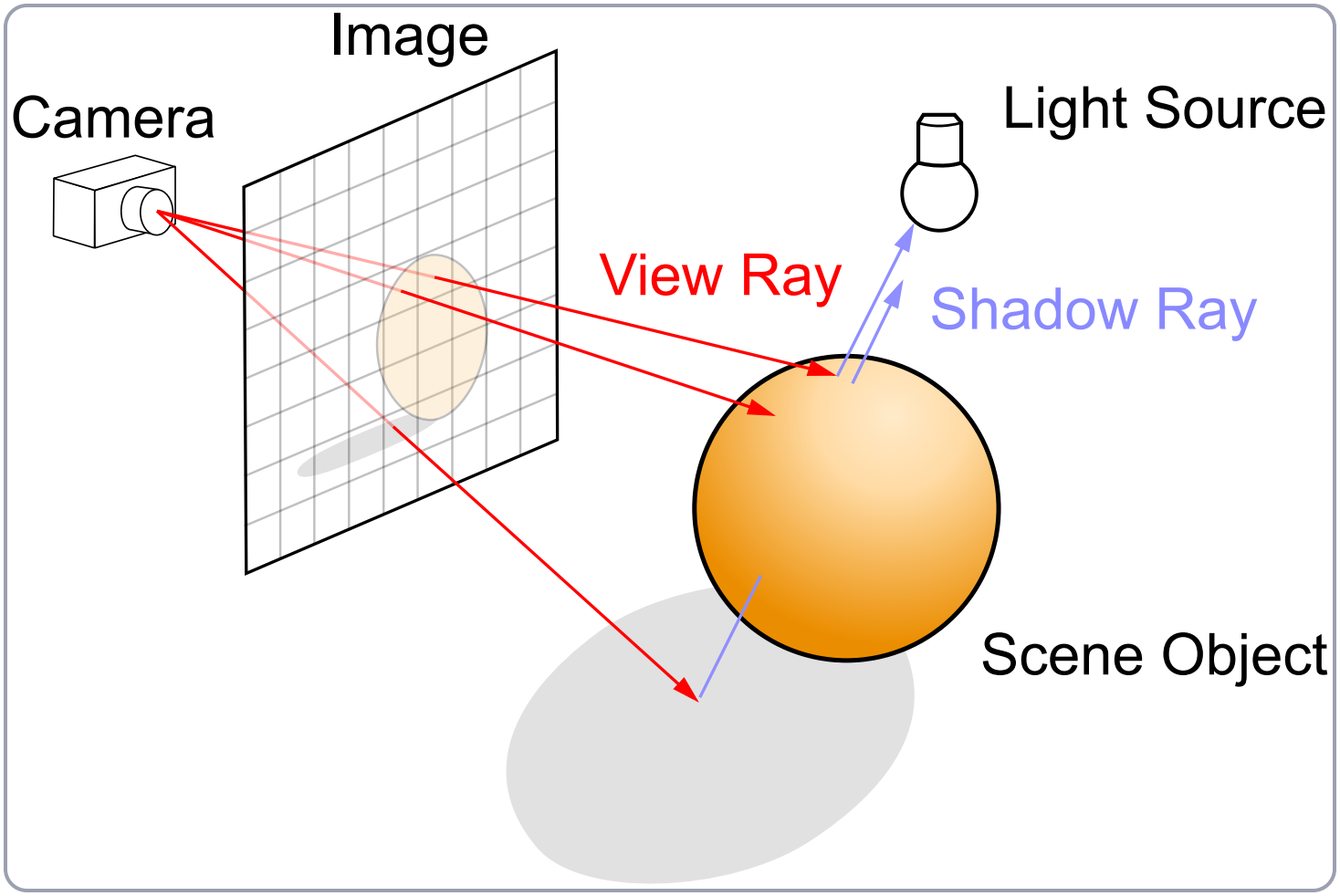
# 3 - Premières images

## Objectif

Le but de cette nouvelle étape est de mettre en place la **boucle principale du lanceur de rayons**, avec la détection des intersections, sans avoir à se préoccuper pour l'instant de la couleur des pixels. On utilisera un pixel noir quand un rayon ne touche aucun objet et un pixel de la couleur de l'objet touché dans le cas contraire.

Cela correspondra, dans le schéma suivant, au calcul de l'intersection des rayons en rouge (View Ray) avec la sphere.




Dans cette première partie, on se limitera aux sphères.

# Spécification

## Pseudo-code du lanceur de rayon

Le pseudo-code du lanceur de rayons est le suivant :

```
charger le fichier de scène

pour chaque pixel (i,j) de l'image à générer faire
    calculer le vecteur unitaire d qui représente
          un rayon allant de l'oeil/camera au centre du pixel (i,j)
    rechercher le point d'intersection p
          le plus proche avec un objet de la scène
    si p existe
        alors calculer sa couleur
        sinon utiliser du noir
    peindre le pixel (i,j) avec la couleur adéquate

sauvegarder l'image
```

## Calcul du repère orthonormé $(\vec{u}, \vec{v}, \vec{w})$

Le vecteur réprésentant la direction du rayon de l'oeil au pixel de l'image doit être représenté dans un repère orthonormé $(\vec{u}, \vec{v}, \vec{w})$.

- $\vec{up}$ correspond au vecteur direction de la caméra (**fourni dans le fichier de description de scène**).

**À implémenter**

- $\vec{w}$ correspond à l'axe passant par l'oeil (*lookFrom*) et le point regardé (*lookAt*). C'est vecteur défini par deux points. Il se calcule de la façon suivante :

$$\vec{w} = \frac{lookFrom - lookAt}{||lookFrom - lookAt||}$$

- Il est possible de calculer une normale $\vec{u}$ au plan formé par les vecteurs $\vec{up}$ et $\vec{w}$ à l'aide du produit vectoriel (noté $\times$).

$$\vec{u} = \frac{\vec{up} \times \vec{w}}{||\vec{up} \times \vec{w}||}$$

- Enfin, à partir de $\vec{u}$ et $\vec{w}$, on peut calculer $\vec{v}$ :

$$\vec{v} = \frac{\vec{w} \times \vec{u}}{||\vec{w} \times \vec{u}||}$$

## Calcul des dimensions d'un pixel dans le repère $(\vec{u}, \vec{v}, \vec{w})$

- Les dimensions d'un pixel dans la scène varient avec l'angle de vue $fov$ (*field of view*) .
- On notera $imgwidth$ et $imgheight$ les dimensions de l'image en pixels.
- $fov$ étant en degrés, il doit être transformé en radians ($fovr$) :

> **À implémenter**
>
> $$fovr = \frac{fov * \pi}{180}$$

- **On suppose dans notre cas que l'image en pixels se trouve à une unité de l'oeil sur l'axe $\vec{w}$.**

> **À implémenter**
>
> - La hauteur d'un pixel dans la scène est donc de
>
> $$pixelheight = tan(\frac{fovr}{2})$$
>
> - La largeur du pixel doit respecter le ratio $\frac{imgwidth}{imgheight}$.
>
> $$pixelwidth = pixelheight * \frac{imgwidth}{imgheight}$$

## Calcul du vecteur direction $\vec{d}$ pour un pixel (i,j)

- Les coordonnées d'une image vont de `(0,0)` dans le coin **supérieur gauche** à `(imgwidth,imgheight)` au coin **inférieur droit** de l'image.

> **Attention**
>
> C'est une spécificité Java qui nécessitera d'inverser les pixels des colonnes (haut/bas) lors du rendu de l'image, sinon vous aurez des images à l'envers.

- Notre scène est centrée en $(\frac{imgwidth}{2}, \frac{imgheight}{2})$.
  - On doit donc traduire les coordonnées du pixel $(i, j)$ dans $(u, v)$ en les translatant de $(-\frac{imgwidth}{2}, -\frac{imgheight}{2})$.
- De plus, on cherche à viser le centre du pixel, donc on translatera la direction de $(0.5, 0.5)$.
- Enfin, on normalise chaque valeur.

**À implémenter**

- Au final on obtient :

$$a = \frac{pixelwidth * (i - \frac{imgwidth}{2} + 0.5)}{\frac{imgwidth}{2}}$$

$$b = \frac{pixelheight * (j - \frac{imgheight}{2} + 0.5)}{\frac{imgheight}{2}}$$

- On obtient $\vec{d}$ avec l'équation :

$$\vec{d} = \frac{\vec{u} * a + \vec{v} * b - \vec{w}}{||\vec{u} * a + \vec{v} * b - \vec{w}||}$$

## Calcul de l'intersection d'un rayon avec une sphère

Pour l'instant, nous supposons que les seuls objets présents sur la scène sont des sphères. Elles sont définies par

- un centre $c$ (point)
- un rayon $r$ (scalaire)

On définit également :

- $o$ : point représentant l'oeil ou la caméra (`lookFrom`)
- $\vec{d}$ : le vecteur direction calculé à l'étape précédente
- $t$ : scalaire qui représente la distance entre l'oeil et le point d'intersection le plus proche du rayon dirigé par $\vec{d}$ avec la sphere

Par ailleurs, une sphère est définie par l'ensemble des points $p$ tels que $||p - c|| = r$.

- Nous cherchons donc un point $p$ qui réponde aux deux équations :
  - $p = o + \vec{d} * t$
  - $||p - c|| = r$
- Soit :

$$(p - c) \cdot (p - c) - r^2 = 0$$

$$(o - c + \vec{d} * t) \cdot (o - c + \vec{d} * t) - r^2 = 0$$

**À implémenter**

Il s'agit d'une équation du second degré en $t$ de la forme $a * t^2 + b * t + c = 0$ avec

- $a = \vec{d} \cdot \vec{d}$
- $b = 2 * (o - c) \cdot \vec{d}$
- $c = (o - c) \cdot (o - c) - r^2$

On calcule le discriminant $\Delta = b^2 - 4 * a * c$.

- Si $\Delta$ est négatif, pas d'intersection.
- Si $\Delta$ est nul, alors il y a une seule intersection avec $t = \frac{-b}{2*a}$.
- Si $\Delta$ est positif, alors il y a deux intersections
  $t_1 = \frac{-b+\sqrt{\Delta}}{2*a}$ et $t_2 = \frac{-b-\sqrt{\Delta}}{2*a}$.

$t_2$ est plus proche (petit) que $t_1$ mais $t_2$ pourrait être négatif.

Si $t_2$ est positif l'intersection la plus proche est $t_2$ sinon si $t_1$ est positif l'intersection la plus proche est $t_1$ sinon il n'y a pas d'intersection dans le champs de l'image.

**Rappel** : la soustraction de deux points donne un vecteur (exemple $(o - c)$)

## Calcul de la couleur du point d'intersection

Cette semaine, on ne se pose pas la question de calculer une valeur correcte pour la lumière. On utilisera tout simplement la lumière ambiante de la scène (valeur de l'attribut `ambient` dans le fichier de description de scène) comme couleur du point d'intersection.

# Aide

Bon courage, ce jalon est un peu plus complexe.

## 1 - Implémentation des formules

Cette semaine, beaucoup de formules à implémenter. C'est ici que votre travail du Jalon 1 (calcul vectoriel) va servir.

- N'hésitez pas à faire un schéma de la situation (avec une caméra, une sphere, l'image et les noms des différents éléments) pour vous aider à bien comprendre.
- Procédez par étapes, écrivez minutieusement les opérations et relisez-vous bien
- Suivez le pseudo-code en haut des spécifications
- Faites une méthode par formule, nommez-les correctement
- N'hésitez pas à ajouter des tests unitaires pour valider vos formules

## 2 - Calculer le point d'intersection *le plus proche* ?

Il va falloir une méthode pour calculer le point d'intersection le plus proche, car c'est le seul qui sera visible. Imaginez que plusieurs objets se suivent les uns derrière les autres, tous sur la trajectoire de votre rayon. Seul celui tout devant sera visible et il masquera les autres.

Il faut donc trouver le point d'intersection dont la distance à l'oeil soit la plus petite (mais toujours positive, sinon le point est derrière vous), et ce après avoir calculé toutes les intersections possibles du rayon.

Si on récapitule :

- on calcule les intersections possibles avec tous les objets de la liste `shapes` de notre scène
- pour chaque objet on prend le point le plus proche s'il existe (la racine positive la plus petite de notre équation du second degré, pour la sphère)
- puis on ordonne toutes les intersections trouvées, en retirant celles dont la distance est négative
- et on prend la plus petite d'entre elles. C'est elle qui nous servira à calculer la couleur du pixel (même si pour le moment ce sera `ambiant`, cette couleur)

## 3 - Classes à créer

Vous aurez encore quelques classes à créer cette fois ci.

- Tout d'abord, il va falloir créer votre classe traceur de rayons : `RayTracer`. Elle servira à calculer la couleur de chaque pixel grâce à une méthode `getPixelColor`.
- Il faudra créer une classe pour le rendu de votre image, appelée dans `main` qui
  - aura une méthode `render` qui prend une `Scene` en paramètre et calcule le rendu
  - cette méthode interrogera votre `RayTracer` dans sa boucle pour dessiner l'image et la retourner à votre `main`
- Le repère orthonormé serait une classe pertinente : `Orthonormal`
- Vous pourrez également (mais pas obligatoirement) créer une classe pour écrire le fichier. Cette classe sera utilisée dans `main`
- Il pourrait être pratique de créer une classe `Intersection` pour stocker les informations concernant les intersections trouvées.
- Pour finir, une classe `Ray`, comportant un rayon pourrait être utile. Un rayon a un `Point` comme origine et un `Vector` comme direction

## 4 - Où implémenter les formules ?

- Les coordonnées du repère orthonormé dans le constructeur de `Orthonormal`
- Les dimensions des pixels servent à calculer le vecteur direction du rayon. Et le traçage de rayons est le rôle du `RayTracer`. Celui-ci devra calculer le rayon pour un pixel.
- Le calcul de l'intersection la plus proche doit être fait par un objet qui a connaissance de tous les élements présents. Et quoi de mieux que la `Scene` pour ça ?

- En revanche, le calcul de la distance du point d'intersection avec chaque objet doit être délégué à la `Shape` (ici sphère) concernée : c'est elle qui saura quelle équation utiliser selon sa propre forme.

## 5 - Gérer les intersections inexistantes

Pour les cas où une intersection n'existe pas, il peut être pratique d'utiliser le type paramétrable `Optional<Intersection>`.

`Optional` est une classe permettant de représenter des objets facultatifs : soit un objet, soit pas d'objet.

Pour l'utiliser, il faut importer la classe : `import java.util.Optional;`

Vous pouvez consulter la documentation d'`Optional` en ligne. Voici les méthodes dont nous avons besoin :

- `static Optional<T> empty()` : pour créer une option sans objet
- `static Optional<T> of(T t)` : pour créer une option contenant l'objet `t`
- `boolean isPresent()` : pour vérifier si l'option contient un objet
- `T get()` : pour récupérer l'objet présent dans l'option.