Rendus 3D

TIPE de Yann RUELLAN

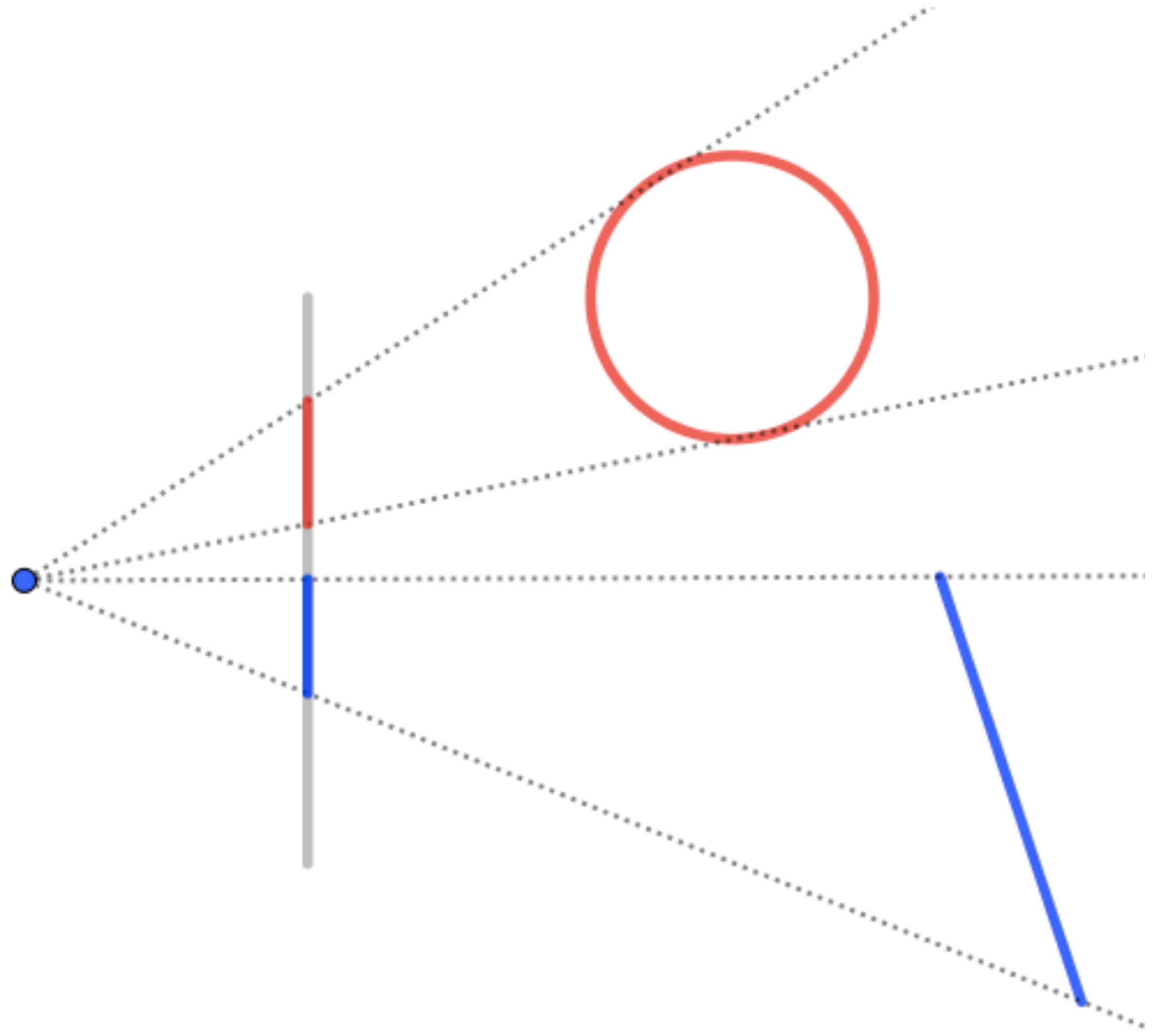
Comment afficher des objects en trois dimensions sur un écran en deux dimensions ?

Plan

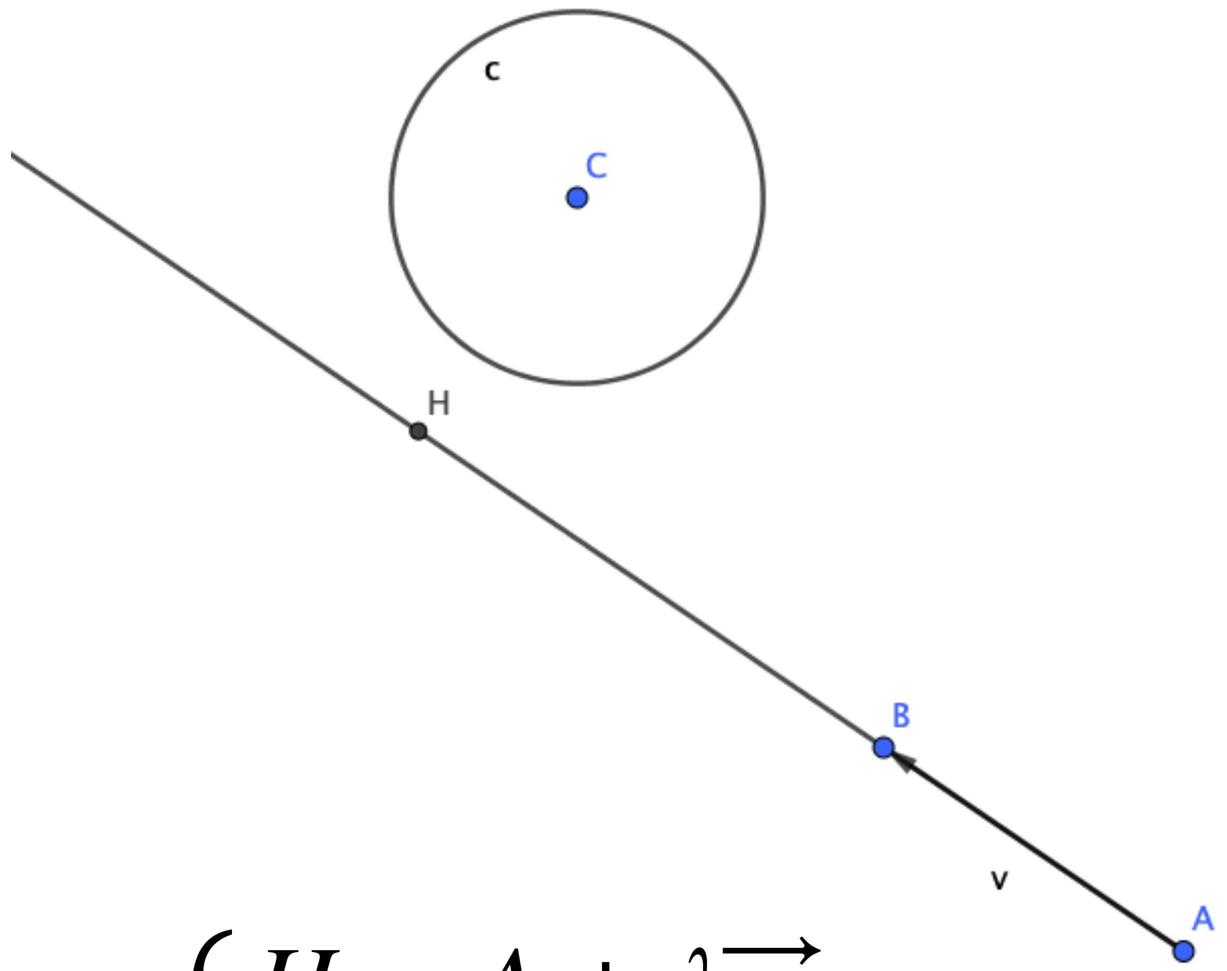
- Perspective
- Physique
- Géométrie
- Programmation

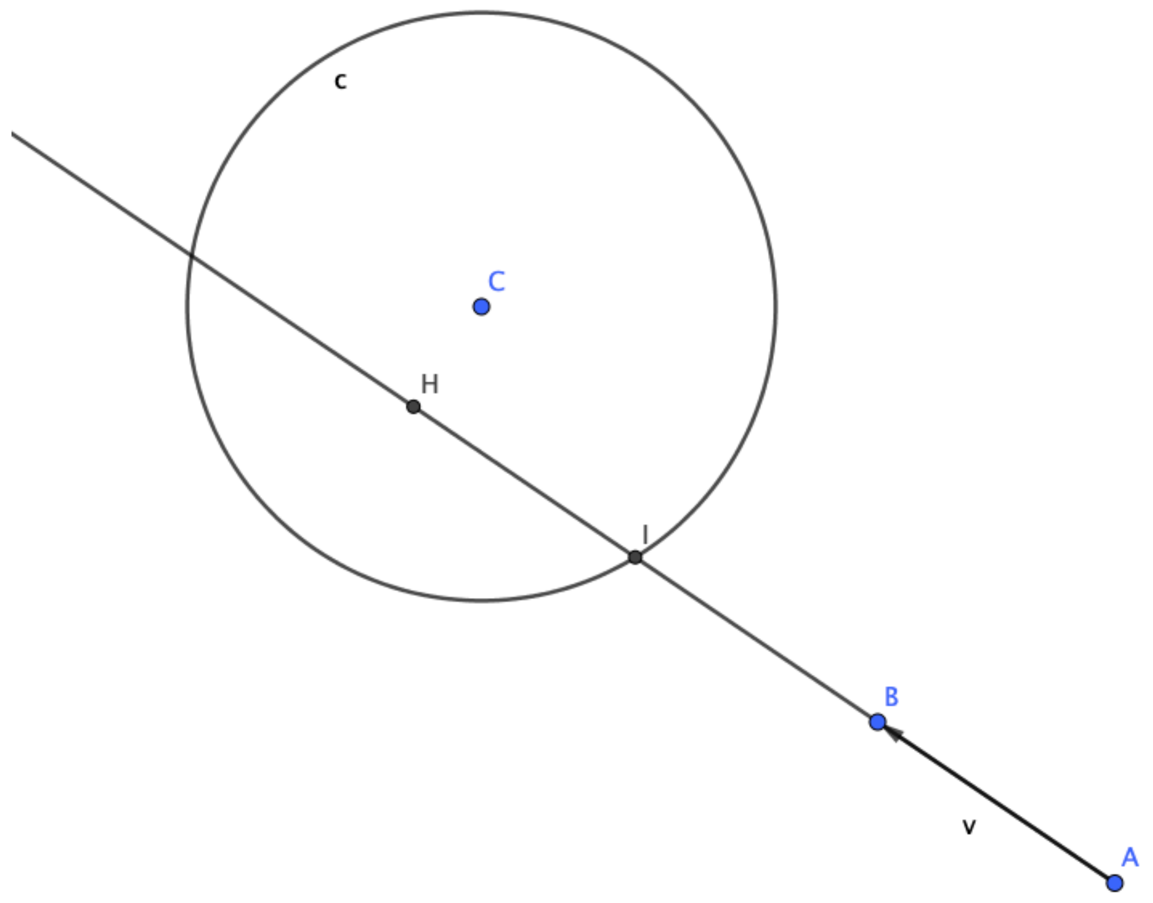
La Perspective

Albrecht Dürer : perspectographe.



Intersection
Sphère



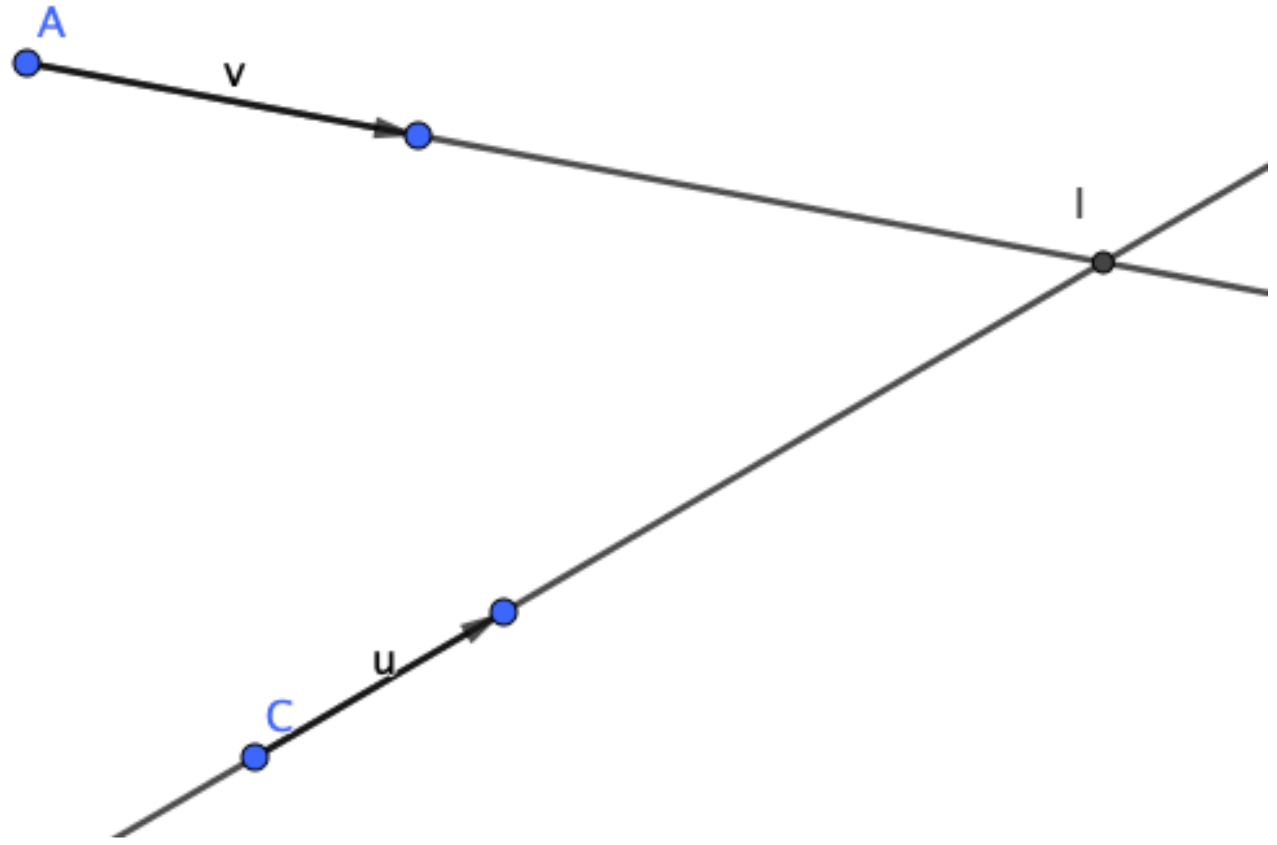


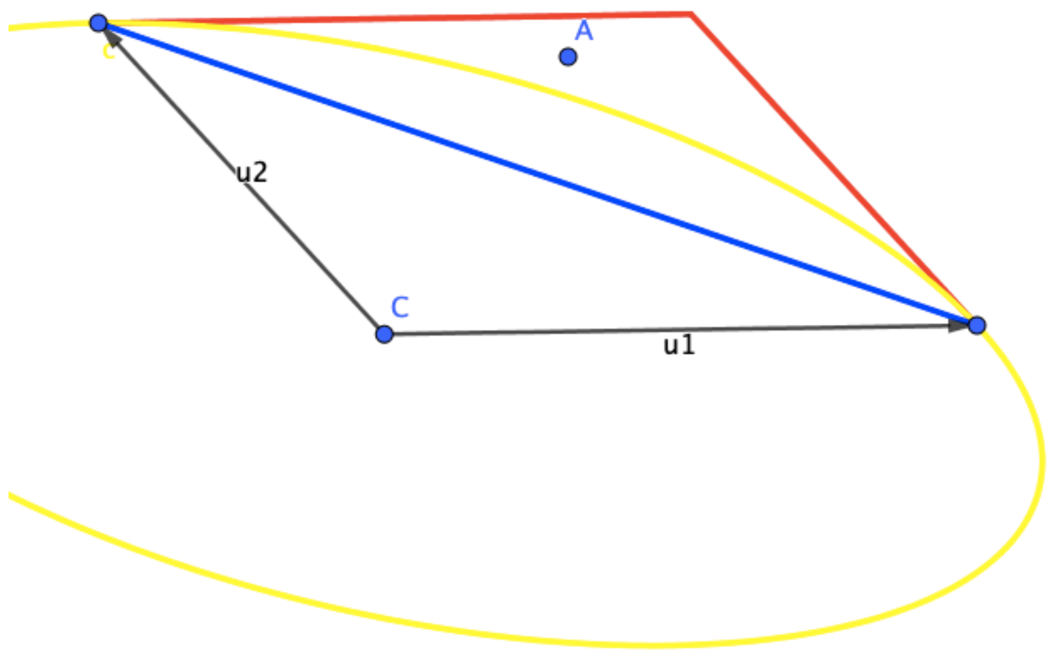
$$\begin{cases} H = A + \lambda \overrightarrow{v} \\ \overrightarrow{AH} \cdot \overrightarrow{v} = \overrightarrow{AC} \cdot \overrightarrow{v} \end{cases}$$

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

Intersection

Rectangle





$$\begin{cases} I = A + \lambda \overrightarrow{v} \\ I = C + \mu \overrightarrow{u} \end{cases}$$

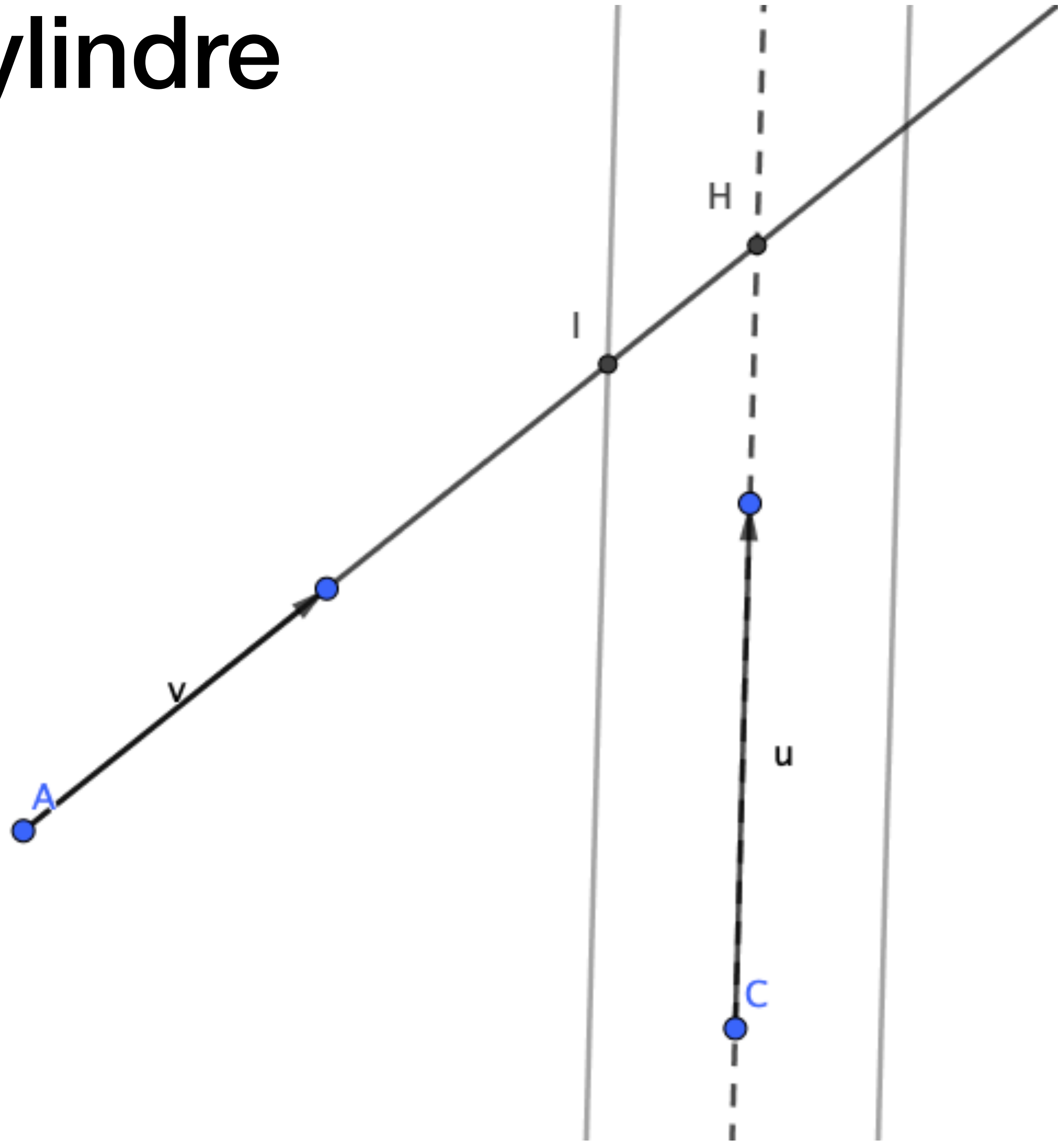
$$x^2 + y^2 = 1$$

$$\begin{cases} 0 \leq x \leq 1 \\ 0 \leq y \leq 1 \end{cases}$$

$$\begin{cases} 0 \leq x \\ 0 \leq y1 \\ x + y \leq 1 \end{cases}$$

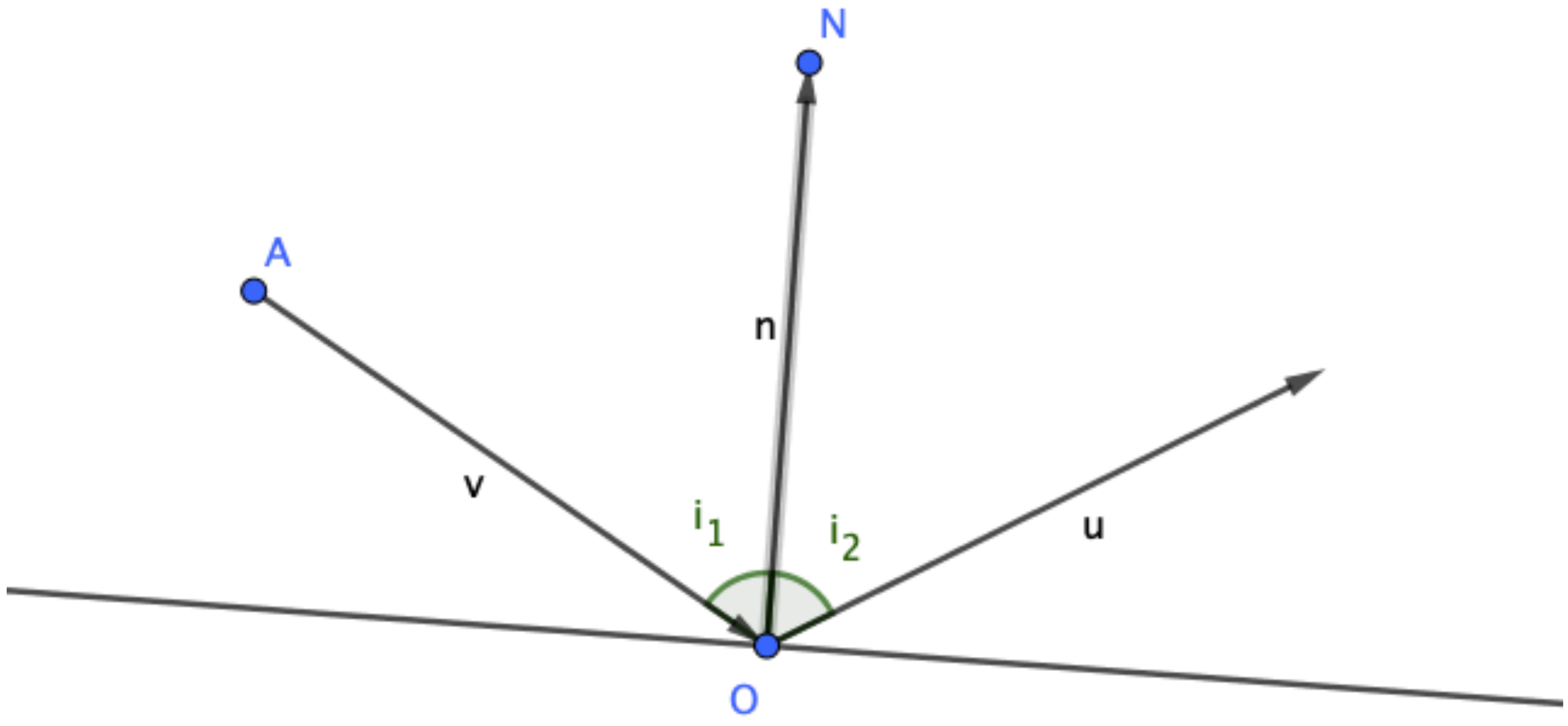
Intersection
Cylindre

$$\begin{cases} I = A + \lambda \overrightarrow{v} \\ I \in \mathscr{C} \end{cases}$$



Lois de Descartes

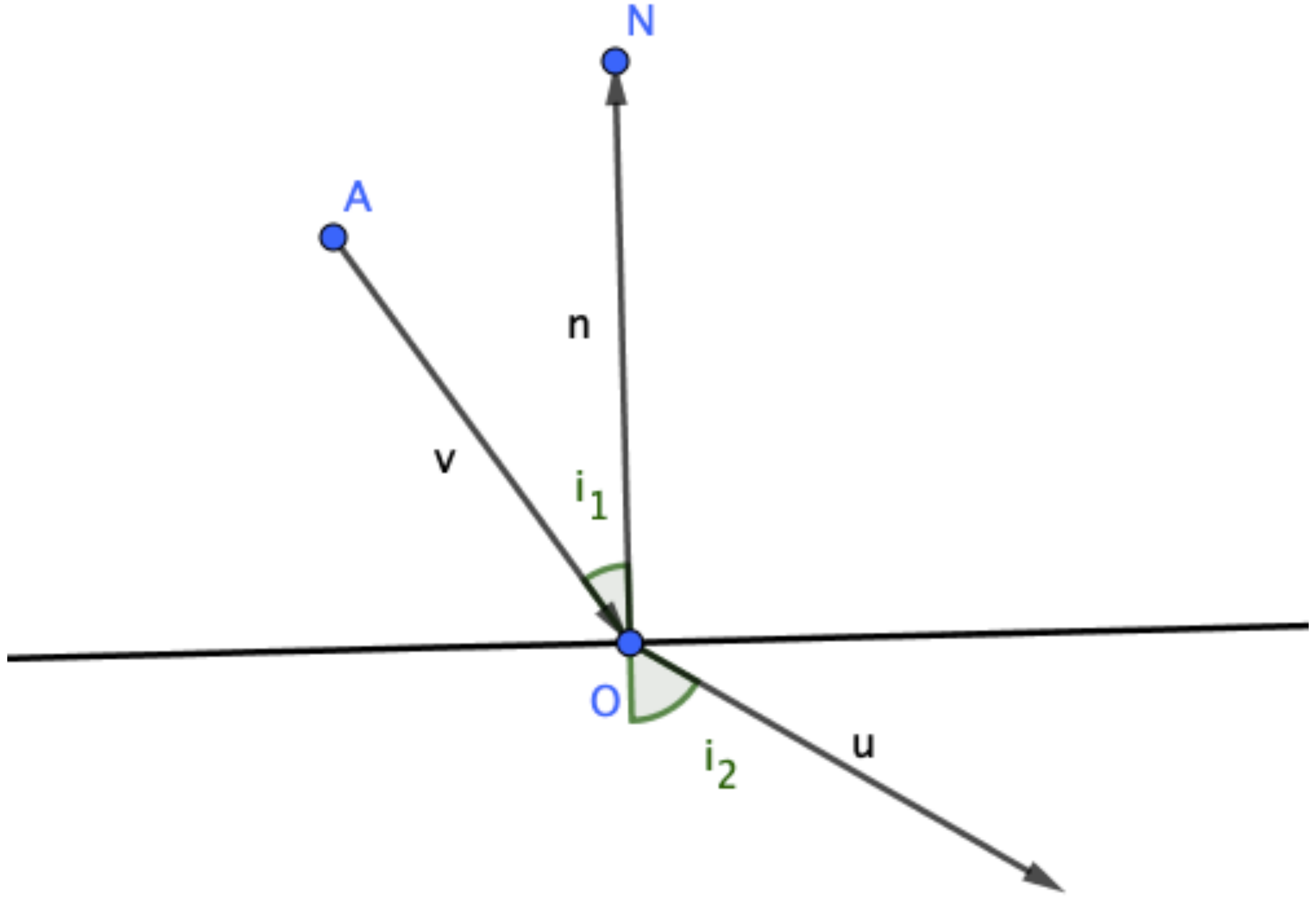
Réflexion



$$i_1 = i_2$$

Lois de Descartes

Réfraction



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

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

-William James Durant

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 ;
}
```

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.));}

}
```

Calcul d’une Couleur

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

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

    if ( Obj.color == vec3(0.0, 0.0, 0.01)){ // Color with
image

        vec3 v = Normal.origin - Obj.plane.origin ;
        vec3 pos =
locals_cord(v,Obj.plane.u1,Obj.plane.u2);
        obj_col = texture2D(u_tex0,vec2(pos.x / pos.z, 1-
pos.y / pos.z) ).rgb;


    } else if ( Obj.color == vec3(.5,.5,.5)) { // Color
the floor
        vec3 H = Normal.origin ;
        bool cond = fract(H.x)-fract(H.z) < 0.0;
        obj_col = cond ? vec3(0.25) : vec3(0.75) ;
    } else { // base color
        obj_col = Obj.color ;
        // obj_col = normalize(Normal.v) ;
    }


    if (Obj.is_light){
        return vec4(Obj.color,-1.0);
    } else if (light_type != 0 && Obj.mirror != 1.0) { //
Add light
        Object Light = objects[0];

        vec3 center = Light.type == 0 ?
Light.sphere.center : Light.plane.origin ;
        vec3 light_dir = center - Normal.origin ;
        Line ray = Line(center,-light_dir);
        Object inter = get_intersection(ray);
        if (
            inter.type == Obj.type &&
            inter.color == Obj.color &&
            inter.mirror == Obj.mirror &&
            inter.is_light == Obj.is_light &&
            inter.sphere == Obj.sphere &&
            inter.plane == Obj.plane
        ){
            // angle : 1 if normal    0 if colinear
            float angle =
dot(normalize(Normal.v),normalize(light_dir));
            angle = clamp(angle,0.,1.);
            float dist = (sqrt(dot(light_dir,light_dir)));
            float coef ;
            coef = 4 * angle / dist ;
            if (angle != 0){
                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 += ( 1 - Obj.mirror ) * obj_col ;
        n_col += ( 1 - Obj.mirror ) ;

    }

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

Calcul d’un pixel

```
vec3 draw(Line Ray){
    vec3 col = vec3(0.);
    float n_col = 0;
    float col_weight = 1;
    float dist_from_origin = 0;


    const int n_reflection = 4;

    for (int i = 0; i < n_reflection; i++){

        Object best_obj = get_intersection(Ray);

        if (best_obj.type == -1){
            if (light_type == 2) col += col_weight
* vec3(0.0) ;
            else col += col_weight * vec3(0.0, 0.0,
0.3) ;
            n_col += col_weight ;
            break ;
        }

        Line N = get_intersection(Ray,best_obj);
        dist_from_origin += distance(N.origin,
Ray.origin) ;

        vec4 res = calc_color(best_obj, N,
dist_from_origin );

        if ( res.w == -1.0) {
            // if best_obj is a light
            col += col_weight * res.xyz;
            n_col += col_weight;
            break ;
        }
        col += col_weight * res.w * res.xyz ;
        n_col += col_weight * res.w ;
        col_weight *= best_obj.mirror ;

        if (col_weight == 0) break ;

        if (best_obj.refrac != 1.0 ){
            bool is_reverse ;
            if (best_obj.type == 0) is_reverse =
dot(N.origin - best_obj.sphere.center,Ray.v) >
0.0 ;
            else if (best_obj.type == 1)
                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
;
            Ray.v = refraction(Ray.v,N.v,r) ;
            if (Ray.v == vec3(0.0)) break ;

        } else {
            Ray.v = reflection(Ray.v,N.v);
        }
        Ray.origin = N.origin ;


    }

    return col / n_col ;
}
```

Fonction main

```
void main() {
    float size = max(u_resolution.x,u_resolution.y);
    vec2 st =  1.0 * (gl_FragCoord.xy / vec2(size,size) - .5);
    vec3 col = vec3(0);

    const int calc_by_pixel = 2 ;

    for (int i = 0; i < calc_by_pixel; i++){
        for (int j = 0; j < calc_by_pixel; j++){
            vec2 mouse = 3 * ((u_mouse + vec2(i,j)/calc_by_pixel) /
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,0,position.y);

            Line Ray = Line(
                A,
                normalize(M * B)
            );
            col += draw(Ray);
        }
    }

    gl_FragColor = vec4(col / (calc_by_pixel * calc_by_pixel), 1.0);

}
```