

A still life photograph featuring several glasses filled with different colored liquids (red, yellow, and clear). In the foreground, a large, clear ice cube sits in a glass. Two blue dice are also visible on the surface. The background is softly blurred, showing more glasses and a green object. The overall lighting is warm and focused on the central elements.

**IFT 3355: INFOGRAPHIE**

# **L'ÉCLAIRAGE GLOBAL**

Livre de référence: G: 20, S: 4

<http://tinyurl.com/ift3355>

Mikhail Bessmeltsev

# LES MODÈLES ET ALGORITHMES D'ÉCLAIRAGE

## L'éclairage local - rapide

- Plus loin de la physique
- Approximer l'apparence

L'interaction de chaque objet avec la lumière



## L'éclairage global – lent

- Plus proche de la physique
- Les interactions entre les objets

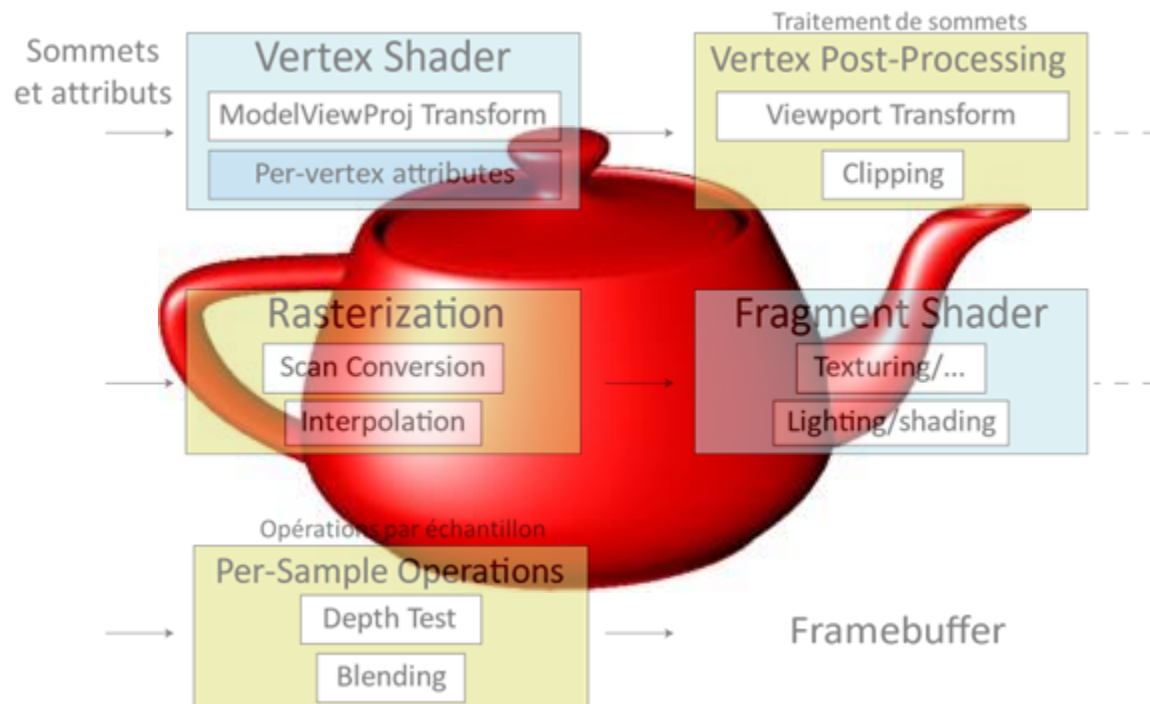


# LES MODÈLES ET ALGORITHMES D'ÉCLAIRAGE

## L'éclairage local - rapide

- Plus loin de la physique
- Approximer l'apparence

L'interaction de chaque objet avec la lumière



## L'éclairage global – lent

- Plus proche de la physique
- Les interactions entre les objets

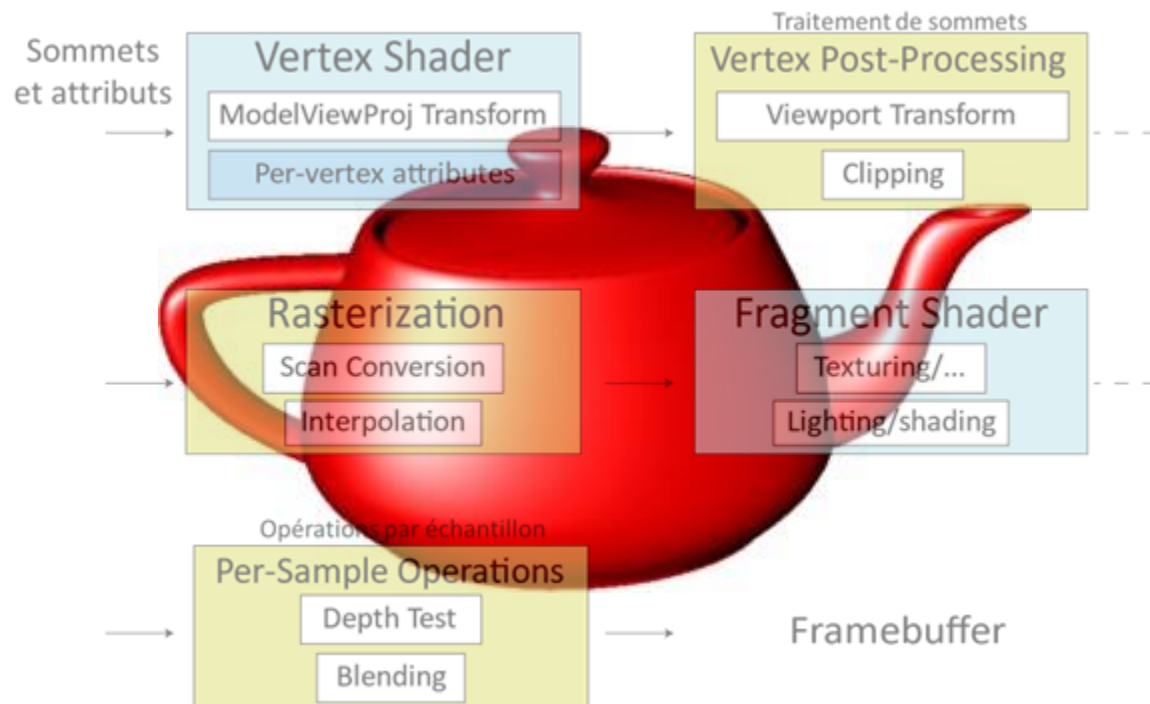


# LES MODÈLES ET ALGORITHMES D'ÉCLAIRAGE

## L'éclairage local - rapide

- Plus loin de la physique
- Approximer l'apparence

L'interaction de chaque objet avec la lumière



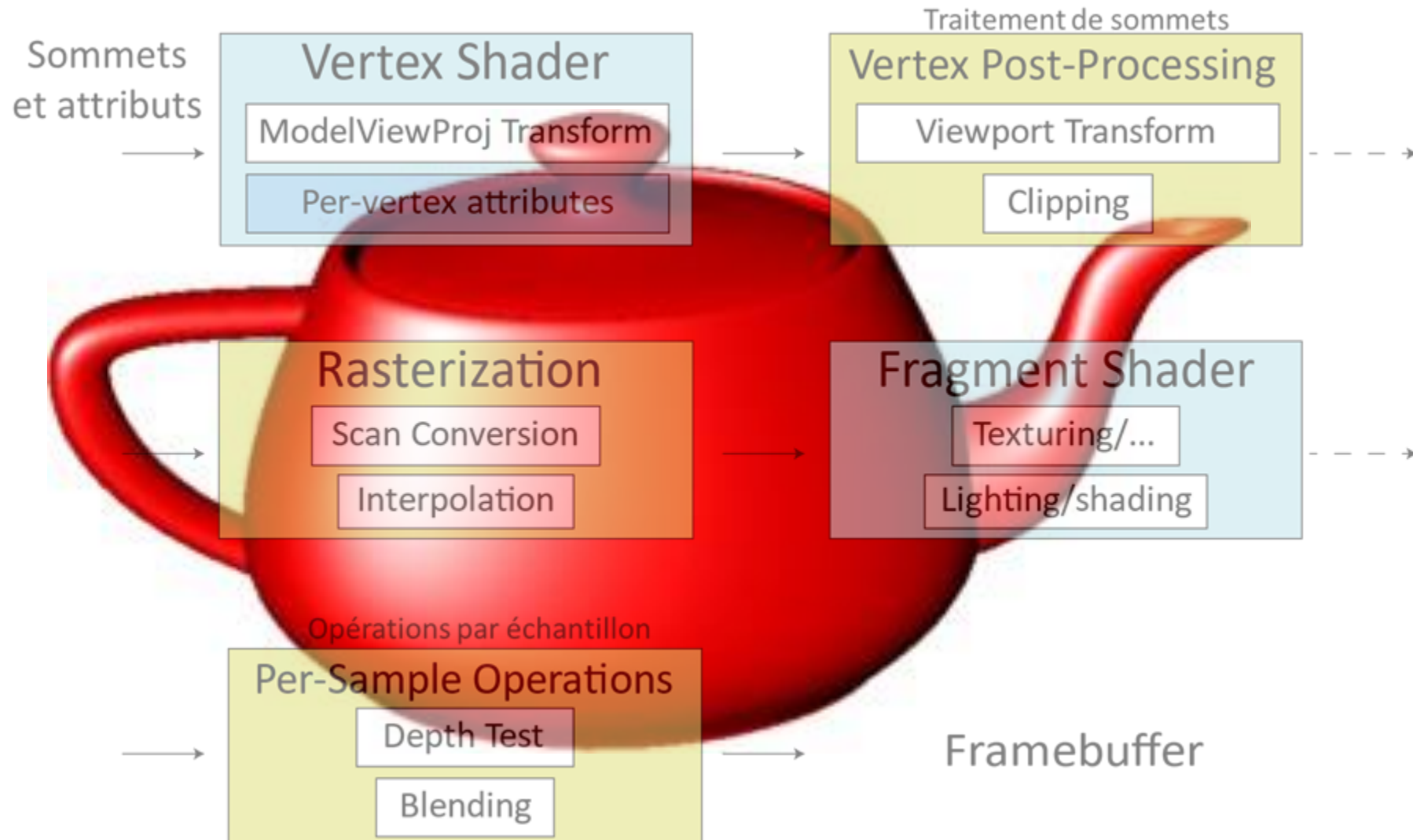
## L'éclairage global – lent

- Plus proche de la physique
- Les interactions entre les objets

Comment?



# QU'EST-CE QUI EST NON PHYSIQUE DANS L'ÉCLAIRAGE LOCAL?



# LES ALGORITHMES DE L'ÉCLAIRAGE GLOBAL

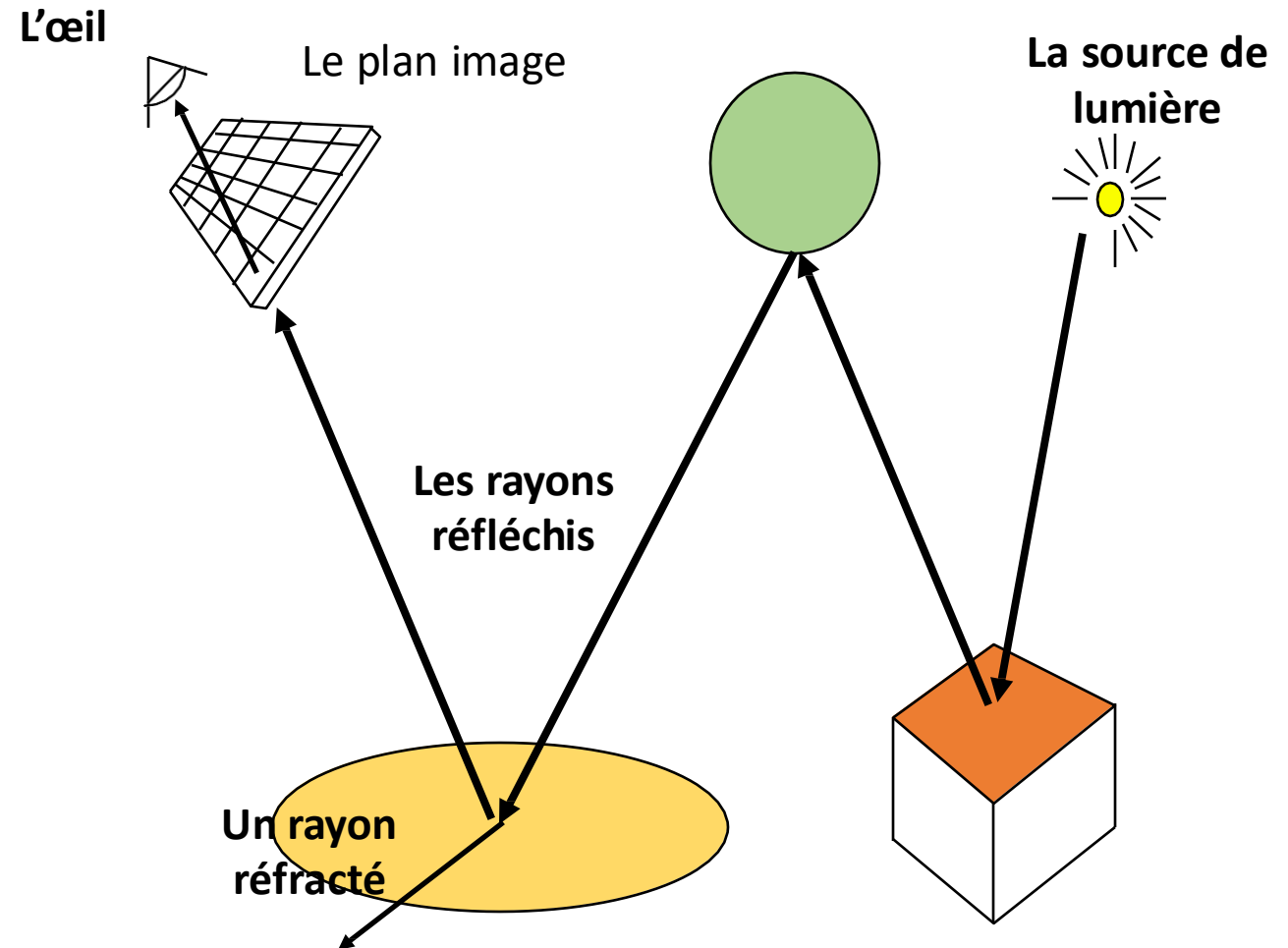
- Lancer de rayons (*Raytracing*)
- Tracer de chemins (*Path tracing*)
- *Photon Mapping*
- Radiosité (*Radiosity*)
- *Metropolis light transport*
- ...

**COMMENT L'ÉCLAIRAGE GLOBAL DEVRAIT-IL  
FONCTIONNER?**

# COMMENT L'ÉCLAIRAGE GLOBAL DEVRAIT-IL FONCTIONNER?

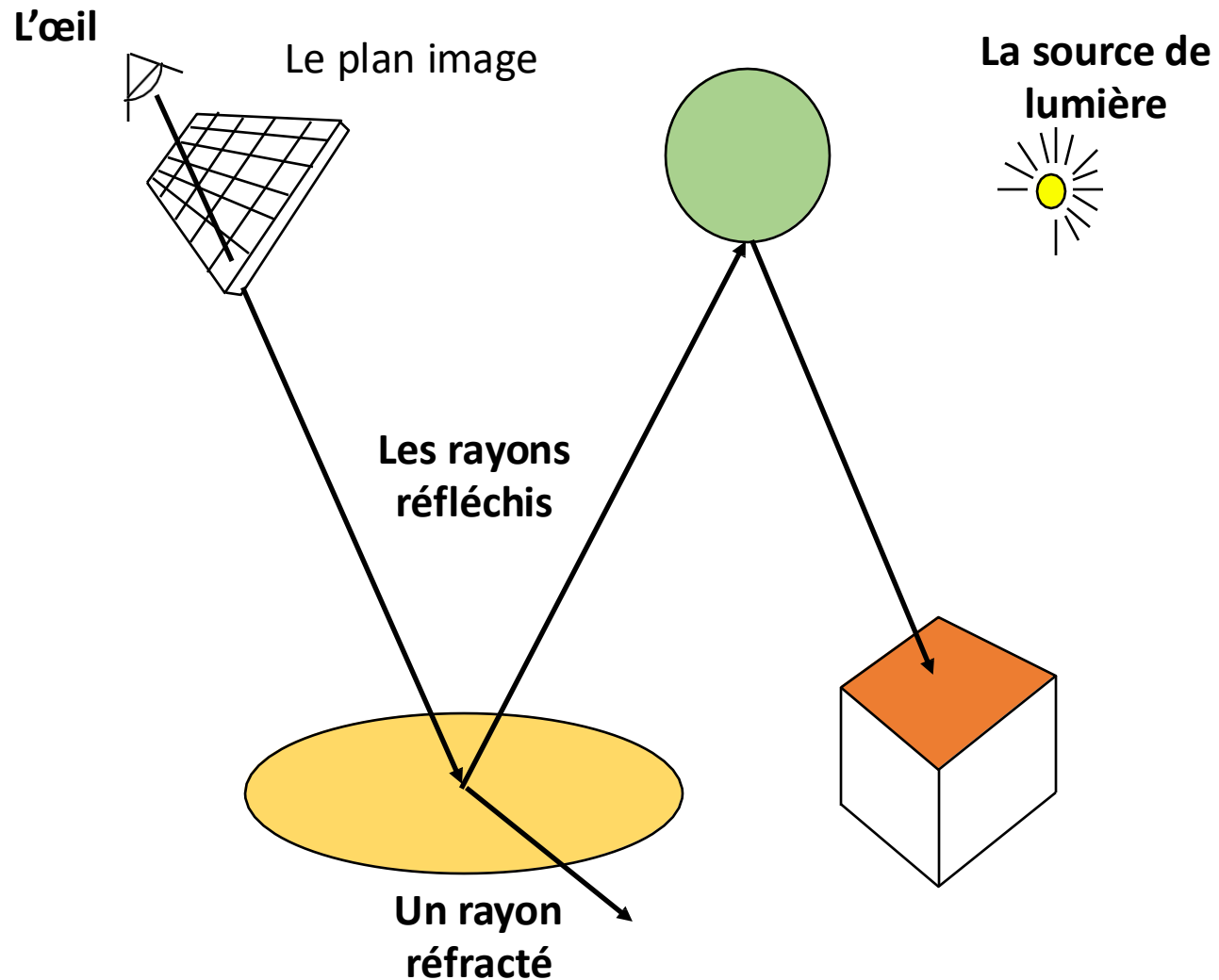
Simuler la lumière

- Telle qu'elle est émise par les sources de lumière
- Telle qu'elle rebondit sur les surfaces / est absorbée / réfléchie / réfractée
- Telle qu'elle atteint la caméra

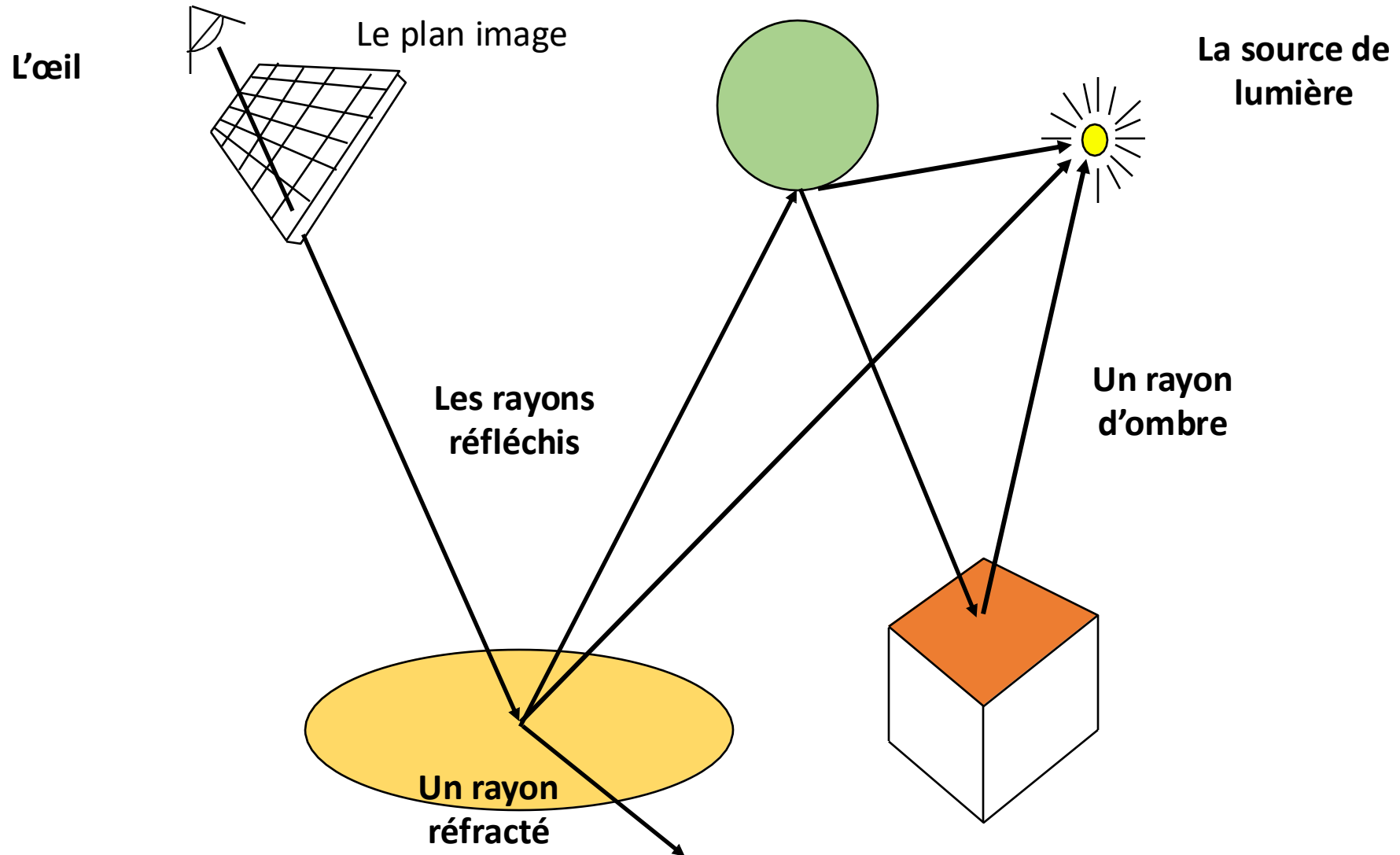


**DES PROBLÈMES?**

# L'IDÉE DU LANCER DE RAYONS



# L'IDÉE DU LANCER DE RAYONS

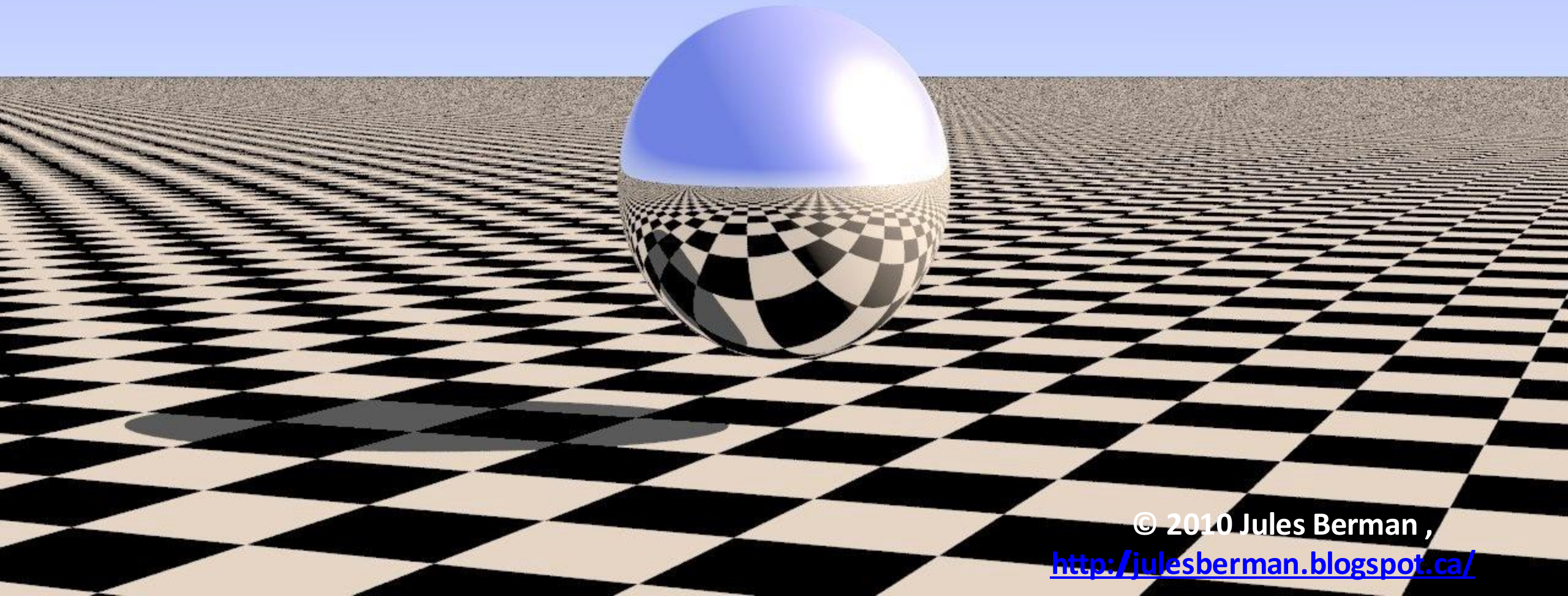
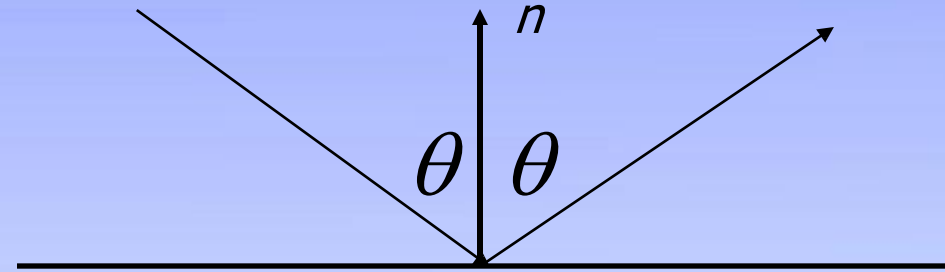


# LANCER DE RAYONS

- Inverser les directions de rayons!
- Lancer les rayons de la caméra à travers chaque pixel
  - "Tracer les rayons à l'envers"
- Simuler ce que les rayons font:
  - Réflexion
  - Réfraction
  - ...
- Chaque interaction d'un rayon avec un objet contribue à la couleur finale
- La majorité des rayons ne vont pas intersecter la lumière
  - Lancer les "*shadow rays*" pour calculer l'illumination directe

# RÉFLEXION

- Les effets d'un miroir
  - La réflexion spéculaire



# RÉFRACTION

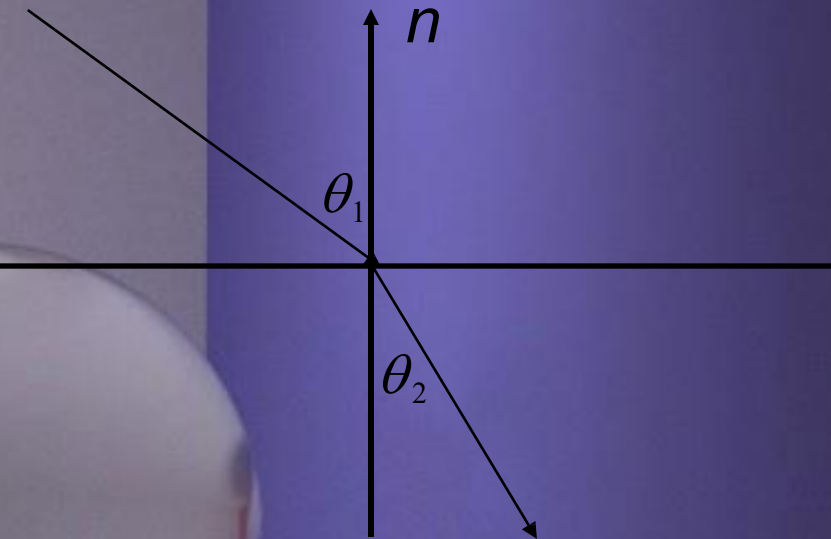
- L'interface entre l'objet non-opaque et l'environnement

- E.g. la frontière entre le verre et l'air

- Le rayon change sa direction selon les indices de réfraction  $c_1$ ,  $c_2$

La loi de  
Snell-Descartes

$$c_2 \sin \theta_1 = c_1 \sin \theta_2$$



# LANCER DE RAYONS: L'ALGORITHME DE BASE

```
RayTrace(r,scene)
obj = FirstIntersection(r,scene)

if (no obj) return BackgroundColor;
else {
    if (Reflect(obj))
        reflect_color = RayTrace(ReflectRay(r,obj));
    else
        reflect_color = Black;

    if (Transparent(obj))
        refract_color = RayTrace(RefractRay(r,obj));
    else
        refract_color = Black;

    return Shade(reflect_color, refract_color, obj);
}
```

# QUAND ARRÊTER?

- Cet algorithme ne se termine pas
- Les critères de terminaison
  - Aucune intersection
  - La contribution des rayons secondaires est inférieure à un seuil
  - La profondeur maximale / nombre de générations est atteinte

# SOUS-PROGRAMMES

- `ReflectRay(r,obj)` – calculer le rayon réfléchi (utiliser la normale de l'objet à l'intersection)
- `RefractRay(r,obj)` – calculer le rayon réfracté
  - NB: le rayon peut être à l'intérieur de l'objet
- `Shade(reflect_color,refract_color,obj)` – calculer l'éclairage direct

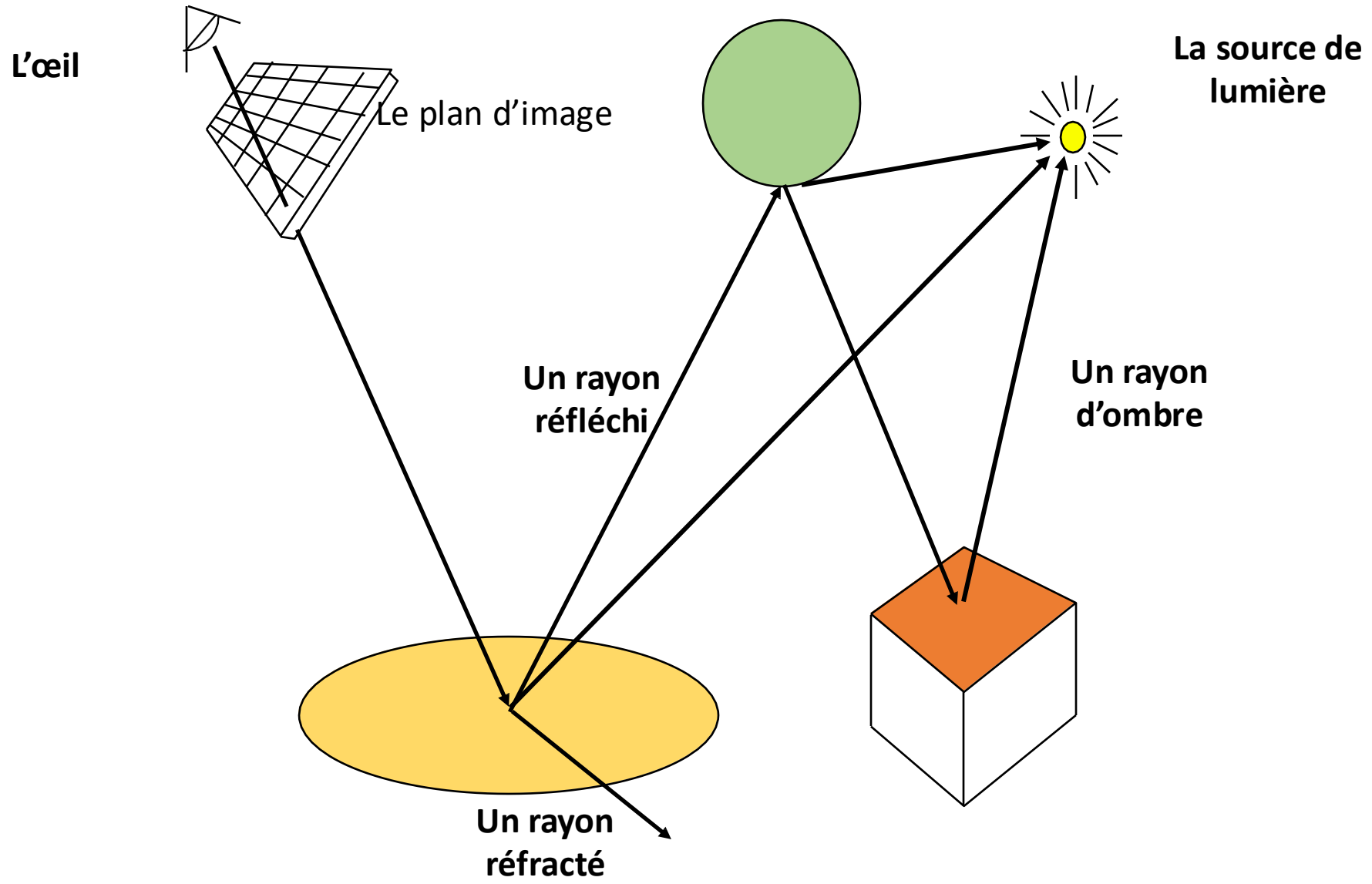
# LES OMBRES

- Tracer un rayon de l'objet à la source de lumière
  - Si le rayon intersecte quelque chose  $\Rightarrow$  le point est dans l'ombre

```
shadow = RayTrace(LightRay(obj,r,light));
```

```
return Shade(shadow,reflect_color,refract_color,obj);
```

# L'IDÉE DU LANCER DE RAYONS

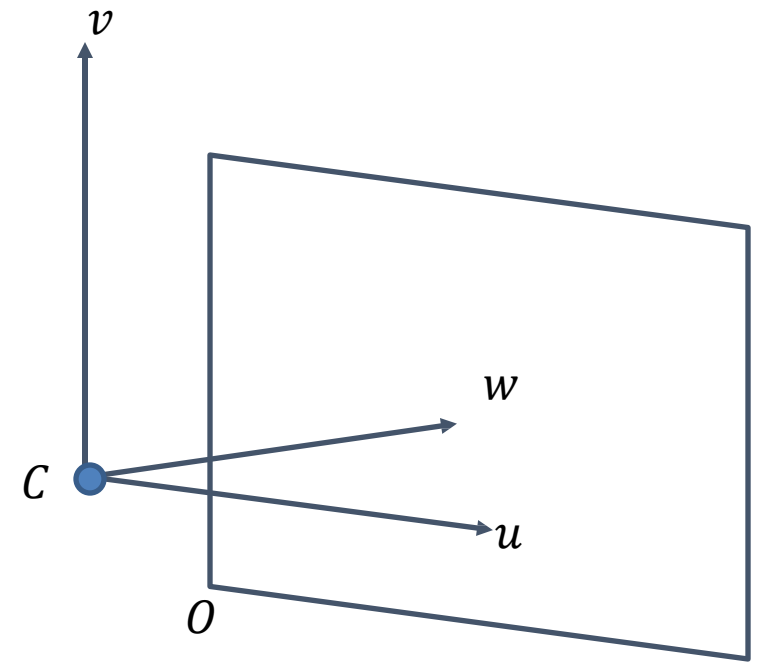


# LANCER DE RAYONS: LES DÉTAILS PRATIQUES

- Génération de rayons
- Les intersections entre les rayons et les objets
- Les transformations géométriques
- L'éclairage et *shading*
- La vitesse: Réduire le nombre de tests d'intersections
  - E.g. utiliser les arbres BSP ou les autres types de la partition de l'espace

# LANCER DE RAYONS: GÉNÉRATION DE RAYONS

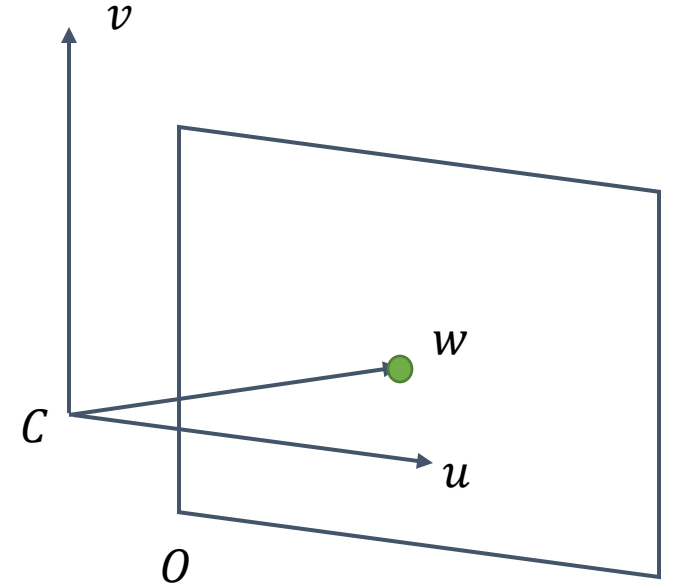
- Le système de coordonnées de la caméra
  - L'origine:  $C$  (la position de la caméra)
  - La direction de la vue:  $w$
  - Le vecteur vers le haut:  $v$
  - $u = w \times v$
- Similaire à la transformation de la vue



# LANCER DE RAYONS: GÉNÉRATION DE RAYONS

- La distance au plan image:  $d$
- Les dimensions physiques du plan:  $l, r, t, b$
- Coin:

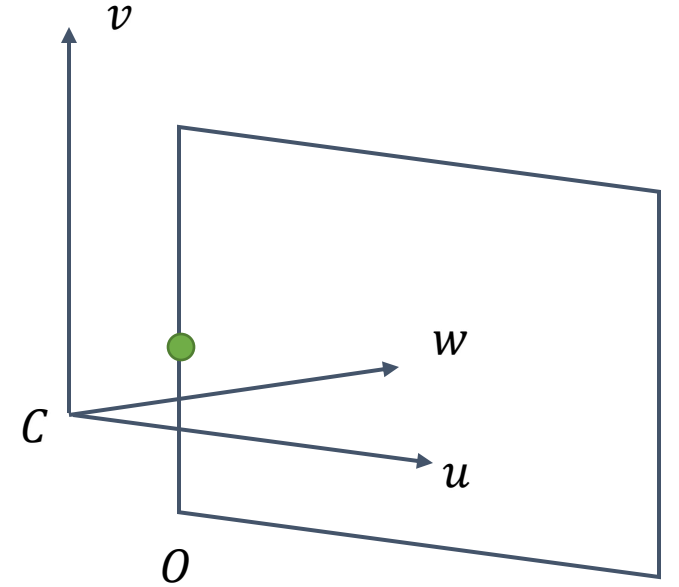
$$\mathbf{O} = \mathbf{C} + d\vec{\mathbf{w}} + l\vec{\mathbf{u}} + b\vec{\mathbf{v}}$$



# LANCER DE RAYONS: GÉNÉRATION DE RAYONS

- La distance au plan image:  $d$
- Les dimensions physiques du plan:  $l, r, t, b$
- Coin:

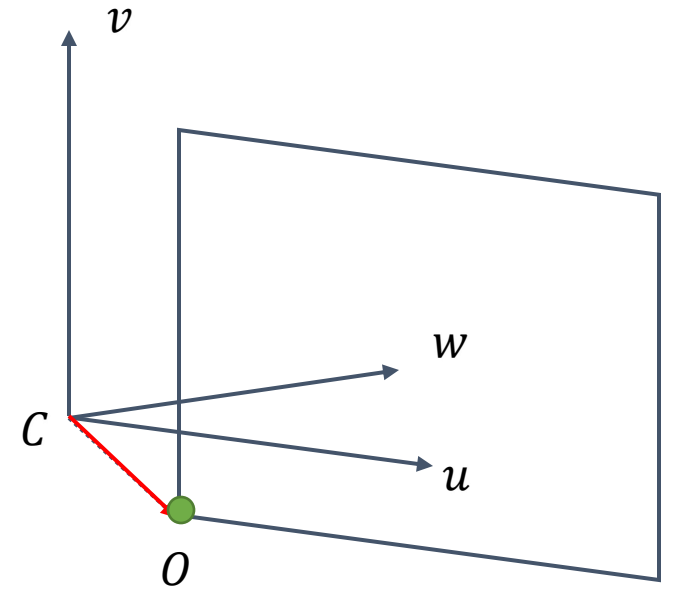
$$\mathbf{O} = \mathbf{C} + d\vec{w} + l\vec{u} + b\vec{v}$$



# LANCER DE RAYONS: GÉNÉRATION DE RAYONS

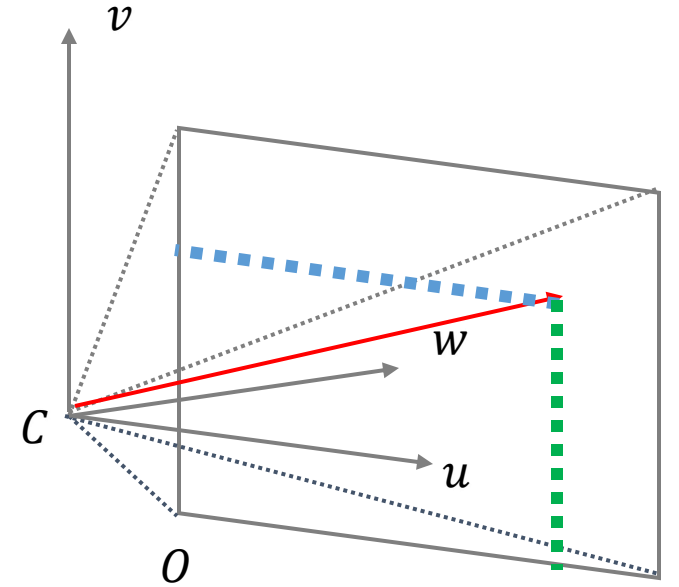
- La distance au plan image:  $d$
- Les dimensions physiques du plan:  $l, r, t, b$
- Coin:

$$\mathbf{O} = \mathbf{C} + d\vec{w} + l\vec{u} + b\vec{v}$$



# LANCER DE RAYONS: GÉNÉRATION DE RAYONS

- La distance au plan image:  $d$
- Les dimensions physiques du plan:  $l, r, t, b$
- Coin:  $\mathbf{O} = \mathbf{C} + d\vec{w} + l\vec{u} + b\vec{v}$
- La résolution d'image (pixels):  $N_x, N_y$
- Pixel à la position  $i, j$  ( $i = 0, \dots, N_x - 1; j = 0, \dots, N_y - 1$ )



$$\begin{aligned} P_{i,j} &= O + (i + 0.5) \cdot \frac{r - l}{N_x} \cdot \vec{u} + (j + 0.5) \cdot \frac{t - b}{N_y} \cdot \vec{v} \\ &= O + (i + 0.5) \cdot \Delta u \cdot \vec{u} + (j + 0.5) \cdot \Delta v \cdot \vec{v} \end{aligned}$$

# LANCER DE RAYONS: GÉNÉRATION DE RAYONS

- L'équation paramétrique d'un rayon:

$$\boxed{?} \quad R_{i,j}(t) = C + t \cdot (P_{i,j} - C) = C + t \cdot \mathbf{v}_{i,j}$$

où  $t = 0 \dots$

# LANCER DE RAYONS: LES DÉTAILS PRATIQUES

- Génération de rayons
- ***Les intersections entre les rayons et les objets***
- Les transformations géométriques
- L'éclairage et *shading*
- La vitesse: Réduire le nombre de tests d'intersection
  - E.g. utiliser les arbres BSP ou les autres types de la partition de l'espace

# LES INTERSECTIONS

- Dans le pipeline OpenGL, nous étions limités à des objets discrets
  - Les maillages de triangles
- Ici, on peut utiliser des formes analytiques
  - Presqu'aucun problème avec l'interpolation des normales

# LES INTERSECTIONS

- Le cœur du lancer de rayons  $\Rightarrow$  Elles doivent être calculées très rapidement
- Normalement, elles impliquent de résoudre des équations implicites

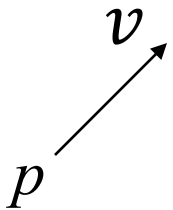
**Exemple:** L'intersection entre un rayon et une sphère

Un rayon:  $x(t) = p_x + v_x t$ ,  $y(t) = p_y + v_y t$ ,  
 $z(t) = p_z + v_z t$

Une sphère unitaire:  $x^2 + y^2 + z^2 = 1$

L'équation quadratique en  $t$ :

$$\begin{aligned} 0 &= (p_x + v_x t)^2 + (p_y + v_y t)^2 + (p_z + v_z t)^2 - 1 = \\ &= t^2(v_x^2 + v_y^2 + v_z^2) + 2t(p_x v_x + p_y v_y + p_z v_z) \\ &\quad + (p_x^2 + p_y^2 + p_z^2) - 1 \end{aligned}$$



# INTERSECTIONS AVEC D'AUTRES PRIMITIVES

- Les fonctions implicites:
  - Les sphères et les sections de coniques (hyperboloïdes, ellipsoïdes, paraboloides, cônes, cylindres)
    - Toutes sont décrites par des équations quadratiques
  - Les fonctions d'ordre supérieur (e.g. tores et les autres surfaces quartiques)
    - Sont similaires, mais trouver les racines est plus difficile
    - Les méthodes numériques

# LES INTERSECTIONS AVEC LES AUTRES PRIMITIVES

- Les polygones:
  - D'abord, intersecter le rayon avec le plan
    - La fonction implicite est linéaire
  - Après, tester si le point est à l'intérieur ou à l'extérieur (test 2D)
- Pour les polygones convexes
  - Tester si le point est du même côté de chaque ligne (segment)

# LANCER DE RAYONS: LES DÉTAILS PRATIQUES

- Génération de rayons
- Les intersections entre les rayons et les objets
- **Les transformations géométriques**
- L'éclairage et *shading*
- La vitesse: Réduire le nombre de tests d'intersection
  - E.g. utiliser les arbres BSP ou les autres types de la partition de l'espace

# LANCER DE RAYONS: TRANSFORMATIONS

- Remarque: les rayons remplacent la projection perspective
- Les transformations géométriques:
  - Le même but que dans le pipeline:
    - Modéliser la scène avec les objets dans leurs systèmes de coordonnées
  - Un problème:
    - Certaines représentations de la géométrie sont difficiles à transformer
      - Dans le pipeline, nous n'avons que les polygones

# LANCER DE RAYONS: TRANSFORMATIONS

- Les transformations de rayons:
  - Pour un test d'intersections, il est important que les rayons et les objets soient dans un même système de coordonnées
  - Transformer tous les rayons dans le système de coordonnées de l'objet
    - Transformer la position de la caméra et la direction du rayon par l'inverse de la matrice *modelview*
- Le *shading* est fait normalement dans les coordonnées du monde
  - Transformer le point du système de coordonnées de l'objet dans le système de coordonnées du monde
  - Il faut avoir les rayons dans les deux systèmes de coordonnées

# LANCER DE RAYONS: LES DÉTAILS PRATIQUES

- Génération de rayons
- Les intersections entre les rayons et les objets
- Les transformations géométriques
- **L'éclairage et *shading***
- La vitesse: Réduire le nombre de tests d'intersection
  - E.g. utiliser les arbres BSP ou les autres types de la partition de l'espace

# LANCER DE RAYONS: L'ÉCLAIRAGE DIRECT

- Les sources de lumière:
  - Pour le moment: les sources ponctuelles et directionnelles
  - On peut modéliser les lumières plus complexes
    - Les lumières surfaciques
    - Fluorescence

# LANCER DE RAYONS: L'ÉCLAIRAGE DIRECT

L'information de la surface locale (la normale...)

- Pour les surfaces implicites,  $F(x, y, z) = 0$ :  
la normale  $n(x, y, z)$  est le gradient de  $F$ :

$$n(x, y, z) = \nabla F(x, y, z) = \begin{pmatrix} \partial F(x, y, z) / \partial x \\ \partial F(x, y, z) / \partial y \\ \partial F(x, y, z) / \partial z \end{pmatrix}$$

- Exemple:

$$F(x, y, z) = x^2 + y^2 + z^2 - r^2$$

$$\mathbf{n}(x, y, z) = \begin{pmatrix} 2x \\ 2y \\ 2z \end{pmatrix}$$

Elle doit être normalisée

# LANCER DE RAYONS: L'ÉCLAIRAGE DIRECT

Pour les maillages de triangles

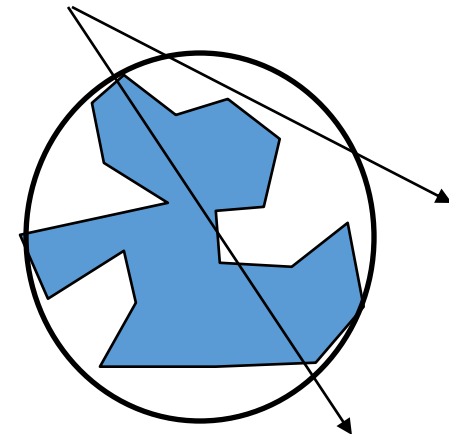
- Interpoler l'information par sommets
  - Phong shading!
  - Comme qu'avant
- La différence entre le pipeline:
  - Il faut calculer les coordonnées barycentriques pour chaque point d'intersection

# LANCER DE RAYONS: LES DÉTAILS PRATIQUES

- Génération de rayons
- Les intersections entre les rayons et les objets
- Les transformations géométriques
- L'éclairage et *shading*
- **La vitesse: Réduire le nombre de tests d'intersection**
  - E.g. utiliser les arbres BSP ou les autres types de la partition de l'espace

# LANCER DE RAYONS: OPTIMISÉ

- L'algorithme de base est simple mais trop lent
- Optimiser:
  - Réduire le nombre de rayons
  - Réduire le nombre de tests d'intersection
- Paralléliser
  - Cluster
  - GPU
- Les méthodes
  - Boîtes englobantes
  - La division de l'espace
    - Visibilité, Intersection/Collision
  - Élagage des arbres

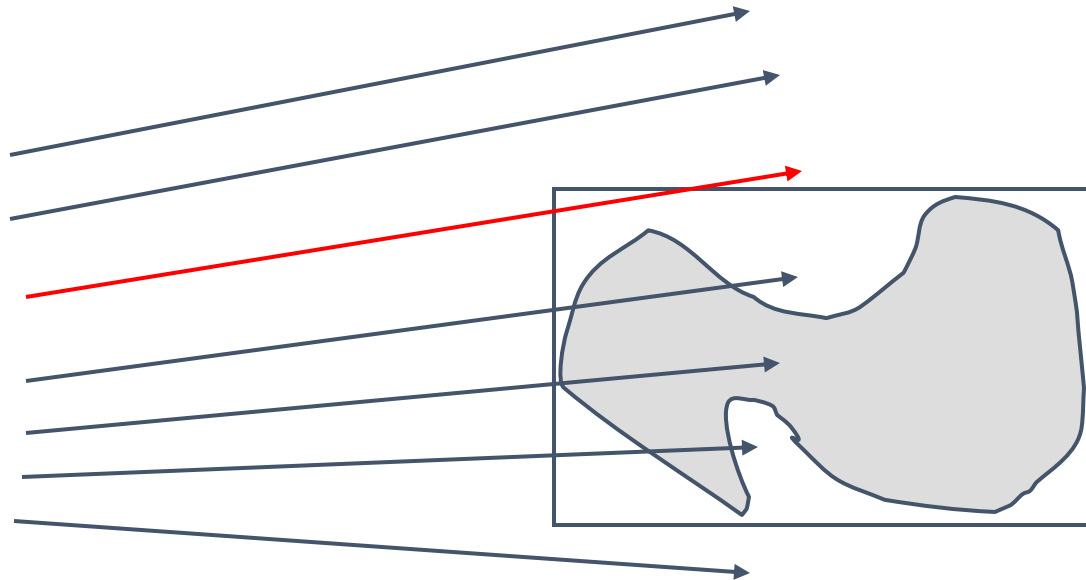


# LES STRUCTURE DE DONNÉES POUR LA SUBDIVISION DE L'ESPACE

- Le but: réduire le nombre d'intersections pour chaque rayon
- Beaucoup d'approches:
  - Hiérarchie de volumes englobants
  - La subdivision de l'espace hiérarchique
    - Octree, l'arbre k-D, l'arbre BSP

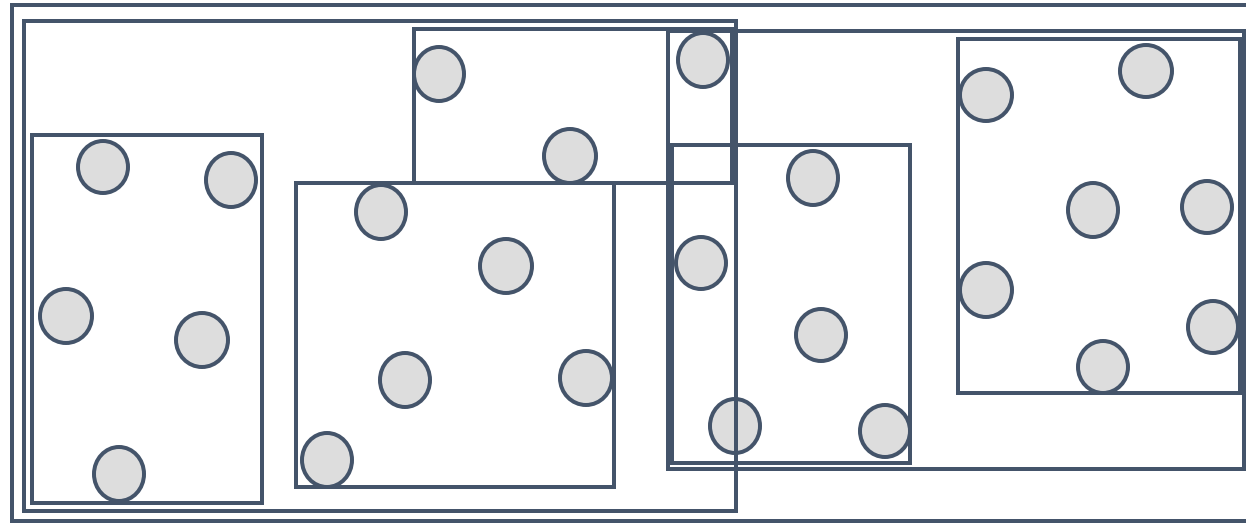
# VOLUMES ENGLOBANTES: L'IDÉE

- Éviter le calcul des intersections avec les objets complexes (e.g. un maillage de triangles)
- Faire un test conservateur rapide
  - Entourer l'objet d'une géométrie simple (e.g. une sphère ou une boîte)
  - Réduire les faux positifs: créer la géométrie aussi serrée que possible



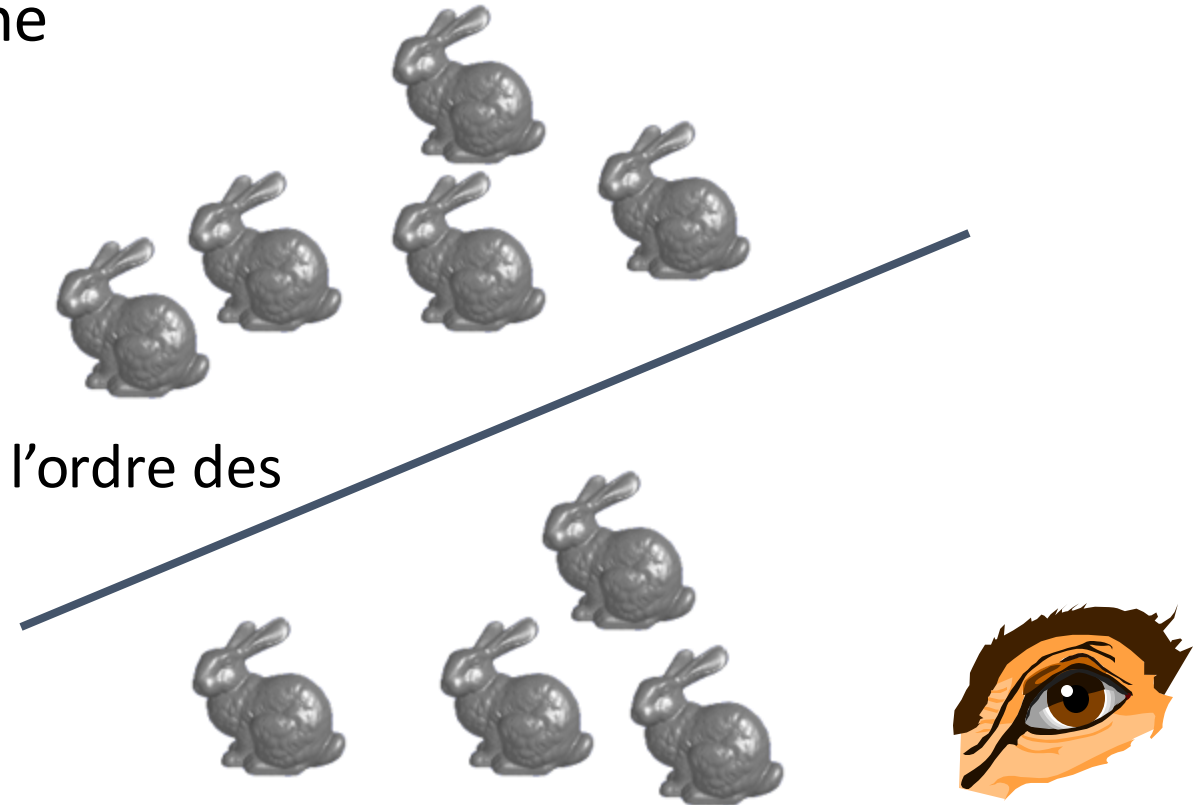
# HIÉRARCHIE DE VOLUMES

- Une extension de la dernière idée:
  - Utiliser les boîtes pour des groupes d'objets

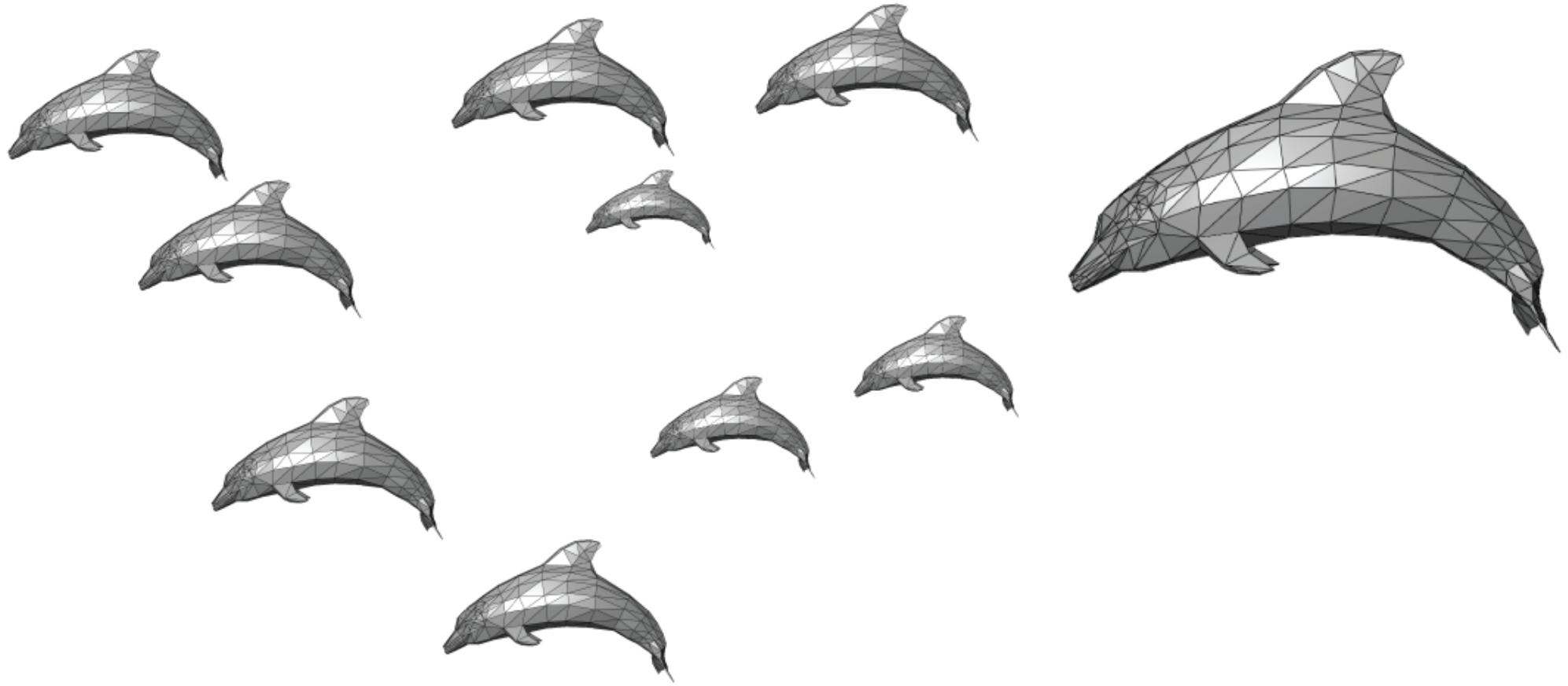


# LES ARBRES BSP : L'IDÉE

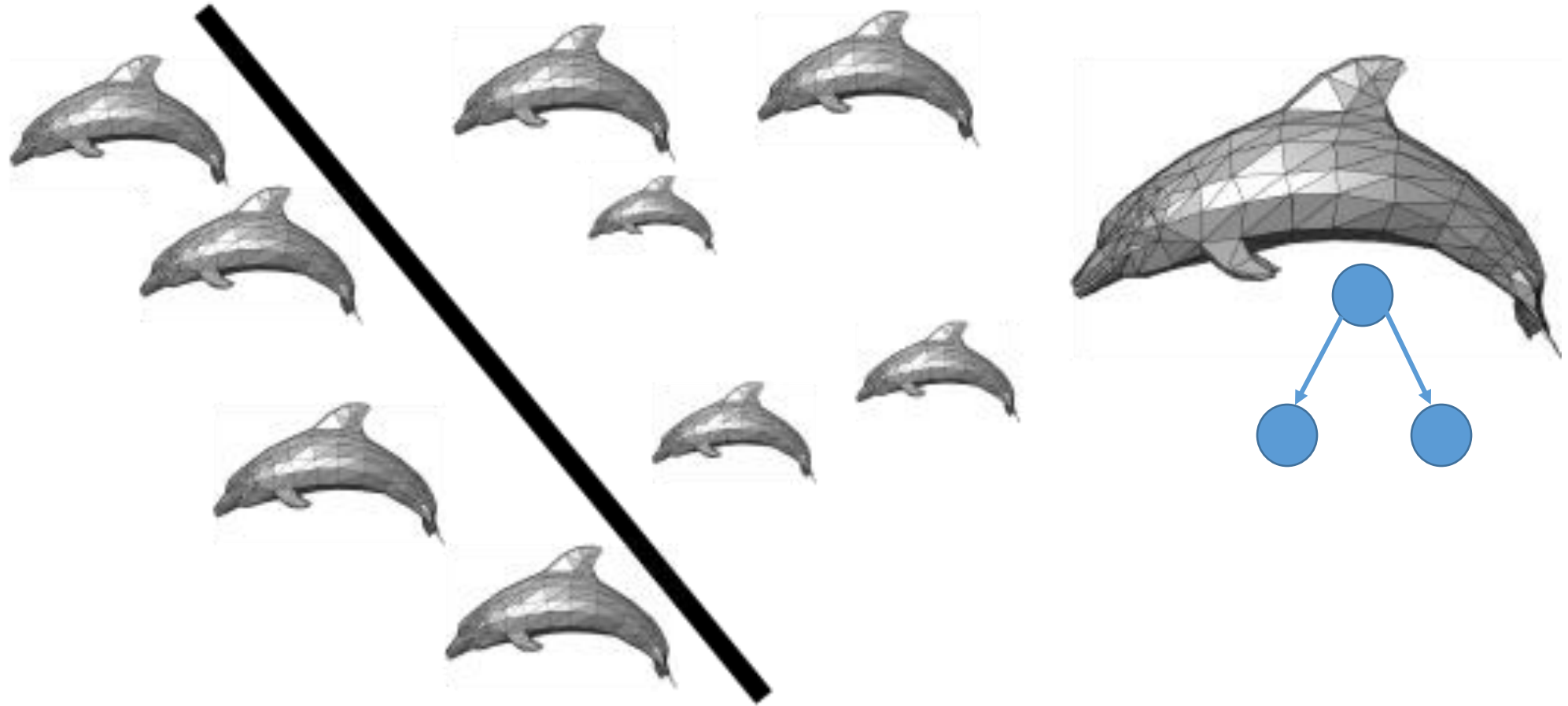
- Pour un plan, les objets du même côté que la caméra **NE PEUVENT PAS** être cachés par les objets d'un autre côté
- D'abord intersecter le côté le plus proche
- Si le rayon n'intersecte pas le plan?
  - **Il ne peut pas intersecter l'autre côté!**
- L'idée:
  - Diviser l'espace récursivement
  - Traverser l'arbre de plans pour calculer l'ordre des intersections/rendu
    - Tester la position de l'œil avec chaque plan



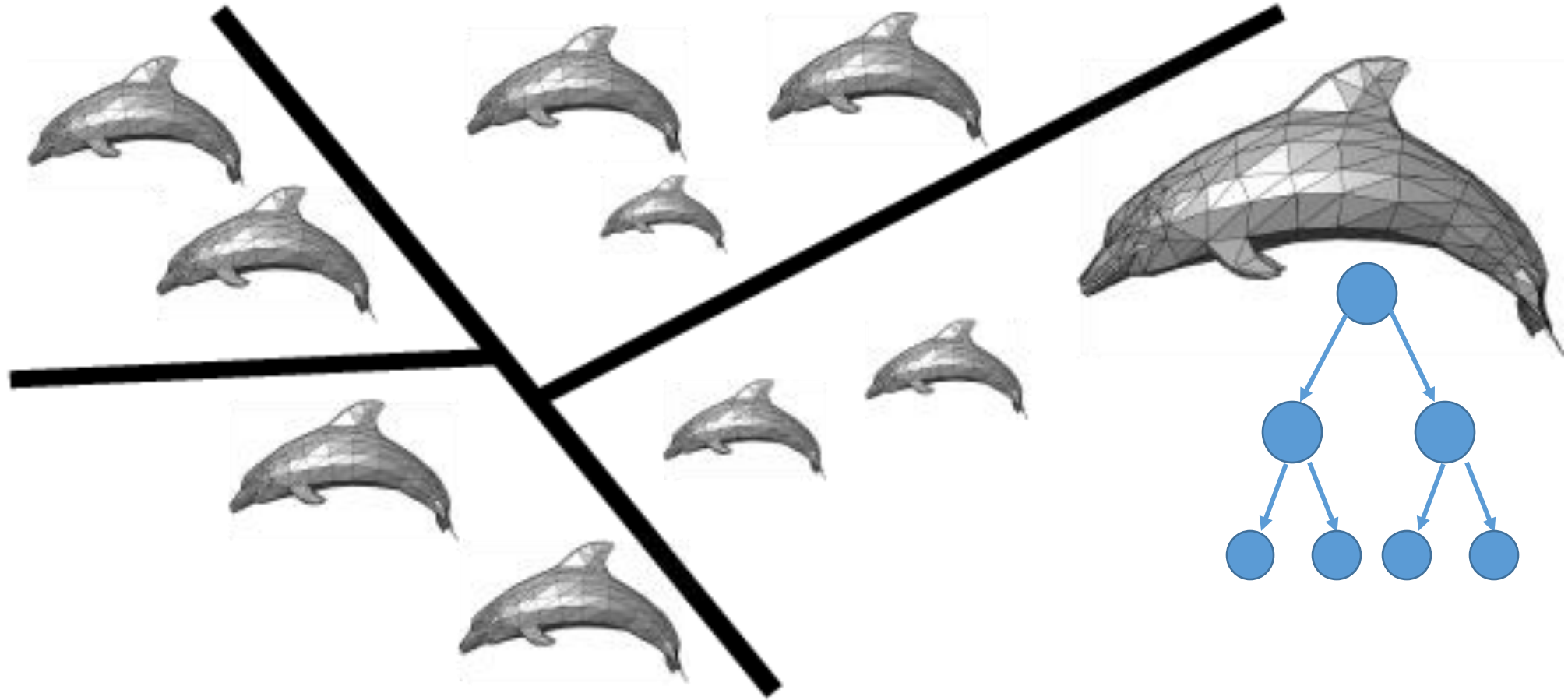
# LES ARBRES BSP: UNE CONSTRUCTION



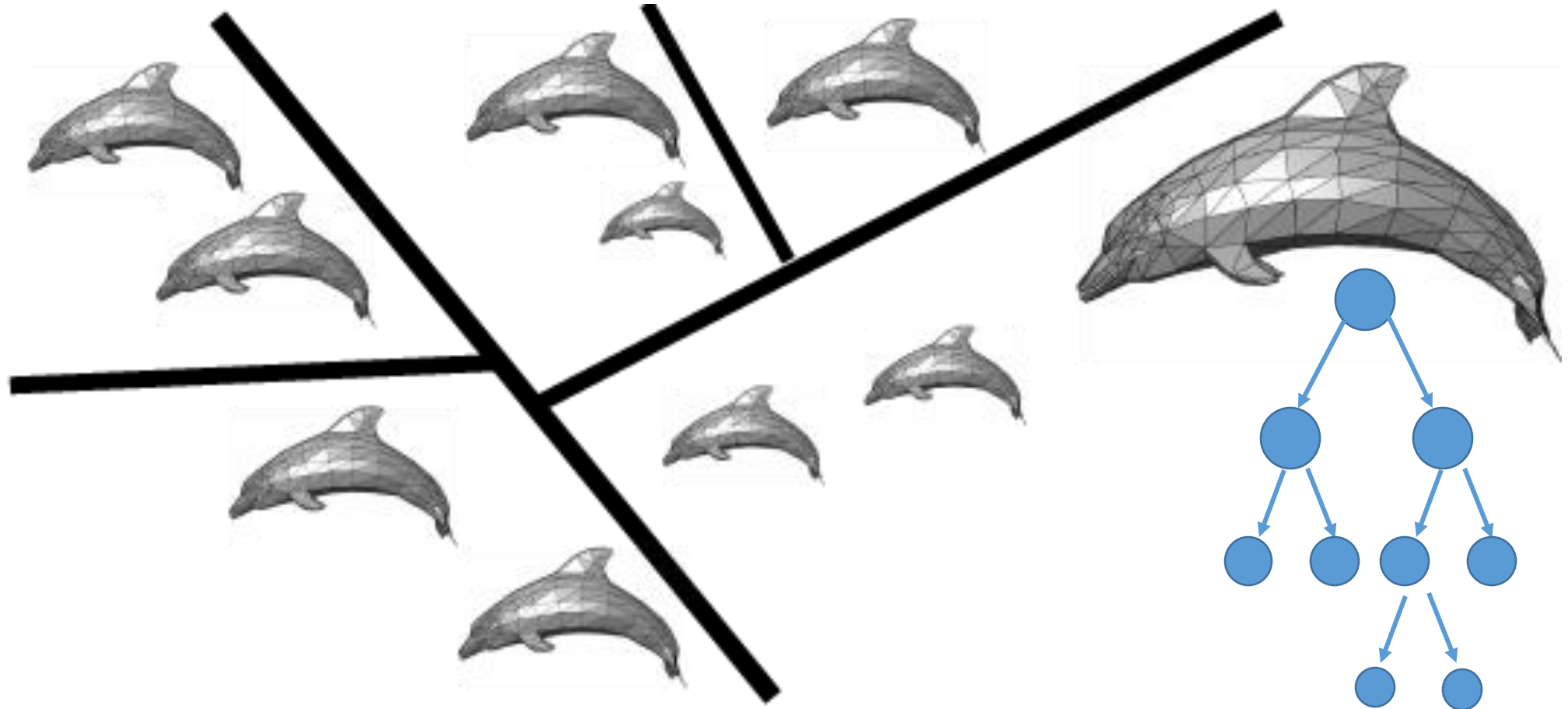
# LES ARBRES BSP: UNE CONSTRUCTION



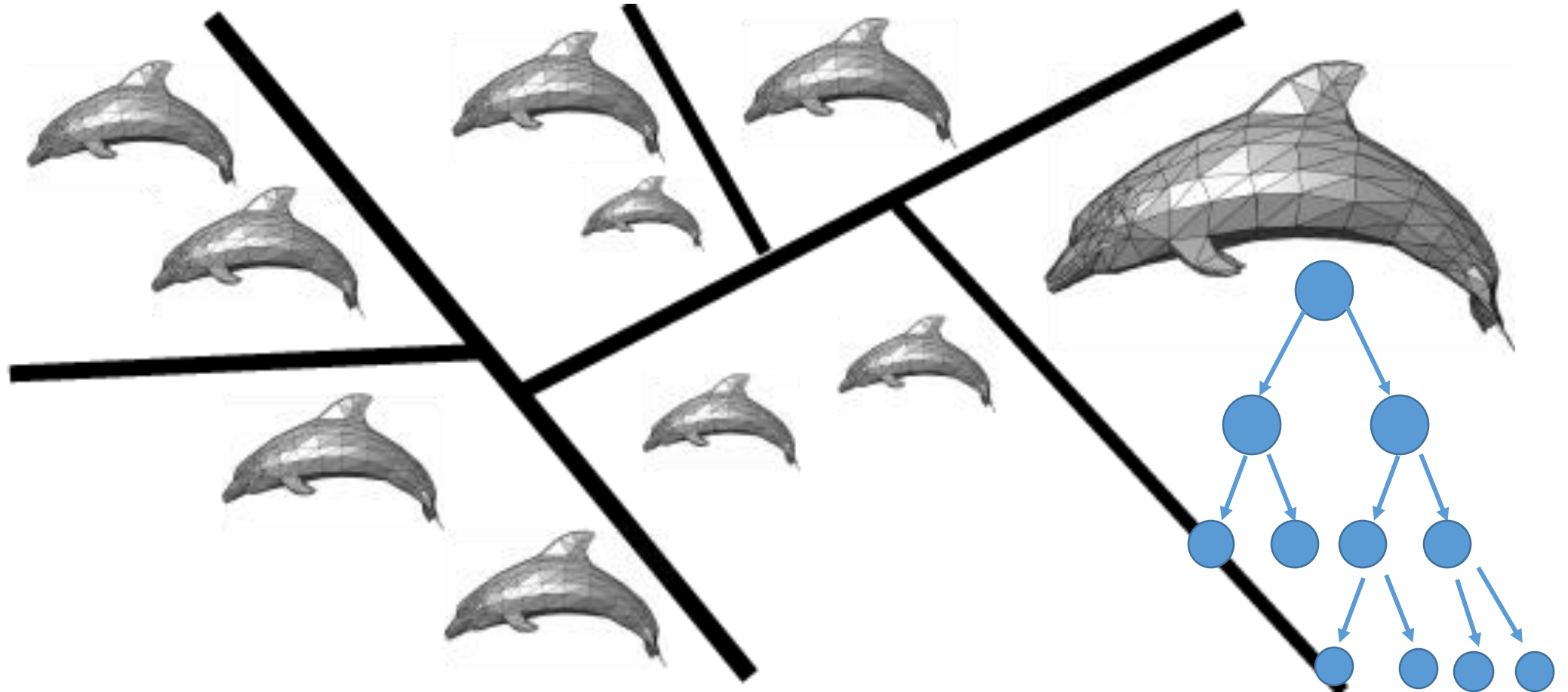
# LES ARBRES BSP: UNE CONSTRUCTION



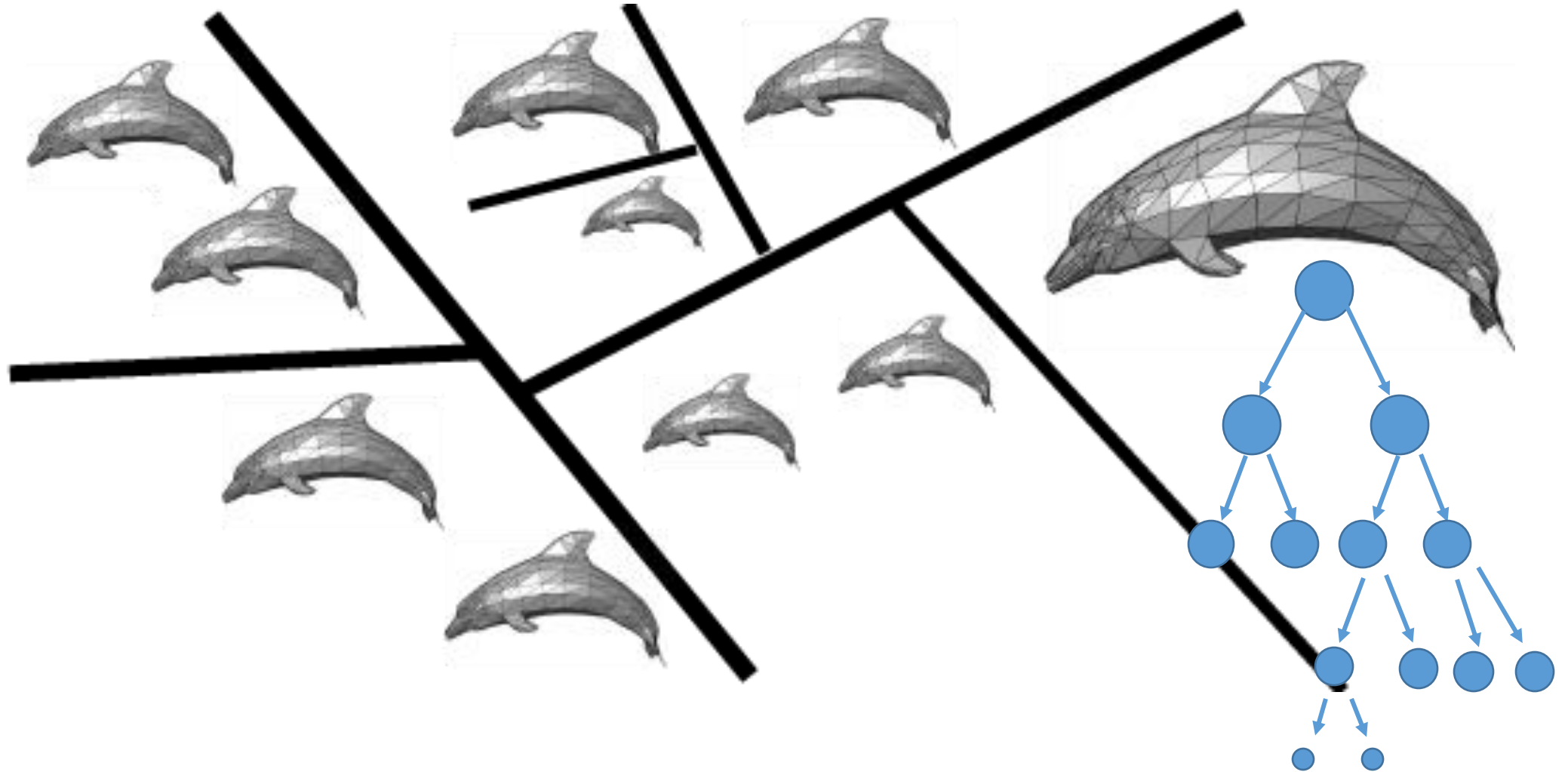
# LES ARBRES BSP: UNE CONSTRUCTION



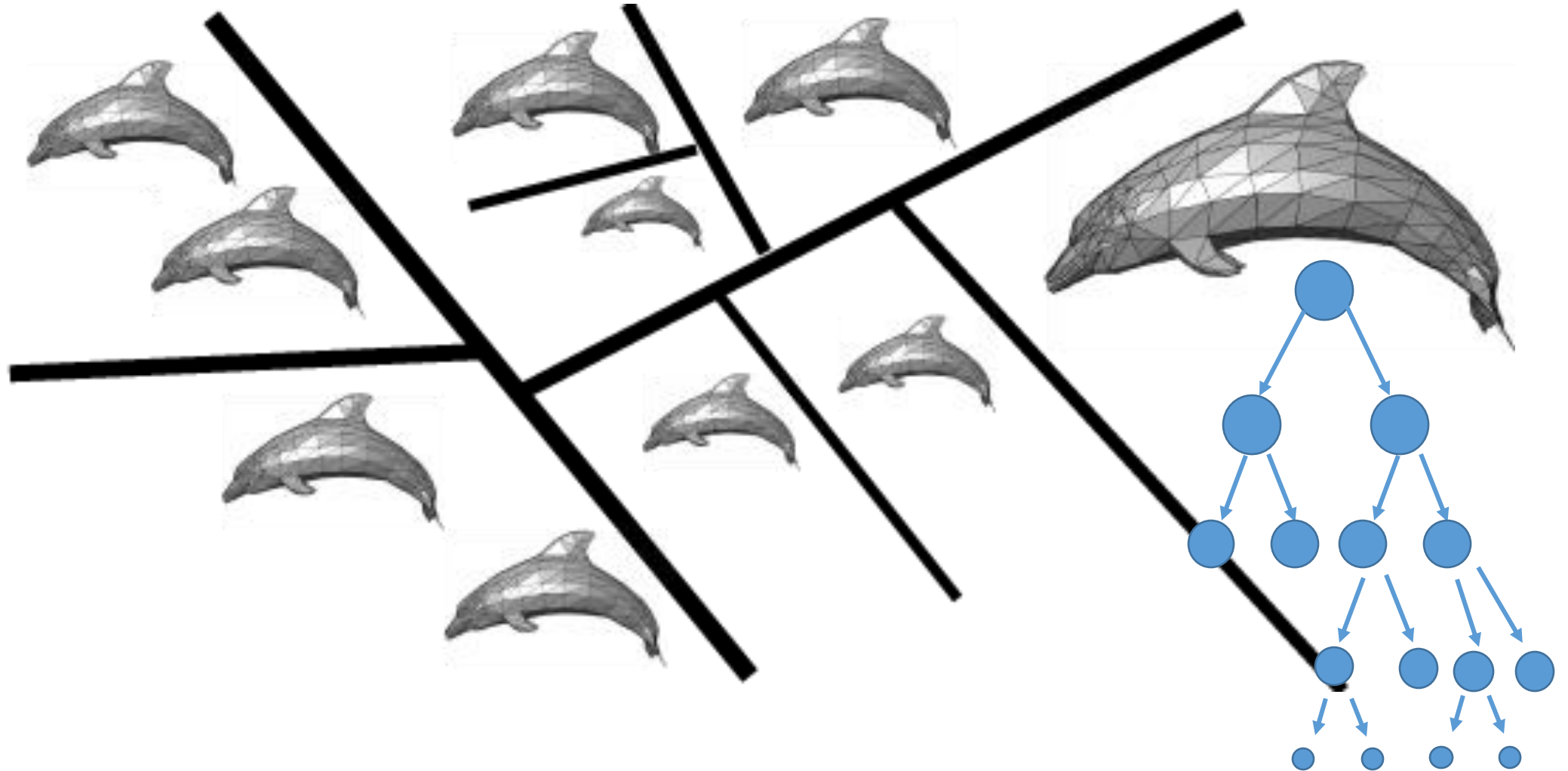
# LES ARBRES BSP: UNE CONSTRUCTION



# LES ARBRES BSP: UNE CONSTRUCTION

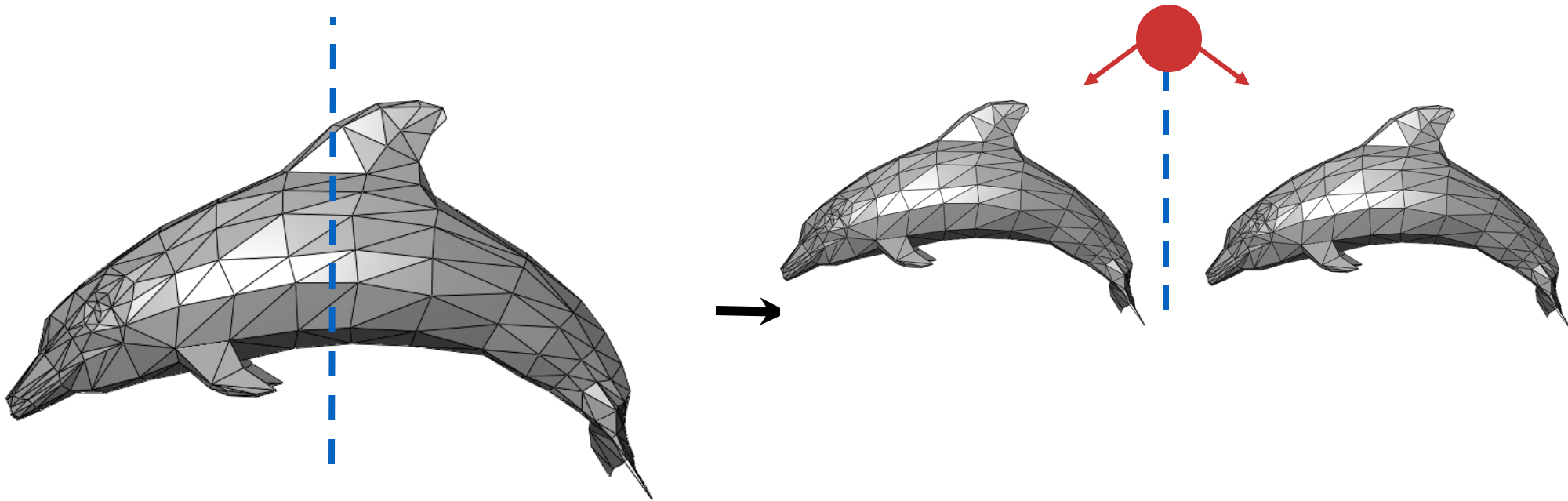


# LES ARBRES BSP: UNE CONSTRUCTION



# DIVISER LES OBJETS

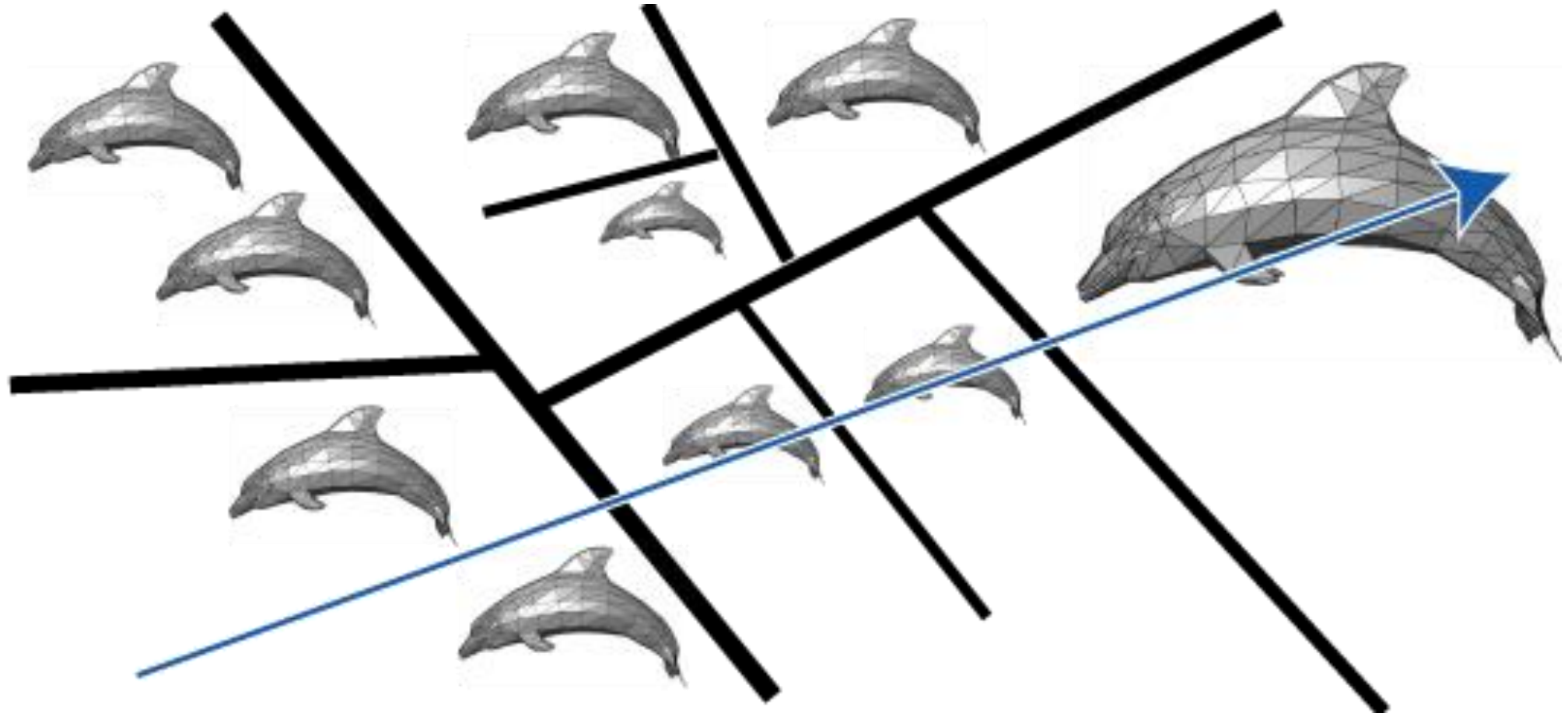
- Mais qu'est-ce qu'on fait si le plan intersecte un objet?
  - Dupliquer (considérer l'objet dans les deux parties)



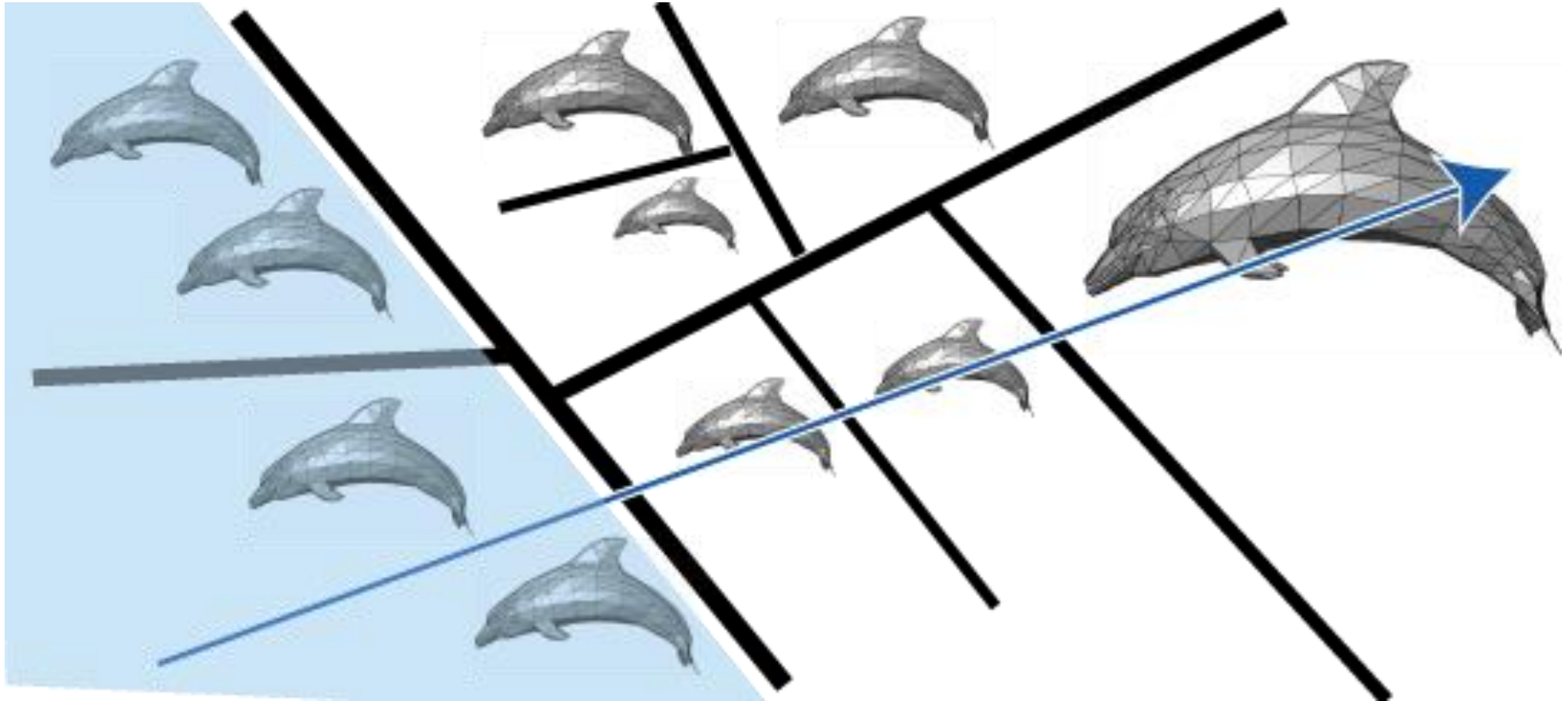
# TRAVERSER LES ARBRES BSP

- La création est indépendante de la position de l'œil
  - Une étape de prétraitement
- Une traversée utilise l'origine du rayon
  - Lors de l'exécution, on utilise l'arbre pour plusieurs rayons (=les origines différentes)

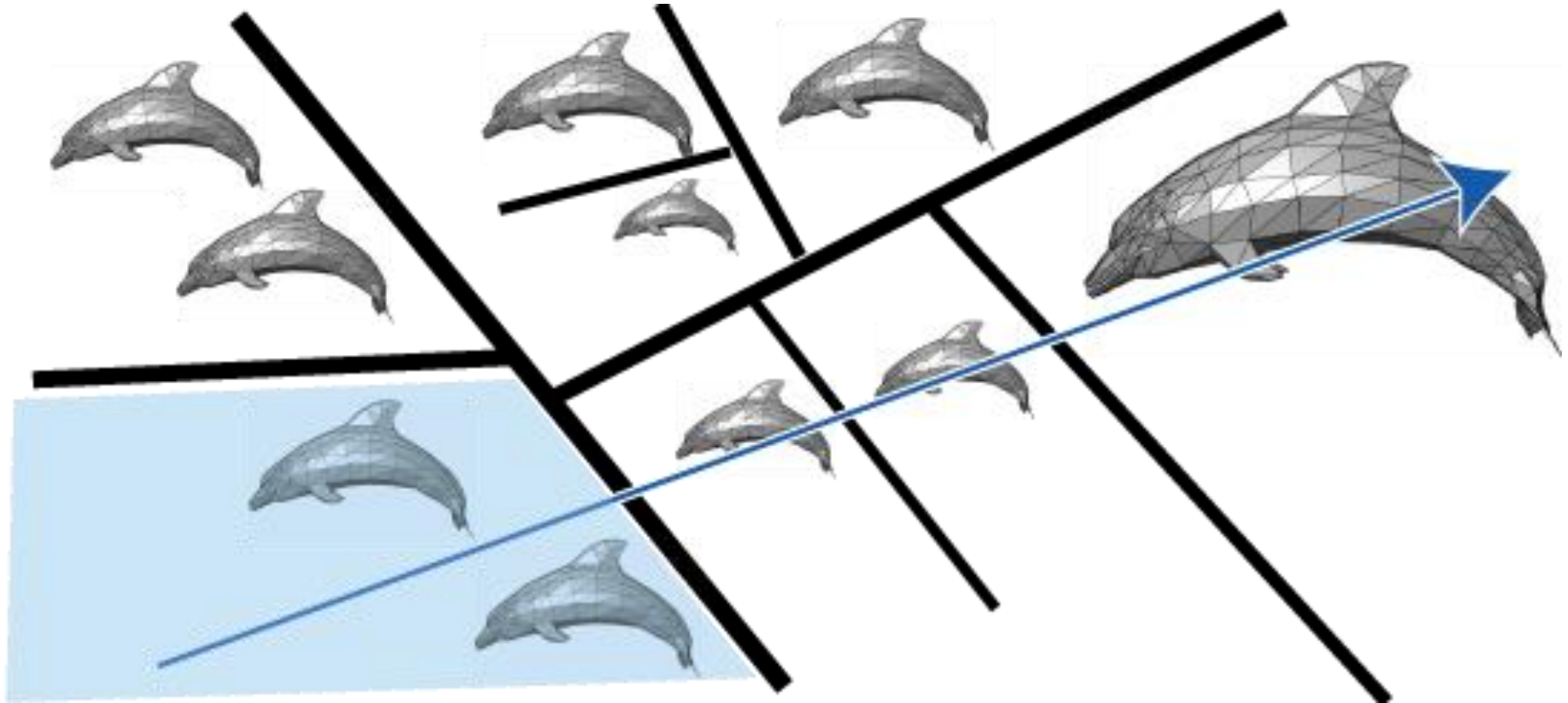
# LES ARBRES BSP : LA TRAVERSÉE



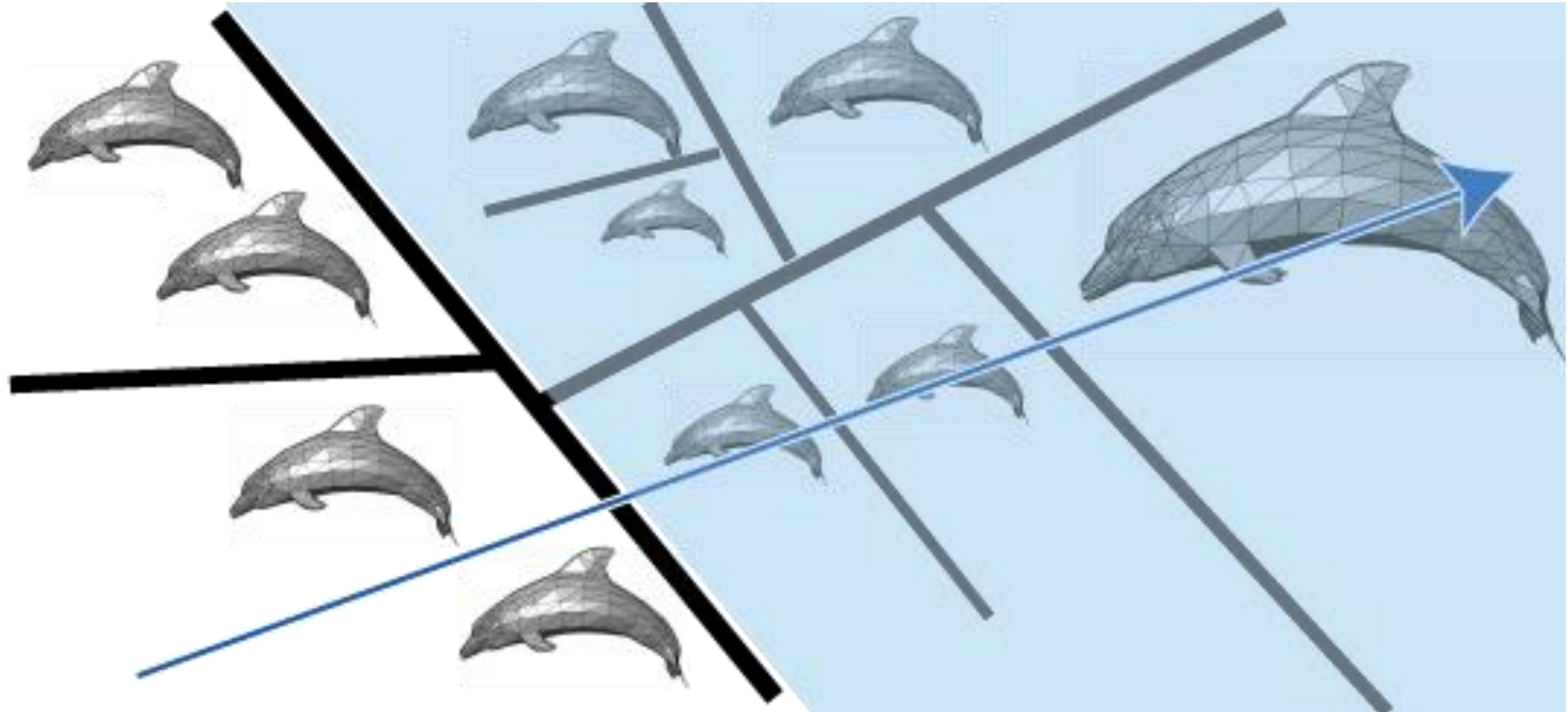
# LES ARBRES BSP : LA TRAVERSÉE



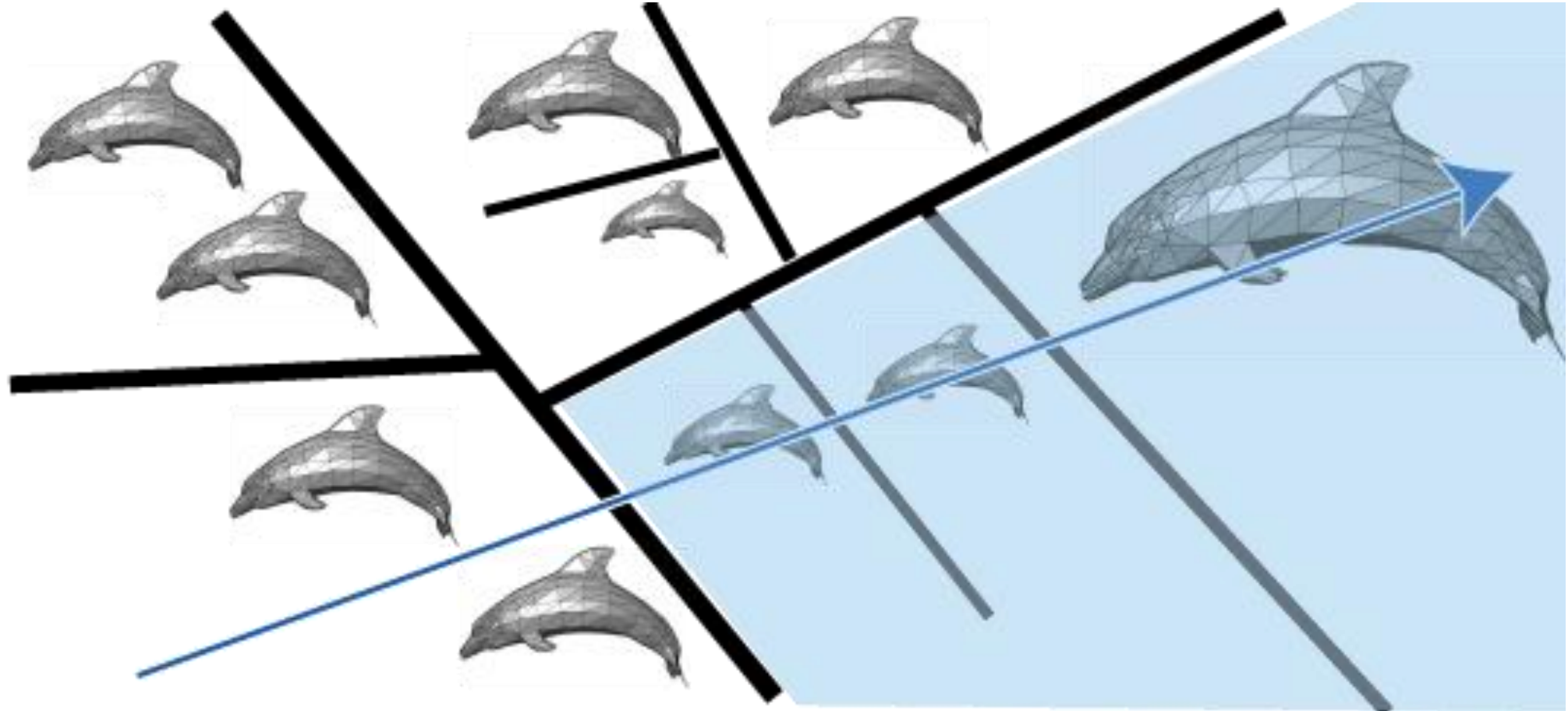
# LES ARBRES BSP : LA TRAVERSÉE



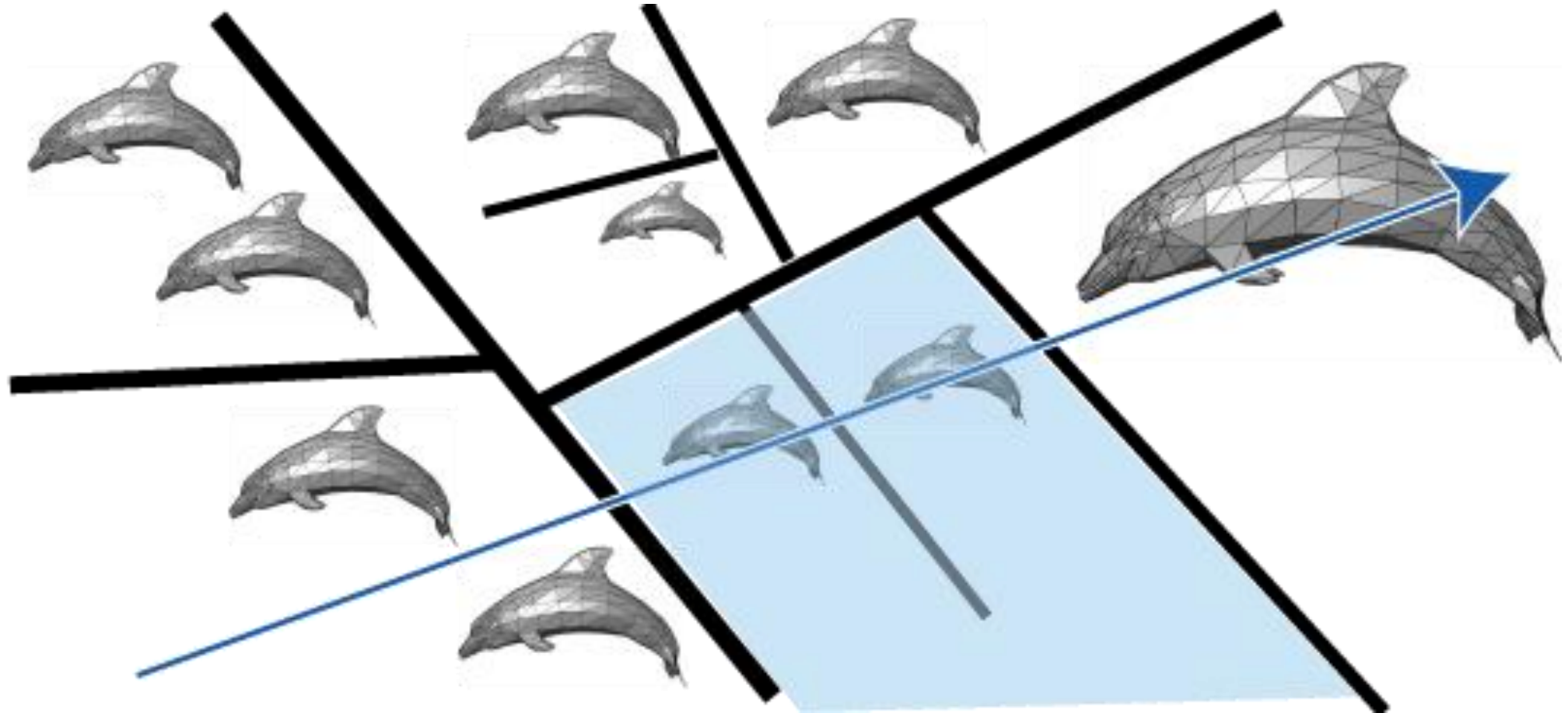
# LES ARBRES BSP : LA TRAVERSÉE



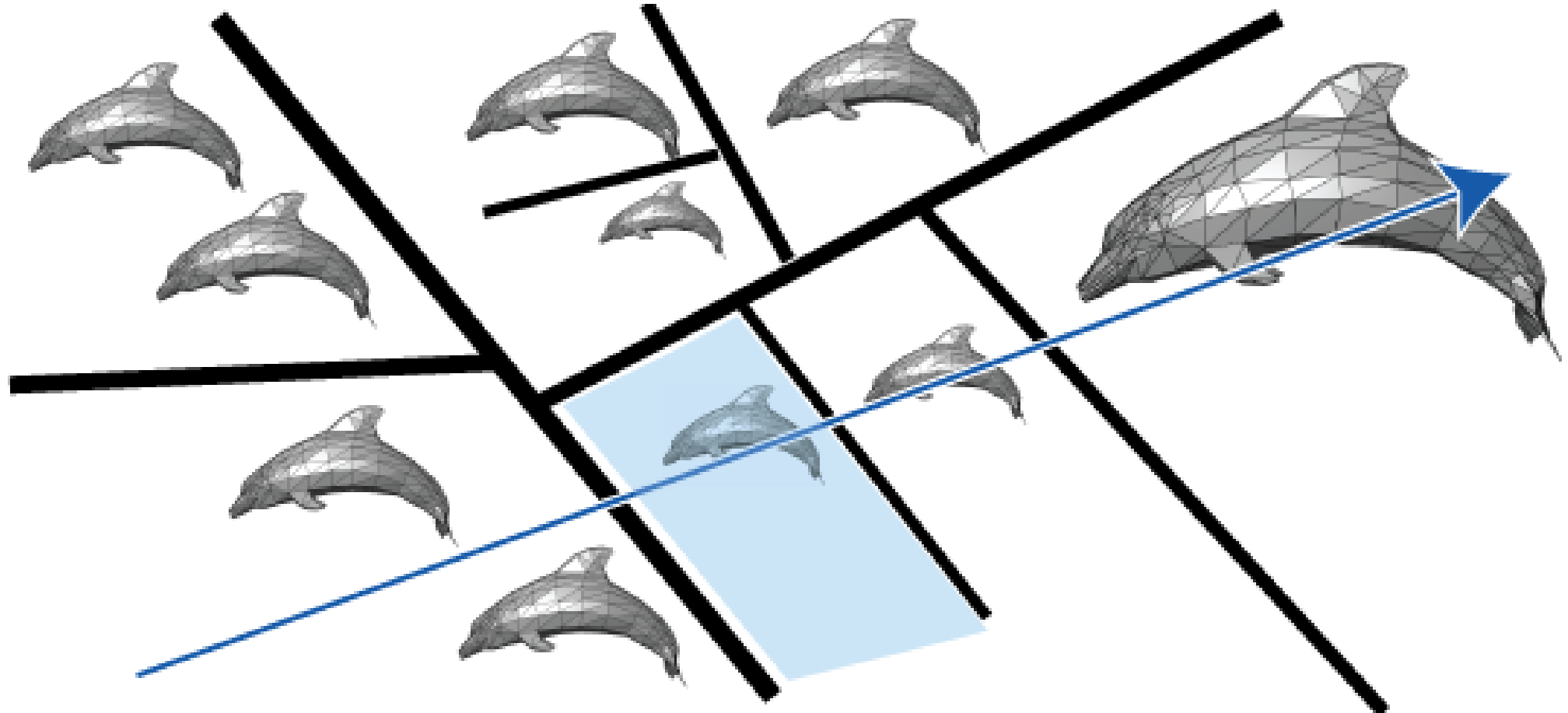
# LES ARBRES BSP : LA TRAVERSÉE



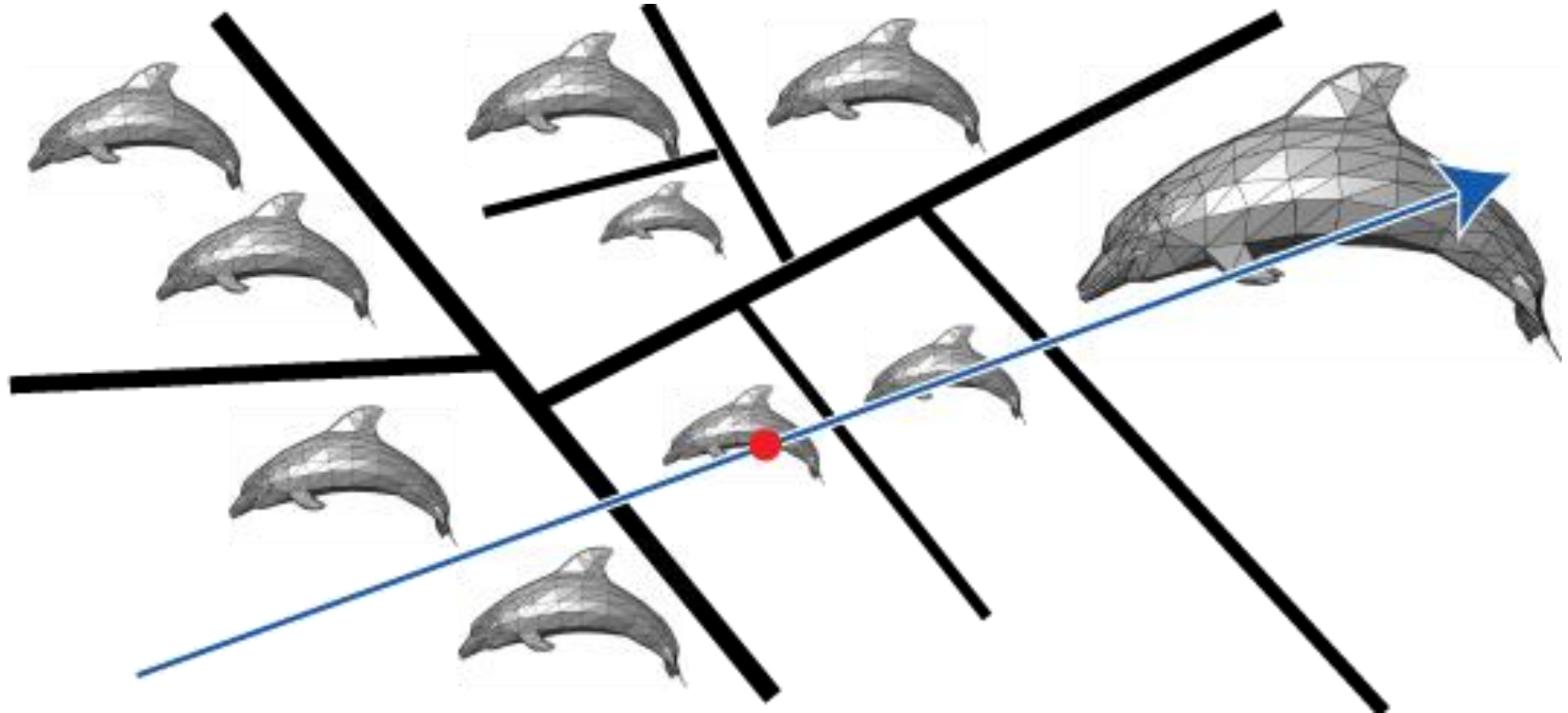
# LES ARBRES BSP : LA TRAVERSÉE



# LES ARBRES BSP : LA TRAVERSÉE



# LES ARBRES BSP : LA TRAVERSÉE



# LES ARBRES BSP : LA TRAVERSÉE

- Chaque plan divise le monde en proche et loin
  - Pour un rayon, décider quel côté est proche
    - Proche = le même côté que l'œil
    - L'œil définit l'ordre de la traversée
- L'algorithme est récursif
  - Intersecter avec le côté proche
  - Si il n'y a pas d'intersection
    - Intersecter avec le côté loin

# LES ARBRES BSP : LA TRAVERSÉE

Soit  $v$  un nœud,  $r$  est un rayon

**Intersect**( $v, r$ )

**if**  $v$  est une feuille

    intersecter  $r$  avec chaque objet dans  $v$

    retourner le plus proche ou **nil** si pas trouvé

$near$  = le nœud enfant de demi-espace qui contient l'origine de  $r$

$far$  = l'autre enfant

$hit$  = Intersect( $near, r$ )

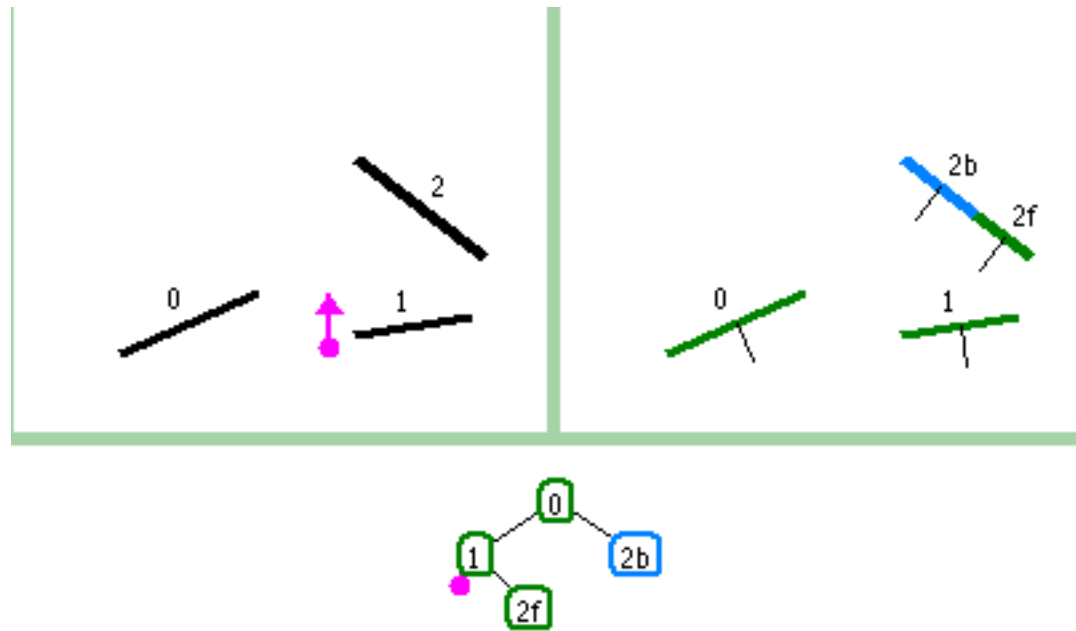
**if**  $hit$  est **nil** **AND**  $r$  intersecte le plan défini par  $v$

$hit = \text{Intersect}(far, r)$

**return**  $hit$

# BSP DÉMO

- Démonstration
  - <http://symbolcraft.com/graphics/bsp>



# UN RÉSUMÉ: LES ARBRES BSP

- Avantages:

- Un schéma élégant et simple
- Les intersections rapides
- La version correcte de l'algorithme de peintre

- Inconvénients:

- L'arbre est lent à construire:  $O(n \log n)$
- La division d'objets augmente le nombre de polygones:  $O(n^2)$  pire cas
- => L'algorithme est limité aux cas statiques

$n$  est le nombre  
d'objects

# LES STRUCTURE DE DONNÉES POUR LA SUBDIVISION DE L'ESPACE

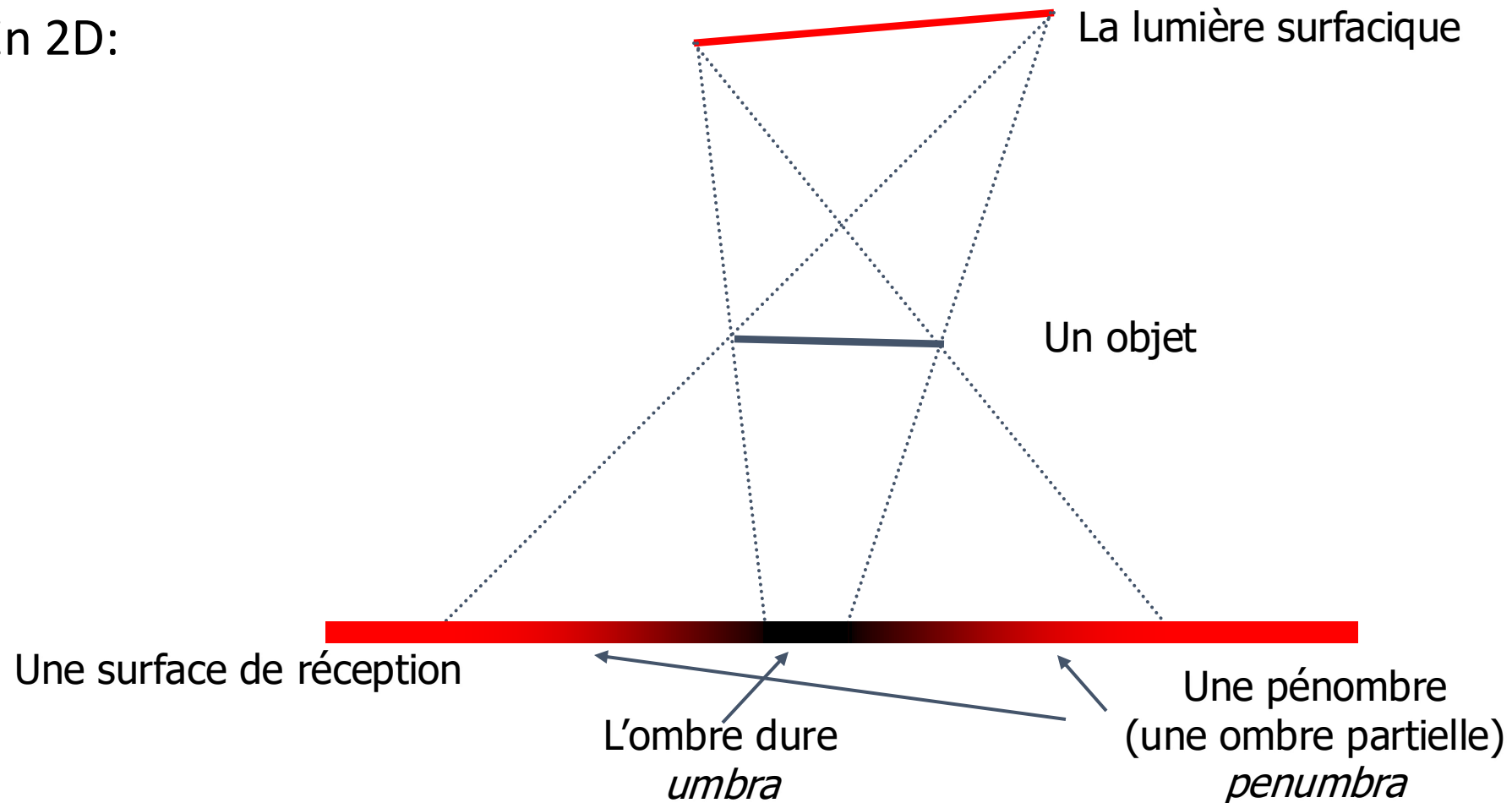
- Volumes englobants:
  - Trouver un objet assez simple pour entourer les objets complexes
    - Les boîtes, les sphères
  - Les combiner hiérarchiquement
- La structure de données pour la subdivision de l'espace:
  - Diviser l'espace en cellules
    - Les grilles, les *octrees*, les arbres k-d, (les arbres BSP)
  - Simplifier et accélérer la traversée
  - La performance dépend moins de l'ordre dans lequel les objets sont insérés

# LES OMBRES DOUCES: LES LUMIÈRES SURFACIQUES

- Jusqu'ici:
  - Toutes les lumières étaient ponctuelles ou directionnelles
    - Pour le pipeline OpenGL et lancer de rayons
  - Donc, à chaque point, on avait besoin de calculer l'éclairage pour **UNE** direction par lumière
- En réalité:
  - Toutes les lumières ont une aire finie
  - Maintenant on a besoin **d'intégrer** sur toutes les directions vers la lumière

# LES LUMIÈRES SURFACIQUES

- Les sources de lumière surfaciques produisent des ombres douces:
  - En 2D:

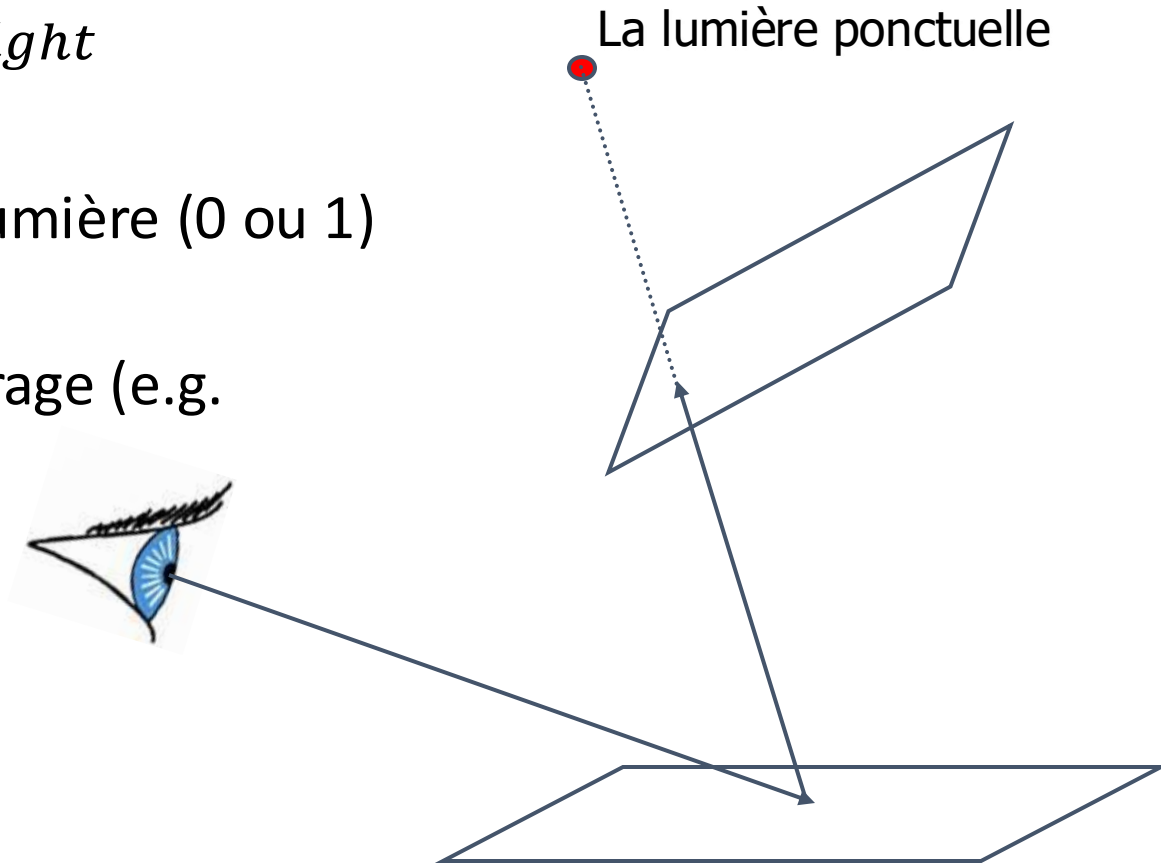


# LES LUMIÈRES SURFACIQUES

- Les lumières ponctuelles:
  - ont une seule direction (par point):

$$I_{reflected} = \rho V I_{light}$$

- $V$  est la visibilité de la lumière (0 ou 1)
- $\rho$  est un modèle d'éclairage (e.g. Lambert/Phong)

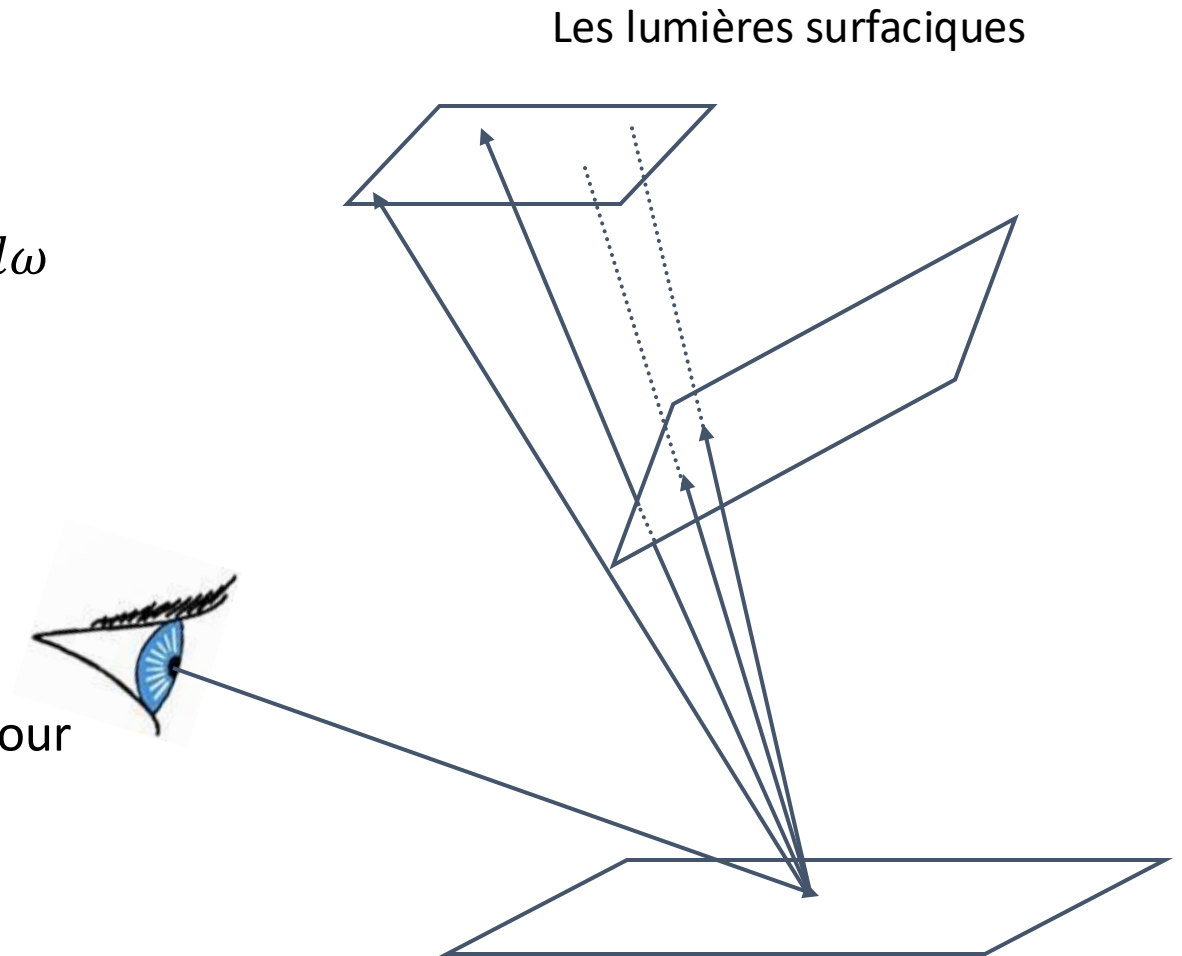


# LES LUMIÈRES SURFACIQUES

- Les lumières surfaciques:
  - Nombre infini de rayons lumineux
  - Il faut intégrer:

$$I_{reflected} = \int_{\substack{\text{light} \\ \text{directions}}} \rho(\omega, \mathbf{n}) V(\omega) I_{light}(\omega) d\omega$$

- La visibilité et l'intensité peuvent changer pour différents points



# LES LUMIÈRES SURFACIQUES

Récrire l'intégrale, au lieu d'intégrer sur les **directions**

$$I_{reflected} = \int_{\substack{\text{light} \\ \text{directions}}} \rho(\omega, \mathbf{n}) V(\omega) I_{light}(\omega) d\omega$$

intégrer sur les **points** de la lumière

$$I_{reflected}(q) = \int_{s,t} \rho(\mathbf{p} - q, \mathbf{n}) V(\mathbf{p} - q) I_{light}(\mathbf{p}) ds dt$$

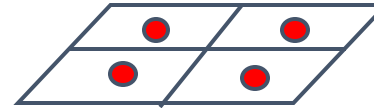
- $q$  est un point sur l'objet
- $\mathbf{p} = F(s, t)$  est un point sur la lumière

# L'INTÉGRATION

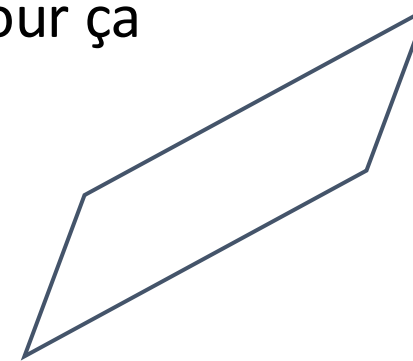
- Un problème:
  - Sauf un cas de base, **on ne peut pas le résoudre analytiquement!**
    - En partie, en raison de la visibilité
- Puis:
  - Utiliser l'intégration numérique

# L'INTÉGRATION NUMÉRIQUE

La lumière  
surfactive



- La grille des lumières ponctuelles
  - Il faut utiliser beaucoup de points de lumière pour ça



# LA SOLUTION: MONTE-CARLO

- La prochaine fois!