L3-Synthèse

Lancer de rayons et rendu

J.C. Iehl

January 24, 2023

résumé des épisodes précédents

calculer une image :

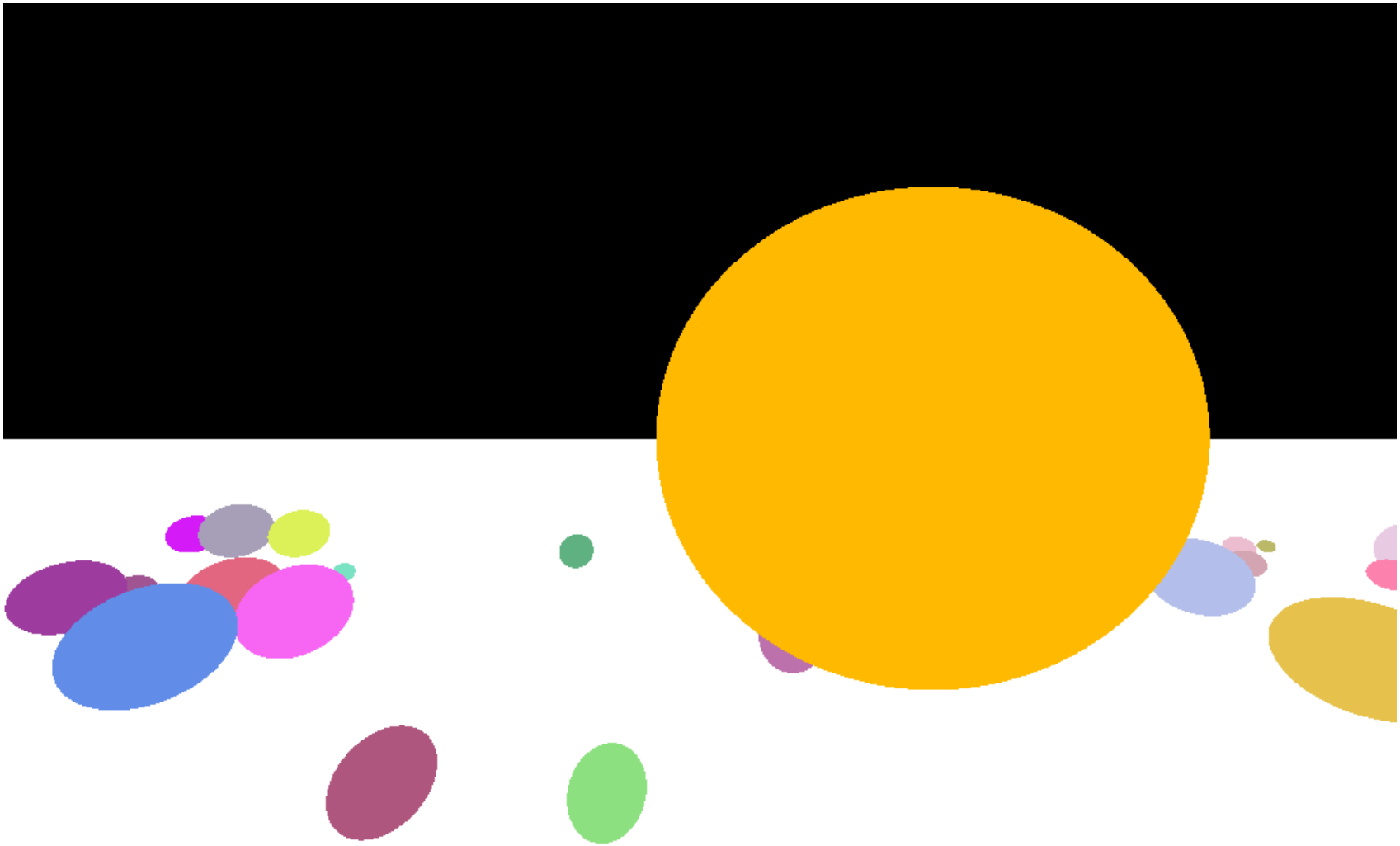
- trouver quel objet est visible pour chaque pixel,
- trouver comment l'objet est éclairé,
- calculer sa couleur...

résumé des épisodes précédents

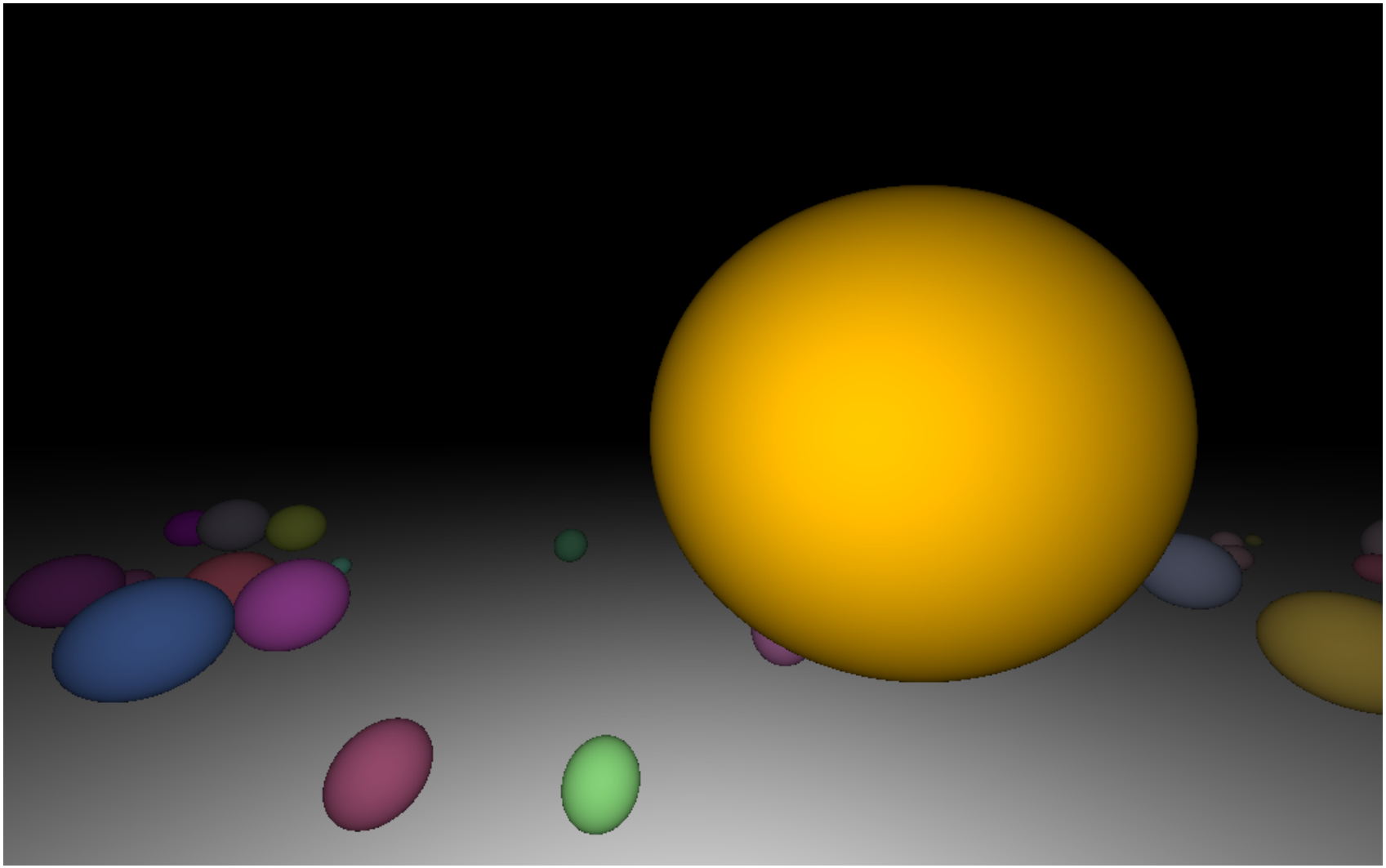
trouver l'objet visible :

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

résumé des épisodes précédents



couleur de l'objet ?

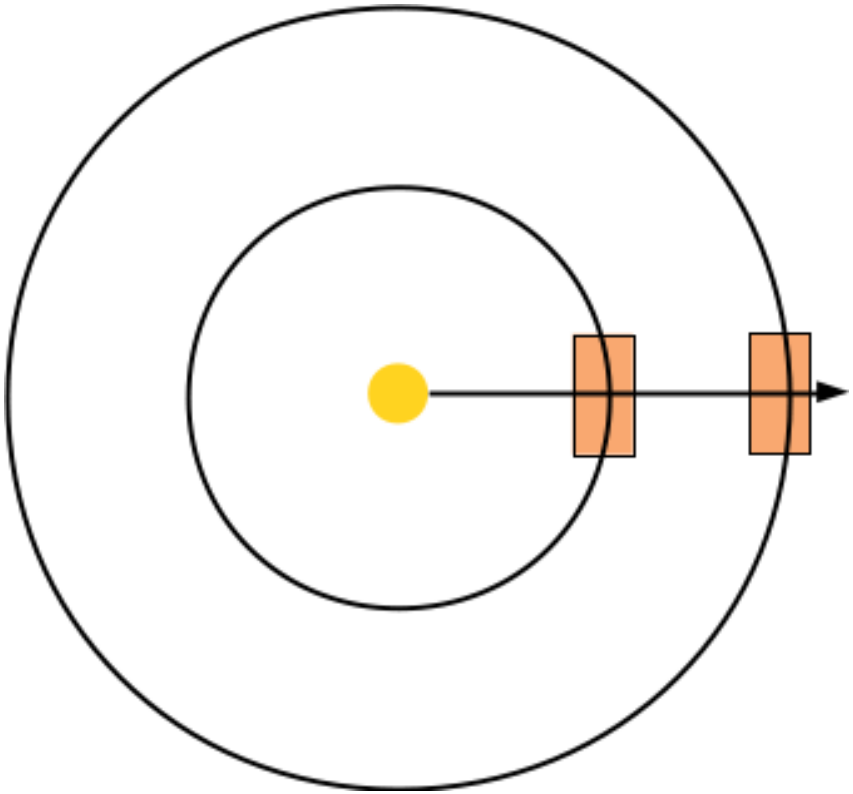


et pendant ce temps...

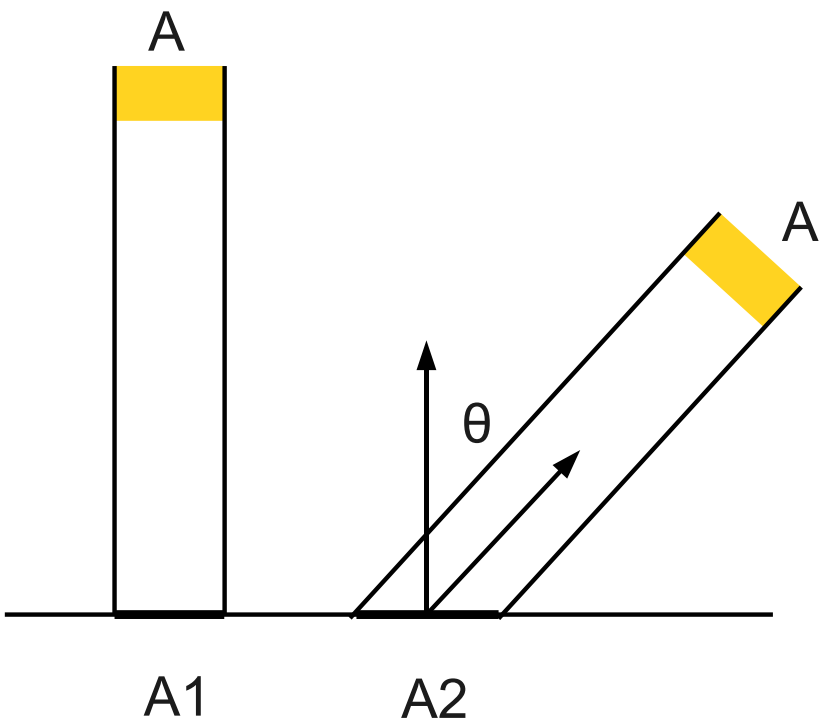
Lambert en 1760 :

- 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...



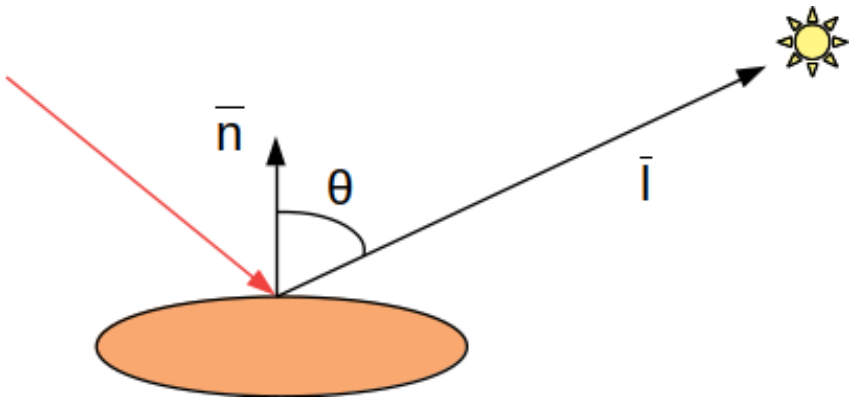
orienté vers la lumière...



lumière incidente

lumière incidente :

- $L_i = L_e \times \frac{1}{||\vec{l}||^2} \cos\theta$



lumière réfléchie

lumière réfléchie :

- $L_r = L_i \times f_r = f_r \times L_e \frac{1}{||\vec{l}||^2} \cos\theta$

f_r fonction de réflectance : comment la matière réfléchit la lumière.

lumière réfléchie

lumière réfléchie :

- pour une matière diffuse, f_r est une constante, ou "la couleur" de l'objet...
- $L_r = L_i \times f_r = color \times L_e \frac{1}{||\vec{l}||^2} \cos\theta$

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

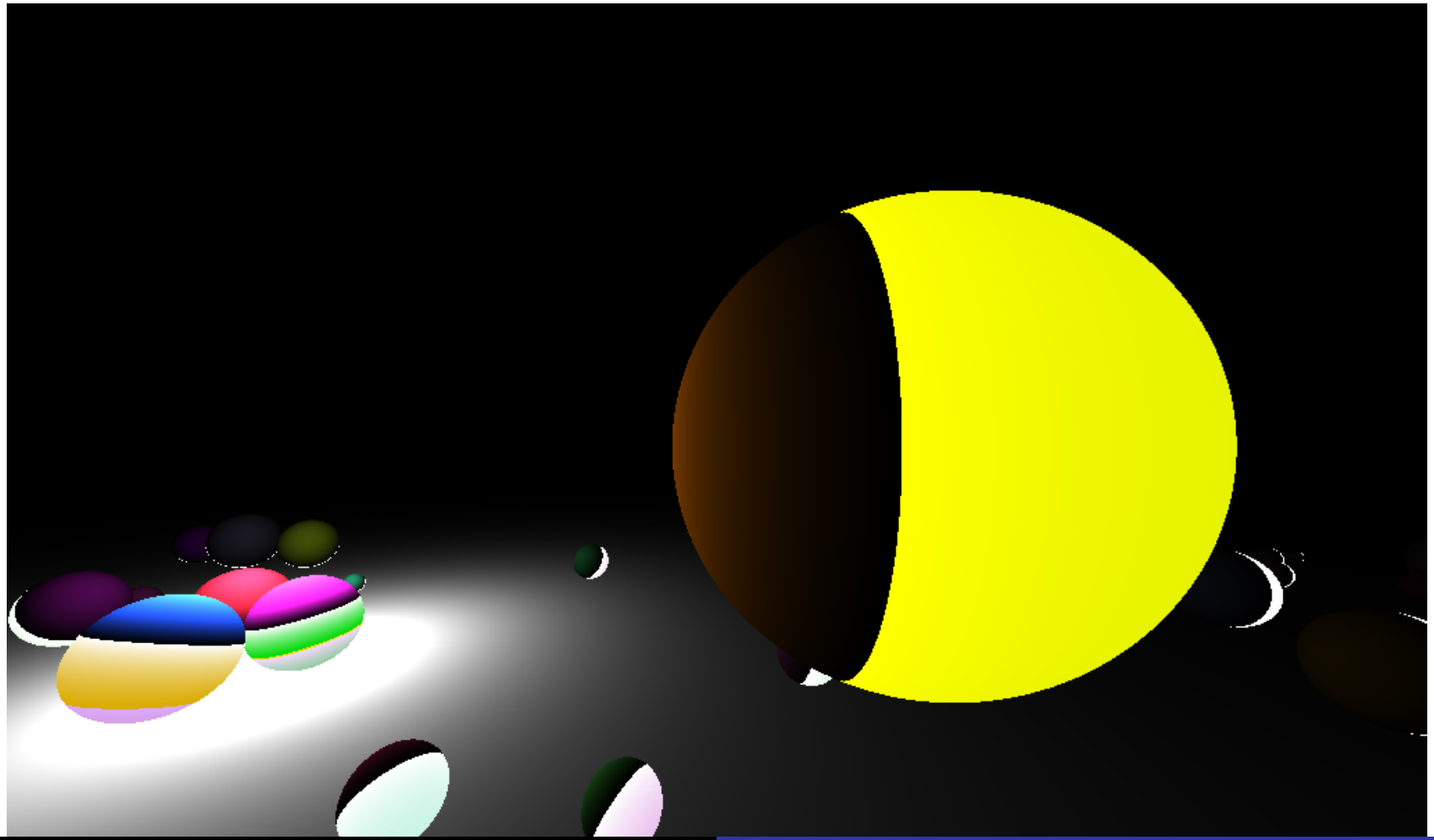
comment ça se code ?

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"


Point source= { ... };     // position source de lumière
Color emission= { ... };   // lumière emise par la source

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

Vector l= Vector(p, source);
float cos_theta= dot(normalize(n), normalize(l));

Color pixel=  emission * color * cos_theta / length2(l);
```

lumière incidente

euh ?

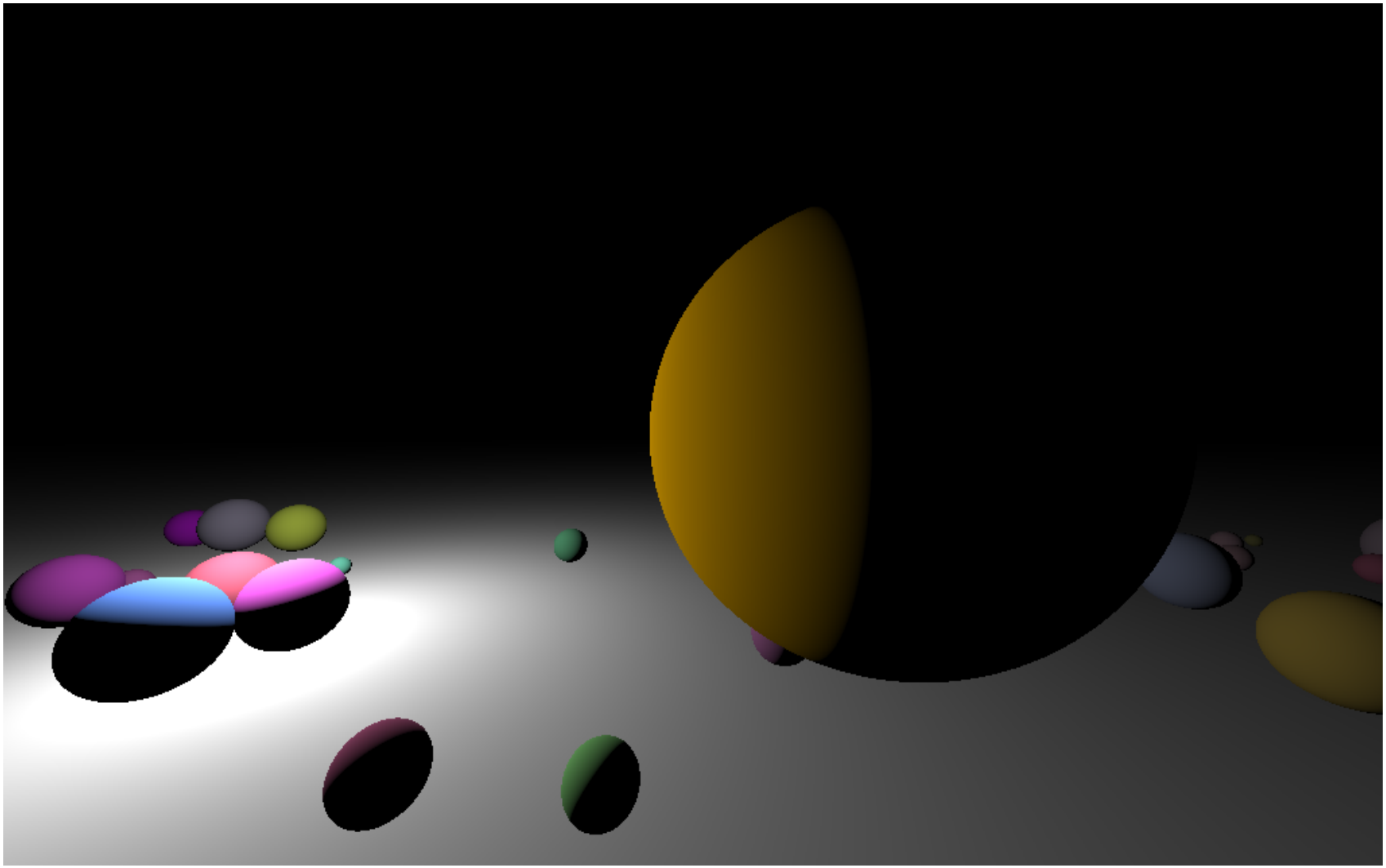


ah !

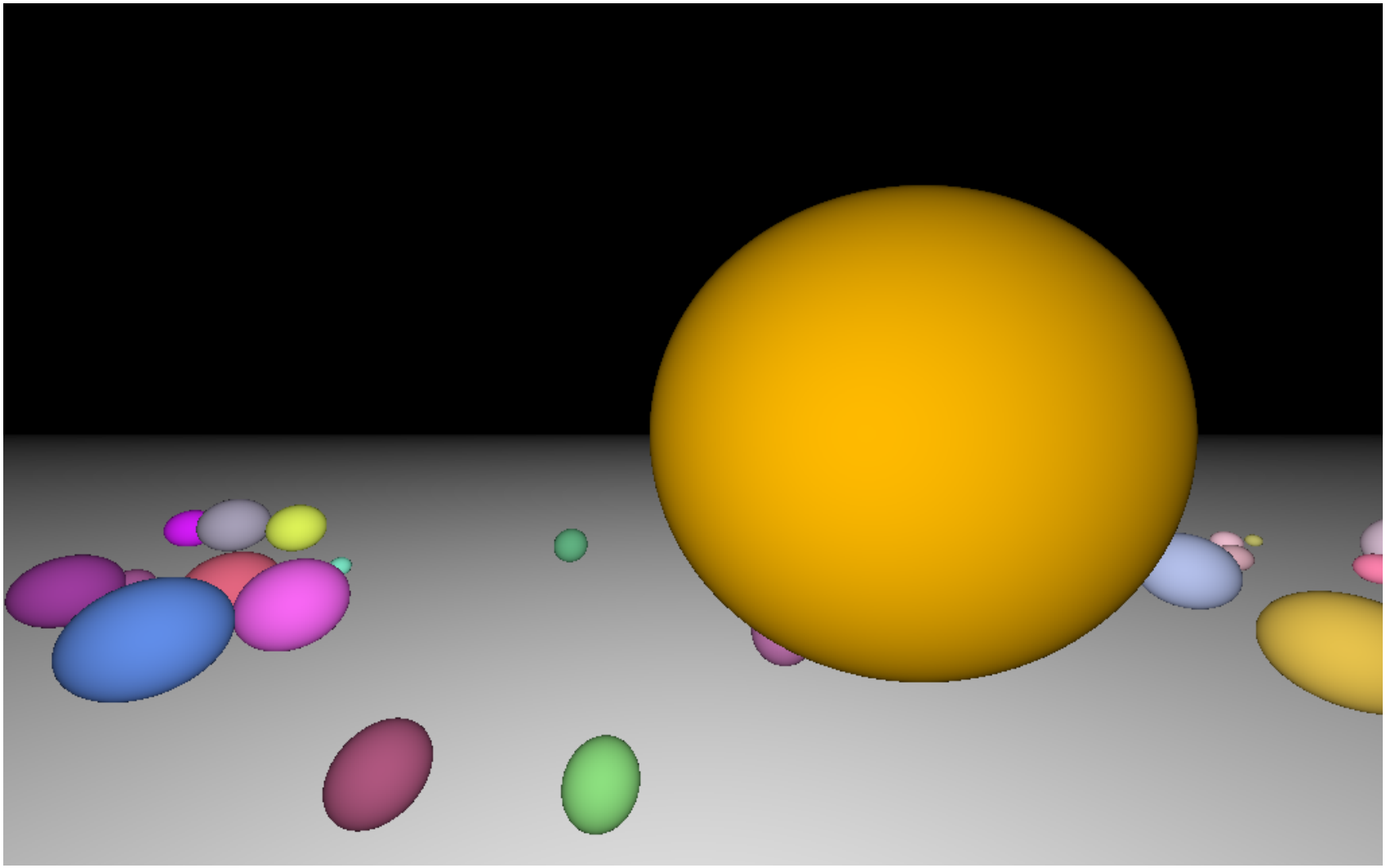
mais attention, $\cos\theta$ ne doit pas etre < 0

```
#include <algorithm>

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


Point source= { ... };     // position source de lumière
Color emission= { ... };   // lumière emise par la source

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

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

Color pixel=  emission * color * cos_theta / length2(l);
```

ah !



ah !



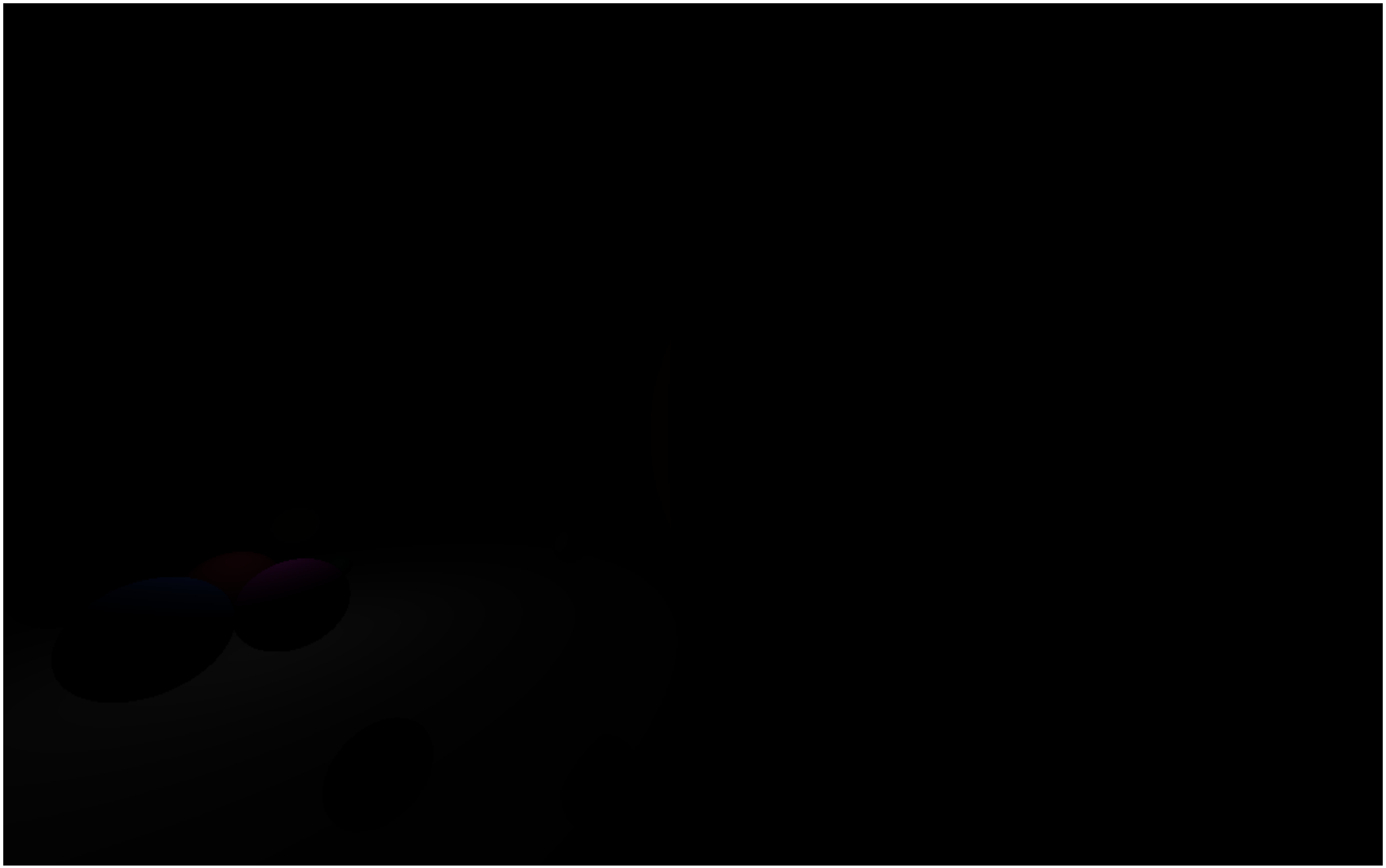
emission, réflectance et couleurs

couleurs ?

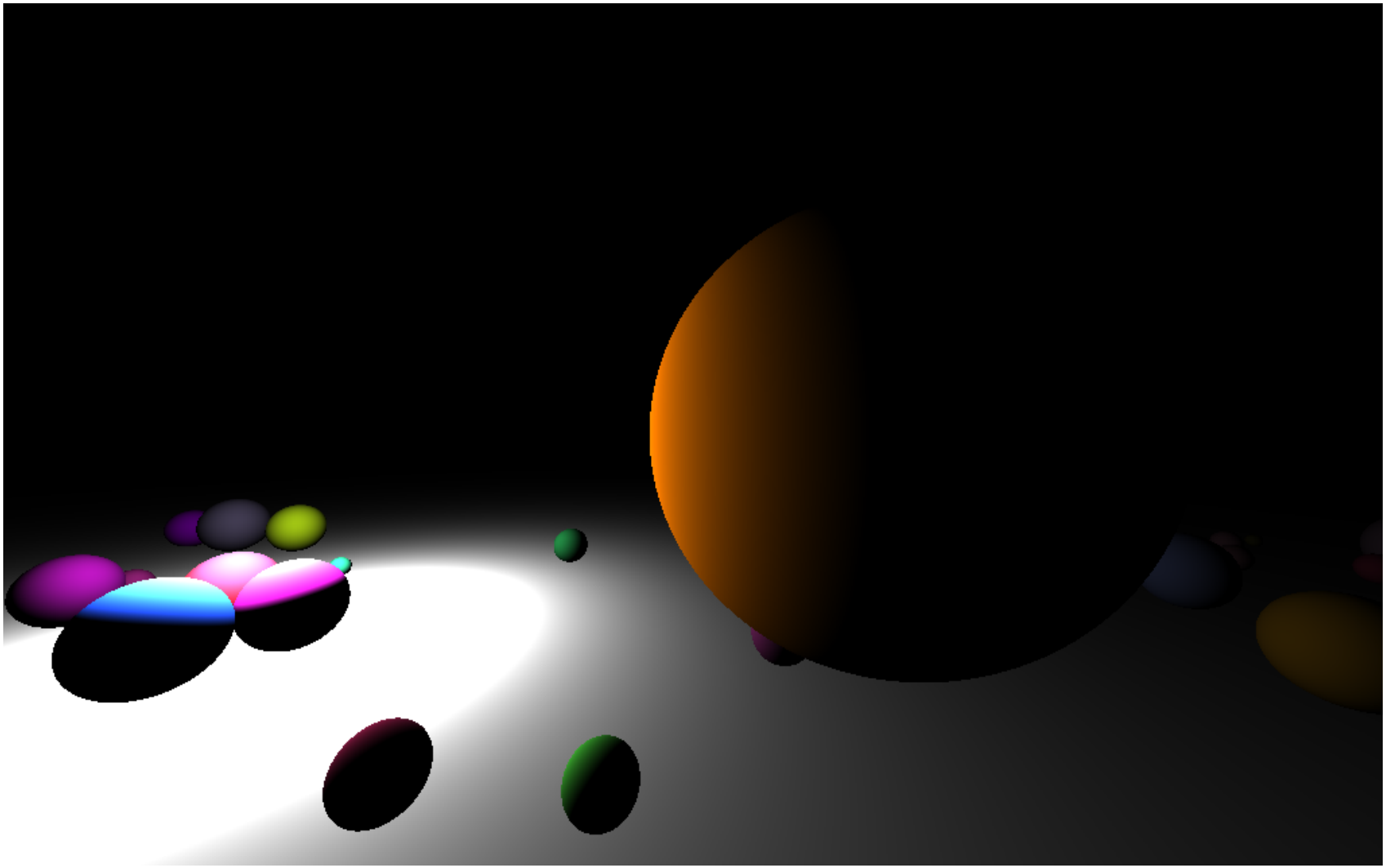
- quelles valeurs ?
- entre 0 et 1 ?
- ou ça dépend de la distance aux objets ??

on peut essayer plusieurs valeurs...

emission = 1



emission = 100

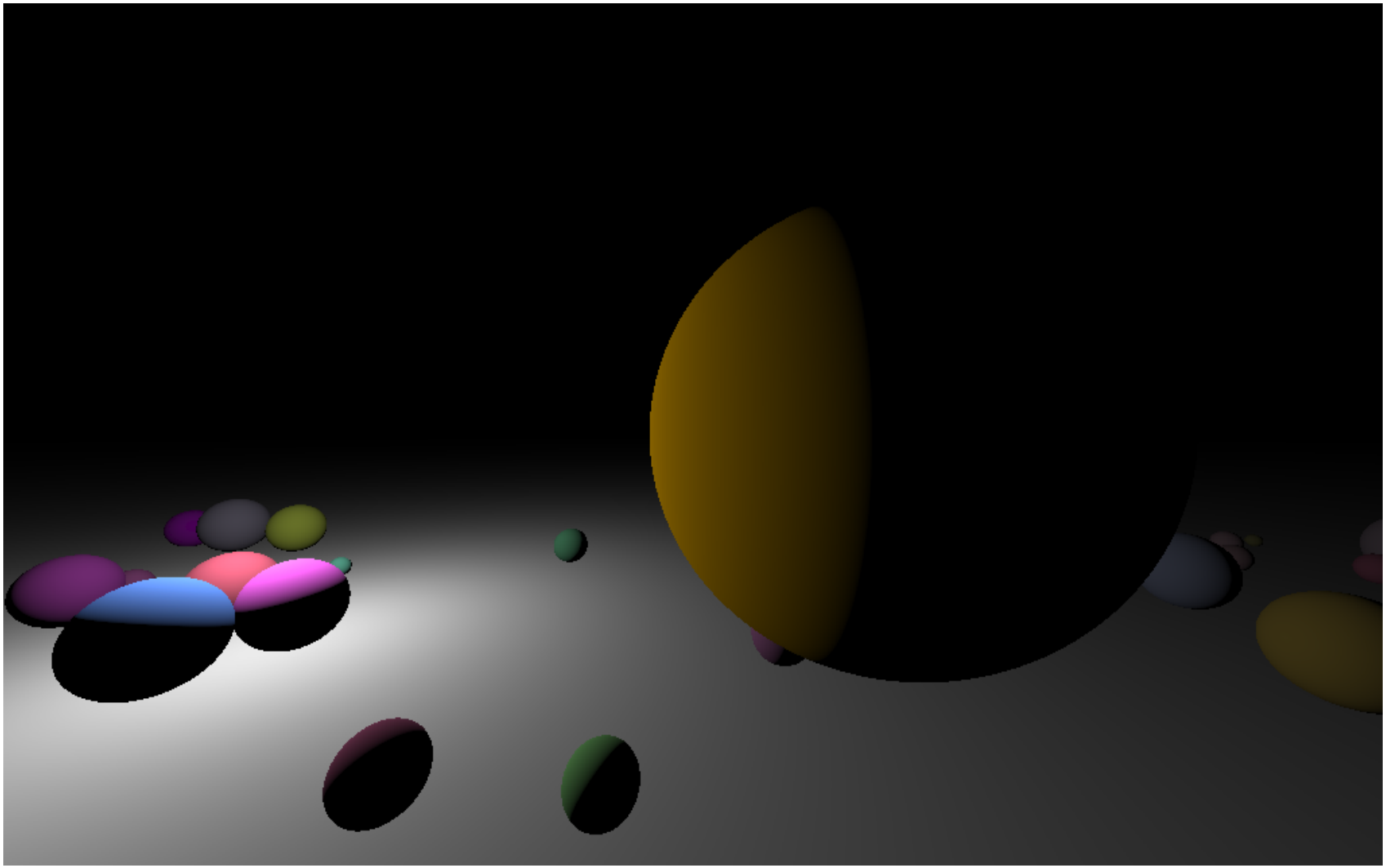


emission = 1...

on peut aussi enregistrer l'image dans un format hdr...
et utiliser image_viewer

```
#include "color.h"
#include "image.h"
#include "image_hdr.h"

int main( )
{
    Image image(1024, 640);

    for(int py= 0; py < image.height(); py++)
    for(int px= 0; px < image.width(); px++)
    {
        Color pixel;
        // trouver l'objet visible pour le pixel
        // trouver comment il est eclaire
        // calculer sa couleur
        image(px, py)= pixel;
    }

    write_image_hdr(image, "image.hdr");
    return 0;
}
```

emission = 1 + image_viewer



ombres et lumières

ombres ?

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

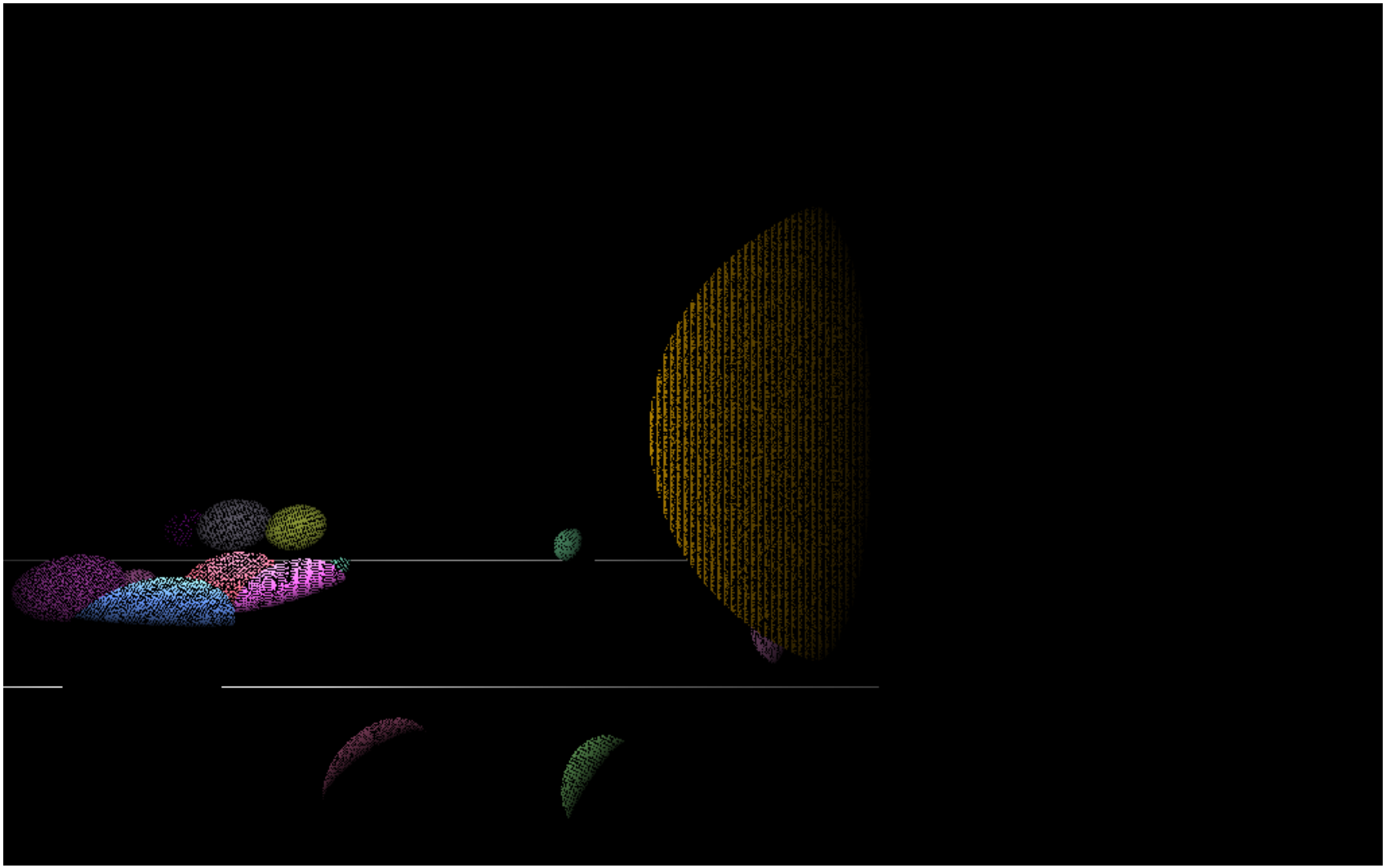
il suffit de construire un nouveau rayon et de calculer les intersections avec les objets...

ombres et lumières

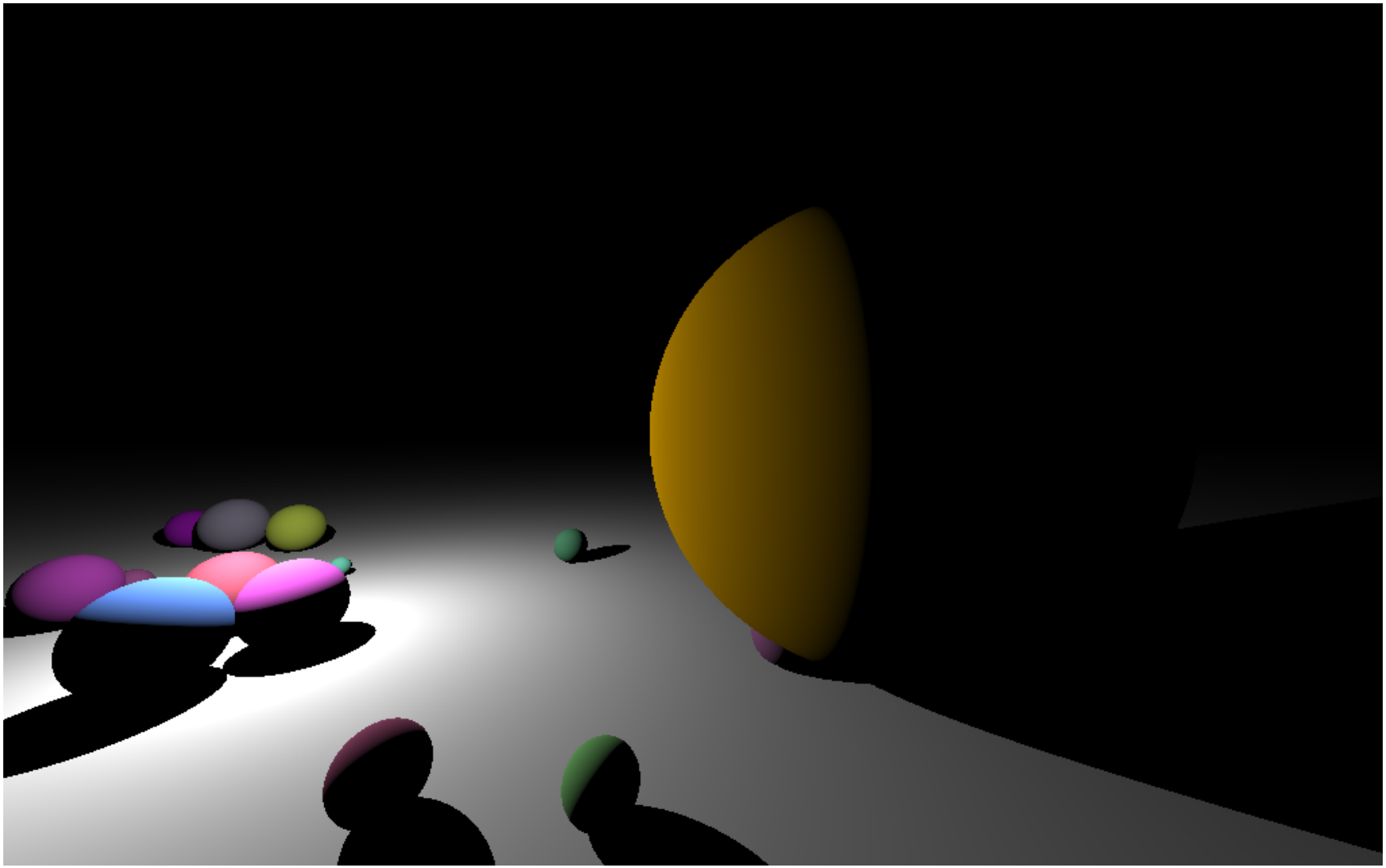
rayon d'ombre ?

- origine : le point d'intersection,
- direction : vers la source de lumière,
- $+$ calculer les intersections entre le point et la source...
- le point sera à l'ombre si on trouve une intersection valide : $t > 0$ et $t < 1$,
- si le point est à l'ombre, le pixel sera noir !

ombres et lumières



c'est mieux, non ?

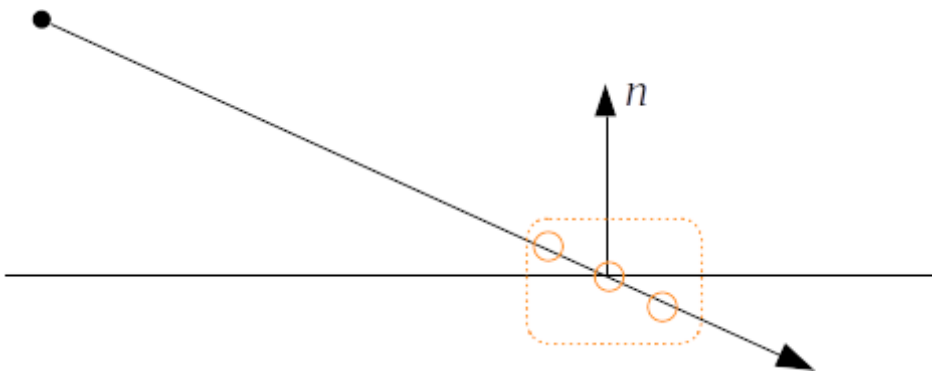


ombres et lumières

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

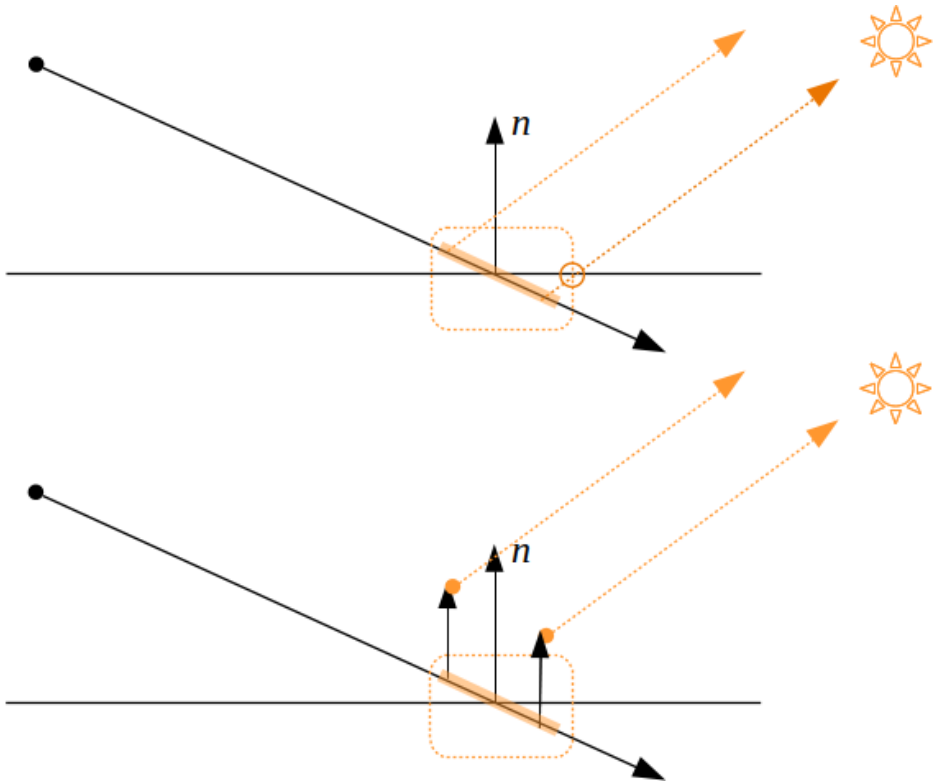


ombres et lumières

et alors ?

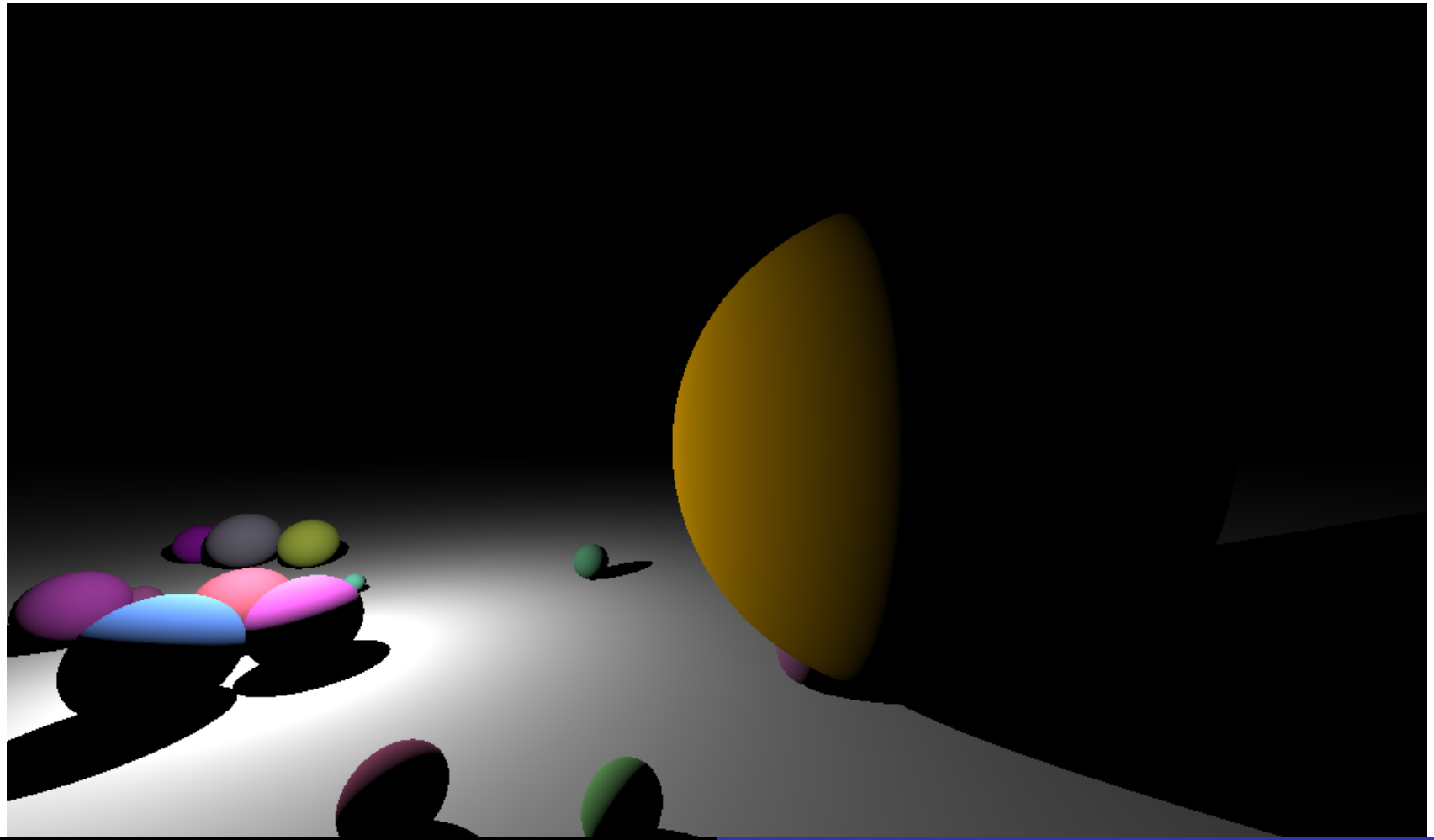
- il faut décoller l'origine du rayon de la surface,
- le long de la normale :
- $o = p + \epsilon \cdot \vec{n}$

ombres et lumières



c'est mieux, non ?

$\epsilon = 0.001$



et les pénombres ?

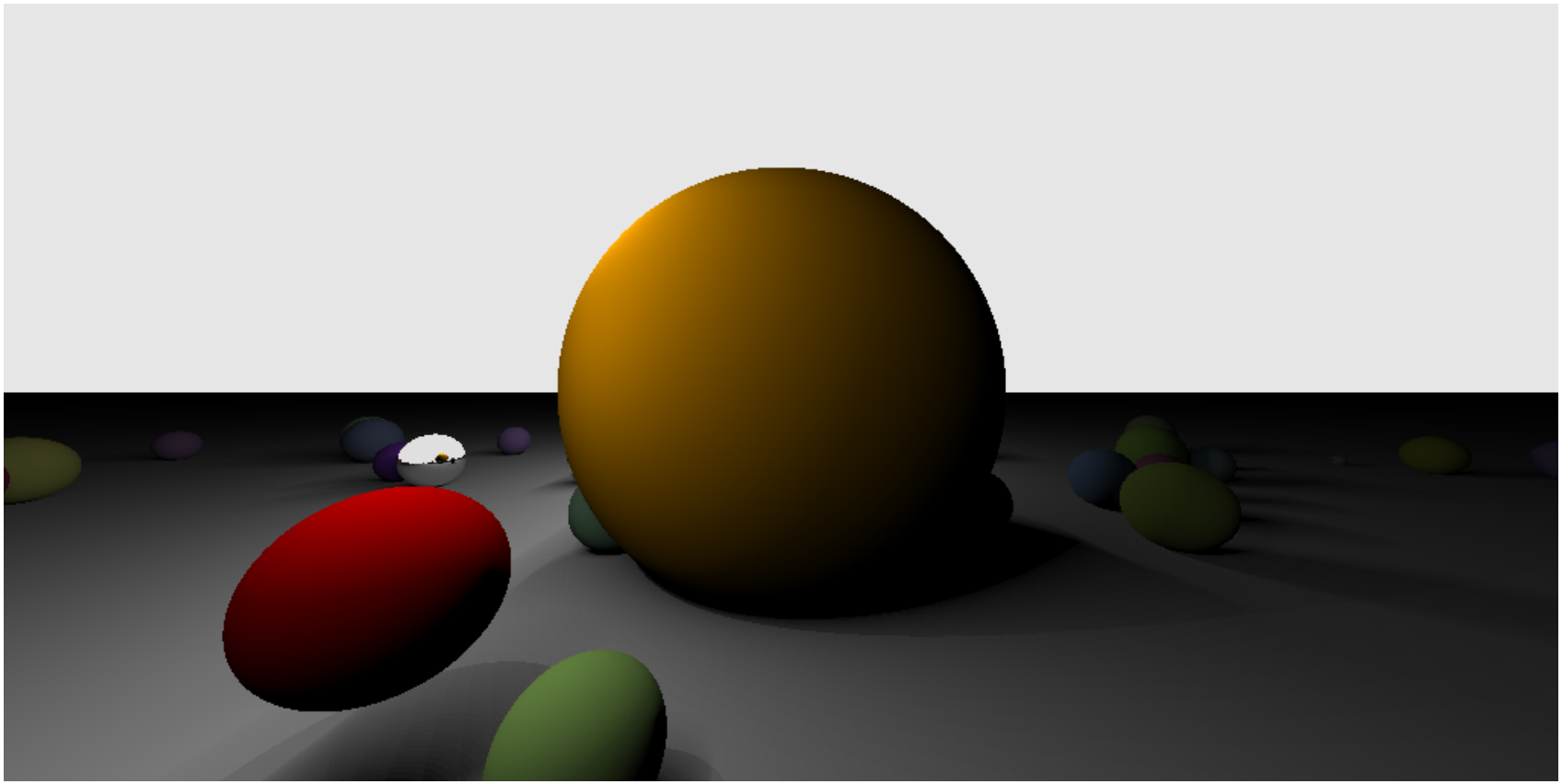
pénombres ?

- il suffit de créer plein de sources de lumière,
- disposées sur une ligne,
- ou disposées sur un carré,
- ou ...

mais attention aux temps de calculs...

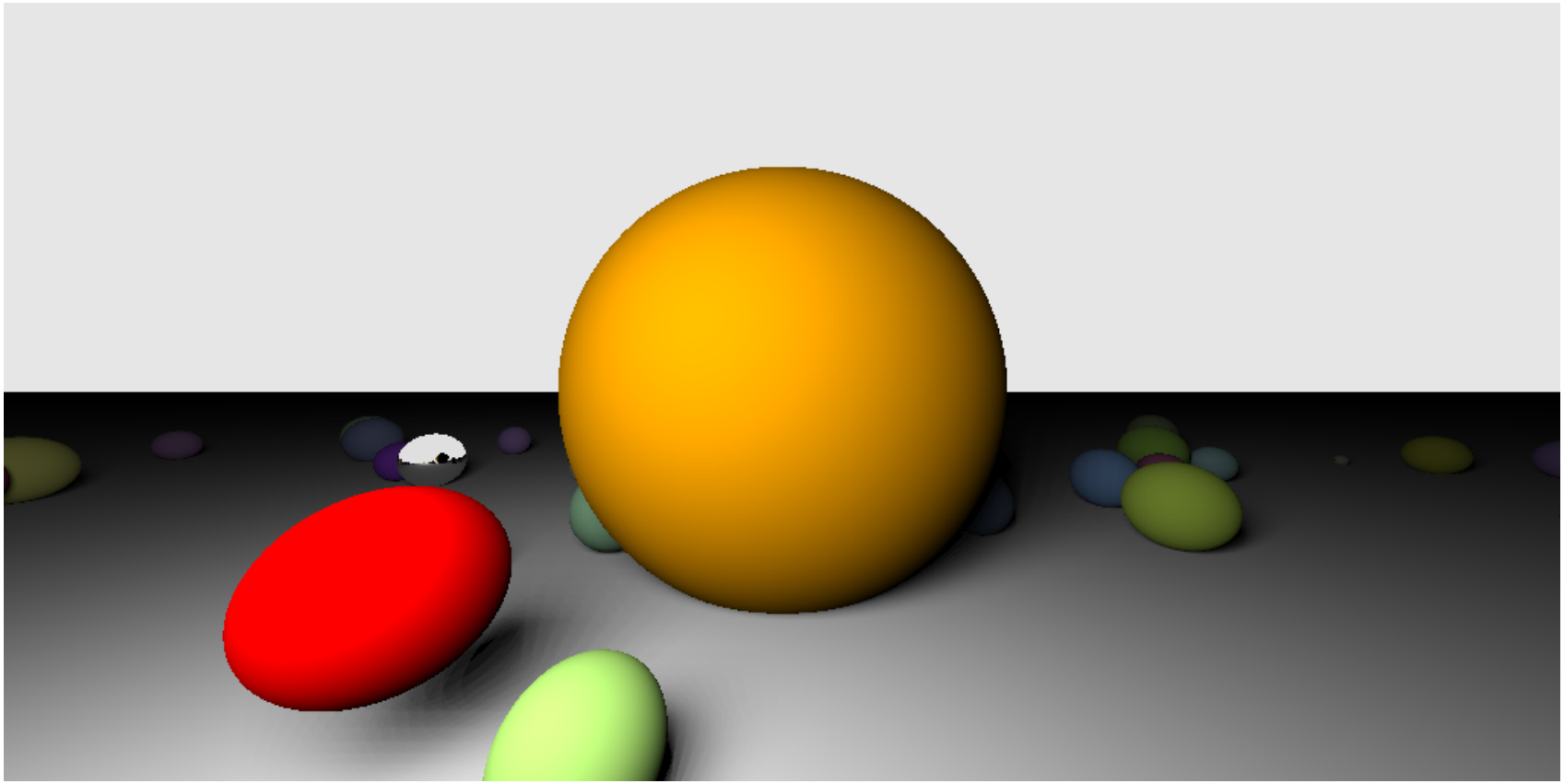
ombres et pénombres...

sur une ligne...



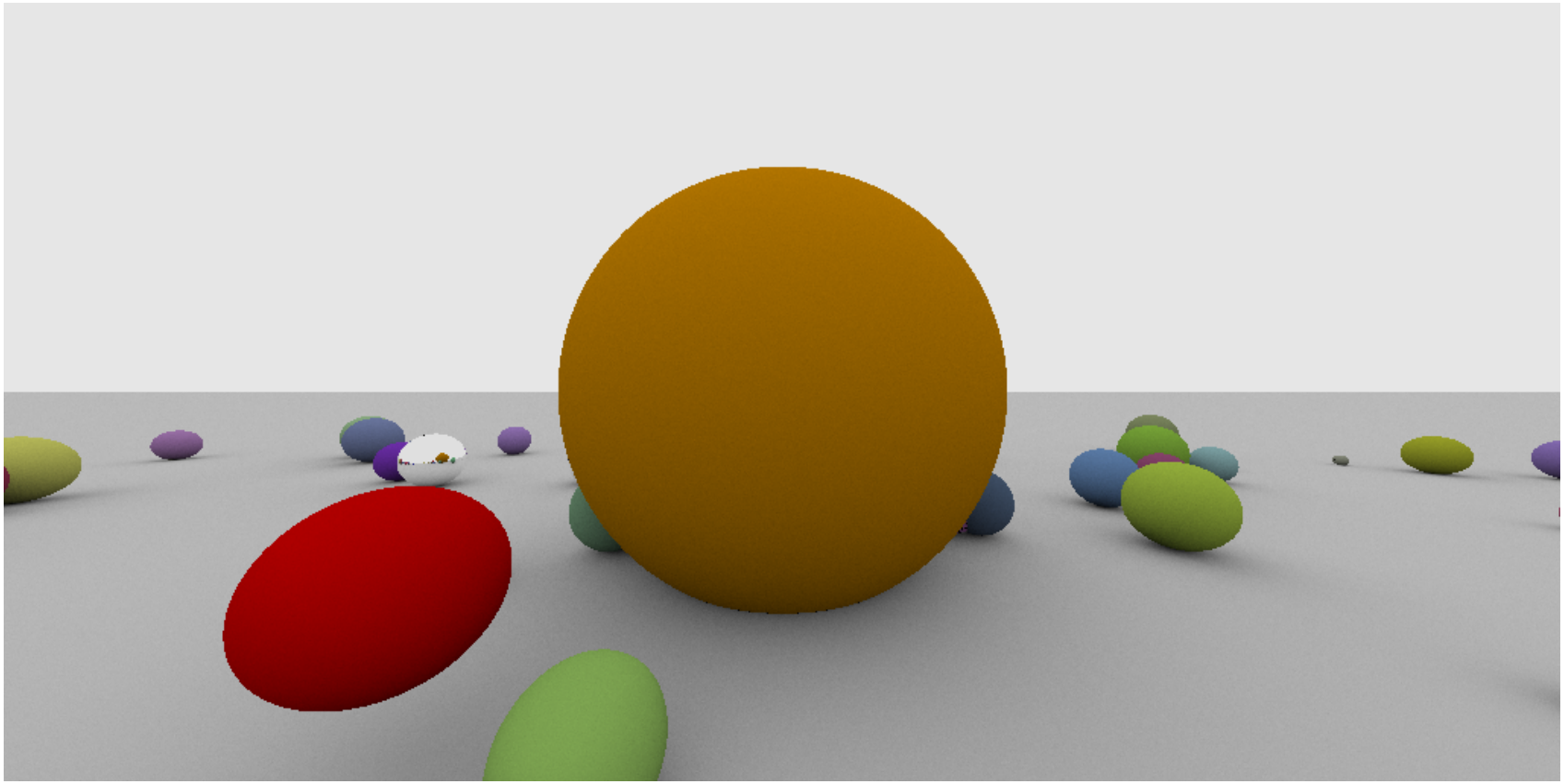
ombres et pénombres...

dans un carré...



ombres et pénombres...

dans le ciel...



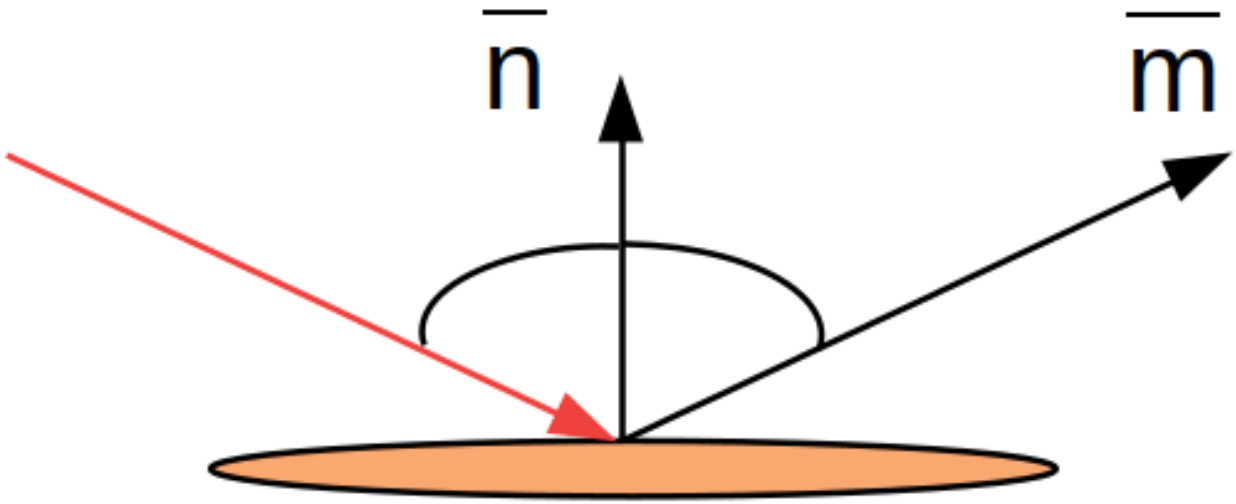
et pendant ce temps ?

Fresnel en 1820 :

- un miroir réfléchit la lumière,
- dans la direction symétrique à la normale...

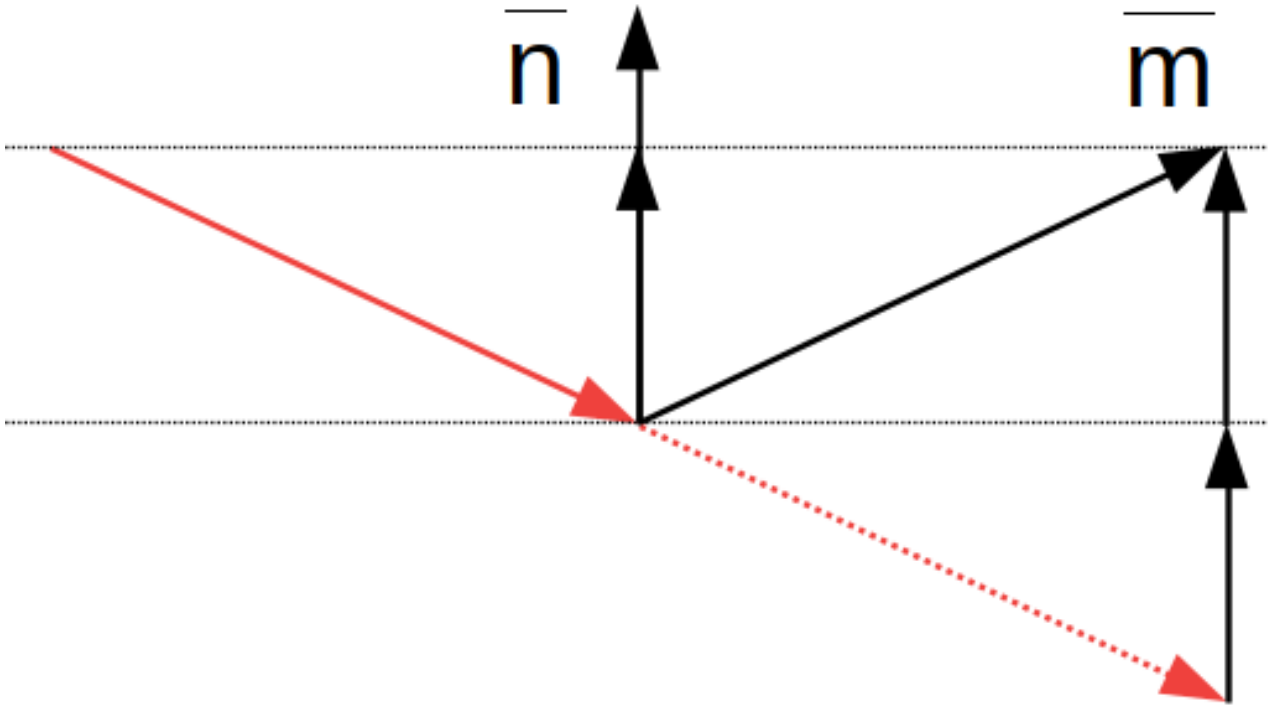
comment construire cette direction ?

direction miroir



direction miroir

$$\vec{m} = \vec{v} - 2(\vec{n} \cdot \vec{v}) \cdot \vec{n}$$



comment ça se code ?

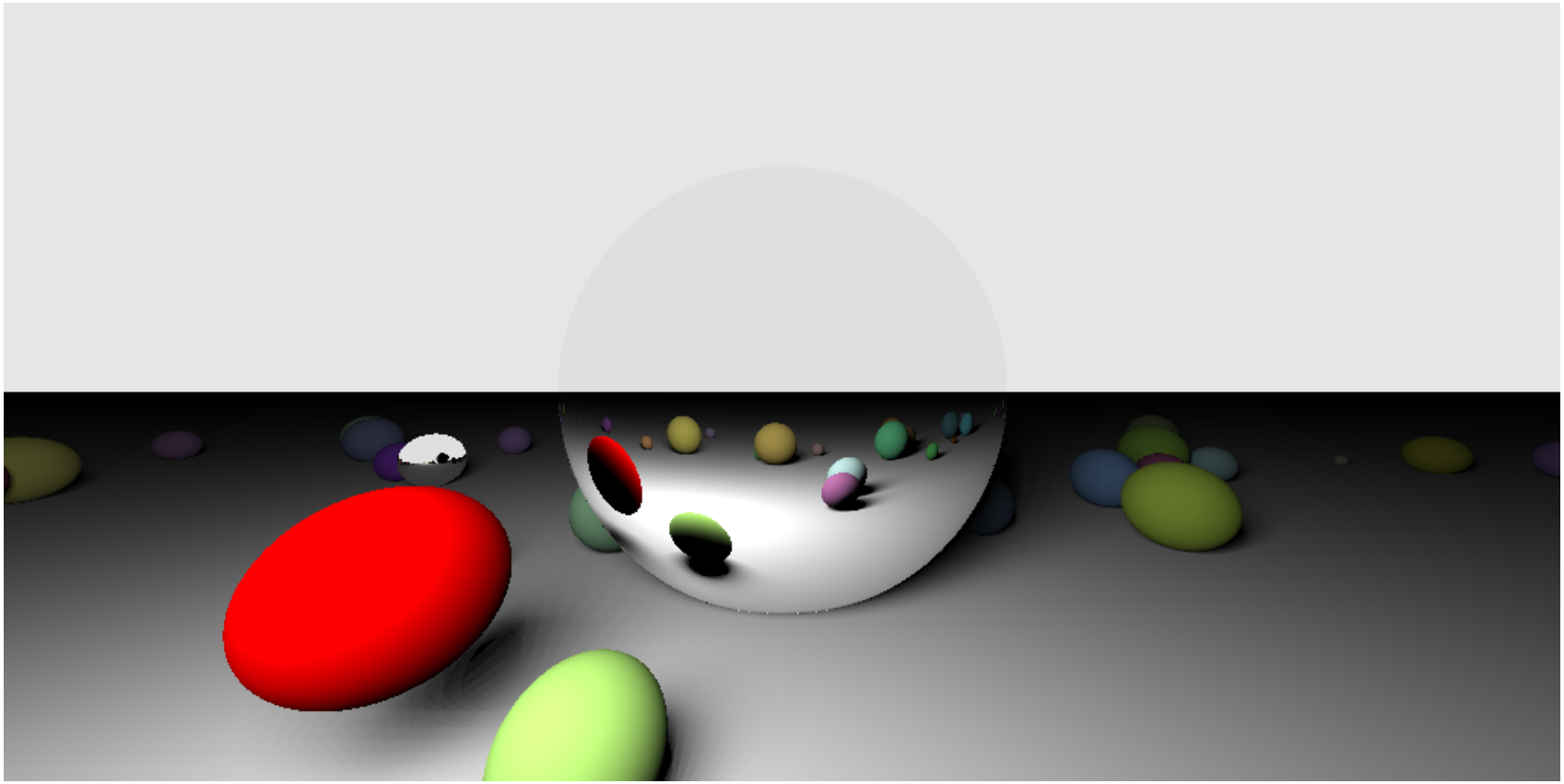
```
// direction miroir de v (par rapport a n)
Vector reflect( const Vector& n, const Vector& v )
{
    assert(dot(n, v) < 0);
    return v - 2*dot(n, v) * n;
}
```

miroir

quelle couleur ?

- un miroir réfléchit la couleur de l'objet visible dans la direction miroir...
- construire un nouveau rayon pour trouver cet objet :
- origine : $p + \epsilon\vec{n}$,
- direction : $\vec{m}$
- calculer sa couleur,
- et hop, on connait la couleur du miroir....

miroir...



miroir, miroir...

reste un petit détail :

- c'est un peu plus compliqué...
- en fonction de l'angle entre $\vec{n}$ et $\vec{v}$,
- la quantité de lumière réfléchie change...

$$F(\vec{n},\vec{v}) = \frac{1}{2}\left(\frac{g-c}{g+c}\right)^2\left(1+\left(\frac{c(g+c)-1}{c(g-c)+1}\right)^2\right)$$

$$g^2 = \eta^2 + c^2 - 1 \text{ ou } g = \sqrt{\eta^2 + c^2 - 1}$$

$$c = \cos\theta = cos\angle(\vec{n},\vec{v})$$

et η est l'indice de réfraction de la matière...

qui est la plus belle ?

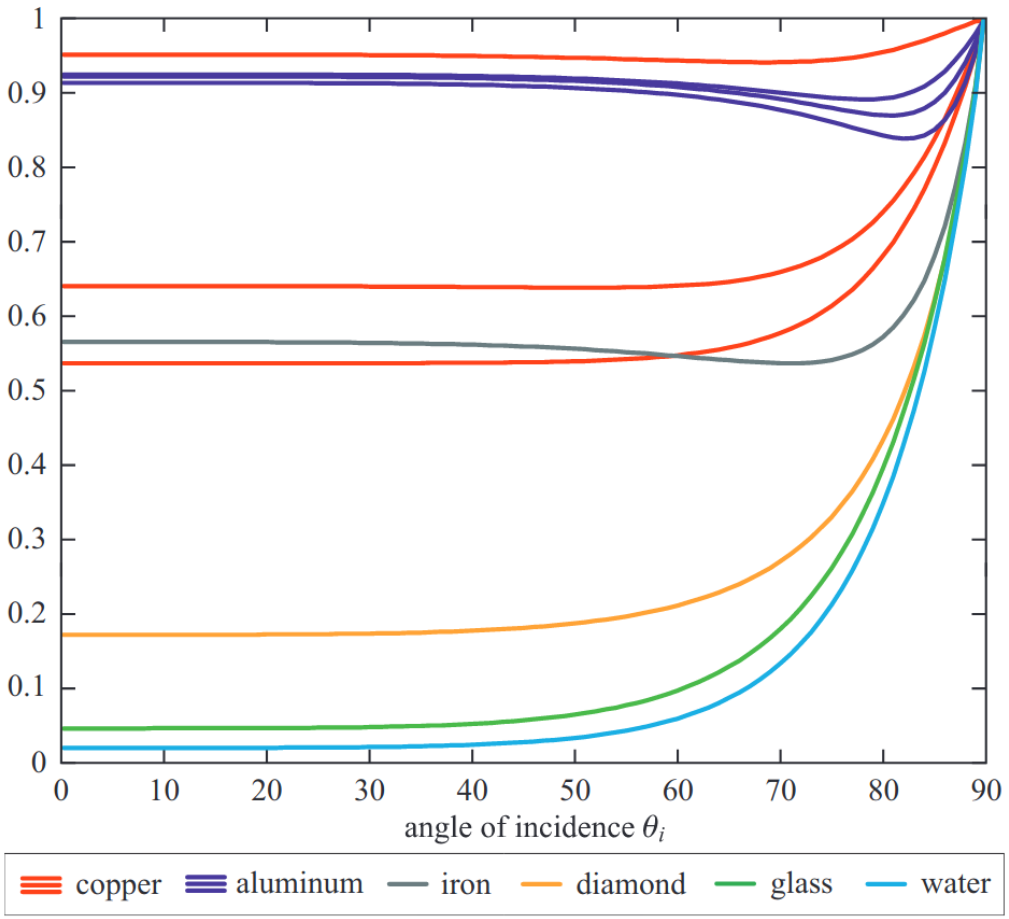


Figure 24: Fresnel reflectance for external reflection from a variety of substances. Since copper and aluminum have significant variation in their reflectance over the visible spectrum, their reflectance is shown as three separate curves for R, G, and B. Copper's R curve is highest, followed by G, and finally

qui est la plus belle ?

et sinon, il y a plus simple ?

- oui !
- et plus pratique à manipuler...
- approximation "Schlick Fresnel" :
- $F(\vec{n}, \vec{v}) = F0 + (1 - F0)(1 - c)^5$,
- et $F0$ représente la quantité de lumière réfléchie pour $\theta = 0$

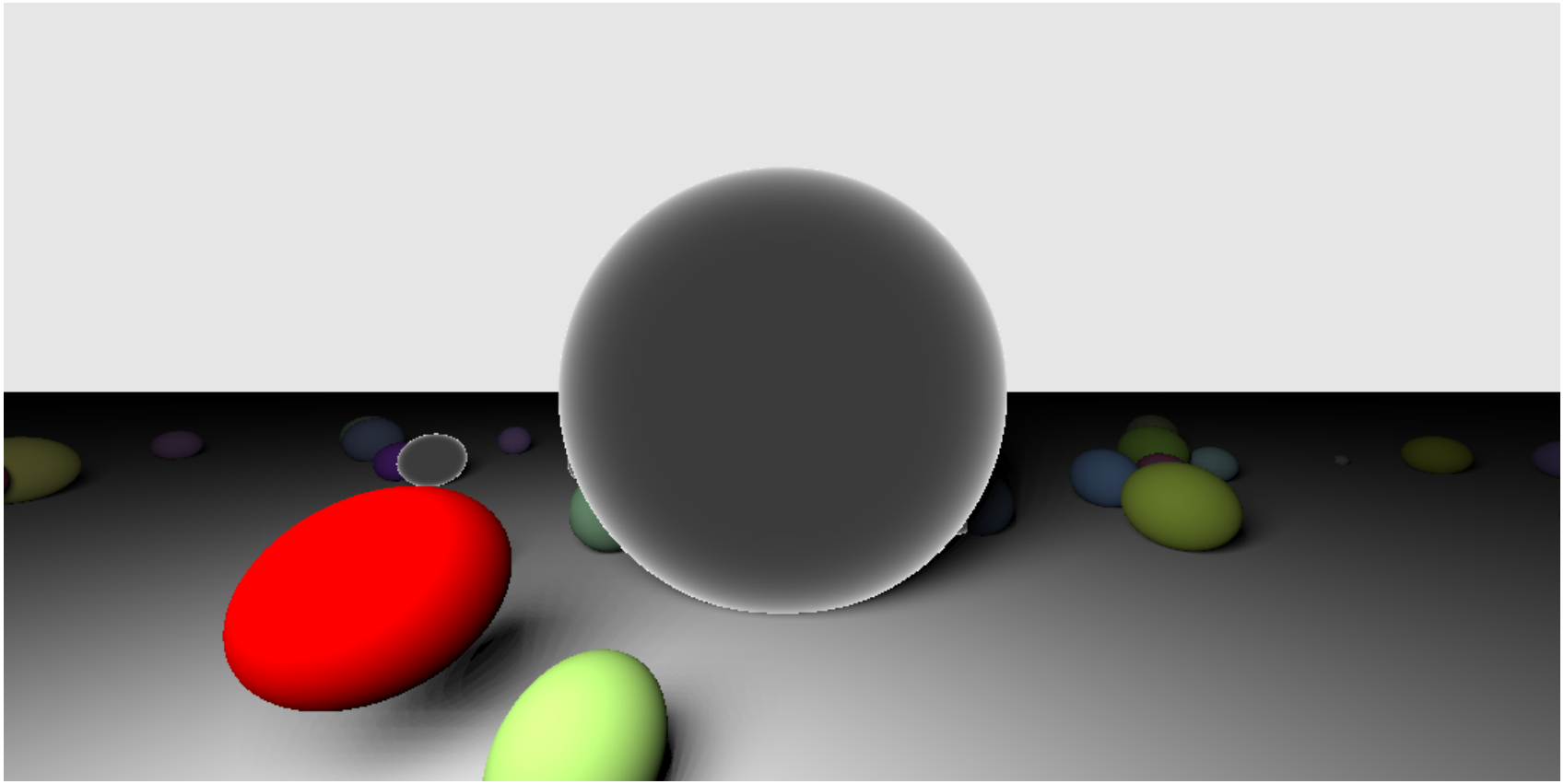
cf "An Inexpensive BRDF Model for Physically-based Rendering", C. Schlick, 94

qui est la plus belle ?

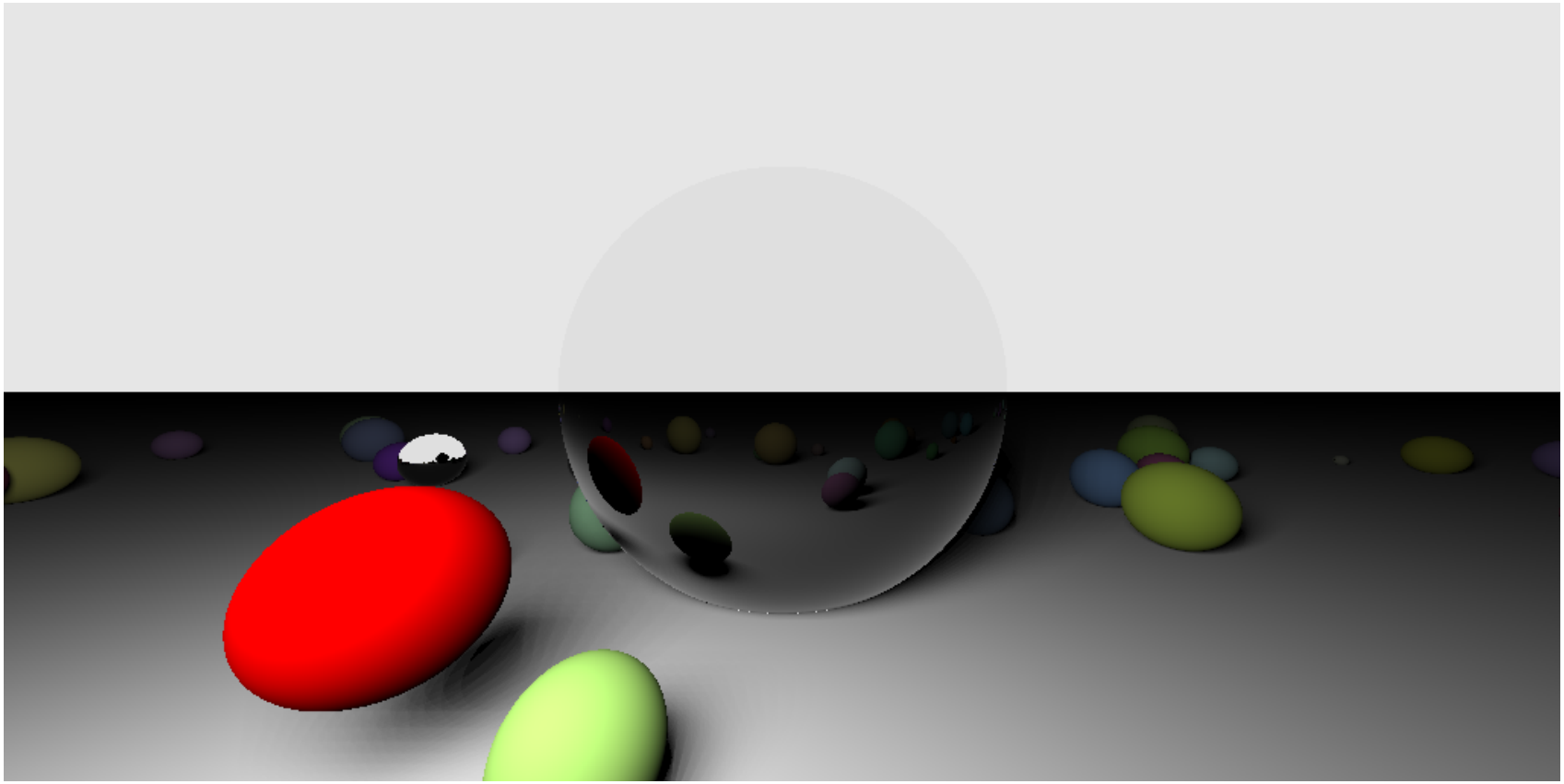
Material	$F(0°)$ **(Linear)**	$F(0°)$ **(sRGB)**	**Color**
Water	0.02,0.02,0.02	0.15,0.15,0.15	
Plastic / Glass (Low)	0.03,0.03,0.03	0.21,0.21,0.21	
Plastic High	0.05,0.05,0.05	0.24,0.24,0.24	
Glass (High) / Ruby	0.08,0.08,0.08	0.31,0.31,0.31	
Diamond	0.17,0.17,0.17	0.45,0.45,0.45	
Iron	0.56,0.57,0.58	0.77,0.78,0.78	
Copper	0.95,0.64,0.54	0.98,0.82,0.76	
Gold	1.00,0.71,0.29	1.00,0.86,0.57	
Aluminum	0.91,0.92,0.92	0.96,0.96,0.97	
Silver	0.95,0.93,0.88	0.98,0.97,0.95	

Table 1: Values of $F(0°)$ for various materials. (*table from "Real-Time Rendering, 3rd edition" used with permission from A K Peters*).

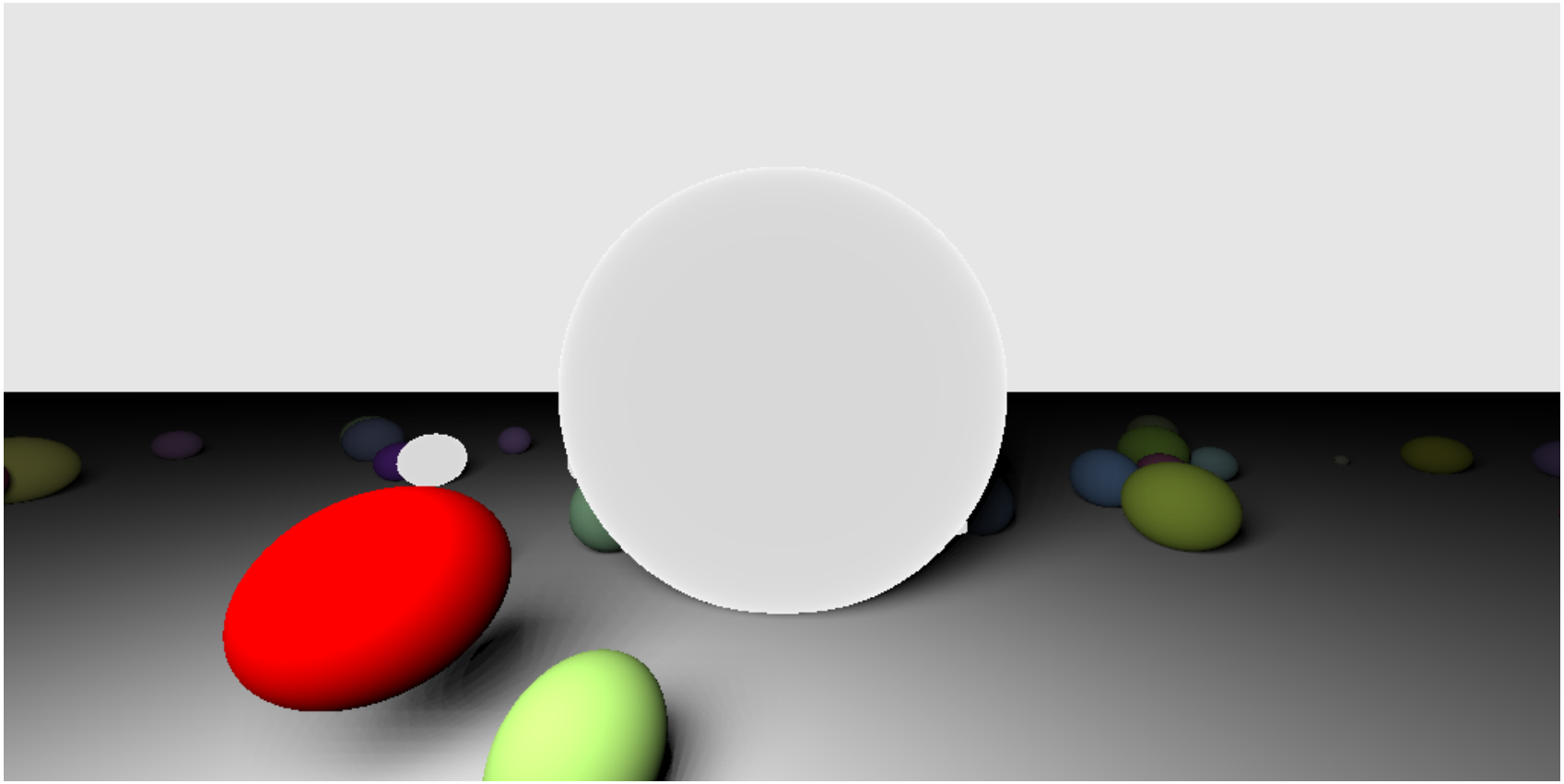
coefficients de Fresnel $\eta = 1.3$, $F0 = 0.02$ (eau)



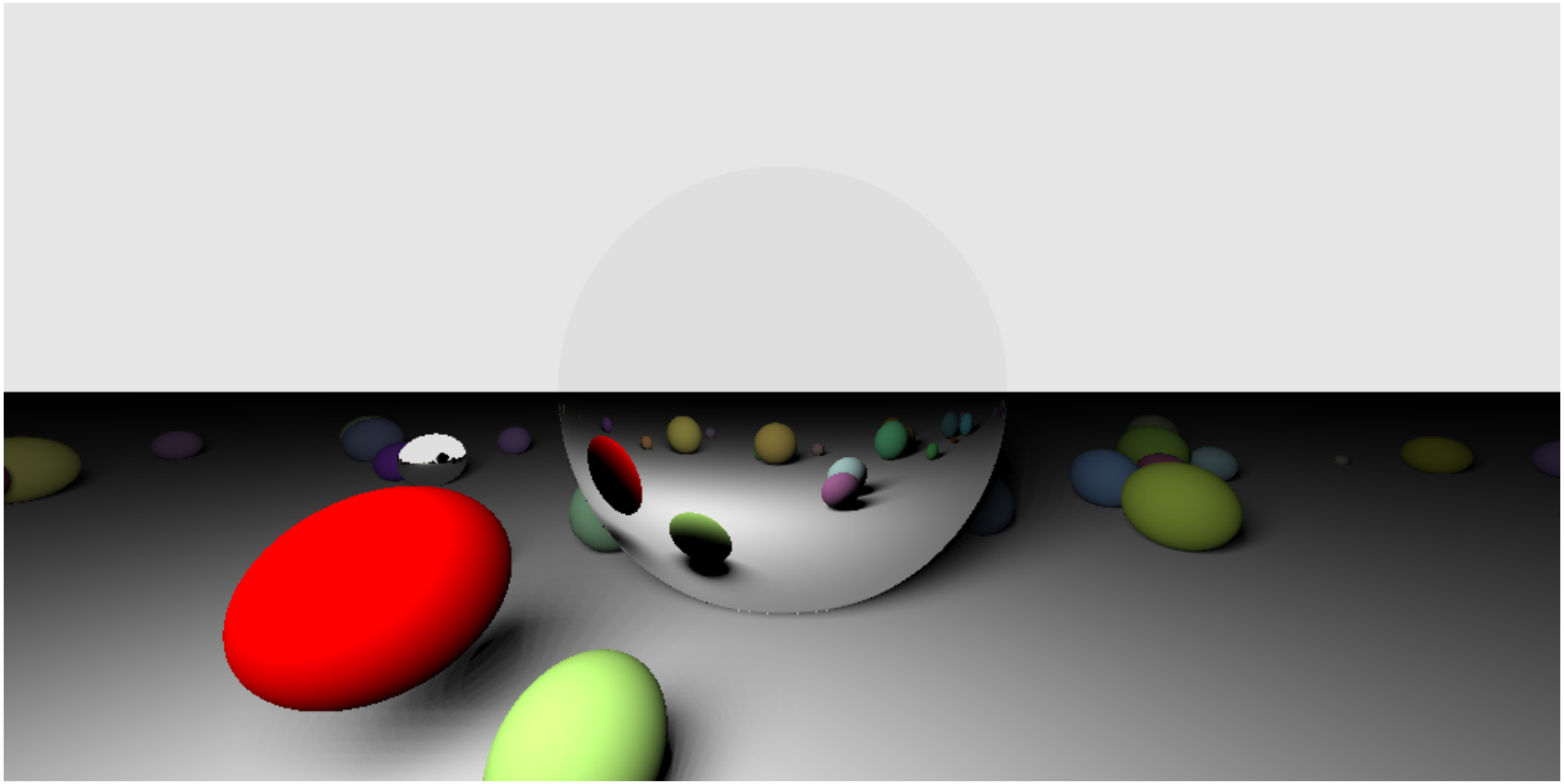
coefficients de Fresnel $\eta = 1.3$, $F0 = 0.02$ (eau)



coefficients de Fresnel $\eta = ?$, $F0 = 0.60$ (metal)



coefficients de Fresnel $\eta = ?$, $F0 = 0.60$ (metal)



et alors ?

en résumé :

- ombre : créer un rayon vers une source,
  tester les intersections, si intersection le point est à l'ombre,
  sinon le point est éclairé, évaluer le modèle de Lambert.
- miroir : créer un rayon dans la direction miroir,
  tester les intersections, trouver le point visible dans le reflet,
  vérifier s'il est éclairé ou à l'ombre,
  + évaluer le coefficient de Fresnel (ou son approximation).

il faut stocker, pour chaque objet, la couleur pour le modèle de Lambert, ou l'indice de réfraction / F0 si c'est un miroir.

et alors ?

pénombres et lumières :

- on teste plein de sources disposées dans un carré, sur une ligne, sur une (hemi) sphère...
- mais selon le nombre, la pénombre est découpée en bandes, ou pas,
- c'est quand même un peu moche...

pourquoi utiliser plein de sources ? les images ne sont pas très naturelles avec une seule source / point qui éclaire tous les objets.

comment ça se code ?

restructurer le code :

- utiliser une structure pour décrire un ensemble d'objets,
- utiliser une structure pour décrire la matière de chaque objet : couleur diffuse / Lambert, F0 pour les miroirs,
- écrire une fonction intersection() qui teste tous les objets,
- écrire une fonction qui détermine si un point est éclairé ou à l'ombre.

les détails dans le tp...

éclairage par un panneau

2 solutions :

- disposer les sources sur une grille dans le panneau,
- ou disposer les sources aléatoirement dans le panneau ! pour chaque pixel.

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

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...

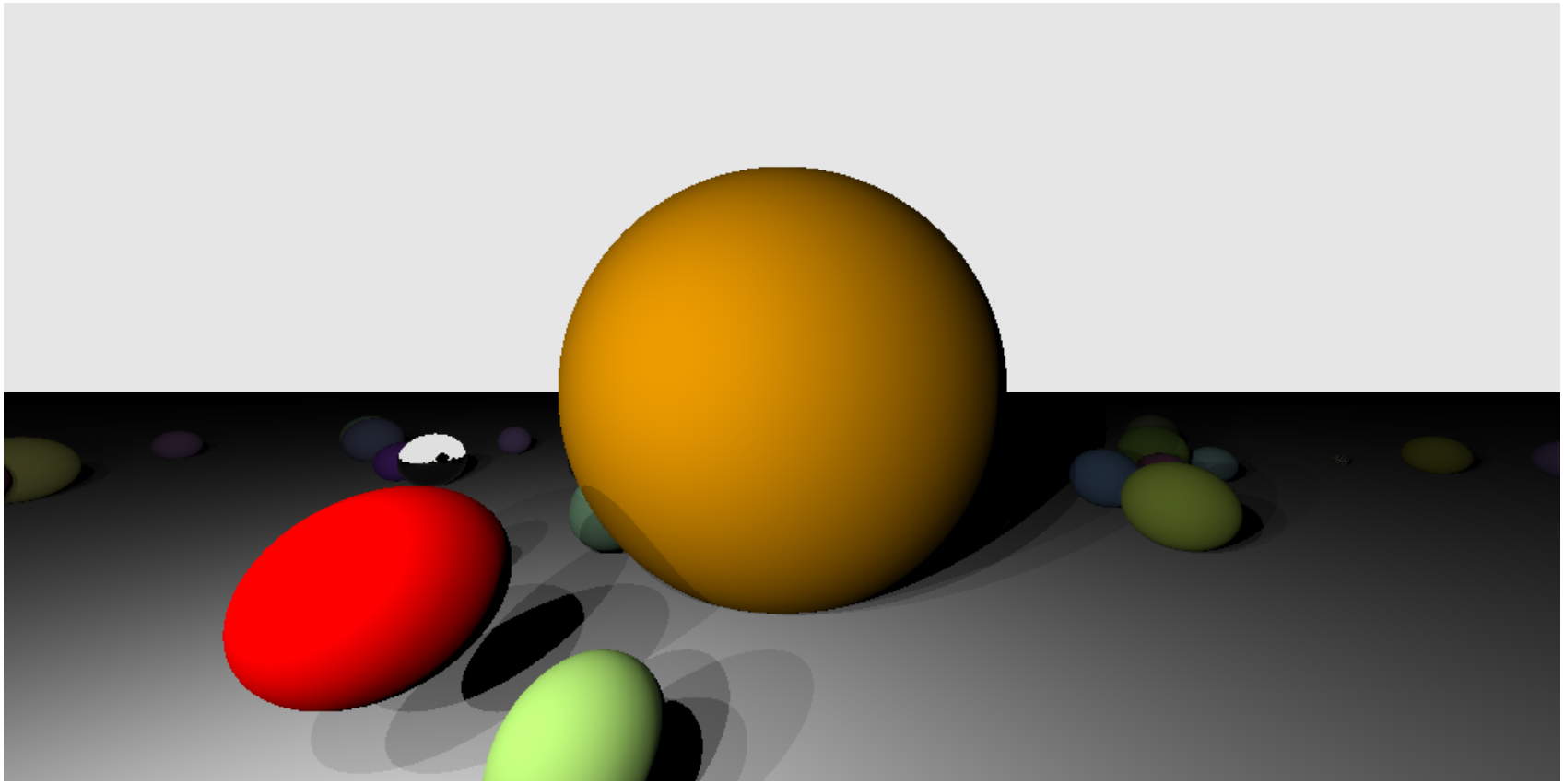
des points dans une grille

mieux :

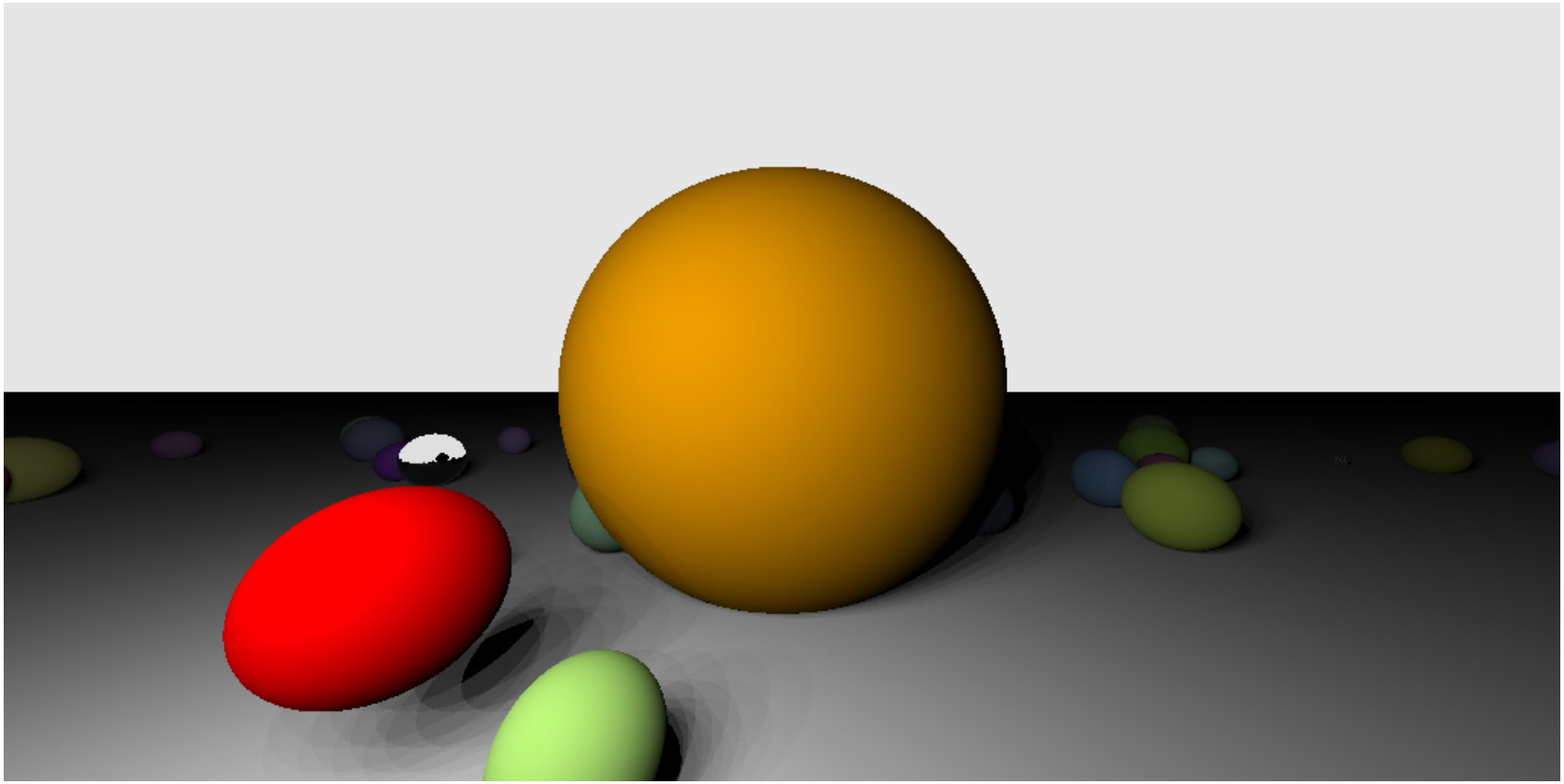
```
Point a= Point(0, 0, -1);          // origine
Vector u= Vector(1, 0, 0);         // axe 1
Vector v= Vector(0, 1, 0);         // axe 2

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 $\vec{v}$...

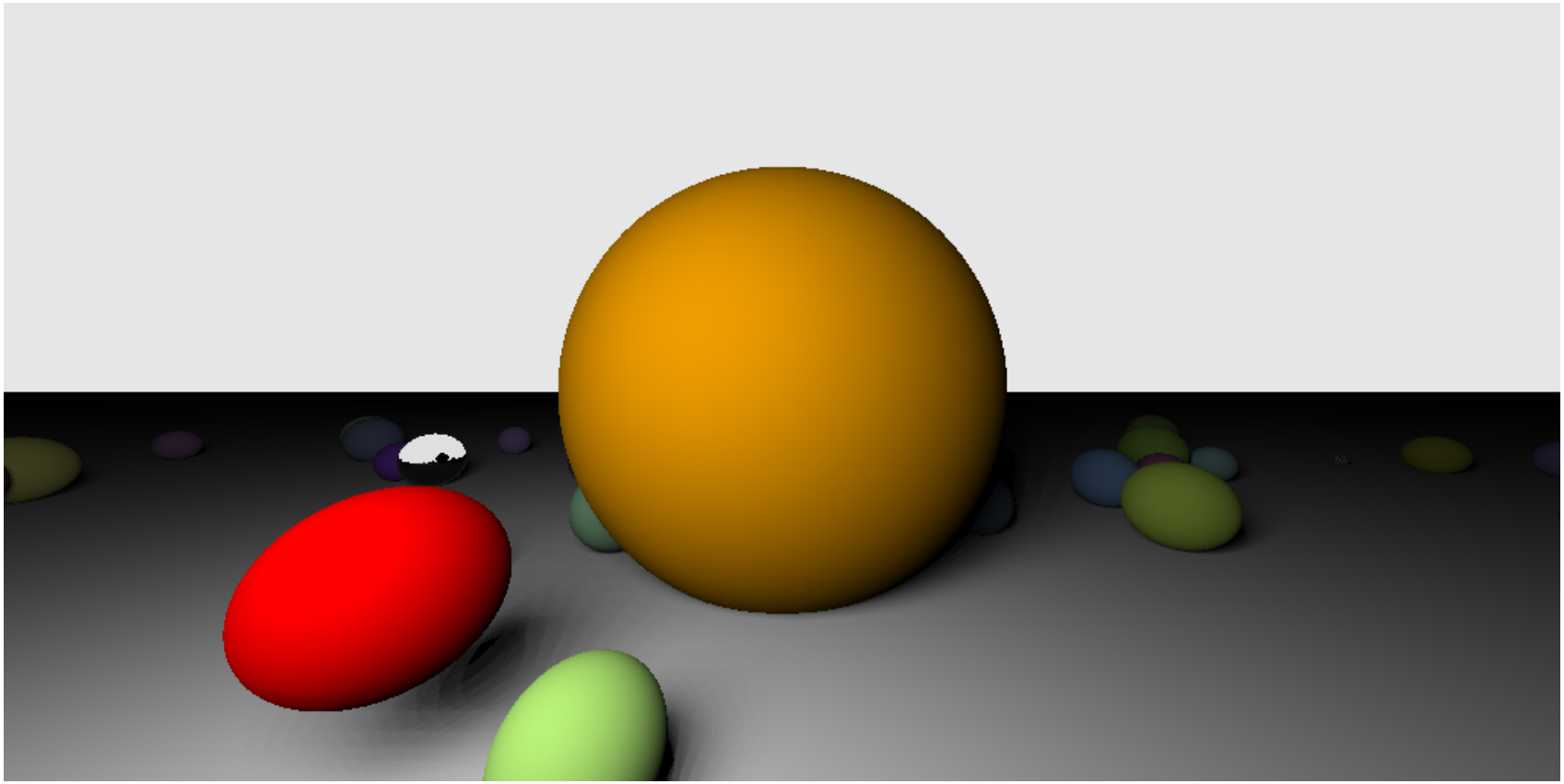
des points dans une grille, 4



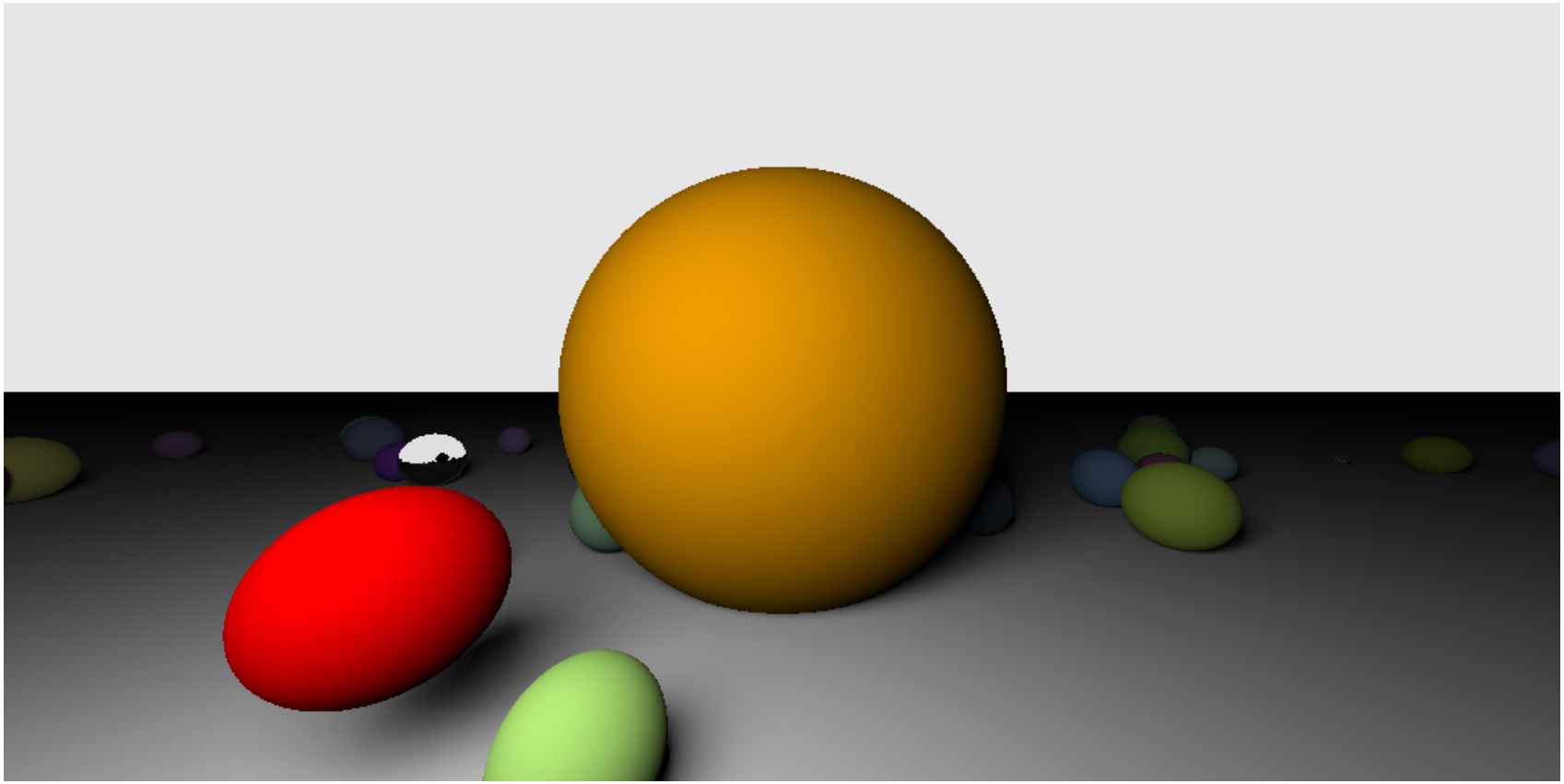
des points dans une grille, 16



des points dans une grille, 64



des points dans une grille, 256



des nombres aléatoires

```
#include <random>

// initialisation
std::random_device hwseed;
unsigned seed= hwseed();

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

float u1= u(rng);        // nombre aleatoire entre 0 et 1
float u2= u(rng);
```

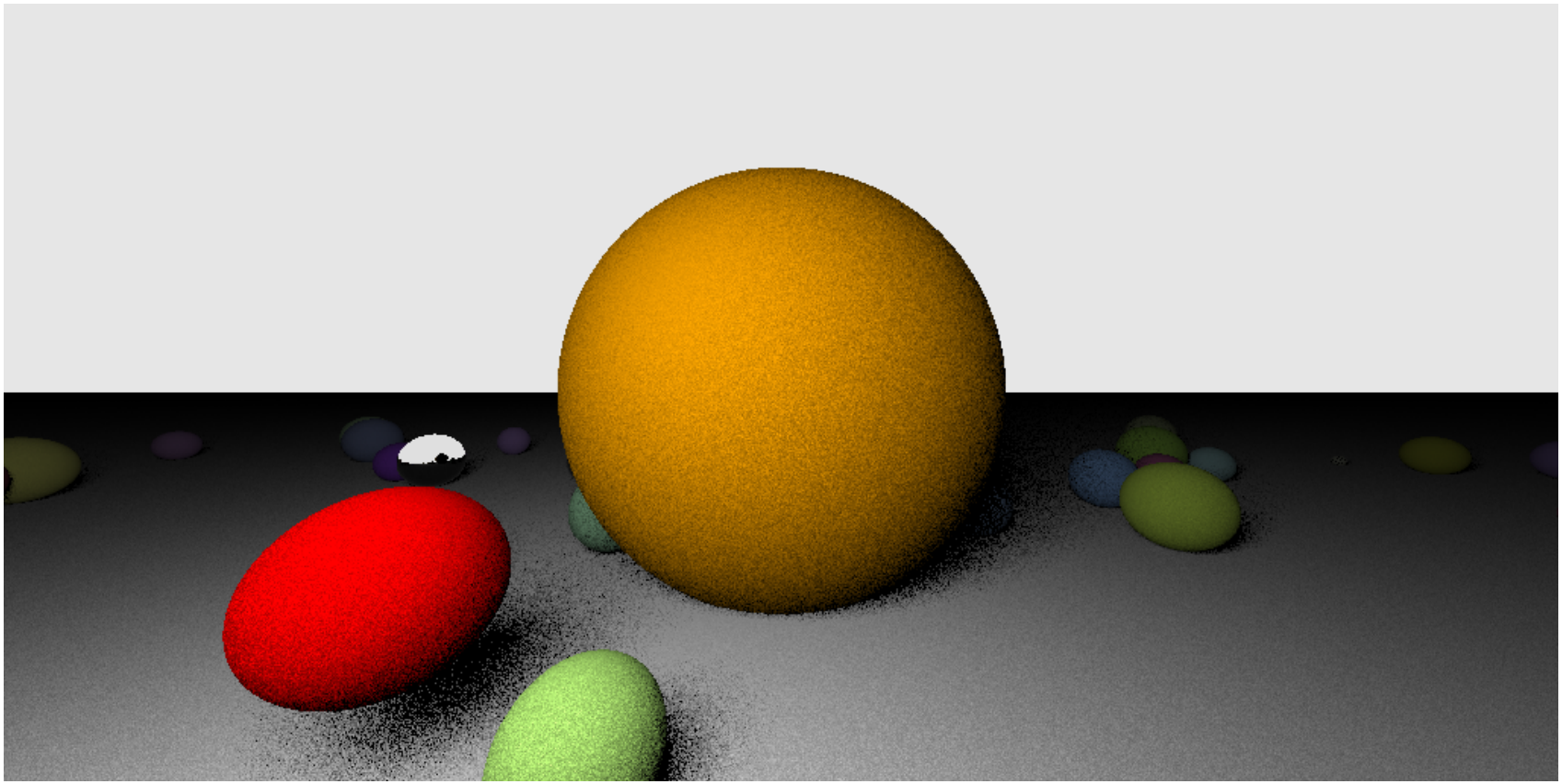
des points aléatoires dans une grille

```
#include <random>

std::random_device hwseed;
unsigned seed= hwseed();
std::default_random_engine rng( seed );
std::uniform_real_distribution<float> u;

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 < 10*10; i++)
{
    float u1= u(rng);        // nombre aleatoire entre 0 et 1
    float u2= u(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...

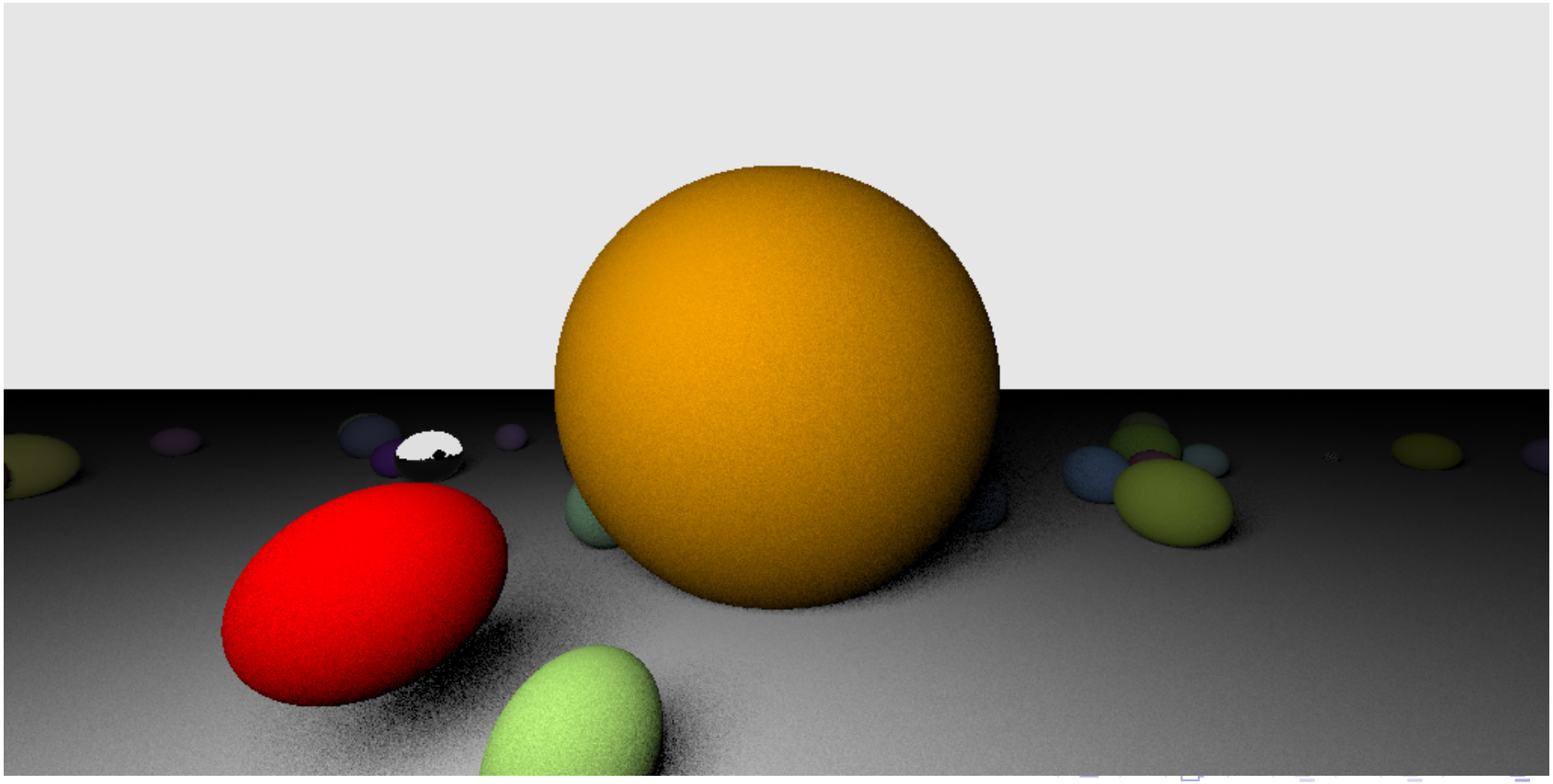
des points aléatoires dans une grille, 4

rappel : chaque pixel utilise des points différents...

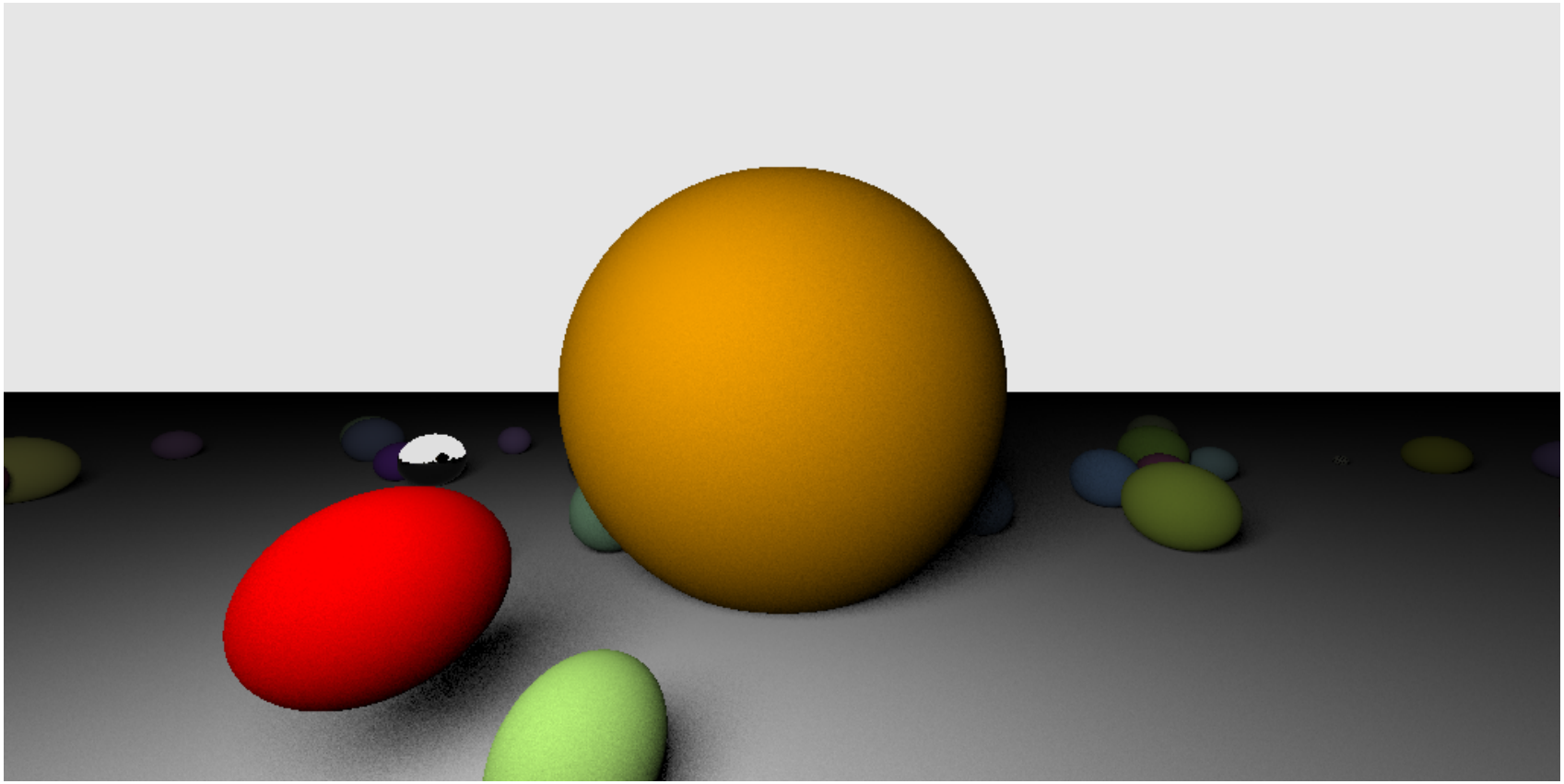


des points aléatoires dans une grille, 16

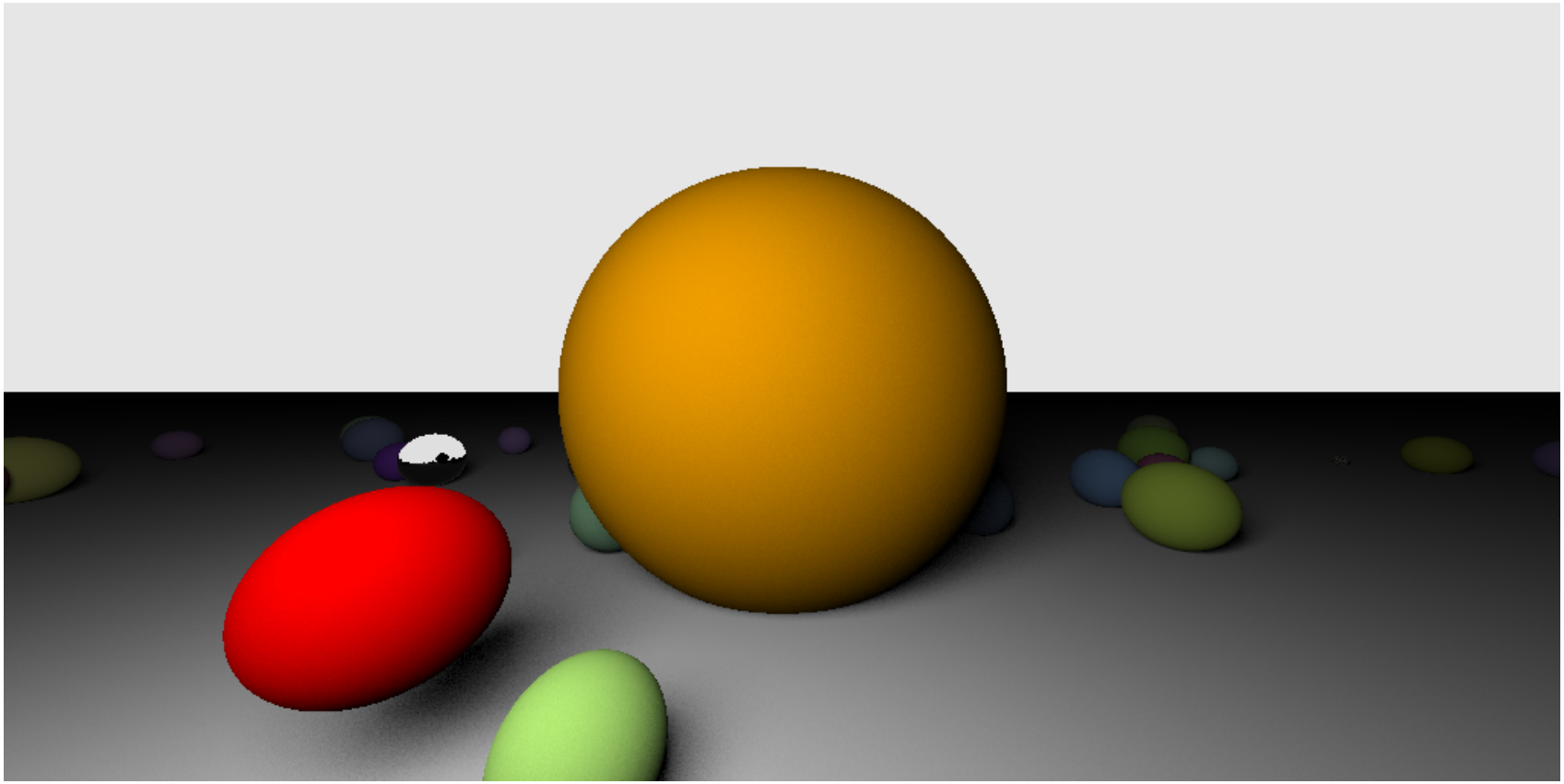
c'est assez moche aussi,
mais on imagine plus facilement le "bon" résultat...



des points aléatoires dans une grille, 64



des points aléatoires dans une grille, 256



des points aléatoires dans une grille

et alors ?

- nouveau type de source de lumière,
- lumière réfléchie par un objet dépend de :
- soit un ensemble de points,
- soit un panneau lumineux
  (remplacé par un ensemble aléatoire de points...)
- et bien sur de sa matière !!

restructurer le code... écrire une fonction qui calcule comment le panneau lumineux éclaire un point d'intersection.

éclairage par un dome / ciel / hemisphère

construire des directions ?

- avec 2 angles, cf $\theta \in [0..\pi/2]$ et $\phi \in [0..2\pi]$,
- et passage en coordonnées xyz :
  $x = \cos\phi \sin\theta$
  $y = \cos\theta$
  $z = \sin\phi \sin\theta$

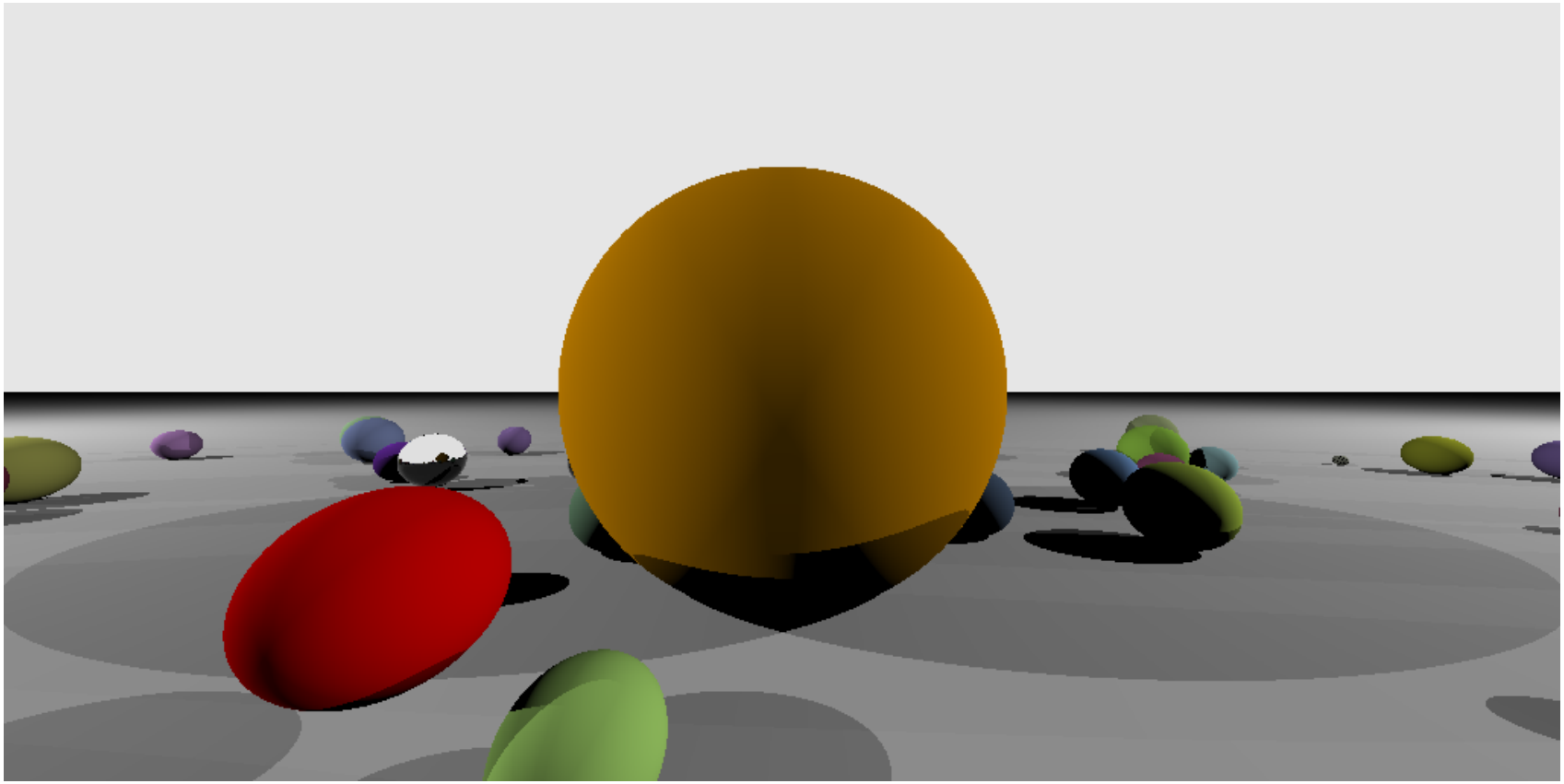
on va utiliser une recette de cuisine,
les détails dans le cours de M2...

des sources sur un dome

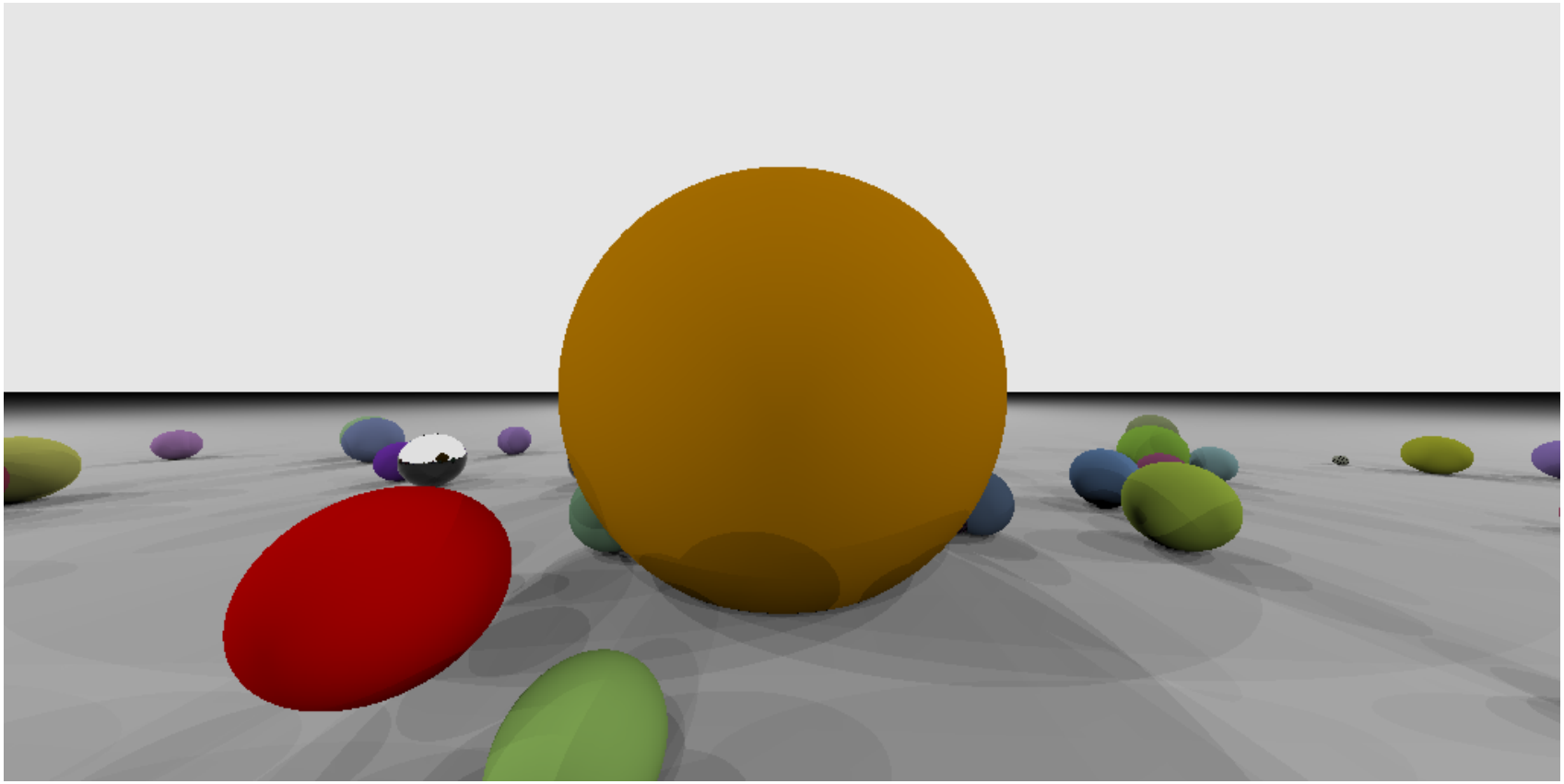
```
#include <cmath>

std::vector<Vector> directions;
for(int i= 0; i < 10; i++)
for(int j= 0; j < 10; j++)
{
    float cos_theta= float(i) / float(10);
    float sin_theta= std::sqrt(1 - cos_theta*cos_theta);

    float phi= float(j) / float(10) * float(M_PI*2);

    Vector d= Vector(std::cos(phi) * sin_theta,
                     cos_theta,
                     std::sin(phi) * sin_theta);
    directions.push_back( d );
}
```

une relation utile : $\cos^2\theta + \sin^2\theta = 1$, ou $\sin\theta = \sqrt{1 - \cos^2\theta}$

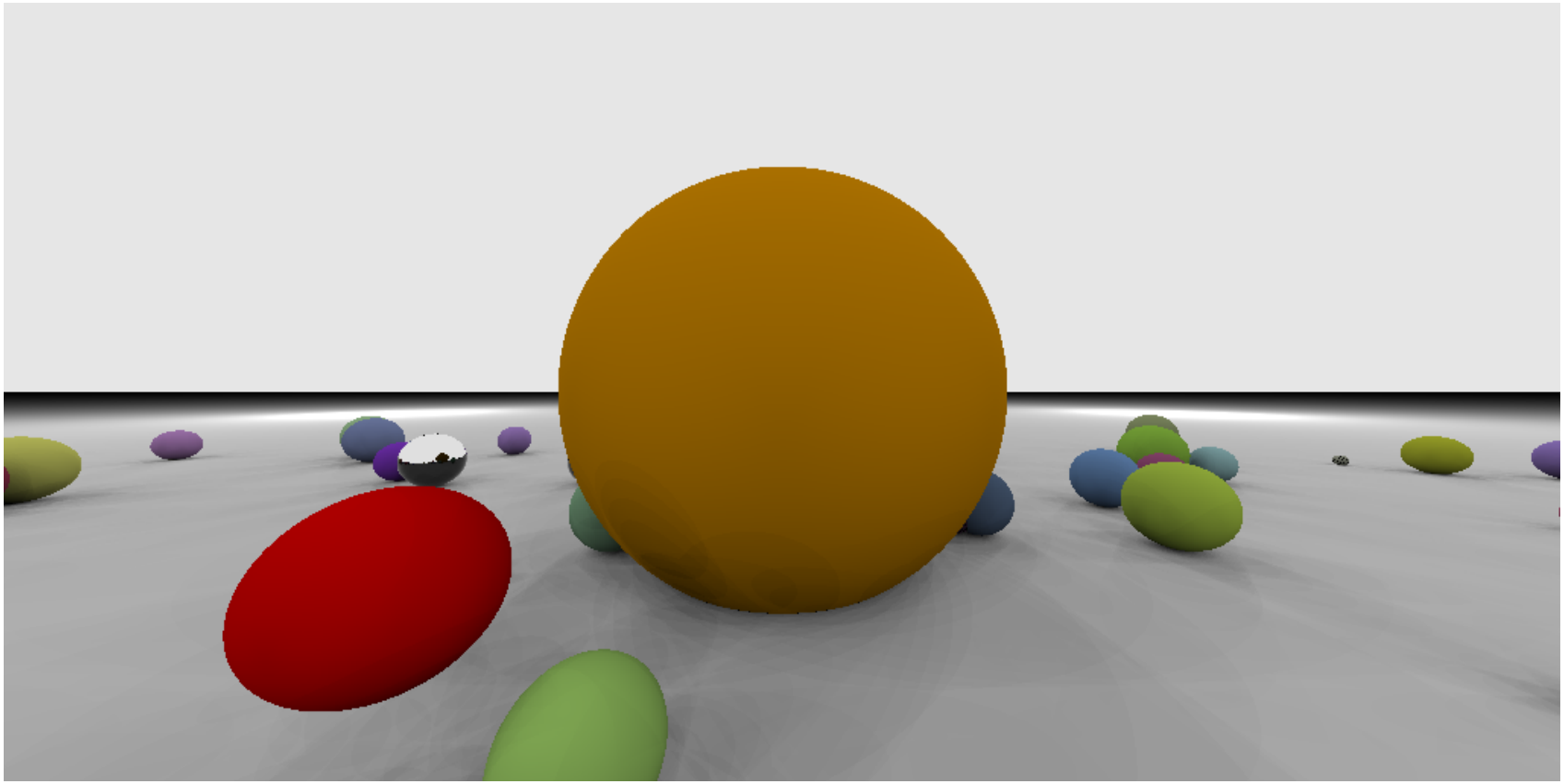
des sources sur un dome, 4



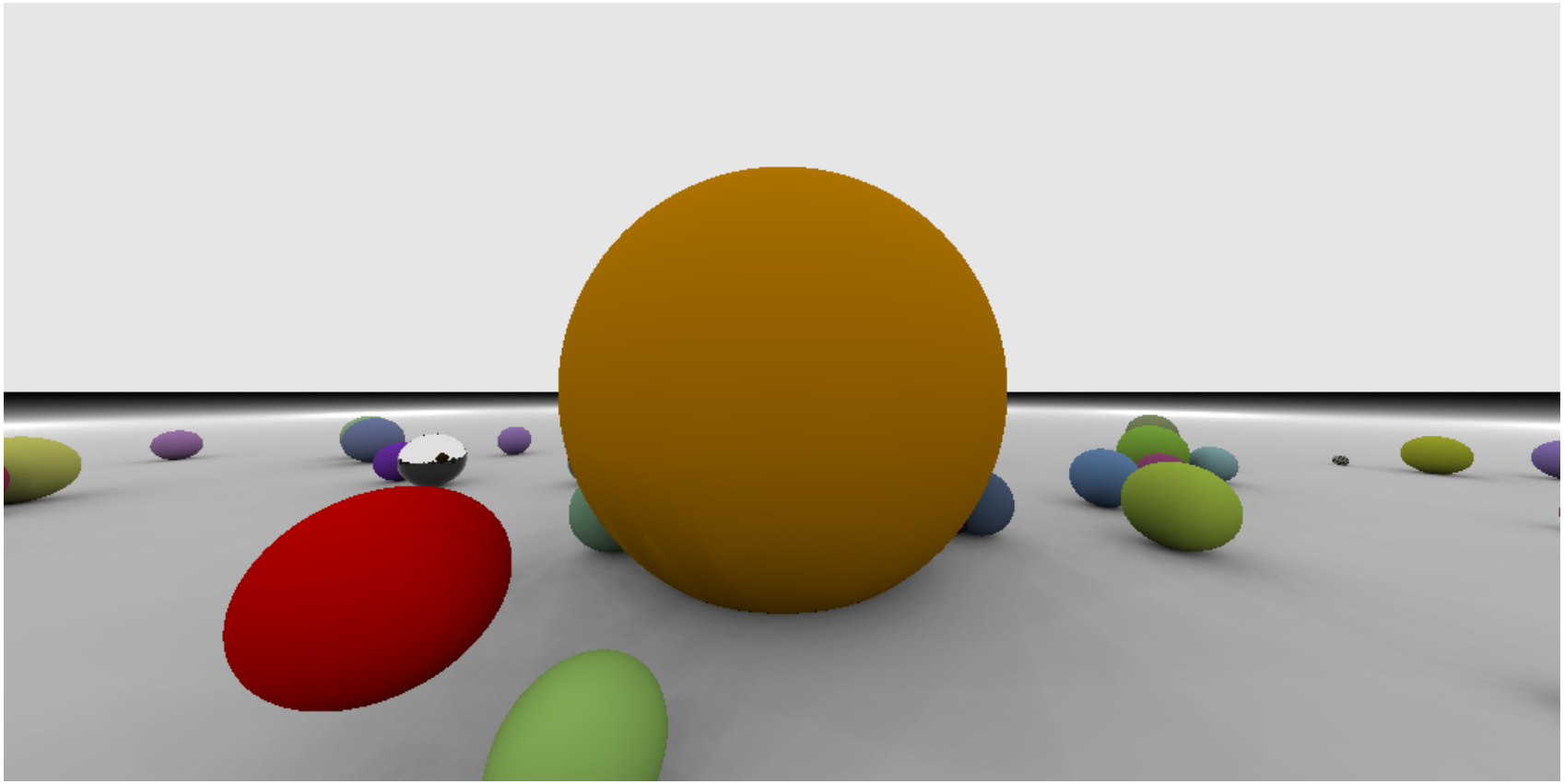
des sources sur un dome, 16



des sources sur un dome, 64



des sources sur un dome, 256



des directions aléatoires dans un dome

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

std::random_device hwseed;
unsigned seed= hwseed();
std::default_random_engine rng( seed );
std::uniform_real_distribution<float> u;

std::vector<Vector> directions;
for(int i= 0; i < 10*10; i++)
{
    float cos_theta= u(rng);
    float sin_theta= std::sqrt(1 - cos_theta*cos_theta);

    float phi= u(rng) * float(M_PI*2);

    Vector d= Vector(std::cos(phi) * sin_theta,
                     cos_theta,
                     std::sin(phi) * sin_theta);
    directions.push_back( d );
}
```

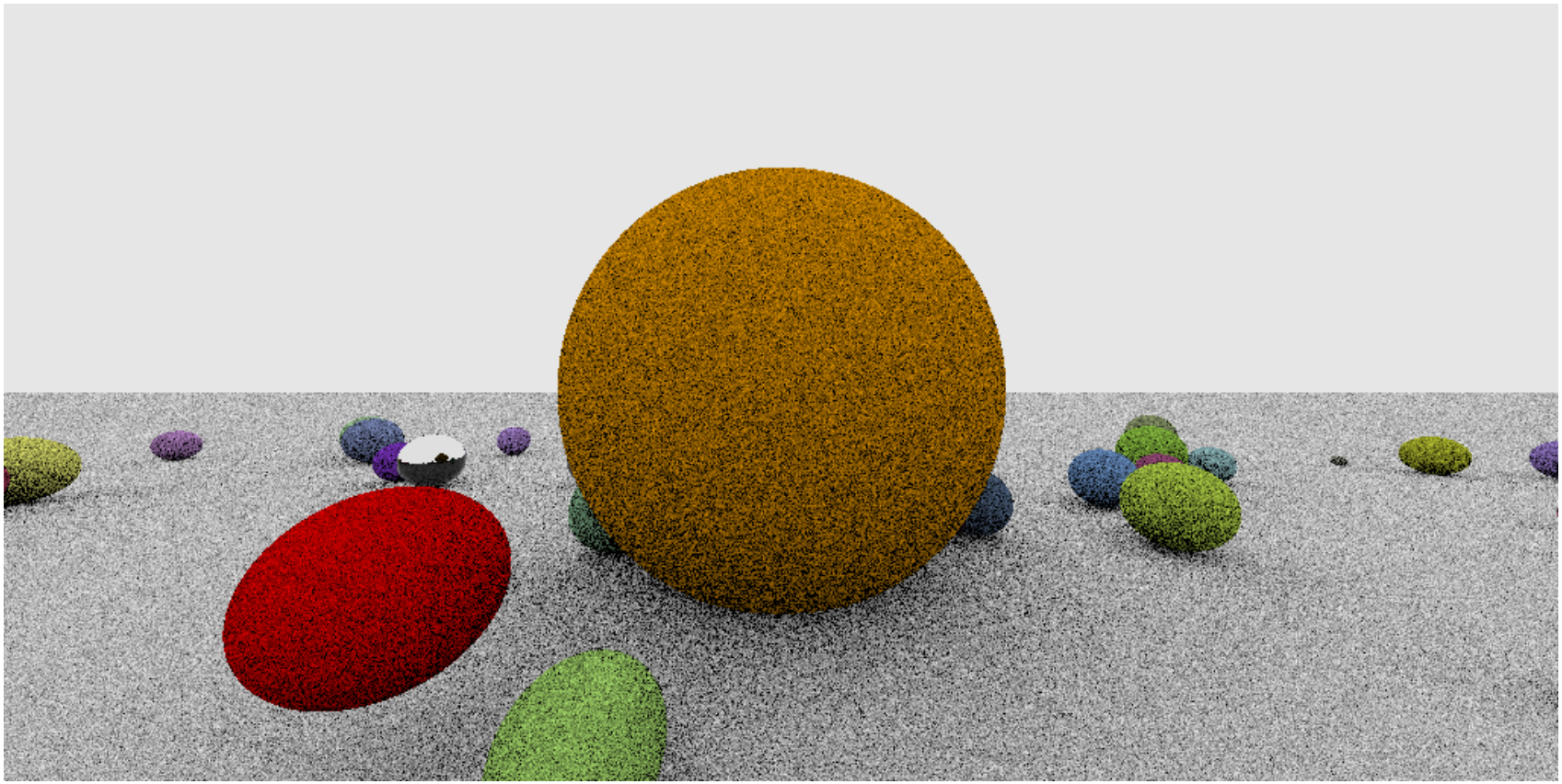
des directions aléatoires dans un dome

qu'est ce qu'on calcule ?

- comme pour les points, mais...
- on peut simplifier :
- les directions sont unitaires (longueur 1), donc $\frac{1}{||\vec{d}||^2} = 1$
- $L_r = f_r \times L_e \times \cos\theta$, pour une matière diffuse / Lambert,
- $L_r = F(\vec{n}, \vec{v}) \times L_e$, pour un miroir / Fresnel, si le ciel / dome est visible dans la direction miroir...

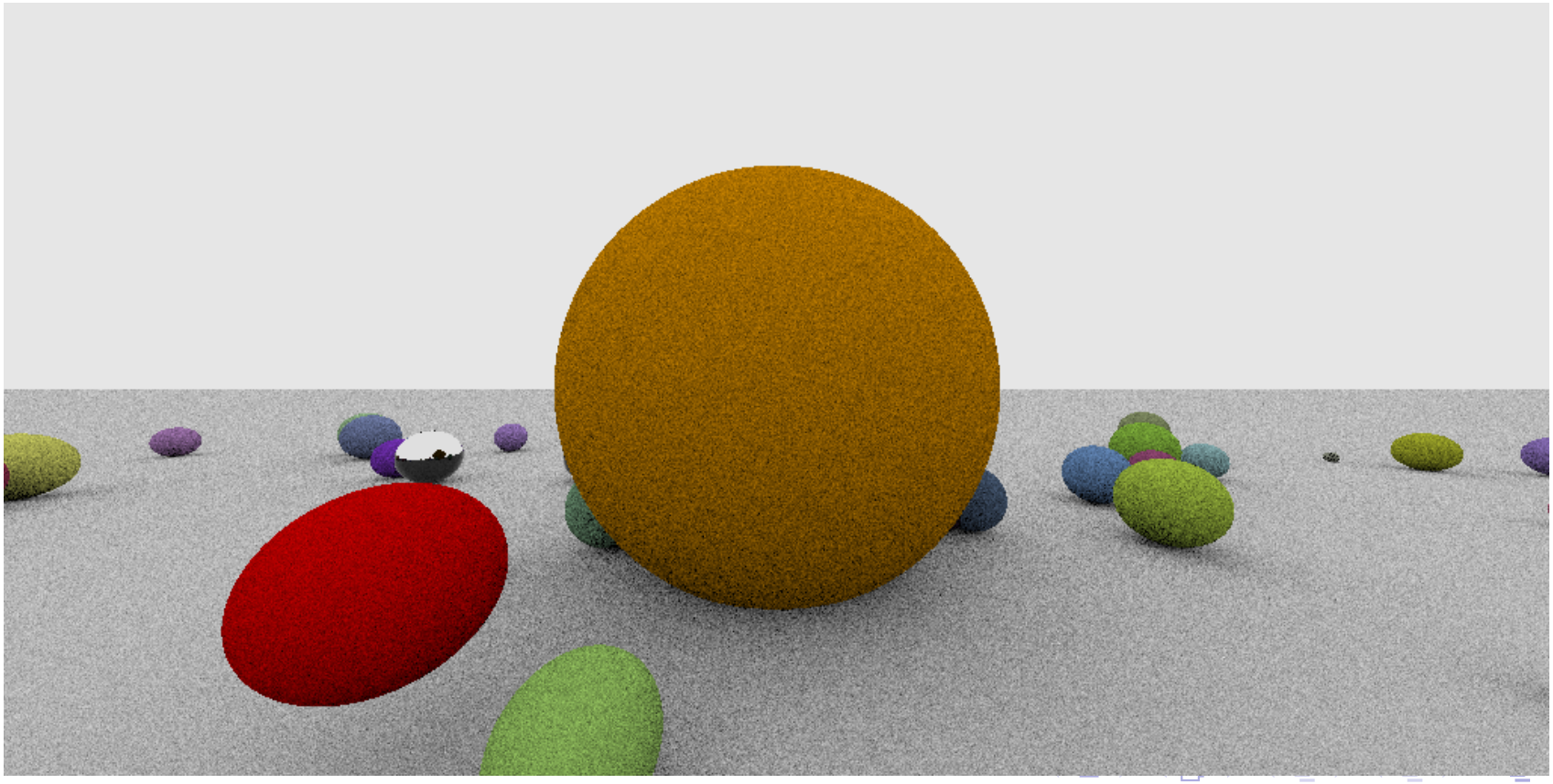
des directions aléatoires dans un dome, 4

rappel : chaque pixel utilise des directions différentes...

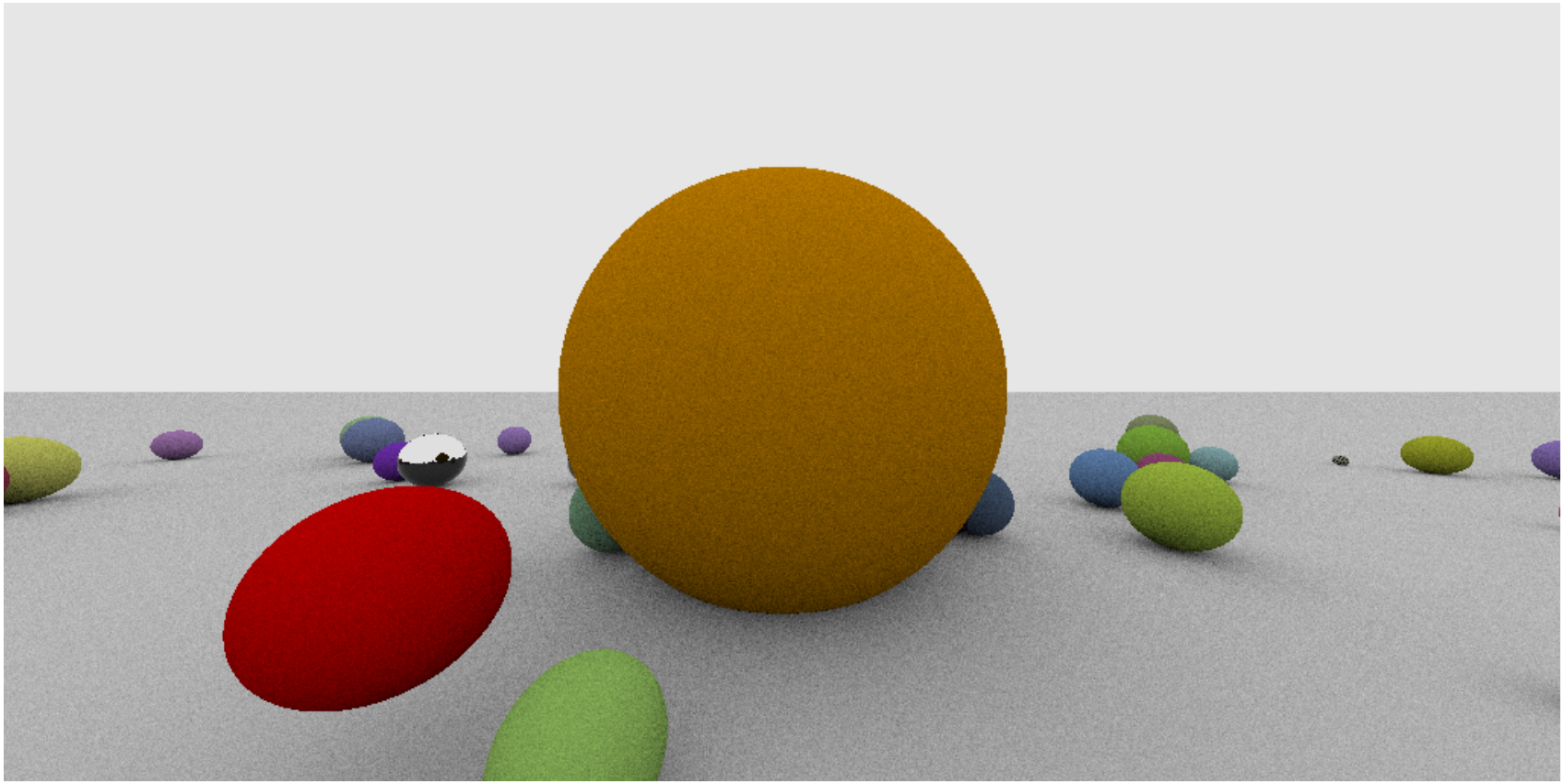


des directions aléatoires dans un dome, 16

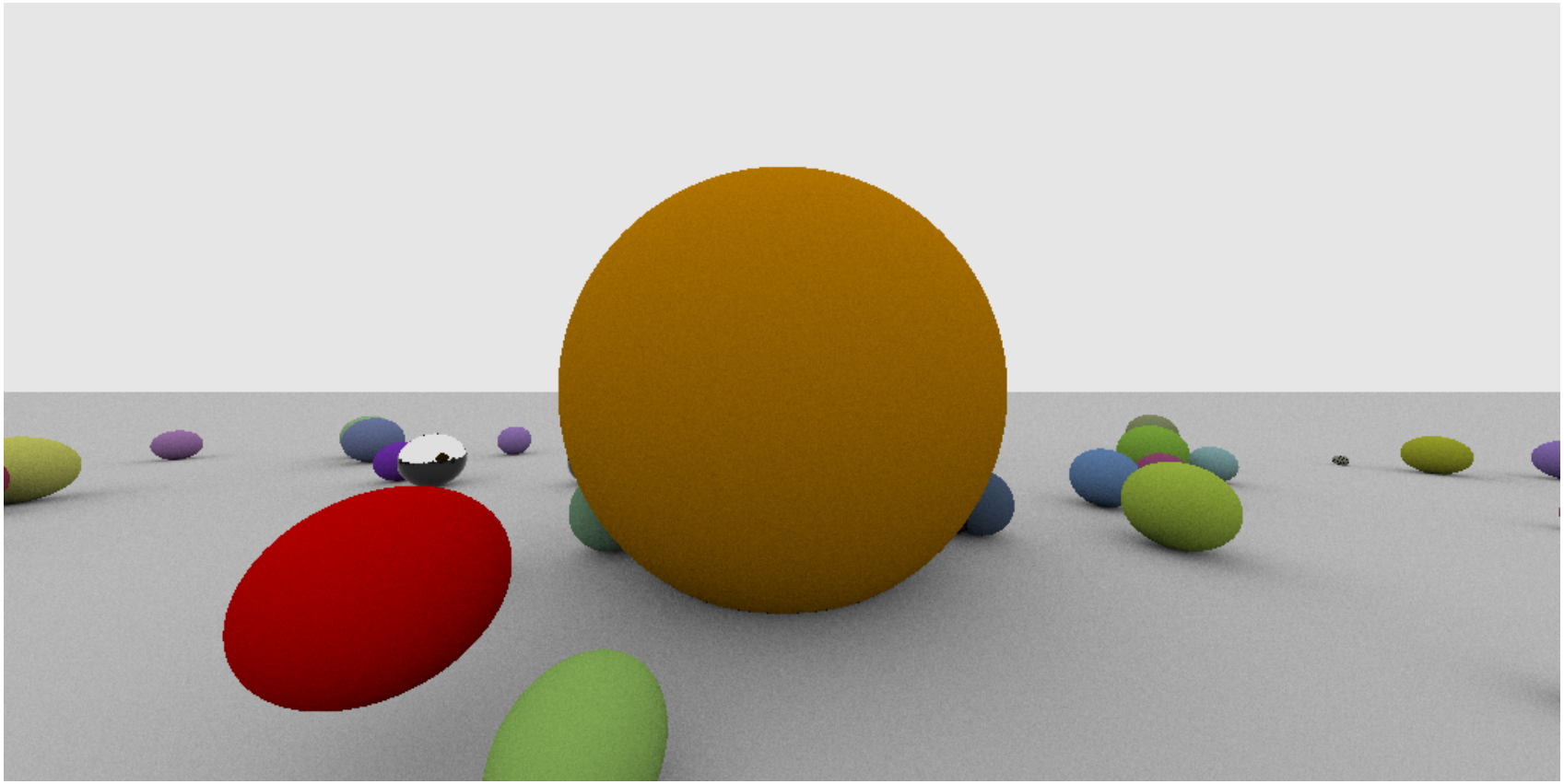
même constat : c'est moche,
mais on imagine facilement le "bon" résultat...



des directions aléatoires dans un dome, 64



des directions aléatoires dans un dome, 256



des directions aléatoires dans un dome

et alors ?

- nouveau type de source de lumière,
- lumière réfléchie par un objet dépend de :
- soit un ensemble de points,
- soit un panneau lumineux
  (remplacé par un ensemble aléatoire de points...),
- soit un dome lumineux
  (remplacé par un ensemble aléatoire de directions...),
- et bien sur de sa matière !!

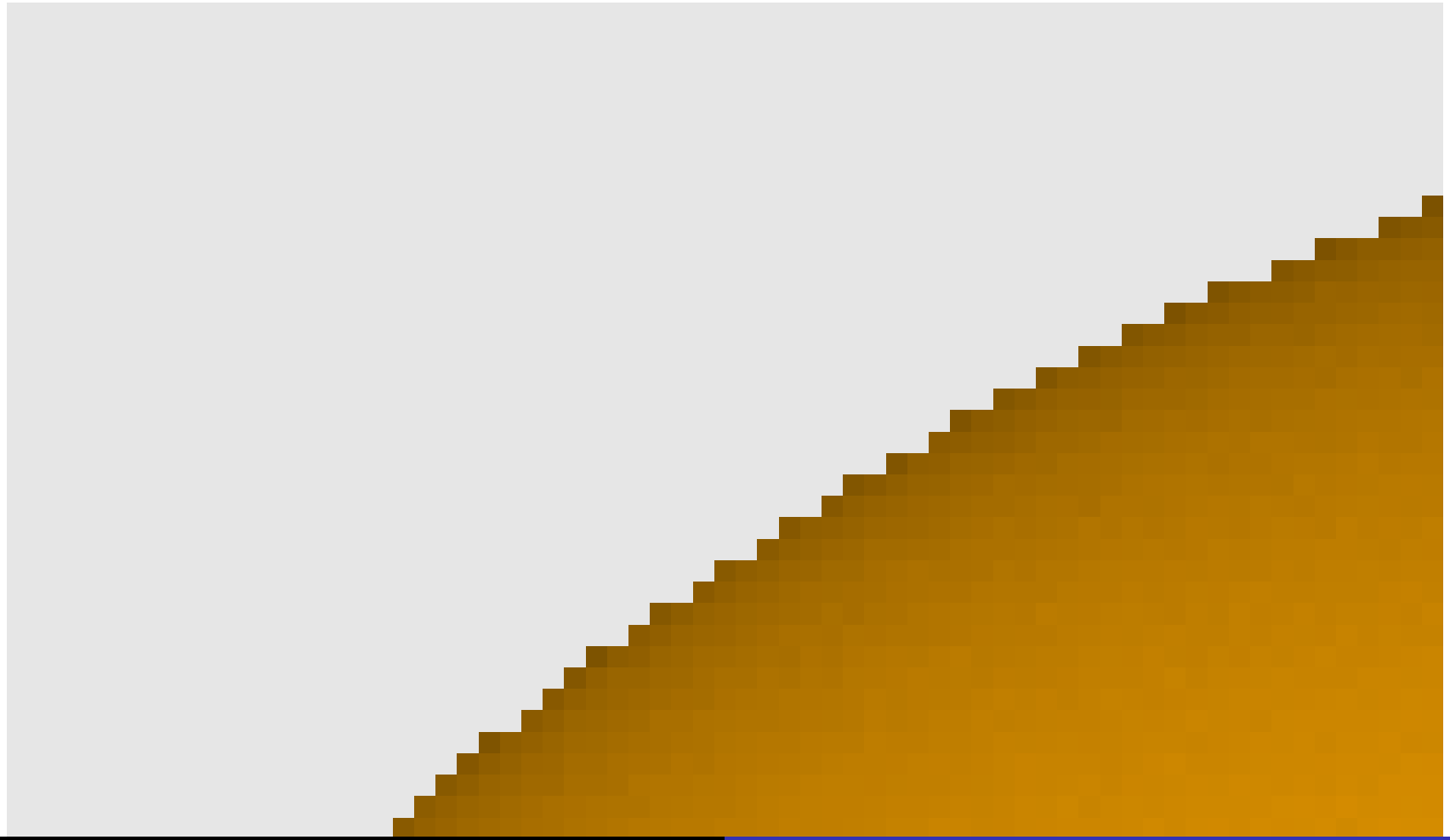
restructurer le code...

et alors ?

```
Color eclairage_direct_points(
    /* parametres du point */ const Point& p, const Vector& n,
    /* parametres de la matiere */ const Color& color, const Color& F0,
    /* sources */ const std::vector<Source>& points )
{
    ...
}

Color eclairage_direct_panneau(
    /* parametres du point */ const Point& p, const Vector& n,
    /* parametres de la matiere */ const Color& color, const Color& F0,
    /* panneau */ const Panneau& panneau, + RNG )
{
    ...
}

Color eclairage_direct_dome(
    /* parametres du point */ const Point& p, const Vector& n,
    /* parametres de la matiere */ const Color& color, const Color& F0,
    /* dome */ const Dome& dome, + RNG )
{
    ...
}

// representer aussi les parametres de la matiere par une structure ??
```

bonus : anti-aliasing

ça aussi, c'est bien moche...



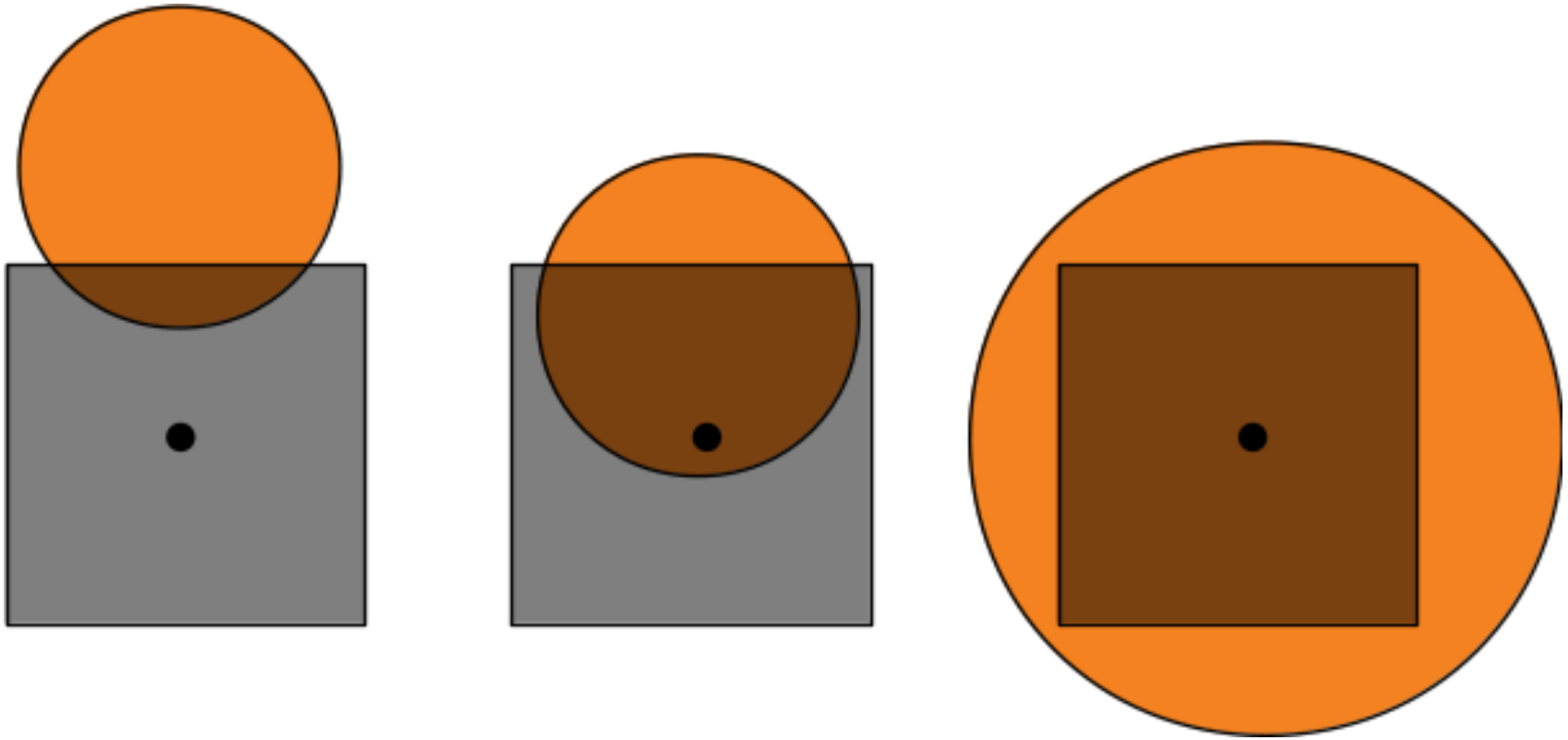
anti-aliasing

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 ?

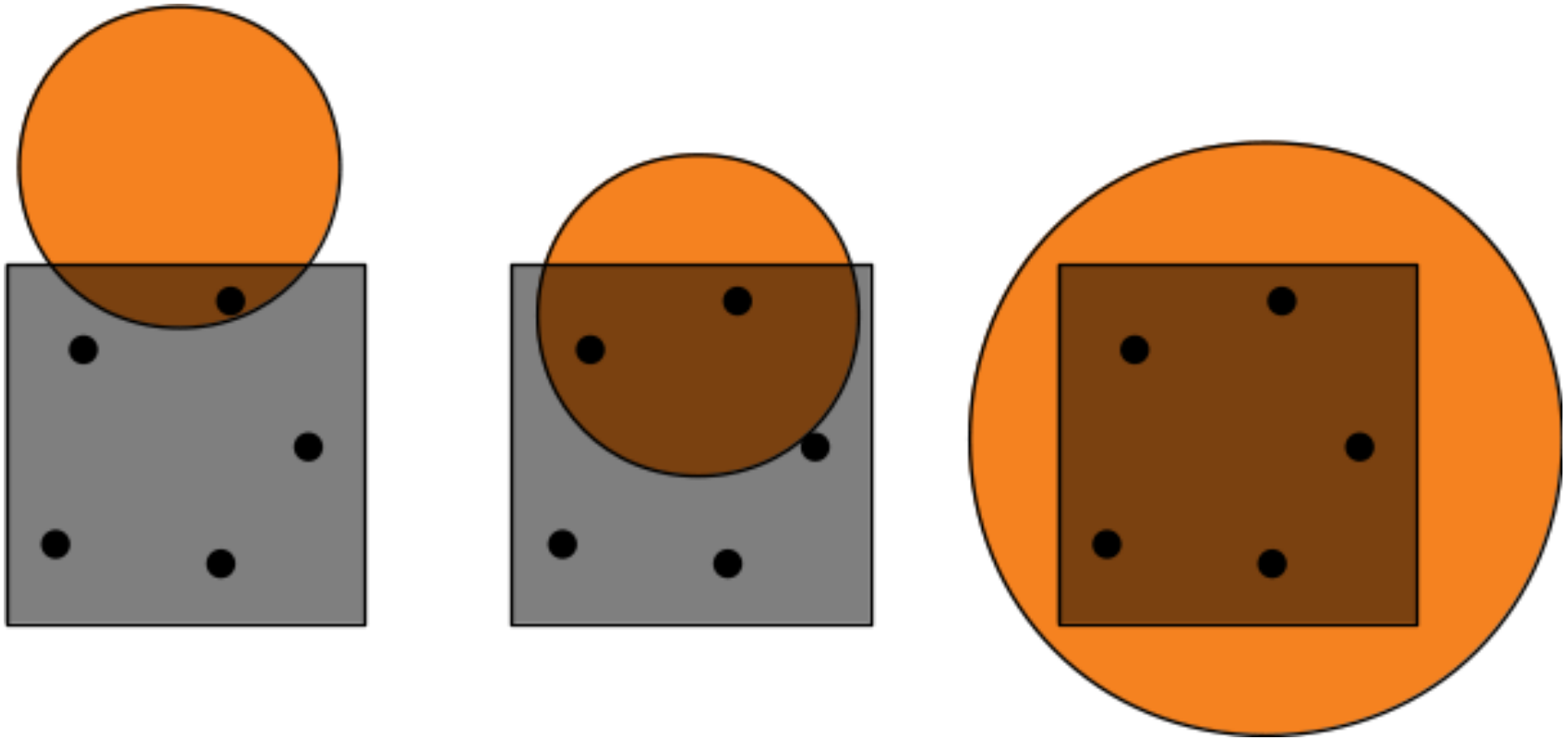
- on voudrait que la couleur du pixel dépende de l'objet,
- si l'objet couvre tout le pixel, c'est correct...
- mais, on voudrait la "bonne" proportion de la couleur du fond et de l'objet...

anti-aliasing et filtrage

2 solutions !

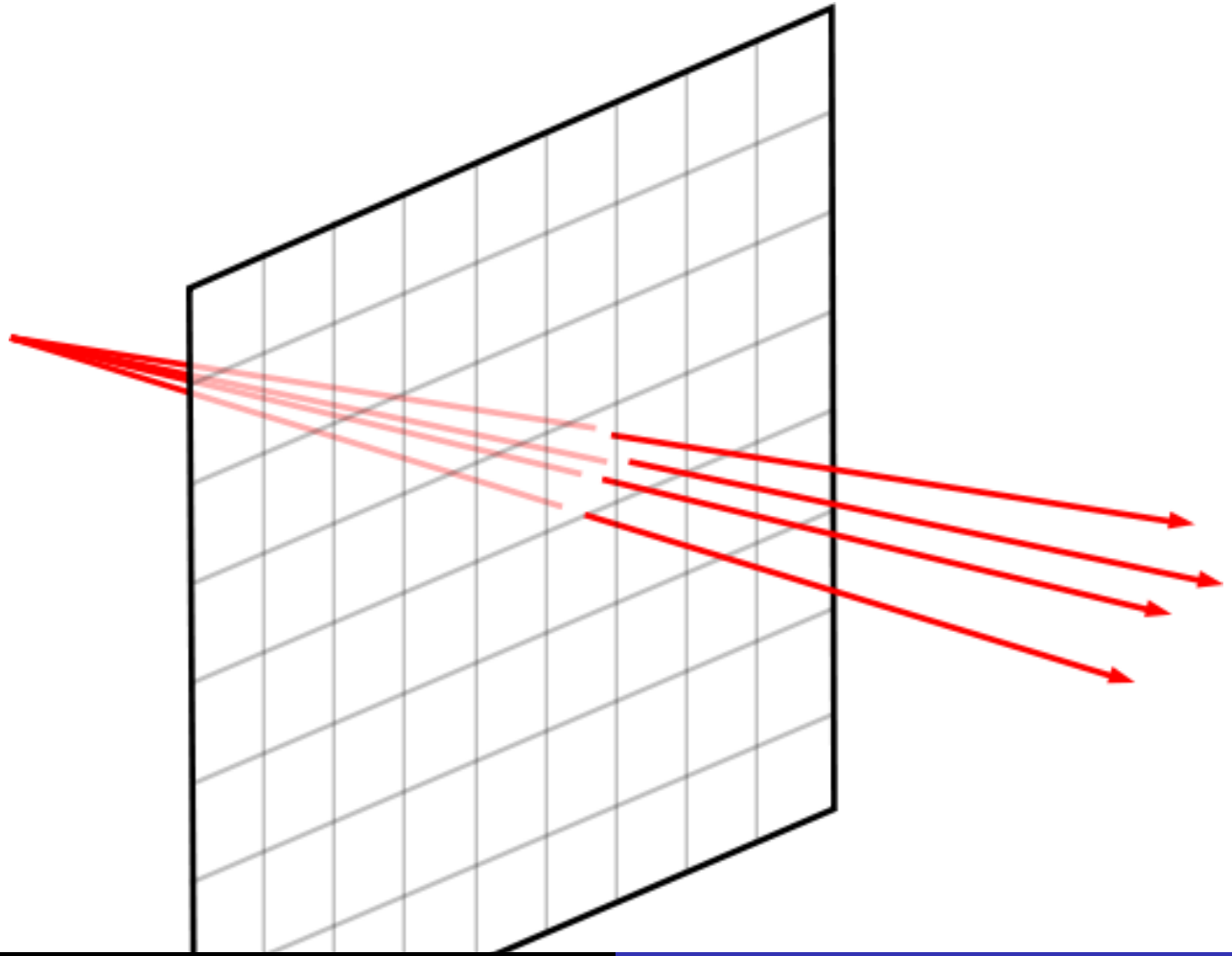
- on calcule une image plus grande, on la filtre,
  (pour éliminer les fréquences non représentables)
  et on la sous échantillonne pour obtenir la bonne résolution...
- ou, on construit plusieurs rayons par pixel et on moyenne les couleurs...

aliasing



aliasing

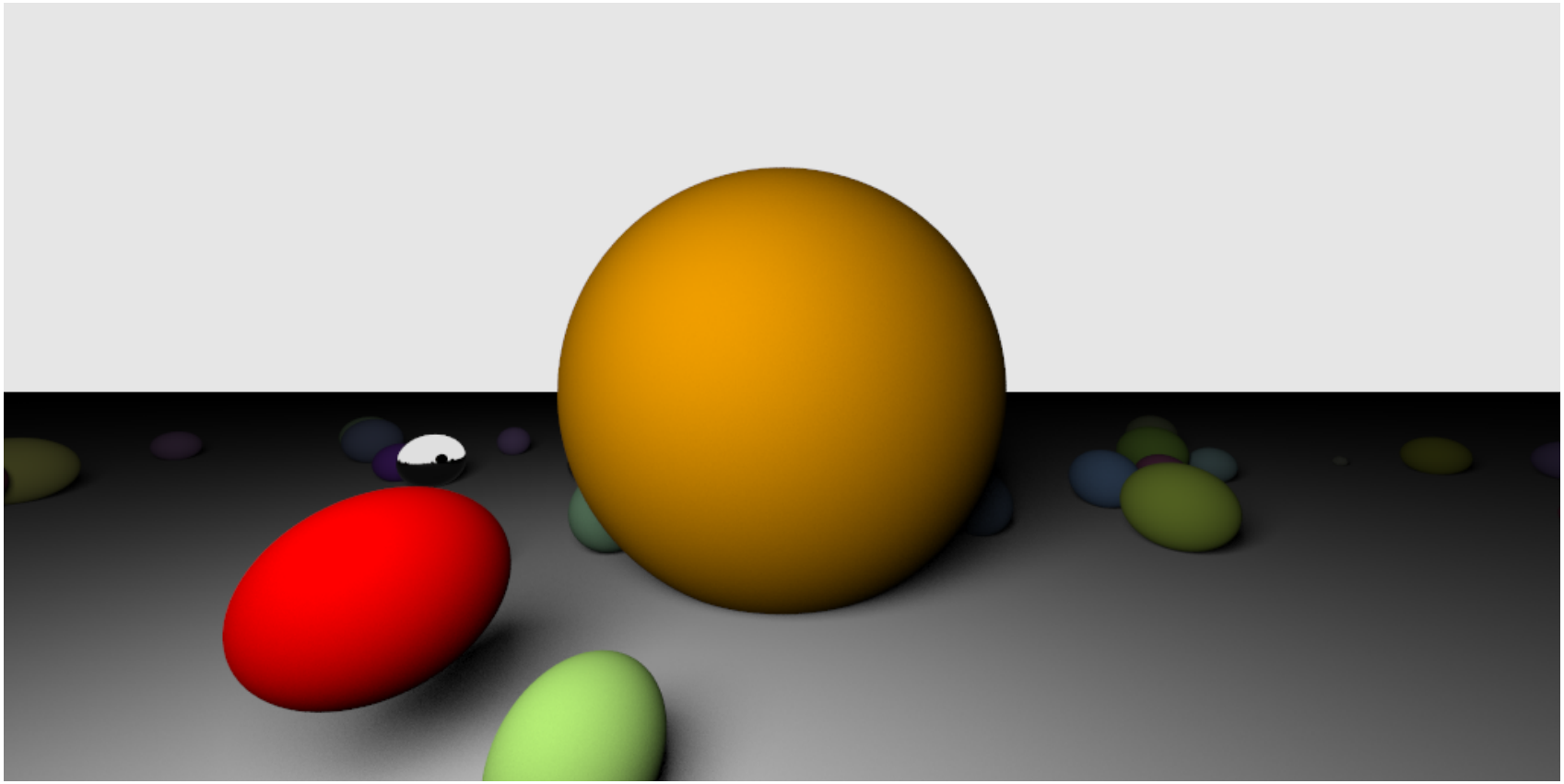
il faut générer plusieurs rayons dans chaque pixel...



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> u;

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

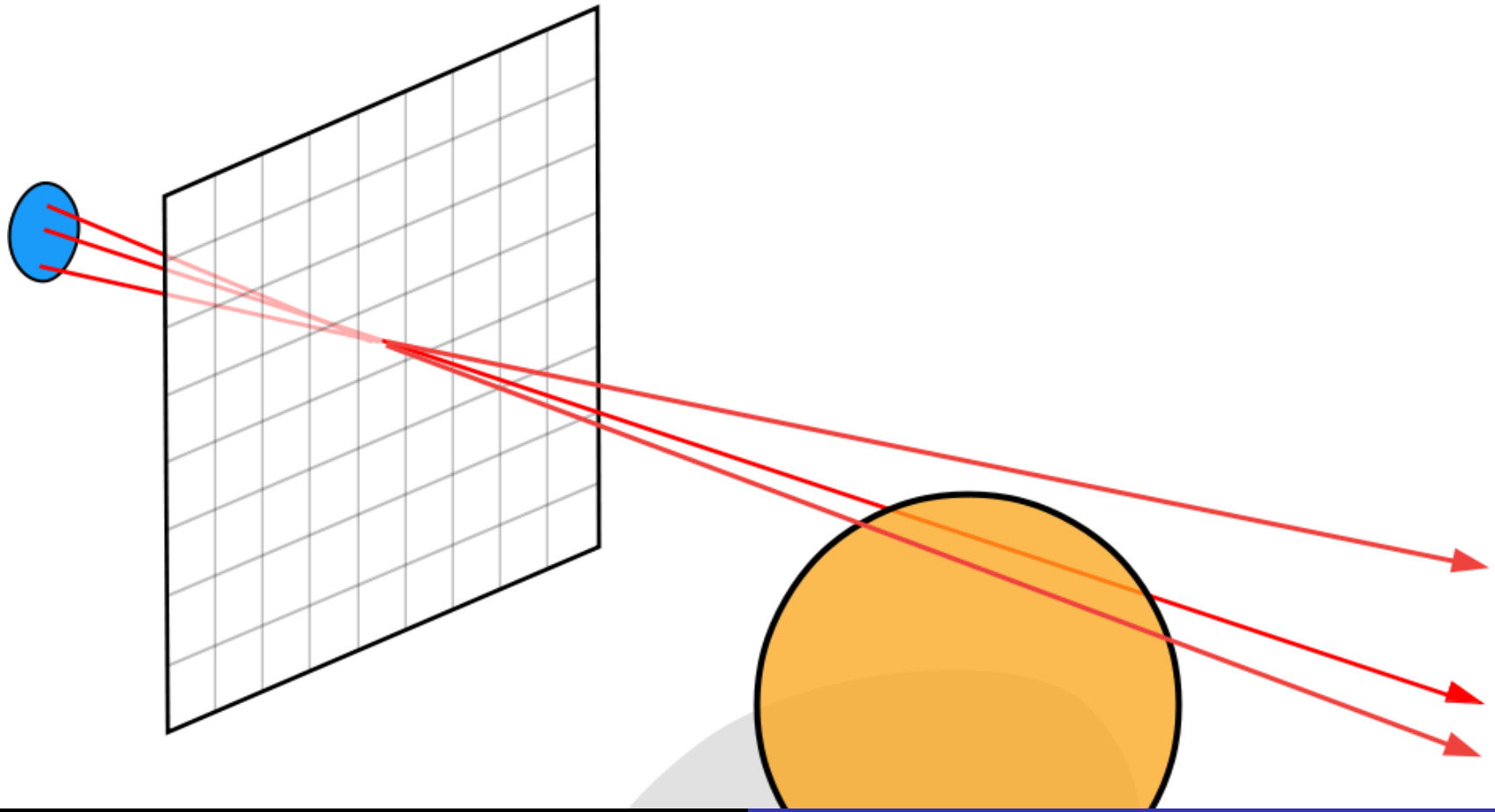
la camera est *très* simplifiée...

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 !

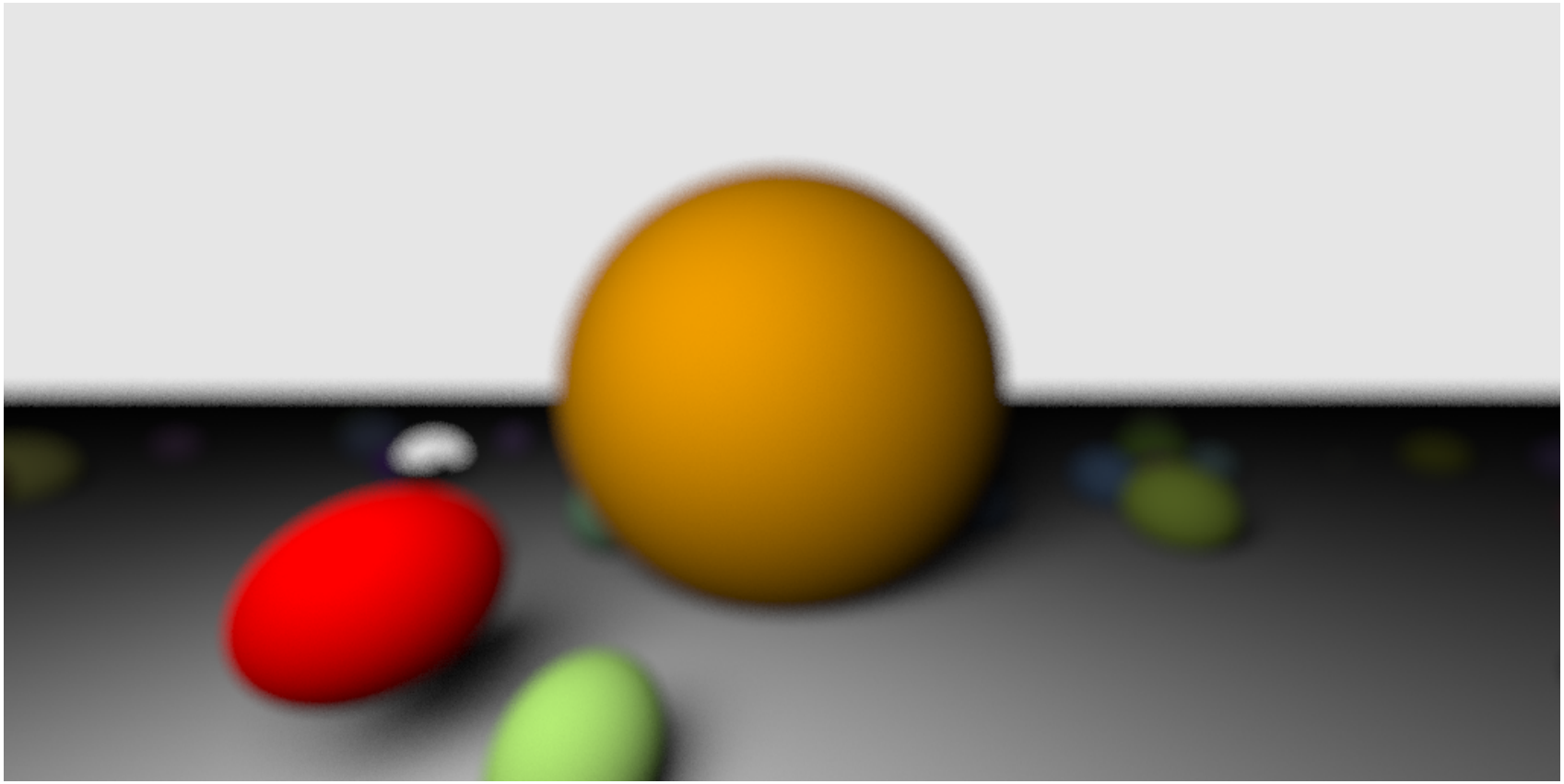
+ corriger les dimensions du plan image...

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> u;

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

// droite (o e) passant par le pixel (px py)
Point e= Point(x, y, z);
Vector d= Vector(o, e);
...
```

bonus : profondeur de champ / dof



bonus : profondeur de champ / dof