L3-Synthèse

Lancer de rayons et rendu

J.C. Iehl

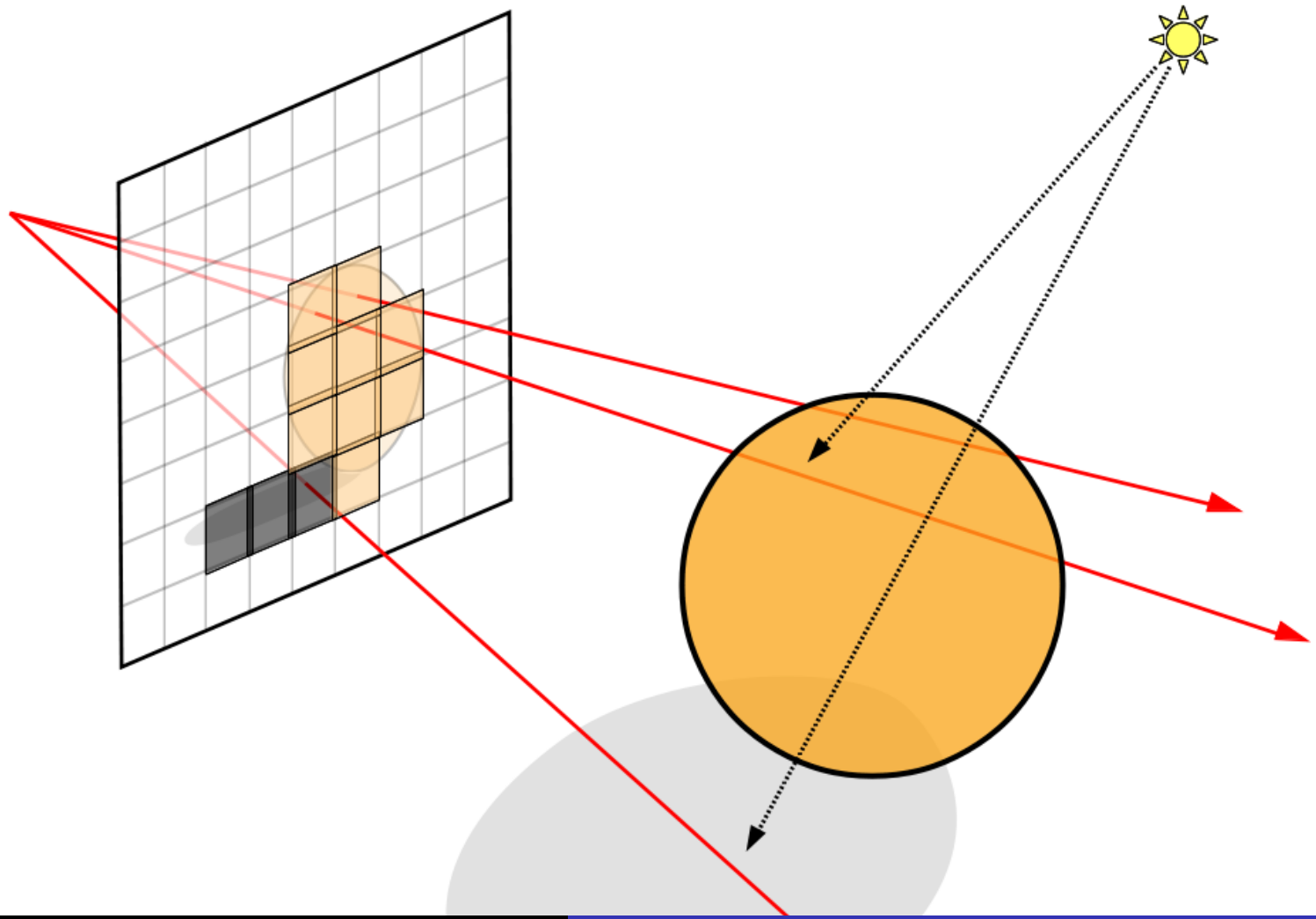
January 15, 2024

résumé des épisodes précédents

calculer une image :

- trouver quel objet est visible pour chaque pixel,
- trouver comment l'objet est éclairé,
- calculer sa couleur...

calculer une image :

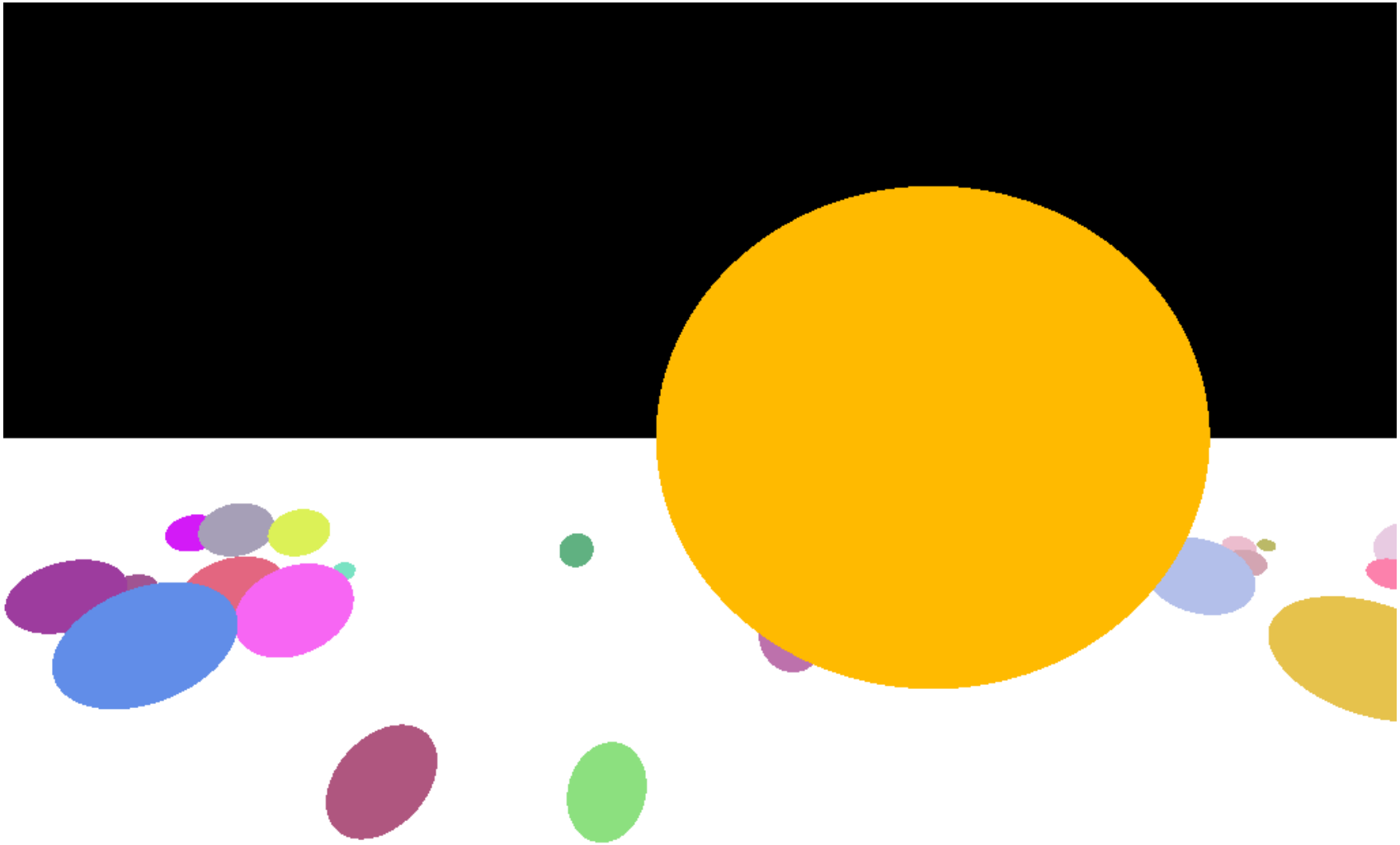


résumé des épisodes précédents

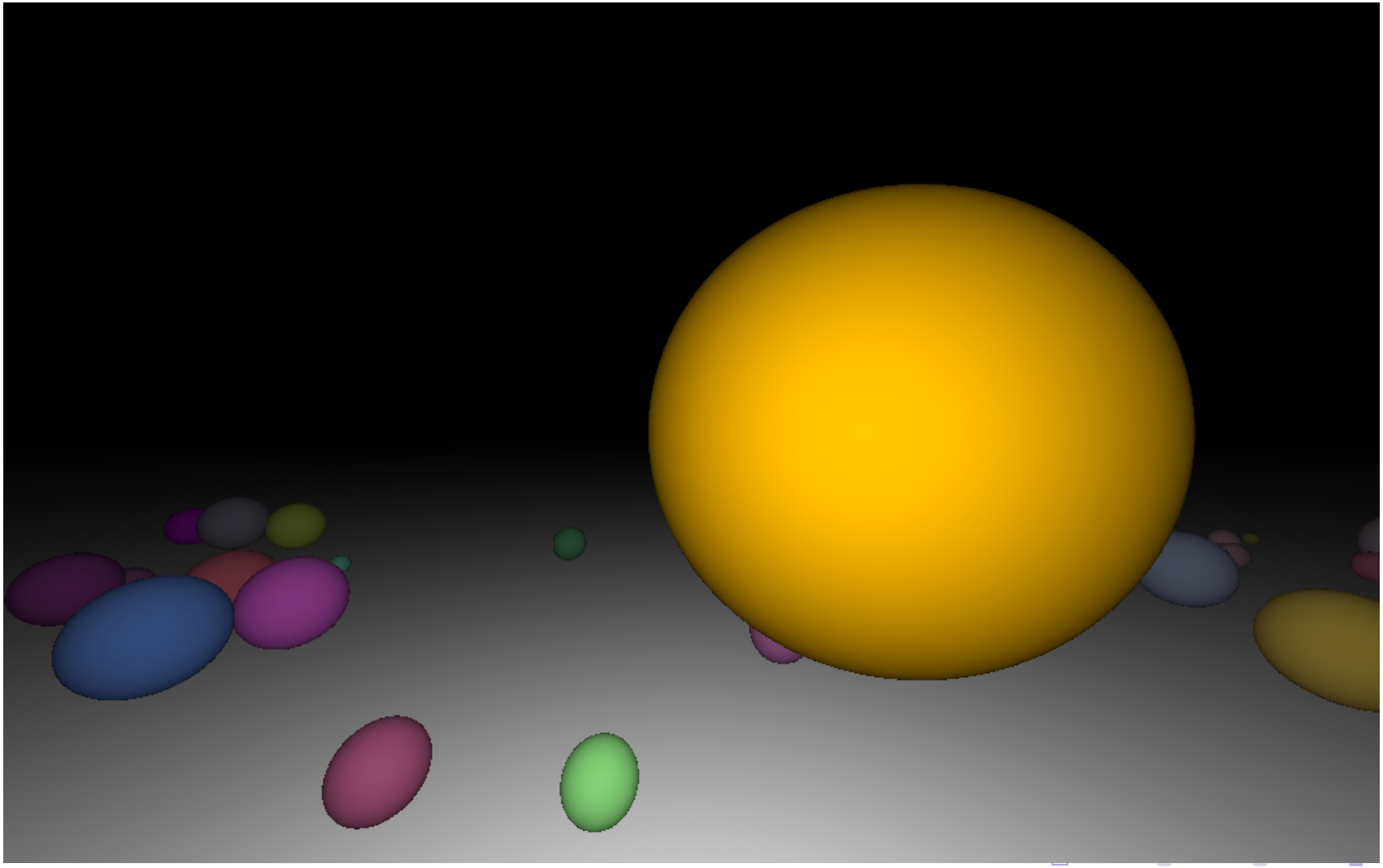
trouver l'objet visible :

- camera,
- plan image,
- rayon,
- intersections,
- garder la plus proche / l'objet visible

résumé des épisodes précédents



couleur de l'objet ?

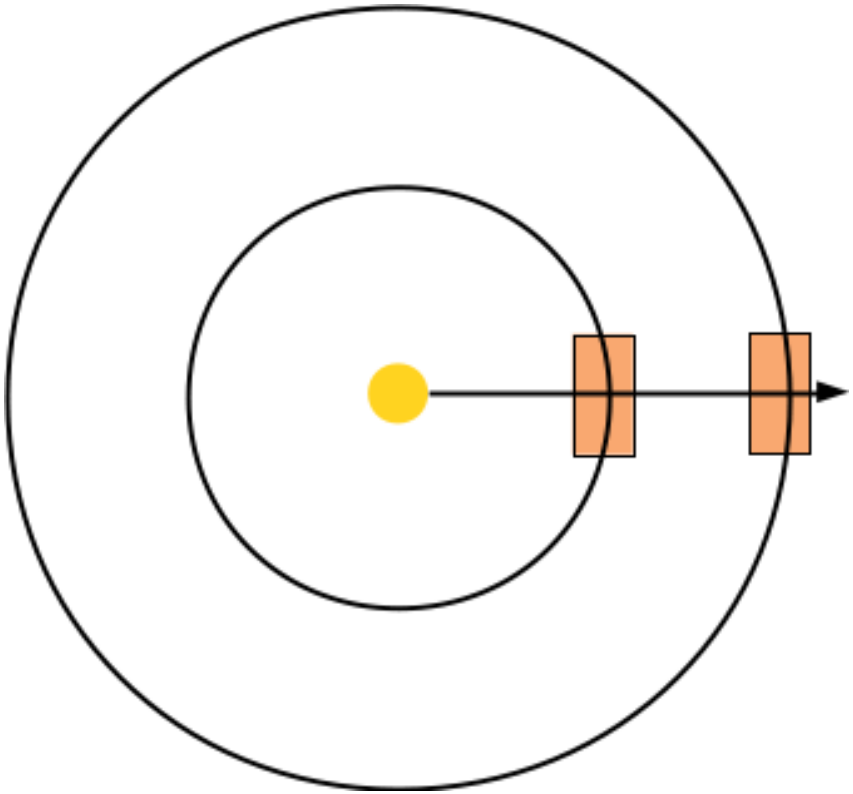


et pendant ce temps... les physiciens...

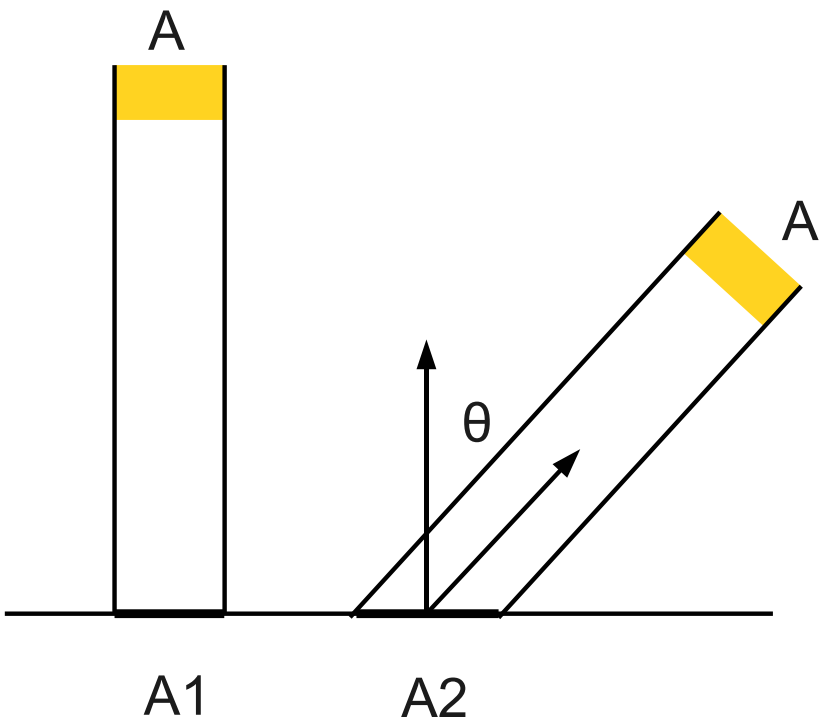
Lambert en 1760 :

- une surface diffuse réfléchit la même quantité de lumière dans toutes les directions,
- une surface orientée vers la lumière reçoit plus de lumière,
- une surface proche de la lumière reçoit beaucoup plus de lumière...

proche de la lumière...



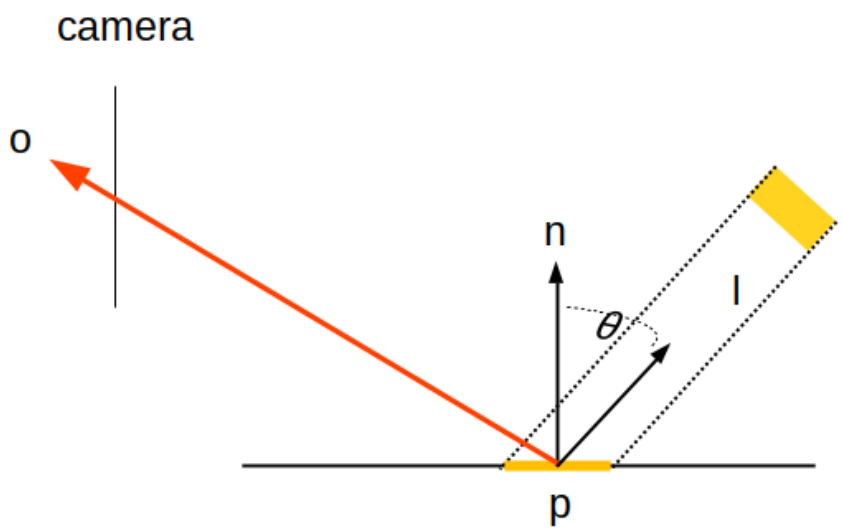
orienté vers la lumière...



lumière incidente

lumière incidente :

- $L_i(\vec{l}) = L_e(\vec{l}) \times \cos\theta$



lumière réfléchie

lumière réfléchie :

- $L_r(\vec{o}) = f_r(\vec{l}, \vec{o}) \times L_i(\vec{l}) = f_r(\vec{l}, \vec{o}) \times L_e(\vec{l}) \times \cos\theta$

$f_r(\vec{l}, \vec{o})$ fonction de réflectance : comment la matière réfléchit la lumière vers $\vec{o}$.

lumière réfléchie

lumière réfléchie :

- pour une matière diffuse, f_r est une constante, ou "la couleur" de l'objet...
- $L_r = f_r \times L_i = color \times L_e \times \cos\theta$

rappel : Lambert, une matière diffuse réfléchit la même quantité de lumière dans toutes les directions...

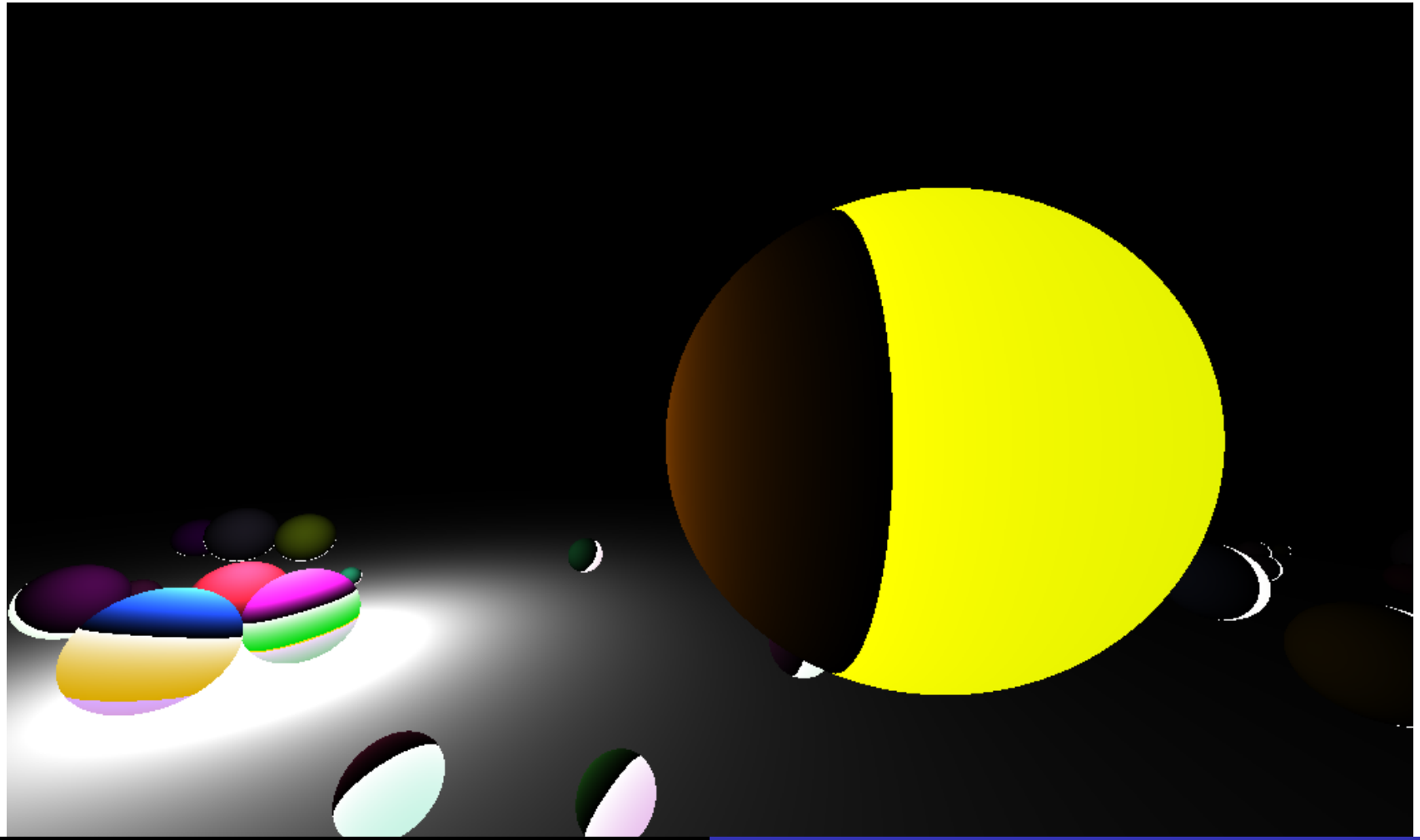
comment ça se code ?

une relation utile : $\cos\theta = \frac{\vec{n}\cdot\vec{l}}{||\vec{n}||||\vec{l}||}$, ou $\cos\theta = \frac{\vec{n}}{||\vec{n}||} \cdot \frac{\vec{l}}{||\vec{l}||}$

```
#include "vec.h"
#include "color.h"


Vector l= { ... };              // direction source de lumière
Color emission= { ... };        // lumière emise par la source

// point d'intersection et sa normale
Point p= { ... };
Vector n= { ... };
Color color= { ... };           // couleur de la matiere en p

float cos_theta= dot(normalize(n), normalize(l));

Color pixel= color * emission * cos_theta;
```

lumière incidente

euh ?

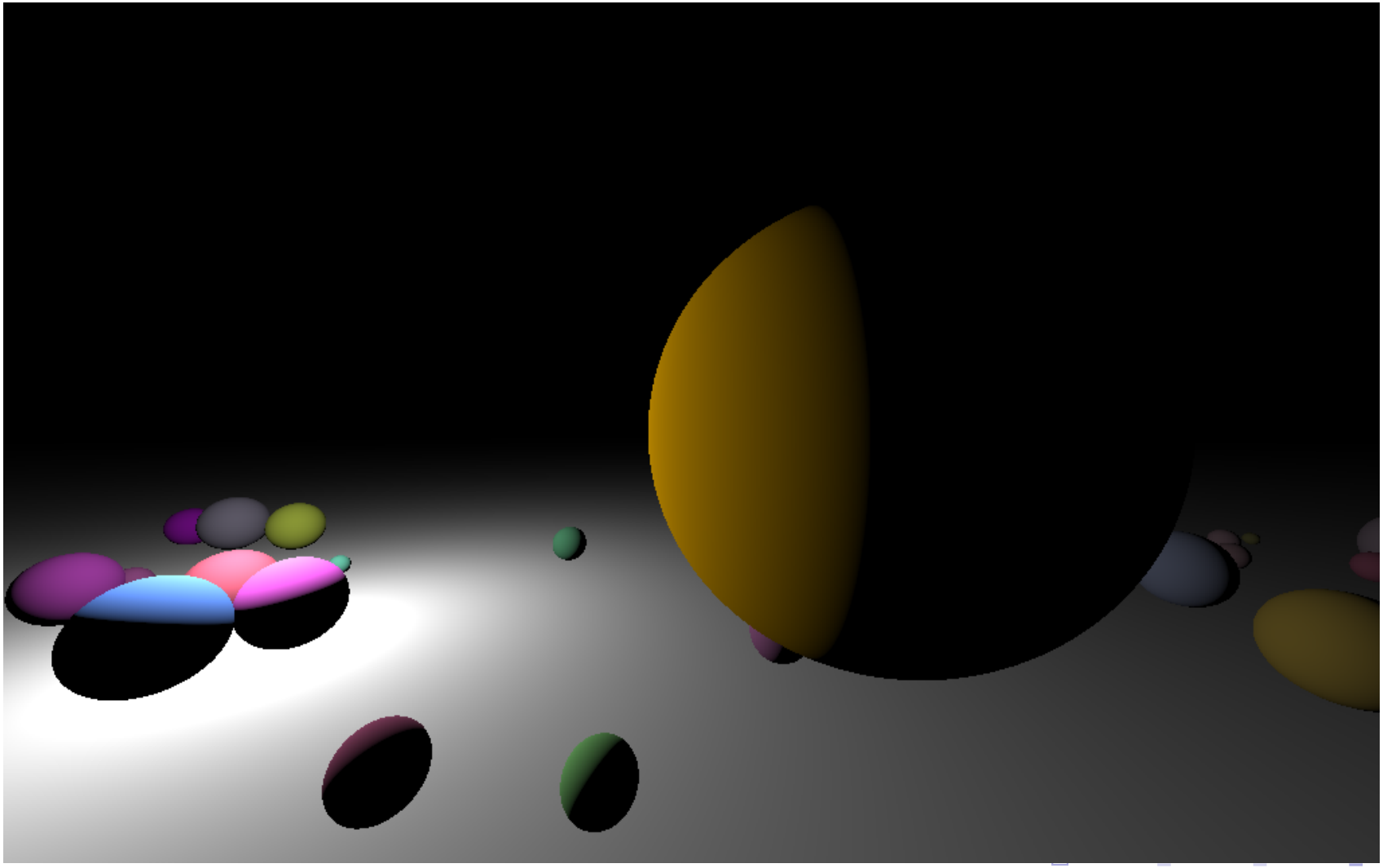


ah !

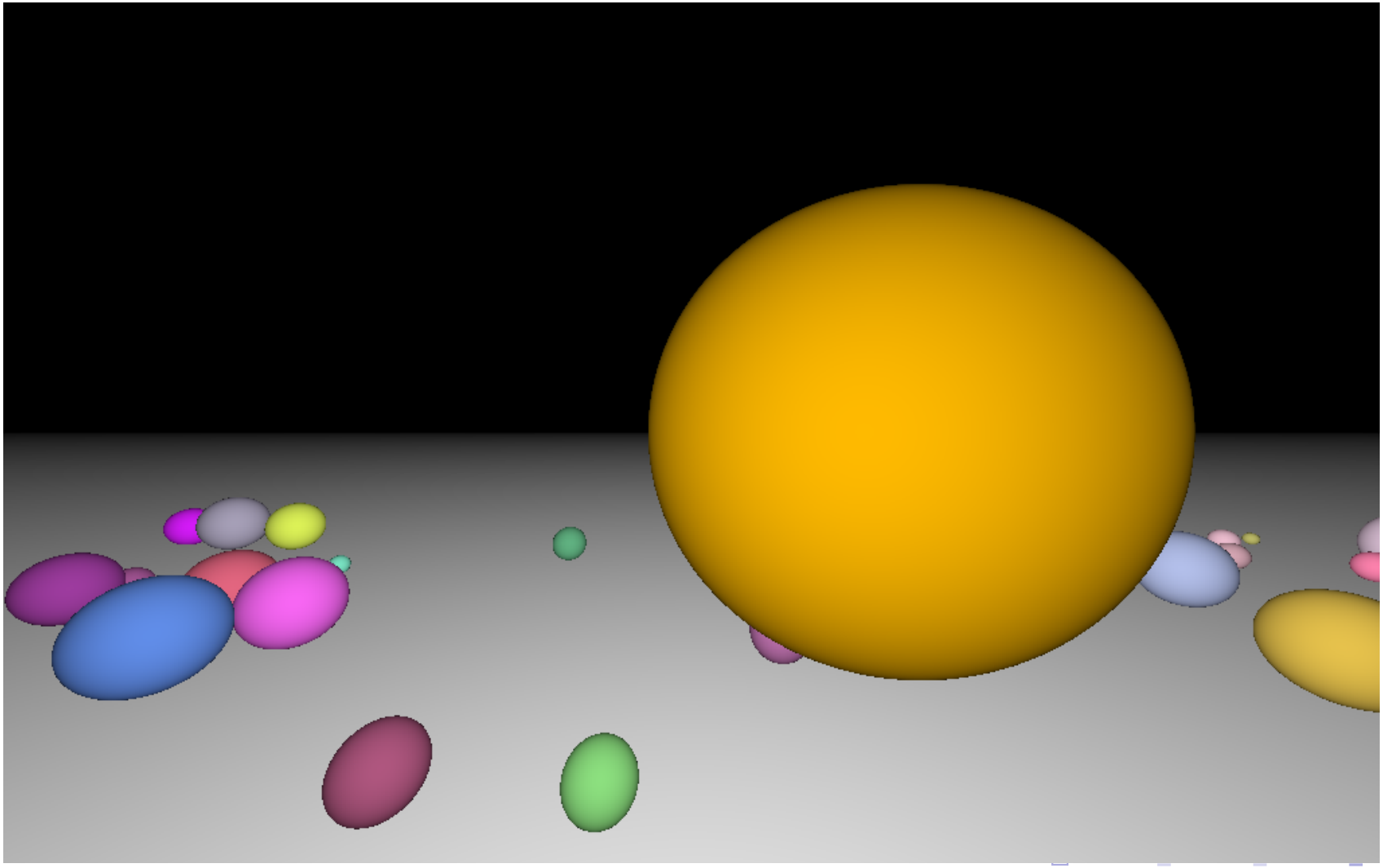
mais attention, $\cos\theta$ ne doit pas etre < 0

```
#include <algorithm>
#include "vec.h"
#include "color.h"


Vector l= { ... };              // direction source de lumière
Color emission= { ... };        // lumière emise par la source

// point d'intersection et sa normale
Point p= { ... };
Vector n= { ... };
Color color= { ... };           // couleur de la matiere en p

float cos_theta= std::max(float(0), dot(normalize(n), normalize(l)));
// ou if(cos_theta < 0) cos_theta= 0;

Color pixel= color * emission * cos_theta;
```

ah !



ah !



ombre et lumière

ombre ?

- un point est à l'ombre...
- si un objet se trouve entre le point et la source de lumière.

$$L_r(\vec{o}) = f_r(\vec{l}, \vec{o}) \times V(p, \vec{l}) \times L_e(\vec{l}) \times \cos\theta$$

$V(p, \vec{l}) = 1$ s'il n'y a pas d'intersection,
ie pas d'objet entre p et la lumière, sinon $V = 0$

ombres et lumières

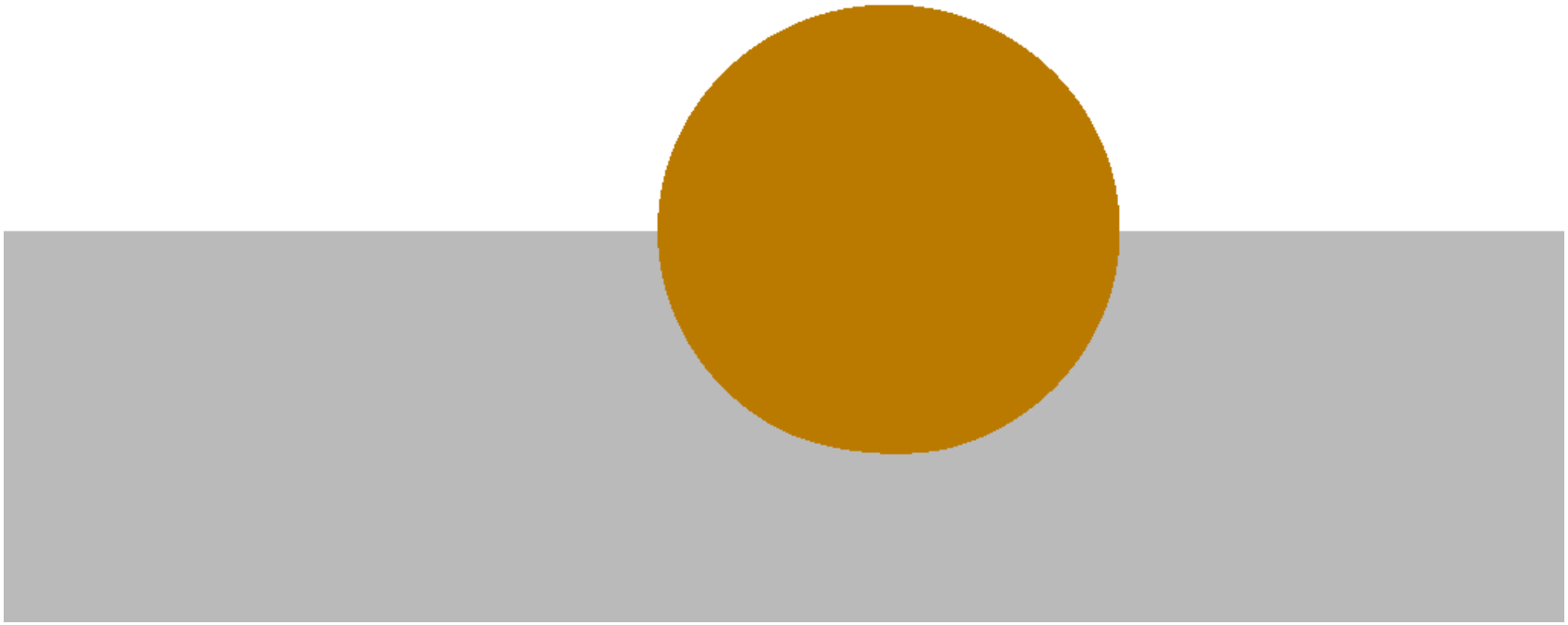
comment ça marche ?

- pour évaluer V, on va construire un nouveau rayon :
- origine : p
- direction : $\vec{l}$, vers la source de lumière,
- $+$ calculer les intersections avec les objets

p sera à l'ombre si on trouve une intersection valide, sinon il est éclairé par la source !

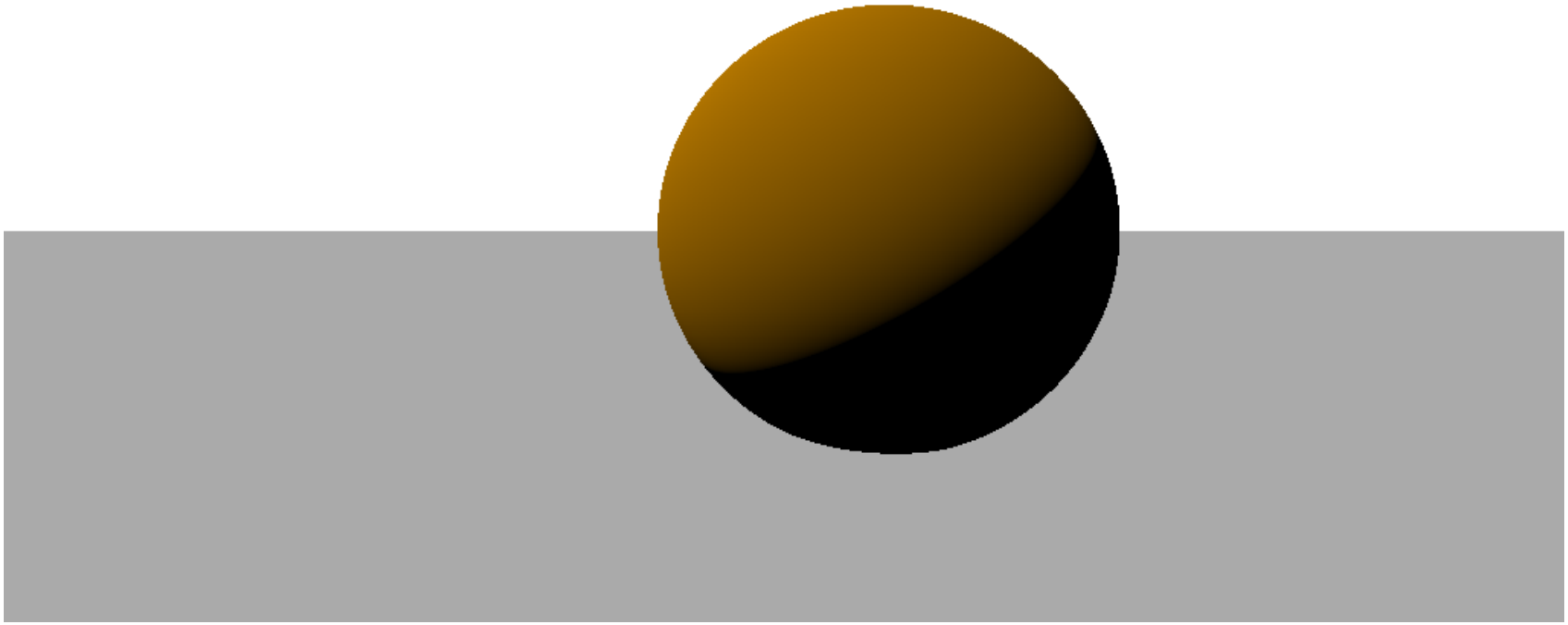
ombres et lumières

$$L_r(\vec{o}) = f_r(\vec{l}, \vec{o})$$



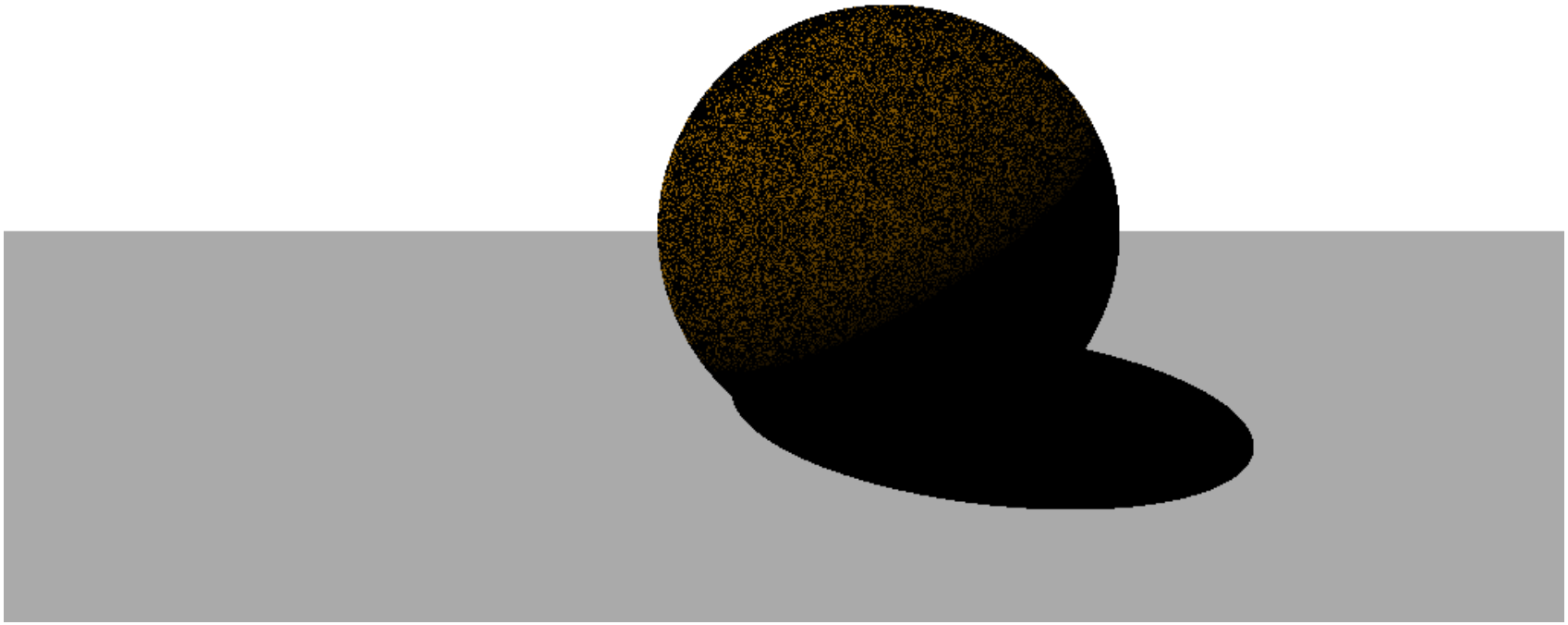
ombres et lumières

$$L_r(\vec{o}) = f_r(\vec{l}, \vec{o}) \times L_e(\vec{l}) \times \cos\theta$$



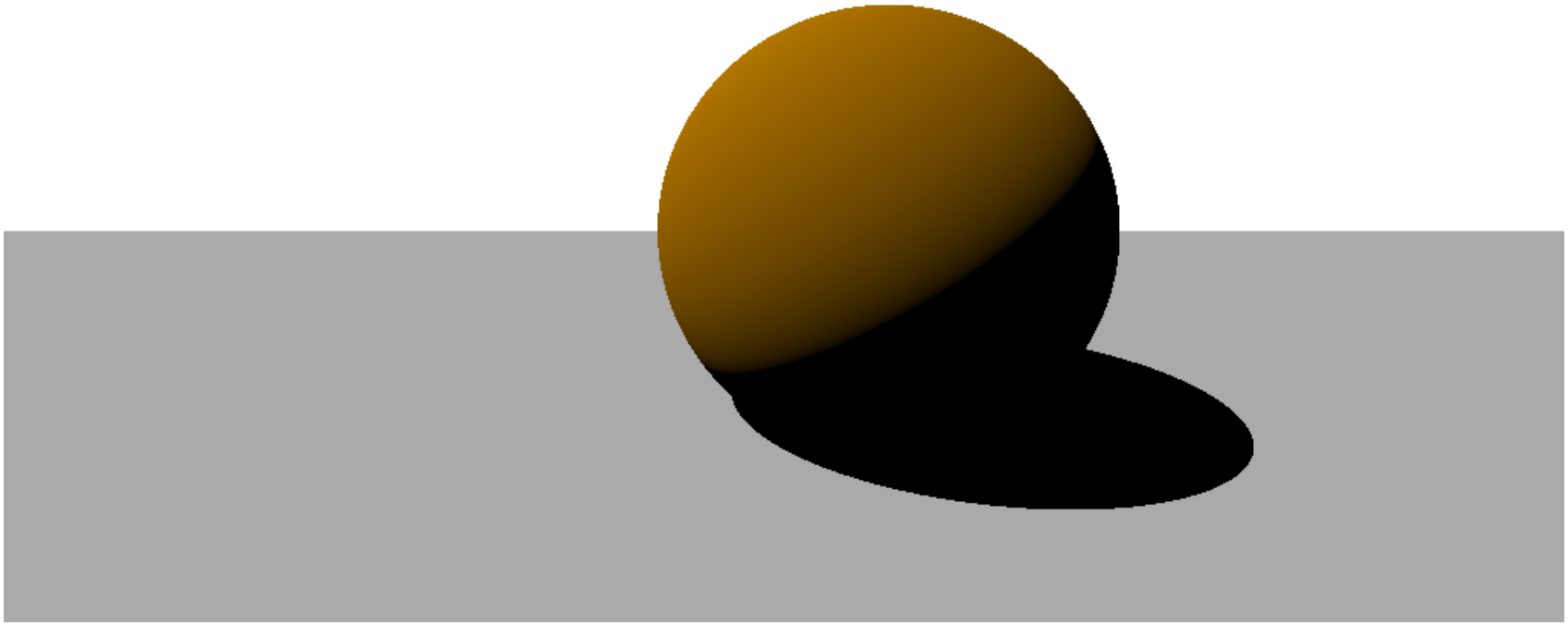
ombres et lumières

$$L_r(\vec{o}) = f_r(\vec{l}, \vec{o}) \times V(p, \vec{l}) \times L_e(\vec{l}) \times \cos\theta$$



c'est mieux, non ?

$$L_r(\vec{o}) = f_r(\vec{l}, \vec{o}) \times V(p + \epsilon \cdot \vec{n}, \vec{l}) \times L_e(\vec{l}) \times \cos\theta$$

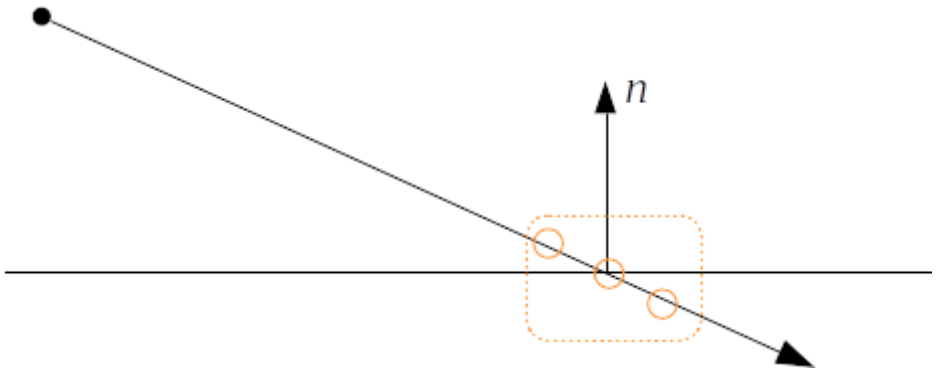


ombres et lumières

mais pourquoi ?

- les calculs avec les float sont des approximations...
- le point d'intersection ne se trouve *jamais* sur la surface de l'objet,
- mais un peu au dessus, ou, un peu au dessous...

ombres et lumières

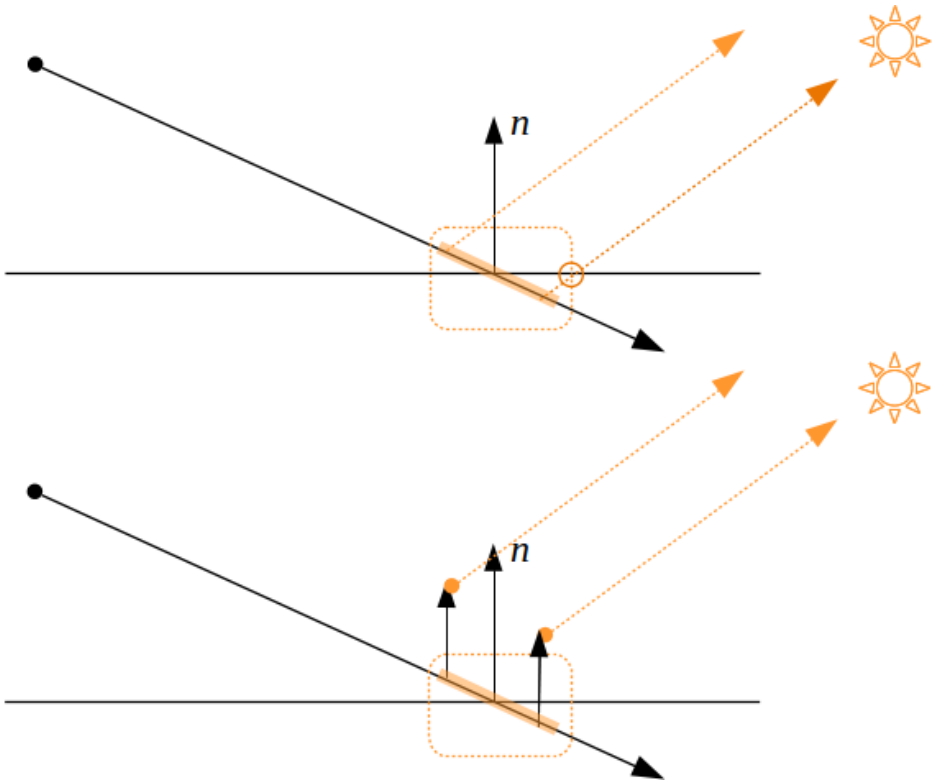


ombres et lumières

et alors ?

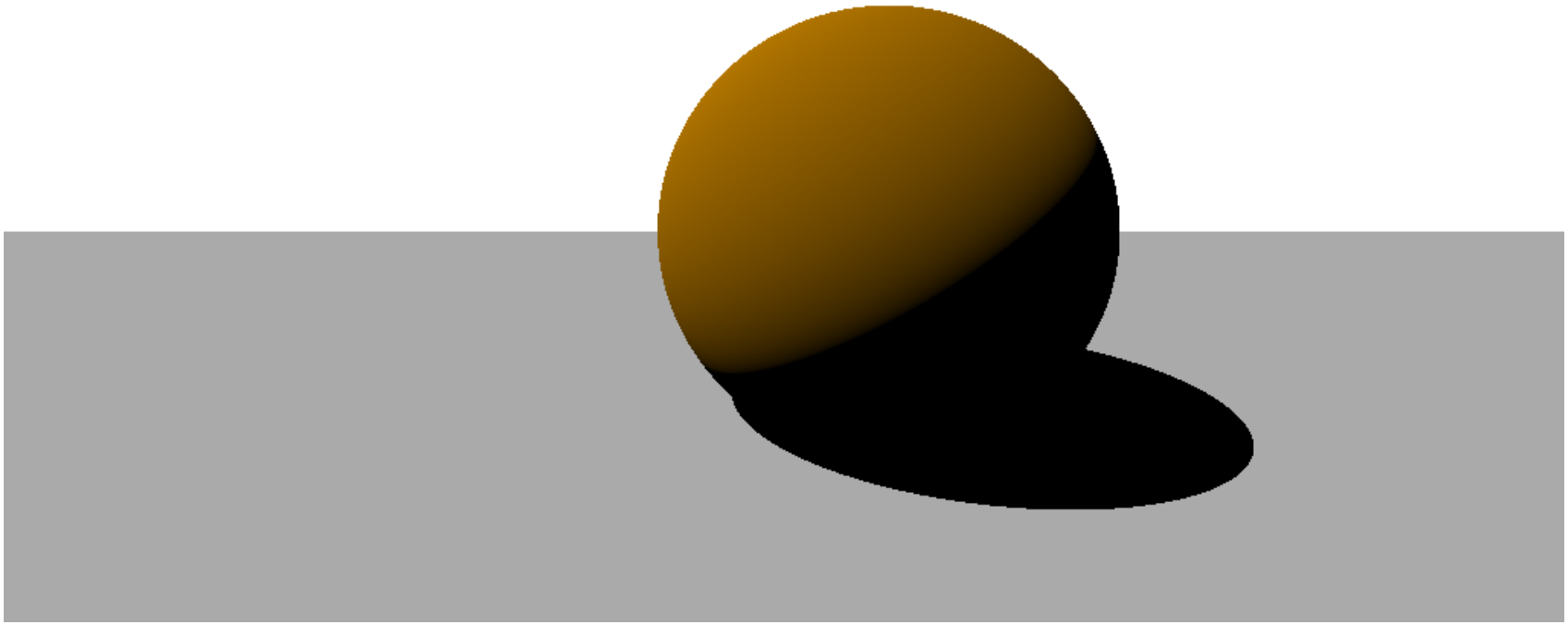
- il faut décoller l'origine du rayon de la surface,
- en le poussant le long de la normale :
- $o = p + \epsilon \cdot \vec{n}$

ombres et lumières



c'est mieux, non ?

$\epsilon = 0.001$



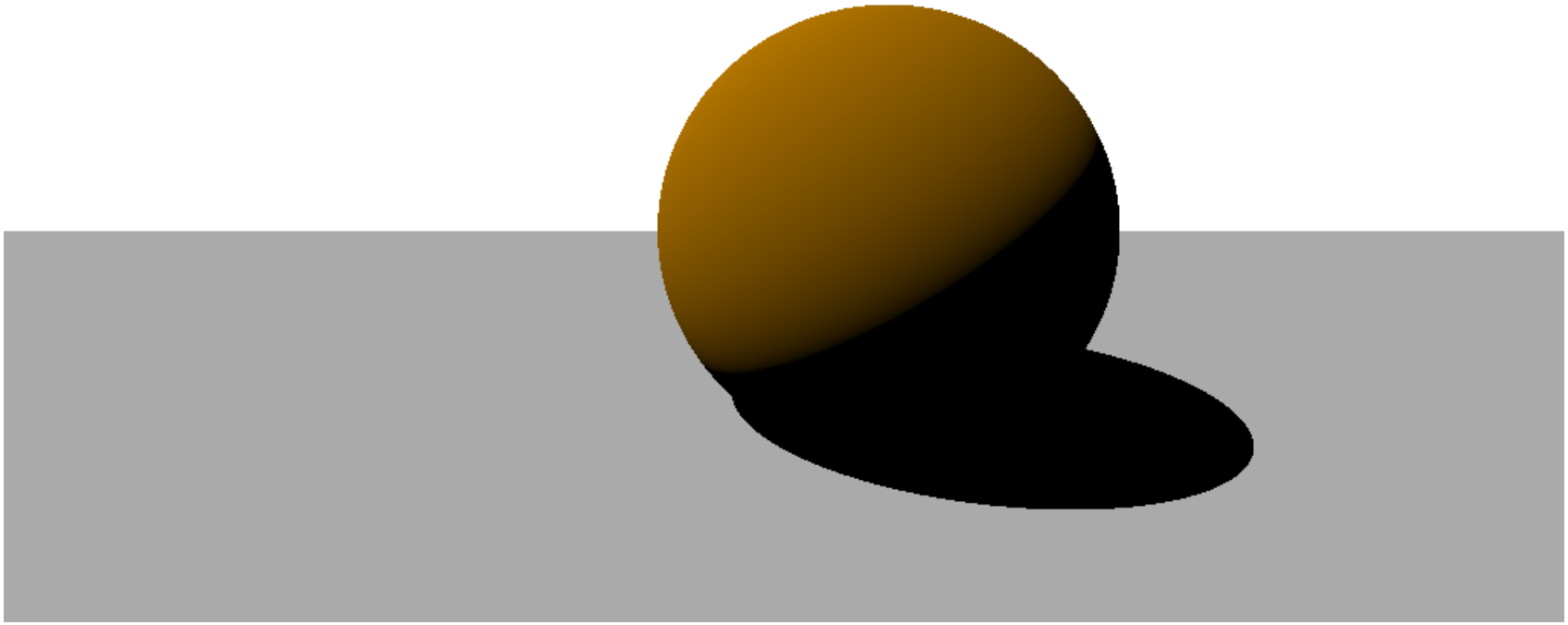
et les pénombres ?

pénombres ?

- ??
- les ombres du soleil sont marquées / nettes,
- mais le ciel éclaire presque autant que le soleil...

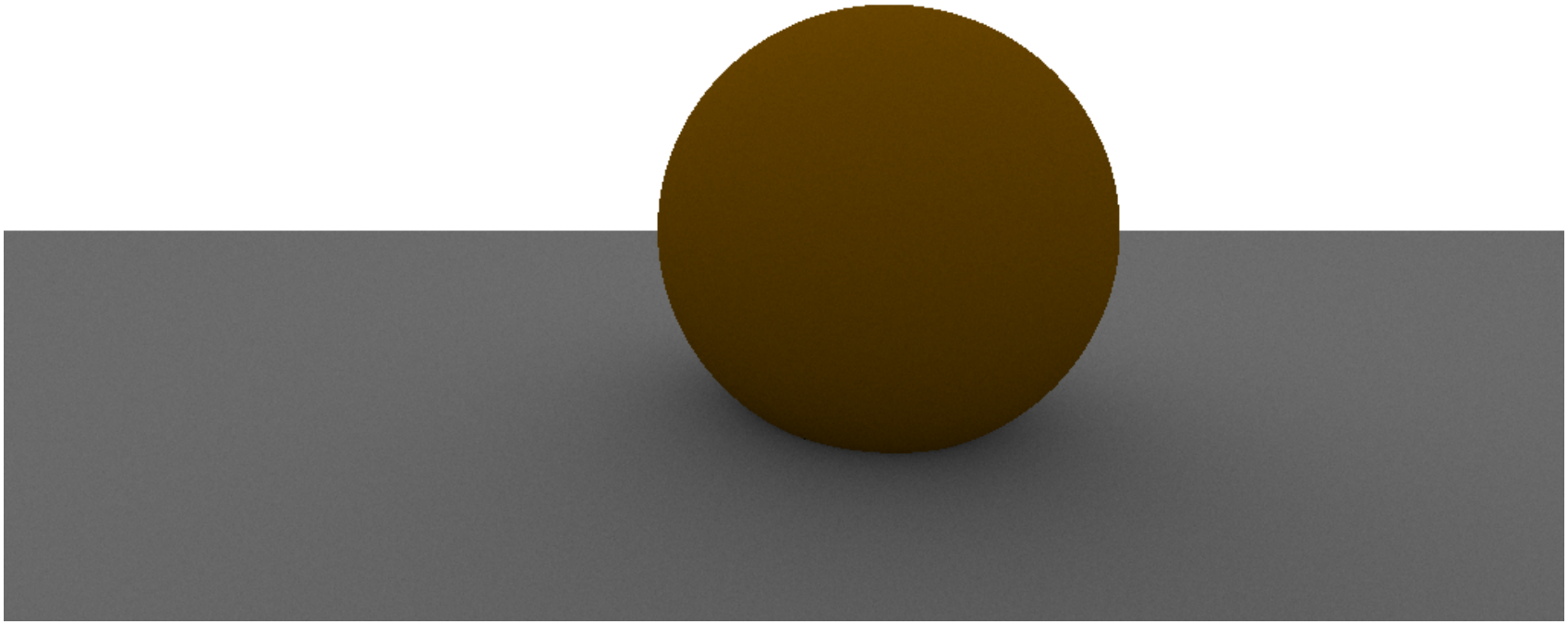
ombres et pénombres...

un soleil (une direction)



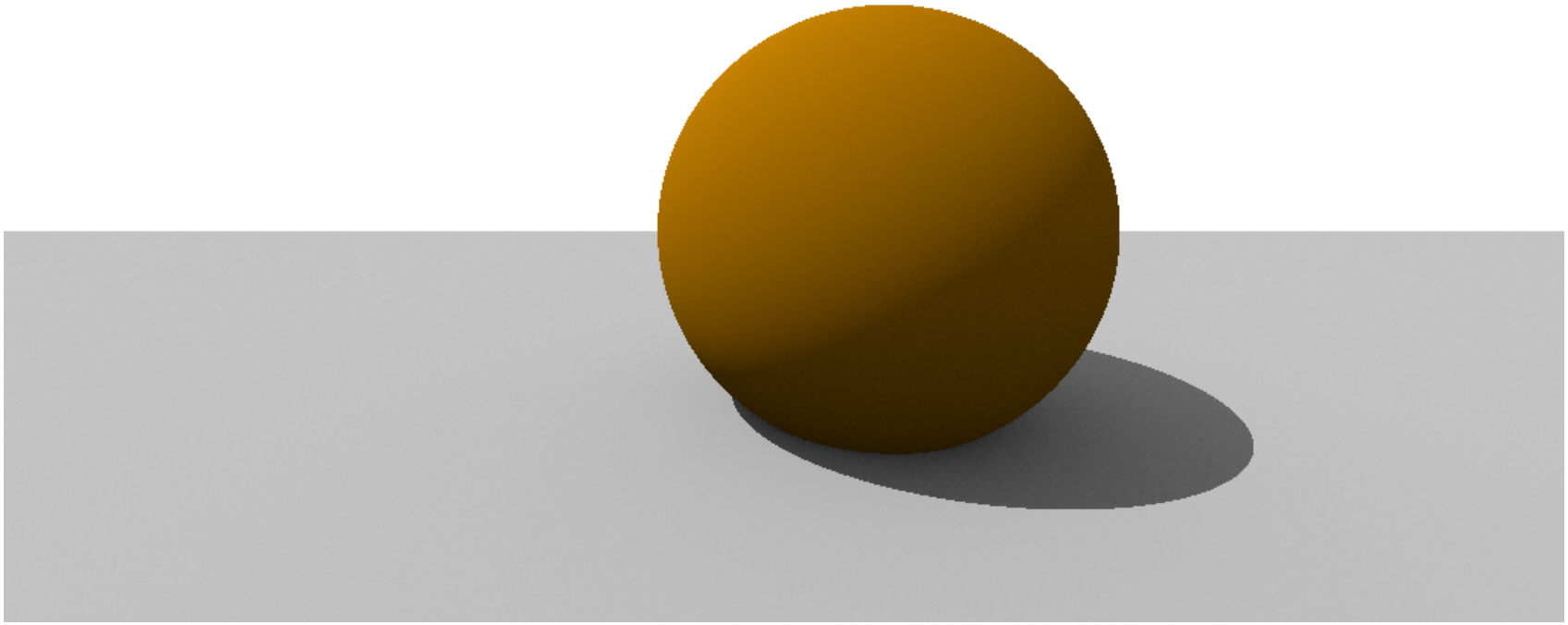
ombres et pénombres...

le ciel

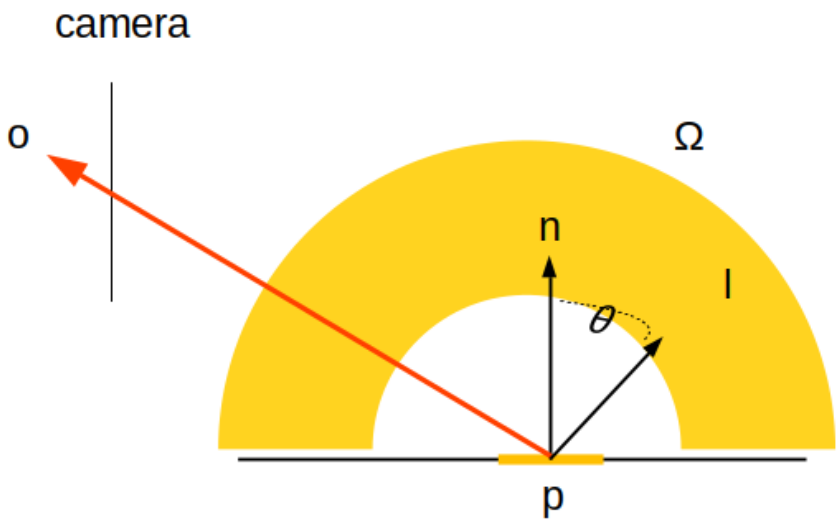


ombres et pénombres...

un soleil + le ciel...



ombres et pénombres...



ombres et pénombres...

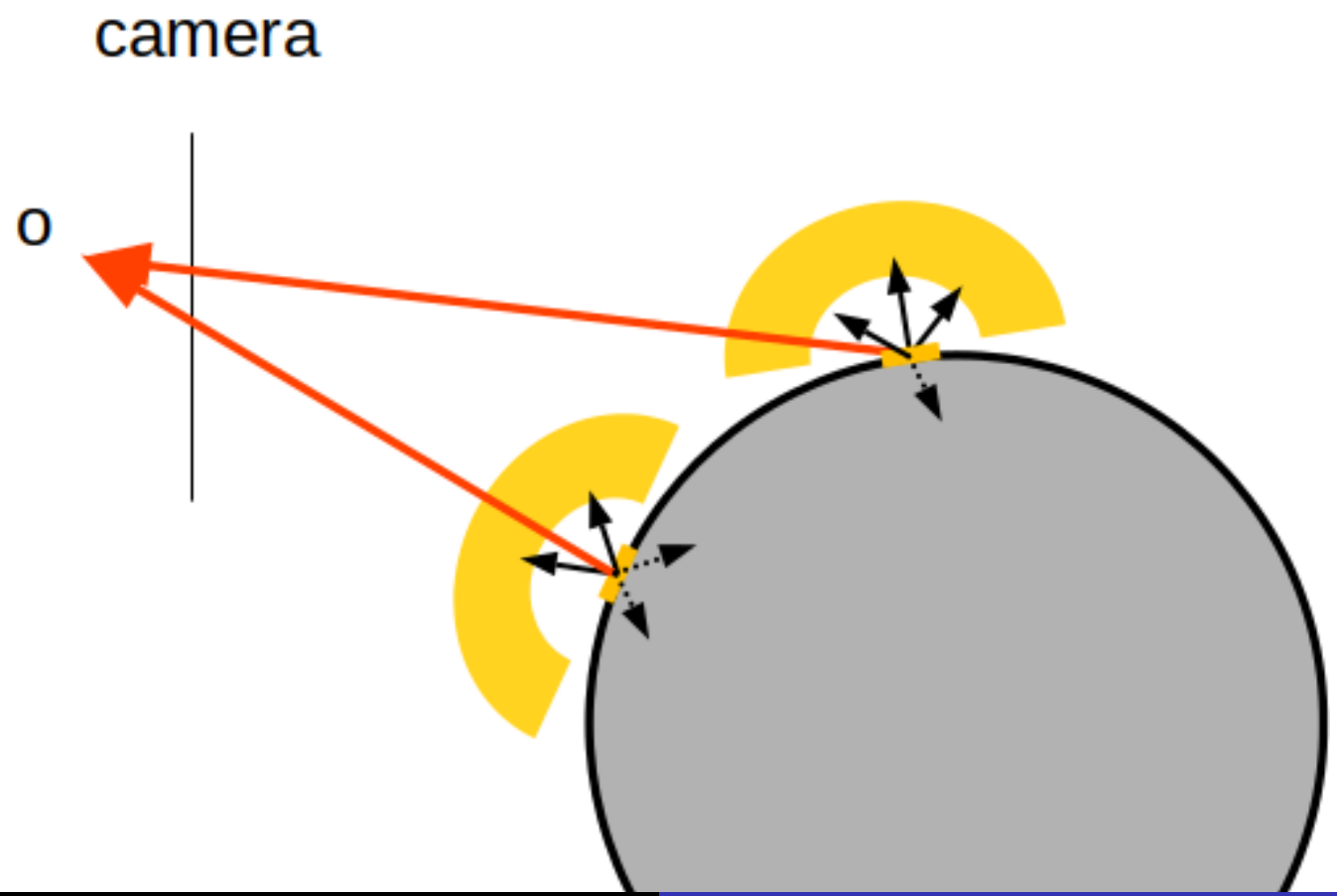
euh ?

- avec pleins de directions choisies dans le dome, cf Ω ?
- comment construire des directions ?
- avec 2 angles $\theta \in [0..\pi]$ et $\phi \in [0..2\pi]$, on peut retrouver les coordonnées du vecteur :
  $x = \cos\phi \sin\theta$
  $y = \sin\phi \sin\theta$
  $z = \cos\theta$

attention, on construit des directions dans toutes les directions, sur une sphère, il faut vérifier que la direction fait bien partie de Ω...

$\vec{l} \in \Omega$

on veut trouver des directions au dessus de la surface au point p, quelque soit son orientation...



ombres et pénombres...

euh ?

- avec pleins de directions choisies dans le dome, cf Ω ?
- il ne reste plus qu'à calculer la moyenne de la lumière réfléchie par chaque direction $\vec{l_k}$:

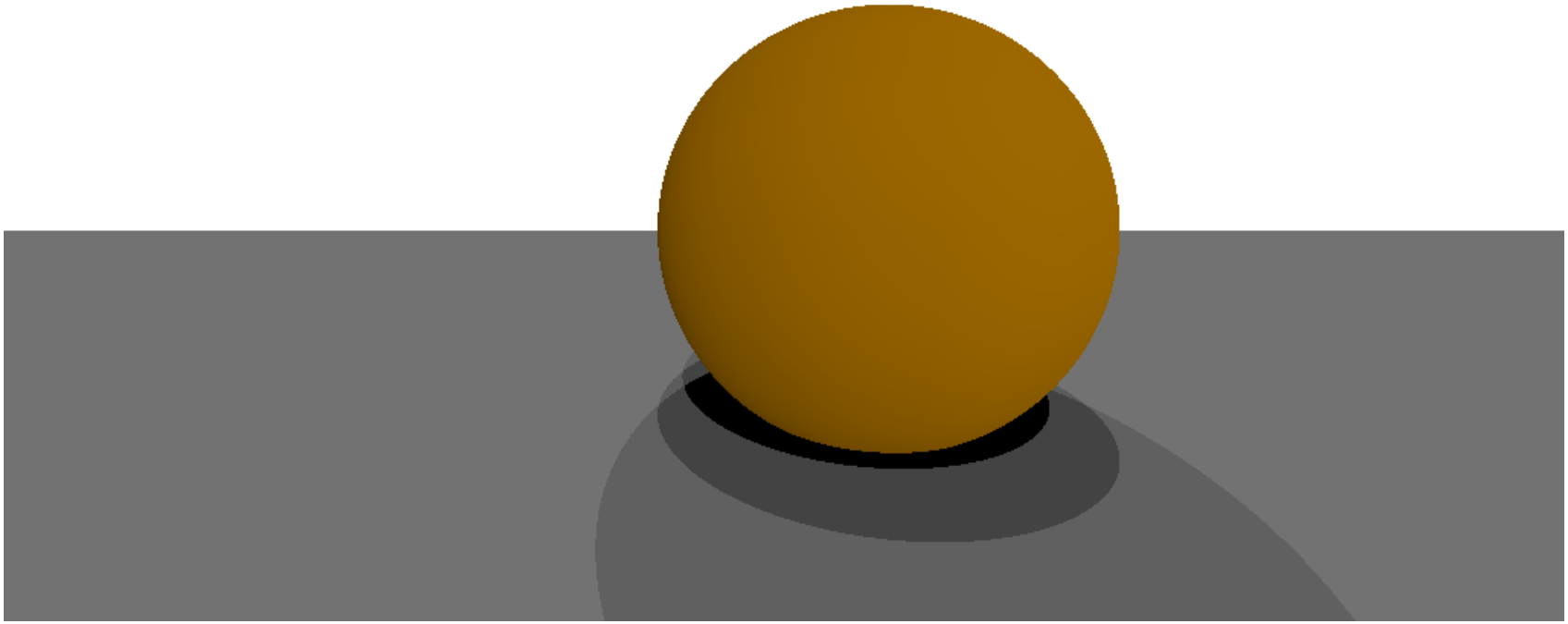
$$L_r(p,\vec{o}) = \frac{1}{N}\sum_{k=1}^{N} f_r(\vec{l_k},\vec{o}) \times V(p,\vec{l_k}) \times L_e(\vec{l_k}) \times \cos\theta_k$$

remarque : $L_e(\vec{l}) = 0$ si $\vec{l}$ n'est pas orientée vers le ciel,
ie $L_e(\vec{l}) = 0$ si $\vec{l} \cdot \vec{Y} < 0$.

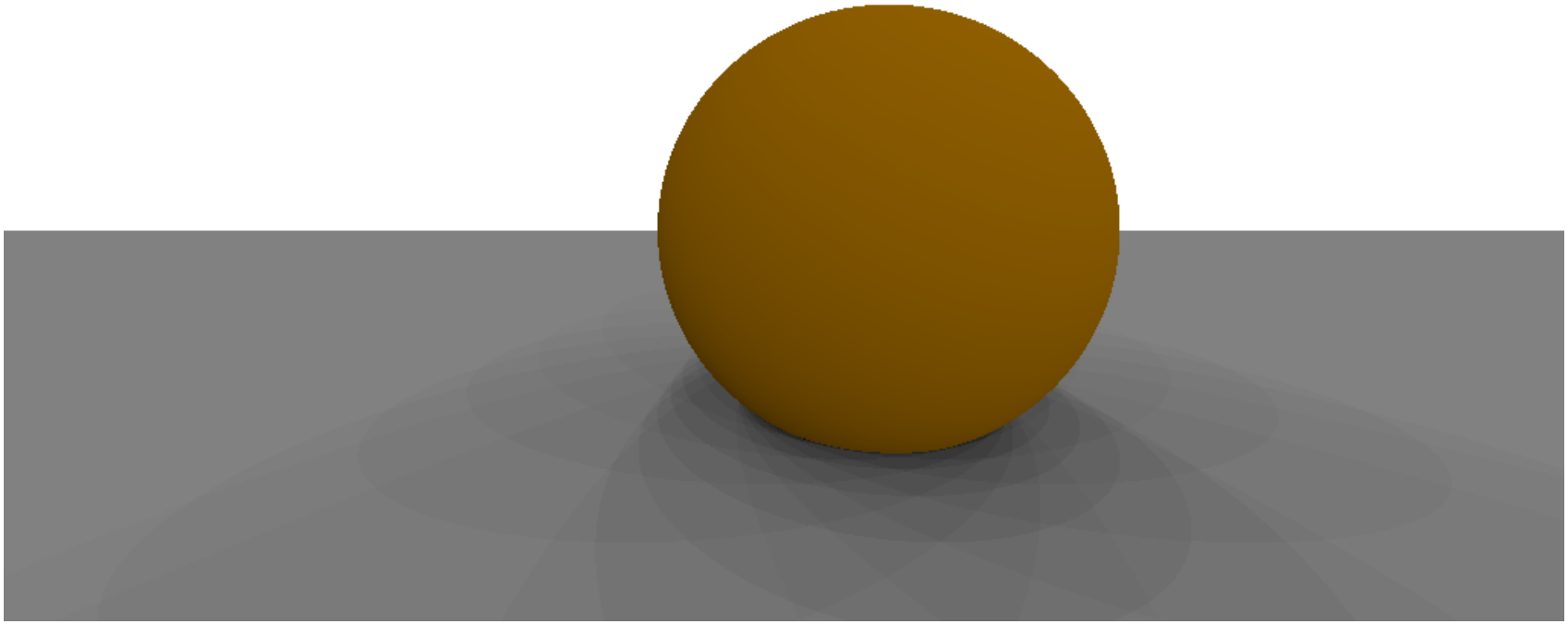
des sources sur un dome

```
#include <cmath>

std::vector<Vector> directions;
for(int i= 0; i < D; i++)
for(int j= 0; j < D; j++)
{
    float r1= float(i) / float(D);
    float r2= float(j) / float(D);
    float cos_theta= 1 - 2*r2;
    float sin_theta= 2 * std::sqrt( r2 * (1 - r2) );
    float phi= r1 * float(2*M_PI);

    Vector d= Vector(
        std::cos(phi) * sin_theta,
        std::sin(phi) * sin_theta,
        cos_theta
    );

    if(dot(d,n) > 0)     // dans le ciel...
        directions.push_back( d );
}
```

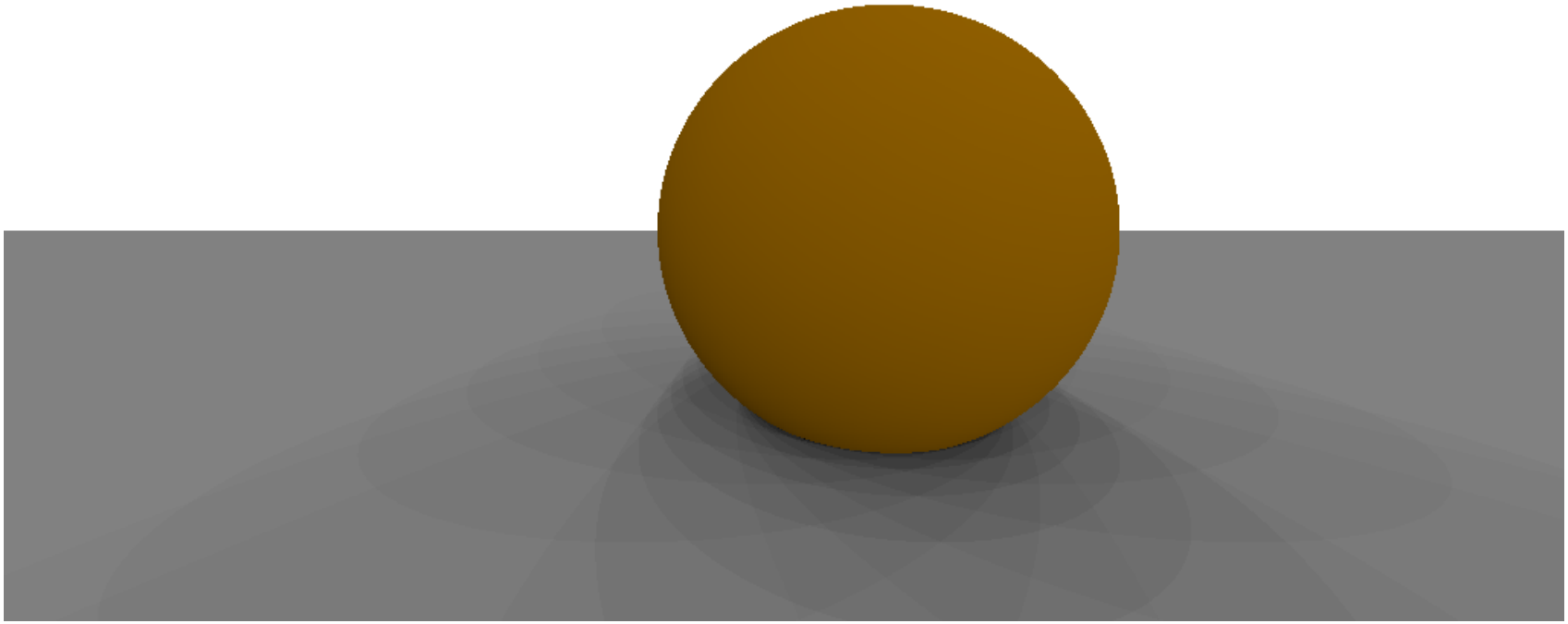
des sources sur un dome, 16, $D = 4$



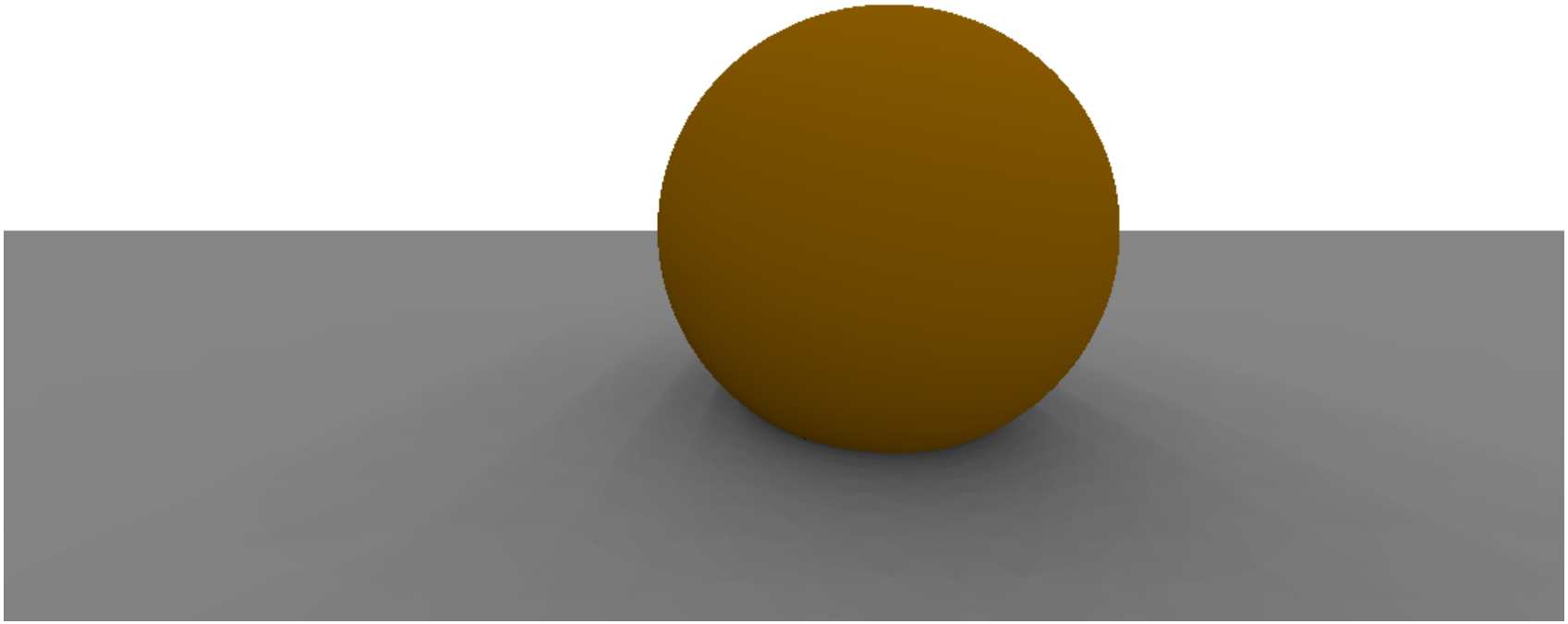
des sources sur un dome, 36, $D = 6$



des sources sur un dome, 64, $D = 8$



des sources sur un dome, 256, $D = 16$



des sources sur un dome...

euh ?

- c'est bizarre, ces espèces de bandes...
- elles disparaissent si D est grand,
- mais les calculs commencent à être longs...

autre chose ?

des directions aléatoires dans un dome

2 solutions :

- on peut répartir les directions sur un dome,
- ou on peut aussi le faire aléatoirement ! pour chaque pixel.

et oui, ça marche ! cf méthodes de Monte Carlo, 1950...
les détails sont dans le cours de M2

des nombres aléatoires

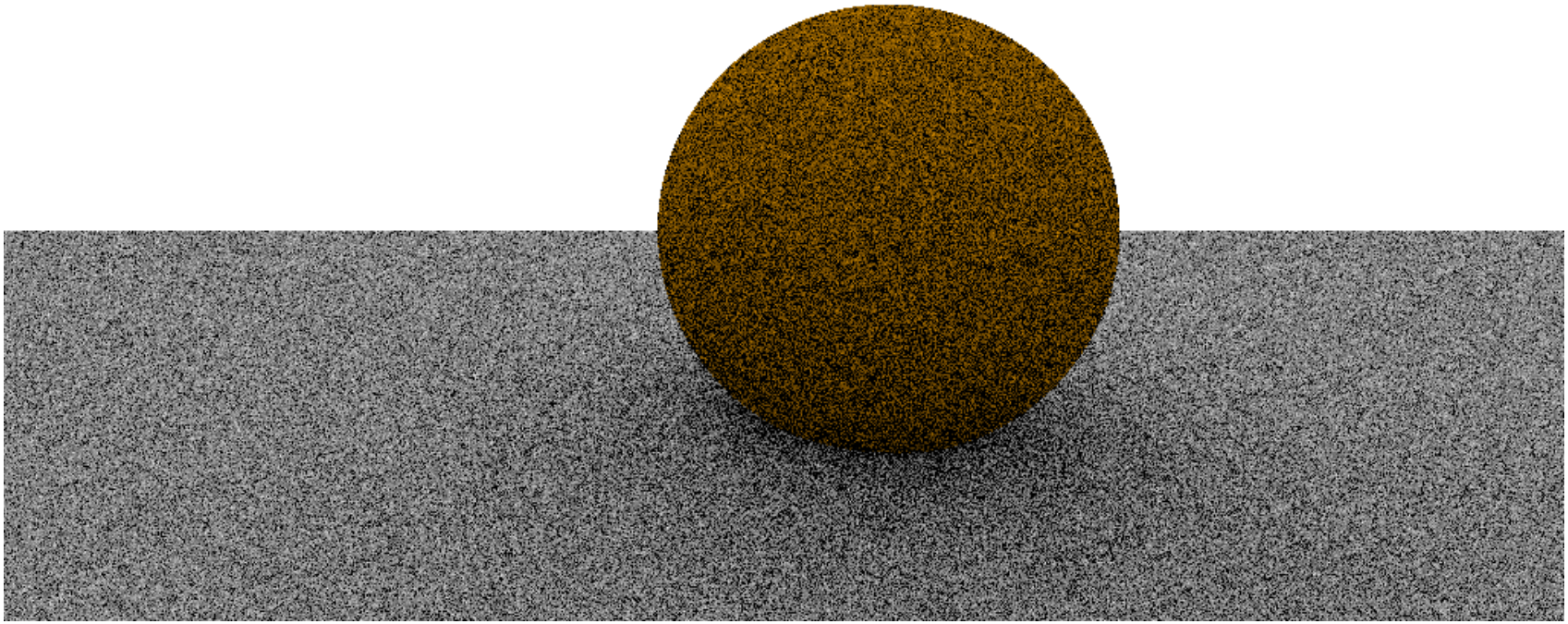
```
#include <random>

// etape 1 / initialisation : 1 fois au debut du programme
std::random_device hwseed;
unsigned seed= hwseed();

// etape 1 / generateur de nombres aleatoires
std::default_random_engine rng( seed );
std::uniform_real_distribution<float> uniform;

// etape 2 / a chaque fois :
float u1= uniform( rng );       // nombre aleatoire entre 0 et 1
float u2= uniform( rng );       // un autre...
```

des directions aléatoires dans un dome

```
#include <cmath>
#include <random>

std::vector<Vector> directions;
for(int i= 0; i < N; i++)
{
    float r1= uniform( rng );
    float r2= uniform( rng );
    float cos_theta= 1 - 2*r2;
    float sin_theta= 2 * std::sqrt( r2 * (1 - r2) );
    float phi= r1 * float(2*M_PI);

    Vector d= Vector(
        std::cos(phi) * sin_theta,
        std::sin(phi) * sin_theta,
        cos_theta
    );

    if(dot(d,n) > 0)     // dans le ciel...
        directions.push_back( d );
}
```

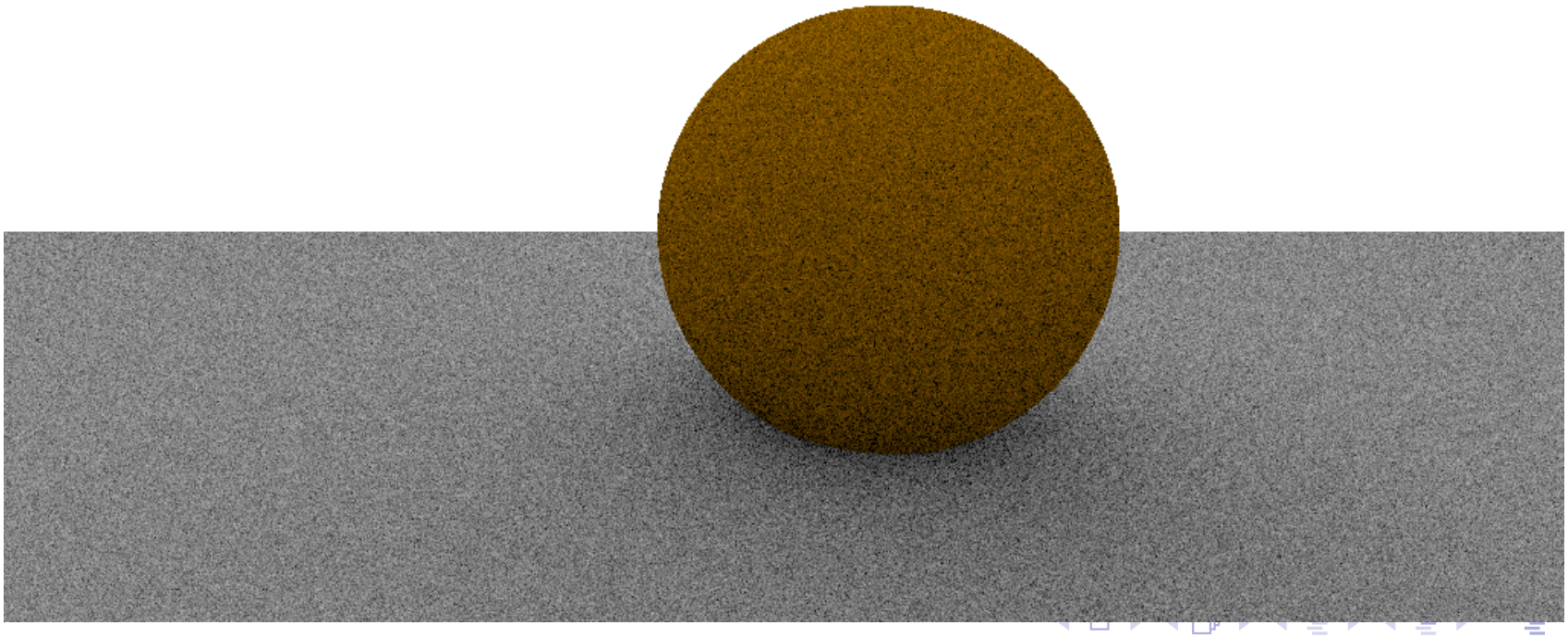
des directions aléatoires dans un dome, 4

rappel : chaque pixel utilise des directions différentes...

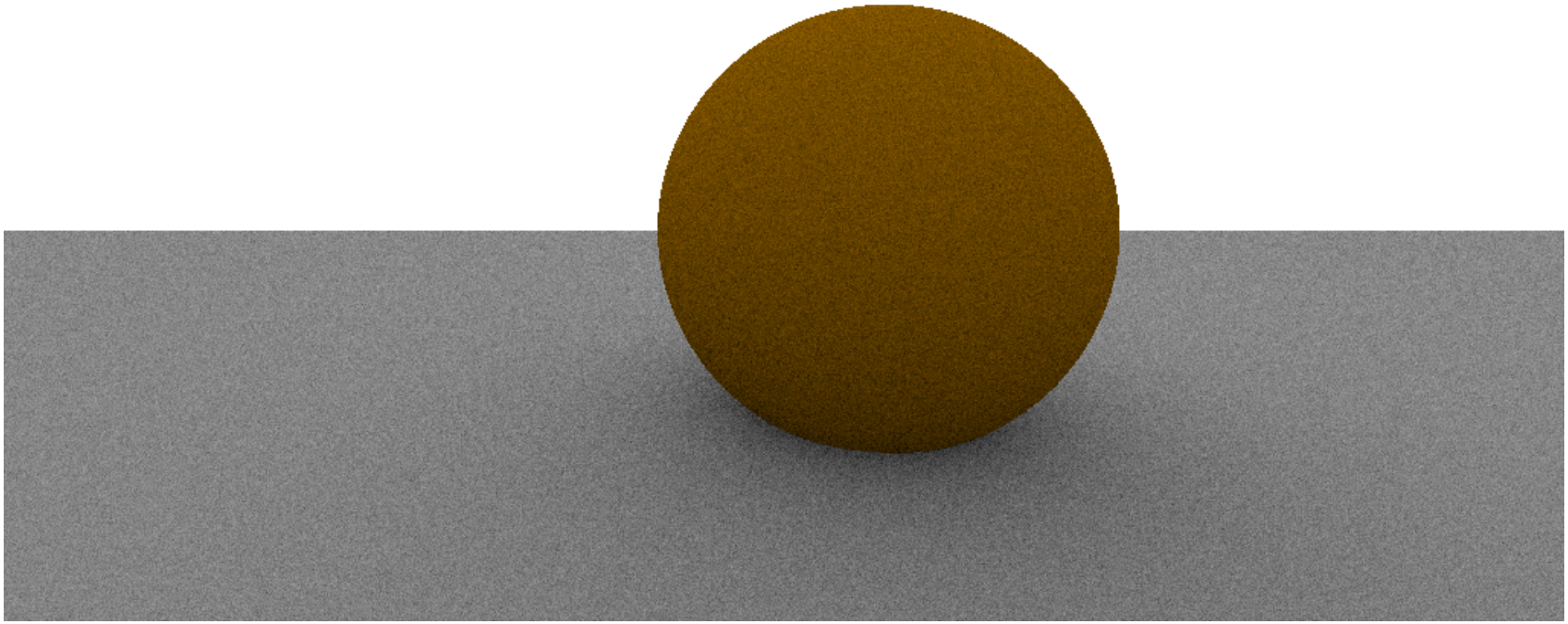


des directions aléatoires dans un dome, 16

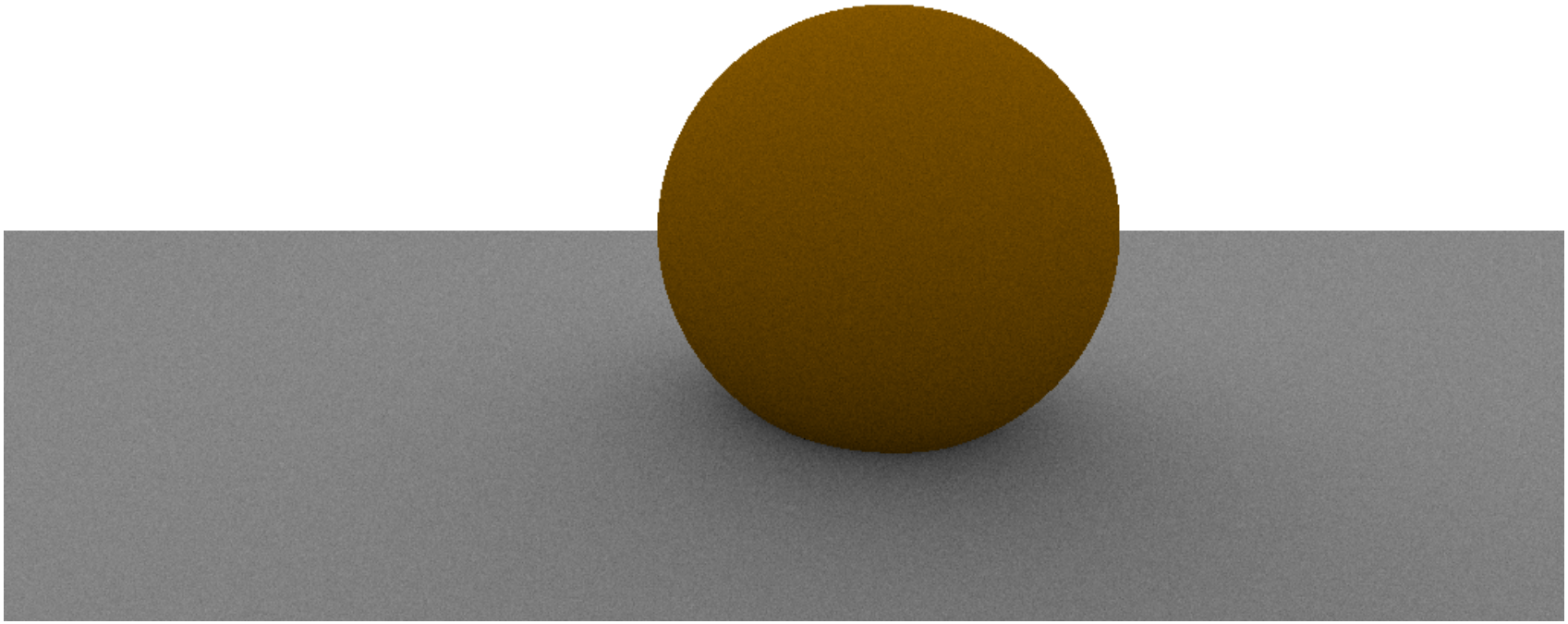
constat : c'est moche,
mais on imagine facilement le "bon" résultat...



des directions aléatoires dans un dome, 64

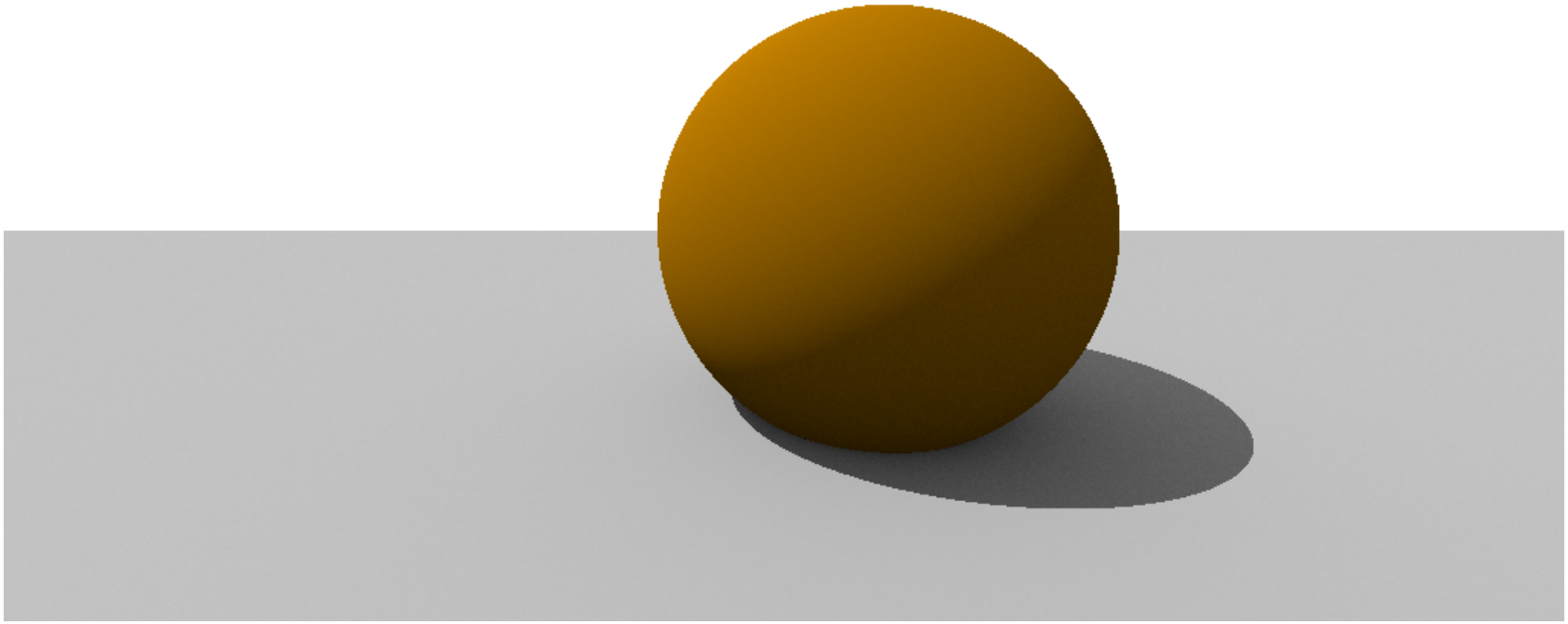


des directions aléatoires dans un dome, 256



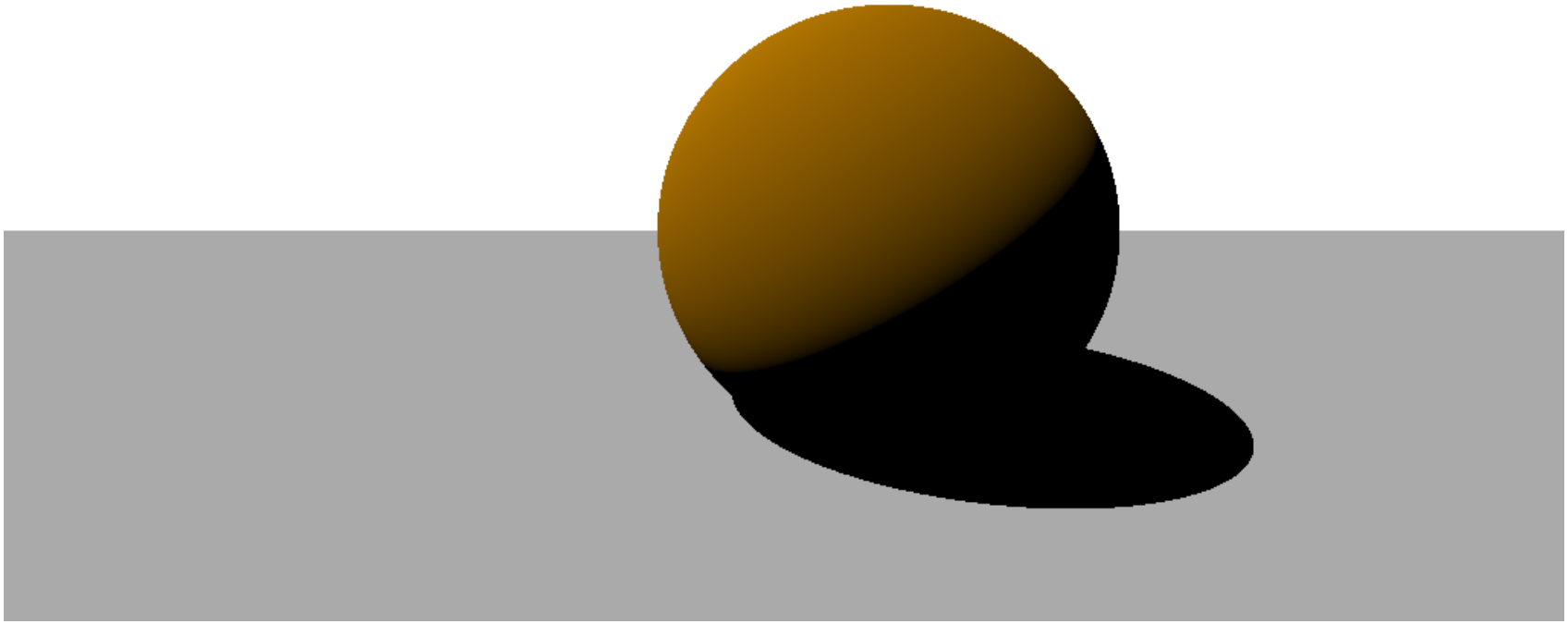
ombres et pénombres...

mais maintenant, on peut faire ça : soleil + ciel



ombres et pénombres...

rappel : soleil tout seul



sources ponctuelles

on peut faire la même chose avec des sources "points" :

- mais la physique est un peu differente,
- un point s éclaire le point p :
- $L_r = f_r \times L_e \frac{1}{||s-p||^2} \times \cos\theta$

sources surfaciques

panneau lumineux :

- disposer des sources points dans un carré, ie un panneau :
- sur une grille,
- ou aléatoirement,
- en suivant le même principe que pour une direction et un dome...

des points dans une grille

```
Point a= Point(0, 0, -1);          // origine
Vector u= Vector(1, 0, 0);         // axe 1
Vector v= Vector(0, 1, 0);         // axe 2

std::vector<Point> sources;
for(int i= 0; i < 10; i++)
for(int j= 0; j < 10; j++)
{
    Point s= a + i*u + j*v;        // position du point dans la grille
    sources.push_back( s );
}
// les coordonnées sont entre 0 et 10
```

mais pas très pratique :
le nombre de points change la taille de la grille...

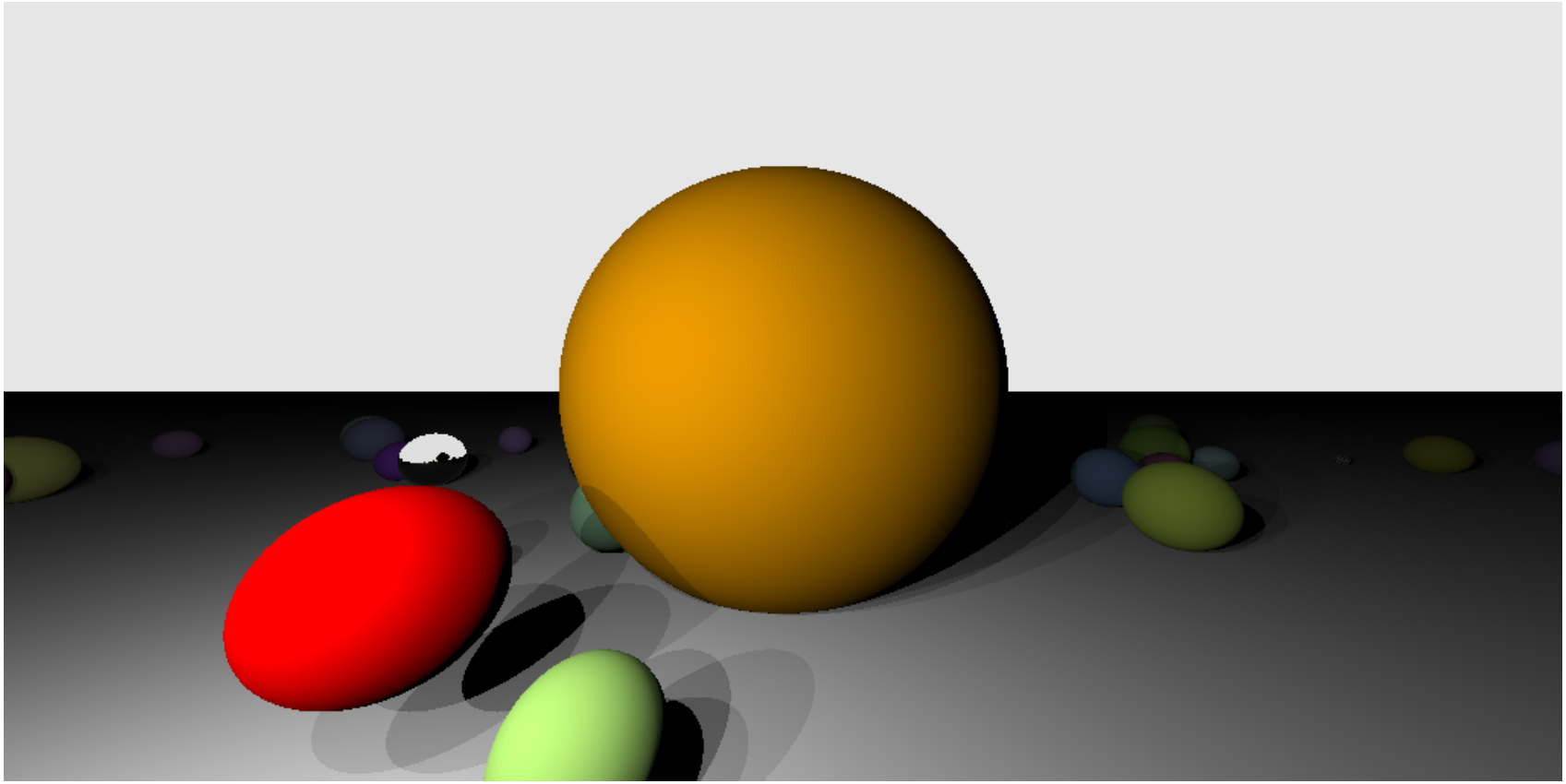
des points dans une grille

mieux :

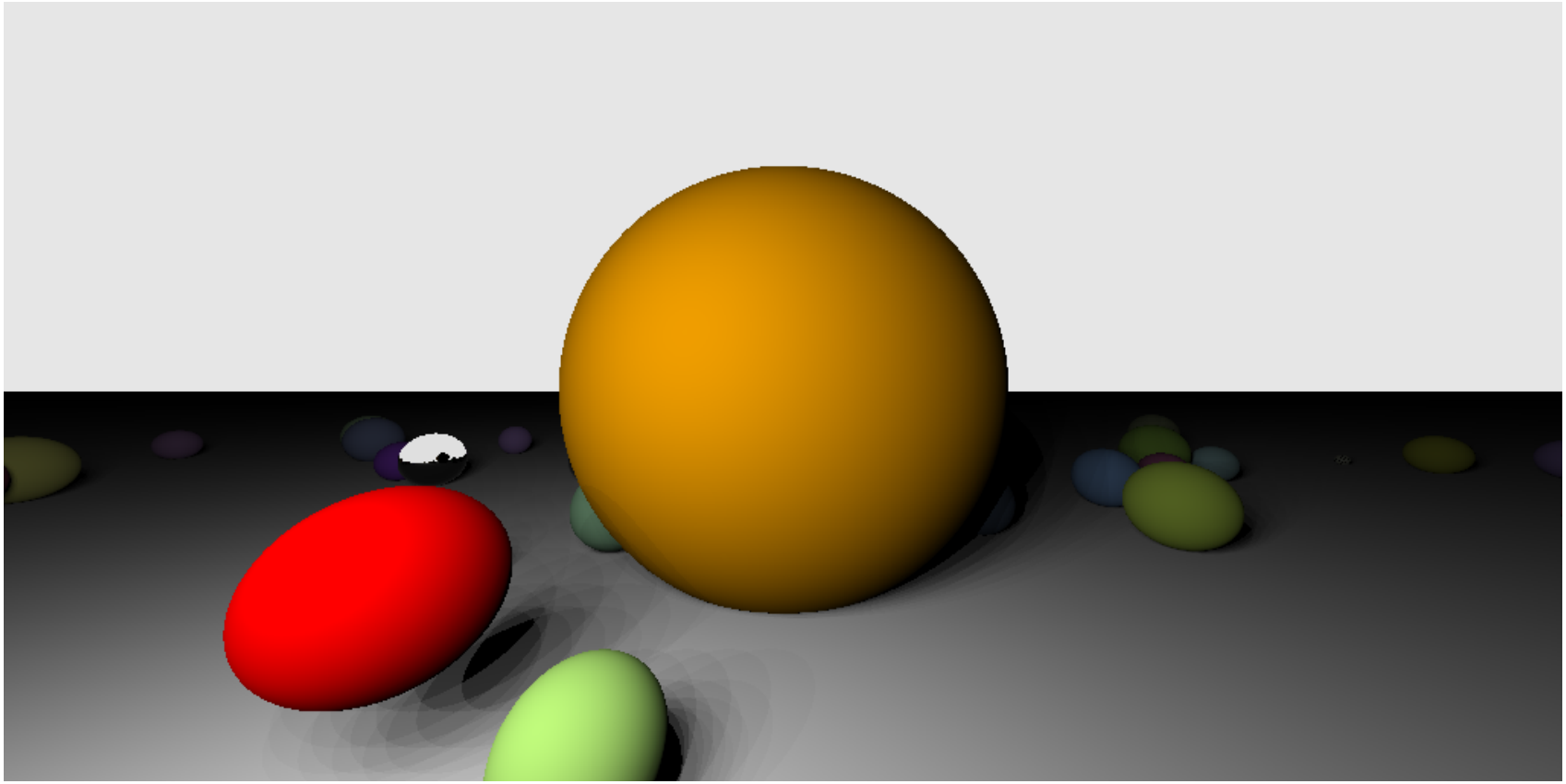
```
Point a= Point(0, 0, -1);          // origine
Vector u= Vector(1, 0, 0);         // axe 1
Vector v= Vector(0, 1, 0);         // axe 2

std::vector<Point> sources;
for(int i= 0; i < 10; i++)
for(int j= 0; j < 10; j++)
{
    float b= float(i) / float(10);
    float c= float(j) / float(10);

    Point s= a + b*u + c*v;        // position du point dans la grille
    sources.push_back( s );
}
// les coordonnées sont entre 0 et 1
```

il suffit d'ajuster la longueur des axes $\vec{u}$ et $\vec{v}$...

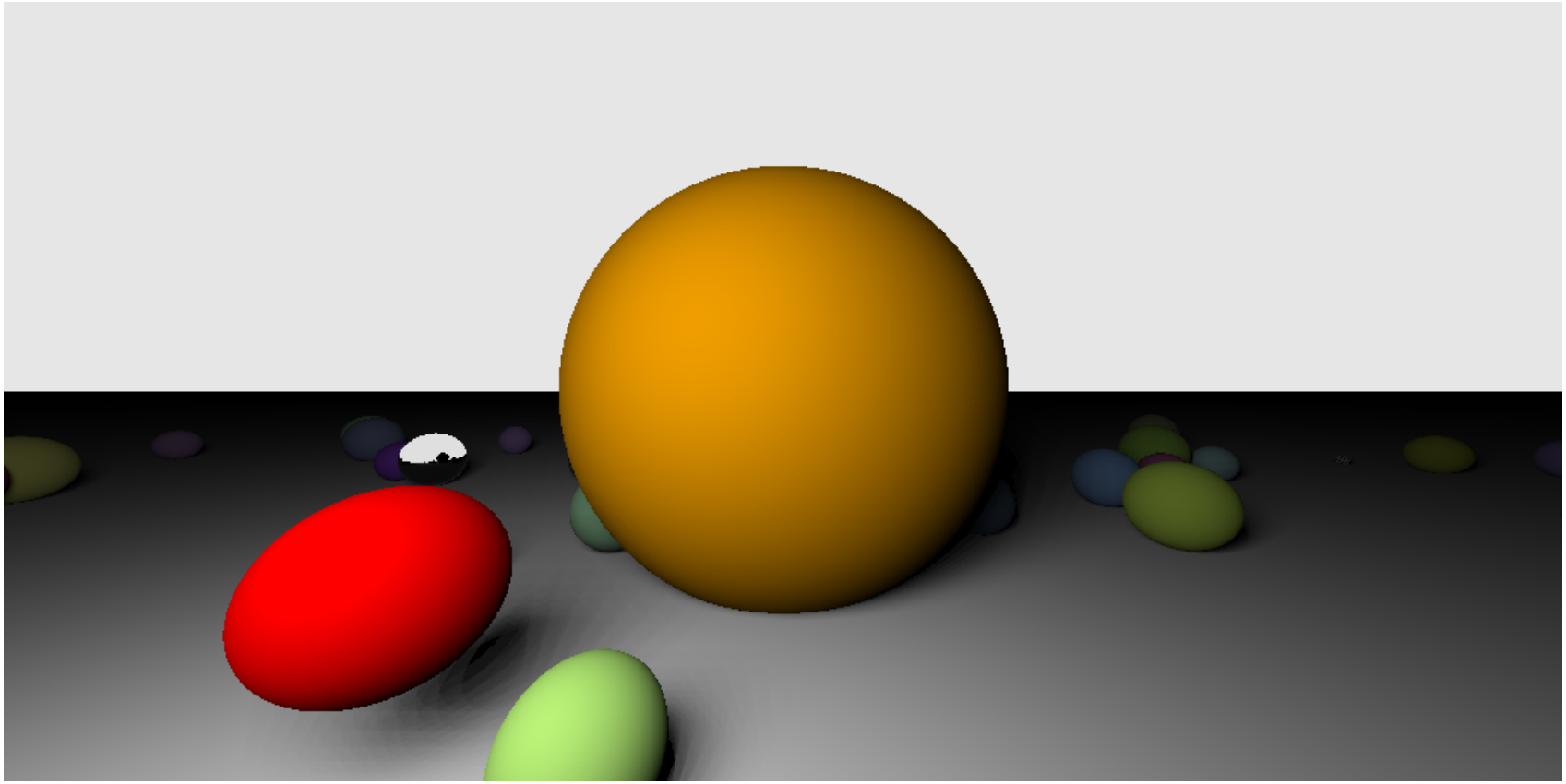
des points dans une grille, 4



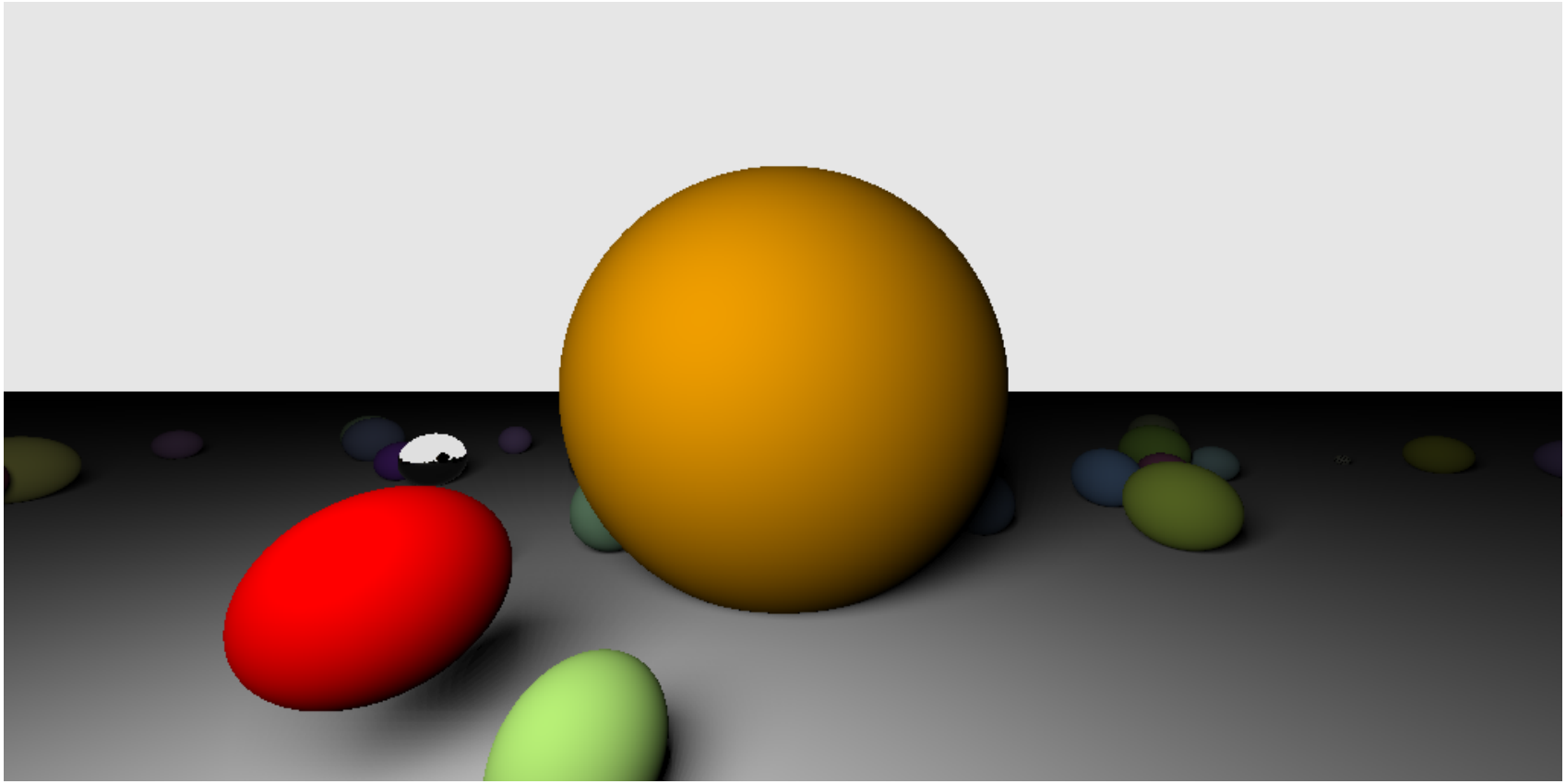
des points dans une grille, 16



des points dans une grille, 64



des points dans une grille, 256



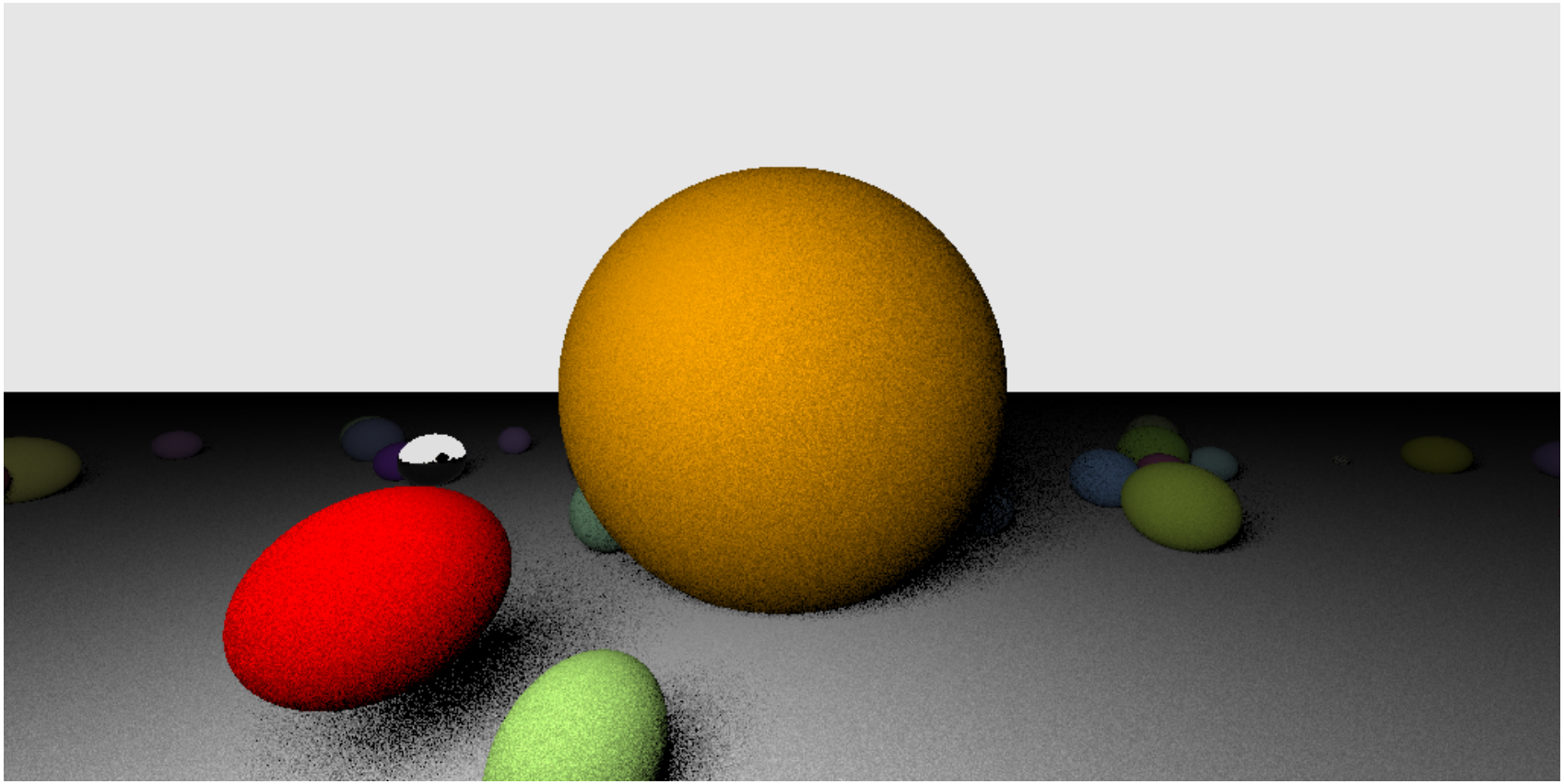
des points aléatoires dans une grille

```
#include <random>

Point a= Point(0, 0, -1);
Vector u= Vector(1, 0, 0);
Vector v= Vector(0, 1, 0);

std::vector<Point> sources;
for(int i= 0; i < N; i++)
{
    float u1= uniform( rng );
    float u2= uniform( rng );

    // les coordonnées sont entre 0 et 1
    Point s= a + u1*u + u2*v;

    sources.push_back( s );
}
```

très facile de changer le nombre de points...

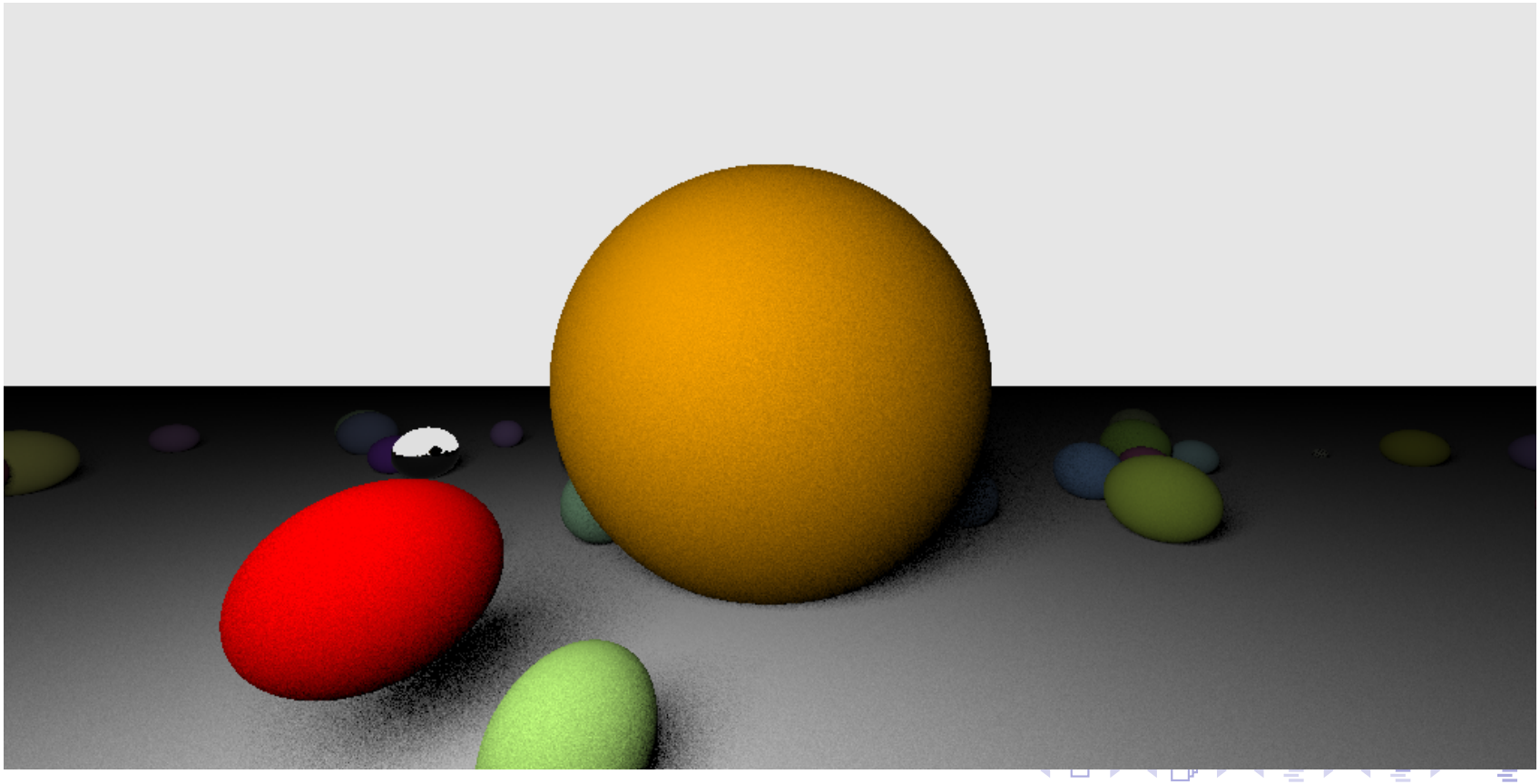
des points aléatoires dans une grille, 4

rappel : chaque pixel utilise des points différents...

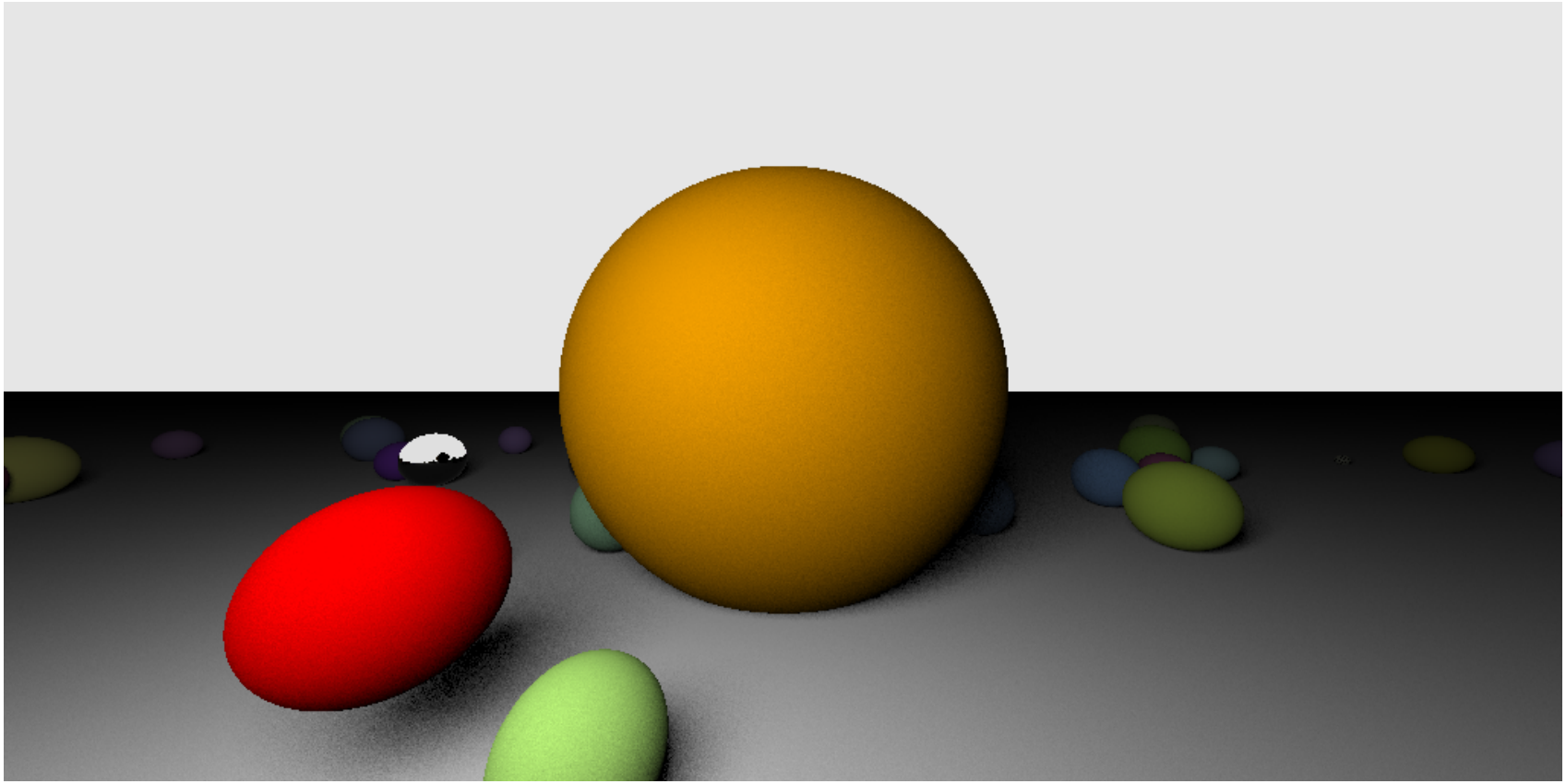


des points aléatoires dans une grille, 16

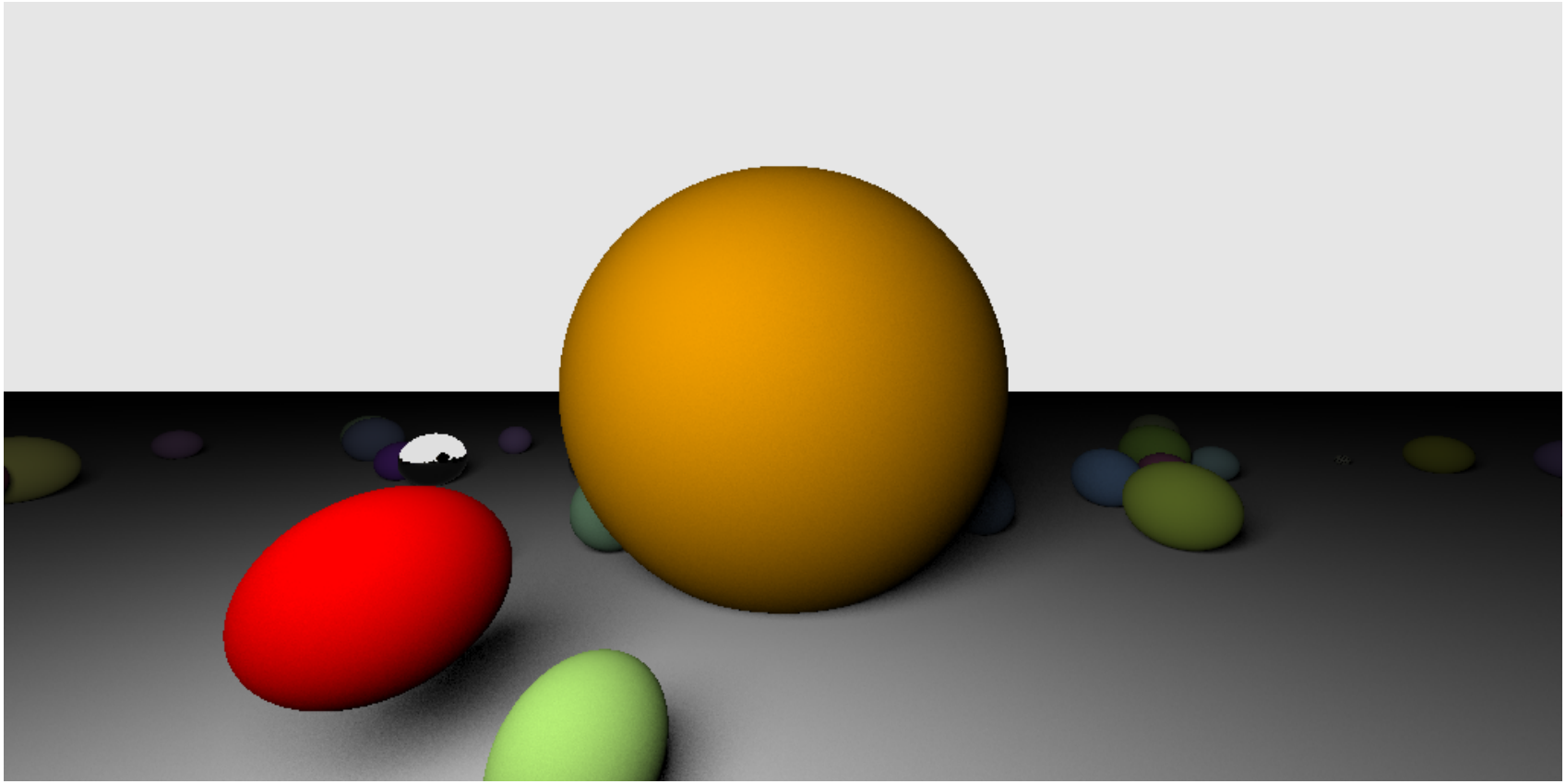
c'est assez moche aussi,
mais on imagine plus facilement le "bon" résultat...



des points aléatoires dans une grille, 64

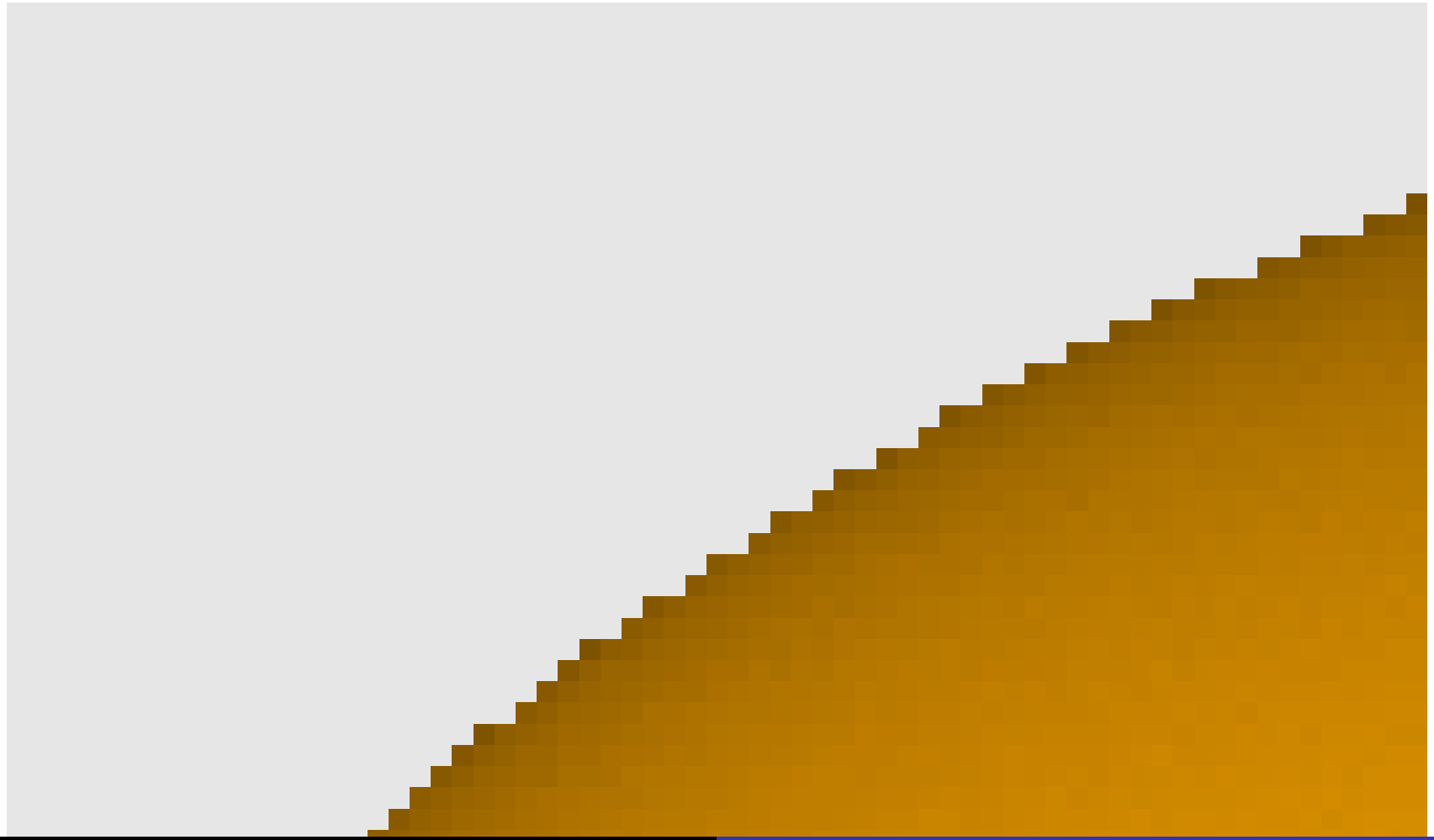


des points aléatoires dans une grille, 256



bonus : anti-aliasing

ça aussi, c'est bien moche...



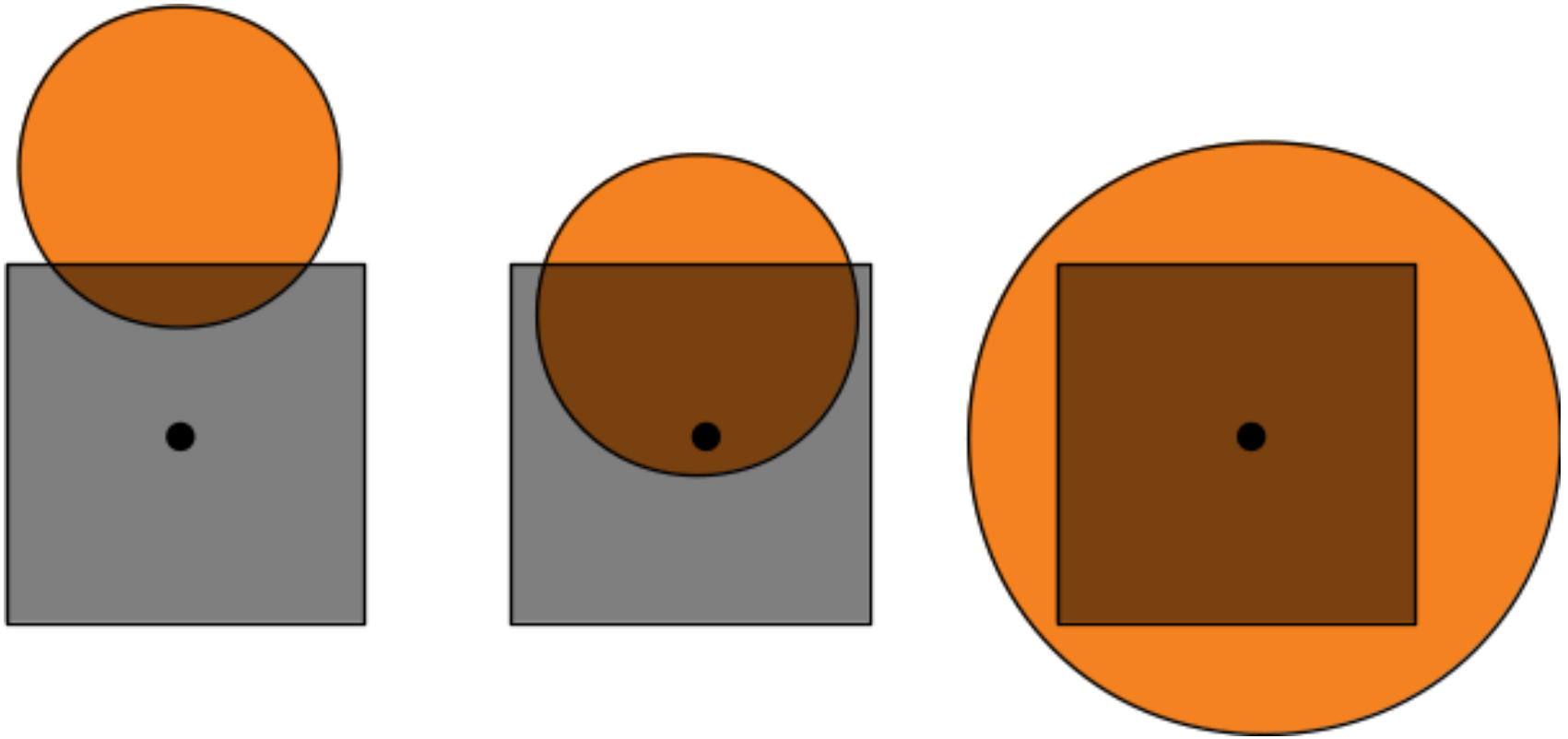
anti-aliasing

pourquoi ?

- pour chaque pixel, on construit un rayon et on trouve, ou pas, une intersection,
- et on calcule, ou pas, la couleur de l'objet touché...

mais : l'objet couvre une partie plus ou moins importante du pixel...

aliasing



anti-aliasing et filtrage

et alors ?

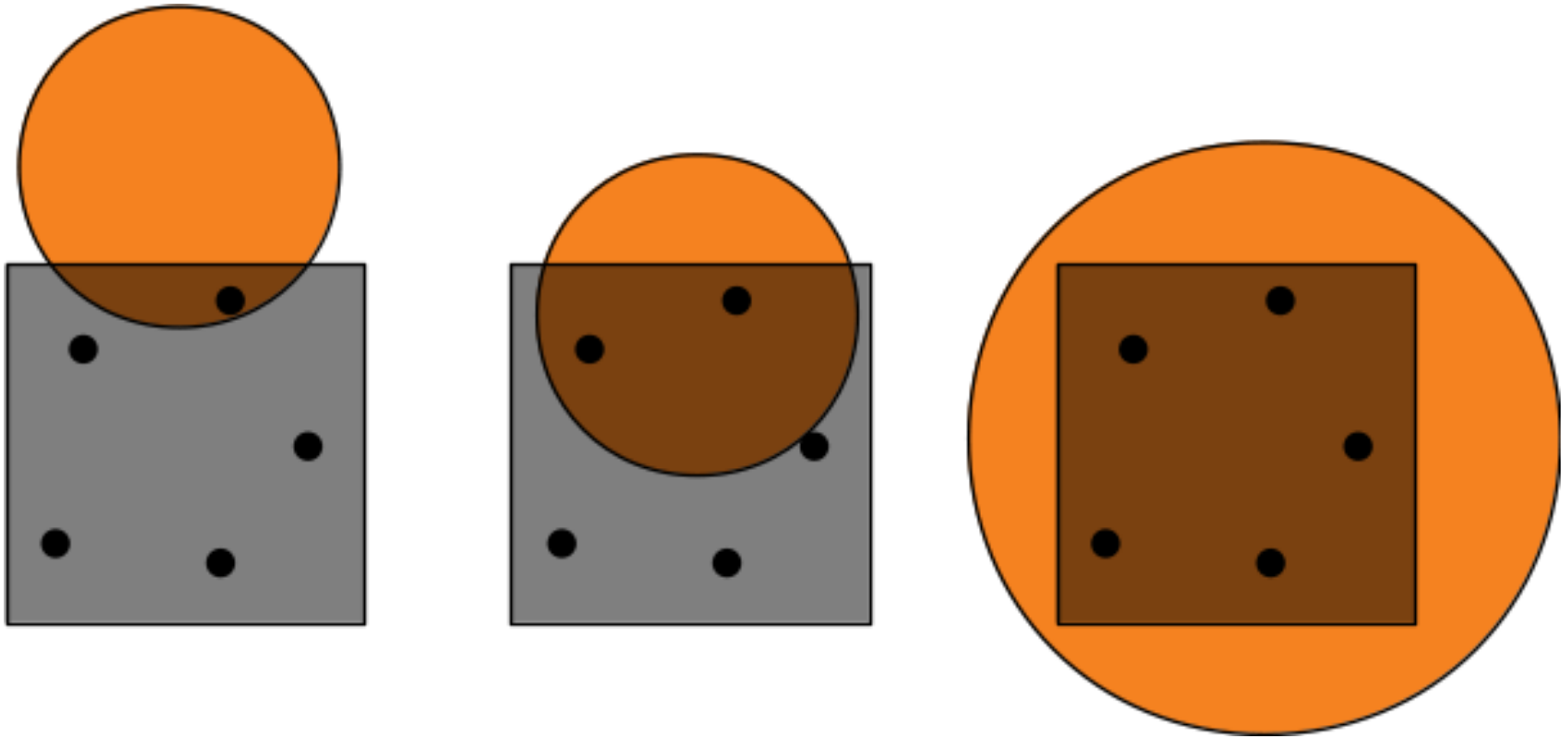
- on voudrait que la couleur du pixel dépende de l'objet,
- si l'objet couvre tout le pixel, c'est correct...
- mais, on voudrait la "bonne" proportion de la couleur du fond et de l'objet...

anti-aliasing et filtrage

2 solutions !

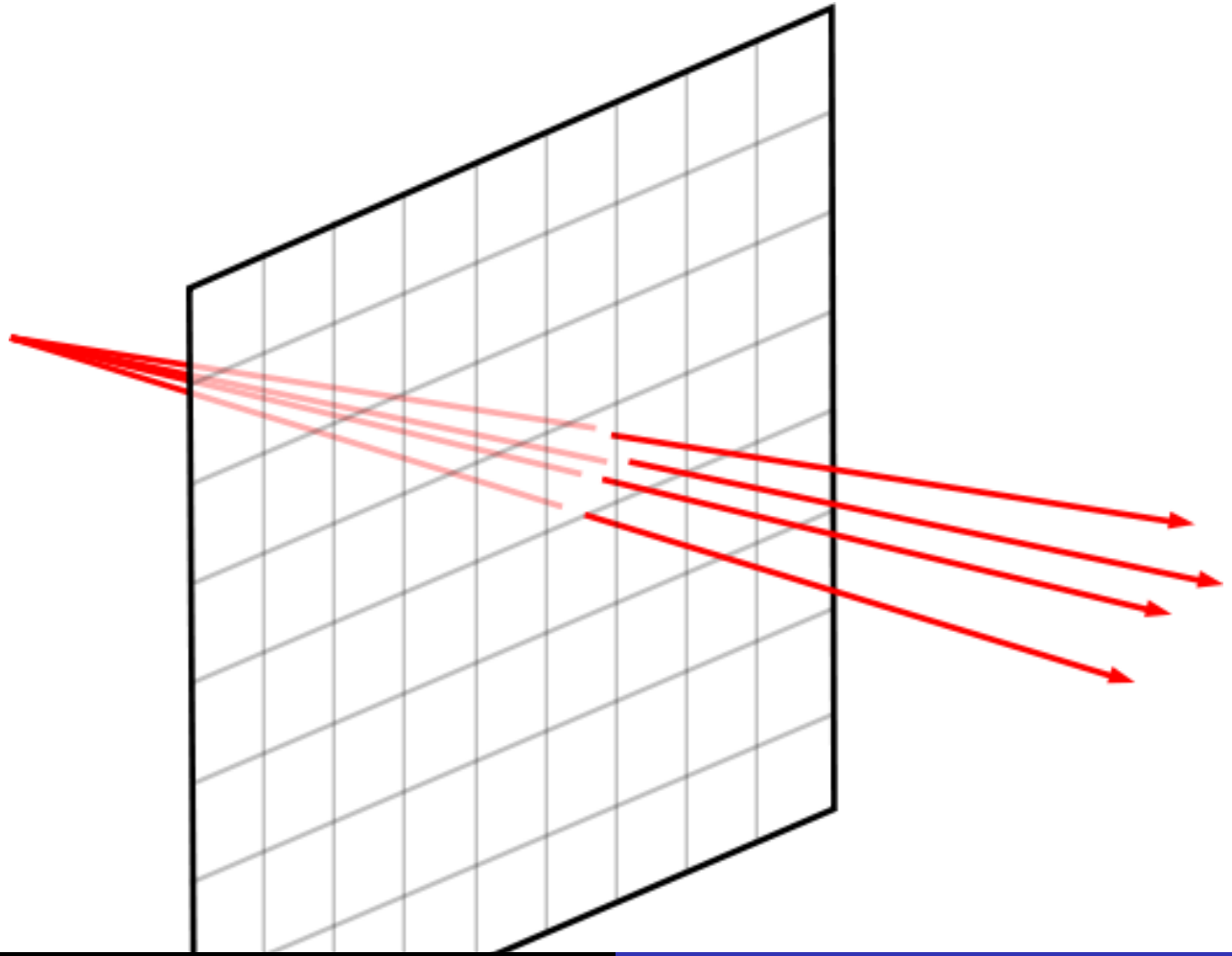
- on calcule une image plus grande, on la filtre,
  (pour éliminer les fréquences non représentables)
  et on la sous échantillonne pour obtenir la bonne résolution...
- ou, on construit plusieurs rayons par pixel et on moyenne les couleurs...

aliasing



aliasing

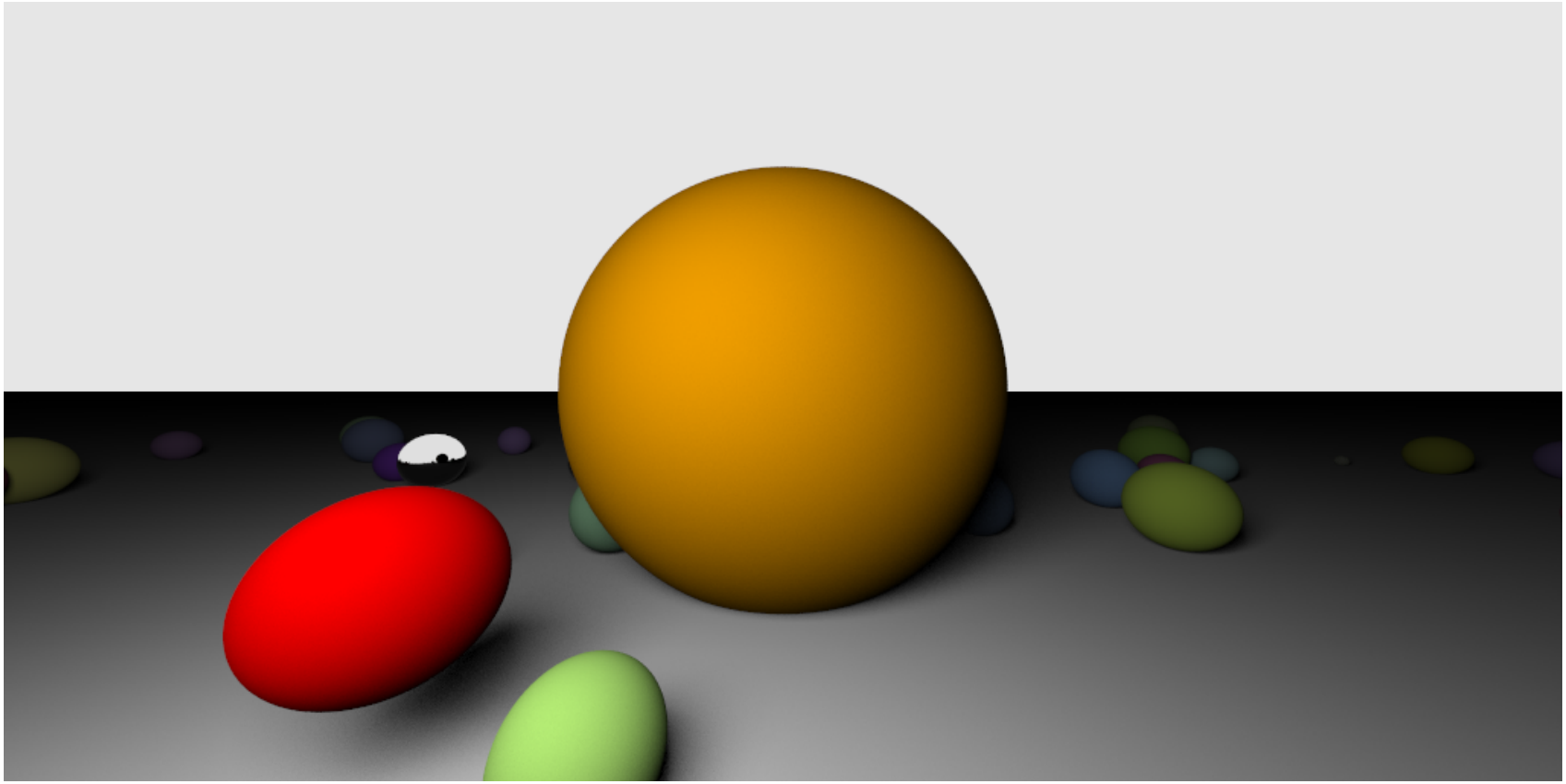
il faut générer plusieurs rayons dans chaque pixel...



comment ça se code ?

```
for(int py= 0; py < image.height(); py++)
for(int px= 0; px < image.width(); px++)
{
    // std::default_random_engine rng;
    // std::uniform_real_distribution<float> uniform;

    Color pixel;
    for(int pa= 0; pa < aa; pa++)
    {
        float ux= uniform( rng ); float uy= uniform( rng );

        // point (x y z) du plan image
        float x= float(px + ux) / float(image.width()) * 2 -1;
        float y= float(py + uy) / float(image.height()) * 2 -1;
        float z= -1;
        // droite (o e) passant par le pixel (px py)
        Point o= Point(0, 0, 0);
        Point e= Point(x, y, z);
        Vector d= Vector(o, e);
        ...
        pixel= pixel + { ... };
    }
    image(px, py)= Color(pixel / aa, 1);
}
```

anti-aliasing



bonus : profondeur de champ / dof

bizarre ?

- tous les objets sont nets dans l'image...
- même ceux qui sont loin de la camera,
- ou très près de la camera ?

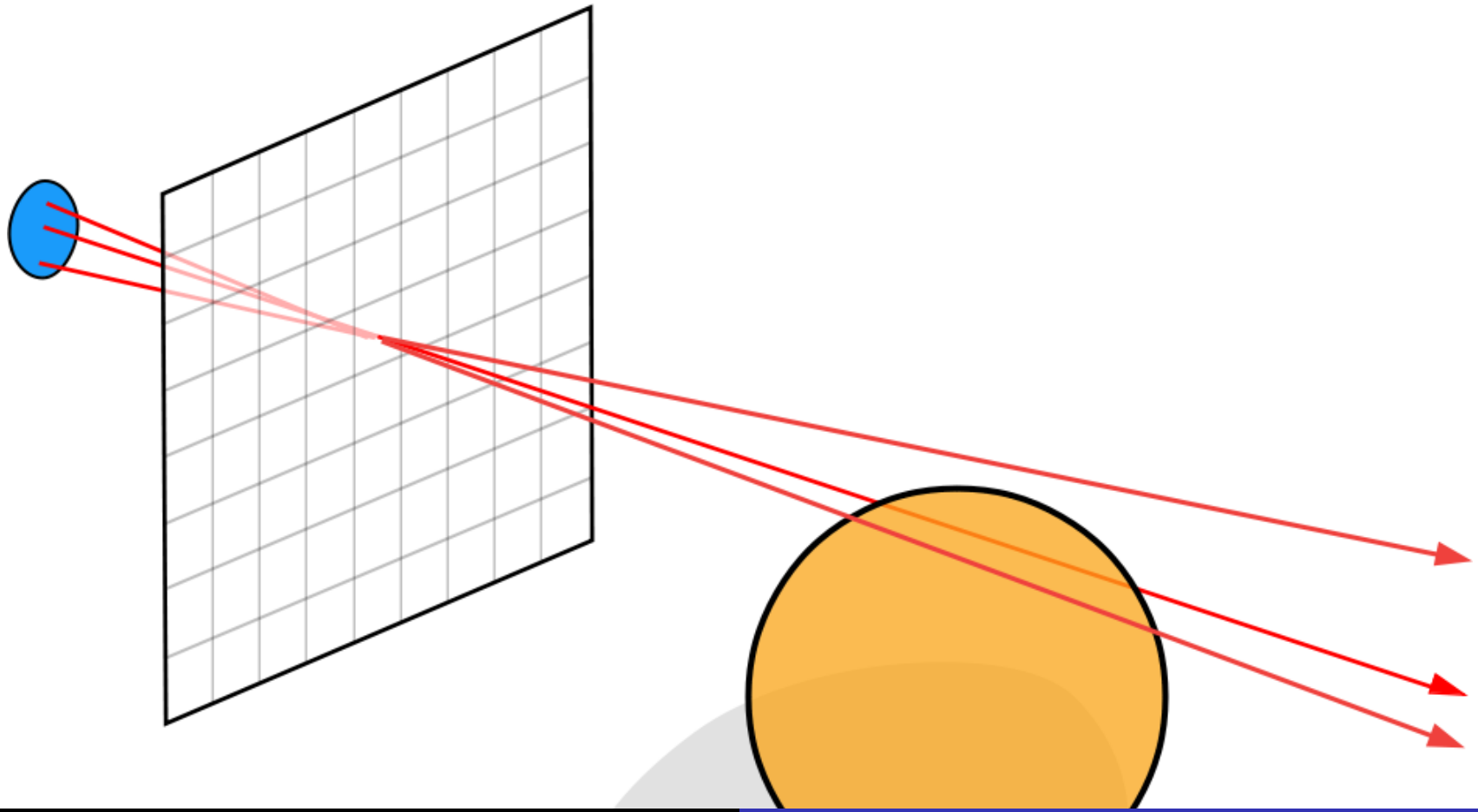
la camera est *très* simplifiée...

profondeur de champ / flou de profondeur

Monster U. Pixar, 2013

bonus : profondeur de champ / dof

pas difficile : tous les rayons ne se passent pas exactement par la camera.



bonus : profondeur de champ / dof

et on peut régler la distance ou les objets apparaissent "nets", il suffit de bouger le plan image !

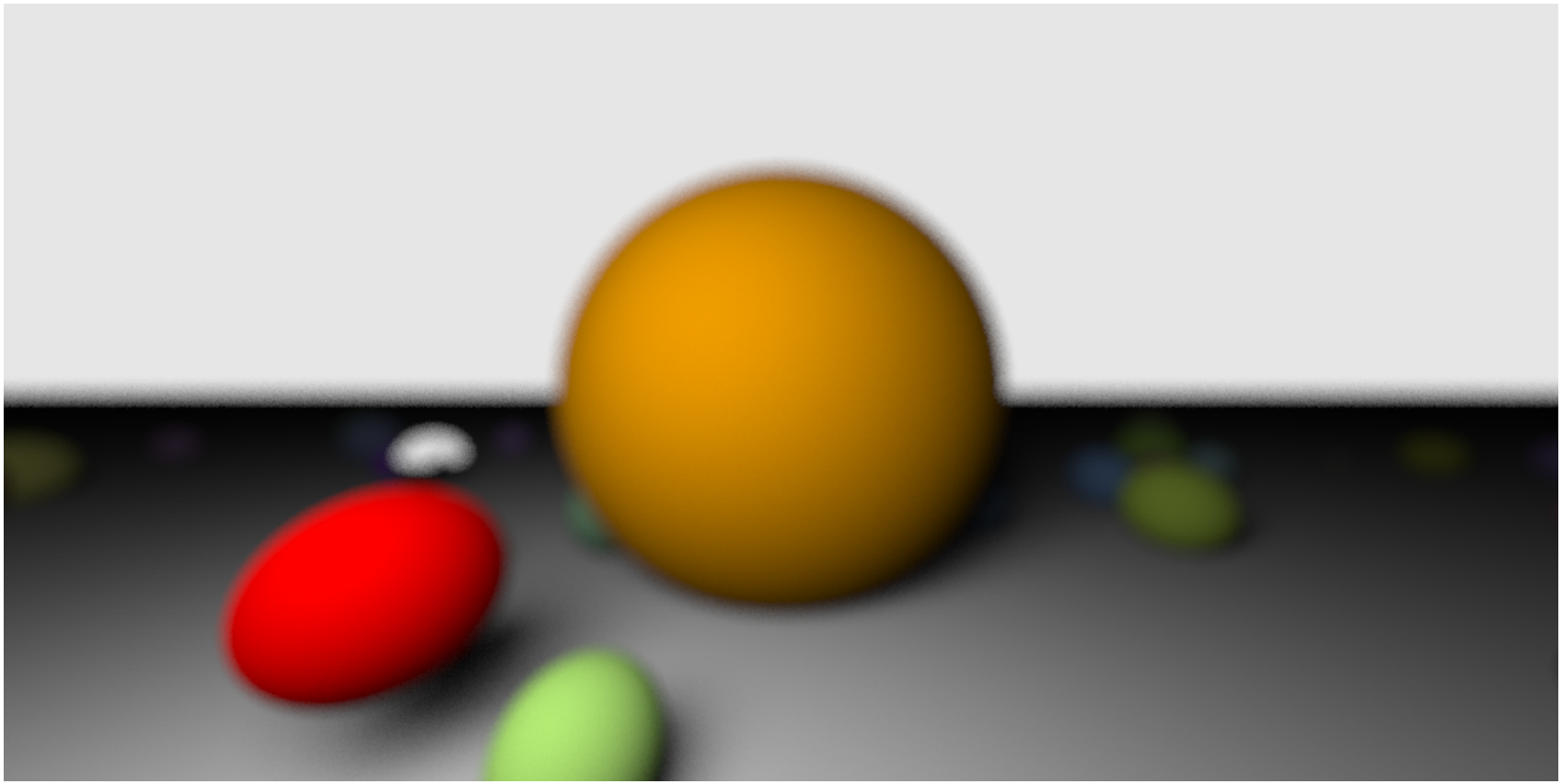
+ corriger les dimensions du plan image...

bonus : profondeur de champ / dof

l'origine des rayons se trouve dans un disque,
pas exactement à la position de la camera...

```
// std::default_random_engine rng;
// std::uniform_real_distribution<float> uniform;

// origine dans un disque de centre 0 et de rayon R
float r= std::sqrt( uniform( rng ) ) * R;
float phi= uniform( rng ) * float(2*M_PI);
Point o= Point(
    r * std::cos(phi),
    r * std::sin(phi),
    0
);

// droite (o e) passant par le pixel (px py)
Point e= { ... };   // comme d'habitude...
Vector d= Vector(o, e);
...
```

bonus : profondeur de champ / dof



bonus : profondeur de champ / dof