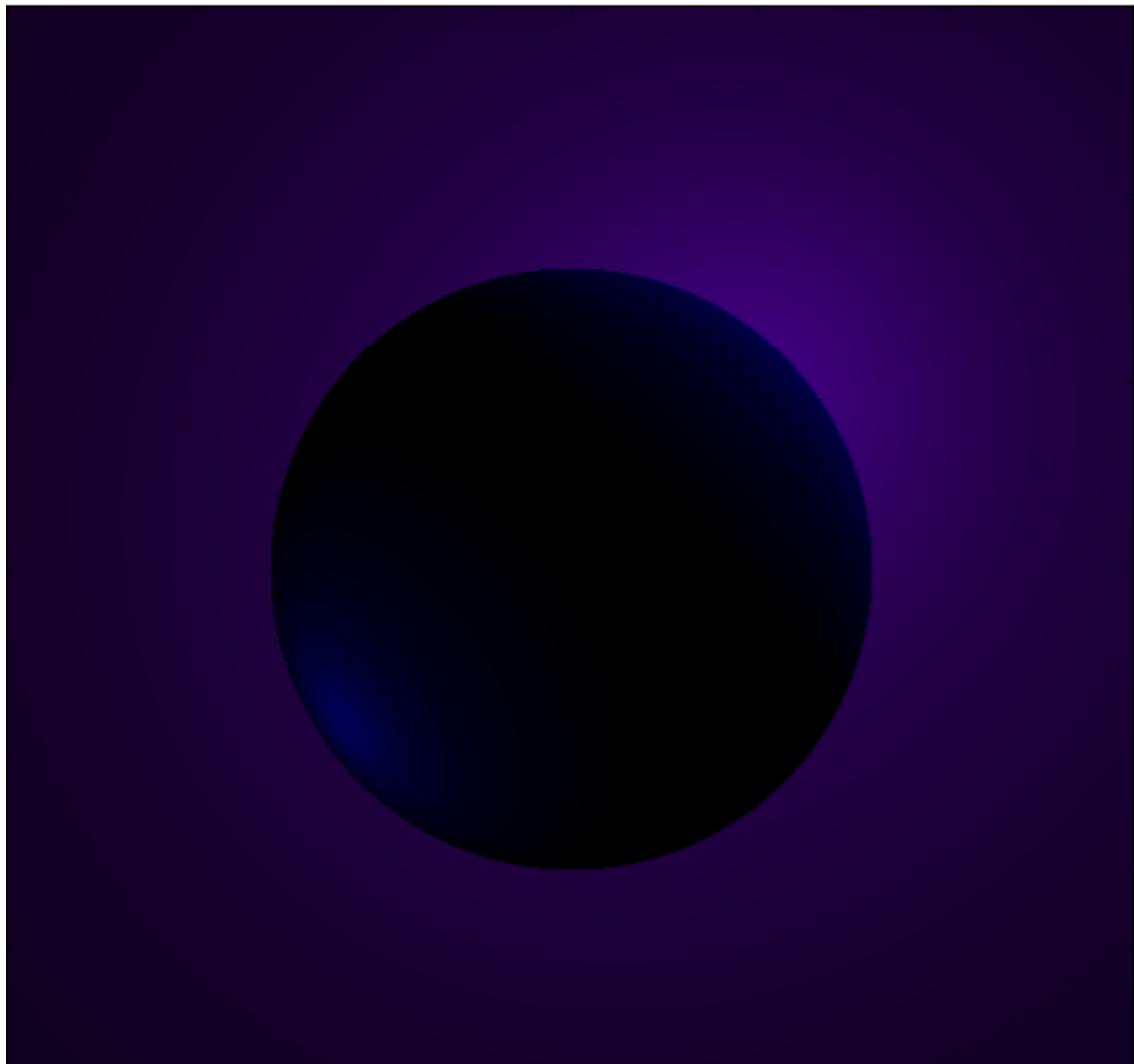


Figure 14 : Illustration d'une sphère bleue sur fond mauve

Un autre réalisation que nous considérons comme réussie représente trois sphères qui se croisent. Les sphères sont bleue, rose et jaune. Elles se trouvent toutes les trois sur un fond noir. Ici nous pouvons aussi voir que les ombres ne sont pas flous si il s'agit de l'ombre d'un objet sur l'autre. La caméra a ici été placée de face. Cette image a été générée afin de montrer la qualité de la réflexion des objets les uns sur les autres. En effet, nous pouvons apercevoir les reflets des sphères jaune et rose sur la sphère bleue. Il en est de même pour les autres sphères. Voici l'image dont il est question :

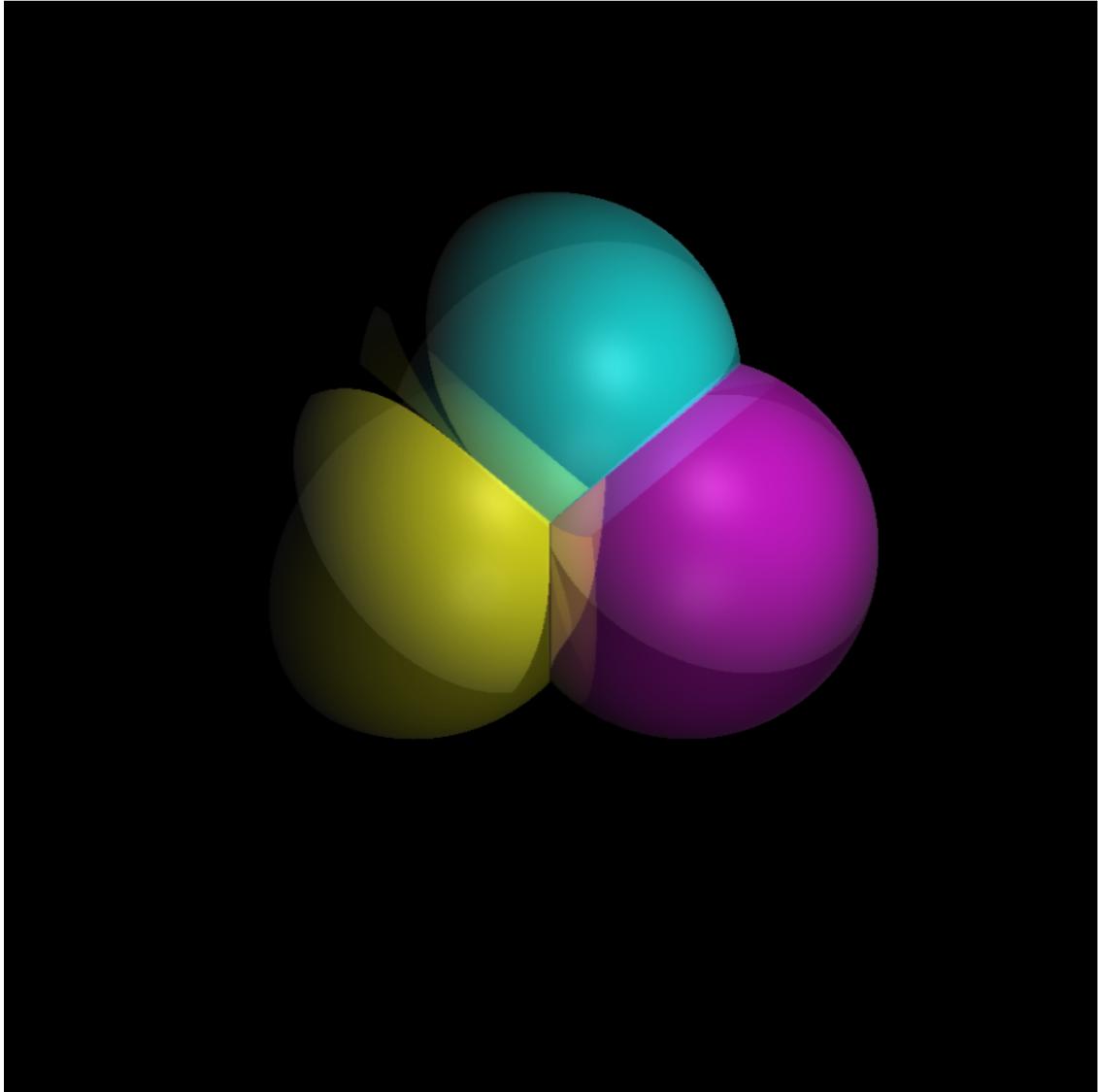


Figure 15 : Illustration de trois sphères colorées sur fond noir

La dernière réalisation que nous voulions aborder représente 9 sphères rouges sur un fond en damier blanc et cyan. La caméra a été placée de façon à voir les sphères avec une vue en profondeur. Une unique source lumineuse a été placée en haut de l'image. Les sphères sont alignées. La lumière n'étant pas au milieux de l'image, les sphères se trouvent plus ou moins éclairées en fonction de leur distance avec la lumière. Une fois de plus, nous pouvons constater la qualité des reflets puisqu'en regardant bien nous pouvons voir le reflet de la lumière sur les sphères. Nous pouvons également constater que malheureusement les ombres sont opaques ce qui demeure toujours un problème à l'heure actuelle. Cependant celle-ci sont très bien matérialisée par rapport à la source de lumière.

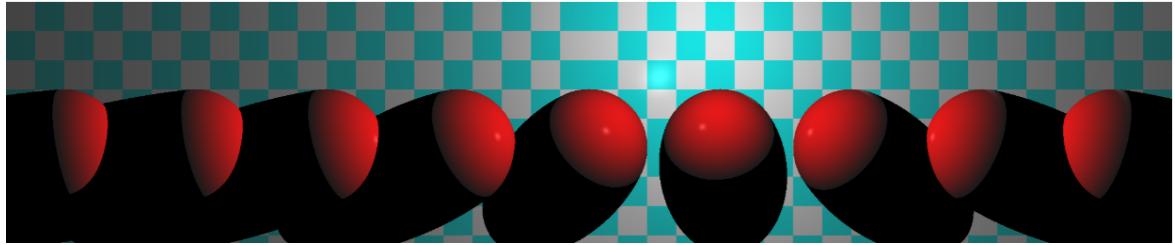


Figure 16 : Illustration de neuf sphères rouges sur fond en damier blanc et cyan

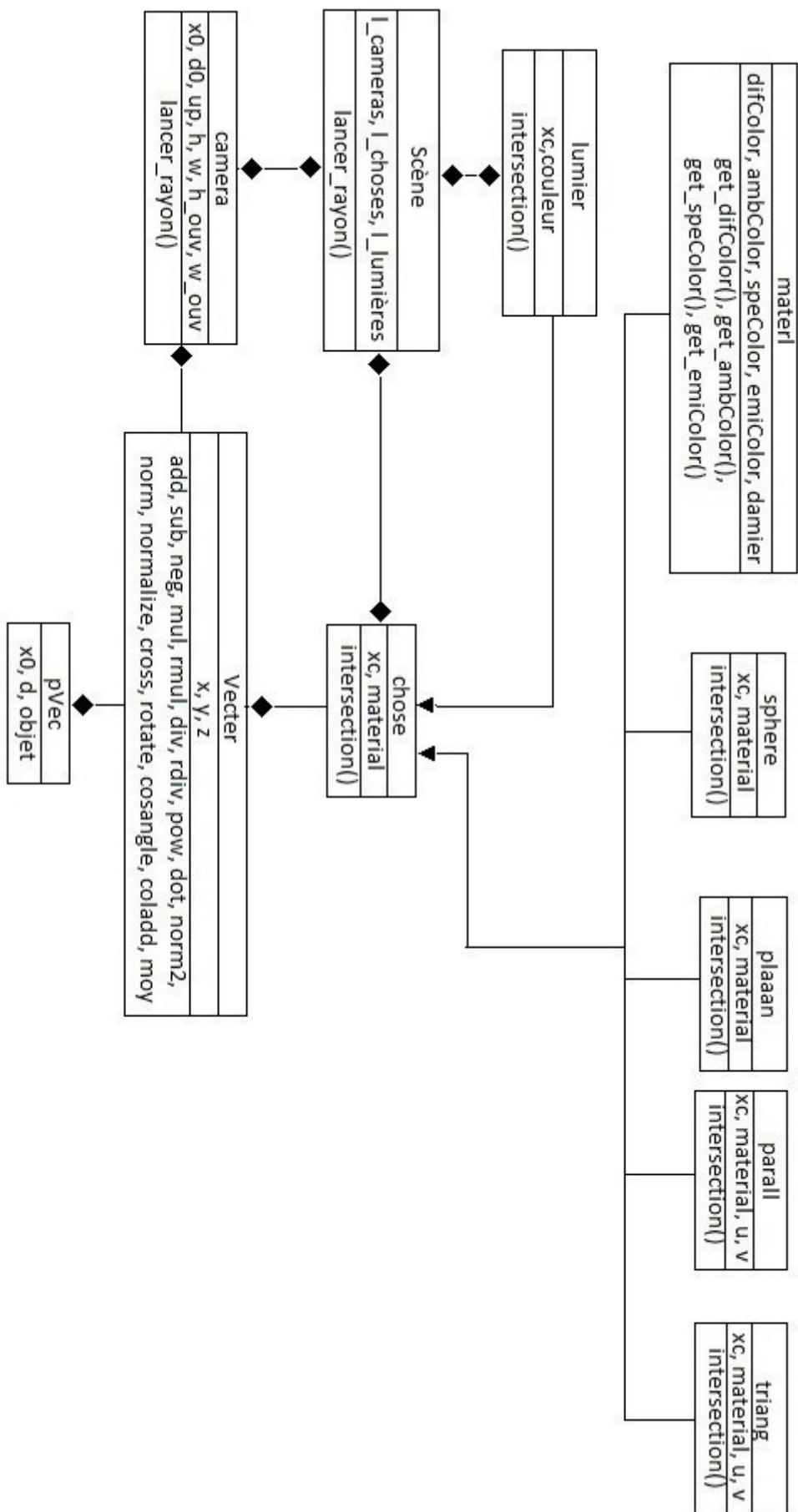
7 Sources

```

http://fr.wikipedia.org/wiki/Lancer\_de\_rayon
http://fr.wikipedia.org/wiki/Réfraction
http://en.wikipedia.org/wiki/Ray\_tracing\_\(graphics\)
http://heigeas.free.fr/laure/ray\_tracing/index.html
http://heigeas.free.fr/laure/ray\_tracing/realisation.htm
http://www.codermind.com/articles/Raytracer-in-C++-Part-I-First-rays.html
http://www.codermind.com/articles/Raytracer-in-C++-Part-II-Specularity-post-processing.html
http://www.alteryxa.net/article-raytracing-by-roxtarmy-106494723.html
http://www.alrj.org/docs/3D/raytracer/raytracertutchap1.htm
http://www.alrj.org/docs/3D/raytracer/raytracertutchap2.htm
http://blog.lexique-du-net.com/index.php?post/2009/07/24/AmbientDiffuseEmissive-and-specular-colors
http://www.massal.net/article/raytrace/page1.html
https://karczmarczuk.users.greyc.fr/TEACH/OGL/tp2210.html
https://karczmarczuk.users.greyc.fr/TEACH/OGL/tp1510.html
http://physique.paris.iufm.fr/lumiere/cielbleu.html
http://wiki.blender.org/index.php/Doc:FR/2.4/Manual/Lighting/Ambient\_Light

```

ANNEXE : Schema « UML » du projet en python



ANNEXE : Un exemple de fichier de configuration

```
#Imports
from lumier import *
from colors import *
from materl import *
from camera import *
from sceene import *
from sphere import *

#MATERIAUX
matb=Material(difColor=cya,speColor=whi,traRatio=0.8,ambColor=cya,briPower=10)
matr=Material(difColor=ros,speColor=whi,traRatio=0.8,ambColor=ros,briPower=10)
maty=Material(difColor=yel,speColor=whi,traRatio=0.8,ambColor=yel,briPower=10)

#LUMIERES
lum=Lumiere(Vec(30,15,-15),whi)
lum2=Lumiere(Vec(20,-15,-15),whi)

#SPHERES
sp1=Sphere(Vec(0,-6,0),matb,7)
sp2=Sphere(Vec(0,0,-5),matr,7)
sp3=Sphere(Vec(0,0,5),maty,7)

#CAMERA
x0=30*X3
d0=-X3
up=Y3
w_ouv=40
h_ouv=40
w_size=500
h_size=500
cam=Camera(x0,d0,up,w_size,h_size,w_ouv,h_ouv)

#SCENE
balayage=False #Conique:True - Cylindrique:False
scan=0 #indice de la camera a scanner
recursivite=2

#--Lumieres
l_lum=[lum,lum2]

#--Objets
l_obj=[sp1,sp2,sp3]

#--Cameras
l_cam=[cam]

sce=Scene(l_cam,l_obj,l_lum)
save_fi="test.png"
```