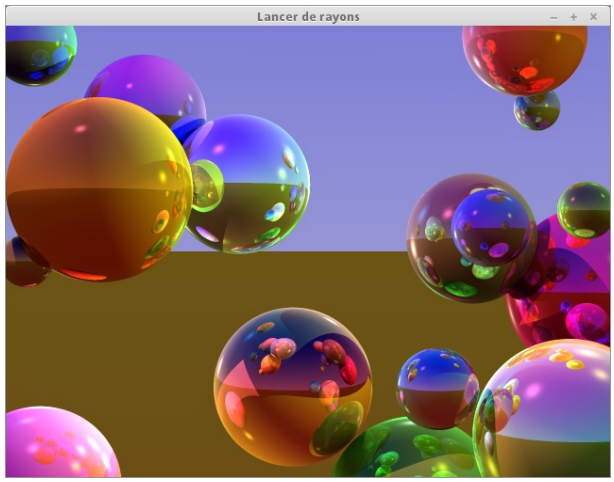
TP n°1 : lancer de rayons

Le but du TP est de concevoir complètement un programme de synthèse d'images par lancer de rayons. Ce programme ne sera pas optimisé, car le but est de comprendre certains mécanismes qu'on retrouve dans les programmes OpenGL. Plusieurs concepts sont employés sans être définis, comme les éclairements de Lambert et de Phong. Ils seront pleinement justifiés dans la suite du cours. Ici, on va seulement constater leur justesse expérimentalement. Vous obtiendrez ceci à la fin :



Le point de départ est un logiciel inachevé écrit en Java. Il est à télécharger sur la page Moodle du cours.

Note : le projet Java est destiné à Java8 et l'environnement JavaSE1.7, au minimum et AWT et Swing pour l'interface. Il est préférable de travailler avec Eclipse ou IntelliJ, dans une version récente. Mais attention aux fichiers supplémentaires que ces outils créent et qu'il faudra enlever de votre rendu de TP.

Dans l'IDE, il faut importer le projet `lancer-base`. C'est un projet maven, avec un pom.xml qui précise les dépendances (JUnit5 et ini4j).

Important : choisir `UTF-8` comme encodage par défaut, y compris sur Mac. Sur Eclipse, c'est dans les préférences : menu `Window`, item `General`, sous-item `Workspace`, `Text File encoding`. Et de préférence, utilisez des 4 espaces pour indenter, pas des TABS.

Important : ne changez pas le package, ne changez pas les noms des classes et autres. Ça complique grandement la correction et on vous enlèvera des points.

Pour l'exécution, il faut créer une *configuration de lancement* pour démarrer sur la classe `Main` et avec un paramètre qui doit être « `scenes/scene1.ini` ».

La première version de ce logiciel affiche une fenêtre entièrement verte. C'est signe que tout s'est bien passé avec le code qui vous est fourni. Ça ne veut pas dire que la synthèse d'image s'est bien passée. Ce logiciel est à compléter tout au long de ce TP, avec les indications de cet énoncé.

Structure du logiciel

Prenez d'abord le temps de regarder les classes de ce logiciel :

- `Tuple`, `Vecteur`, `Point`, `Couleur` : pour les calculs mathématiques,

- `Objet3D`, `Sphere`, `Lampe`, `Camera`, `Scene` : pour les éléments à dessiner,
- `Constantes`, `Rayon`, `PointContact`, `Lancer` : pour l'algorithme de dessin,
- `Main, Tests` : permettant de lancer l'exécution.

Le logiciel fourni contient des tests unitaires dans la classe `Tests` (à lancer avec le *runner* de Junit5). Ce n'est pas parfait, cela signale seulement des erreurs très grossières et évite de galérer ultérieurement.

On va compléter chaque classe dans un ordre permettant de comprendre le tout.

Points et vecteurs

La classe `Tuple` représente un triplet de coordonnées x,y,z. Cette classe est dérivée pour représenter des `Point` et des `Vecteur` ayant des méthodes spécifiques. Ce n'est pas le cas, mais on pourrait en faire une variante pour représenter les `Couleur` — de fait, les couleurs sont une classe spécifique car leurs composantes s'appellent r,v,b.

Dans ces classes, les méthodes créent toutes de nouvelles instances, ex : `Vecteur.add(u,v)` et `u.add(v)` retournent un nouveau vecteur somme de u et de v. C'est un choix. On aurait pu faire en sorte que ces méthodes modifient le vecteur u ; ça aurait réduit le nombre d'allocations mémoire, mais obligé à créer des instances pour recevoir le résultat, ce qui alourdit l'écriture.

✍ Compléter le constructeur de la classe `Vecteur` qui reçoit deux `Point` en paramètres. Enlevez le commentaire FIXME quand c'est fait, ceci à chaque fois.

✍ Compléter les méthodes `sub` présentes dans les classes `Point` et `Vecteur`.

✍ Compléter la méthode qui calcule le produit scalaire entre deux Vecteurs. Elle s'appelle `dot`. Réveillez vos souvenirs du lycée : quel lien y-a-t-il entre le produit scalaire et la norme d'un vecteur ? Quel est le rapport entre le produit scalaire de deux vecteurs et l'angle qui existe entre eux ?

✍ Compléter la méthode `norme2` de la classe `Vecteur` qui retourne le carré de la norme d'un vecteur. Cette méthode est utilisée pour normaliser un vecteur, c'est à dire mettre sa « longueur » à 1 sans changer sa « direction ». Il y a deux méthodes pour normaliser.

Représentation des rayons

Soit une classe `Rayon` représentant une demi-droite 3D, elle contient les variables d'instance `P` (point de départ de la demi-droite) et `V` (vecteur directeur, normé).

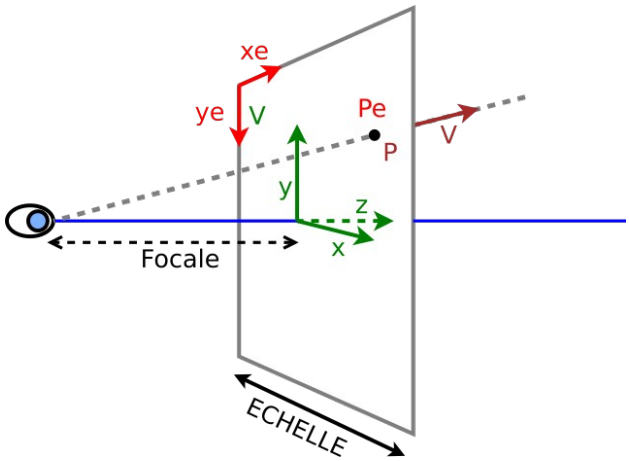
✍ Écrire le constructeur `Rayon(Point p1, Point p2)` auquel on fournit deux points : `p1` est le point de départ du rayon et son vecteur directeur va de `p1` à `p2`. Il faut normaliser ce vecteur.

✍ Il y a un autre constructeur qui prend directement un point et un vecteur. Le vecteur est à normaliser (sauf si on considère qu'il l'est toujours, mais il faudrait voir ce qu'il en est à chacun des appels).

Écran, œil et projection : classes Rayon et Camera

On s'intéresse aux calculs géométriques pour construire un Rayon partant de l'œil et traversant un pixel. Il y a deux repères en compétition : le repère 3D correspondant aux objets et lampes à dessiner et le repère de l'écran. On veut pouvoir positionner la scène comme on veut, mais cependant pas faire trop compliqué, donc on choisit de n'avoir qu'un seul repère, celui du plan de l'écran, l'œil étant derrière et la scène devant.

Pour commencer, on doit écrire les équations qui permettent de convertir des coordonnées écran en coordonnées du monde. Soit un point $P_e(x_e,y_e)$ sur le schéma ci-dessous. On veut écrire ses coordonnées P(x,y,z) dans le monde. L'écran a une largeur de L et une hauteur de H (x_e va de 0 à L–1 vers la droite, y_e de 0 à H–1 vers le bas).



On voit déjà que z=0. On voit aussi qu'il y a une relation directe entre x_e et x d'une part et entre y_e et y d'autre part. Ce sont deux relations affines, c'est à dire qu'elles sont linéaires avec un décalage.

Il ne serait pas approprié d'écrire seulement $x = x_e - L/2$, et $y = H/2 - y_e$, parce que ça ferait des coordonnées x et y qui varieraient selon la taille de l'écran.

✍ Faites le calcul : (x_e,y_e) valant (0,0) pour un écran (L=800, H=600) et pour un écran 1280x1024.

Ça aurait un effet sur l'image produite, son cadrage varierait beaucoup, parce que les objets ont une taille fixe, de quelques unités. Plus l'écran serait grand, plus la scène apparaîtrait minuscule.

On va donc appliquer une homothétie pour que les coordonnées (x,y) soient indépendantes de la taille (L,H) de l'écran. On va poser que la largeur L de l'écran est ramenée à ECHELLE unités du monde. Par exemple, dans le TP, cette constante vaut 2. Si l'écran est en 800x600, alors 400 pixels correspondent à 1 unité du monde.

✍ Quelles sont les coordonnées (x,y,0) des pixels (0,0) et (L,H) avec ECHELLE valant 2 ?

✍ Exprimez les calculs pour obtenir (x,y,0) en fonction de (x_e,y_e) en tenant compte de la constante ECHELLE. Vérifiez que le pixel (0,0) correspond toujours au même point (x,y,0) pour deux tailles d'écran différentes, ou prouvez-le dans le cas général.

On s'intéresse maintenant au champ de vision de la caméra.

Soit un `Point` appelé `Œil`, situé à une distance appelée *Focale* de l'écran (distance comptée en unités du monde) et centré au milieu de l'écran. Au contraire des appareils photo, cette distance n'est pas connue (en photo argentique, avec des images 36x24mm, une focale classique est le 50mm). À la place, dans ce TP, on va travailler avec un angle de champ. Cet angle est défini en considérant la largeur entière de l'écran vu de l'œil. Dans le TP, cet angle est défini à 30°, voir Constantes.CHAMP.

Il y a une relation entre cet angle, la constante ECHELLE et la distance focale.

✍ Exprimez le calcul de la distance focale.

✍ Programmez tous ces calculs dans la classe `Camera`.

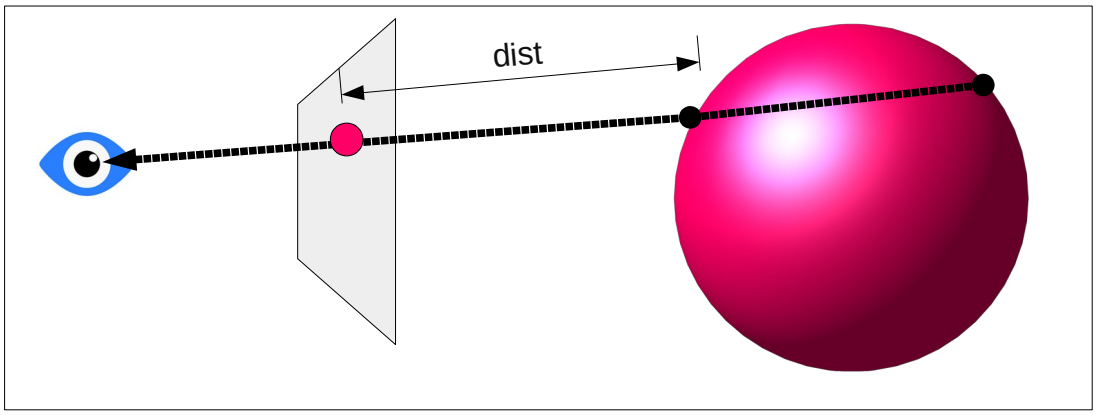
NB : dans un logiciel plus complet, la caméra serait mobile, orientable dans n'importe quelle direction.

Intersection de rayons et de sphères : classe Sphere

La classe `Sphere` représente une sphère, ses membres sont un `Point` appelé `centre` et un réel `rayon`[1]. C'est une sous-classe de `Objet3D`.

✍ Écrire la méthode `getIntersectionDistance` qui renvoie la distance de la plus proche intersection entre cette sphère et le point de départ du rayon. NB : on ne considère qu'une seule sphère pour l'instant. La méthode ne doit pas retourner de valeurs négative qui signifieraient un contact dans le dos de la demi-droite. La méthode renvoie la distance $+\infty$ s'il n'y a pas d'intersection. Voir le schéma suivant, le rayon touche la sphère rouge en deux points, et on renvoie la distance du plus proche des deux.

NB : le rayon est dessiné partant de l'œil, mais en fait, il part du pixel.



Normalement à ce stade, tous les tests unitaires doivent réussir.

Scène de sphères : classes Scene et Rayon

La scène est une liste d' `Objet3D` qui sont tous des sphères. Voir la classe `Scene`.

Pour dessiner la scène, il va falloir chercher les intersections d'un `Rayon` avec tous les objets de cette scène. Plusieurs objets peuvent être coupés par le rayon, mais seul le plus proche nous intéresse car c'est lui seul qui est visible et qui donne la couleur au pixel.

Pour représenter le point le plus proche de l'écran et l'exploiter utilement ensuite, on emploie la classe `PointContact` qui décrit le point de contact : sa variable `objet` contient le plus proche objet rencontré par le rayon `incident`, à la distance `distance`, enfin, les coordonnées du contact sont dans la super classe.

✍ Lorsqu'une intersection est trouvée, on calcule le point de contact. C'est le rôle de la méthode `calcContact()` de la classe `Rayon`. Programmer cette méthode, sachant qu'elle se base sur `distance`, ainsi que `P` et `V` pour calculer `contact`. Elle retourne null si la distance est infinie.

✍ Écrire la méthode de la classe `Scene` : `PointContact getClosestIntersection(Rayon incident, Sphere sauflui)` qui parcourt la scène à la recherche de l'objet le plus proche traversé par le rayon, sauf l'objet indiqué dans le paramètre, ceci pour ne pas trouver des intersections des rayons émis par les

1 attention, le mot *rayon* a deux sens dans cet énoncé : rayon de la sphère (demi-diamètre, *radius* en anglais) et rayon lumineux (demi-droite, *ray* en anglais).

objets avec eux-mêmes à cause des imprécisions de calcul. Cette méthode renvoie `null` ou le plus proche `PointContact` trouvé.

Algorithme général : classe Main

Soit une image de taille (Largeur, Hauteur) à dessiner en lancer de rayons. Admettons que, dans la classe `Objet3D` (la superclasse de `Sphere`), on dispose déjà de la méthode `Couleur getColor(…, PointContact contact, …);` qui calcule la couleur d'un rayon percutant une sphère. Elle sera à programmer ultérieurement, car elle est assez complexe. Actuellement, elle retourne du jaune vif.

La classe `Main` fournit la méthode `drawPixel(Couleur coul, int xe, int ye);` qui dessine un pixel de cette couleur. Cette méthode est appelée par la méthode `TracerImage` qui est déjà programmée car elle est emmêlée avec l'affichage d'une jauge d'avancement sur l'interface graphique, et également l'affichage rapide d'un brouillon d'image.

✍ Étudier l'algorithme de la méthode `TracerImage`. Son cœur est une double boucle sur tous les pixels. Sur chacun, elle appelle la méthode `getCouleurPixel(xe, ye, …)`. Celle-ci est à programmer. Enfin, la méthode dessine le pixel de cette couleur.

✍ Programmer la méthode `getCouleurPixel`. Elle crée un rayon partant de l'œil, passant par le pixel à dessiner, calcule et retourne la couleur de ce rayon. La couleur est calculée avec la méthode `getColor` de la classe `Objet3D`. À noter que celle qui vous est fournie ne fait que retourner du jaune vif. S'il n'y a pas d'intersection avec un objet, alors il faut retourner la couleur du ciel, calculée avec la méthode `getSkyColor` de la classe `Rayon`.

Bilan intermédiaire

À ce stade, voici la succession simplifiée des appels des méthodes :

Dans `Main.java` :

1) entrée dans la méthode `main`()
2) chargement de la scène : scene = new Scene(nom du fichier à lire, c'est le premier paramètre fourni au lancement);
3) affichage d'une fenêtre à l'écran associée à un `SwingWorker` (thread de dessin)
4) ce thread appelle la méthode `doTracerImage`(largeur, hauteur) : double boucle en xe, ye pour colorer chaque pixel à l'aide de `getCouleurPixel`(xe, ye) :
    a) construction d'un rayon œil → pixel(xe,ye transformés en x,y,z) appelé `initial`
    b) appel à la méthode `Scene.getClosestIntersection`(initial)
        i. appels à la méthode `Sphere.getIntersectionDistance`(initial)
        ii. appel à la méthode `Rayon.calcContact`() sur la sphère la plus proche pour calculer les coordonnées du point de contact
    c) à ce stade, si le rayon initial frappe une sphère, alors cette sphère avec la distance et les coordonnées du point de contact sont présents dans le rayon
    d) appel à la méthode `Objet3D.getColor`(scene, `initial`) ou sinon `initial.getSkyColor`()

Deux méthodes importantes dans la classe `Objet3D` et sa sous-classe `Sphere` :

1) La méthode `getIntersectionDistance`(Rayon incident) calcule la distance d'intersection entre l'objet et le rayon incident, retourne $+\infty$ si aucun. Elle stocke aussi cette distance dans le rayon.
2) La méthode `getColor`(scene, incident) retourne la couleur de l'objet au point d'intersection trouvé. À ce stade, la couleur est constante, jaune vif, mais devra appliquer un modèle d'éclairement réaliste au cours de ce TP. Il faudra utiliser toutes les informations : point de contact, géométrie de la sphère, lampes, etc.

Éclairement diffus : classes Objet3D et Lampe

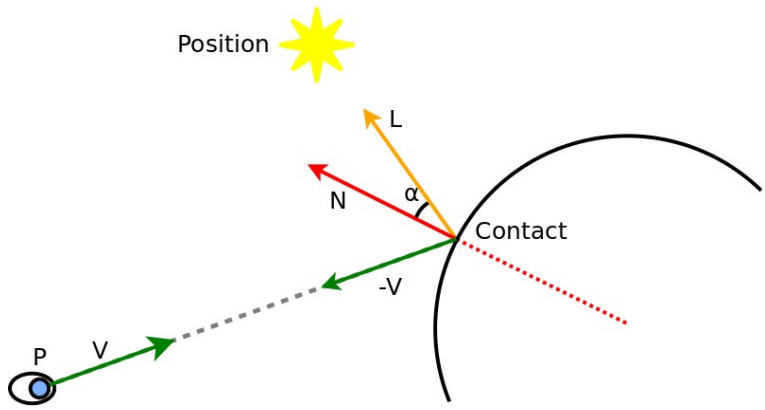
On va maintenant écrire la méthode `getColor` de la classe `Objet3D`. Elle reçoit un `Rayon` en paramètre et calcule la couleur qu'il voit.

Soit une lumière positionnelle ponctuelle représentée par un `Point position` et une `Couleur`. Soit un rayon (P,V) dessiné en pointillés qui arrive sur la sphère au point « *Contact* ». On veut la teinte de ce point pour pouvoir le dessiner à l'écran. L'équation de Lambert donne la teinte (*shade* en anglais) en fonction du vecteur normal N, de la direction de la lumière L, de sa couleur et intensité K_L et des caractéristiques du matériau Kd :

$$\text{Couleur}_{\text{diffuse}} = \cos(\alpha) * Kd * K_L = (N.L) * Kd * K_L$$

Note 1 : Tous les vecteurs concernés doivent être normalisés.

Note 2 : Ce calcul est à faire seulement si le produit scalaire $N.L$ est positif.



Cette équation modélise l'éclairement d'une surface parfaitement mate. Nous verrons dans la partie 4 du cours comment d'autres matériaux sont modélisés.

Questions :

- Comment calculer les coordonnées du vecteur normal N au point de contact ?
- Comment calculer les coordonnées du vecteur L ?
- Comment les ombres apparaissent-elles dans le dos des objets éclairés ?
- Comment fait-on s'il y a plusieurs sources de lumière ?

✍ Programmer le calcul de l'éclairement diffus dans la méthode `Objet3D.getColor`. La liste des lampes est à demander à l'instance de `scene` passée en paramètre.

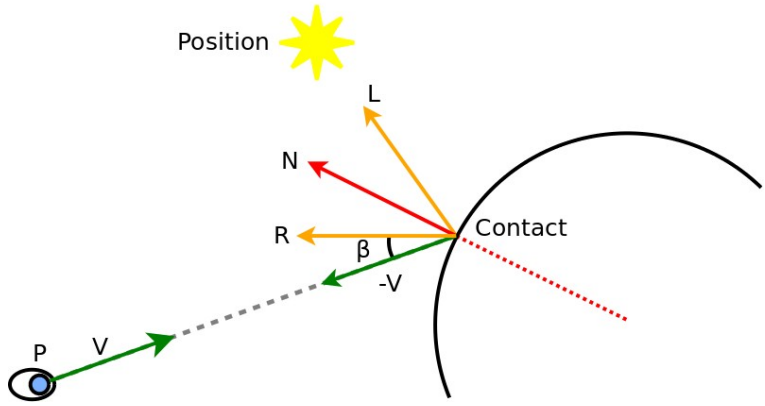
À ce stade, chaque sphère doit avoir une couleur spécifique et plus ou moins claire en fonction de la lumière. Il n'y a pas d'ombres portées, seulement une ombre dans le dos des objets par rapport aux lampes.

Éclairement spéculaire

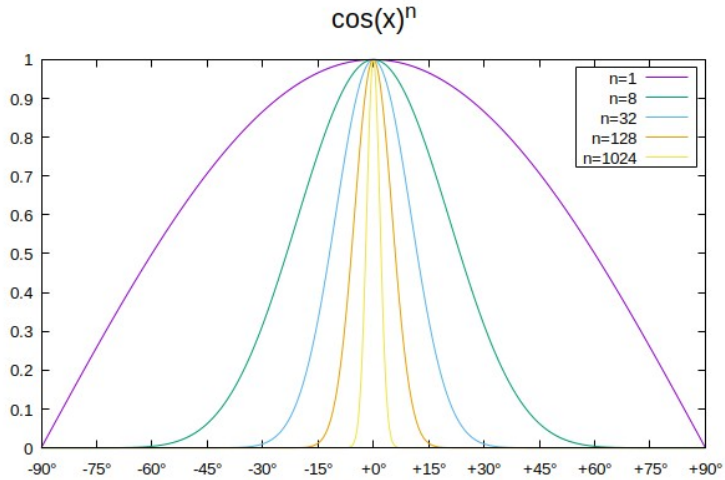
L'éclairement spéculaire provient d'un reflet de la lumière sur la surface quand celle-ci est suffisamment lisse. La formule de Phong donne la teinte en fonction de la couleur de réflexion du matériau *Ks* et de la couleur de la lampe K_L :

$$\text{Couleur}_{\text{spéculaire}} = (R.-V)^{ns} * Ks * K_L$$

Le vecteur *-V* est l'opposé du vecteur directeur du rayon, c'est un vecteur qui désigne l'œil. Le vecteur *R* est le miroir du vecteur *L* par rapport à *N*, c'est le vecteur qui indique là où partent les reflets, et *ns* est un coefficient qui caractérise le poli de la surface. L'intensité du reflet est donc proportionnelle au cosinus de l'angle *β* élevé à la puissance *ns*. L'élévation à la puissance permet de rendre la fonction cosinus « pointue » : plus *ns* est grand, plus la région où la fonction vaut 1 se réduit. Donc le reflet n'est visible que dans une toute petite zone quand *ns* est grand.



Voici le graphe de $cos(x)^n$ avec différents exposants *n* : 1, 8, 32, 128 et 1024 ; x représente l'angle en radians entre *-V* et *R*.



Cette équation modélise le reflet sur une surface parfaitement lisse. Nous verrons dans la suite du cours comment d'autres surfaces sont modélisées. En général, les surfaces combinent les deux types d'éclairements : diffus et spéculaire.

Questions :

- Comment calcule-t-on le vecteur miroir de *L* par rapport à N ?
- Remarquer que l'angle β est entre R et -V ; R étant le miroir de *L* par rapport à *N*. On retrouve le même angle entre *L* et le miroir de *-V* par rapport à *N*. Lequel des deux permet de limiter le nombre de calculs ?

✍ Programmer l'algorithme de calcul de la contribution RVB spéculaire qui s'ajoute à l'éclairement diffus.

À ce stade, les objets montrent une tache brillante indiquant visuellement la position des lampes.

Ombres portées

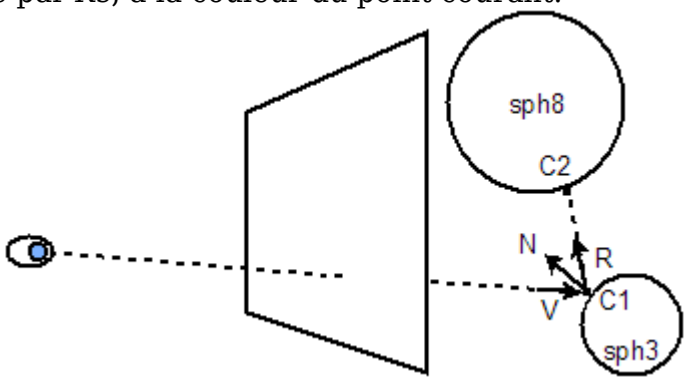
Les deux calculs précédents ne doivent pas être faits s'il y a un objet entre le point de contact et la lumière. Comment fait-on pour relancer un rayon à partir du point de contact pour tester cela ?

En fait, ce n'est pas du point de contact vers la lampe qu'il faut lancer le rayon, mais à l'inverse, sans quoi, on détecterait à tort des objets situés derrière la lampe. Il vaut mieux lancer un rayon partant de la lampe et allant vers le point de contact du rayon initial et regarder si le premier contact de ce rayon désigne bien l'objet considéré : si c'est un autre objet qui est plus près de la lampe, alors le nôtre est dans son ombre.

✍ Programmer le test qui élimine certains calculs d'éclairement là où il y a une ombre portée.

Reflets

On peut ajouter une autre contribution à la couleur du point de contact : les couleurs des objets qui se reflètent à cet endroit. Il « suffit » de relancer un rayon dans la direction miroir du rayon actuel, de demander la couleur qu'on y voit et d'ajouter cette contribution, modulée par *Ks*, à la couleur du point courant.



Explication : la surface de sph3 est très lisse, la lumière qu'on voit venir du point *C1* provient en fait de la sphère sph8 au point *C2*. Le point *C2* est situé sur le rayon partant de *C1* dans la direction *R* miroir du rayon initial. Les calculs se font donc dans le sens inverse du parcours de la lumière.

Questions :

- Le calcul de la direction du vecteur *R* miroir de *-V* par rapport à *N* a déjà été programmé, non ? Finalement, ça tombe assez bien.
- Comment relance-t-on un rayon dans cette direction ?
- Comment mélange-t-on la couleur du reflet avec la couleur du matériau ?

✍ Programmer l'ajout des reflets dans la fonction `getColor`.

À ce stade, les images deviennent très spectaculaires en montrant de très nombreux reflets réalistes.

Que se passe-t-il si on doit dessiner deux surfaces localement planes face à face, quel risque pour le programme ? Implémenter une solution : le paramètre supplémentaire

passé à la fonction `getColor` limite la profondeur d'appel. On pourrait aussi accumuler l'opacité des surfaces et cesser les appels récursifs quand elle dépasse un seuil au-delà duquel le reflet n'est quasiment plus visible.

Extensions

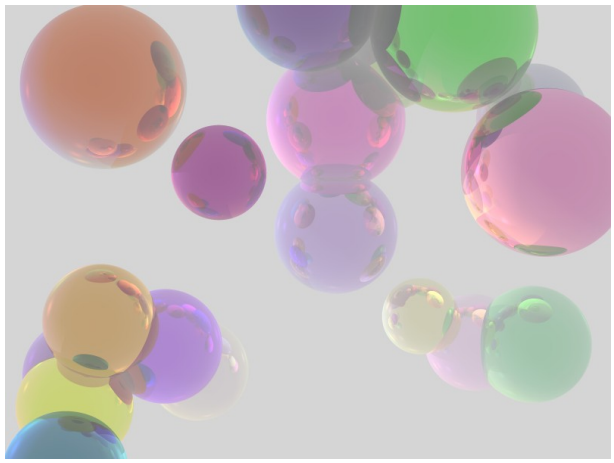
Ce qui suit sont des ajouts au projet initial. On retrouvera ces concepts avec OpenGL. Essayez d'en faire le plus possible. Tous vont être présentés interactivement et expliqués en séance autant que vous voudrez. La notation portera sur la qualité de réalisation.

NB : le corrigé de tout ce qui précède vous sera fourni par mail (base-java.zip).

Important : pour la mise en œuvre, il faudra recopier à chaque fois le dossier lancer-Java que vous avez obtenu (reflets, ombres portées…) et le renommer selon le projet voulu. Vous aurez ainsi : brume, antialias, blinn, etc. Dans chacun de ces dossiers, on trouvera src, scenes ainsi que le Makefile. Le dossier scenes peut être placé au dessus, afin de le mettre en commun.

Brume de distance

Comment peut-on ajouter un effet de brume de distance ? Il s'agit d'une couleur fixe qui remplace peu à peu la couleur des objets en fonction de la distance traversée par le rayon lumineux.



Elle intervient à différents niveaux : sur le rayon initial, et dans chaque rayon indirect lancé pour calculer les réflexions. À chaque fois, on utilise la distance du contact pour calculer la densité de la brume. On calcule un paramètre k variant entre 0 et 1 (empêcher qu'il sorte de cette plage) qui est une fonction de la distance parcourue par le rayon, et on combine la couleur donnée par `getColor` avec la couleur de la brume par cette formule :

$$\text{Couleur}_{\text{finale}} = k*\text{Couleur}_{\text{brume}} + (1-k)*\text{Couleur}_{\text{Phong}}$$

✍ Programmer l'ajout de cette brume.

Vous noterez qu'il est difficile de régler la force de cet effet car il dépend de la scène.

Anticrénelage (*antialiasing*)

Les bords des objets ne sont pas très élégants : il y a des marches d'escaliers. A quoi est-ce dû très précisément ?

Pour corriger le problème, on emploie généralement la technique du sur-échantillonnage : on lance plusieurs rayons par pixel de l'image.

C’est à dire que lors du dessin d’un pixel, on effectue 2x2 ou 3x3 (ou davantage) lancers de rayons très légèrement décalés à l’intérieur du même pixel et on calcule la couleur moyenne. Ainsi, s’il y a une frontière entre deux éléments (objet/ciel), elle sera adoucie. La couleur du pixel sera la couleur majoritaire mais intermédiaire entre les éléments.

Soit un pixel (xe, ye). On veut le subdiviser en n*n sous-pixels et calculer la couleur moyenne. Le traitement se fait avec deux boucles imbriquées : pour dx=0 à n-1, pour dy=0 à n-1, lancer un rayon sur (xe+dx???, ye+dy???) : quelles sont les coordonnées exactes qu’il faut employer ?

✍ Programmer ce calcul dans la méthode `TracerImage` de la classe `Lancer`. On pourrait le coder dans `CouleurPixel` mais alors le dessin initial en mode brouillon ne serait plus si rapide.

Une amélioration intéressante consiste à ne lancer que 4 rayons, dans les 4 coins du pixel, dans un premier temps et regarder si leur couleur diffère beaucoup. Si c’est le cas, on continue avec les rayons restants, mais sinon, on se contente de la moyenne des 4 couleurs, ce qui accélère énormément les calculs sur des plages homogènes.

Reflet spéculaire de Blinn

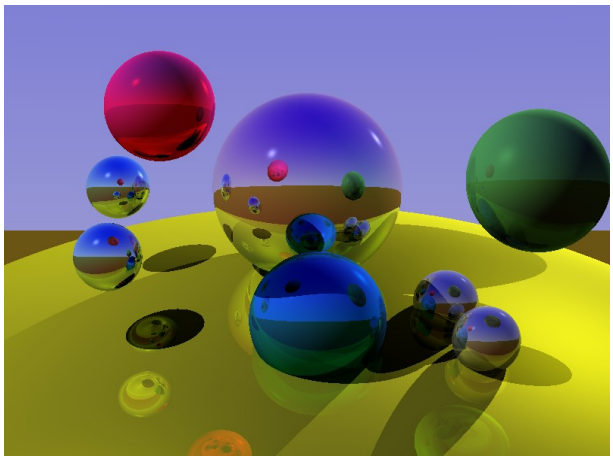
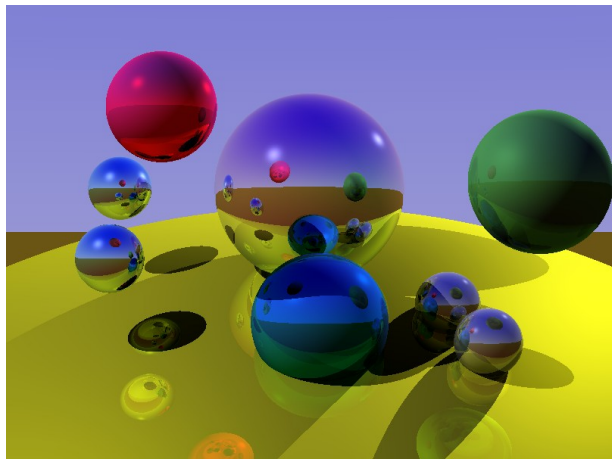
Blinn a proposé un autre calcul pour le reflet spéculaire. Au lieu de calculer $(R.-V)^{ns} * Ks * K_L$, Blinn introduit un vecteur *H* (half-vector) qui est sur la bissectrice entre *L* et *-V*. Il suffit de les additionner et normaliser le résultat. Ensuite, il calcule la teinte du reflet spéculaire par :

$$Couleur_{spéculaire} = (N.H)^{ns} * Ks * K_L$$

Sa méthode est plus rapide que celle de Phong car il ne nécessite pas le calcul d’un vecteur miroir par rapport à *N*.

D’autre part, sur le plan physique, sa méthode semble meilleure. Examinez les reflets des lampes sur cette image suivante. Notez que la réflexion se produit sur une surface plane, ce qui n’est pas le cas des sphères.

Comparez les reflets spéculaires de ces deux images, lesquels semblent les plus réalistes ? Vous regarderez les boules de bowling différemment maintenant…

NB : pour une même valeur *ns*, les reflets de Blinn sont plus étalés que ceux de Phong. Il faudrait calculer $(N.H)^{2ns}$ pour obtenir approximativement le même résultat visuel.