Rendus 3D

TIPE de Yann RUELLAN

Thème : Jeux, Sports

Jeux vidéos :

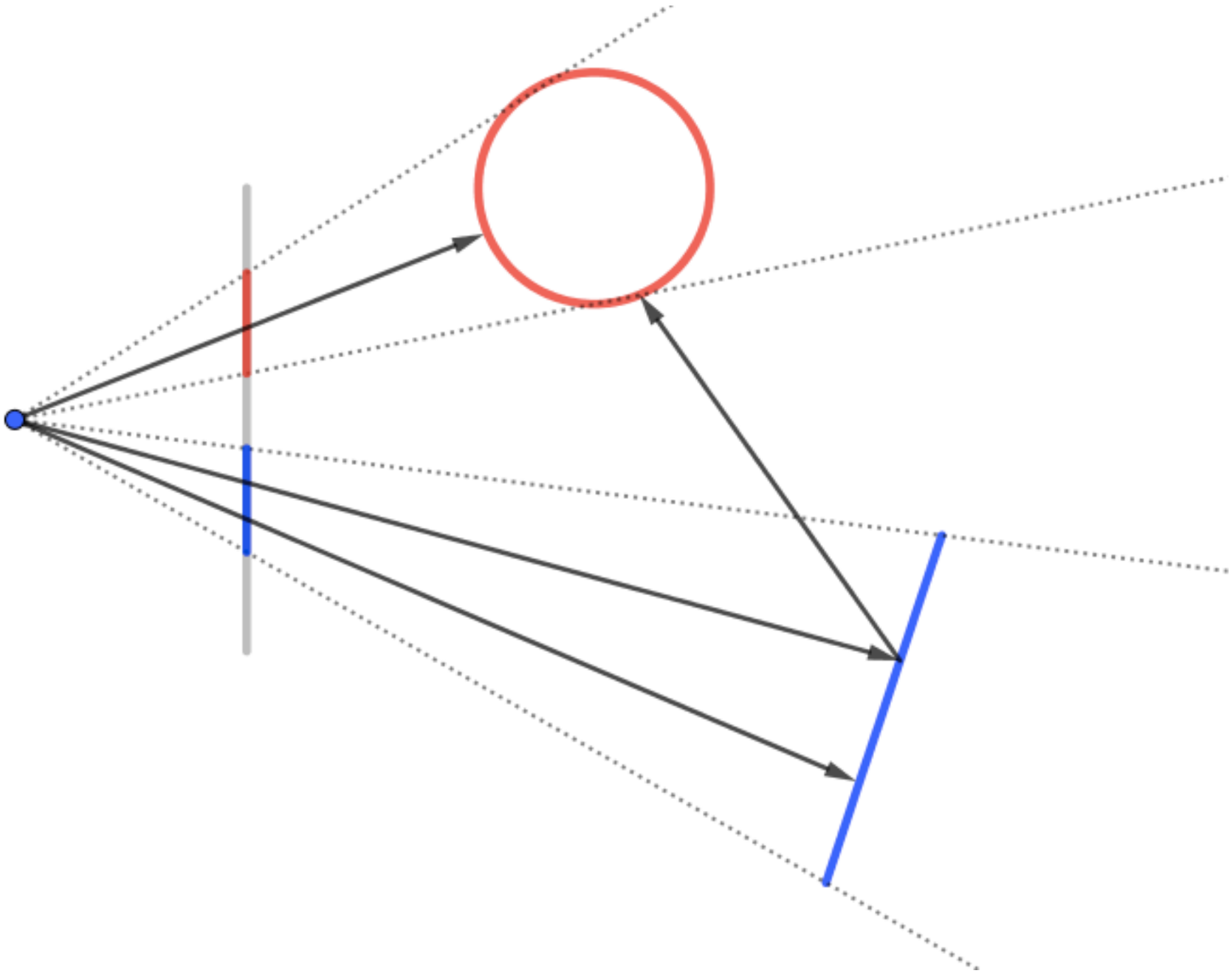
→ Affichage et synthèse d’image

Comment afficher des objets en trois dimensions sur un écran en deux dimensions ?

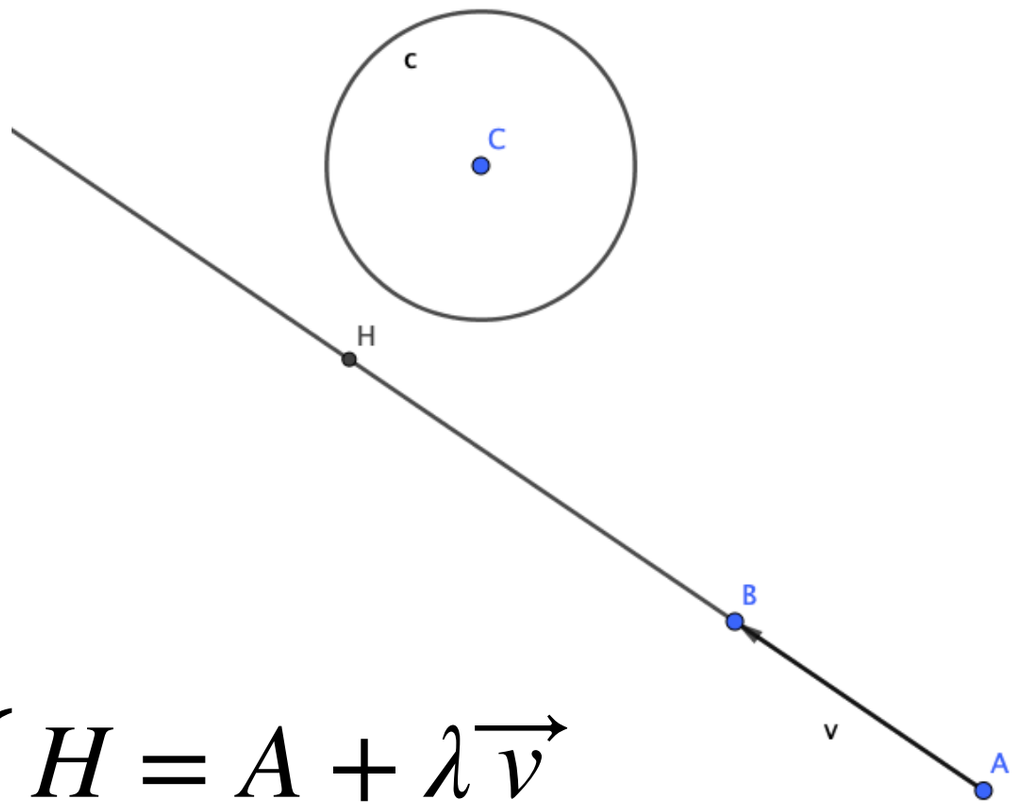
Plan

- Lancer de Rayon
- Géométrie
- Optique
- Programmation

Le lancer de rayons



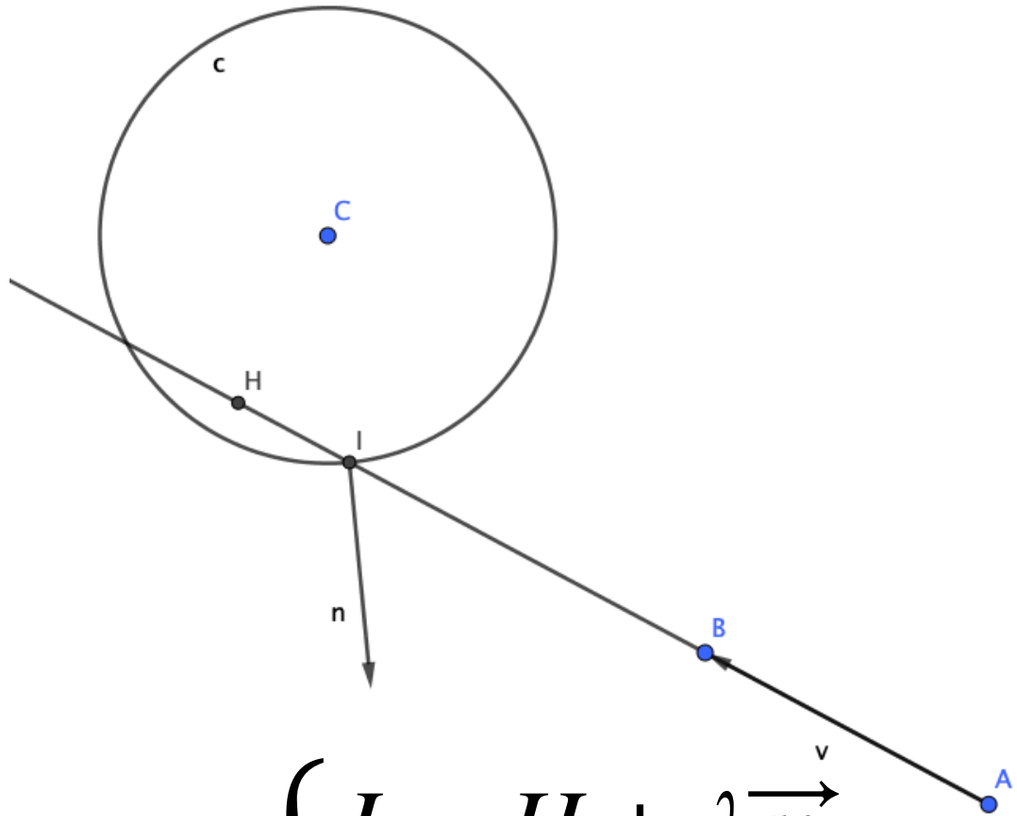
Géométrie
Intersection d'une Sphère



$$\begin{cases} H = A + \lambda \overrightarrow{v} \\ \overrightarrow{AH} \cdot \overrightarrow{v} = \overrightarrow{AC} \cdot \overrightarrow{v} \end{cases}$$

H est sur le rayon

H est à la même « distance » selon $\overrightarrow{v}$

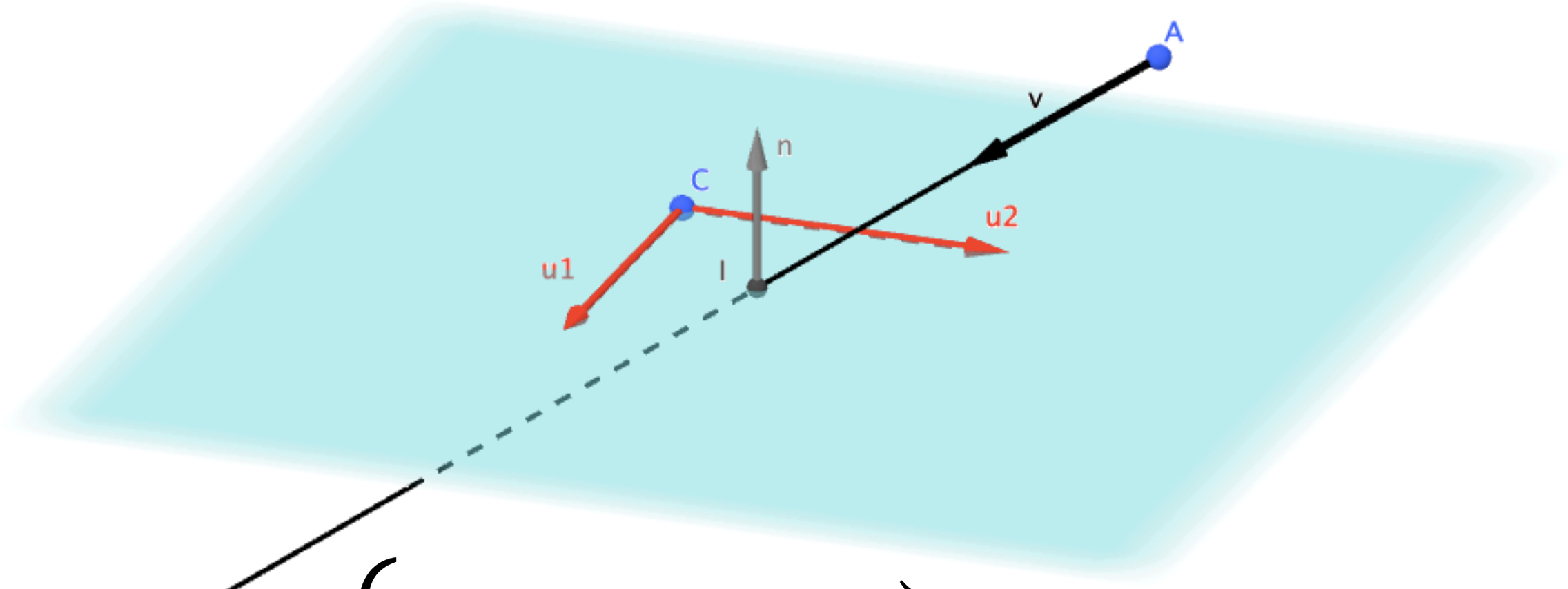


$$\begin{cases} I = H + \lambda \overrightarrow{v} \\ I \in \mathcal{S} \end{cases}$$

I est sur le rayon

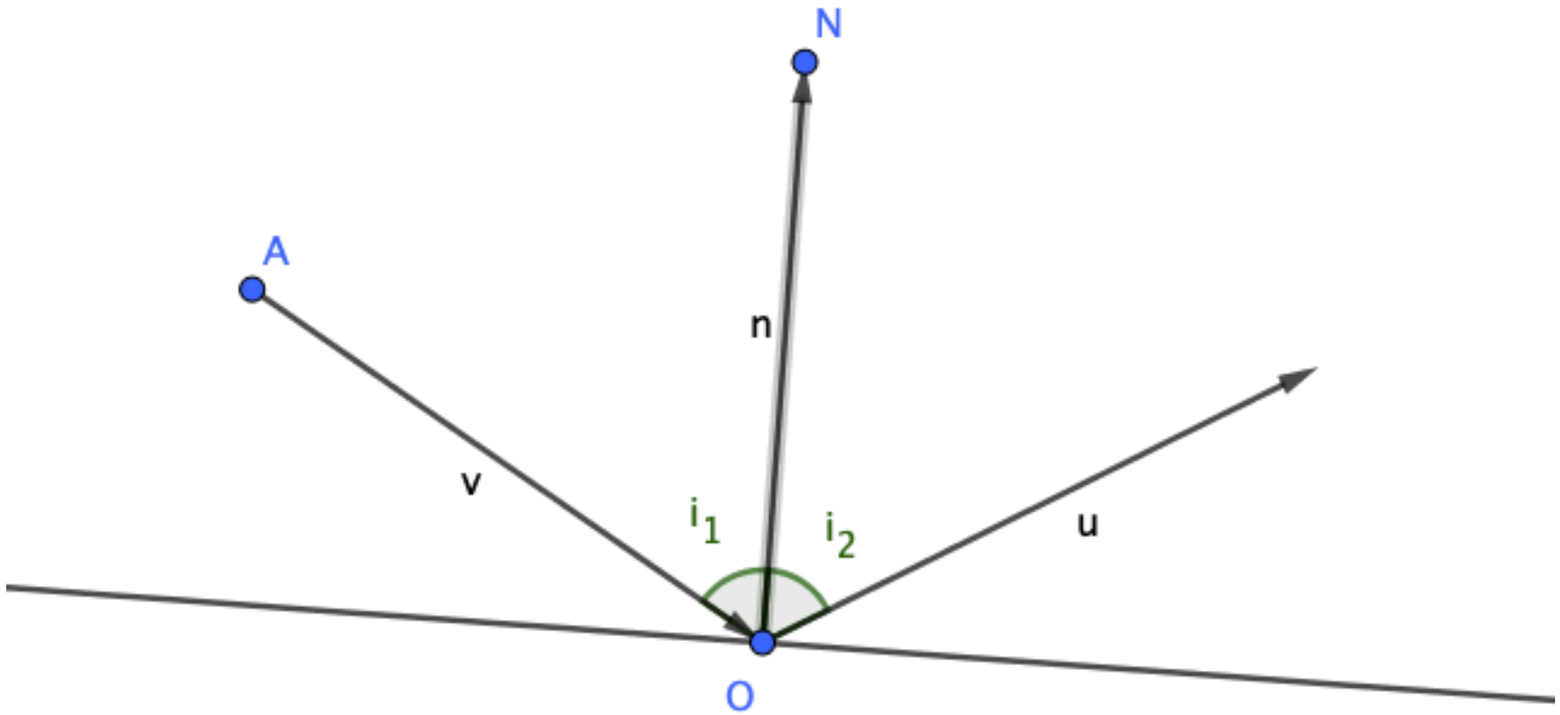
I est dans le cercle

Géométrie
Intersection d'un Plan



$$\begin{cases} I = A + \lambda \overrightarrow{v} & I \text{ est sur le rayon} \\ I \in \mathscr{P} & I \text{ est dans le plan} \end{cases}$$

Optique
Lois de Descartes : Réflexion

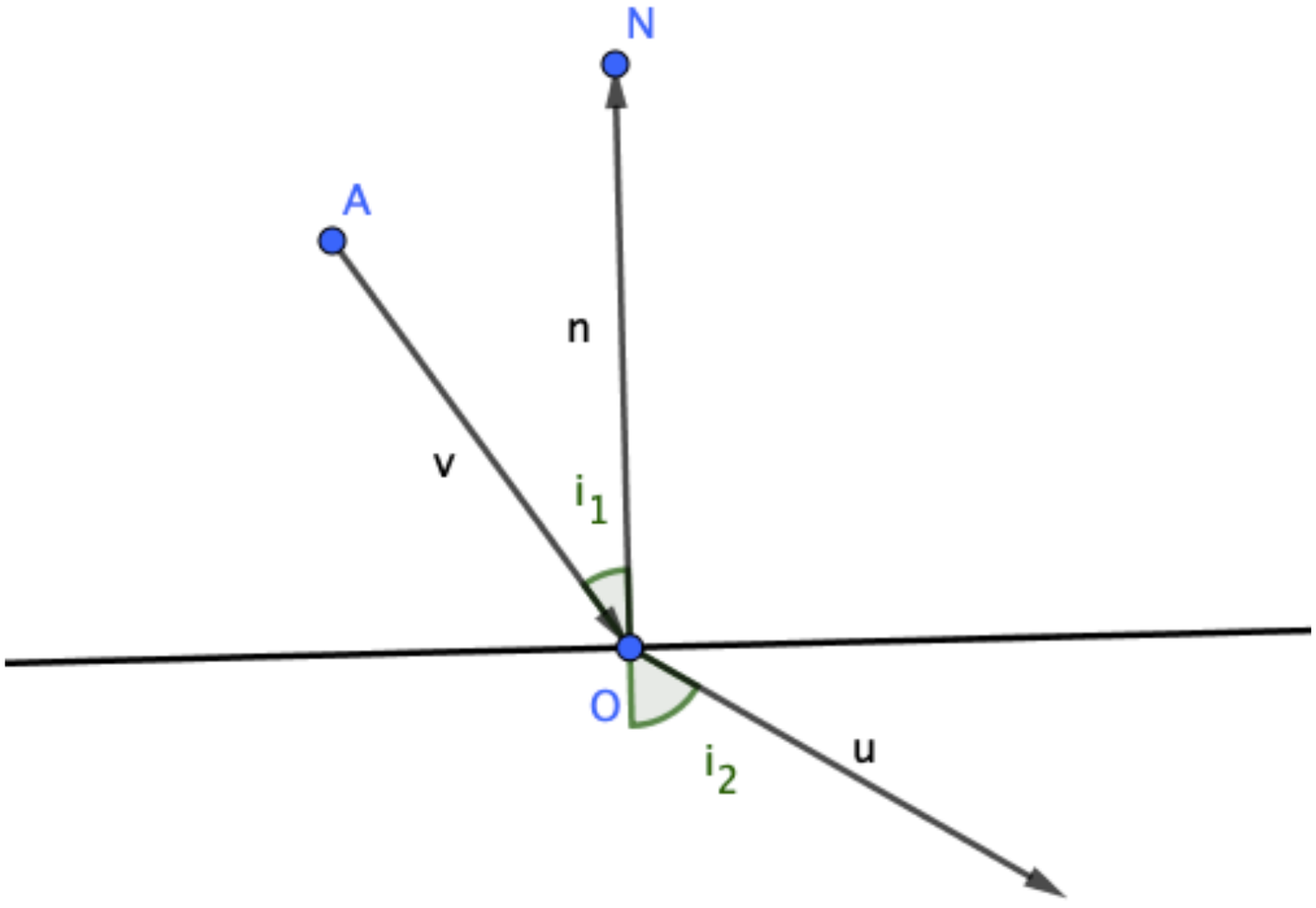


i_1 : angle du rayon incident

i_2 : angle du rayon réfléchi

$$i_1 = i_2$$

Optique
Lois de Descartes : Réfraction



r_1, r_2 : coefficients de réfraction des milieux

i_1 : angle du rayon incident

i_2 : angle du rayon réfléchi

$$r_1 \cdot \sin i_1 = r_2 \cdot \sin i_2$$

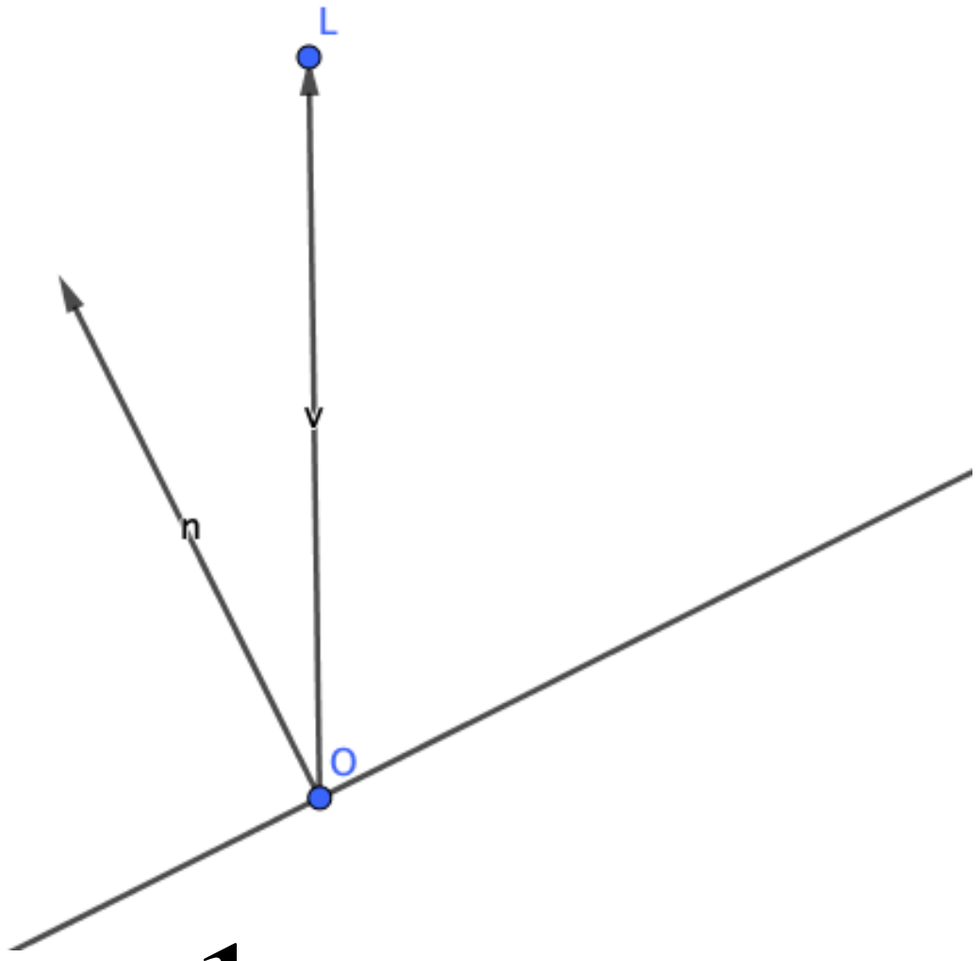
Optique
Luminosité

k : coefficients de luminosité

r : distance à la lumière

$\overrightarrow{n}$: normale à la surface

$\overrightarrow{u_{OL}}$: vecteur unitaire orienté vers la lumière



$$k \propto \frac{1}{r} \overrightarrow{n} \cdot \overrightarrow{u_{OL}}$$

Programmation
Langage

Python3

GLSL

Image 400x400 :
160 000 pixels

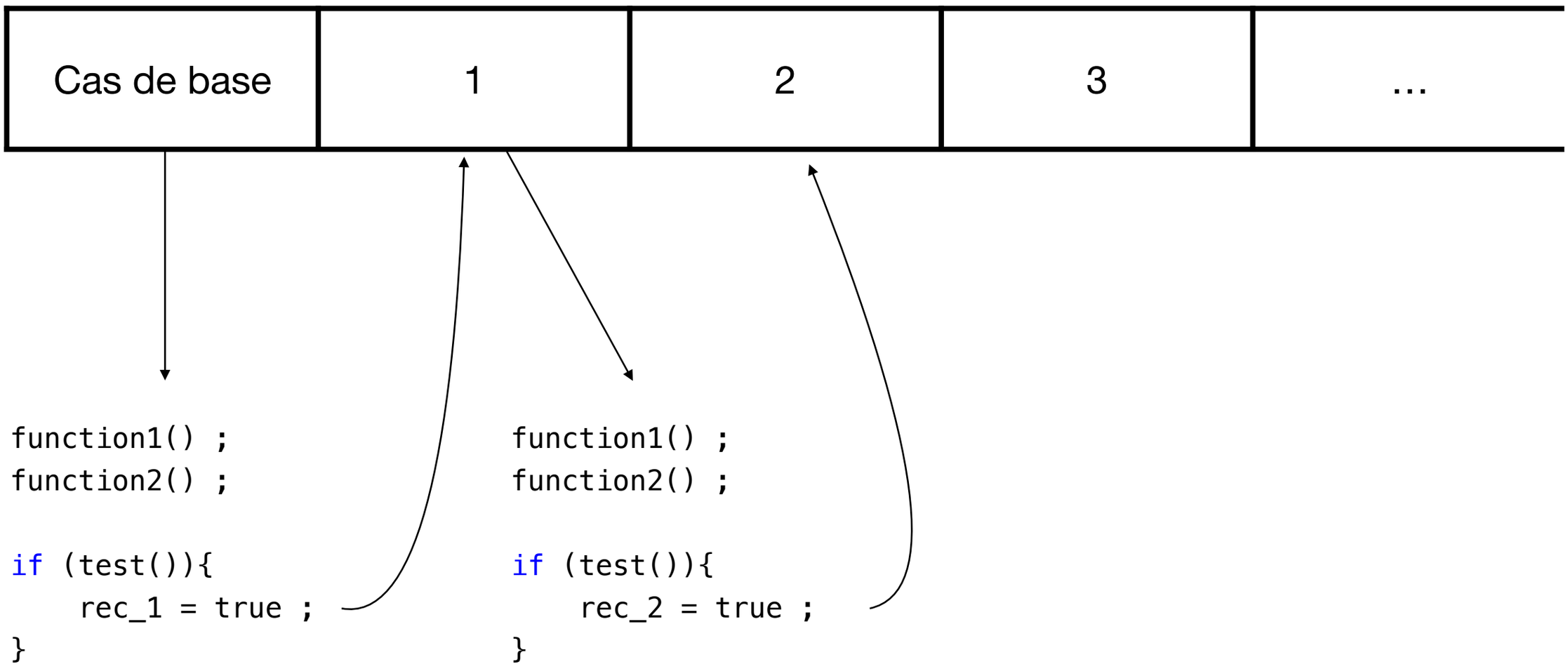
2 min

0.1 s

Programmation
Algorithme

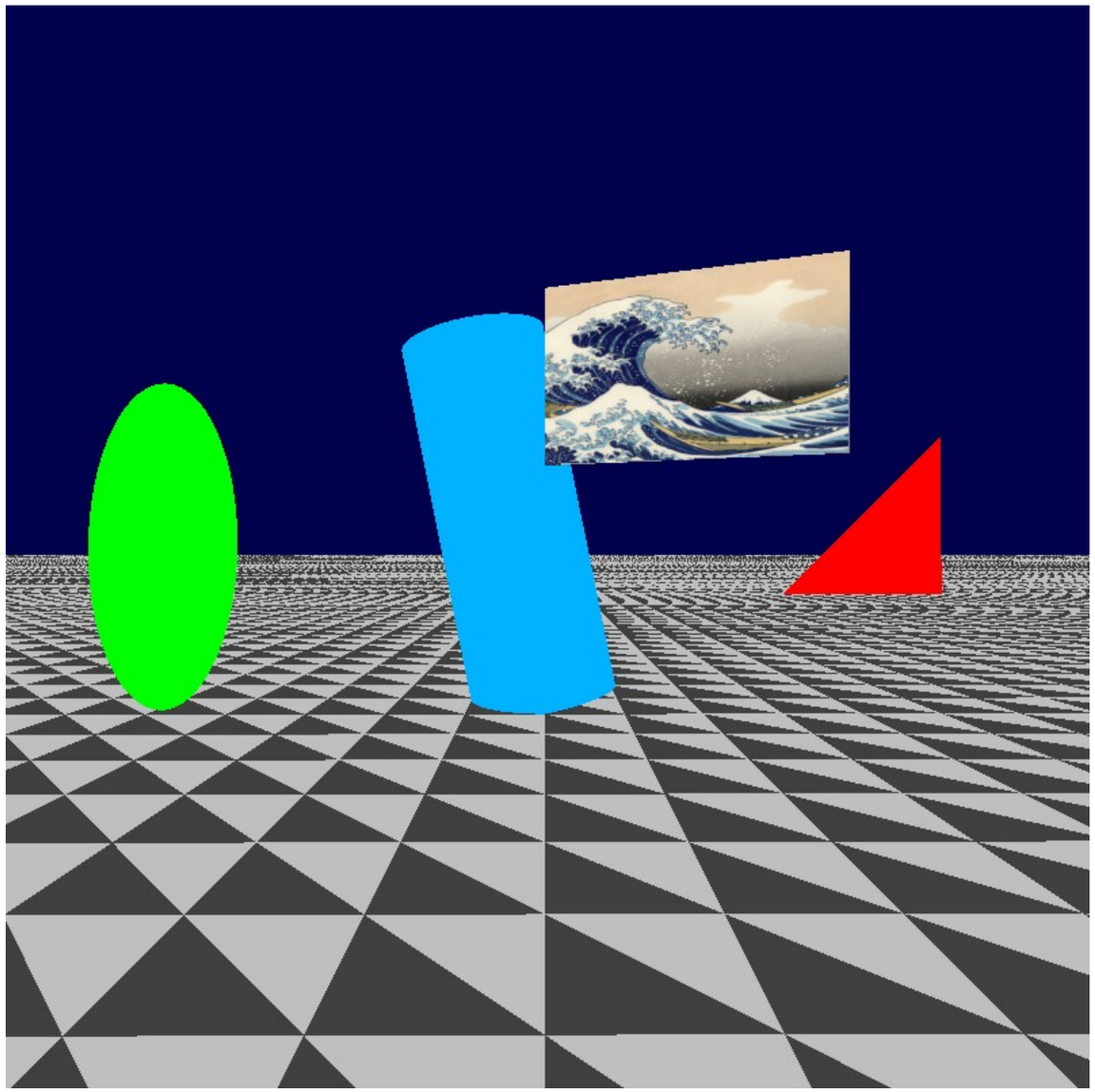
```
Calcule_couleur(R = None) :

    Si R == None:
        R = Rayon( Origin = camera, Direction = pixel)

    Obj = Intersection_la_plus_proche(R, Objects_list)

    Si réflexion :
        R = rayon réfléchi
        couleur_réfléchie = Calcule_couleur(R)

    Si réfraction :
        R = rayon réfracté
        couleur_réfractée = Calcule_couleur(R)


renvoyer Obj.couleur +  couleur_réfléchie + couleur_réfractée
```

Programmation
Récursivité

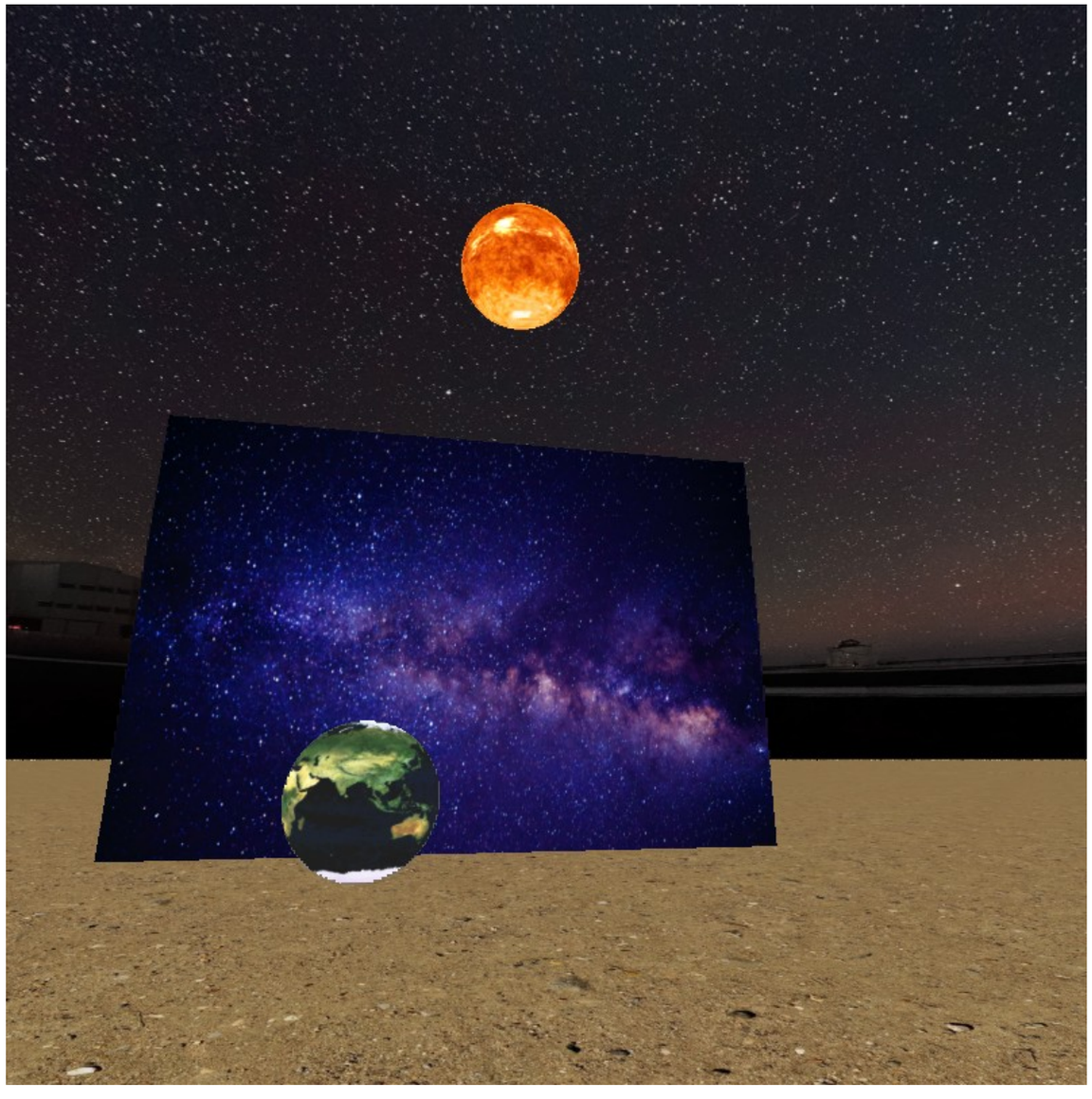


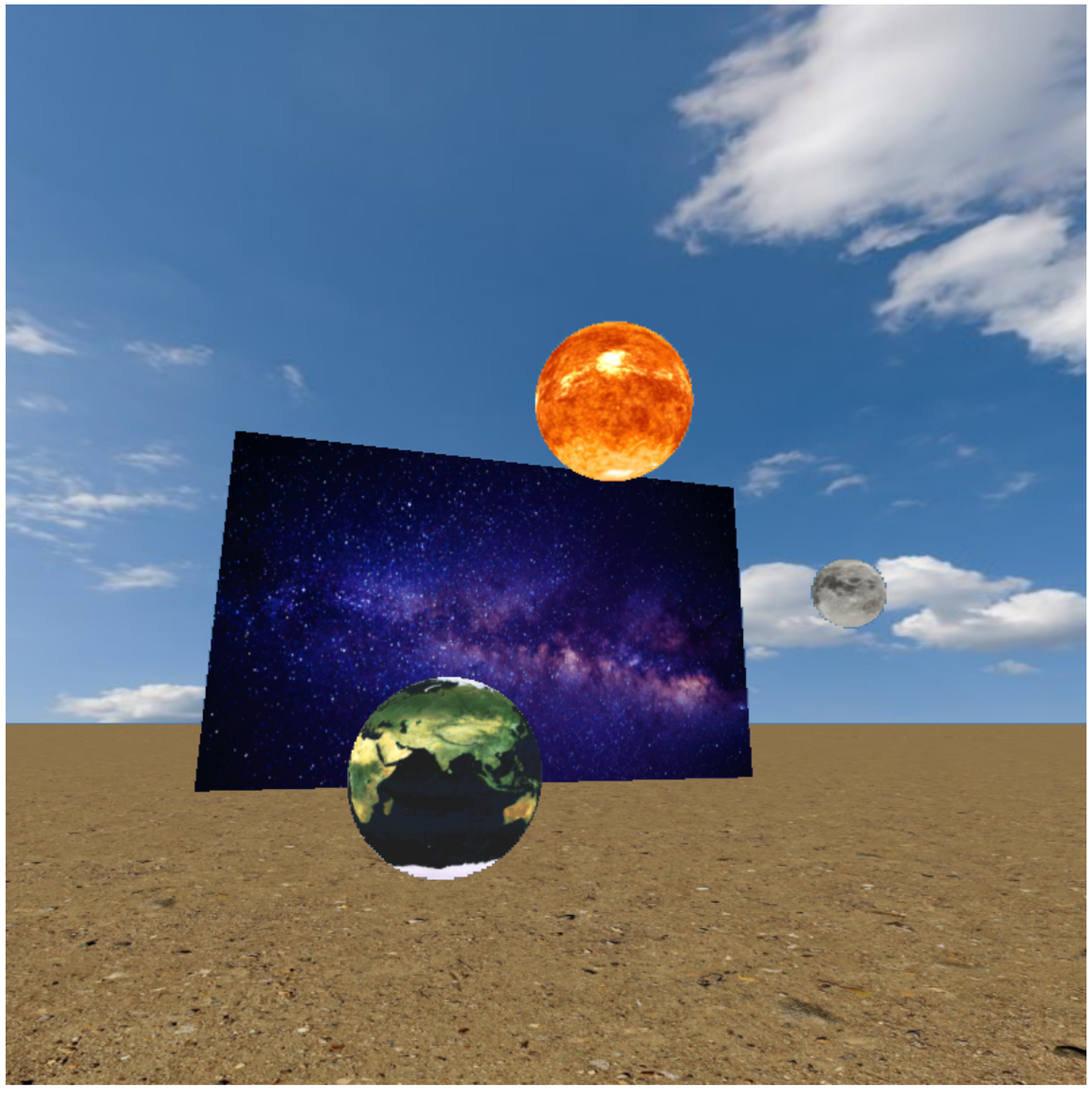
Résultat
Divers Objets

- Disque Vert
- Cylindre Bleu
- Rectangle Image
- Triangle Rouge
- Plancher

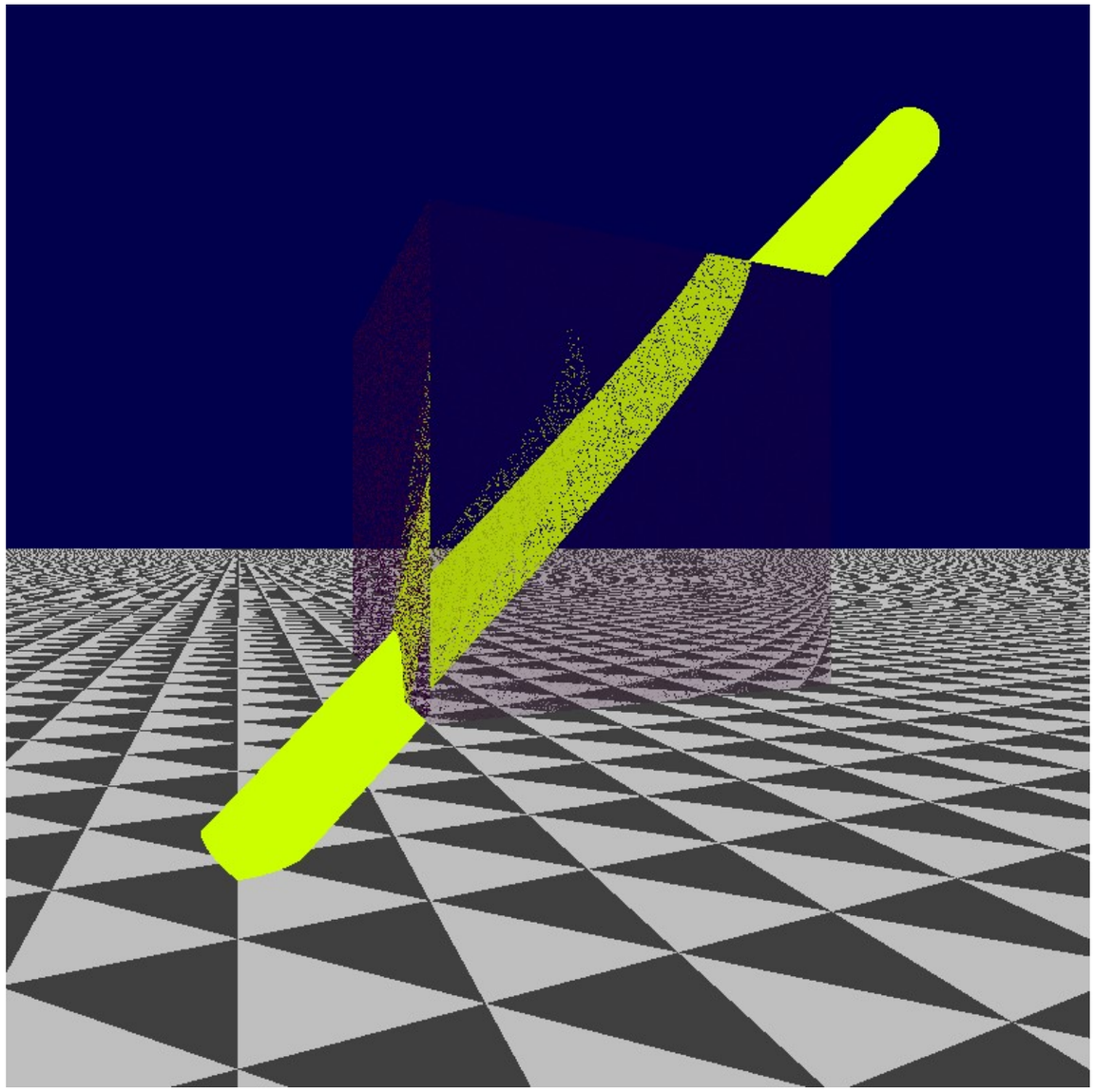


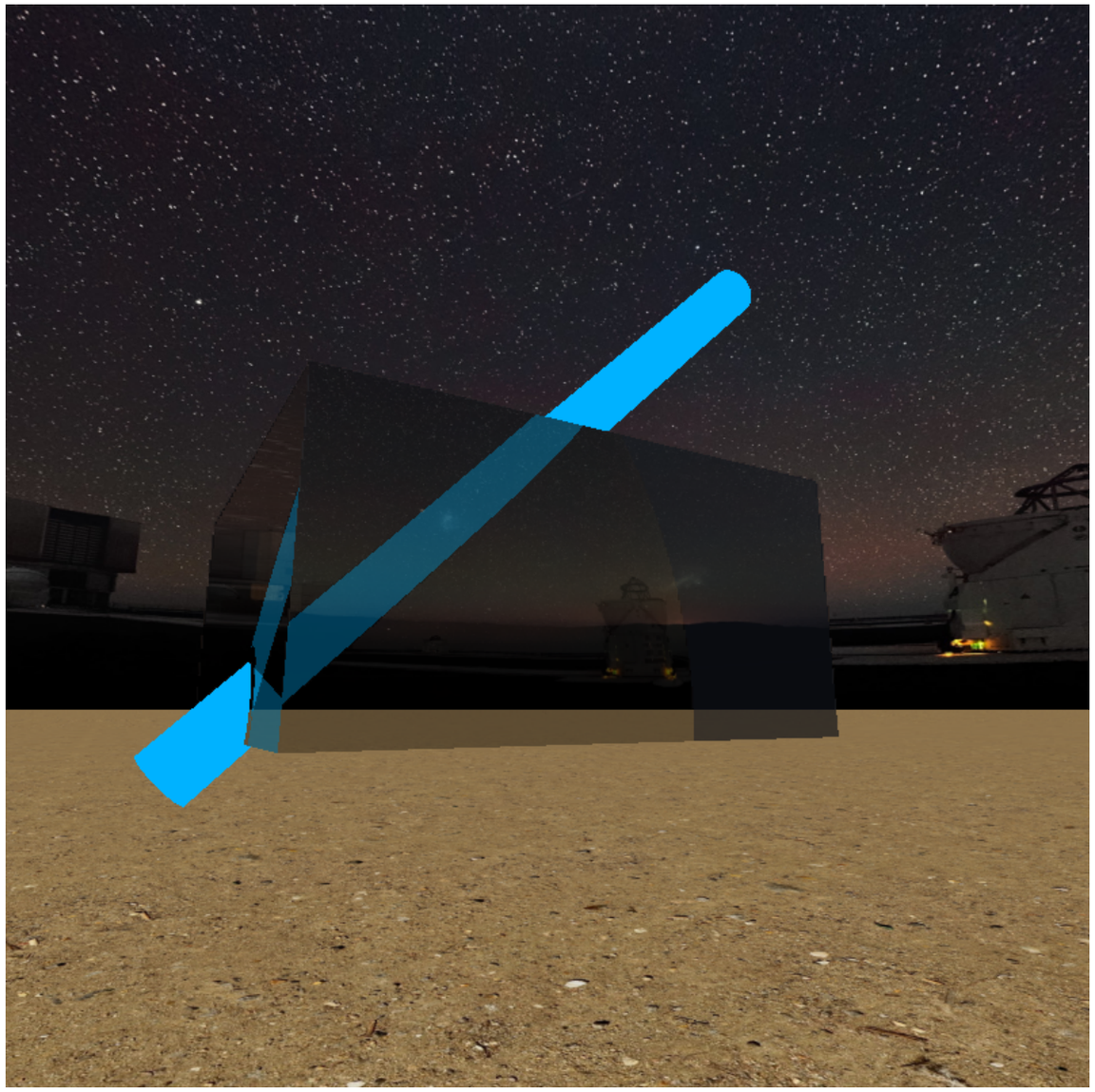
Résultat
Textures



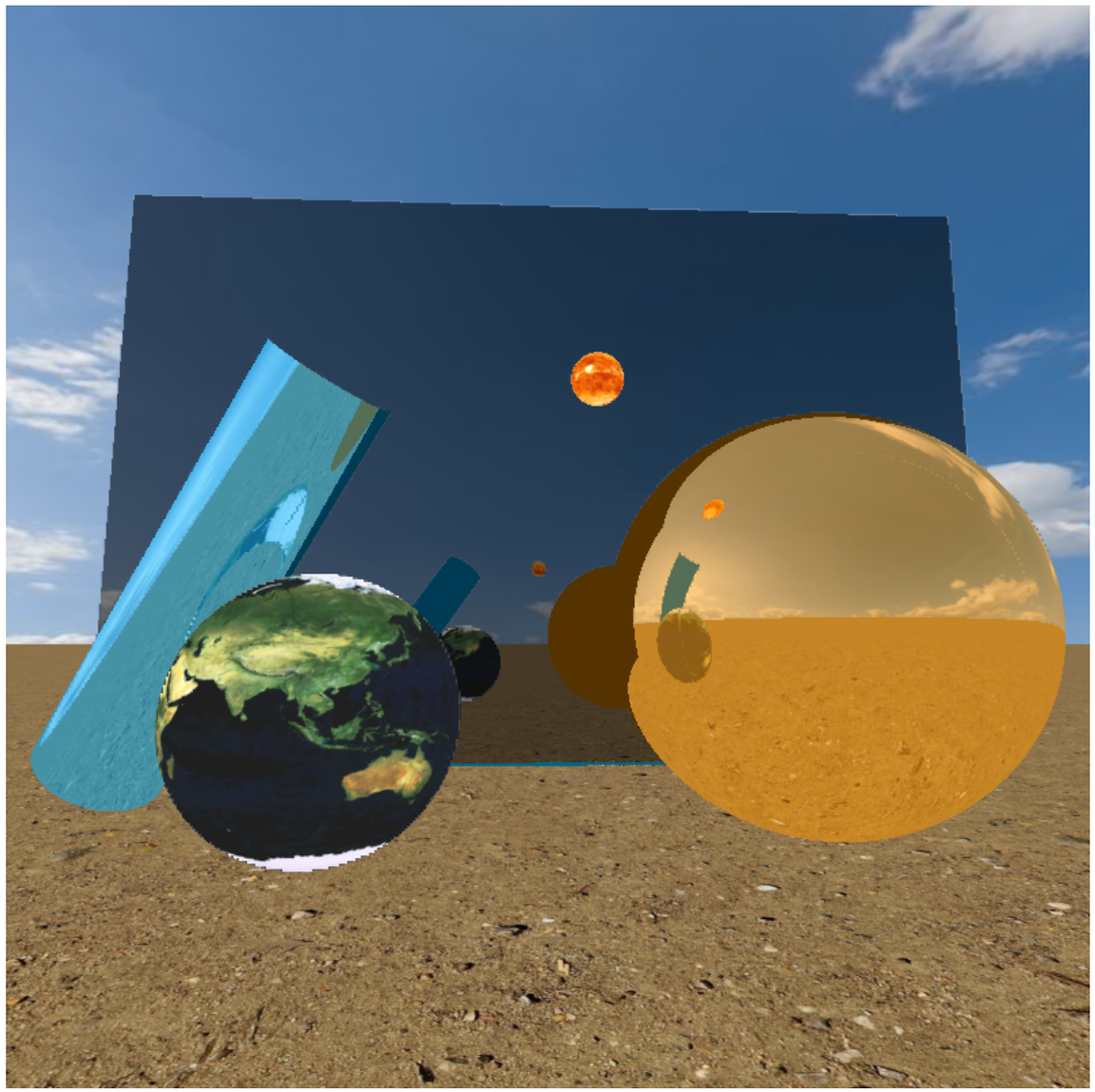


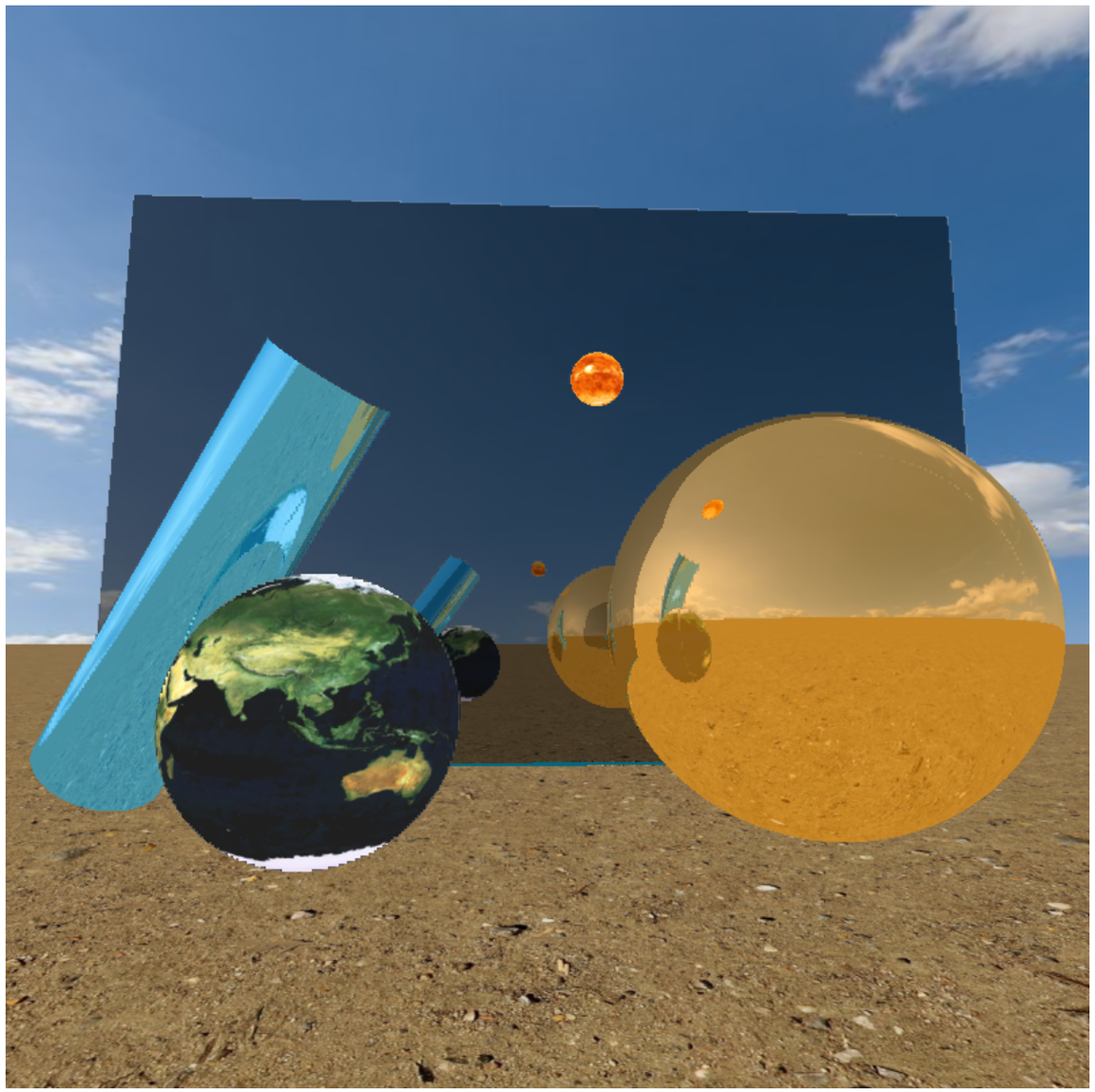
Résultat
Réfraction



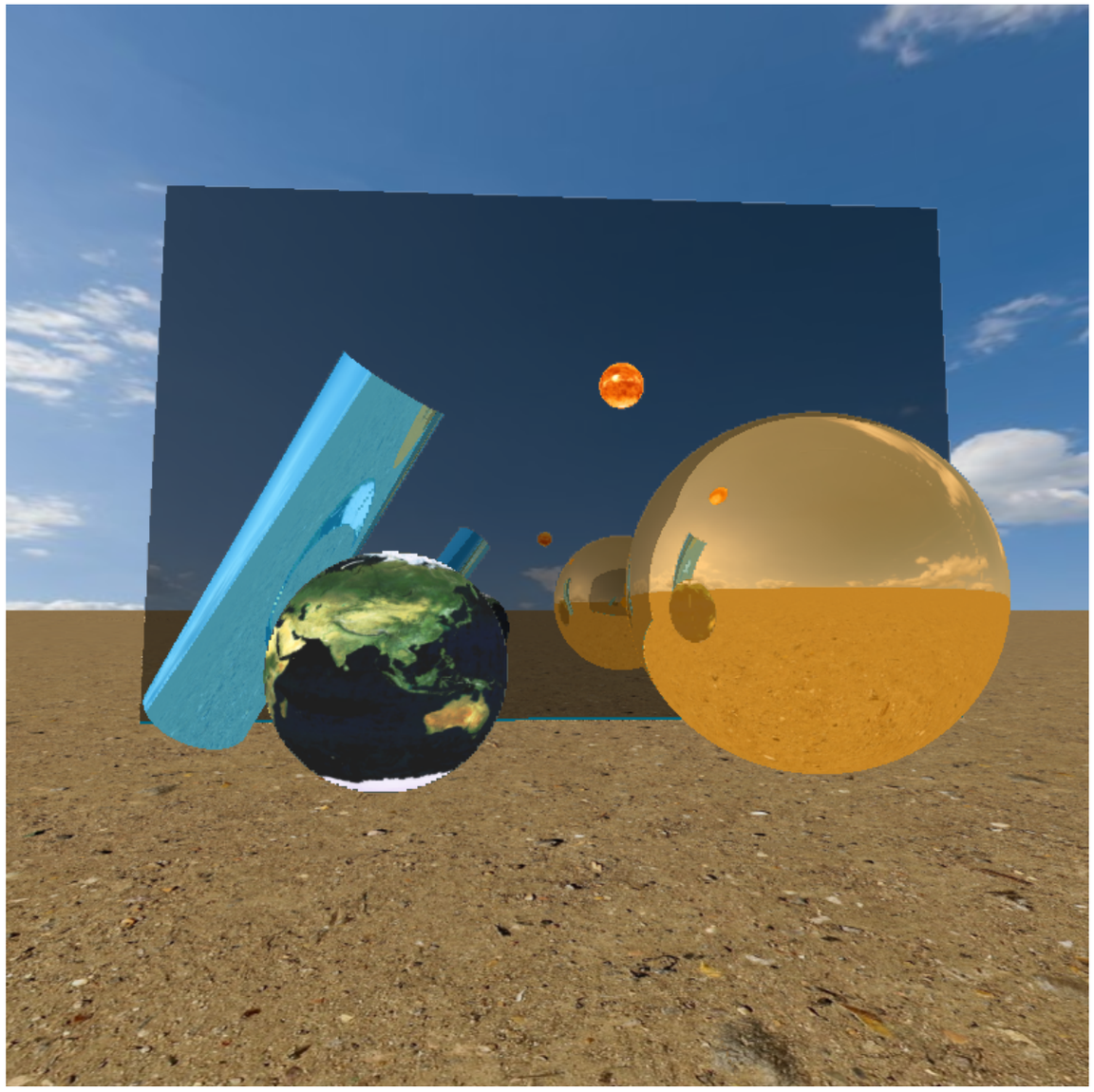


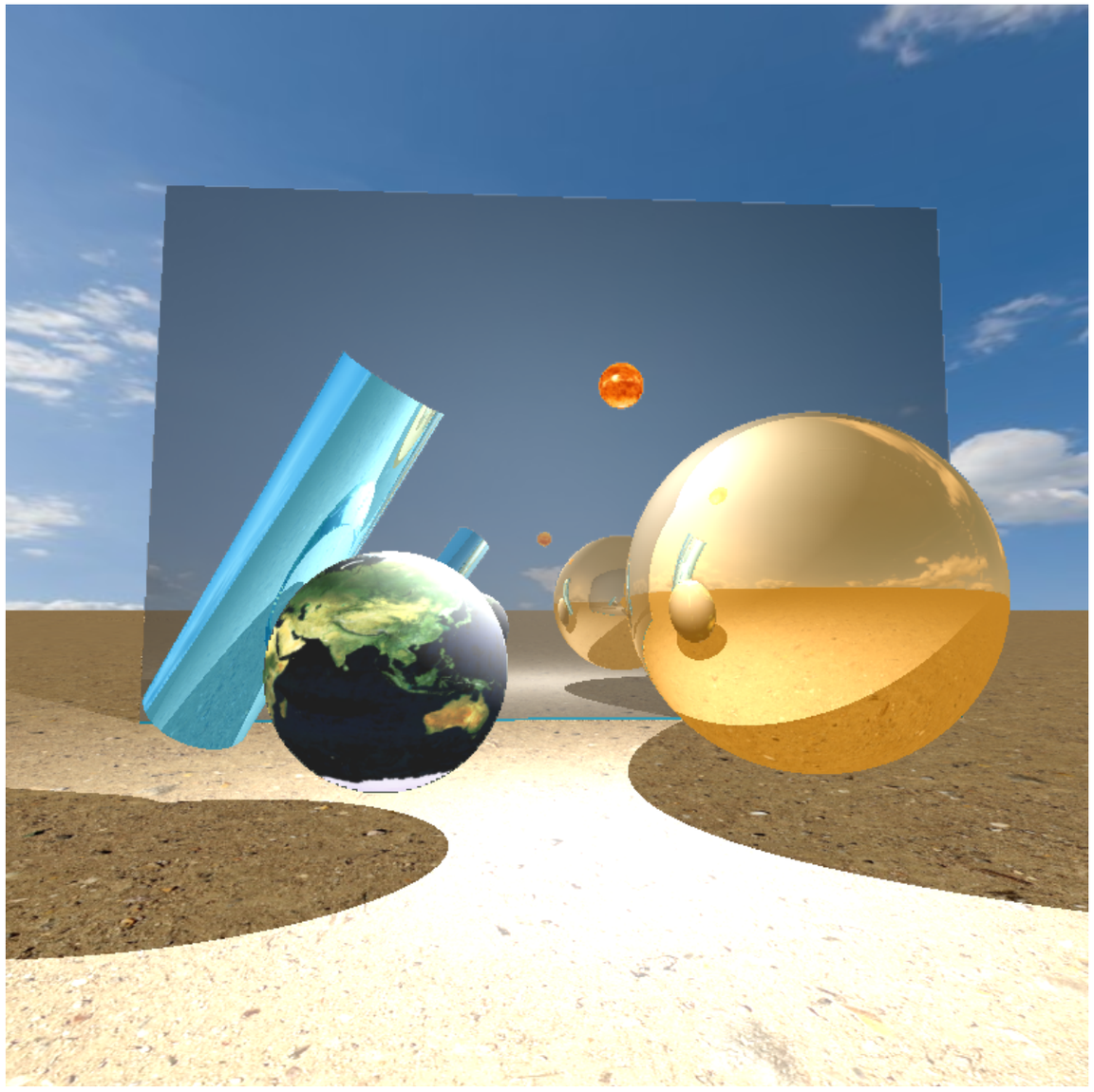
Résultat
Miroirs



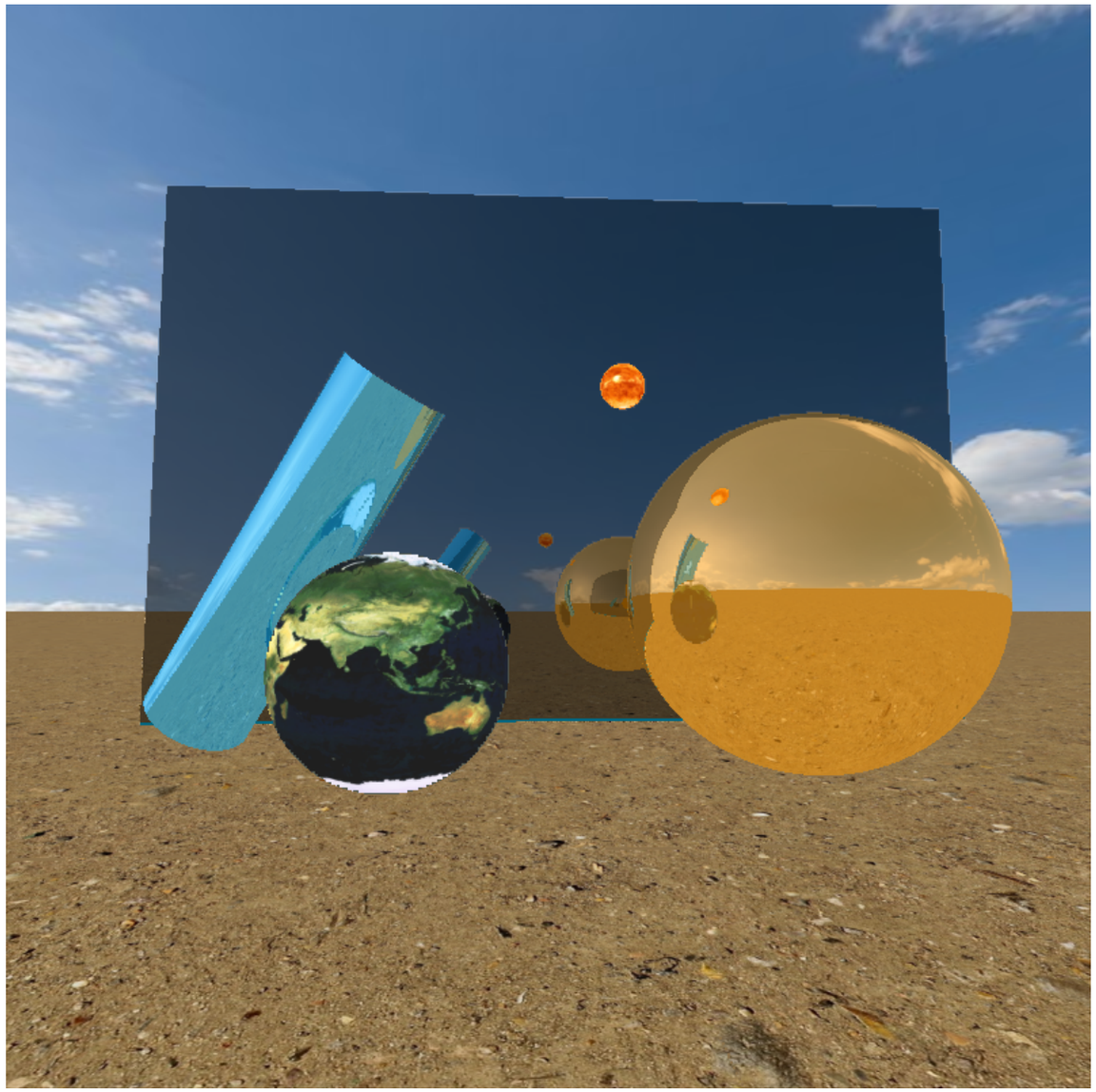


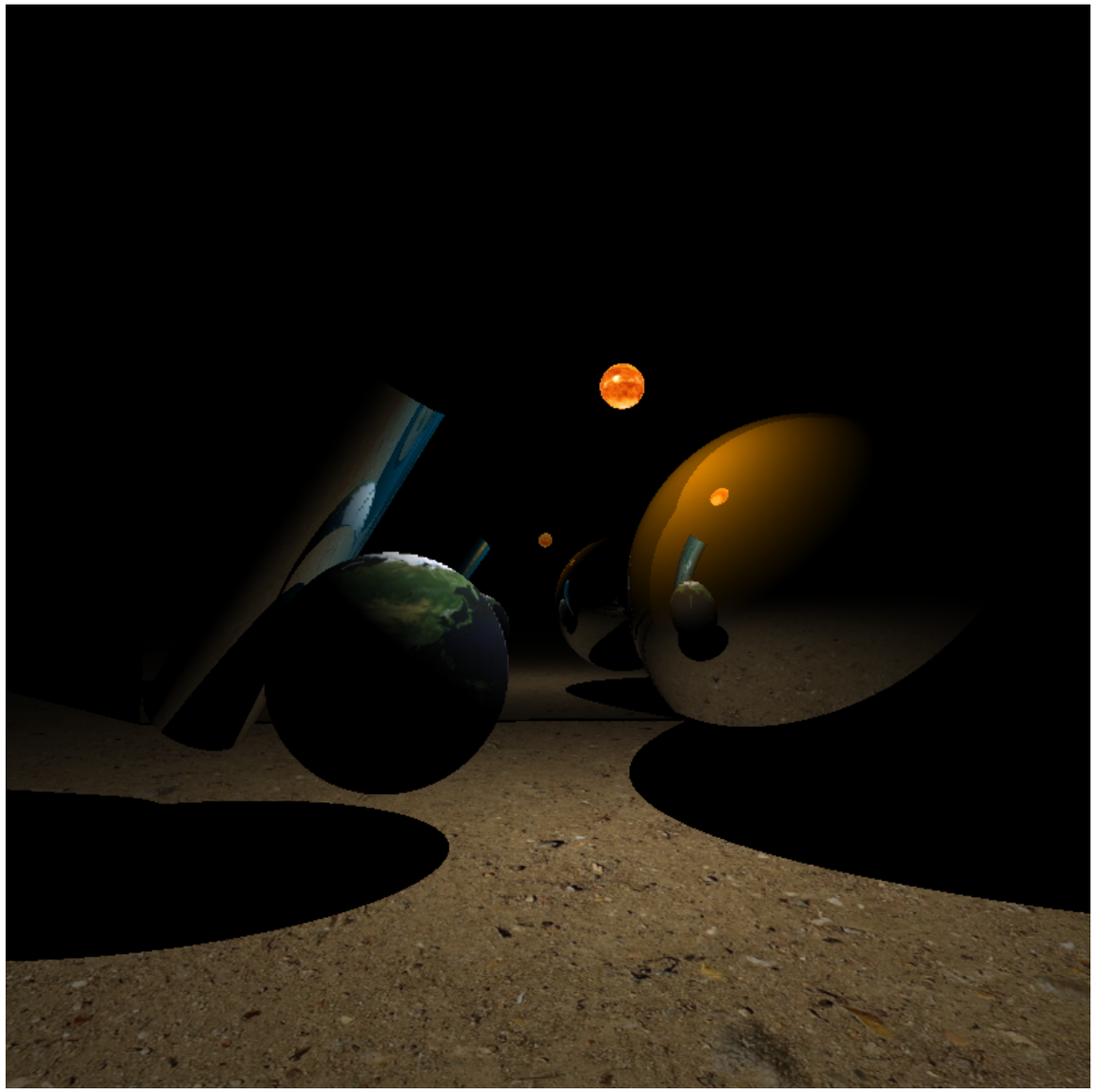
Résultat
Lumière



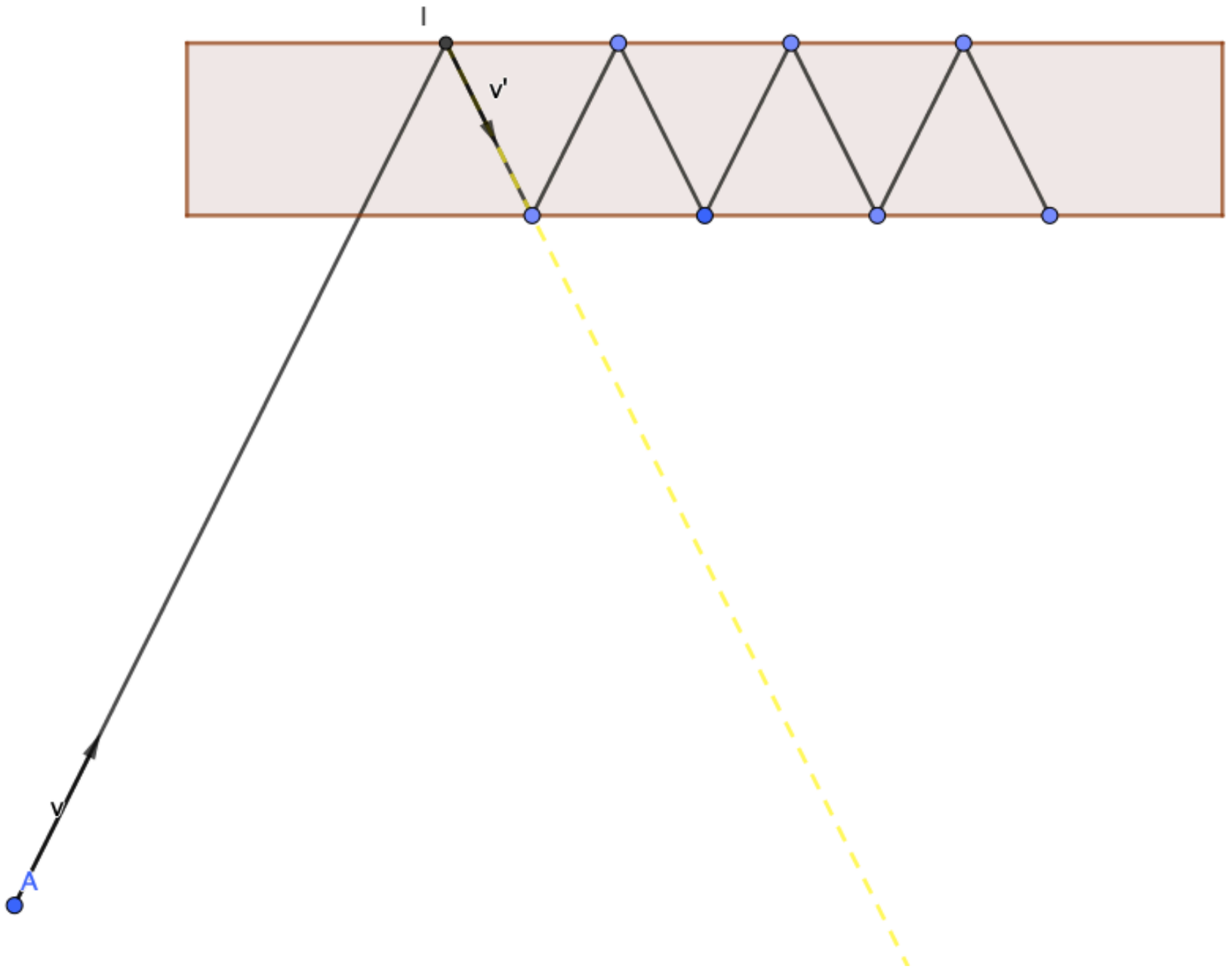


Résultat
Lumière

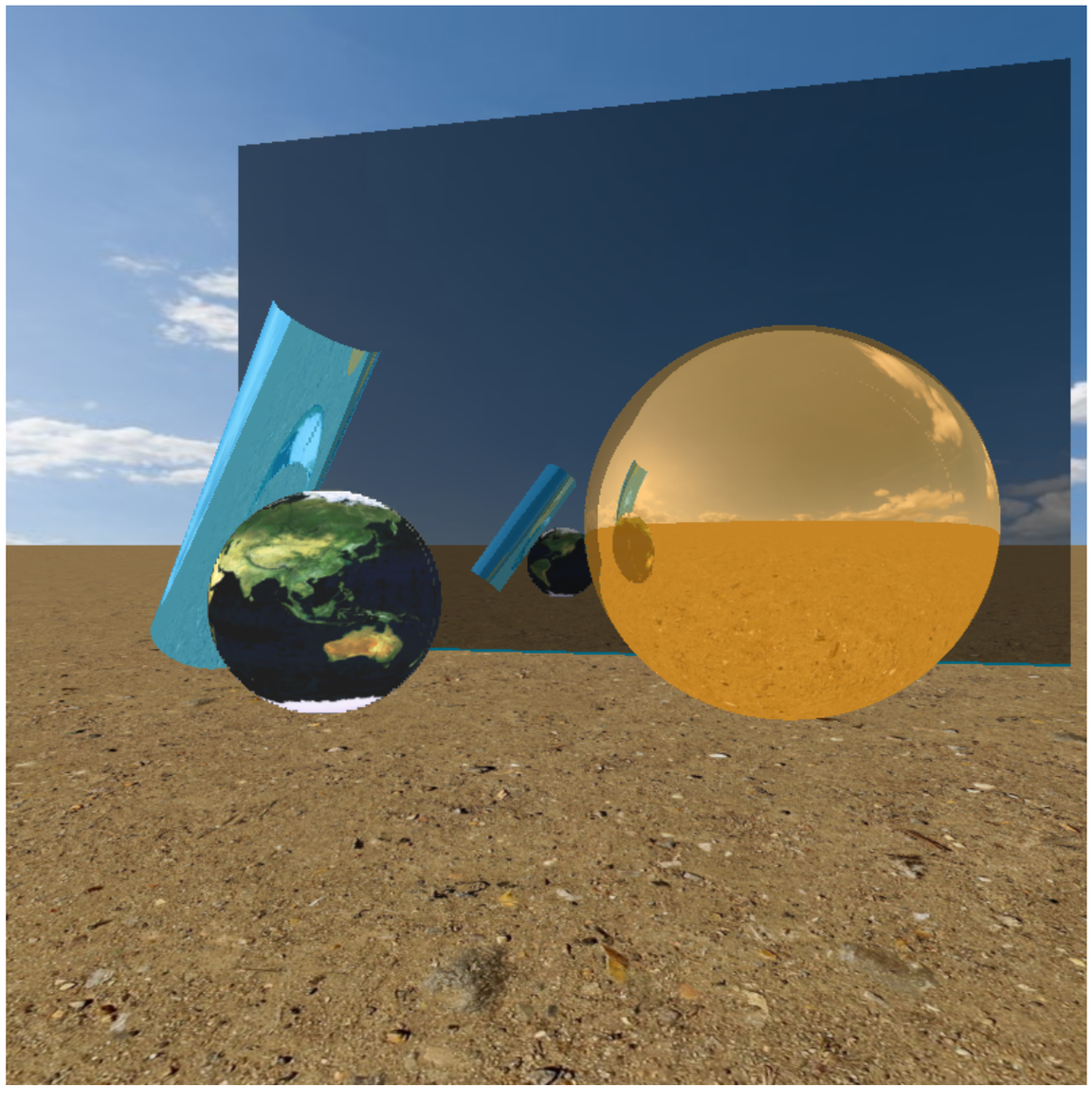


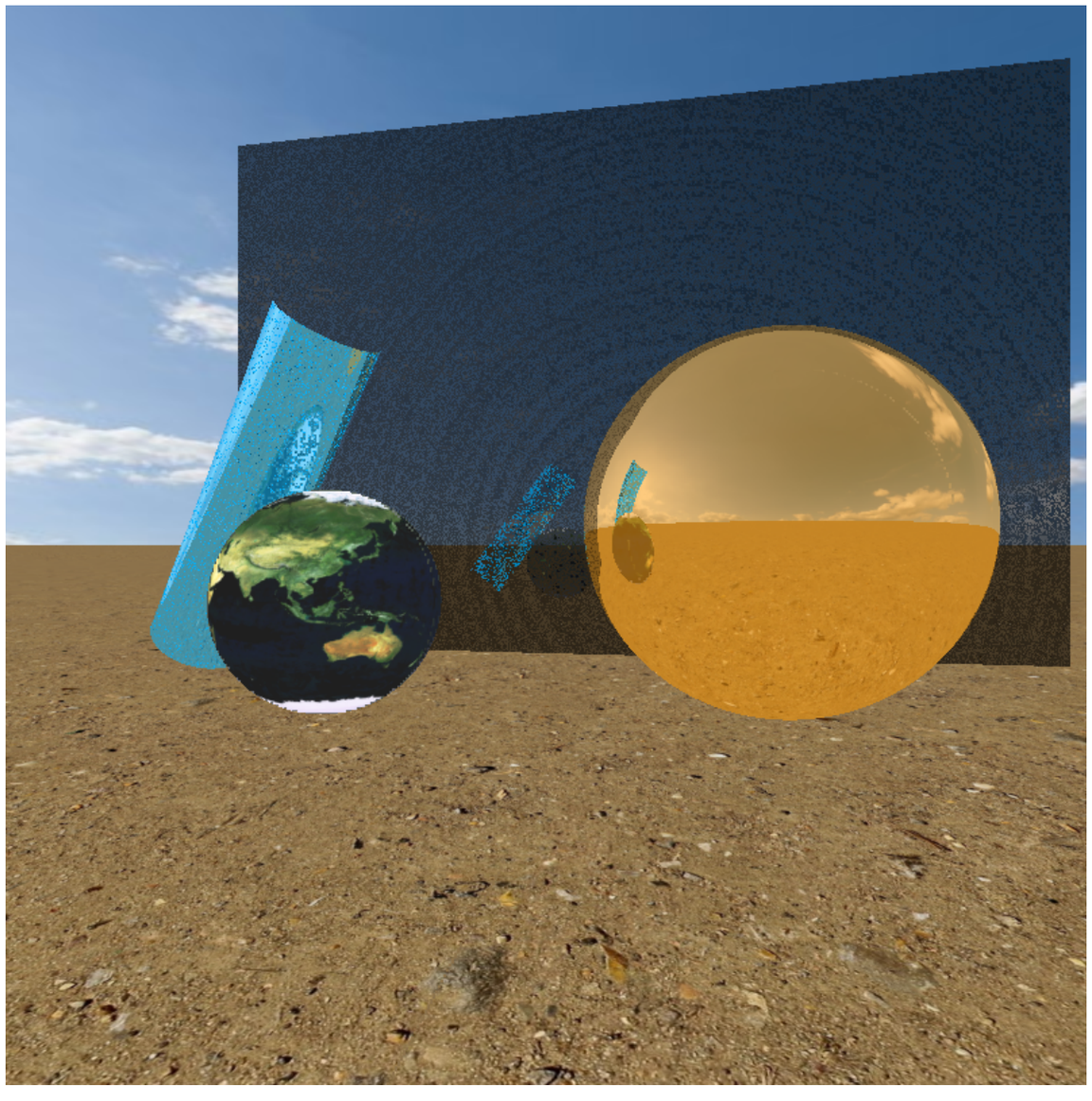


Amélioration
Miroir

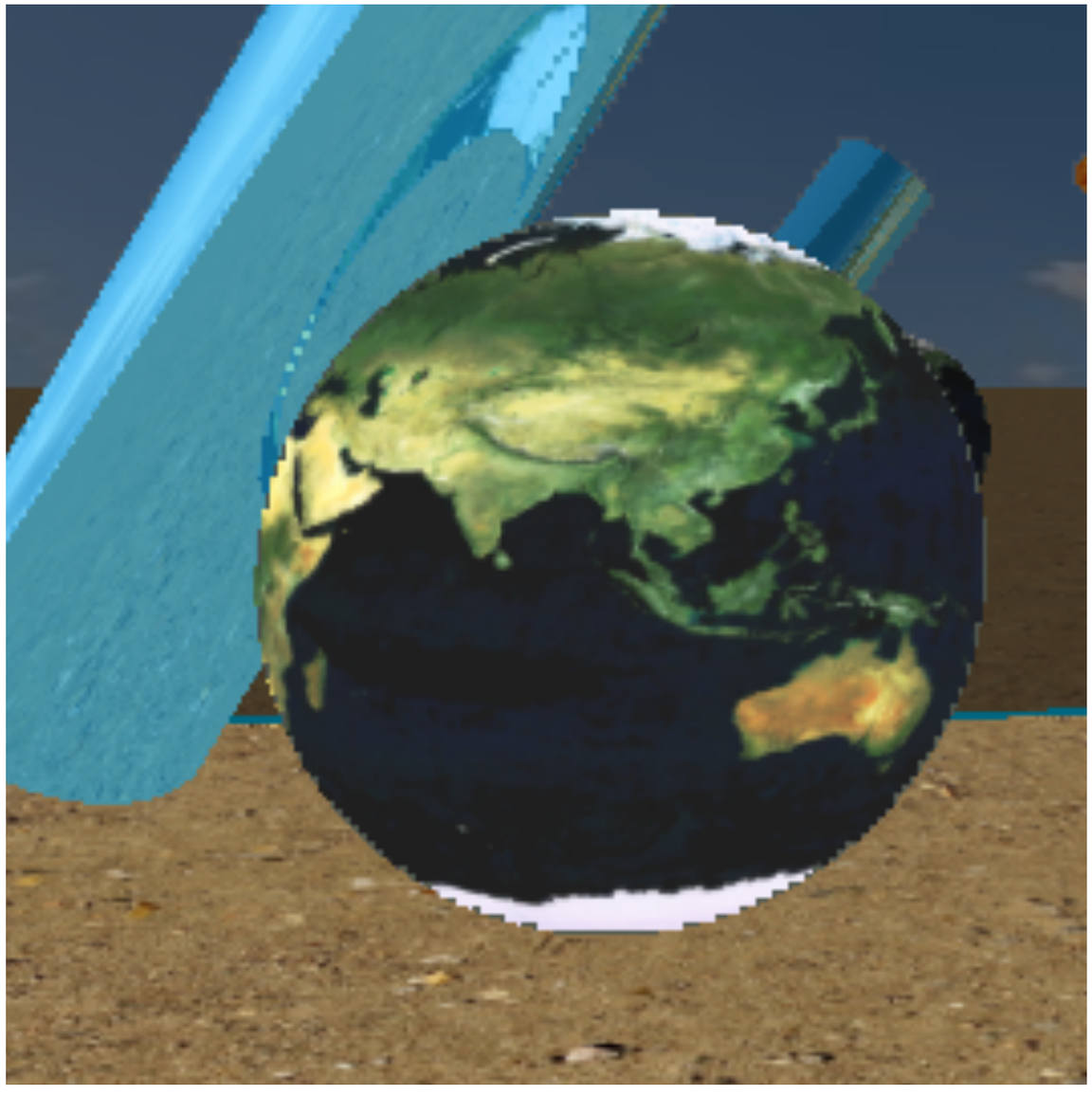


Amélioration
Miroir





Amélioration
Effets de bord



Statistiques

Shader (synthèse d'image):
- 900 lignes de code
- 25 000 caractères

TIPE.pde (interface) :
- 200 lignes de code
- 4 000 caractères

Programme :
- 31 Mo de code

« Toute science commence comme philosophie et se termine en art »

-William James Durant

Constantes

```
precision highp float;

uniform sampler2D hokusai;
uniform sampler2D sky;
uniform sampler2D earth;
uniform sampler2D ground;
uniform sampler2D sun;
uniform sampler2D galaxy;

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;
uniform vec2 position;
uniform float light_height;
uniform int light_type; // 0: no light, 1:
light+ambiente, 2: only light
uniform bool draw_type ;

#define PI
3.14159265358979323846264338327795
#define TWO_PI
6.28318530717958647692528676655900
```

```
#define COLOR_HOKUSAI vec3(0.0, 0.0, 0.01)
#define COLOR_EARTH vec3(0.0, 0.0, 0.02)
#define COLOR_SKY vec3(0.0, 0.0, 0.03)
#define COLOR_GROUND vec3(0, 0, 0.04)
#define COLOR_SUN vec3(0, 0, 0.05)
#define COLOR_GALAXY vec3(0, 0, 0.06)


#define TYPE_ERROR -1
#define TYPE_SPHERE 0
#define TYPE_CYLINDER 1
#define TYPE_CYLINDER_FULL 2
#define TYPE_PLANE 3
#define TYPE_RECTANGLE 4
#define TYPE_CIRCLE 5
#define TYPE_TRIANGLE 6
```

Structures

```
struct Line {
    vec3 origin ;
    vec3 v ;
};

struct Sphere {
    vec3 center ;
    float radius ;
};

struct Cylinder {
    vec3 origin ;
    vec3 v ;
    float radius ;
};
```

```
struct Plane {
    vec3 origin ;
    vec3 u1 ;
    vec3 u2 ;
};


struct Object {
    Sphere sphere ;
    Plane plane ;
    Cylinder cylinder ;
    int type ;
    vec3 color ;
    float mirror ;
    float refrac ;
    bool is_light ;
};
```

Lois d'Optique

```
vec3 reflexion(vec3 v, vec3 n){
    return v - 2.0 * dot(v,n) / dot(n,n) * n ;
}

vec3 refraction(vec3 v_, vec3 n, float r){
    vec3 v = normalize(v_);

    float a = dot(n,n) ;
    float b = r * dot(n,v) ;
    float c = r * r * dot(v,v) - 1 ;

    if (b * b - a * c < 0) return vec3(0.);
    // if (b * b - a * c < 0) return reflexion(v,n);
    float d = sqrt(b * b - a * c);
    if ( dot(v,n) < 0) d*= -1 ;
    float k = (- b + d ) / a ;

    return k * n + r * v ;
}
```

Coefficients Optiques

```
vec2 taux_ref(vec3 u, vec3 v_t, vec3 n, float r){
    // http://www.joelsornette.fr/Archives/exotypes/exotype26.pdf
    // u Light vector, v_t transmitted vector, n normal vector, r quotient of refraction

    float n1 = 1 ;
    float n2 = r * n1 ;

    float i1 = dot(u,n) ;
    float i2 = dot(v_t,-n) ;

    float R_1 = n2 * i1 - n1 * i2 ;
    float R_2 = n2 * i1 + n1 * i2 ;
    float R_3 = R_1 / R_2 ;
    float R = R_3 * R_3 ;

    if (R_2 == 0.0) return vec2(1,0);

    float T_1 = 4 * n1 * n2 * i1 * i2;
    float T_2 = n2 * i1 + n1 * i2 ;
    float T = T_1 / T_2 ;

    return vec2(T,R);
}
```

Intersection d'une Sphère

```
Line get_intersection(Line L, Sphere S){
    // Return the intersection and the normal of the sphere
    float d = dot(S.center - L.origin, L.v) ;
    if (d <= 0.0) return Line(vec3(0.),vec3(0.));
    vec3 H = L.origin + d / norm2(L.v) * L.v;
    vec3 u = H - S.center;

    float l = S.radius * S.radius - dot(u,u);

    // H isnt in the sphere
    if (l < 0.) return Line(H,vec3(0.));

    // H is in the sphere
    // We calculate the intersection based on H

    vec3 I = H ;
    if (d - sqrt(l) < 0) I += sqrt(l) * normalize(L.v);
    else I -= sqrt(l) * normalize(L.v);

    vec3 n = I - S.center;

    return Line(I, n) ;
}
```

Intersection d’un Cylindre

```
Line get_intersection(Line L, Cylinder C){

    vec3 v1 = cross(L.v,C.v);
    vec3 v2 = cross(L.origin - C.origin, C.v);


    float a = norm2(v1) ;
    float b = dot( v1, v2 );
    float c = norm2( v2 ) - C.radius *
C.radius * norm2(C.v) ;

    float d = b*b - a*c ;
    float lambda ;

    if (d < 0.)
        return Line(vec3(0.),vec3(0.));
    else if (d == 0.0) lambda = - b / a ;
    else {
        float lambda1 = (- b + sqrt(d)) / a ;
        float lambda2 = (- b - sqrt(d)) / a ;
        if ( lambda1 < 0 && lambda2 < 0)
            return Line(vec3(0.),vec3(0.));
        else if (lambda1 < 0)
            lambda = lambda2;
        else if (lambda2 < 0)
            lambda = lambda1;
        else if (lambda1 < lambda2)
            lambda = lambda1;
        else lambda = lambda2;
    }

    vec3 I = L.origin + lambda * L.v ;
    vec3 N = I - C.origin - dot(I - C.origin,
C.v) * C.v / norm2(C.v) ;

    if (
        dot(I-C.origin, C.v) < 0.0
        || dot(I-C.origin, C.v) > norm2(C.v)
        || dot(I-L.origin, L.v) < 0.0
    ) return Line(vec3(0.),vec3(0.));

    return Line(I,N);
}
```

Intersection d’un Plan

```
Line get_intersection(Line L, Plane T, int
type){

    vec3 n = cross(T.u1,T.u2);

    if (dot(n,L.v) == 0.)
        return Line(vec3(0.),vec3(0.));

    // The line intersect the plane of the
plane
    float lambda = dot(n, T.origin - L.origin)
/ dot(n,L.v);
    if (lambda <= 0.0)
        return Line(vec3(0.),vec3(0.));
    // H is the intersection
    vec3 H = L.origin + lambda * L.v ;
    vec3 v = H - T.origin ;

    vec3 normal = dot(L.v,n) > 0.0 ? -n : n ;

    if (type == 2) return Line(H, normal);

    // a and b are coordinate relative to the
local base of the plane (T.u1, T.u2)
    vec3 coordinate =
locals_cord(v,T.u1,T.u2);
    if (coordinate.z == 0.0)
        return Line(vec3(0.),vec3(0.));;
    float a = coordinate.x / coordinate.z;
    float b = coordinate.y / coordinate.z;

    // The intersection is in the rectangle
    if (type == 3){
        if (0. <= a && a <= 1. && 0. <= b && b
<= 1.){
            return Line(H, normal);
        } else {
            return Line(vec3(0.),vec3(0.));
        }
    }

    // The intersection is in the circle
    if (type == 4){
        if ( a*a + b*b < 1){
            return Line(H, normal);
        } else {
            return Line(vec3(0.),vec3(0.));
        }
    }

    // The intersection is in the triangle
    if (0. <= a+b && a+b <= 1. && 0. <= a &&
0. <= b){
        return Line(H, normal);
    } else { return Line(vec3(0.),vec3(0.));}

}
```

Coordonnées locales

```
vec3 locals_cord(vec3 v, vec3 u1, vec3 u2){
    // return cords of v in local base (u1,u2)
    float a = dot(u2,u2) * dot(v,u1) - dot(u1,u2) * dot(v,u2);
    float b = dot(u1,u1) * dot(v,u2) - dot(u1,u2) * dot(v,u1);
    float k = dot(u1,u1) * dot(u2,u2) - dot(u1,u2) * dot(u2,u1);
    return vec3(a,b,k) ;
}

vec2 spherical_cord(vec3 u){
    vec3 v = normalize(u) ;
    float angle = atan(v.z,v.x) / TWO_PI ;
    if (angle < 0) angle += 1 ;

    return vec2(angle, (1-v.y)/2 ) ;
}
```

Matrice de Rotation

```
mat3 rot(float t12, float t13, float t23){
    mat3 r1 = mat3(
        cos(t12),sin(t12),0,
        -sin(t12),cos(t12), 0.,
        0., 0., 1.
    );

    mat3 r2 = mat3(
        cos(t13), 0., sin(t13),
        0., 1., 0.,
        -sin(t13), 0., cos(t13)
    );

    mat3 r3 = mat3(
        1., 0., 0.,
        0., cos(t23), sin(t23),
        0., -sin(t23), cos(t23)
    );


    return r1 * r2 * r3;
}
```

Calcul d'une Couleur

```
vec4 calc_color(Object Obj, Line Normal){

    vec3 col = vec3(0.);
    vec3 obj_col = Obj.color ;
    float n_col = 0.0;

    if (Obj.is_light){
        return vec4(obj_col,-1.0);

    } else if (light_type != 0 && Obj.mirror != 1.0) {
        // Add light
        for (int i=0 ; i < nb_object ; i++) {

            Object Light = objects[i];
            if ( ! Light.is_light ){ continue; }

            vec3 center ;
            if (Light.type == TYPE_SPHERE)
               center = Light.sphere.center ;
            else if (Light.type == TYPE_CYLINDER)
               center = Light.cylinder.origin ;
            else if (Light.type == TYPE_CYLINDER_FULL)
               center = Light.cylinder.origin ;
            else center = Light.plane.origin ;

            vec3 light_dir = center - Normal.origin ;
            Line ray = Line(center,-light_dir);
            Object inter = get_intersection(ray);

            if ( inter  == Obj ){
                // angle : 1 if normal    0 if colinear
                float angle =
dot(normalize(Normal.v),normalize(light_dir));
                angle = clamp(angle,0.,1.);
                float dist = sqrt(norm2(light_dir));
                float coef ;
                coef = angle * 5 / (2.0 + dist) ;

                if (angle != 0){
                    vec3 Light_color = vec3(1);
                    if (light_type == 2)
                         col += coef * Light_color *
                         obj_col;
                    else col += coef * Light_color;
                    n_col += coef ;
                        }
                    } else {
                        col += vec3(0);
                        n_col += 1 ;
                    }
                }
            }

            if (light_type != 2) { // Ambient light
                col += obj_col ;
                n_col += 1 ;
            }

            return vec4(col,n_col);
        }
```

Calcul d’un pixel (1)

```
vec3 draw2(Line Ray, const int n_rays_){

    const int n_rays = 4;

    Line Rays[n_rays] ;
    vec3 cols[n_rays] ;
    float coeffs[n_rays] ;
    bool is_init[n_rays] ;

    // Init Array
    for (int i = 0; i < n_rays; i++){
        Rays[i] = Line(vec3(0),vec3(0)) ;
        cols[i] = vec3(0) ;
        coeffs[i] = 1 ;
        is_init[i] = false ;
    }

    // Init first Ray
    int n = 0 ;
    Rays[n] = Ray ;
    is_init[n] = true ;
    n++ ;
    ......
}
```

Calcul d’un pixel (2)

```
vec3 draw2(Line Ray, const int n_rays_){

    ......
    for (int i = 0; i < n_rays; i++){

        if (! is_init[i]) break ;
        Object best_obj = get_intersection(Rays[i]);

        if (best_obj.type == TYPE_ERROR){ // show sky
            if (light_type == 2) cols[i] =  vec3(0.0) ;
            else {
                vec2 v = spherical_cord(Rays[i].v + vec3(0,.1,0)) ;
                if (v.y > .5) cols[i] = vec3(0.0, 0.8, 1.0) ;
                else cols[i] = texture2D(sky,vec2(v.x,2*v.y)).rgb ;
            }
            continue ;
        }

        Line N = get_intersection(Rays[i],best_obj);

        vec4 res = calc_color(best_obj, N );

        if ( res.w == -1.0) { // obj is light
            cols[i] = res.rgb;
            continue ;
        }
        cols[i] = res.rgb ;

        if ( n-2< n_rays && best_obj.refrac != 1){

            bool is_reverse ;
            if (best_obj.type == TYPE_SPHERE)
                is_reverse = dot(N.origin -
best_obj.sphere.center,Ray.v) > 0.0 ;
            else if (best_obj.type == TYPE_CYLINDER || best_obj.type
== TYPE_CYLINDER_FULL)
                is_reverse = dot(N.v,Ray.v) > 0.0 ;
            else is_reverse =
dot(cross(best_obj.plane.u1,best_obj.plane.u2),Ray.v) > 0 ;

            float r = is_reverse ?
                1/best_obj.refrac : best_obj.refrac ;

            // Refraction
            Rays[n].v = refraction(Rays[i].v,N.v,r);
            Rays[n].origin = N.origin ;
            vec2 C = taux_ref(Rays[i].v,Rays[n].v,N.v,r) ;
            coeffs[n] = C.x * coeffs[i]/res.w;
            is_init[n] = true ;
            n++ ;

            // Reflexion
            Rays[n].v = reflexion(Rays[i].v,N.v);
            Rays[n].origin = N.origin ;
            coeffs[n] = C.y * coeffs[i]/res.w;
            is_init[n] = true ;
            n++ ;

        } else if (best_obj.mirror != 0.0 && n<n_rays) {
            // Reflexion
            Rays[n].v = reflexion(Rays[i].v,N.v);
            Rays[n].origin = N.origin ;
            coeffs[n] = coeffs[i] * best_obj.mirror;
            is_init[n] = true ;
            n++ ;
        }
        coeffs[i] *= (1-best_obj.mirror) * res.w ;
    }
......
}
```

Calcul d’un pixel (3)

```
vec3 draw2(Line Ray, const int n_rays_){

    ......

    vec3 col = vec3(0) ;
    float sum_coeffs = 0 ;
    int nb_init = 0 ;

    for (int i = 0; i < n_rays; i++){
        if (is_init[i]){
            col += cols[i] + 0* coeffs[i] ;
            // sum_coeffs += coeffs[i] ;
            sum_coeffs += 1 ;
            nb_init += 1 ;
        }
    }

    if (nb_init == 0) return vec3(0.8039, 0.149, 0.7686) ;
    return col / sum_coeffs ;
}
```

Fonction main

```
void main() {
    float size = max(u_resolution.x,u_resolution.y);
    vec2 st =  1.0 * (gl_FragCoord.xy / vec2(size,size) - .5);
    const int n_rays = 4 ;

    init_objects();

    vec2 mouse = 3 * (u_mouse / u_resolution - .5) ;
    mat3 M = rot(0,-2*mouse.x,mouse.y);
    vec3 B = vec3(st.x,st.y,0.66);
    vec3 A = vec3(position.x,2,position.y);

    Line Ray = Line(
        A,
        normalize(M * B)
    );
    vec3 col = draw2(Ray,n_rays) ;

    gl_FragColor = vec4(col, 1.0);

}
```