

# Mini OpenGL en C++

# Objectifs

- ◆ Ecrire un Renderer en quelques centaines de lignes de C++
- ◆ Purement software, mais en n'oubliant pas qu'en pratique il tournerait sur la carte graphique
- ◆ Pas à pas ...
- ◆ Idéalement jusqu'à arriver à l'image ci-contre



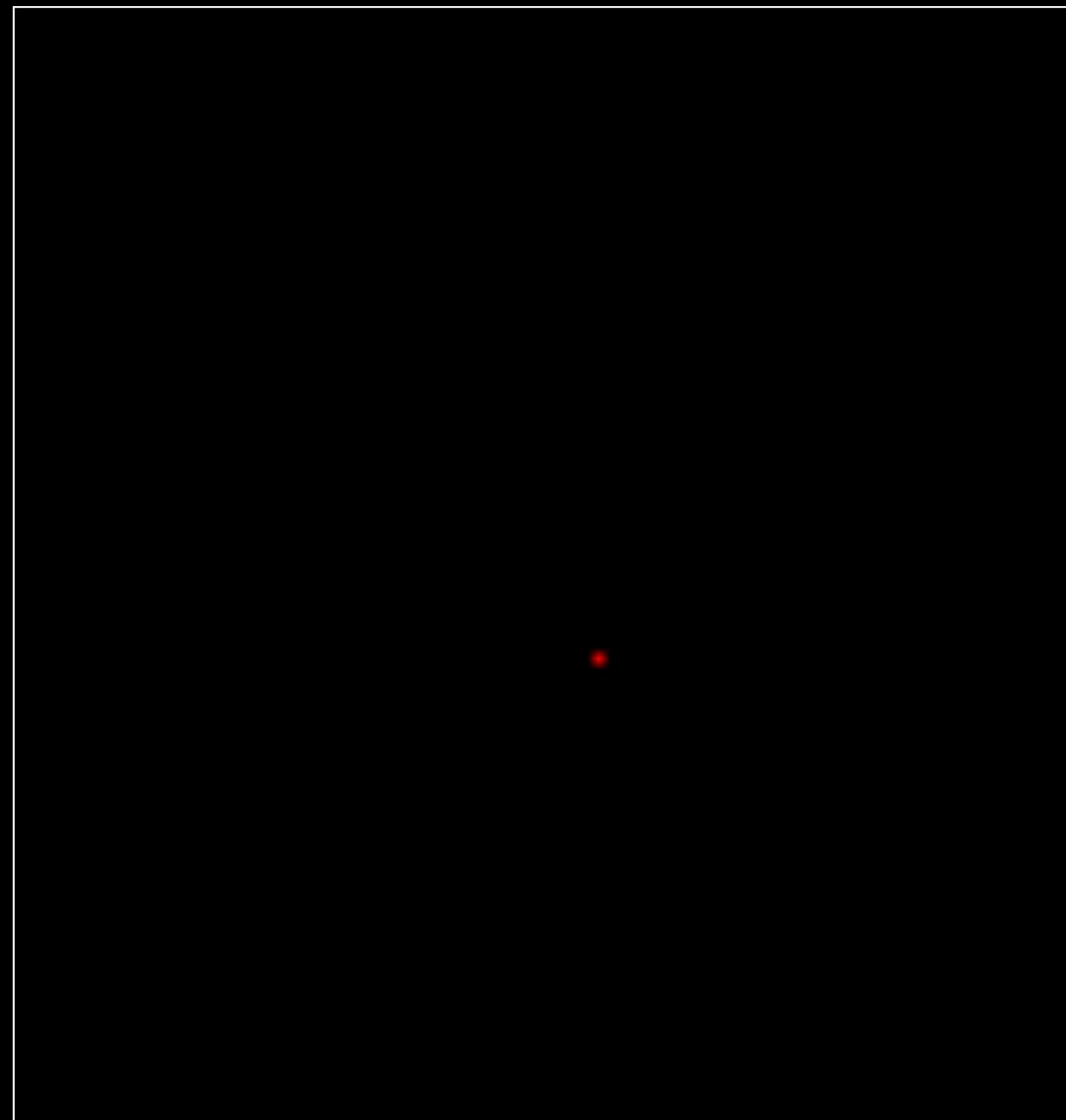
# Format d'affichage

- ◆ Images au format TGA
- ◆ Simple
- ◆ Pixels au formats BW, RGB ou RGBA
- ◆ Je vous fourni une classe TGAIImage minimalist.
- ◆ Constructeur définit la taille
- ◆ La méthode set permet d'écrire dans un pixel

```
class TGAIImage {  
protected:  
    unsigned char* data;  
    int width;  
    int height;  
    int bytespp;  
  
    bool load_rle_data(std::ifstream &in);  
    bool unload_rle_data(std::ofstream &out);  
public:  
    enum Format {  
        GRAYSCALE=1, RGB=3, RGBA=4  
    };  
  
    TGAIImage();  
    TGAIImage(int w, int h, int bpp);  
    TGAIImage(const TGAIImage &img);  
    bool read_tga_file(const char *filename);  
    bool write_tga_file(const char *filename, bool rle=true);  
    bool flip_horizontally();  
    bool flip_vertically();  
    bool scale(int w, int h);  
    TGAColor get(int x, int y);  
    bool set(int x, int y, TGAColor c);  
    ~TGAIImage();  
    TGAIImage & operator =(const TGAIImage &img);  
    int get_width();  
    int get_height();  
    int get_bytespp();  
    unsigned char *buffer();  
    void clear();  
};
```

# Exemple

- ◆ Crée une image 100x100 noire
- ◆ Colorie le pixel 52,41 en rouge
- ◆ Sous le résultat dans output.tga



```
#include "tgaimage.h"

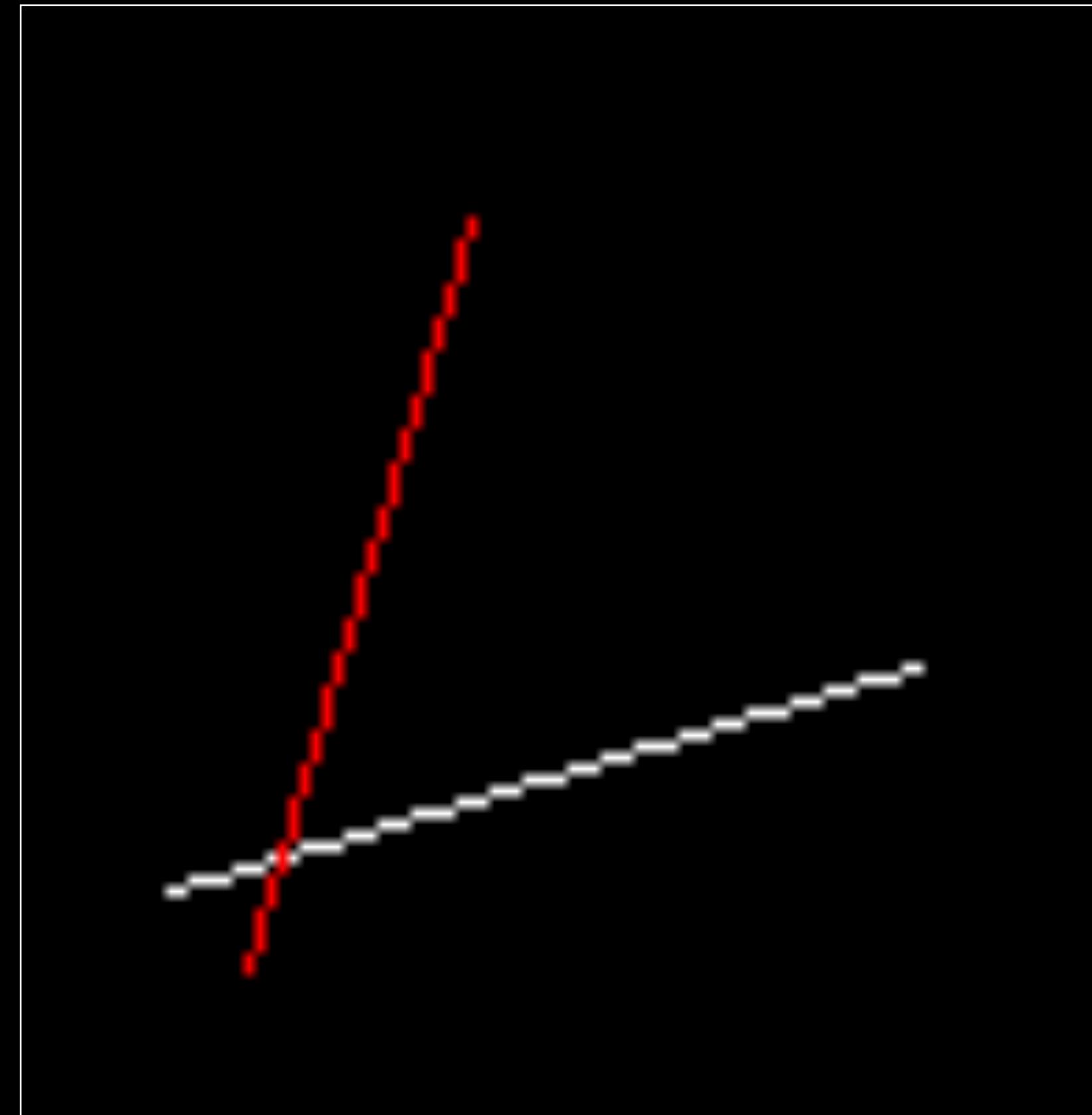
const TGAColor white = TGAColor(255, 255, 255, 255);
const TGAColor red   = TGAColor(255, 0, 0, 255);

int main(int argc, char** argv) {

    TGAImage image(100, 100, TGAImage::RGB);
    image.set(52, 41, red);
    image.flip_vertically();
    image.write_tga_file("output.tga");
    return 0;
}
```

# A vous de jouer...

- ♦ A vous d'écrire la fonction line pour que ce programme produise l'image ci-dessous...



```
#include "tgaimage.h"

const TGAColor white = TGAColor(255, 255, 255, 255);
const TGAColor red   = TGAColor(255, 0, 0, 255);

void line(int x0, int y0, int x1, int y1,
          TGAImage &image, TGAColor color);

int main(int argc, char** argv) {
    TGAImage image(100, 100, TGAImage::RGB);
    line(13, 20, 80, 40, image, white);
    line(20, 13, 40, 80, image, red);
    image.flip_vertically();
    image.write_tga_file("output.tga");
    return 0;
}
```

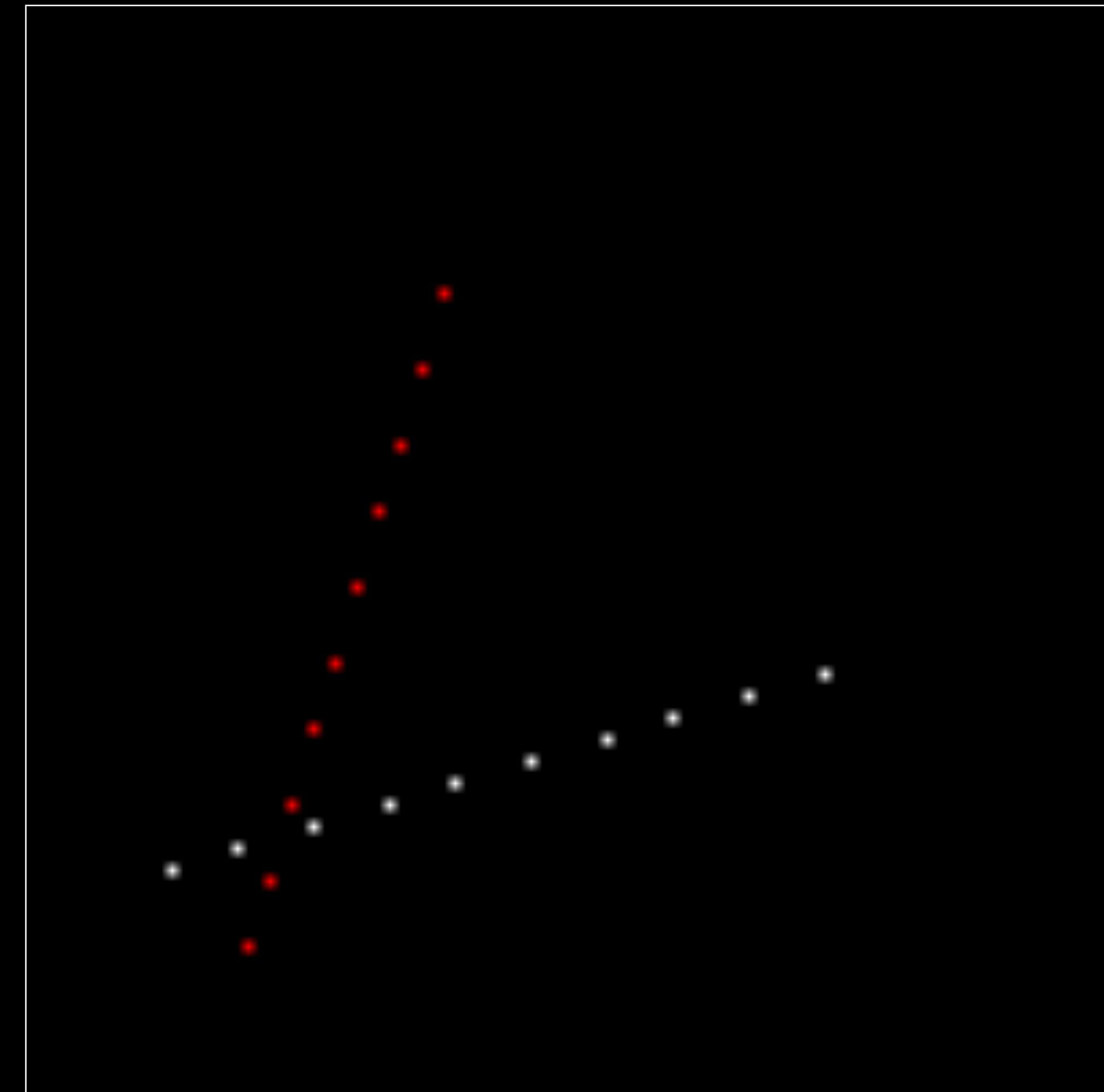
# 1er essai...

```
void line(int x0, int y0, int x1, int y1,  
          TGAIImage &image, TGAColor color)  
{  
    for (float t=0.; t<1.; t+=.01) {  
        int x = x0 + (x1-x0)*t;  
        int y = y0 + (y1-y0)*t;  
        image.set(x, y, color);  
    }  
}
```

- ◆ Fonctionne ... apparemment
- ◆ Méchant nombre magique ...
- ◆ Pas efficace si le pas est trop petit
- ◆ Pas correct si le pas est trop grand

$t += .1$

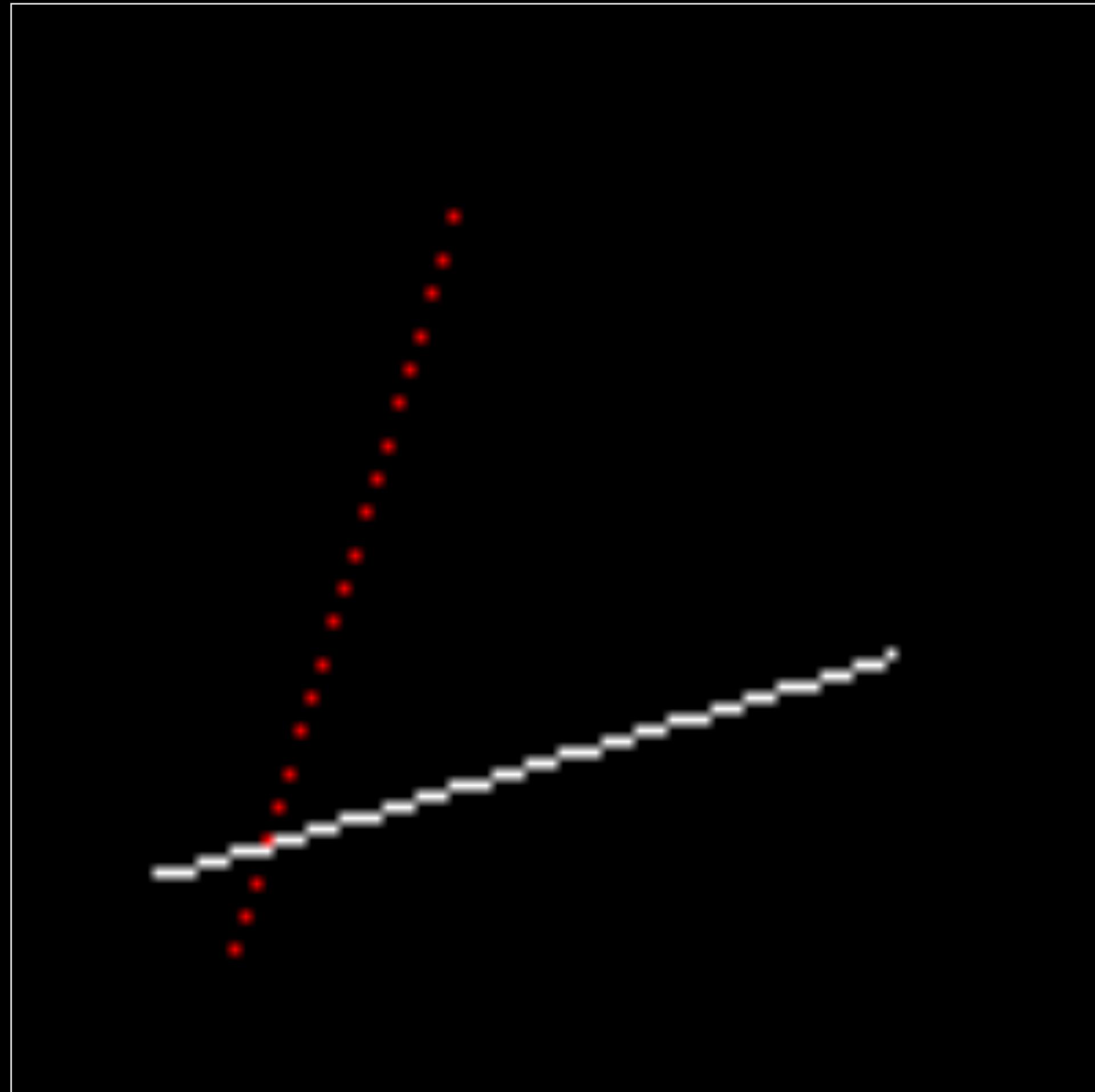
```
void line(int x0, int y0, int x1, int y1,  
TGAImage &image, TGAColor color)  
{  
    for (float t=0.; t<1.; t+=.1) {  
        int x = x0 + (x1-x0)*t;  
        int y = y0 + (y1-y0)*t;  
        image.set(x, y, color);  
    }  
}
```



# Calculons la valeur du pas

```
void line(int x0, int y0, int x1, int y1,  
          TGAImage &image, TGAColor color)  
{  
    for (int x=x0; x<=x1; x++) {  
        float t = (x-x0)/(float)(x1-x0);  
        int y = y0*(1.-t) + y1*t;  
        image.set(x, y, color);  
    }  
}
```

- ◆ Est-ce correct ?



# Corrigeons...

- ◆ Pour les lignes penchées à plus de 45 degrés, transposons x et y
- ◆ Pour les lignes allant de droite à gauche, inversons leur sens
- ◆ C'est mieux ... mais pourquoi faire autant de calculs en virgule flottante ? Et toujours diviser par  $x_1 - x_0$  ?

```
void line(int x0, int y0, int x1, int y1,  
          TGAImage &image, TGAColor color)  
{  
    bool steep = false;  
    if (std::abs(x0-x1)<std::abs(y0-y1)) {  
        std::swap(x0, y0);  
        std::swap(x1, y1);  
        steep = true;  
    }  
    if (x0>x1) {  
        std::swap(x0, x1);  
        std::swap(y0, y1);  
    }  
    for (int x=x0; x<=x1; x++) {  
        float t = (x-x0)/(float)(x1-x0);  
        int y = y0*(1.-t) + y1*t;  
        if (steep) image.set(y, x, color);  
        else       image.set(x, y, color);  
    }  
}
```

- ◆ Sortons la division par  $(x_1 - x_0)$  de la boucle

```
for (int x=x0; x<=x1; x++) {  
    float t = (x-x0)/(float)(x1-x0);  
    int y = y0*(1.-t) + y1*t;  
    if (steep) image.set(y, x, color);  
    else      image.set(x, y, color);  
}  
  
float dt = 1.f/(x1-x0);  
float t = 0;  
for (int x=x0; x<=x1; x++) {  
    t += dt;  
    int y = y0*(1.-t) + y1*t;  
    if (steep) image.set(y, x, color);  
    else      image.set(x, y, color);  
}
```

- ◆ Calculons y directement plutôt que de passer par la variable t

```
float dt = 1.f/(x1-x0);
float t = 0;
for (int x=x0; x<=x1; x++) {
    t += dt;
    int y = y0*(1.-t) + y1*t;
    if (steep) image.set(y, x, color);
    else      image.set(x, y, color);
}
float ystep = (y1-y0)/float(x1-x0);
float y = y0;
for (int x=x0; x<=x1; x++) {
    y += ystep;
    if (steep) image.set(int(y), x, color);
    else      image.set(x, int(y), color);
}
```

- ♦ Séparons les parties entières et décimales de  $y = yi + sign(y1-y0) * yp$

```
float ystep = (y1-y0)/float(x1-x0);           float ystep = std::abs(y1-y0)/float(x1-x0);  
float y = y0;                                 int dysign = (y1 > y0) ? 1 : -1;  
for (int x=x0; x<=x1; x++) {                  int yi = y0;  
    y += ystep;                                float yp = 0; // y = yi + dysign * yp;  
    if (steep) image.set(int(y), x, color);      for (int x=x0; x<=x1; x++) {  
    else      image.set(x, int(y), color);        yp += ystep;  
}                                              if(yp > 0.5) {  
                                                yp -= 1;  
                                                yi += dysign;  
    }  
    if (steep) image.set(yi, x, color);  
    else      image.set(x, yi, color);  
}
```

- ❖ Multiplions tout pas  $2(x_1 - x_0)$  pour effectuer les calculs en nombres entiers

```
float ystep = std::abs(y1-y0)/float(x1-x0);           int dx = (x1-x0);
int dysign = (y1 > y0) ? 1 : -1;
int yi = y0;
float yp = 0; // y = yi + dysign * yp;
for (int x=x0; x<=x1; x++) {
    yp += ystep;
    if(yp > 0.5) {
        yp -= 1;
        yi += dysign;
    }
    if (steep) image.set(yi, x, color);
    else      image.set(x, yi, color);
}
```

```
int ystep = std::abs(2*(y1-y0));
int dysign = (y1>y0) ? +1:-1;
int yi = y0;
int yp = 0; // y = yi + dysign * float(yp)/(2*dx)
for (int x=x0; x<=x1; x++) {
    yp += ystep;
    if(yp > dx) {
        yp -= 2*dx;
        yi += dysign;
    }
    if (steep) image.set(yi, x, color);
    else      image.set(x, yi, color);
}
```

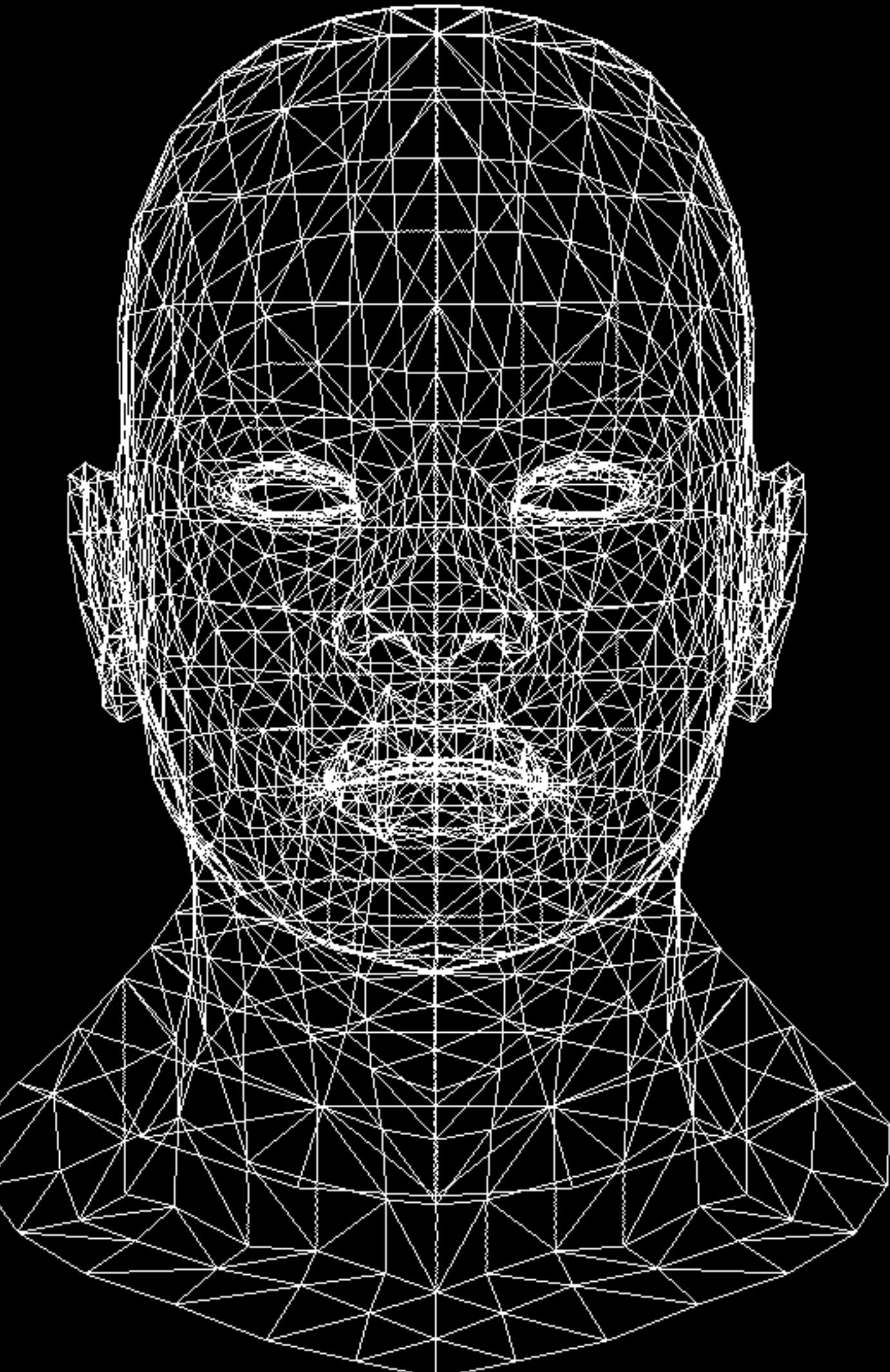
# Algorithme de Bresenham

```
void line(int x0, int y0, int x1, int y1,
          TGAImage &image, TGAColor color)
{
    bool steep = false;
    if (std::abs(x0-x1)<std::abs(y0-y1)) {
        std::swap(x0, y0);
        std::swap(x1, y1);
        steep = true;
    }
    if (x0>x1) {
        std::swap(x0, x1);
        std::swap(y0, y1);
    }
    int dx = (x1-x0);
    int ystep = std::abs(2*(y1-y0));
    int dysign = (y1>y0) ? +1:-1;
    int yi = y0;
    int yp = -dx; // y = yi + dysign * float(yp+dx)/(2*dx)
    for (int x=x0; x<=x1; x++) {
        yp += ystep;
        if(yp > 0) {
            yp -= 2*dx;
            yi += dysign;
        }
        if (steep) image.set(yi, x, color);
        else       image.set(x, yi, color);
    }
}
```

# Wireframe

# Dessiner une ligne ...

- ◆ Le plus simple rendu 3D est le rendu filaire (wireframe rendering)
- ◆ Pour dessiner un triangle, on dessine 3 segments de ligne



# african\_head.obj

```
# List of geometric vertices, with (x, y, z [,w]) coordinates, w is
# optional and defaults to 1.0.
v 0.123 0.234 0.345 1.0
v ...
...
# List of texture coordinates, in (u, [,v ,w]) coordinates, these
# will vary between 0 and 1. v, w are optional and default to 0.
vt 0.500 1 [0]
vt ...
...
# List of vertex normals in (x,y,z) form; normals might not be unit
# vectors.
vn 0.707 0.000 0.707
vn ...
...
# Parameter space vertices in ( u [,v] [,w] ) form; free form
# geometry statement ( see below )
vp 0.310000 3.210000 2.100000
vp ...
...
# Polygonal face element (see below)
f 1 2 3
f 3/1 4/2 5/3
f 6/4/1 3/5/3 7/6/5
f 7//1 8//2 9//3
f ...
...
# Line element (see below)
l 5 8 1 2 4 9
```

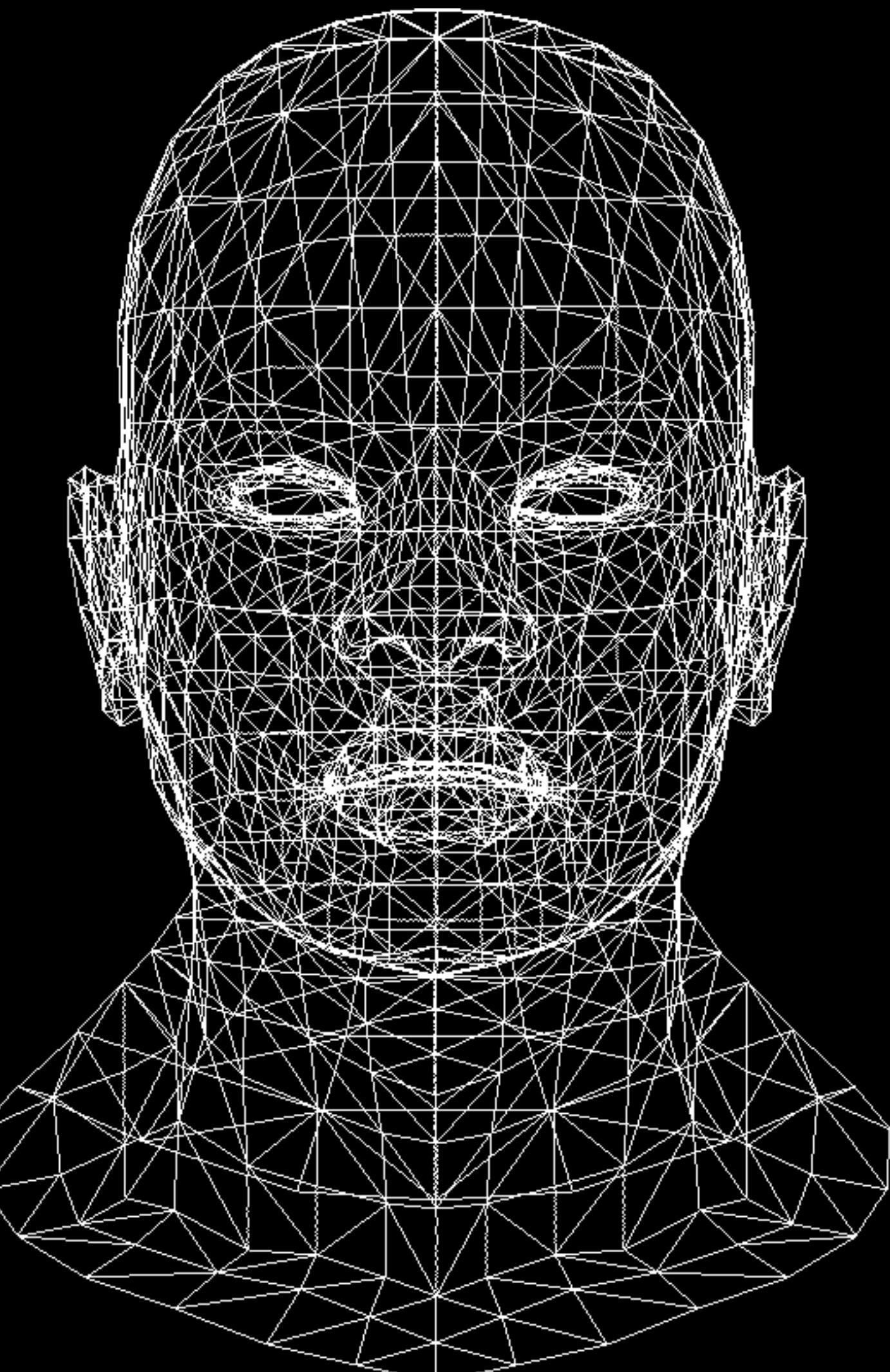
# model.h

```
class Model {
private:
    std::vector<Vec3f> verts_;
    std::vector<std::vector<int>> faces_;
public:
    Model(const char *filename);
    ~Model();
    int nverts();
    int nfaces();
    Vec3f vert(int i);
    std::vector<int> face(int idx);
};
```

```
# List of geometric vertices, with (x, y, z [,w]) coordinates, w is
# optional and defaults to 1.0.
v 0.123 0.234 0.345 1.0
v ...
...
# List of texture coordinates, in (u, [,v ,w]) coordinates, these
# will vary between 0 and 1. v, w are optional and default to 0.
vt 0.500 1 [0]
vt ...
...
# List of vertex normals in (x,y,z) form; normals might not be unit
# vectors.
vn 0.707 0.000 0.707
vn ...
...
# Parameter space vertices in ( u [,v] [,w] ) form; free form
# geometry statement ( see below )
vp 0.310000 3.210000 2.100000
vp ...
...
# Polygonal face element (see below)
f 1 2 3
f 3/1 4/2 5/3
f 6/4/1 3/5/3 7/6/5
f 7//1 8//2 9//3
f ...
...
# Line element (see below)
l 5 8 1 2 4 9
```

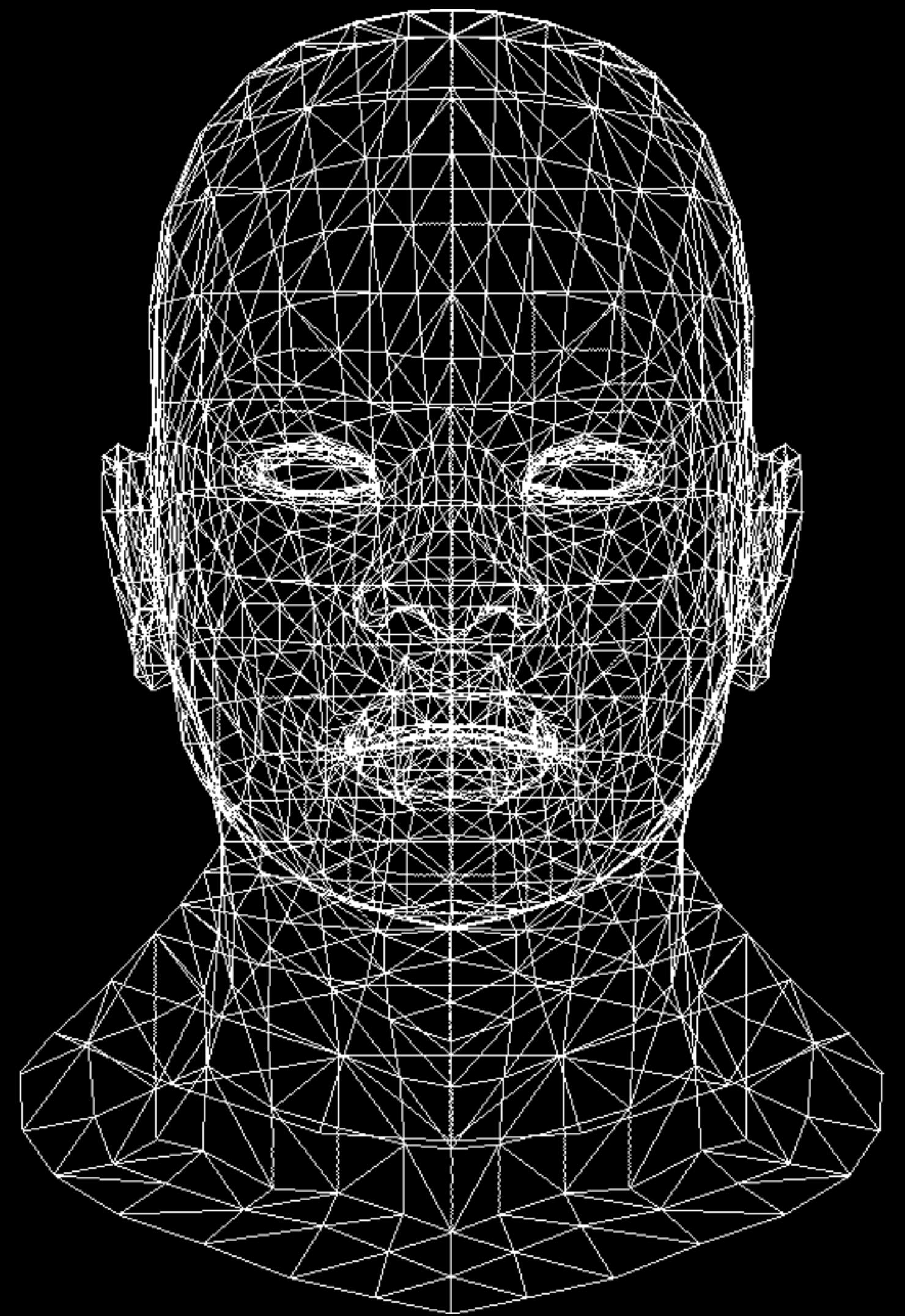
# Votre but

- ◆ Boucler sur tous les triangles
- ◆ Boucler sur les 3 cotés
- ◆ Boucler sur les 2 extrémités du côté
- ◆ Convertir les coordonnées 3d entre -1 et 1 en coordonnées 2d dans l'image
- ◆ Dessiner la ligne

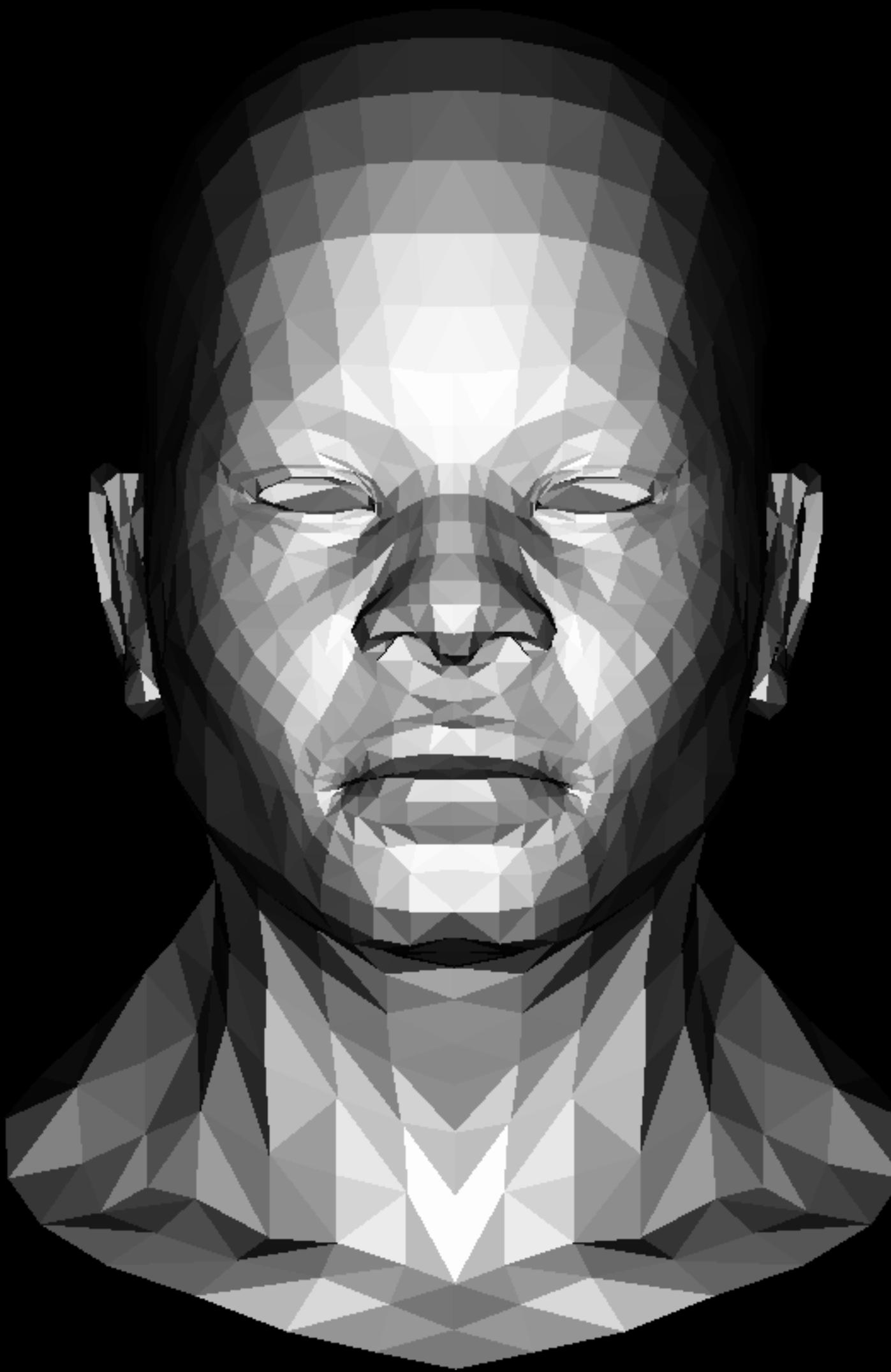


# Dessiner un triangle...

Avant



Après



```
#include "tgaimage.h"
#include "geometry.h"

const TGAColor white = TGAColor(255, 255, 255, 255);
const TGAColor red   = TGAColor(255, 0, 0, 255);
const TGAColor green = TGAColor(0, 255, 0, 255);
const int width   = 200;
const int height  = 200;

void triangle(Vec2i* t,
              TGAIImage &image,
              TGAColor color) {
    line(t[0], t[1], image, color);
    line(t[1], t[2], image, color);
    line(t[2], t[0], image, color);
}
```

# Code fourni

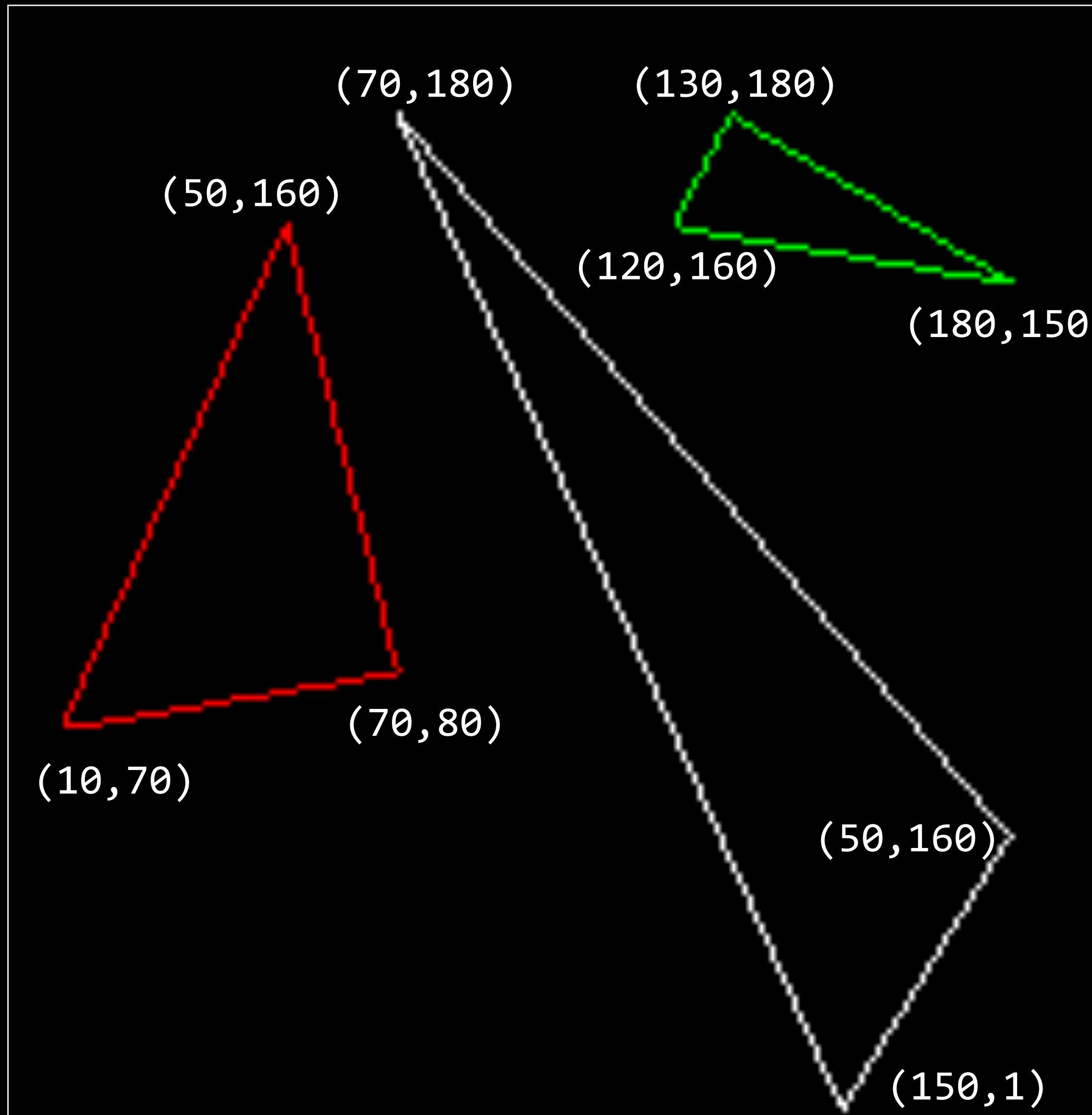
```
int main() {
    TGAIImage image(width, height, TGAIImage::RGB);

    Vec2i t0[3] = {Vec2i(10, 70),
                   Vec2i(50, 160),
                   Vec2i(70, 80)};
    Vec2i t1[3] = {Vec2i(180, 50),
                   Vec2i(150, 1),
                   Vec2i(70, 180)};
    Vec2i t2[3] = {Vec2i(180, 150),
                   Vec2i(120, 160),
                   Vec2i(130, 180)};

    triangle(t0, image, red);
    triangle(t1, image, white);
    triangle(t2, image, green);

    image.flip_vertically();
    image.write_tga_file("triangle.tga");
    return 0;
}
```

# Le résultat fourni



```
int main() {
    TGAImage image(width, height, TGAImage::RGB);

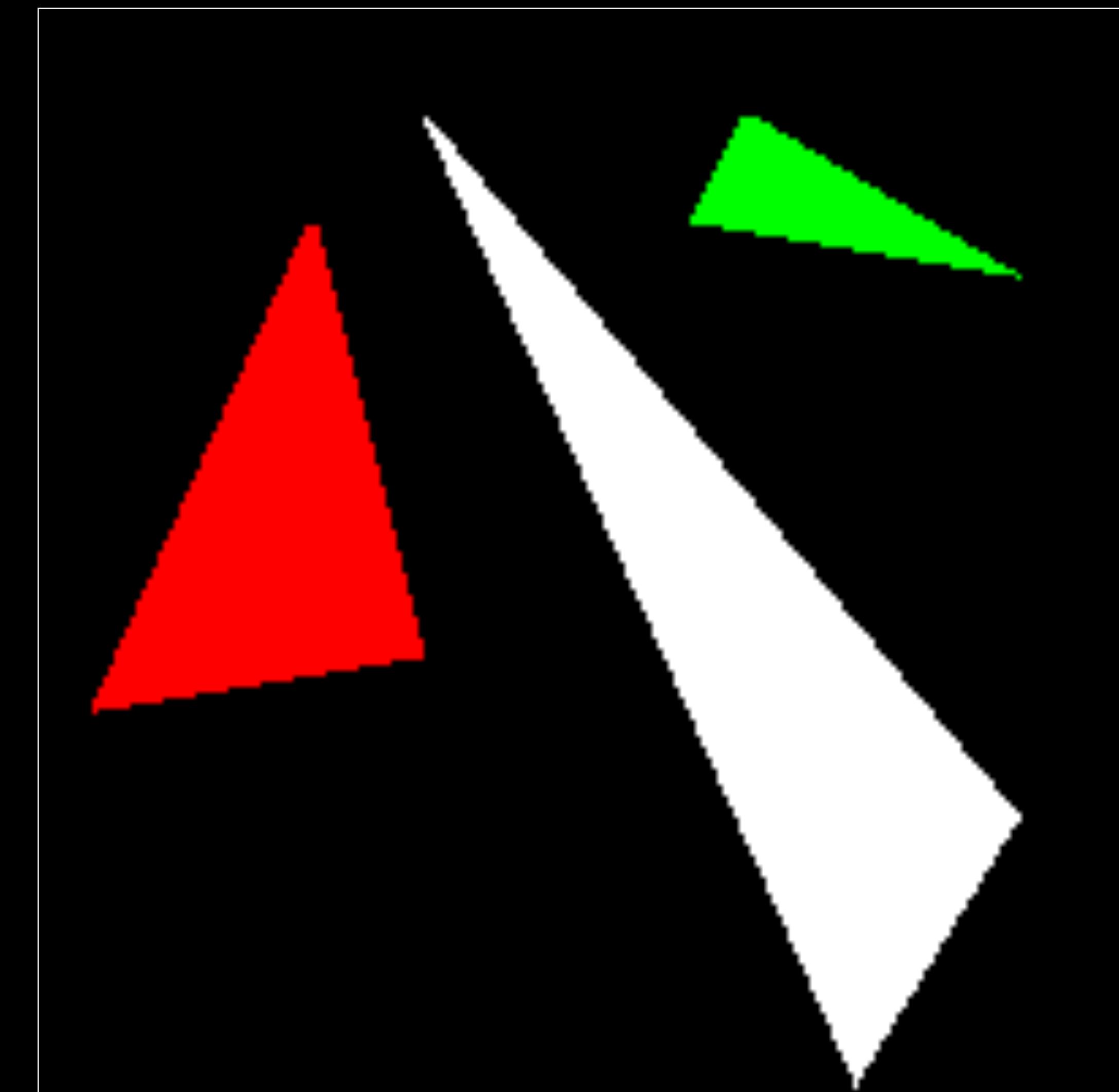
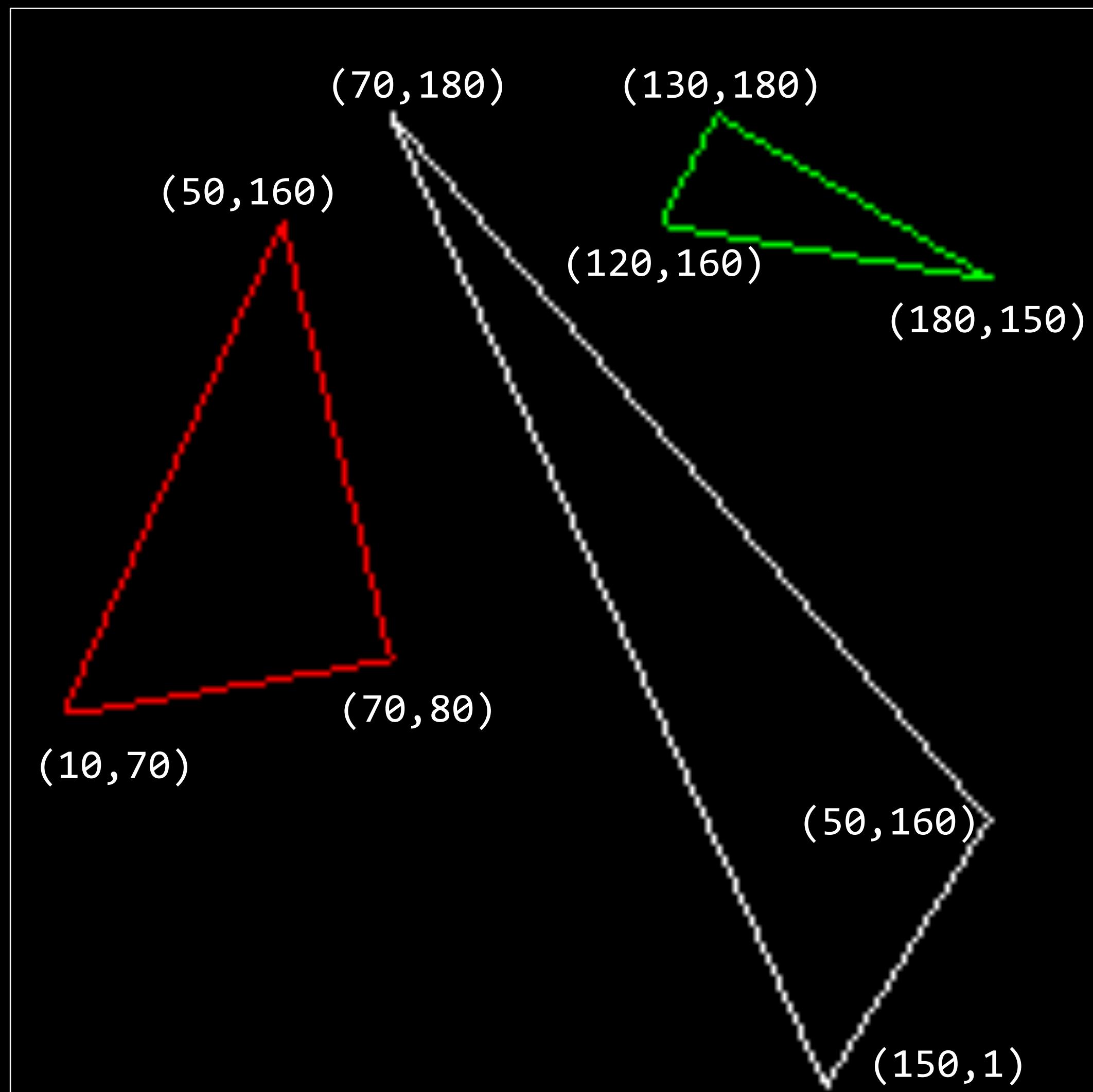
    Vec2i t0[3] = {Vec2i(10, 70),
                   Vec2i(50, 160),
                   Vec2i(70, 80)};
    Vec2i t1[3] = {Vec2i(180, 50),
                   Vec2i(150, 1),
                   Vec2i(70, 180)};
    Vec2i t2[3] = {Vec2i(180, 150),
                   Vec2i(120, 160),
                   Vec2i(130, 180)};

    triangle(t0[0], t0[1], t0[2], image, red);
    triangle(t1[0], t1[1], t1[2], image, white);
    triangle(t2[0], t2[1], t2[2], image, green);

    image.flip_vertically();
    image.write_tga_file("triangle.tga");
    return 0;
}
```

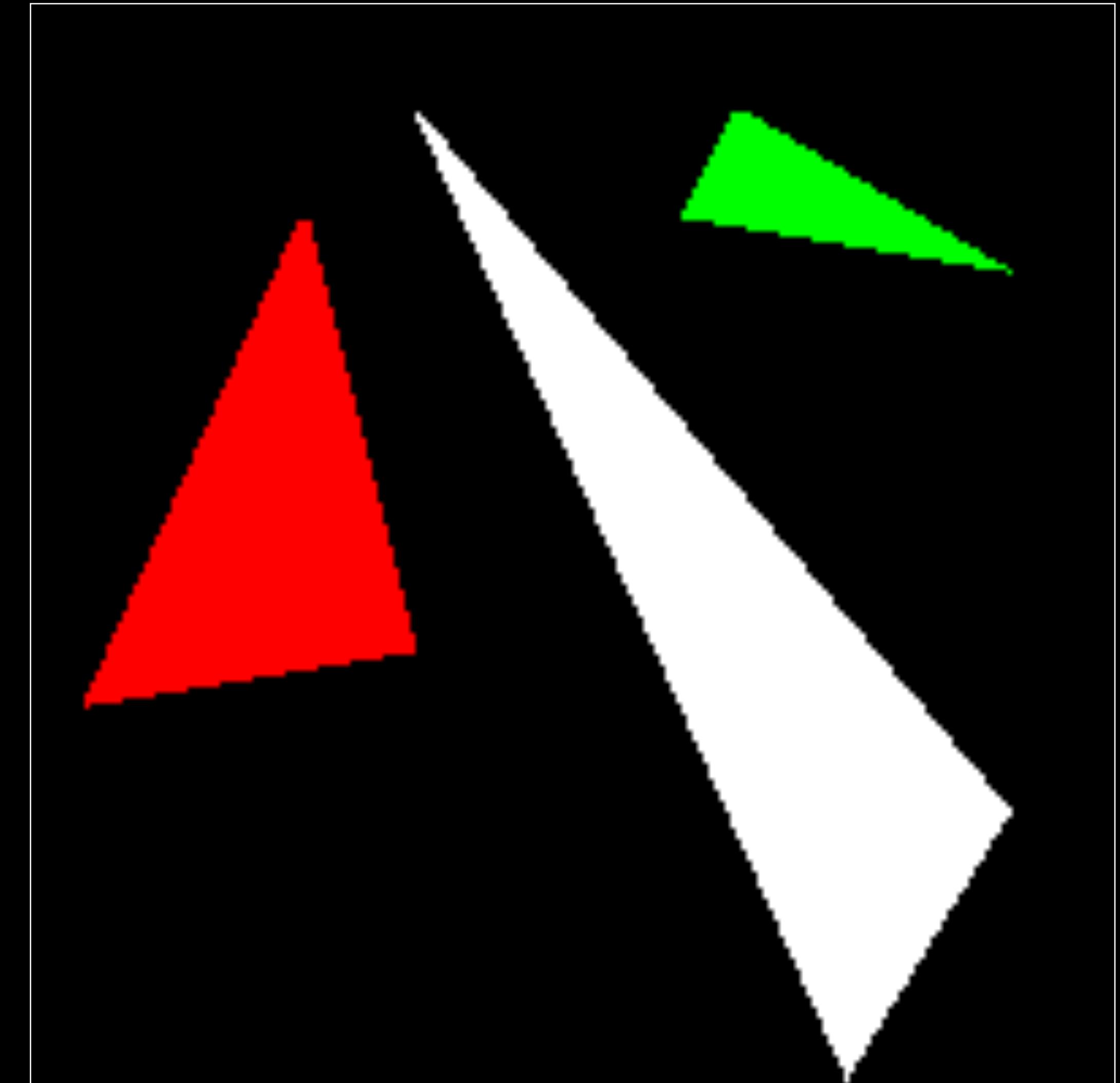
# Le résultat espéré

heig-vd



# La qualité espérée

- ◆ Simple
- ◆ Rapide
- ◆ Symétrique
- ◆ indépendant de l'ordre des sommets
- ◆ Sans trou entre triangles voisins
- ◆ qui partagent un côté, i.e. deux sommets

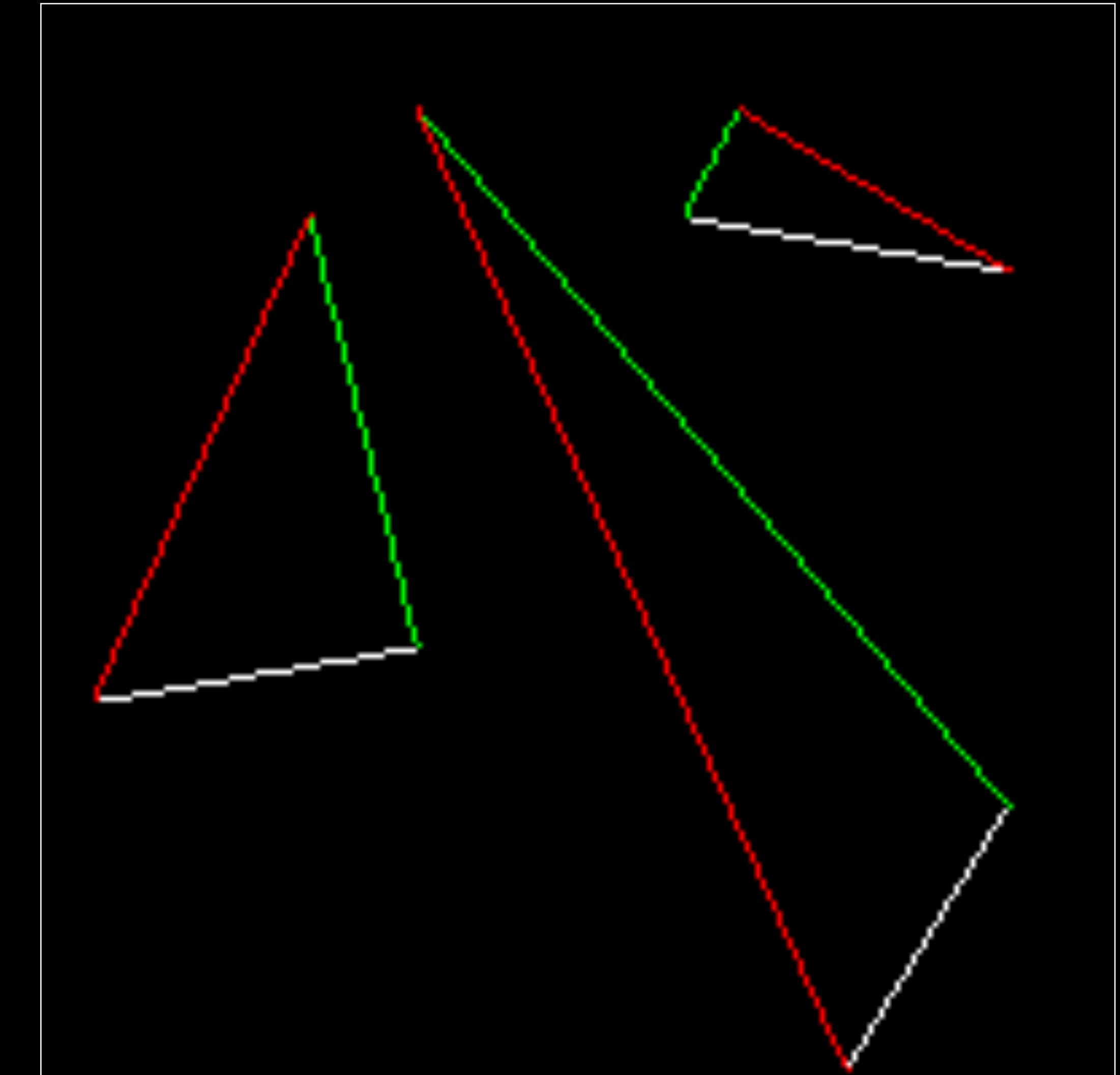


# 1ère approche: line sweeping

- ♦ Remplir le triangle avec des lignes horizontales
- ♦ Boucle sur  $y$  croissant du point le plus bas au point le plus haut
  - ♦ Trier les points dans cet ordre
- ♦ Rasteriser les côtés droits et gauche pour connaître les points de départ et d'arrivée des lignes horizontales
  - ♦ Droite et gauche ?

# Trier les sommets selon l'axe y

```
void triangle(Vec2i*,  
             TGAImage &image,  
             TGAColor color)  
{  
    if(t[1].y < t[0].y) std::swap(t[0],t[1]);  
    if(t[2].y < t[0].y) std::swap(t[0],t[2]);  
    if(t[2].y < t[1].y) std::swap(t[1],t[2]);  
  
    line(t[0], t[1], image, white);  
    line(t[1], t[2], image, green);  
    line(t[2], t[0], image, red);  
}
```



# Remplir l'angle t0-t1 / t0-t2

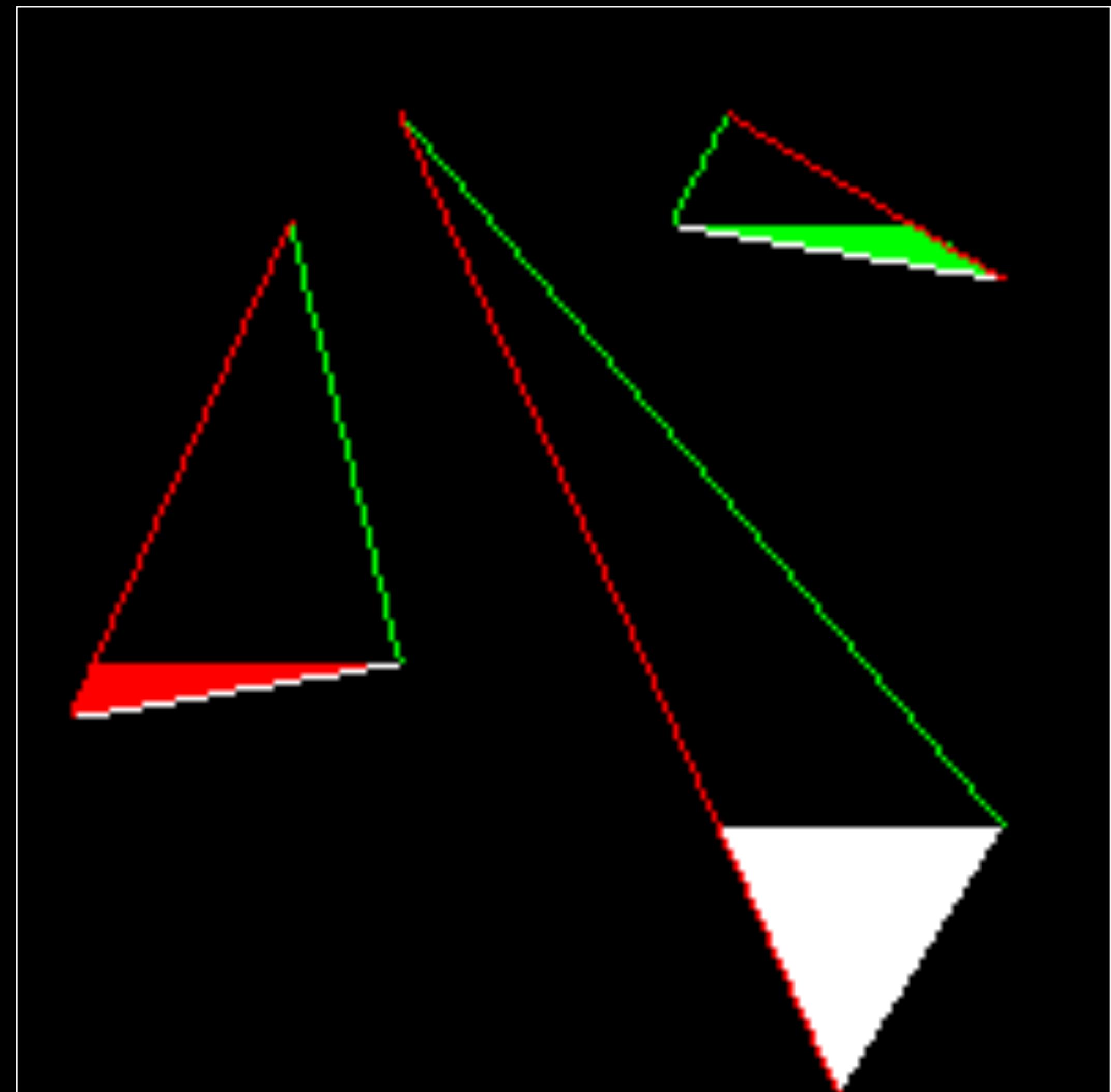
```
void triangle(Vec2i* t,
              TGAIImage &image,
              TGAColor color)
{
    if(t[1].y < t[0].y) std::swap(t[0],t[1]);
    if(t[2].y < t[0].y) std::swap(t[0],t[2]);
    if(t[2].y < t[1].y) std::swap(t[1],t[2]);

    for(int y = t[0].y; y < t1.y; ++y)
    {
        float t01 = ( y - t[0].y ) / float( t[1].y - t[0].y );
        int x01 = t[0].x + t01 * (t[1].x - t[0].x);

        float t02 = ( y - t[0].y ) / float( t[2].y - t[0].y );
        int x02 = t[0].x + t02 * (t[2].x - t[0].x);

        if(x02 < x01) std::swap(x01,x02);
        for(int x = x01; x <= x02; ++x)
            image.set(x,y,color);
    }

    line(t[0], t[1], image, white);
    line(t[1], t[2], image, green);
    line(t[2], t[0], image, red);
}
```



```
void solid_angle(Vec2i t0, Vec2i t1, Vec2i t2, Vec2i t3,
                 TGAImage &image, TGAColor color) {
    if(t0.y == t1.y) ...

    for(int y = t0.y; y <= t1.y; ++y)
    {
        float t01 = ( y - t0.y ) / float( t1.y - t0.y );
        int x01 = t0.x + t01 * (t1.x - t0.x);

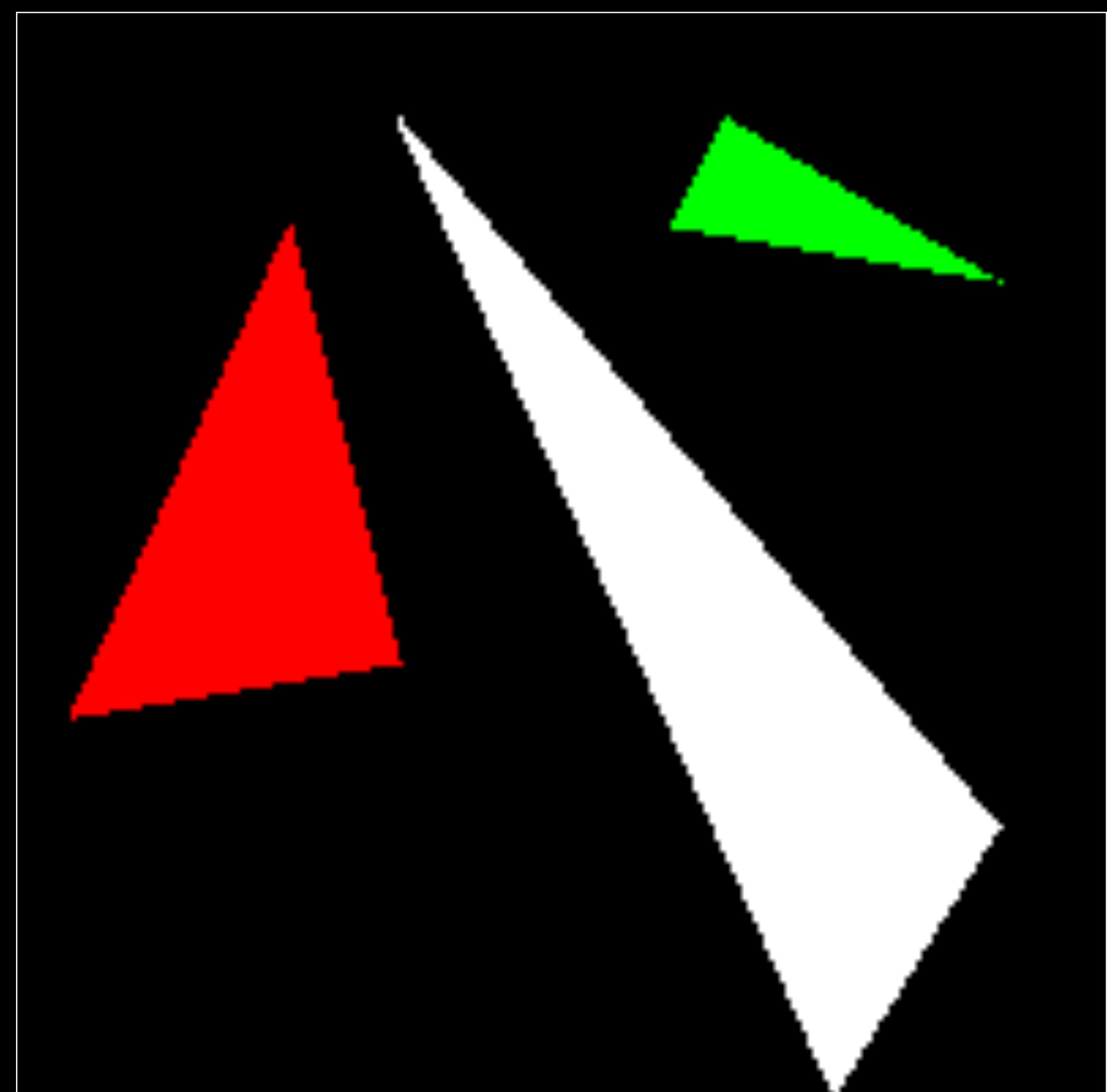
        float t23 = ( y - t2.y ) / float( t3.y - t2.y );
        int x23 = t2.x + t23 * (t3.x - t2.x);

        if(x23 < x01) std::swap(x01,x23);
        for(int x = x01; x <= x23; ++x)
            image.set(x,y,color);
    }
}

void triangle(Vec2i* t,
              TGAImage &image, TGAColor color) {
    if(t[1].y < t[0].y) std::swap(t[0],t[1]);
    if(t[2].y < t[0].y) std::swap(t[0],t[2]);
    if(t[2].y < t[1].y) std::swap(t[1],t[2]);

    solid_angle(t[0], t[1], t[0], t[2], image, color);
    solid_angle(t[1], t[2], t[0], t[2], image, color);
}
```

## Remplir deux demis



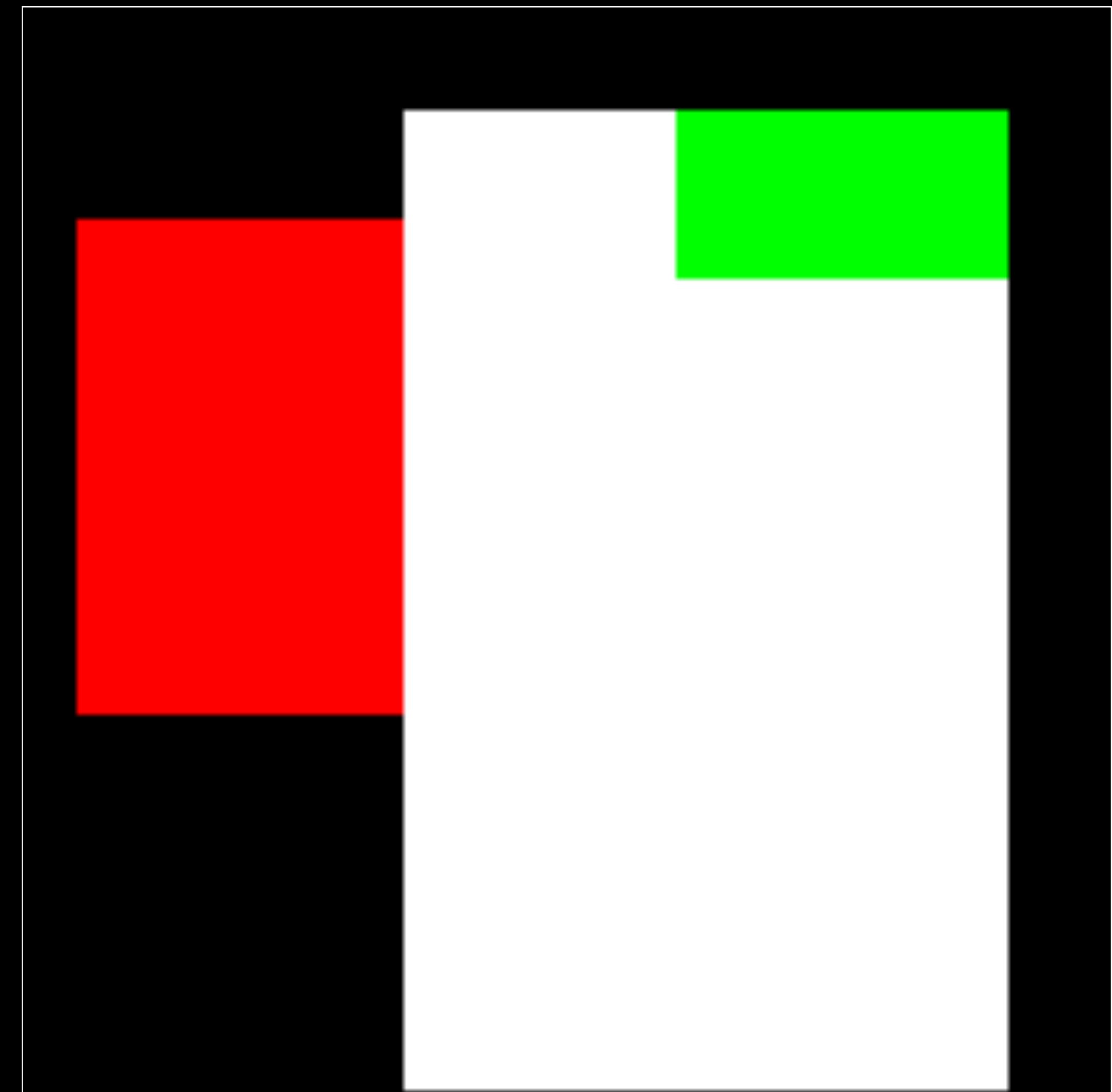
# Et si on exécutait ce code sur GPU ?

```
std::array<Vec2i,2> boite_englante(Vec2i* t, int n = 3);

bool est_dans_le_triangle(Vec2i pt, Vec2i* t);

void triangle(Vec2i *t, TGAImage &image, TGAColor color)
{
    auto bbox = boite_englante(t);
    for(Vec2i p { 0, bbox[0].y }; p.y <= bbox[1].y; ++p.y)
    {
        for(p.x = bbox[0].x; p.x <= bbox[1].x; ++p.x)
        {
            if (est_dans_le_triangle(p, t))
                image.set(p.x,p.y,color);
        }
    }
}
```

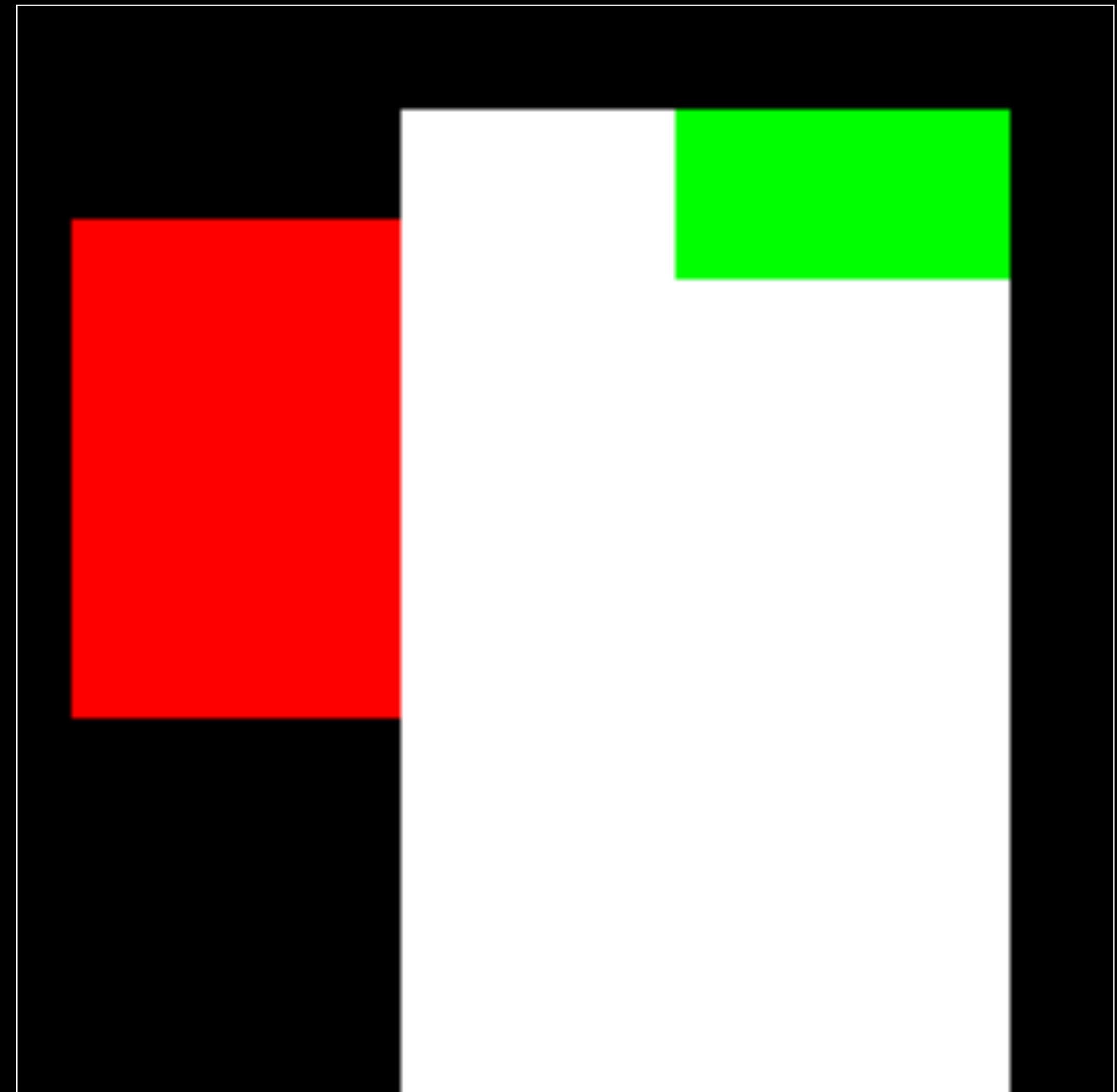
# Bounding Box



# Bounding Box

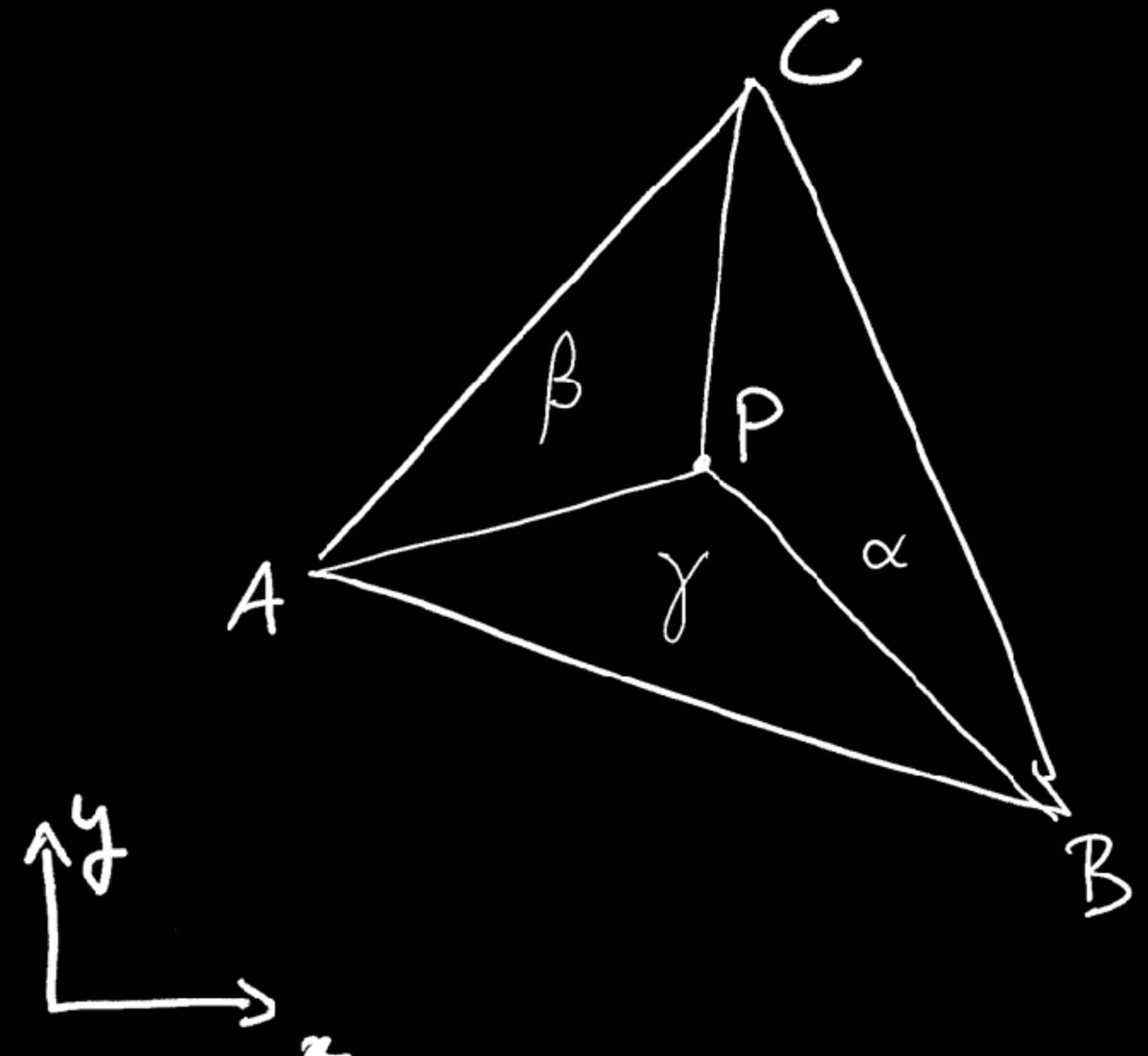
```
std::array<Vec2i,2>
boite_englante(Vec2i* t, int n = 3)
{
    std::array<Vec2i,2> box { t[0],t[0] };
    for(int i = 1; i < n; ++i) {
        if ( t[i].x < box[0].x ) box[0].x = t[i].x;
        if ( t[i].y < box[0].y ) box[0].y = t[i].y;
        if ( t[i].x > box[1].x ) box[1].x = t[i].x;
        if ( t[i].y > box[1].y ) box[1].y = t[i].y;
    }
    return box;
}

bool est_dans_le_triangle(Vec2i pt, Vec2i* t)
{
    return true;
}
```



# Vérifier si p est à l'intérieur d'un triangle ?

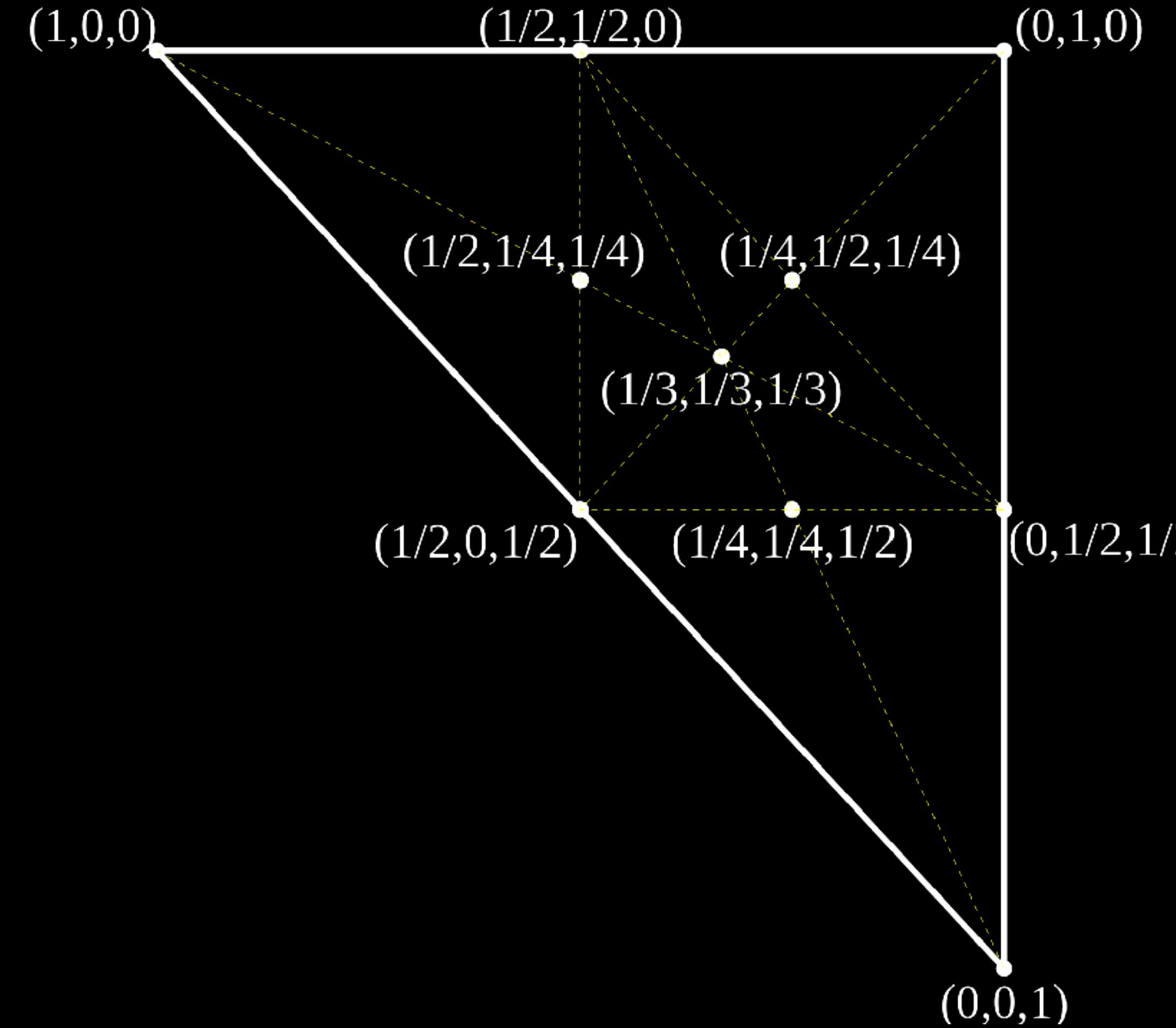
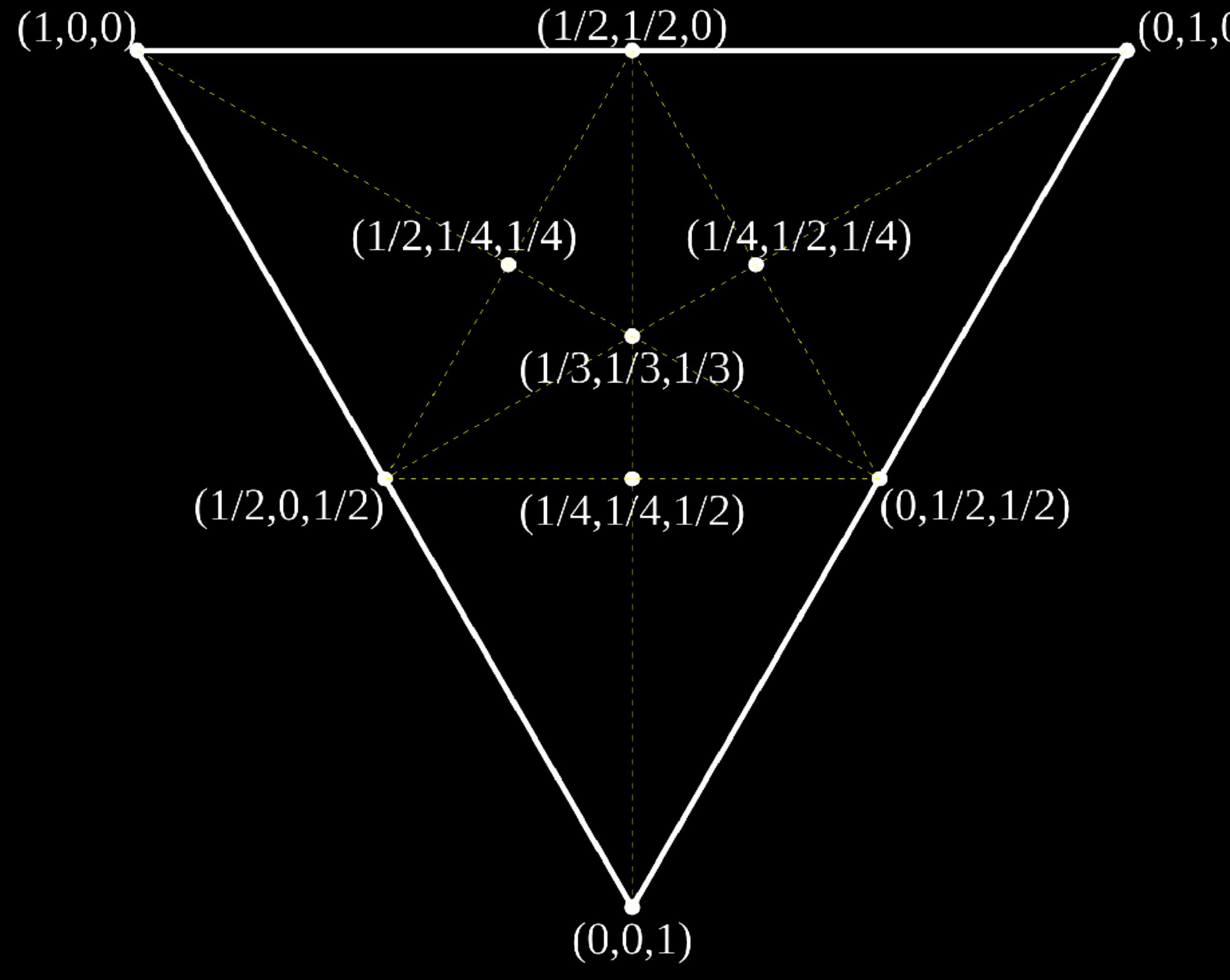
- ♦ Utiliser les coordonnées barycentriques
- ♦ Tout point P est une somme pondérée des sommets A, B et C
- ♦ Sous contrainte de  $\alpha + \beta + \gamma = 1$
- ♦ Le poids d'un sommet est proportionnel à la surface du triangle que P forme avec les 2 autres sommets
- ♦ P est dans le triangle si  $\alpha \geq 0, \beta \geq 0$  et  $\gamma \geq 0$



$$\vec{P} = \alpha \cdot \vec{A} + \beta \cdot \vec{B} + \gamma \cdot \vec{C}$$

# Coordonnées barycentriques

heig-vd

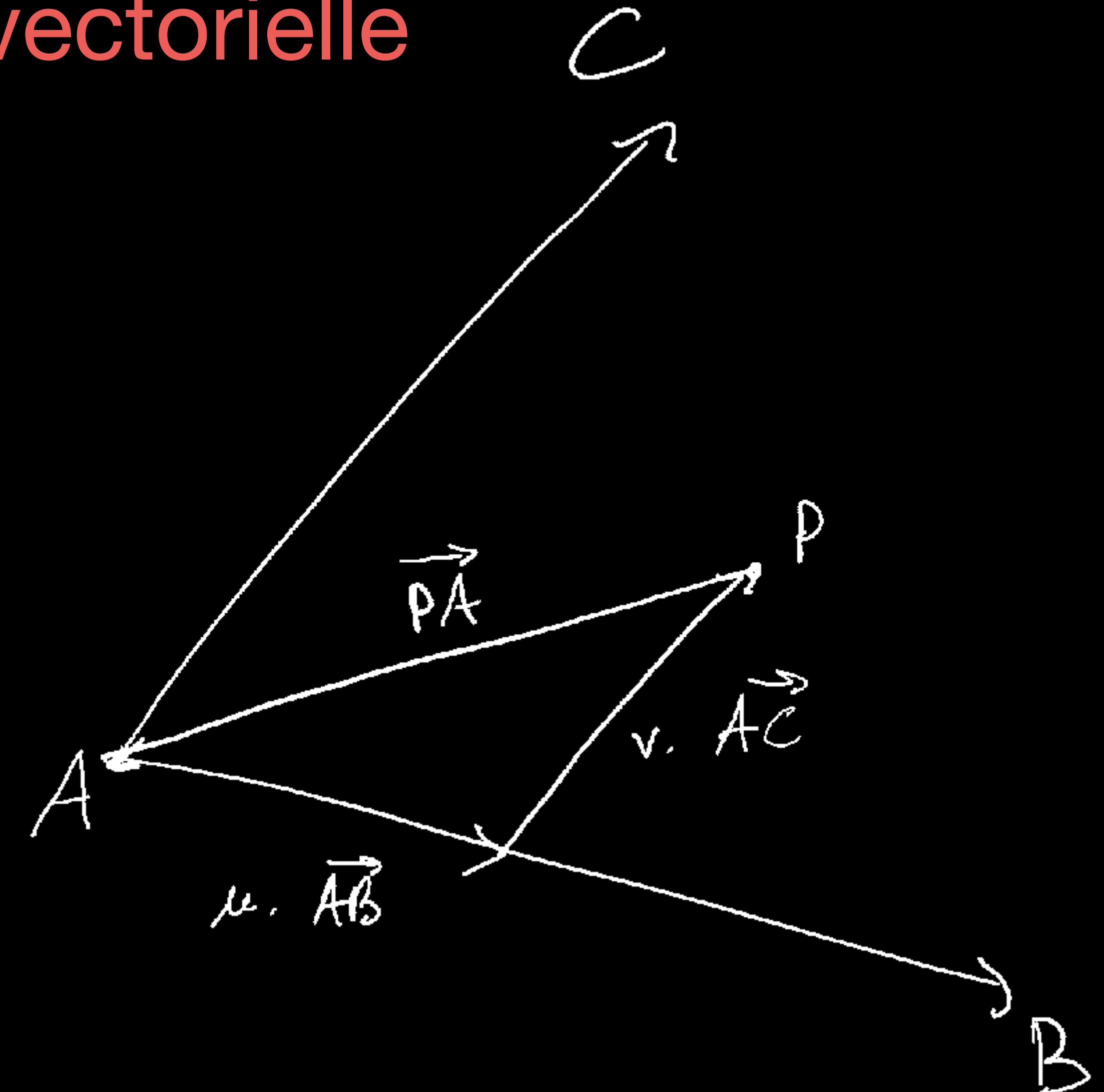


# Sous forme vectorielle

$$\vec{P} = \vec{A} + u \cdot \vec{AB} + v \cdot \vec{AC}$$

$$1 \cdot \vec{A} + \bar{u} \cdot (\vec{B} - \vec{A}) + v \cdot (\vec{C} - \vec{A})$$

$$(1 - u - v) \vec{A} + u \vec{B} + v \vec{C}$$



$$u \cdot \vec{AB} + v \cdot \vec{AC} + \vec{PA} = 0$$

$$u \cdot \overrightarrow{AB} + v \cdot \overrightarrow{AC} + \overrightarrow{PA} = 0$$

$$u \cdot (B_x - A_x) + v \cdot (C_x - A_x) + 1 \cdot (A_x - P_x) = 0$$

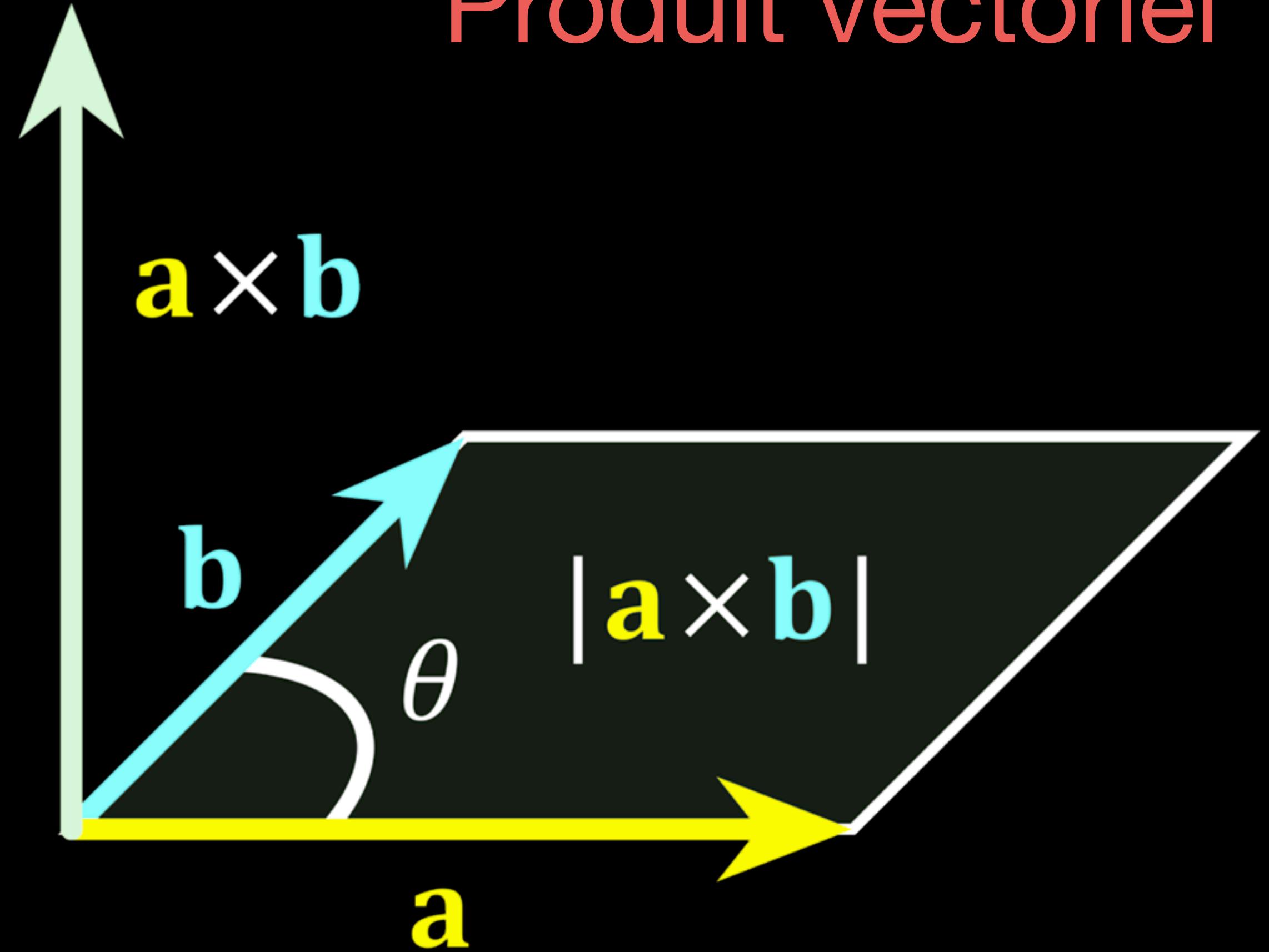
$$u \cdot (B_y - A_y) + v \cdot (C_y - A_y) + 1 \cdot (A_y - P_y) = 0$$

$$\overrightarrow{(u, v, 1)} \perp \overrightarrow{(B_x - A_x, C_x - A_x, A_x - P_x)}$$

$$\overrightarrow{(u, v, 1)} \perp \overrightarrow{(B_y - A_y, C_y - A_y, A_y - P_y)}$$

# Produit vectoriel

heig-vd

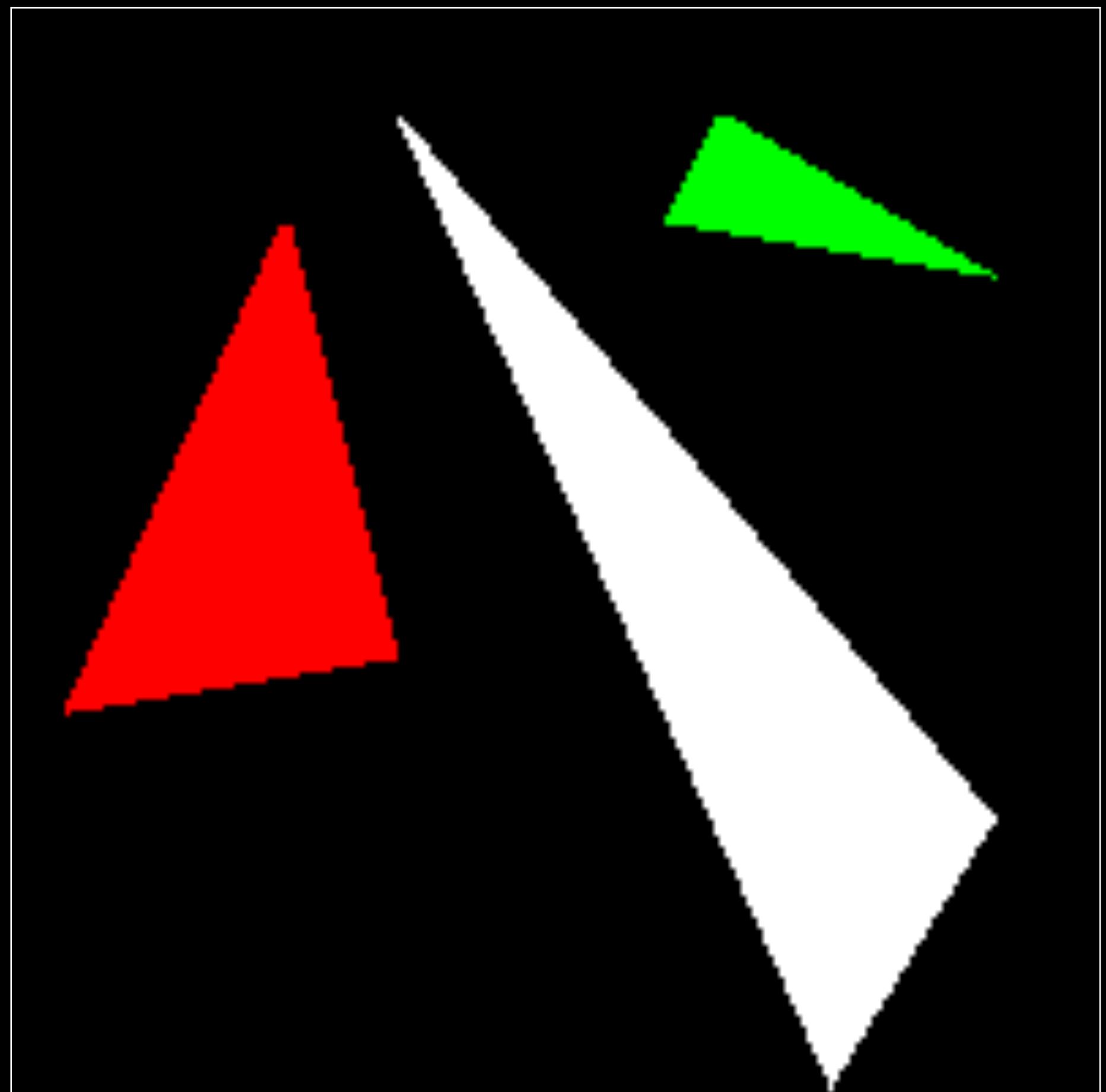


$$\overrightarrow{(u,v,1)} = \overrightarrow{(B_x - A_x, C_x - A_x, A_x - P_x)} \times \overrightarrow{(B_y - A_y, C_y - A_y, A_y - P_y)}$$

# Code C++

```
bool est_dans_le_triangle(Vec2i* t, Vec2i pt)
{
    Vec3f b = barycentriques(t,pt);
    return ( b.x >= 0 and b.y >= 0 and b.z >= 0 );
}

Vec3f barycentriques(Vec2i* t, Vec2i p) {
    Vec3f x { static_cast<float>(t[1].x - t[0].x),
              static_cast<float>(t[2].x - t[0].x),
              static_cast<float>(t[0].x - p.x) };
    Vec3f y { static_cast<float>(t[1].y - t[0].y),
              static_cast<float>(t[2].y - t[0].y),
              static_cast<float>(t[0].y - p.y)};
    Vec3f u = x^y;
    if(abs(u.z) < 1)
        return Vec3f(-1,1,1);
    else
        return Vec3f { 1.f-(u.x+u.y)/u.z, u.x/u.z, u.y/u.z };
}
```



# African Head

```
height  
TGAImage image(width, height, TGAImage::RGB);  
  
for (int i=0; i<model->nfaces(); i++) {  
    std::vector<int> face = model->face(i);  
  
    Vec2i screen_coords[3];  
    Vec3f world_coords[3];  
    for (int j=0; j<3; j++) {  
        world_coords[j] = model->vert(face[j]);  
        screen_coords[j] = Vec2i((world_coords[j].x+1.)*width/2.,  
                               (world_coords[j].y+1.)*height/2.);  
    }  
  
    TGAColor random_color(rand() % 256,  
                          rand() % 256,  
                          rand() % 256,  
                          255);  
    triangle(screen_coords, image, random_color);  
}
```



# Ombrage plat

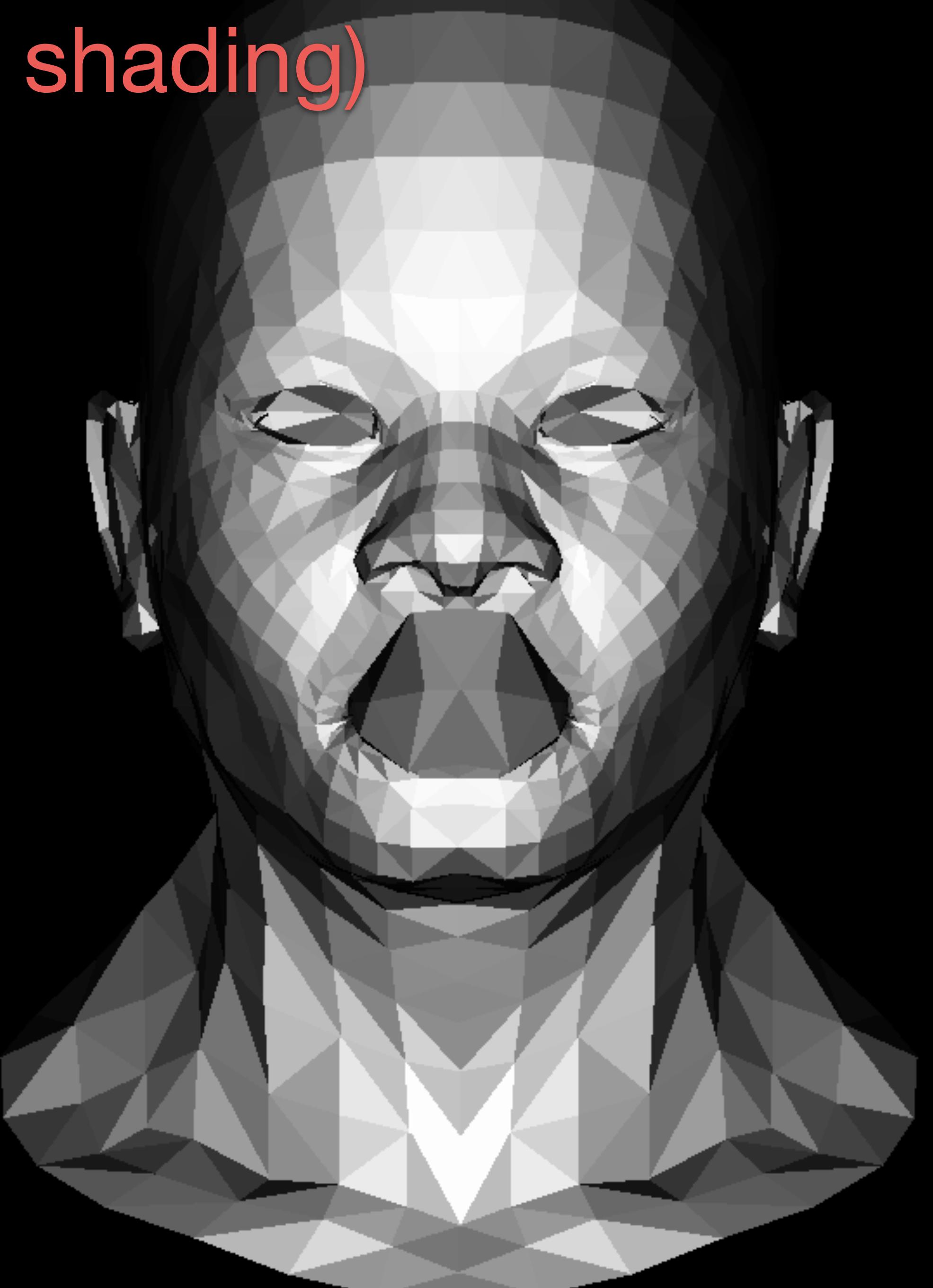
# Principe général de l'ombrage (shading)

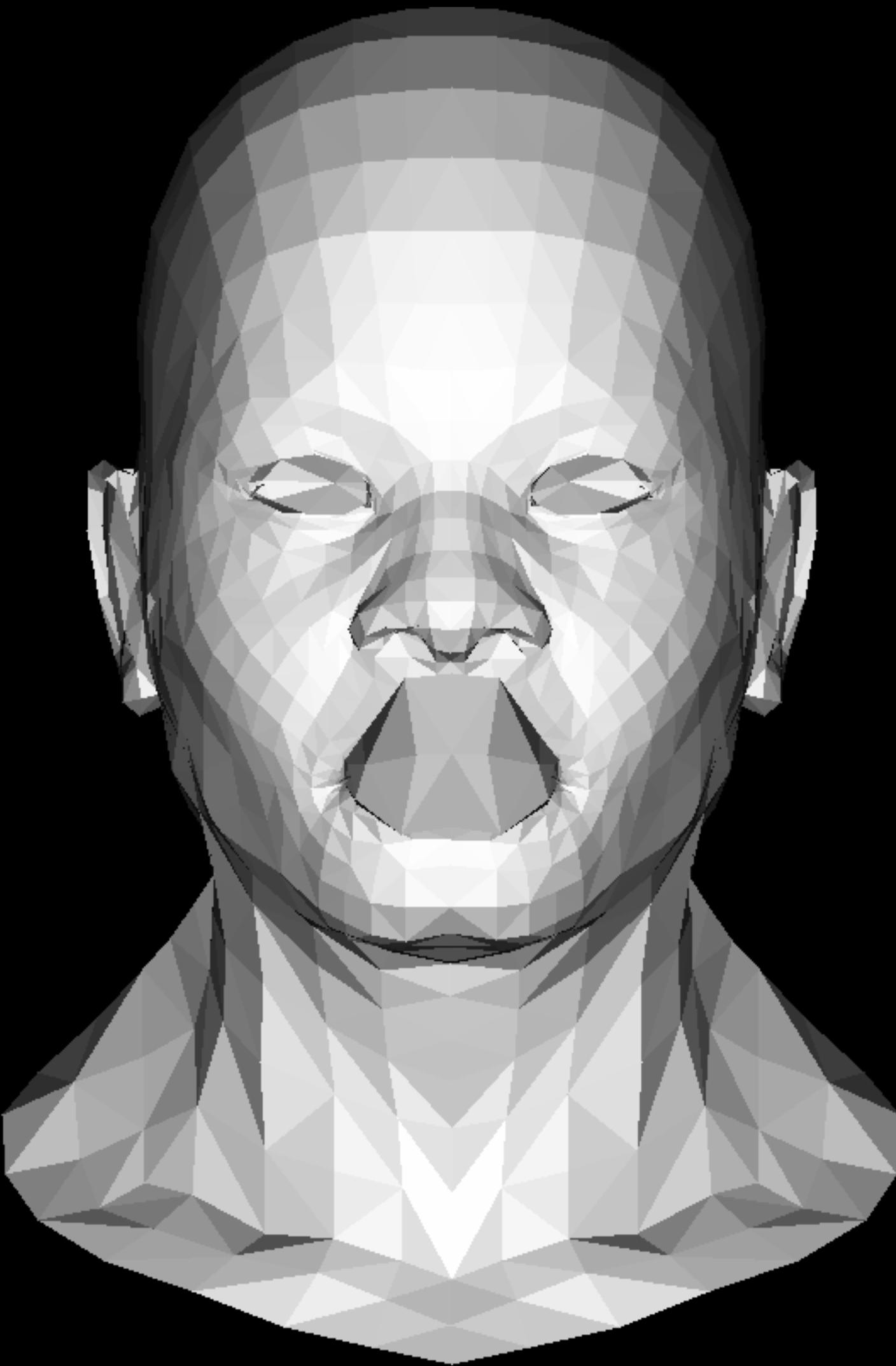
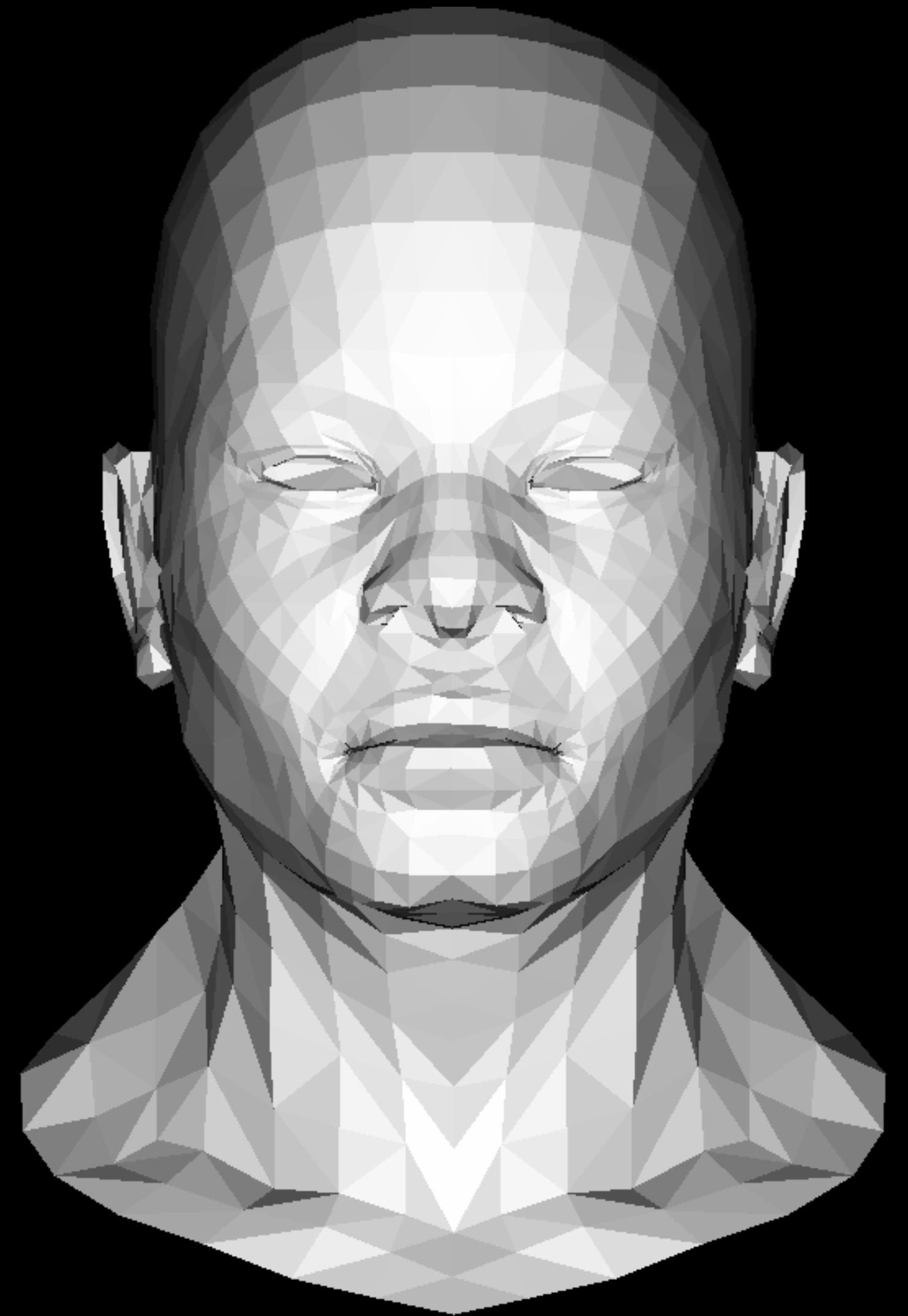
heig-vd



# Ombrage plat (flat shading)

- ◆ Rendu en niveaux de gris
- ◆ Pour simplifier, on utilise  $R = G = B = \text{intensité}$
- ◆ Intensité nulle si l'orientation de la face est parallèle à la direction de la lumière
- ◆ Intensité maximale si elle est perpendiculaire à la direction de la lumière
- ◆ Produit scalaire entre la normale au triangle et la direction de la lumière
- ◆ Et si l'intensité est négative ?





# Ombrage plat (flat shading)

```
const Vec3f light(0,0,-1);

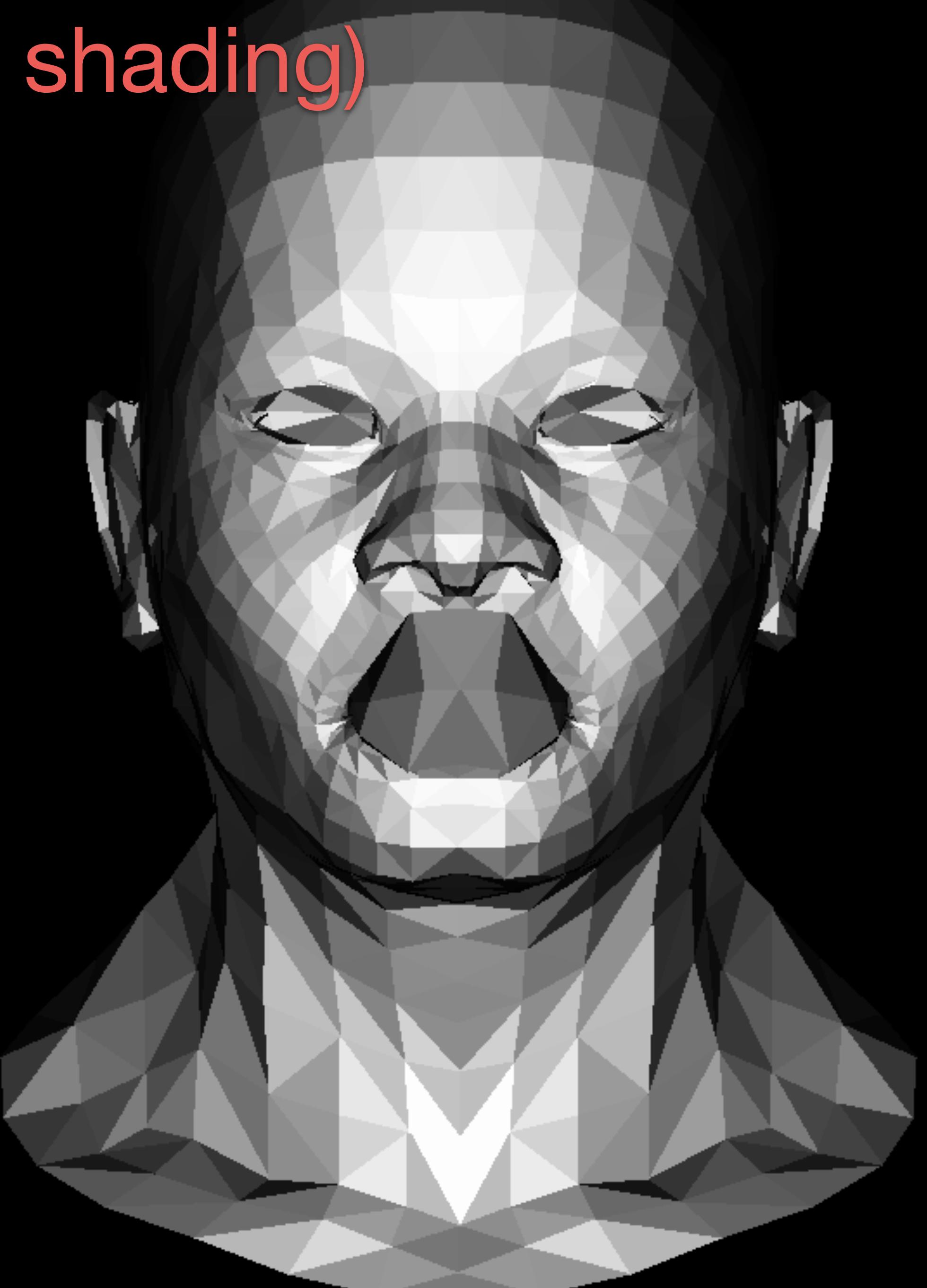
TGAImage image(width, height, TGAImage::RGB);
for (int i=0; i<model->nfaces(); i++) {
    std::vector<int> face = model->face(i);

    Vec2i screen[3];
    Vec3f world[3];

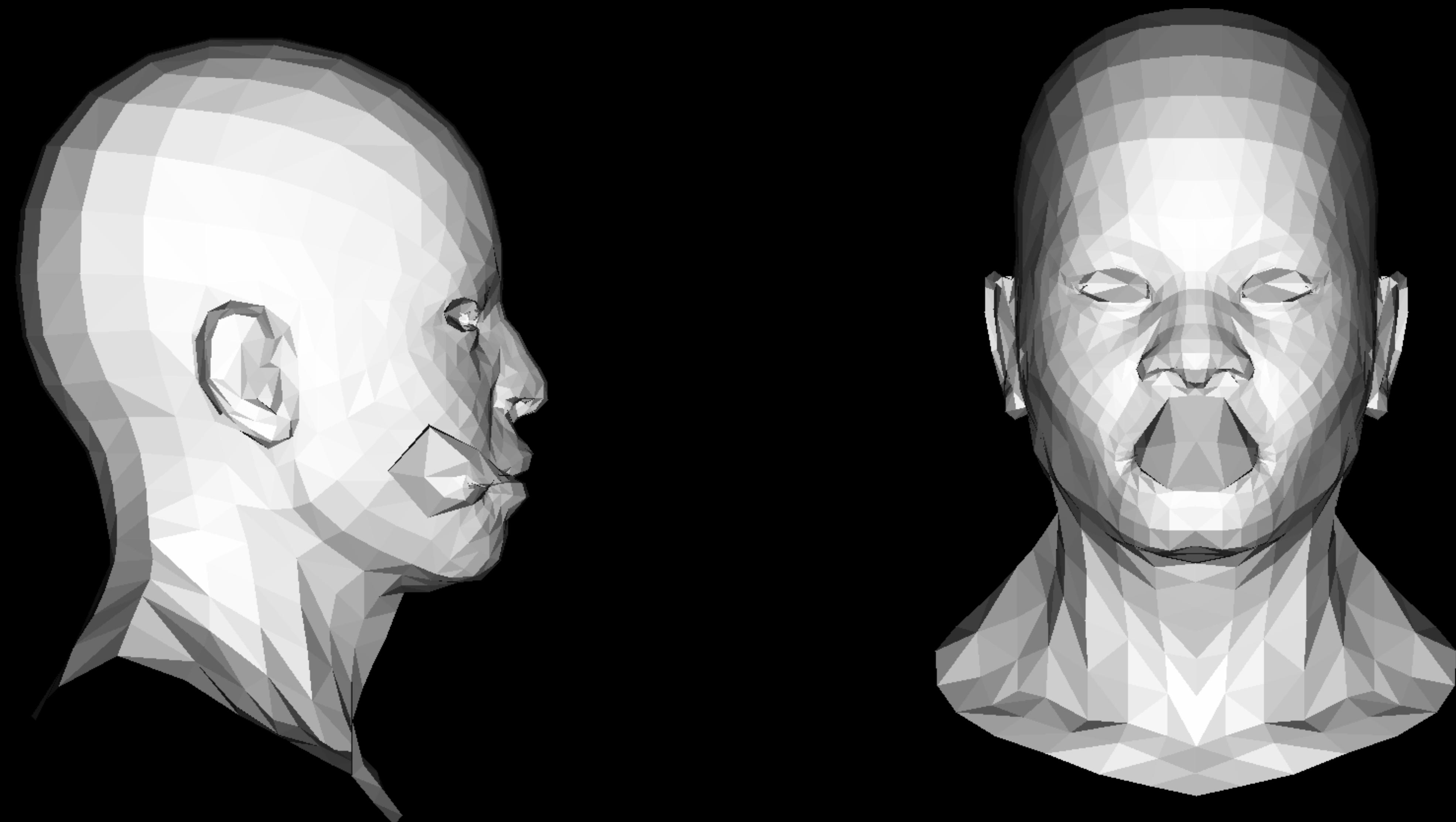
    for (int j=0; j<3; j++) {
        world[j] = model->vert(face[j]);
        screen[j] = Vec2i((world[j].x + 1.) * width / 2.,
                          (world[j].y+1.)*height/2.);
    }

    Vec3f n = (world[2]-world[0])^(world[1]-world[0]);
    n.normalize();
    float I = n * light;

    if(I>=0) {
        TGAColor color(I * 255, I * 255, I * 255, 255);
        triangle(screen, image, color);
    }
}
```



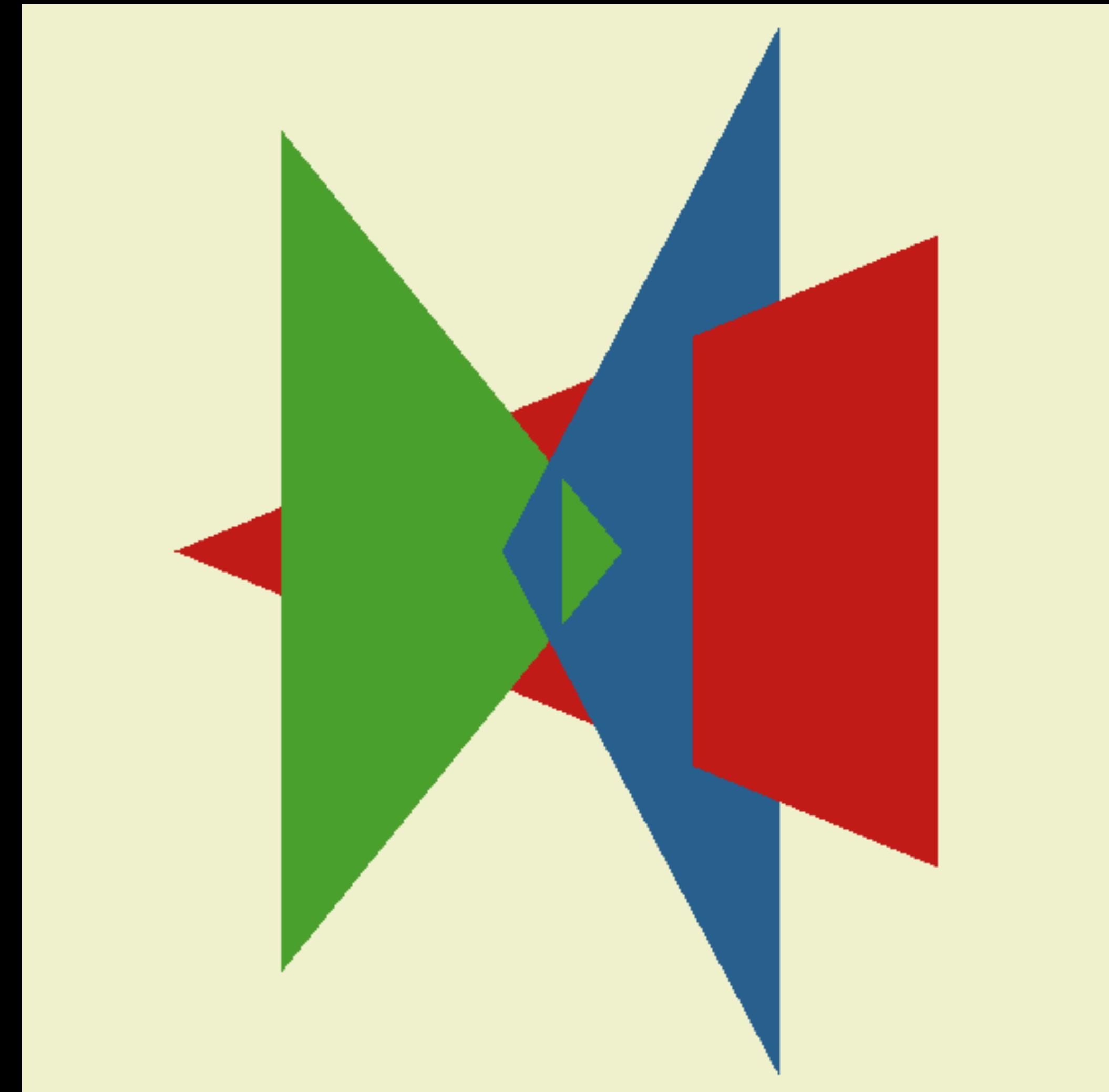
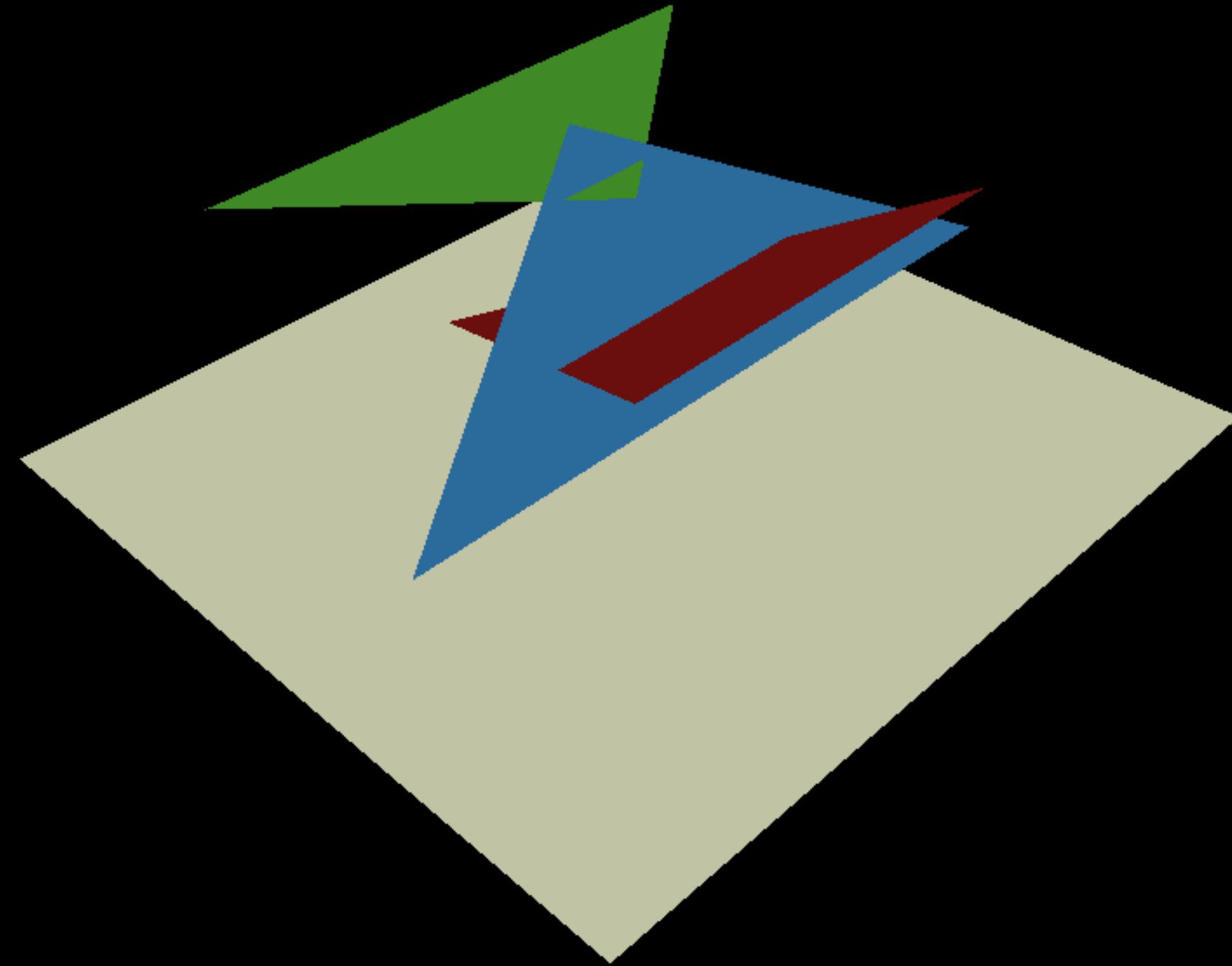
# Que manque-t-il ?



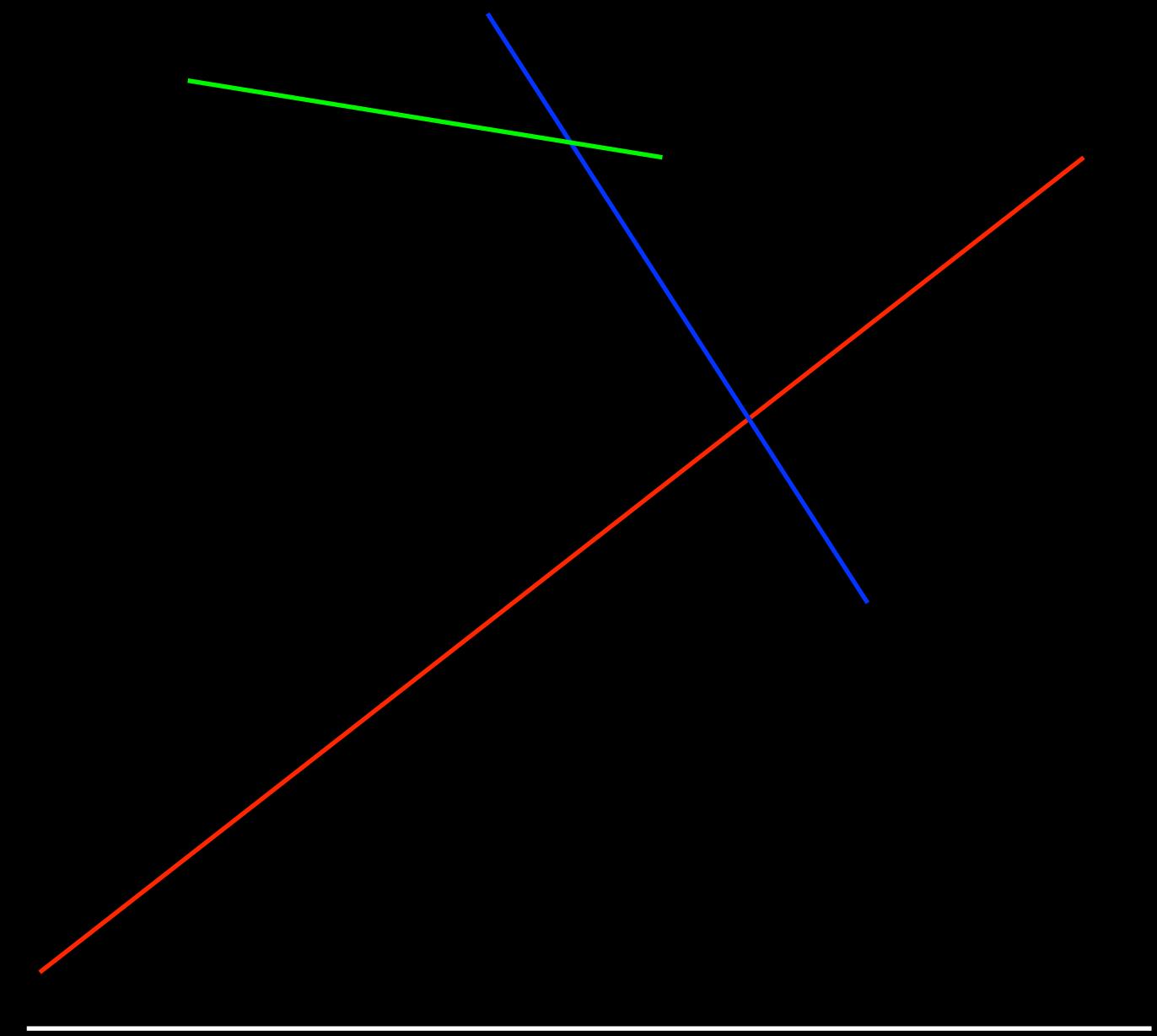
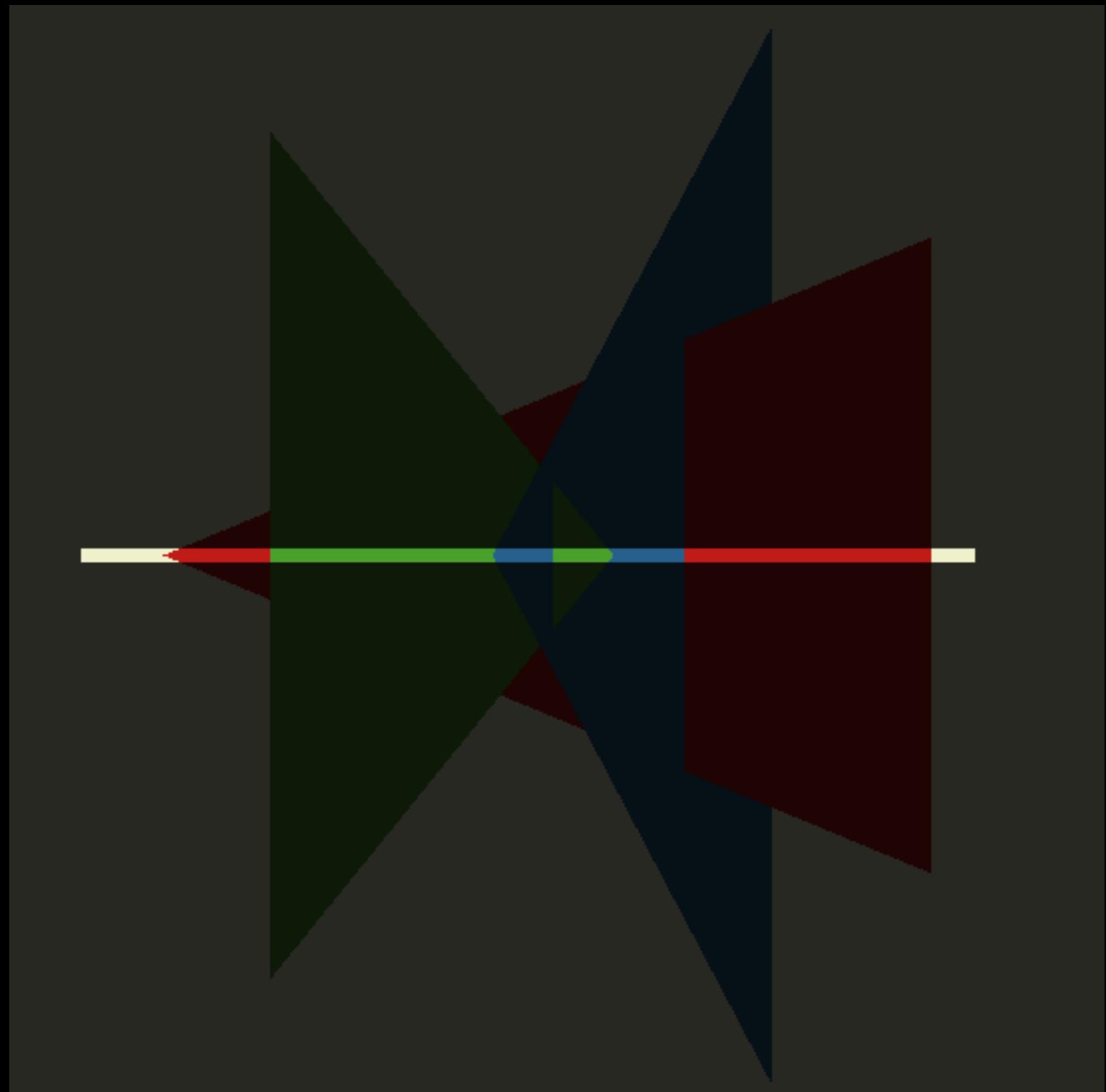
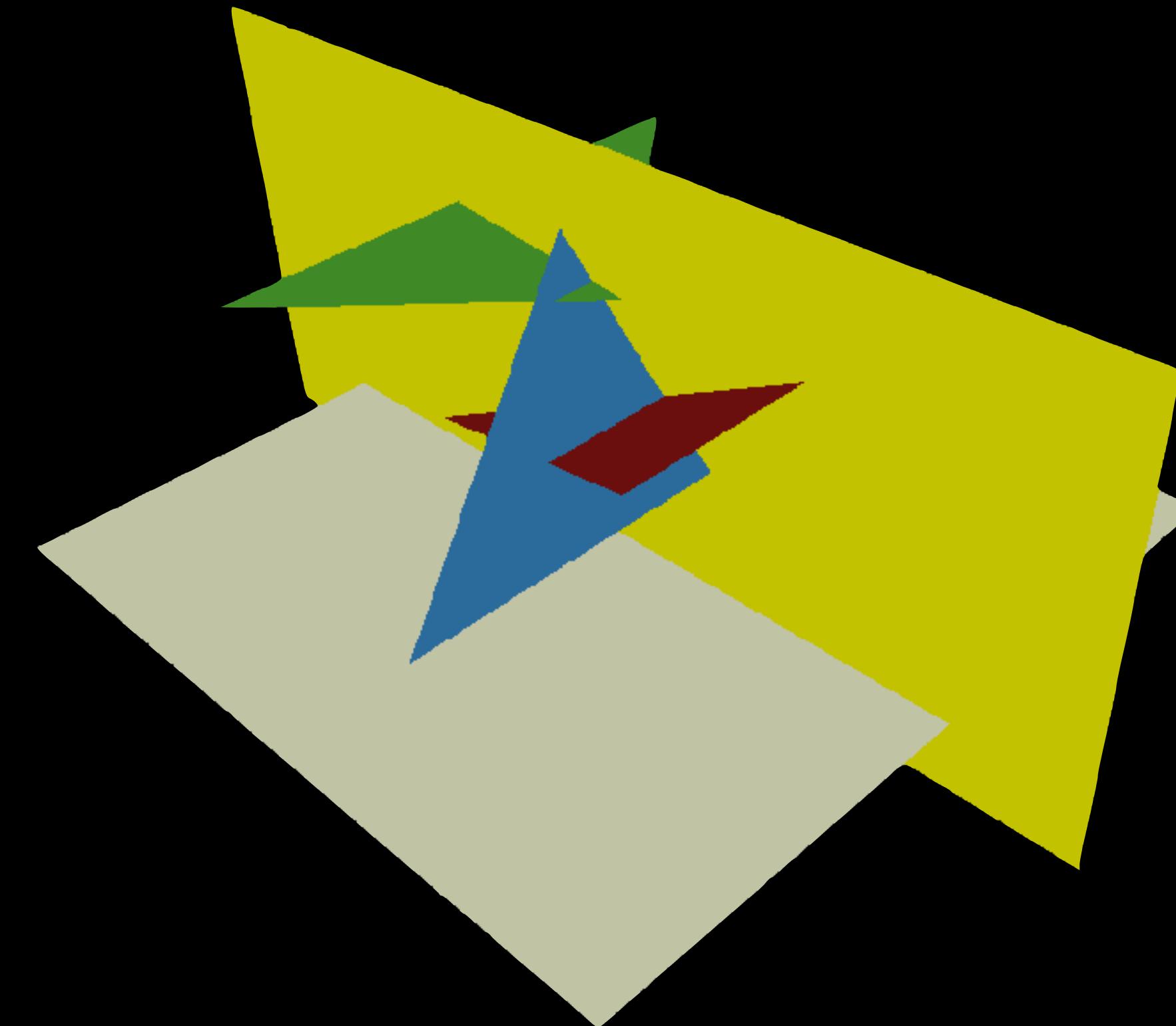
# Z-buffering

# A quoi ressemble cette scène vue d'en haut ?

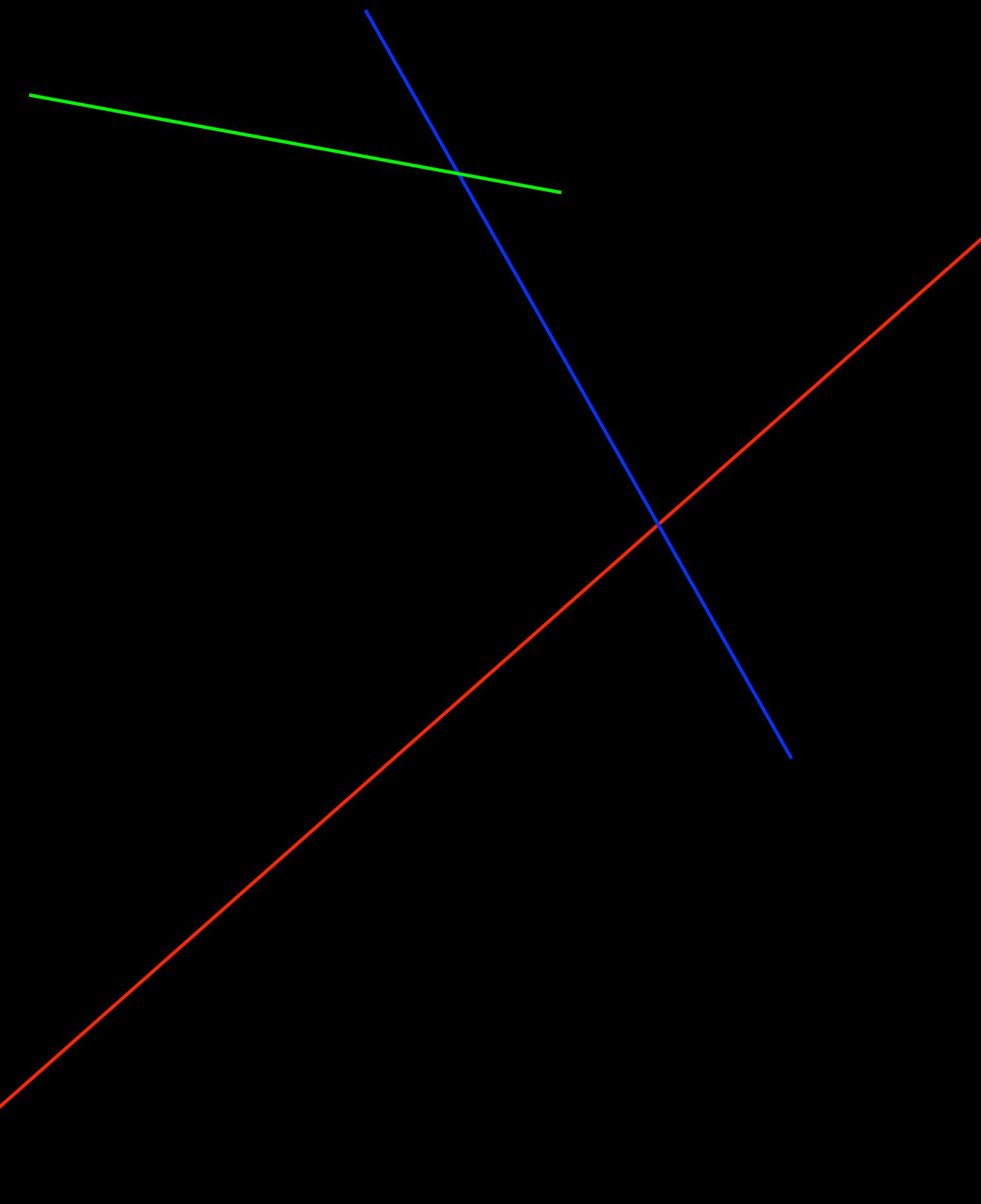
heig-vd



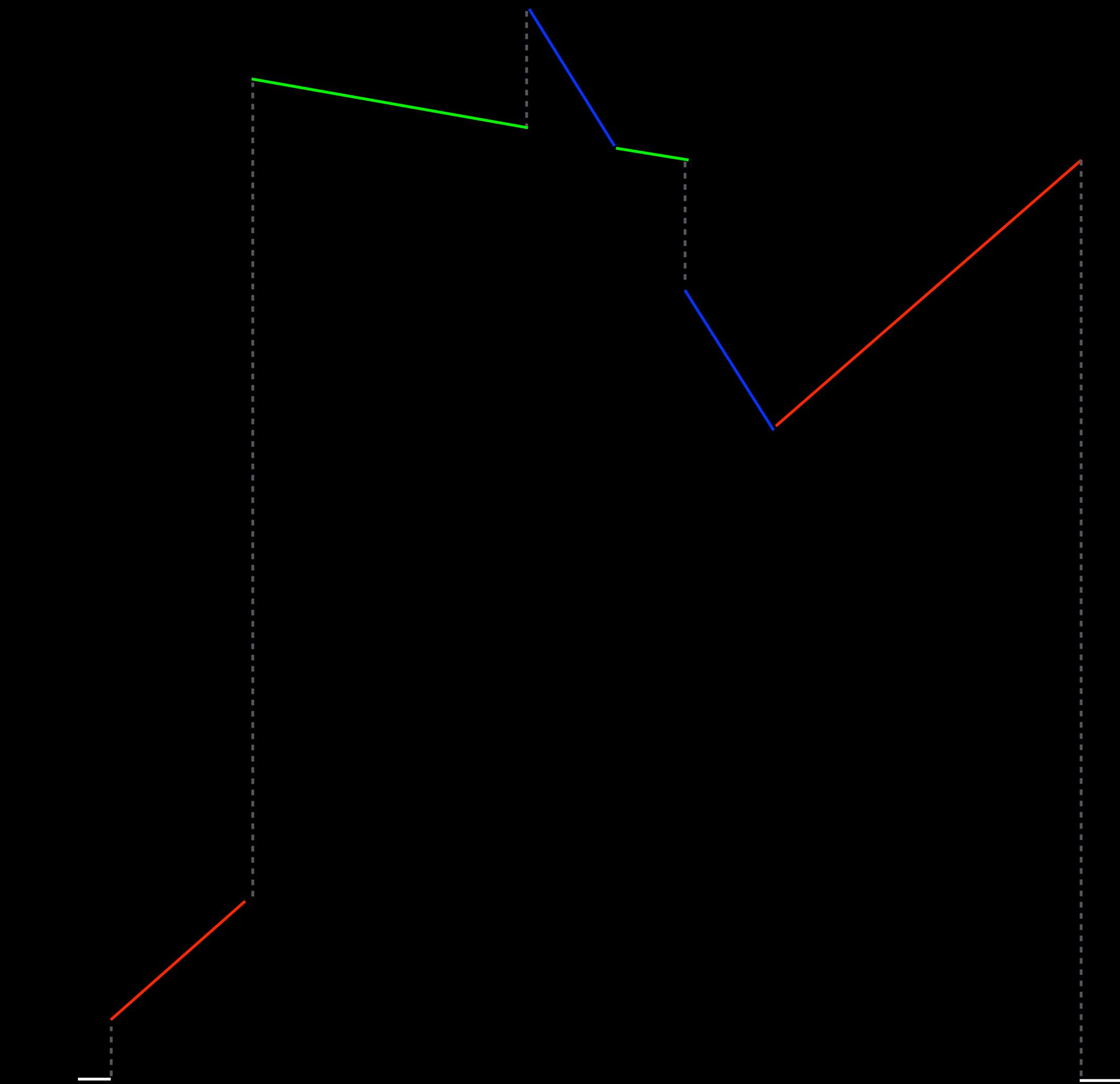
# En 2D



# Les triangles

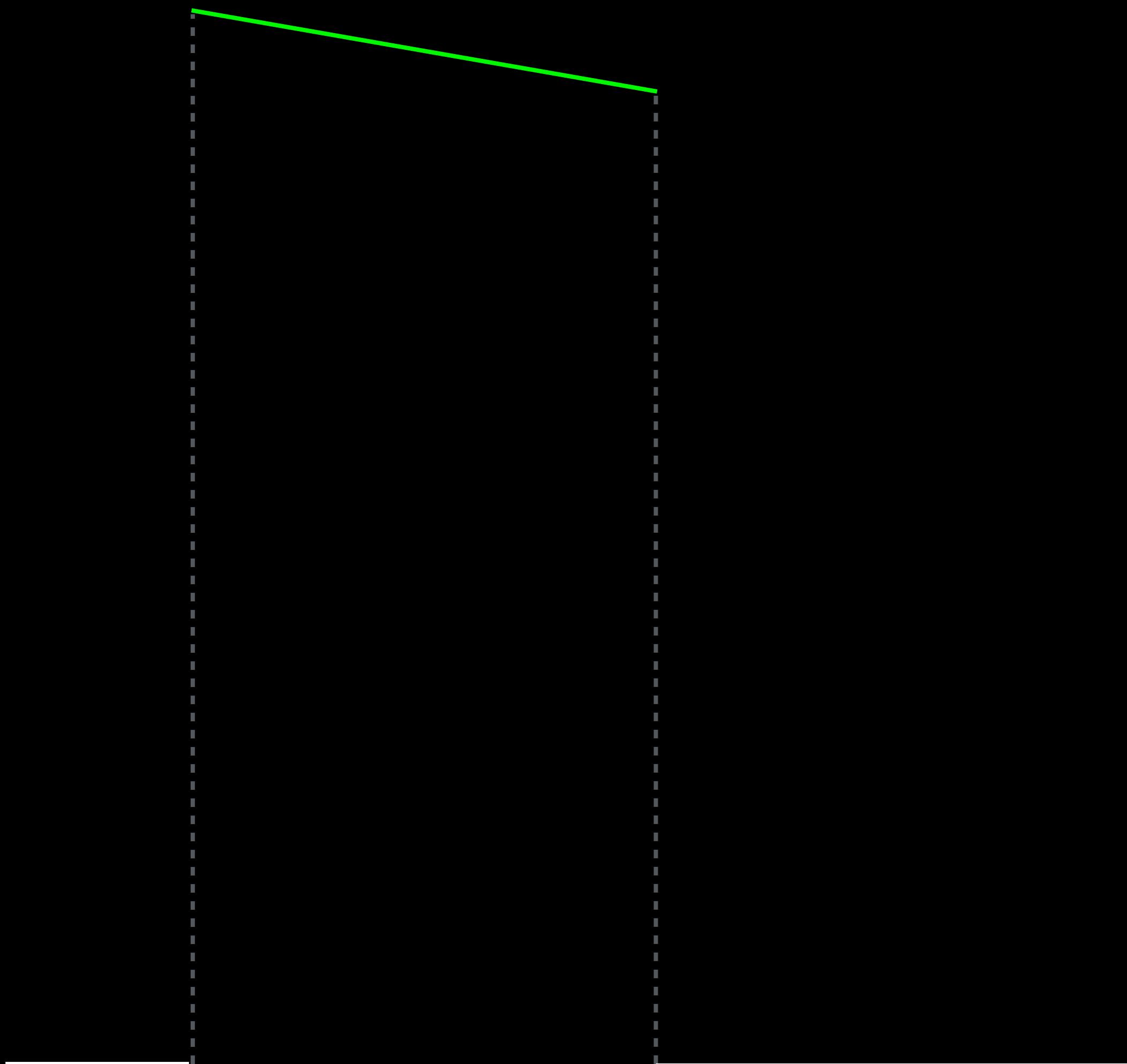


# Les parties visibles

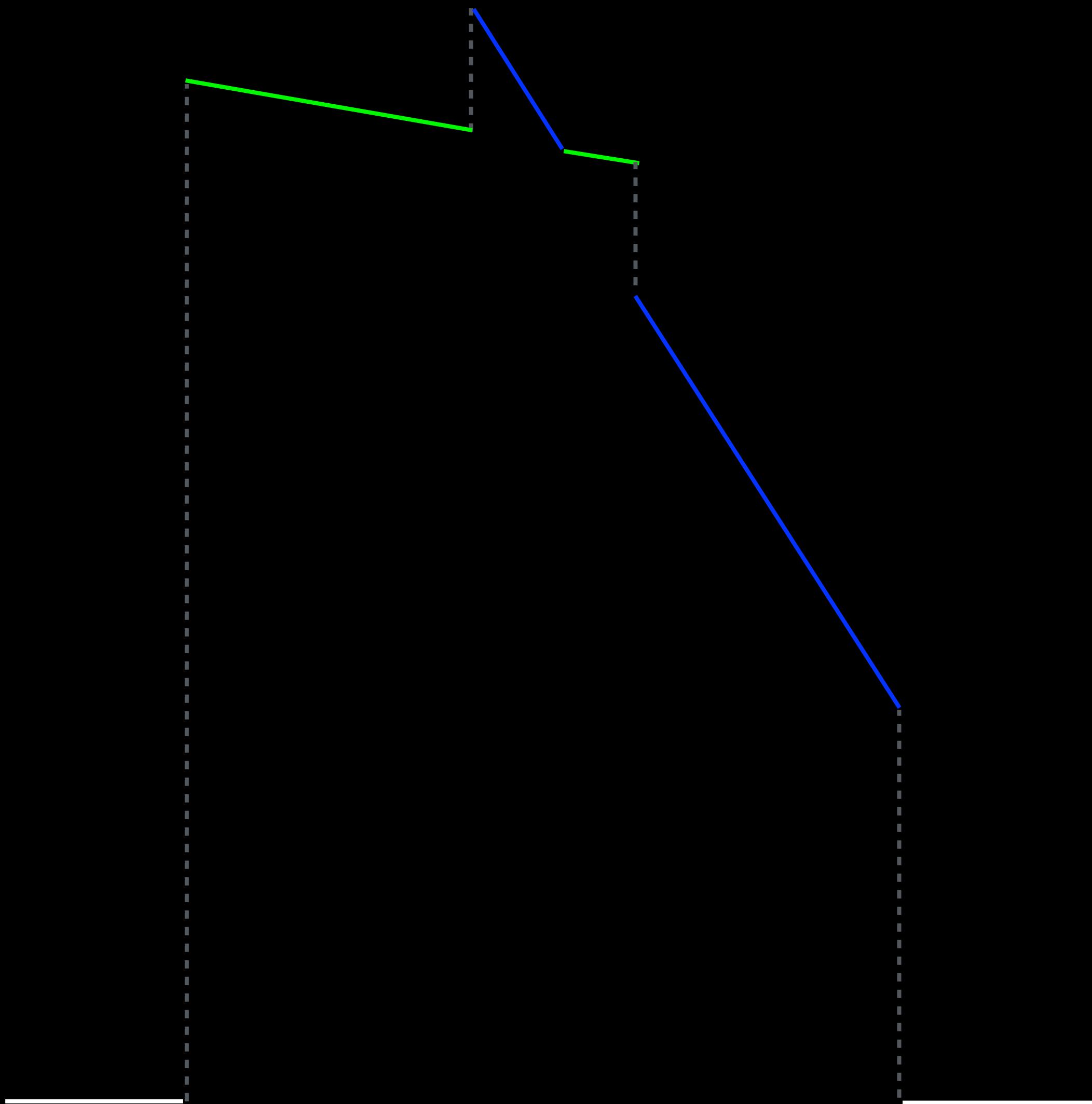


# 1 - Carré blanc

# 1 - triangle vert

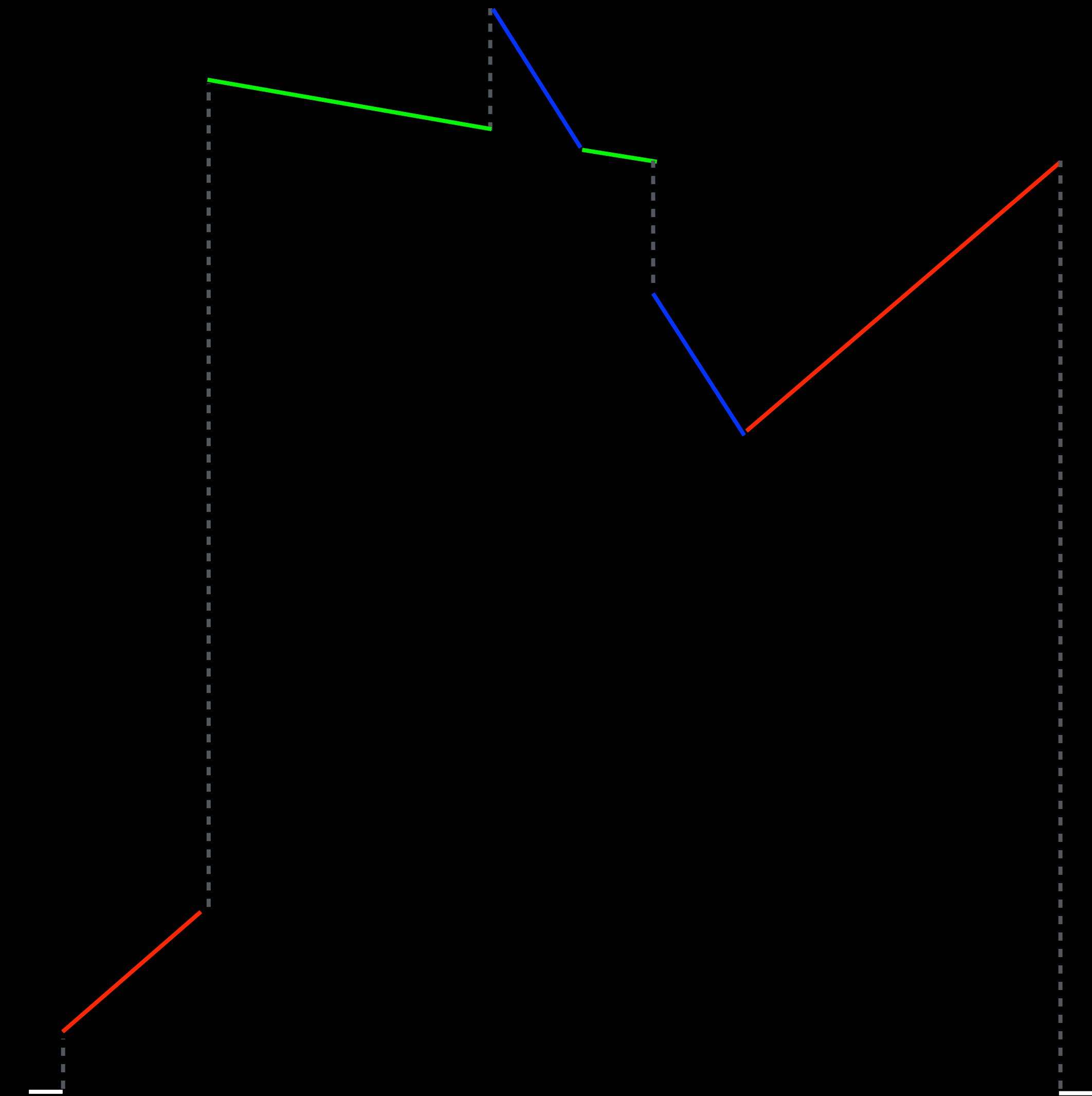


# 2 - triangle bleu



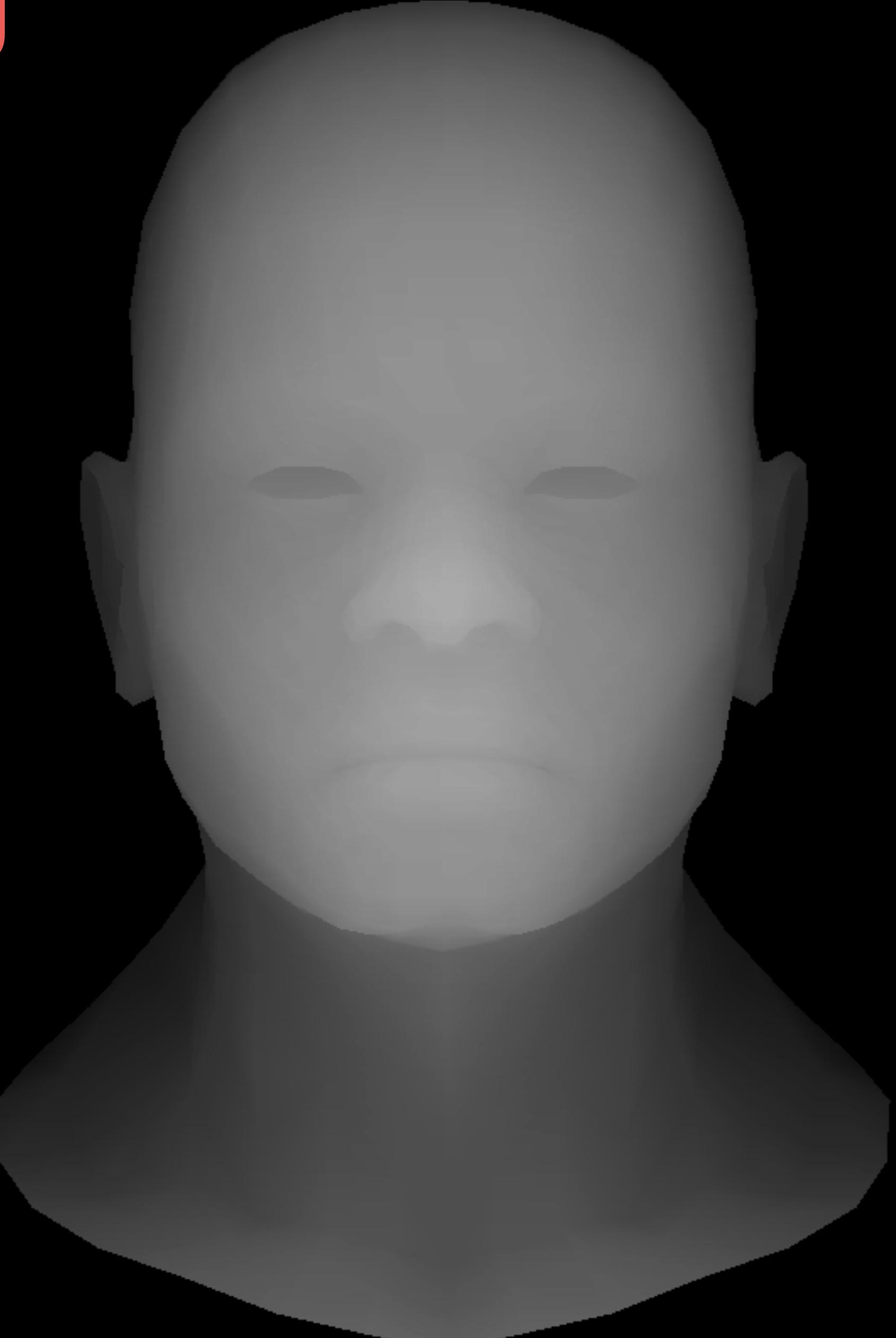
# 3 - triangle rouge

heig-vd



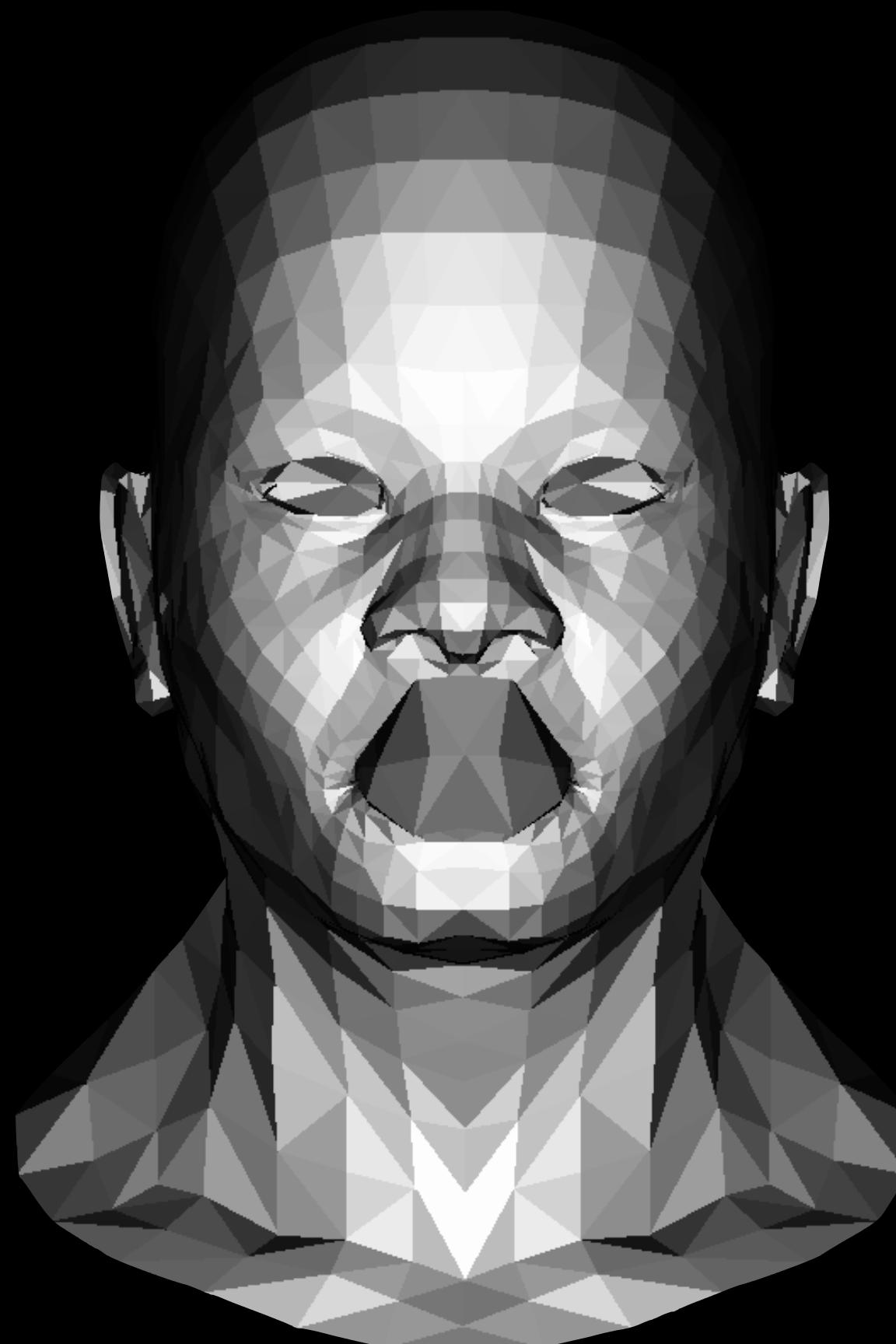
# z-Buffering

- ◆ De la même taille que l'image produite
- ◆ Stocke la proximité du point 3d affiché à la caméra à chaque modification du pixel correspondant
- ◆ Ne modifie un pixel que si le point 3d est plus proche que la valeur stockée
- ◆ Remplace les pixels des triangles plus éloignés
- ◆ Ne touche pas aux pixels des triangles plus proches



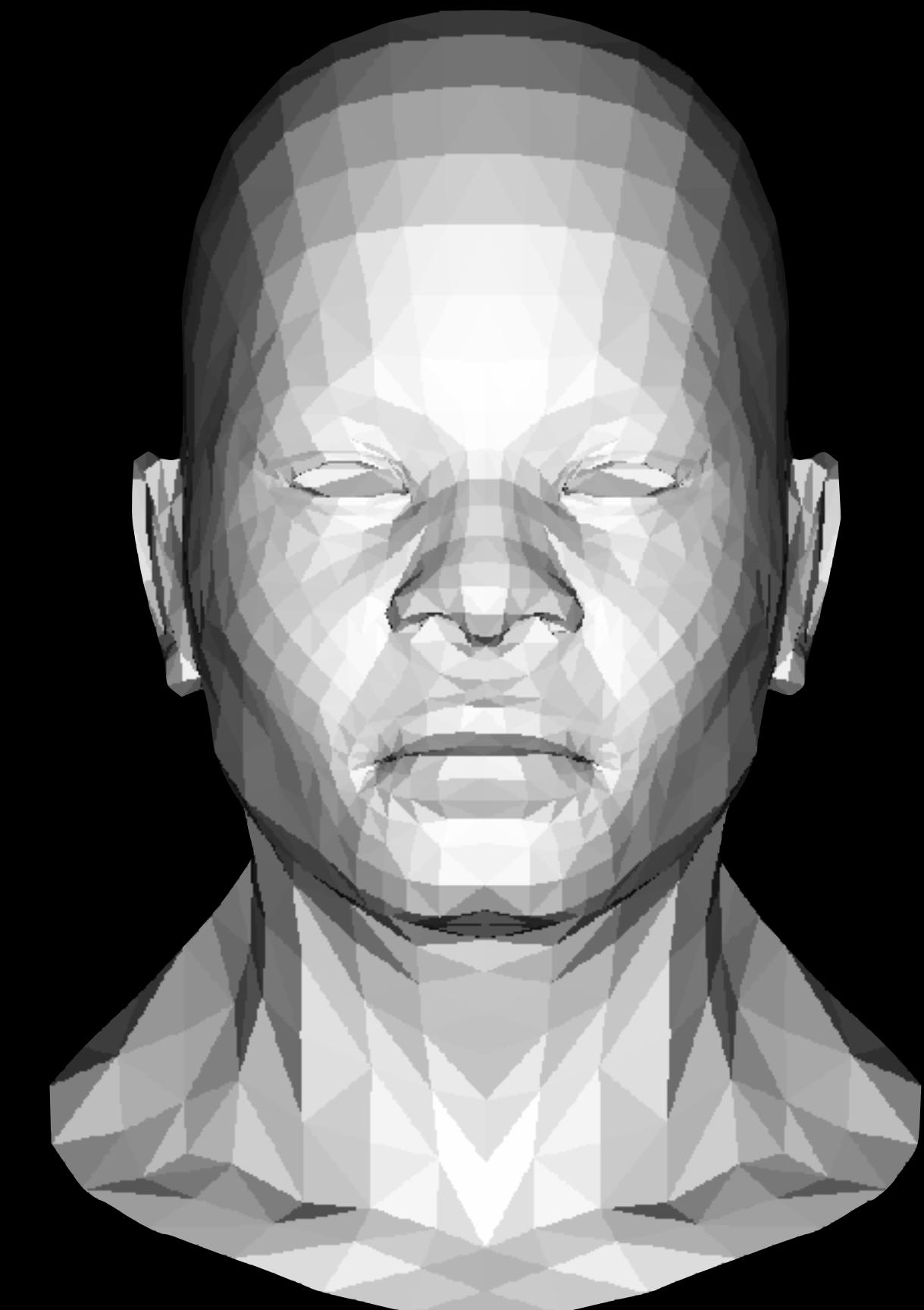
# Code fourni

- ◆ Le code de flat-shading sans z-buffering permettant de générer le rendu suivant



# Votre mission

- ◆ Modifier main() et triangle() pour obtenir le rendu suivant



## Conseils

- ◆ Utiliser des Vec3f pour les coordonnées écran, la troisième dimension servant à coder la profondeur z
- ◆ Allouer un buffer de même taille que l'image de sortie
- ◆ Modifier la fonction triangle pour qu'elle
  - ◆ utilise ce buffer pour décider quels pixels écrire
  - ◆ mette à jour ce buffer à chaque écriture de pixel
- ◆ Interpoler la profondeur d'un pixel à partir des profondeurs des sommets en utilisant ses coordonnées barycentriques

# main( )

- ◆ Allocation et initialisation d'un buffer de la taille de l'image écran
- ◆ Vec3f au lieu de Vec2i pour les coordonnées écran des sommets du triangle.
- ◆ screen[i].z donne la proximité à la caméra
- ◆ Passage du zBuffer et des Vec3f à la fonction triangle

```

float zbuffer[width*height];
for(unsigned i = 0; i < width*height; ++i)
    zbuffer[i] = std::numeric_limits<float>::lowest();

for (int i=0; i<model->nfaces(); i++) {
    std::vector<int> face = model->face(i);

    Vec3f screen[3];
    Vec3f world[3];

    for (int j=0; j<3; j++) {
        world[j] = model->vert(face[j]);
        screen[j] = Vec3f((world[j].x + 1.) * width / 2.,
                          (world[j].y+1.)*height/2., world[j].z);
    }

    Vec3f n = (world[2]-world[0])^(world[1]-world[0]);
    n.normalize();
    float I = n * light;

    if(I>=0) {
        TGAColor color(I * 255, I * 255, I * 255, 255);
        triangle(screen, zbuffer, image, color);
    }
}

```

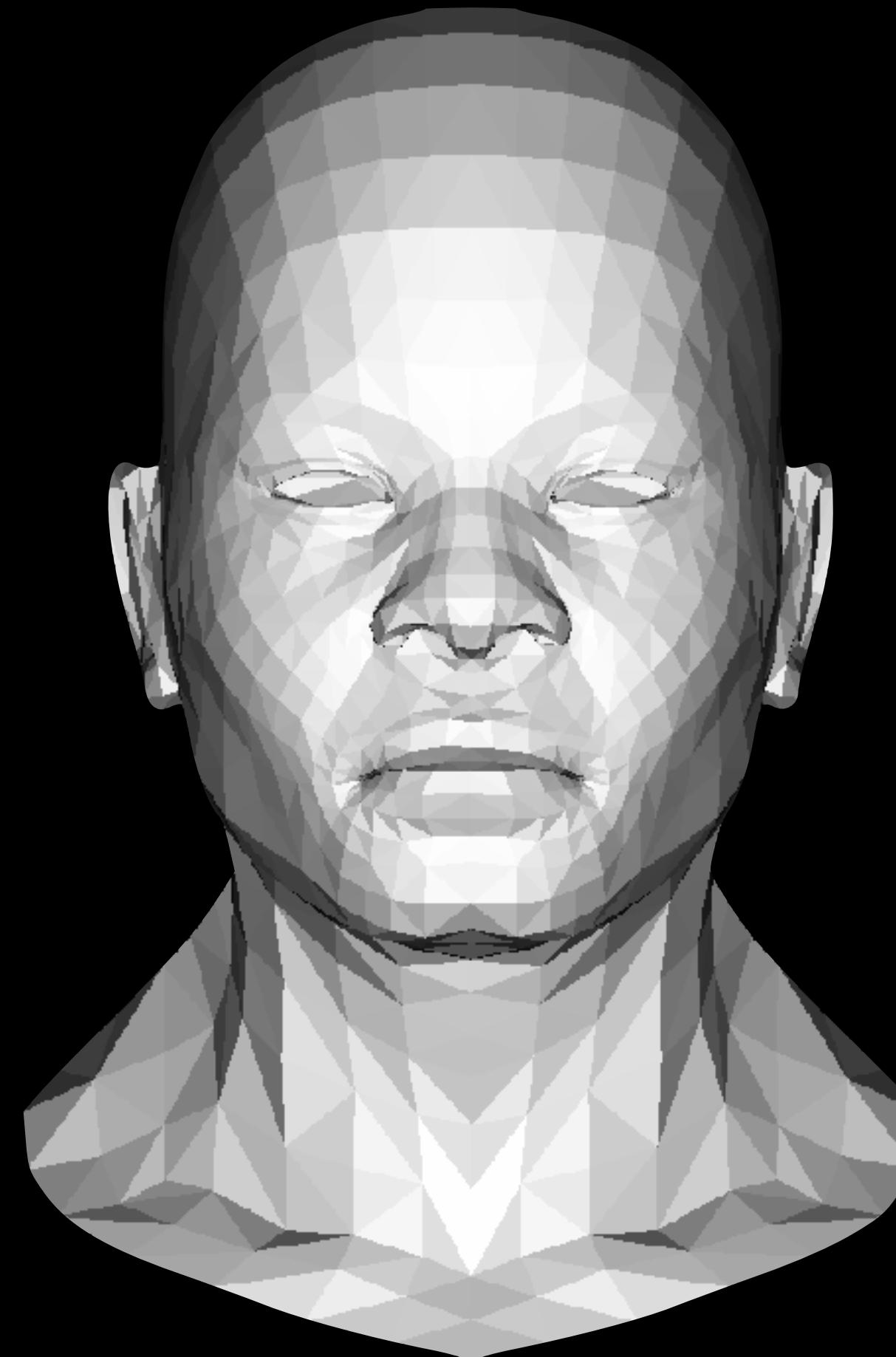
# triangle(...)

- ♦ Le point courant p est aussi un Vec3f
- ♦ Sa coordonnée z est interpolée de celles des sommets du triangle grâce aux coordonnées barycentriques
- ♦ On n'affiche que si p.z est plus proche de la caméra que le dernier point affiché à cet endroit, dont le p.z a été stocké dans le zbuffer
- ♦ Les APIs de boite\_englobante et de barycentriques changent, mais pas leurs contenus

```
void triangle(Vec3f *t, float* zbuffer,
              TGAImage &image, TGAColor color)
{
    auto bbox = boite_englobante(image, t);
    for(int y = bbox[0].y; y <= bbox[1].y; ++y)
    {
        for(int x = bbox[0].x; x <= bbox[1].x; ++x)
        {
            Vec3f p { float(x), float(y), 0 };
            Vec3f b = barycentriques(t, p);
            if ( b.x >= 0 and b.y >= 0 and b.z >= 0 )
            {
                p.z = b.x*t[0].z + b.y*t[1].z + b.z*t[2].z;
                if(zbuffer[x + image.get_width() * y] < p.z)
                {
                    zbuffer[x + image.get_width() * y] = p.z;
                    image.set(p.x, p.y, color);
                }
            }
        }
    }
}
```

# Textures

# Texture...



+



=



Coordonnées (u,v) stockées pour chaque sommet...





# Un autre modèle texturé



# Code fourni

- ◆ Un meilleur lecteur de modèle
- ◆ lisant les lignes « vt %f %f %f » du fichier african\_head.obj contenant les coordonnées de texture
- ◆ parsant mieux les lignes « f %d/%d/%d %d/%d/%d %d/%d/%d », les deuxièmes entiers contenant l'indice de texture de chaque sommet
- ◆ Un fichier principal mettant en oeuvre le flat shading avec z-buffering
- ◆ Le fichier de texture african\_head\_diffuse.tga

# Votre mission

- ◆ Modifier main() et triangle() pour obtenir le rendu suivant



# main()

```
for (int i=0; i<model.nfaces(); i++) {
    Vec3f screen[3];
    Vec3f world[3];
    Vec3f text[3];

    for (int j=0; j<3; j++) {
        world[j] = model.vert(model.face(i)[j]);
        screen[j] = Vec3f(
            (world[j].x + 1.) * width / 2.,
            (world[j].y + 1.) * height / 2.,
            world[j].z );
        text[j] = model.texture(model.face_texts(i)[j]);
    }

    Vec3f n = (world[2]-world[0])^(world[1]-world[0]);
    n.normalize();
    float I = n * light;

    if(I>=0) {
        triangle(screen, text, zbuffer,
                  image, texture, I);
    }
}
```

```
void triangle(Vec3f *t, Vec3f* tc, float* zbuffer, TGAImage &image, TGAImage &texture, float intensity)
{
    auto bbox = boite_englante(image, t);
    for(int y = bbox[0].y; y <= bbox[1].y; ++y) {
        for(int x = bbox[0].x; x <= bbox[1].x; ++x)
        {
            Vec3f p { float(x), float(y), 0};
            Vec3f b = barycentriques(t, p);

            if ( b.x >= 0 and b.y >= 0 and b.z >= 0 )
            {
                p.z = b.x*t[0].z + b.y*t[1].z + b.z*t[2].z;

                if(zbuffer[x + image.get_width() * y] < p.z)
                {
                    zbuffer[x + image.get_width() * y] = p.z;

                    Vec3f ptc = tc[0]*b.x + tc[1]*b.y + tc[2]*b.z;
                    Vec2i pc ( ptc.x * texture.get_width(), ptc.y * texture.get_height() );
                    TGAColor color = texture.get(pc.x, pc.y);

                    for(int i = 0; i < 3; ++i)
                        color.raw[i] *= intensity;

                    image.set(x, y, color);
                }
            }
        }
    }
}
```

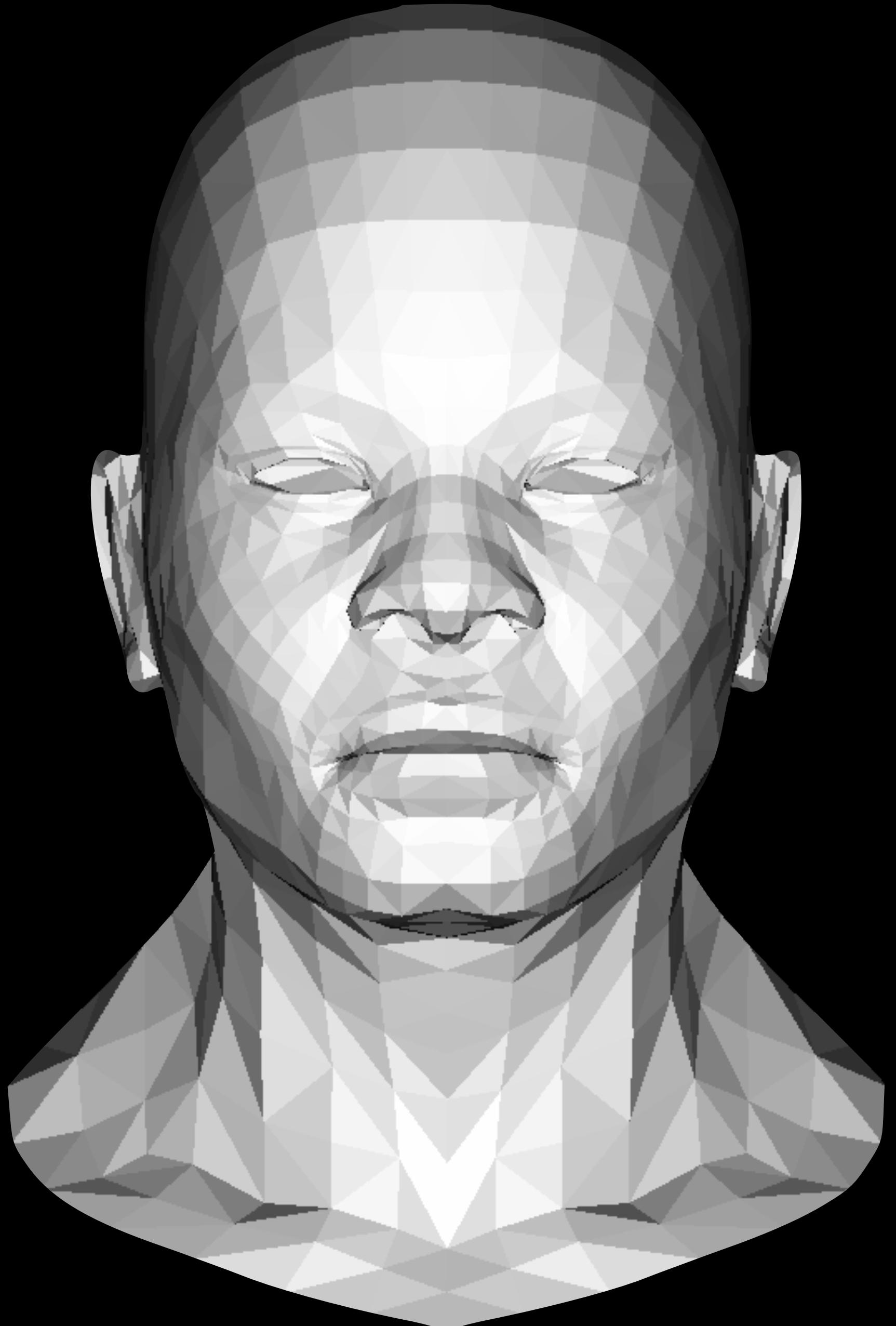
# Ombrage de Gouraud



# No Shading



# Flat Shading



# Gouraud Shading



# Ombrage de Gouraud

- ♦ Une normale lue / calculée en chaque sommet du modèle
- ♦ Une intensité calculée pour chaque sommet par produit scalaire
- ♦ L'intensité d'un point est interpolée à partir des intensité des sommets du triangle auquel il appartient

A vous de jouer...

- ♦ Modifiez votre code de flat-shading texturé



# main( )

```
for (int i=0; i<model.nfaces(); i++) {  
    Vec3f screen[3];  
    Vec3f world[3];  
    Vec3f text[3];  
    Vec3f intensity; ←  
  
    for (int j=0; j<3; j++) {  
        world[j] = model.vert(model.face(i)[j]);  
        screen[j] = Vec3f((world[j].x + 1.) * width / 2.,  
                          (world[j].y+1.)*height/2., world[j].z);  
        text[j] = model.texture(model.face_texts(i)[j]);  
        Vec3f n = model.normal(model.face_normals(i)[j]) * -1; ←  
        intensity.raw[j] = n * light;  
    }  
  
    // Vec3f n = (world[2]-world[0])^(world[1]-world[0]);  
    // if(n * light>=0) {  
  
        triangle(screen, text, zbuffer, image, texture, intensity);  
  
    }  
}
```

```
void triangle(Vec3f t[3], Vec3f tc[3], float* zbuffer,
              TGAImage &image, TGAImage &texture,
              Vec3f intensity) ←
{
    auto bbox = boite_englante(image, t);
    for(int y = bbox[0].y; y <= bbox[1].y; ++y) {
        for(int x = bbox[0].x; x <= bbox[1].x; ++x) {
            Vec3f p { float(x), float(y), 0 };
            Vec3f b = barycentriques(t,p);
            if ( b.x >= 0 and b.y >= 0 and b.z >= 0 ) {
                p.z = b.x*t[0].z + b.y*t[1].z + b.z*t[2].z;
                if(zbuffer[x + image.get_width() * y] < p.z)
                {
                    zbuffer[x + image.get_width() * y] = p.z;

                    Vec3f ptc = tc[0]*b.x + tc[1]*b.y + tc[2]*b.z;
                    Vec2i pc ( ptc.x * texture.get_width() , ptc.y * texture.get_height() );
                    TGAColor color = texture.get(pc.x, pc.y);

                    float gouraud = intensity * b;
                    if(gouraud < 0) gouraud = 0;
                    for(int i = 0; i < 3; ++i)
                        color.raw[i] *= gouraud;

                    image.set(x, y, color);
                }
            }
        }
    }
}
```

# Transformations et projections

# Comment remplacer cette ligne, dessiner en perspective, bouger la caméra, ... ?

heig-vd

```
screen[j] = Vec3f((world[j].x + 1.) * width / 2.,  
                  (world[j].y + 1.) * height/2.,  
                  world[j].z);
```



# Transformations linéaires

- ♦ Ecrites sous forme matricielle

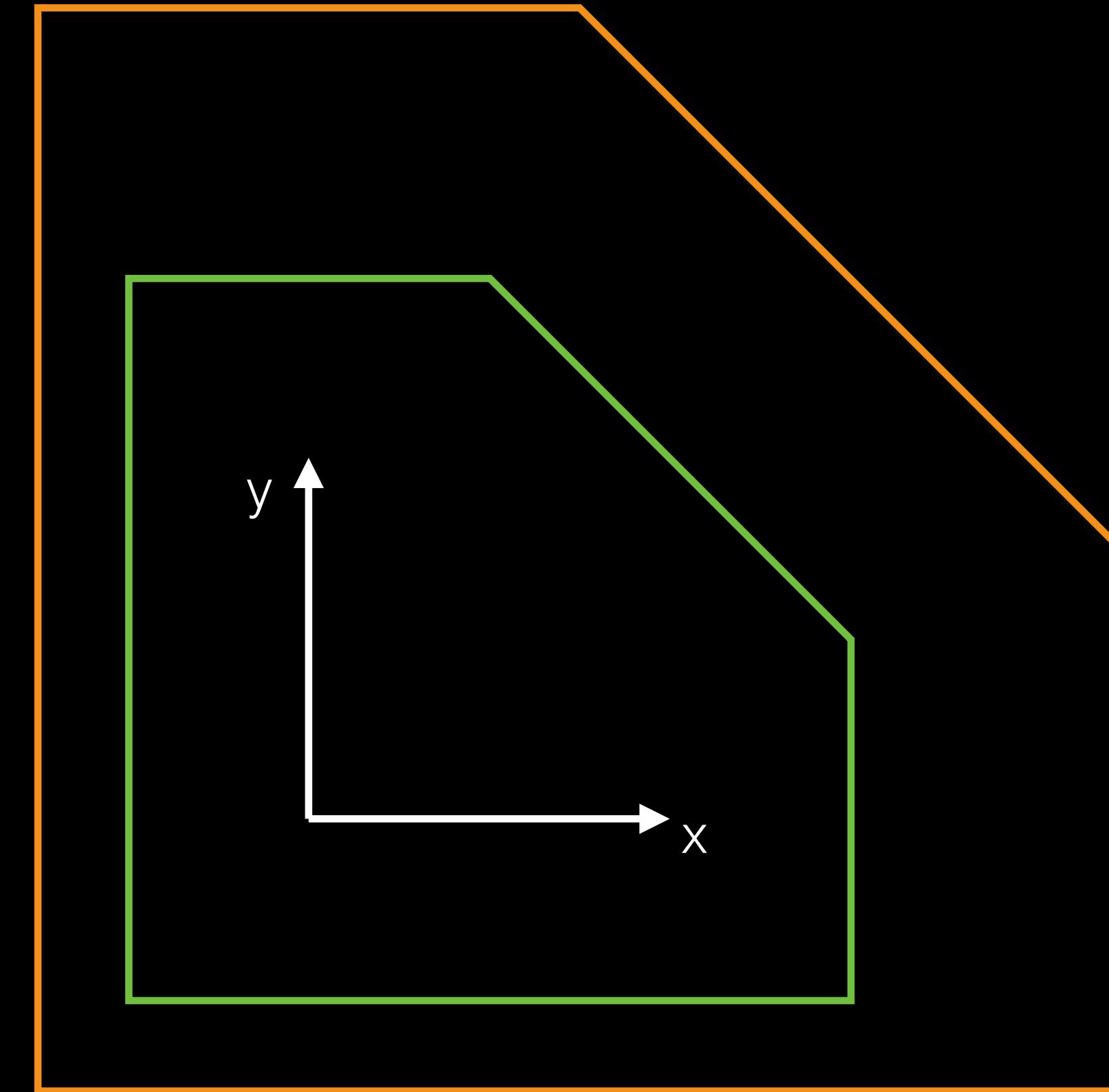
$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix}$$

- ♦ Cas le plus simple : transformation identité

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}$$

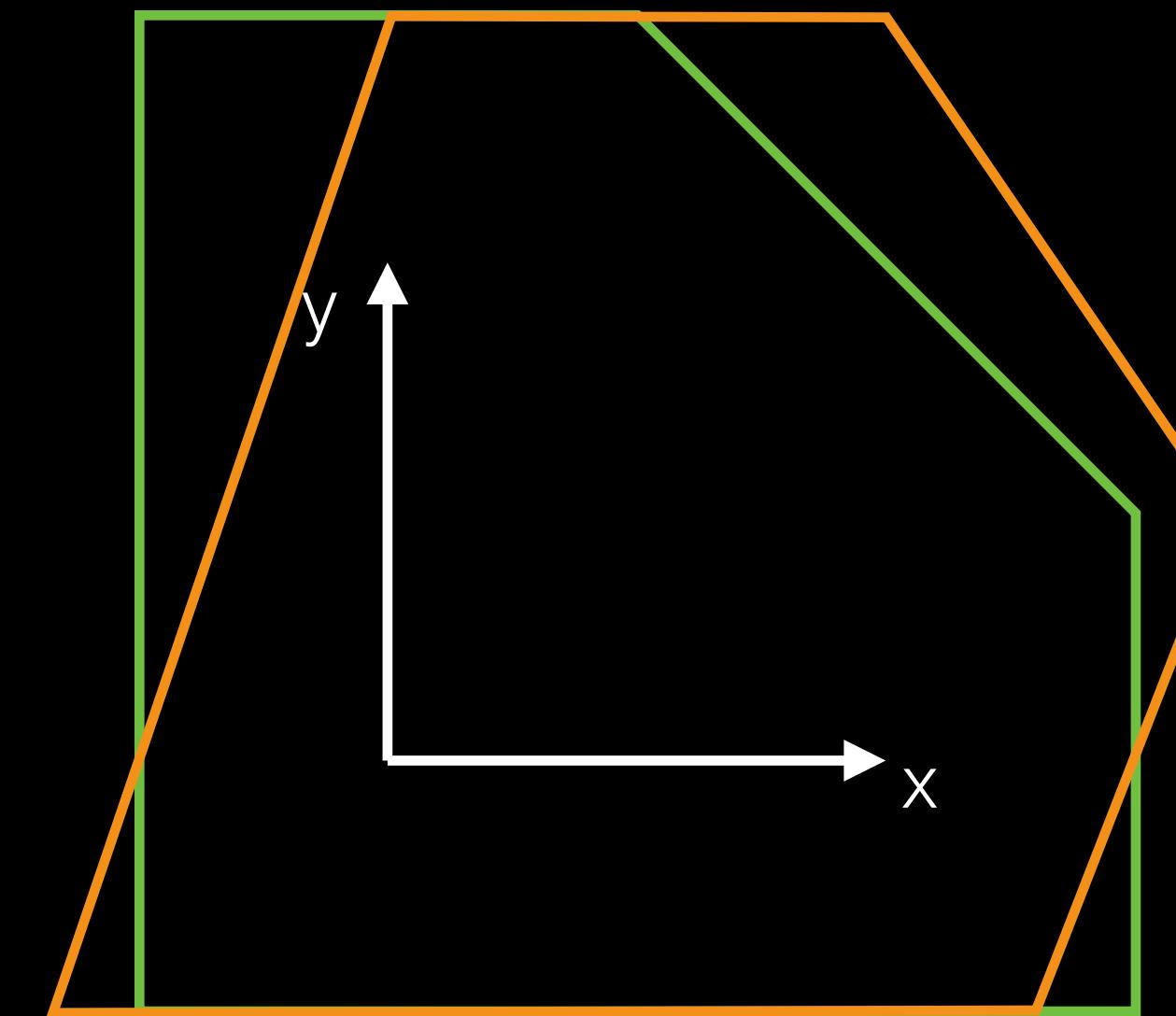
# Mise à l'échelle

$$\begin{pmatrix} 3/2 & 0 \\ 0 & 3/2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 3/2 \cdot x \\ 3/2 \cdot y \end{pmatrix}$$



# Biais

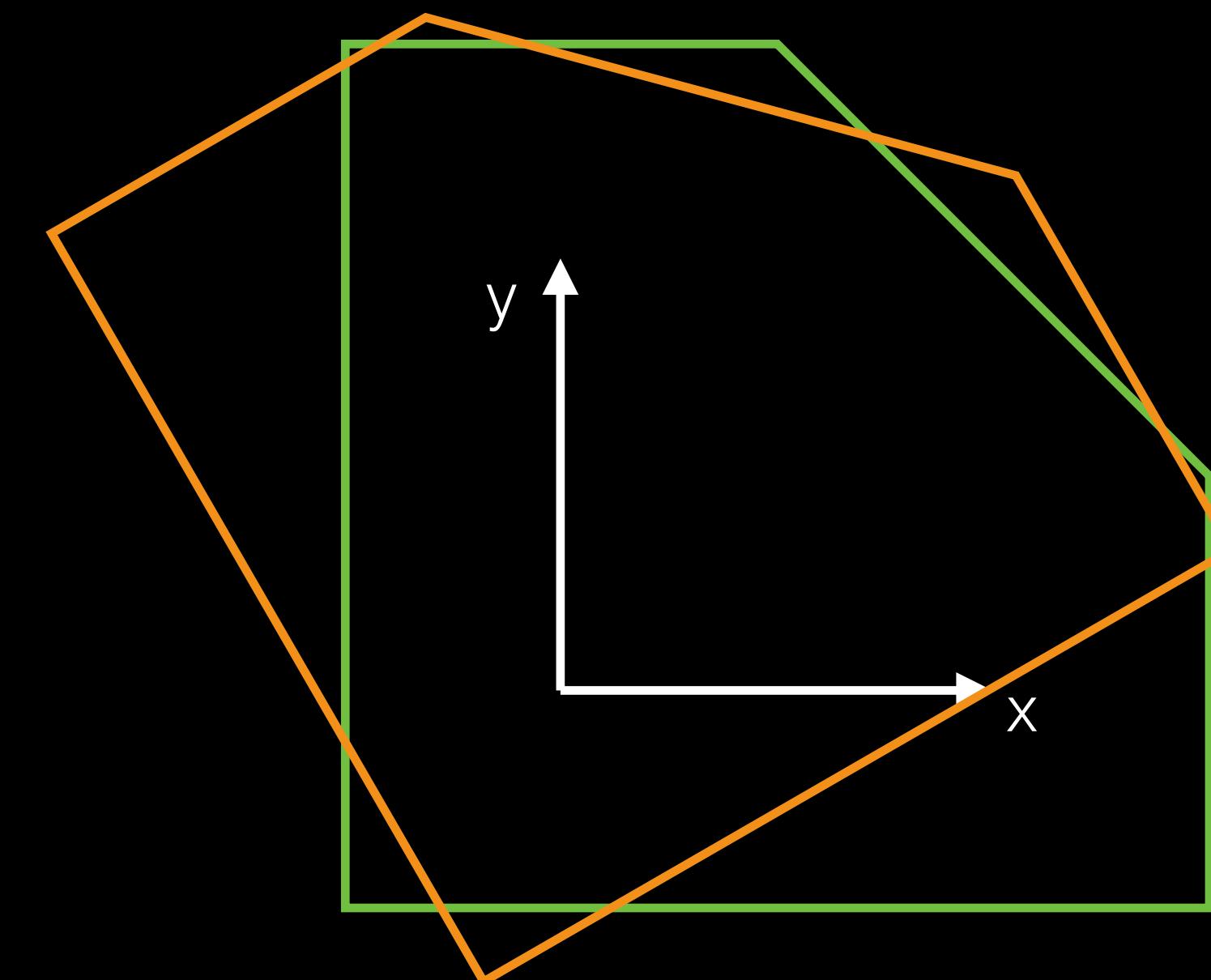
$$\begin{pmatrix} 1 & 1/3 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x + \frac{y}{3} \\ y \end{pmatrix}$$



# Rotation

heig-vd

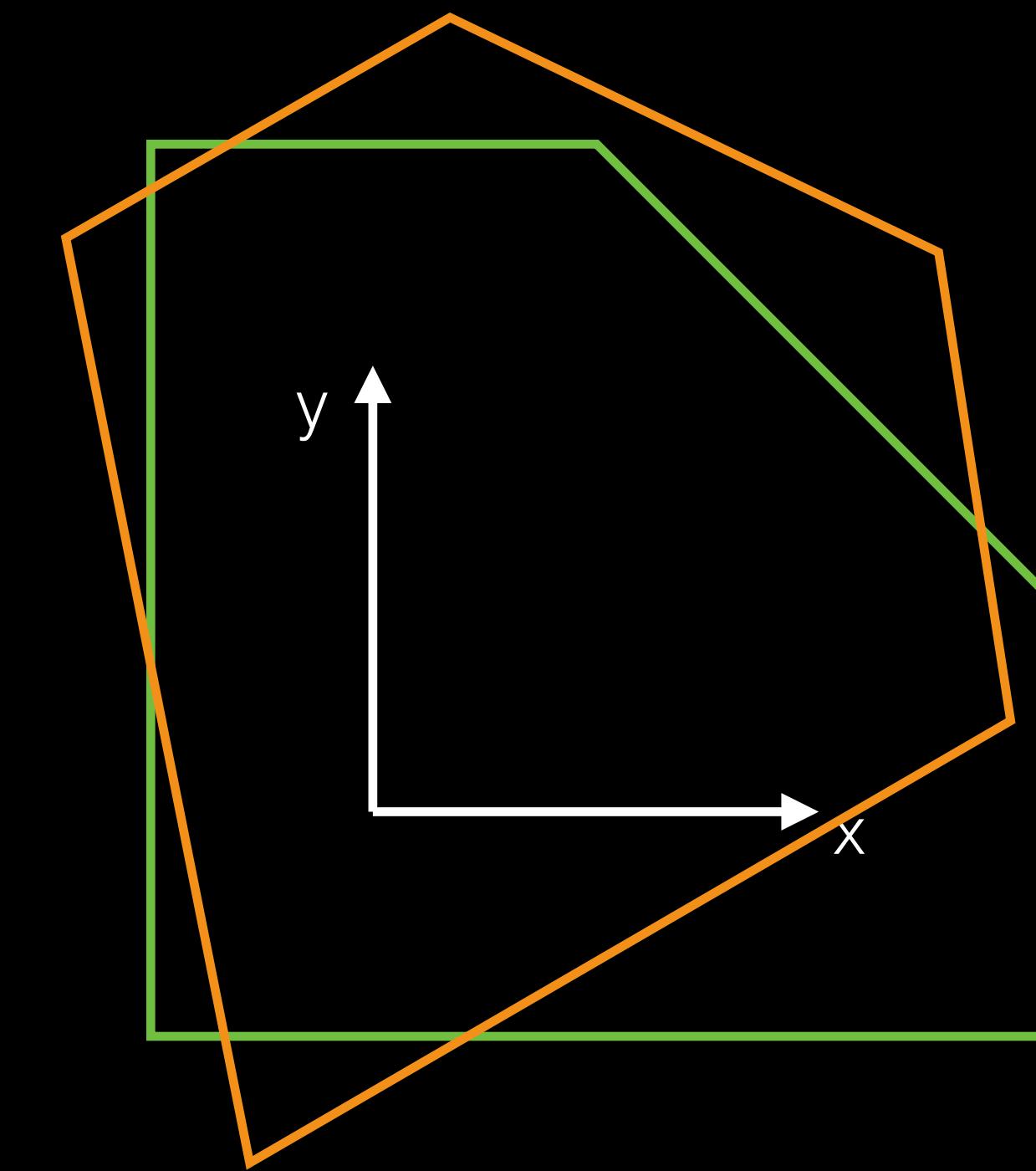
$$\begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$



# Combiner les transformations ...

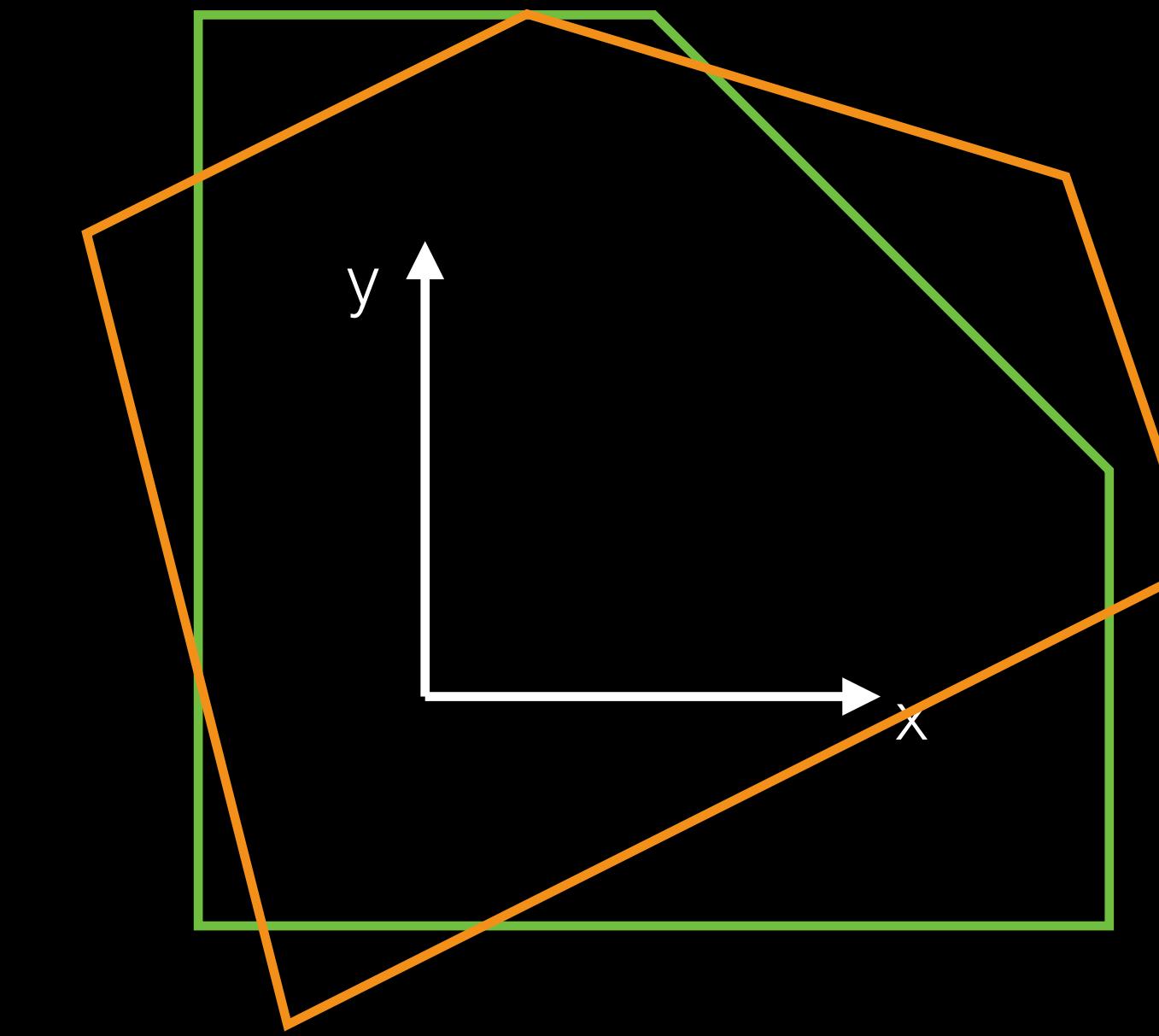
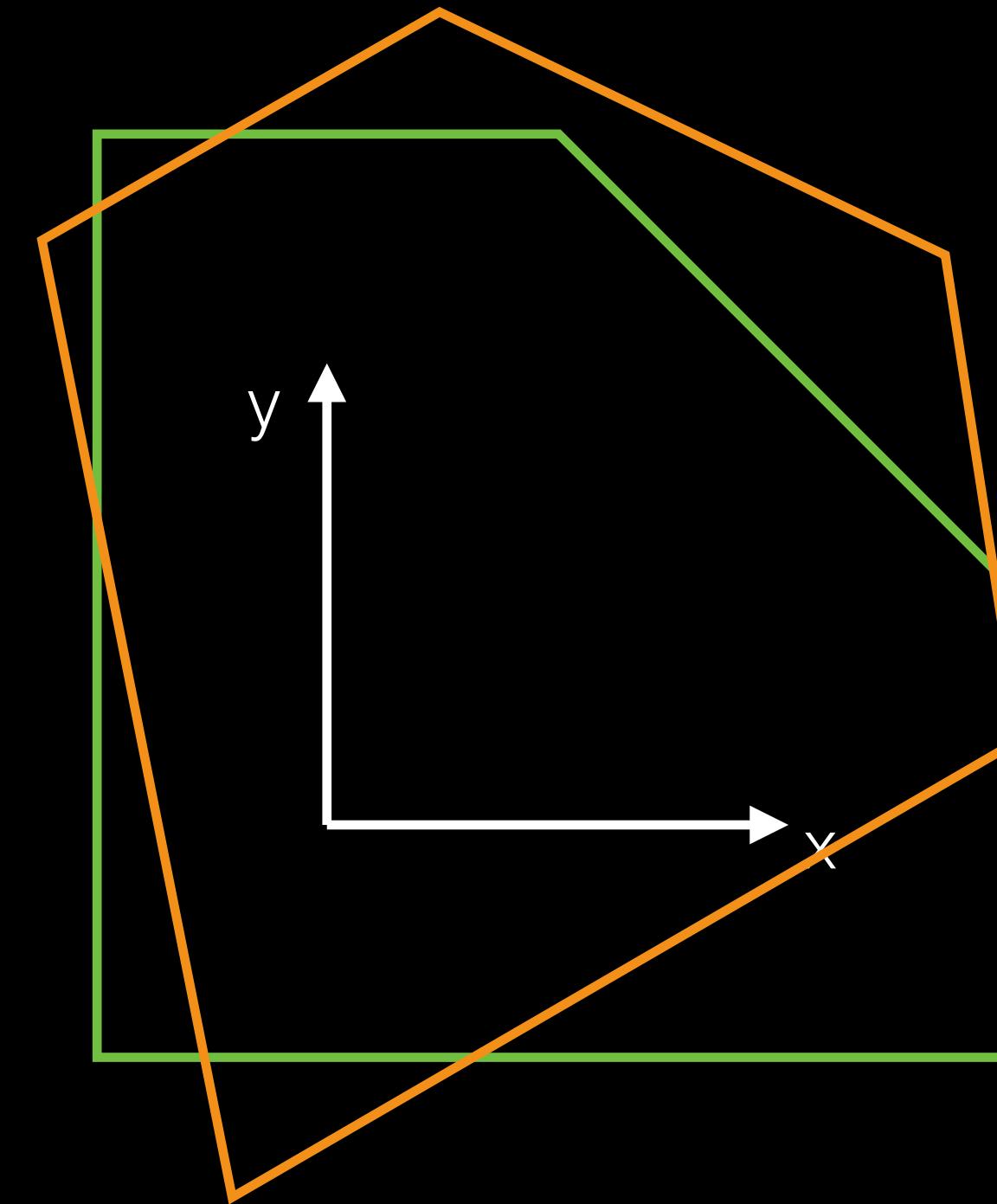
... en multipliant les matrices

$$\begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \begin{pmatrix} 1 & \frac{1}{3} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$



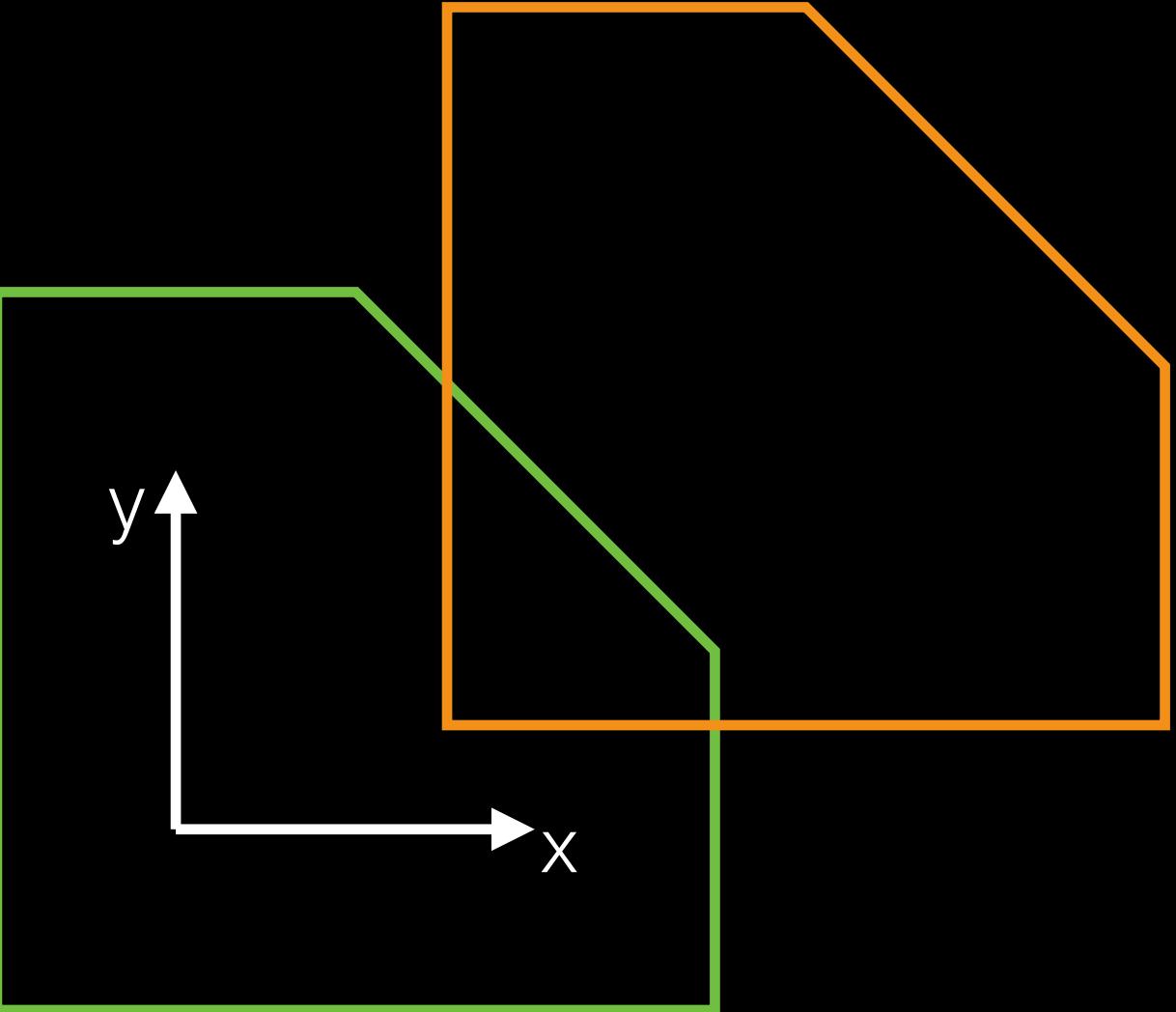
# Attention... pas de commutativité

$$\begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \begin{pmatrix} 1 & \frac{1}{3} \\ 0 & 1 \end{pmatrix} \neq \begin{pmatrix} 1 & \frac{1}{3} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}$$



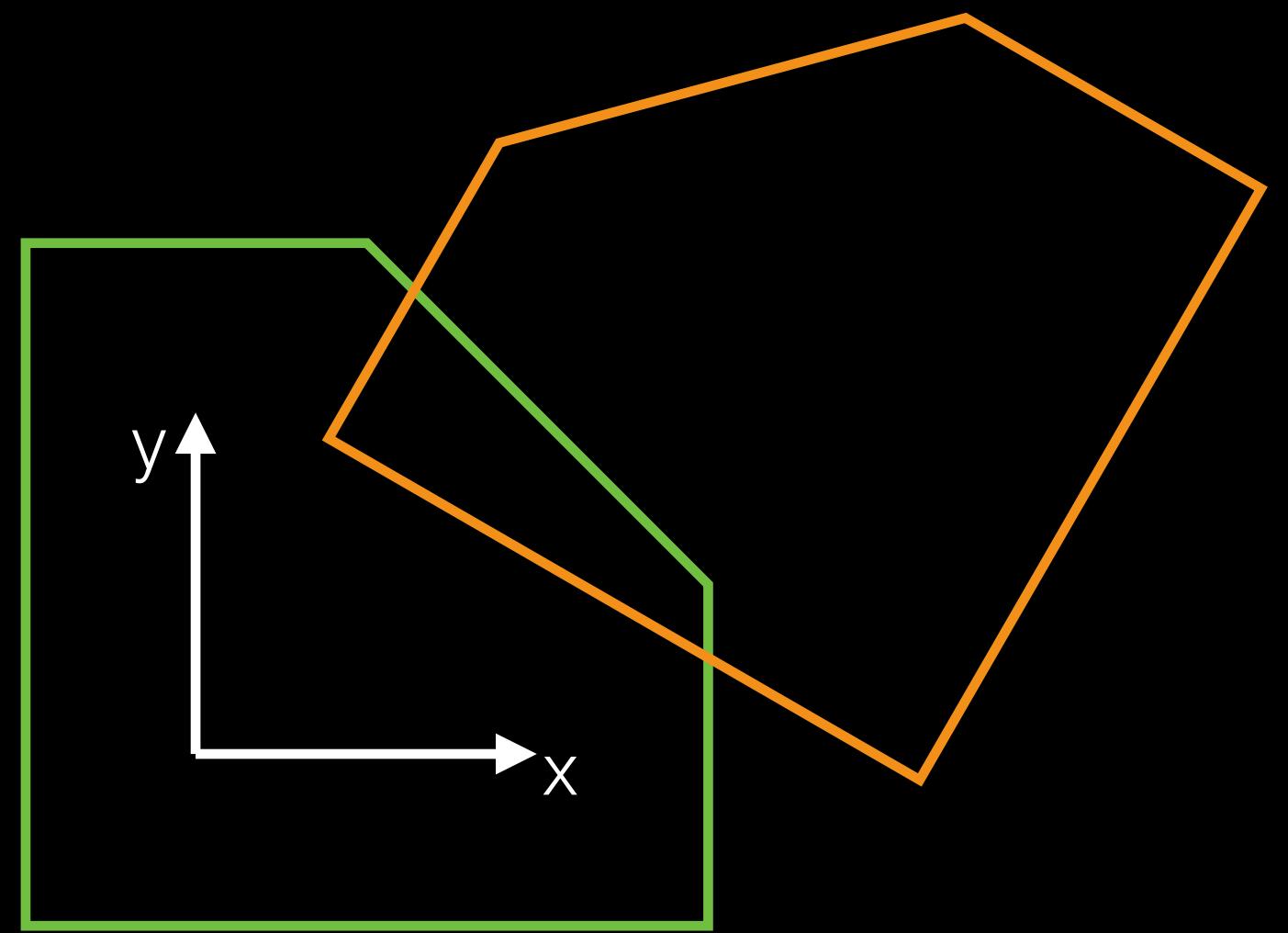
# Et les translations ?

$$\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} x + a \\ y + b \end{pmatrix}$$



# Et les transformations affines ?

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix} = \begin{pmatrix} ax + by + e \\ cx + dy + f \end{pmatrix}$$



# Coordonnées homogènes

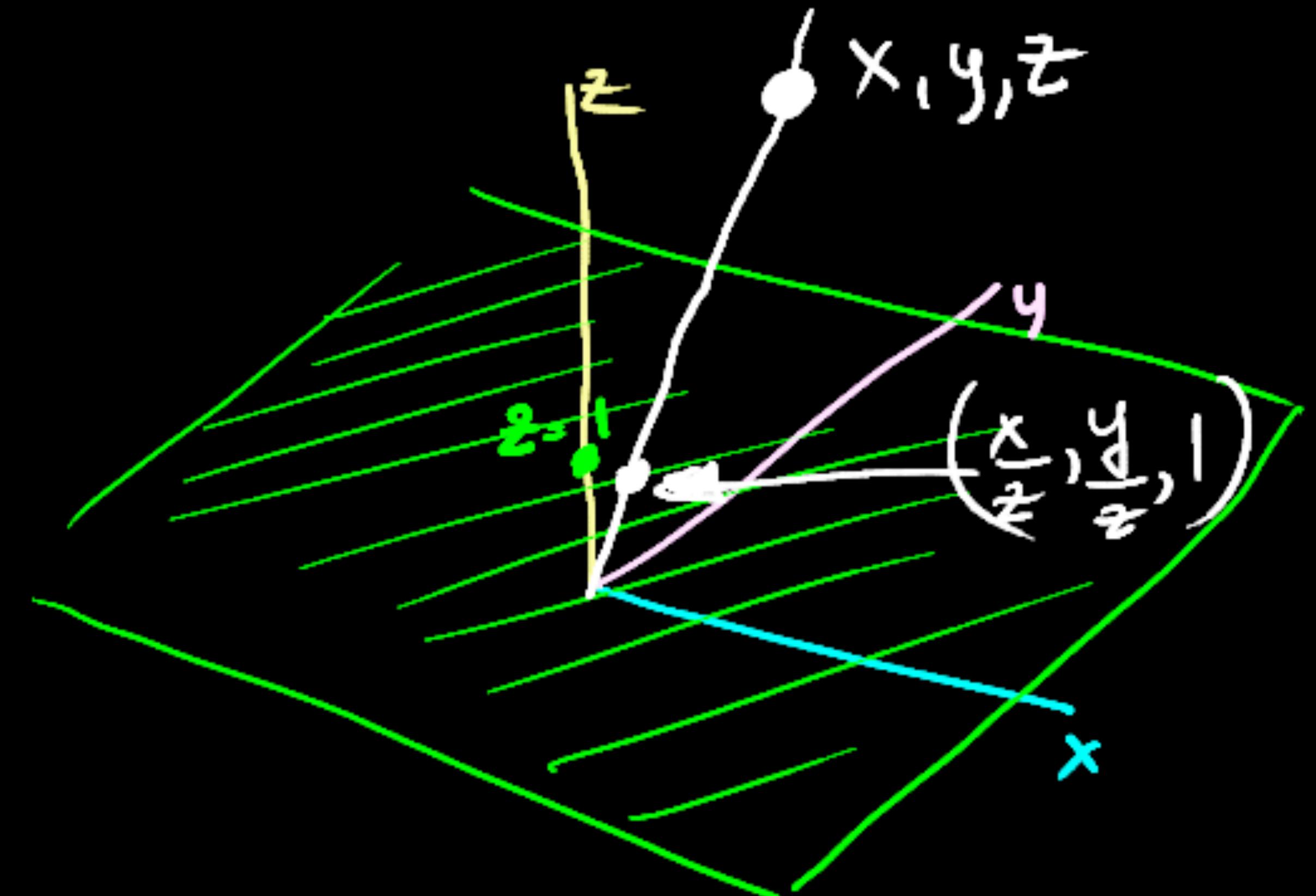
$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix} = \begin{pmatrix} ax + by + e \\ cx + dy + f \end{pmatrix}$$

$$\begin{pmatrix} a & b & e \\ c & d & f \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} ax + by + e \\ cx + dy + f \\ 1 \end{pmatrix}$$

# Interprétation

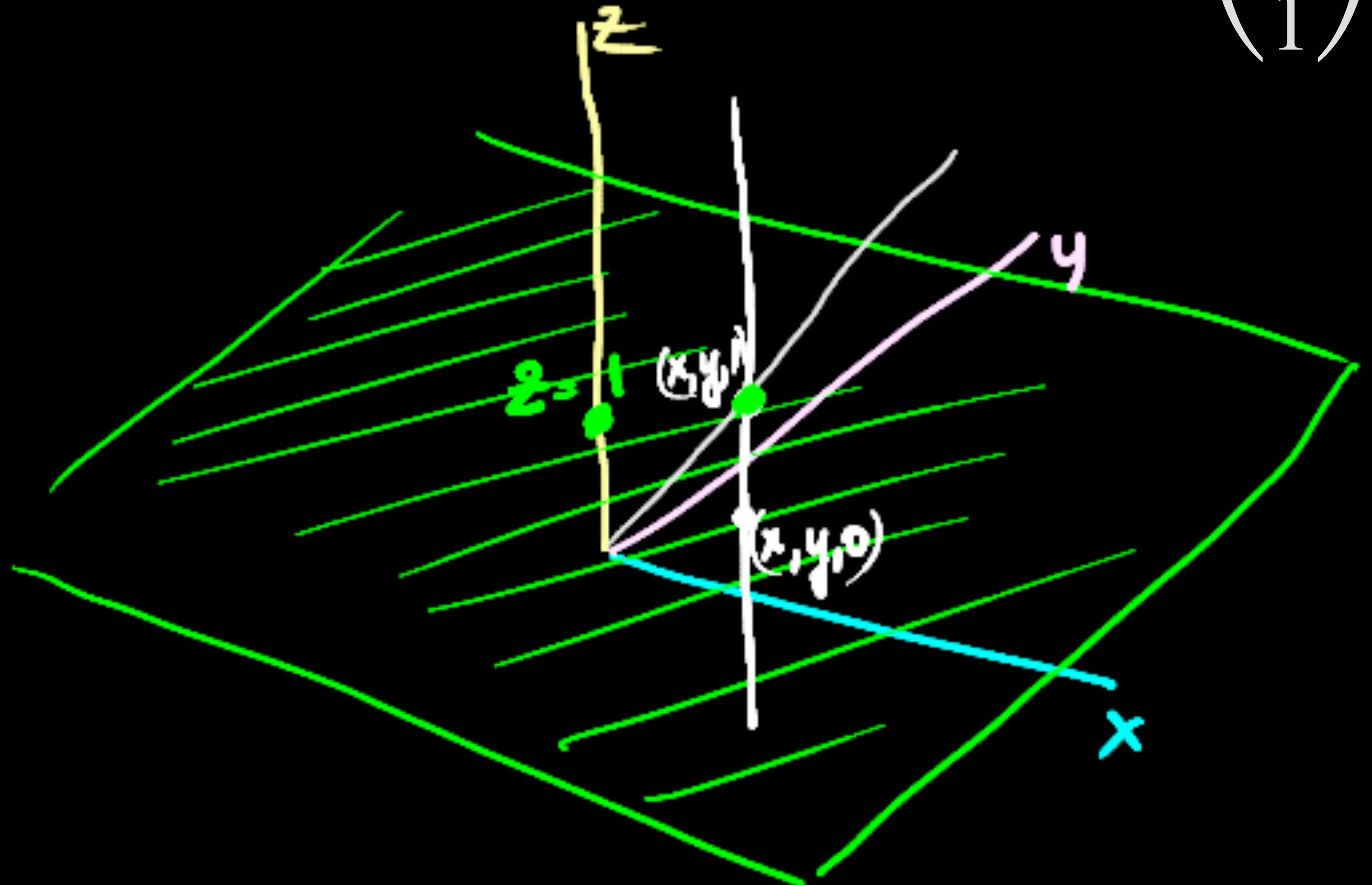
- ♦ L'espace 2d est inséré dans un espace 3d à l'emplacement du plan  $z=1$
- ♦ Les transformations affines en 2d deviennent des transformations linéaires en 3d. Ces transformations préservent  $z=1$
- ♦ On peut imaginer d'autres transformations linéaires qui modifient  $z$ . On revient alors dans l'espace 2d via

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \rightarrow \begin{pmatrix} x/z \\ y/z \end{pmatrix}$$



# Et si $z$ tend vers zéro ?

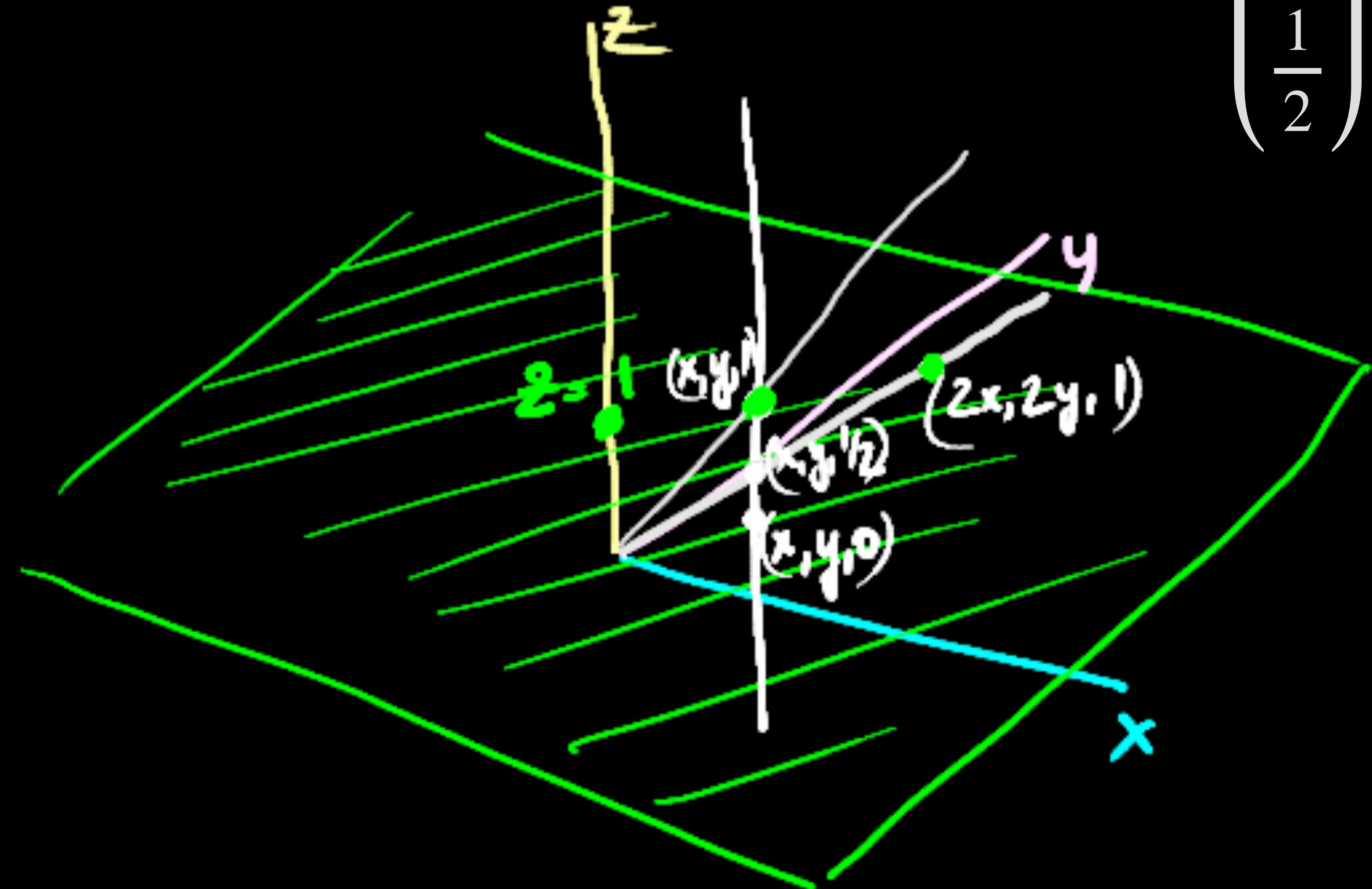
$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \rightarrow \begin{pmatrix} x/z \\ y/z \end{pmatrix}$$



# Et si z tend vers zéro ?

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \rightarrow \begin{pmatrix} x/z \\ y/z \end{pmatrix}$$

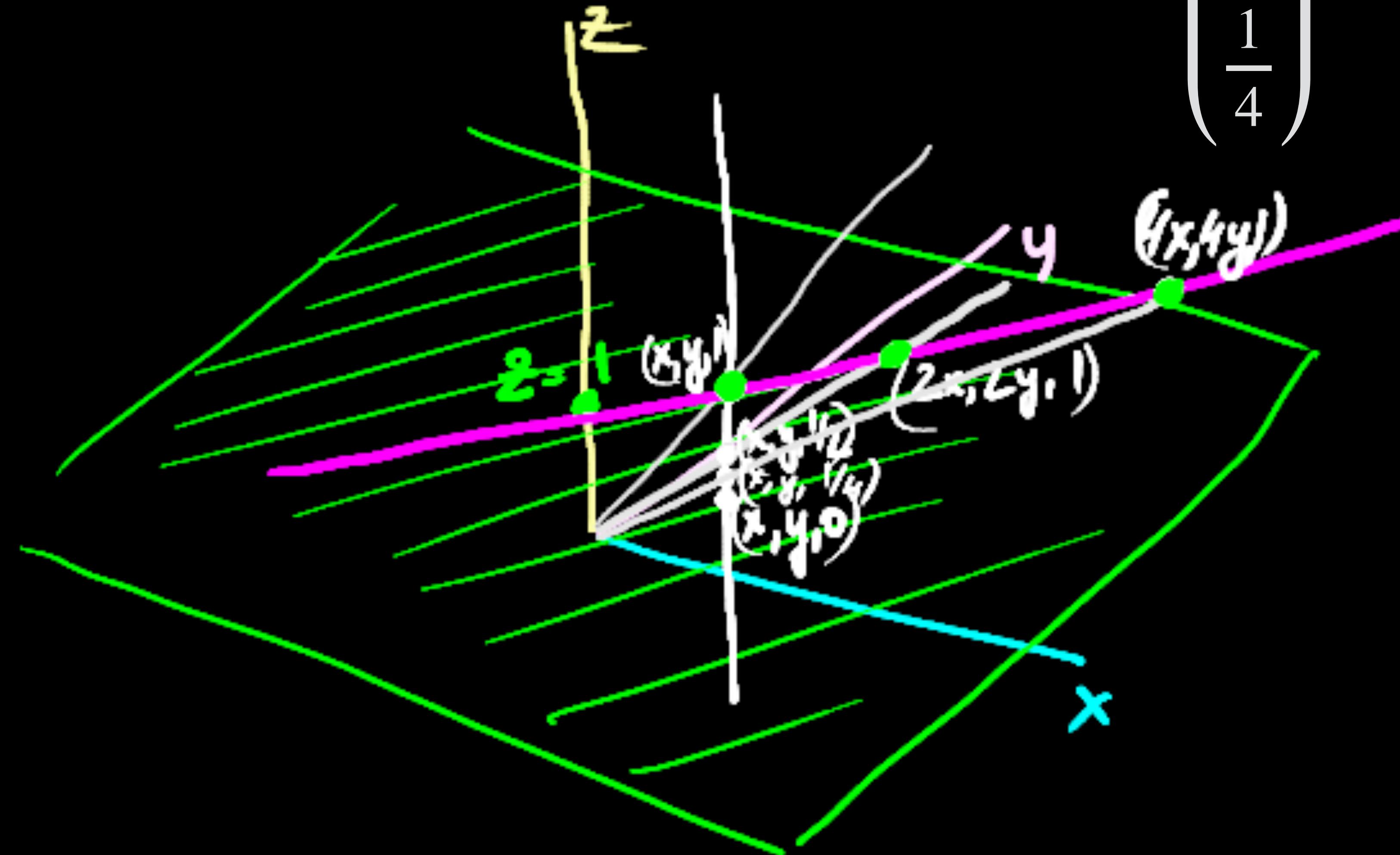
$$\begin{pmatrix} x \\ y \\ \frac{1}{z} \end{pmatrix}$$



# Et si $z$ tend vers zéro ?

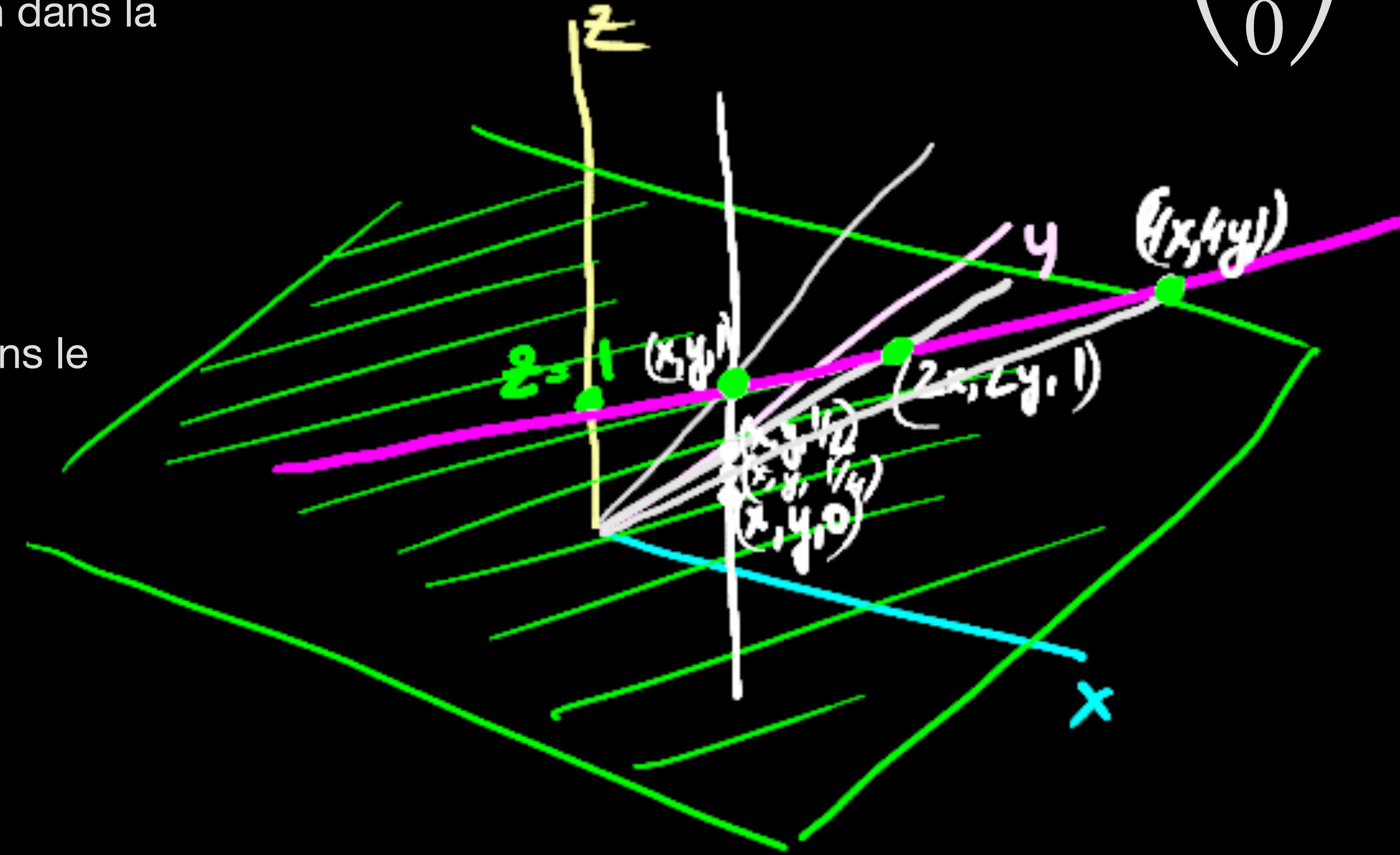
$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \rightarrow \begin{pmatrix} x/z \\ y/z \end{pmatrix}$$

$$\begin{pmatrix} x \\ y \\ \frac{1}{z} \end{pmatrix}$$



# Et si z tend vers zéro ?

- ◆  $(x, y, 0)$  est un point infiniment loin dans la direction  $(x, y)$ .
- ◆ C'est un **vecteur**
- ◆ En coordonnées homogènes, on représente points et vecteurs dans le même formalisme
  - ◆ Points si  $z \neq 0$
  - ◆ Vecteurs si  $z = 0$
  - ◆ Vecteur + vecteur = vecteur
  - ◆ Vecteur - vecteur = vecteur
  - ◆ Vecteur + point = point



# Composition des transformations affines

- ♦ En coordonnées homogènes, ces transformations deviennent linéaires.
- ♦ Elles sont donc comparables par produit matriciel
- ♦ Exemple: rotation autour d'un point  $(x_0, y_0)$

$$M = \begin{pmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & 1 \end{pmatrix}$$

# Et en 3D ?

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} a_{00}x + a_{01}y + a_{02}z + b_0 \\ a_{10}x + a_{11}y + a_{12}z + b_1 \\ a_{20}x + a_{21}y + a_{22}z + b_2 \end{pmatrix}$$

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & b_0 \\ a_{10} & a_{11} & a_{12} & b_1 \\ a_{20} & a_{21} & a_{22} & b_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} a_{00}x + a_{01}y + a_{02}z + b_0 \\ a_{10}x + a_{11}y + a_{12}z + b_1 \\ a_{20}x + a_{21}y + a_{22}z + b_2 \\ 1 \end{pmatrix}$$

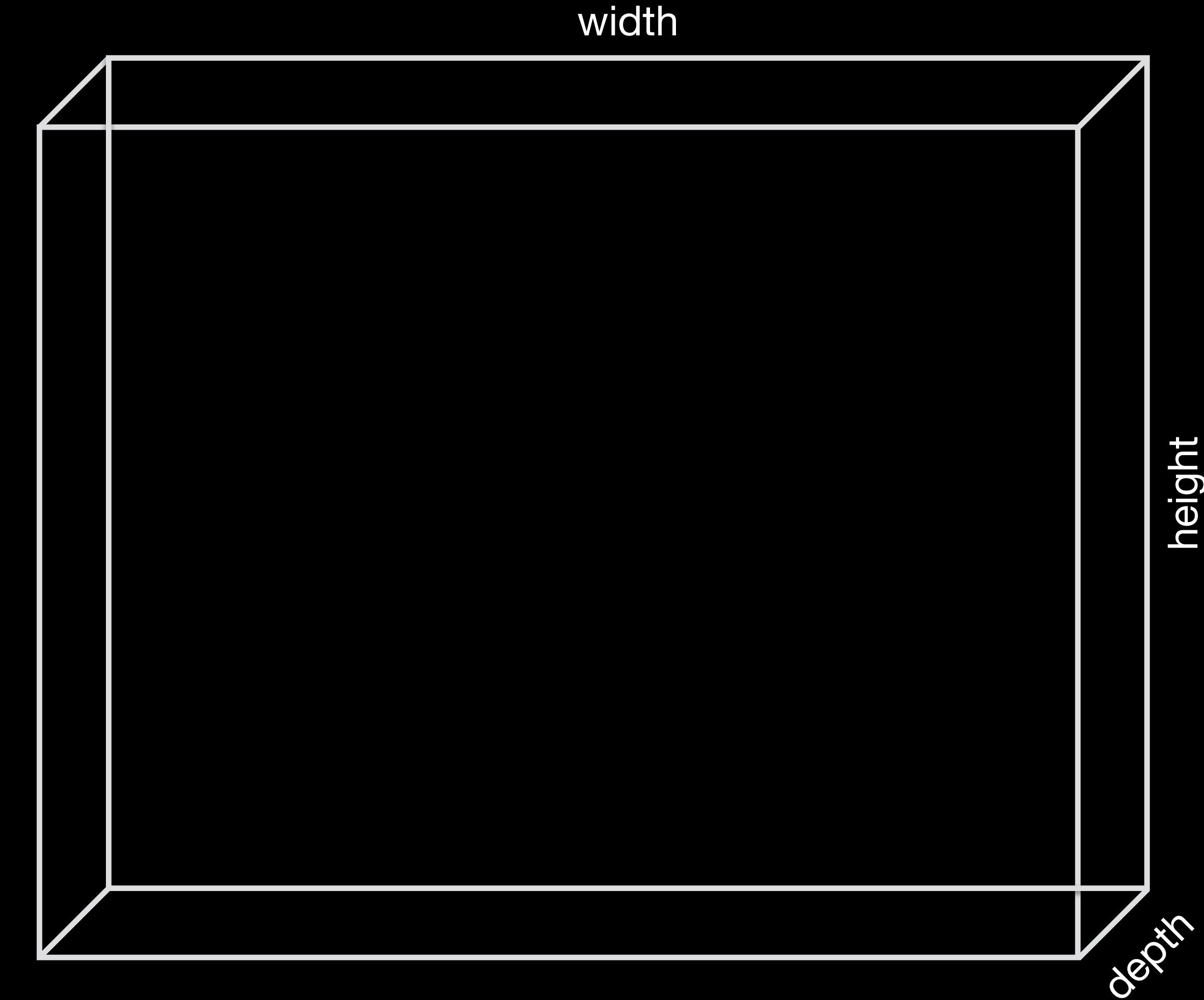
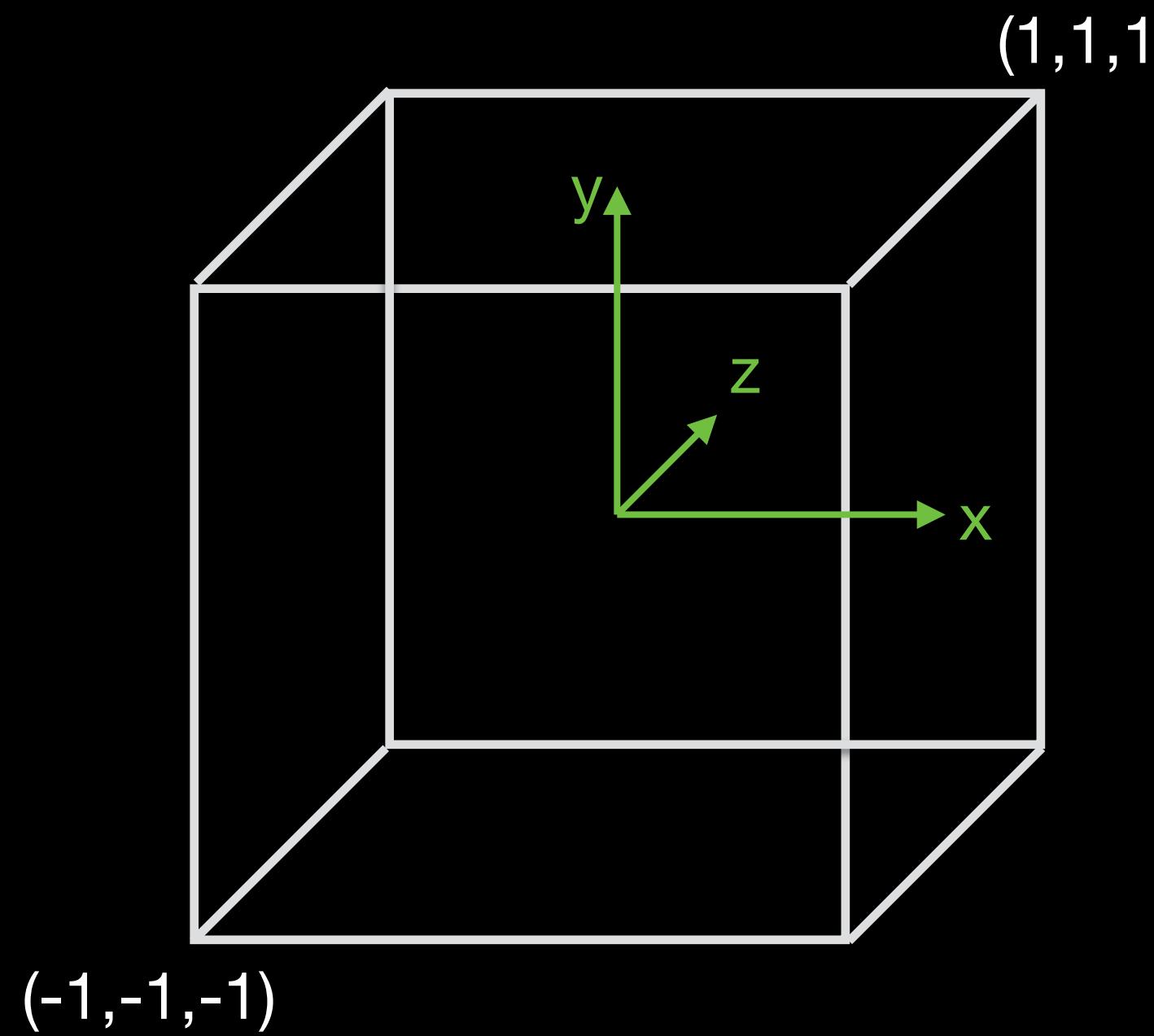
# Interprétation

- ♦ L'espace 3d est inséré dans un espace 4d à l'emplacement de l'hyperplan  $w=1$
- ♦ Les transformations affines en 3d deviennent des transformations linéaires en 4d. Ces transformations préservent  $w=1$
- ♦ On peut imaginer d'autres transformations linéaires qui modifient  $w$ . On revient alors dans l'espace 3d via

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \rightarrow \begin{pmatrix} x/w \\ y/w \\ z/w \end{pmatrix}$$

Désolé, je ne sais pas dessiner en 4D ...

# 1<sup>ère</sup> application - le viewport



# Code fourni

- ◆ Un nouveau fichier `geometry.h` fournissant une classe de calcul matriciel
- ◆ N'importe quel code effectuant le rendu parmi les précédents

## Votre mission

- ◆ Remplacer la ligne `screen[j] = ...` par une ligne ressemblant à

```
screen[j] = xyz(viewport * xyzw(world[j]));
```

- ◆ Avec `viewport` de type `Matrix4x4`

```
template<size_t rows, size_t cols>
class Matrix {
    std::array<std::array<float,cols>,rows> m;
public:
    Matrix(float val = 0.f);

    std::array<float,cols>& operator[](size_t i);
    const std::array<float,cols>& operator[](size_t i) const;

    template<size_t N>
    Matrix<rows,N> operator*(const Matrix<cols,N>& a);

    Matrix<cols,rows> transpose();

    Matrix inverse();

    static Matrix identity();
};

using Matrix4x4 = Matrix<4,4>;
using Vech = Matrix<4,1>;
```

# Viewport

```
Matrix4x4 viewport = make_viewport(0, 0, 800, 800);
```

```
Matrix4x4 make_viewport(int x, int y, size_t w, size_t h) {
    Matrix4x4 t = make_translate(Vec3f{x+w/2.f, y+h/2.f, 100});
    Matrix4x4 s = make_scale(Vec3f{w/2.f, h/2.f, 100});
    return t*s;
}
```

```
Matrix4x4 make_translate(Vec3f v) {
    Matrix4x4 m = Matrix4x4::identity();
    for(int i=0; i < 3; ++i)
        m[i][3] = v.raw[i];
    return m;
}
```

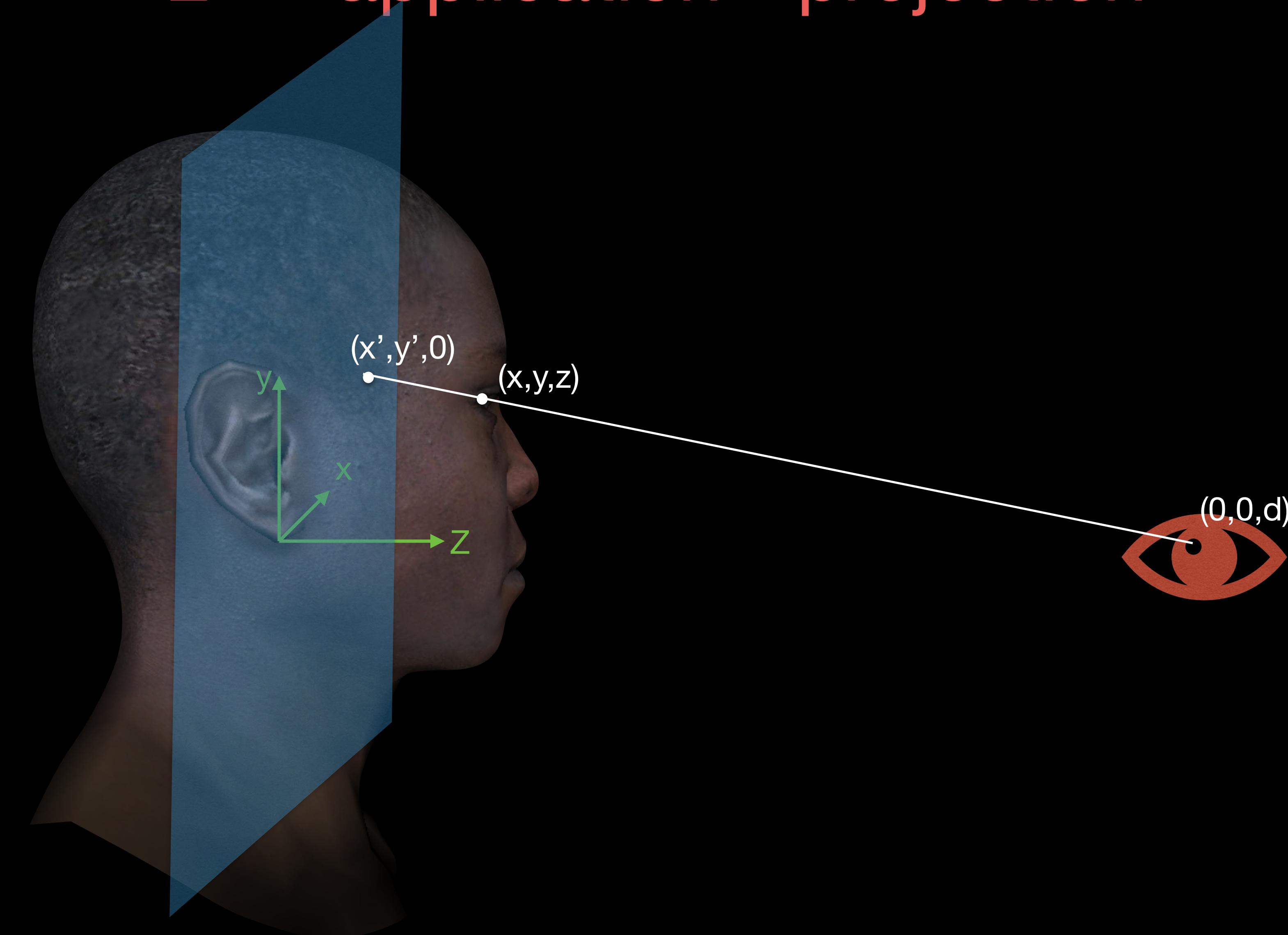
```
Matrix4x4 make_scale(Vec3f v) {
    Matrix4x4 m = Matrix4x4::identity();
    for(int i=0; i < 3; ++i)
        m[i][i] = v.raw[i];
    return m;
}
```

```
VecH xyzw(Vec3f v) {
    VecH m(1.f);
    for(size_t i = 0; i < 3; ++i)
        m[i][0] = v.raw[i];
    return m;
}
```

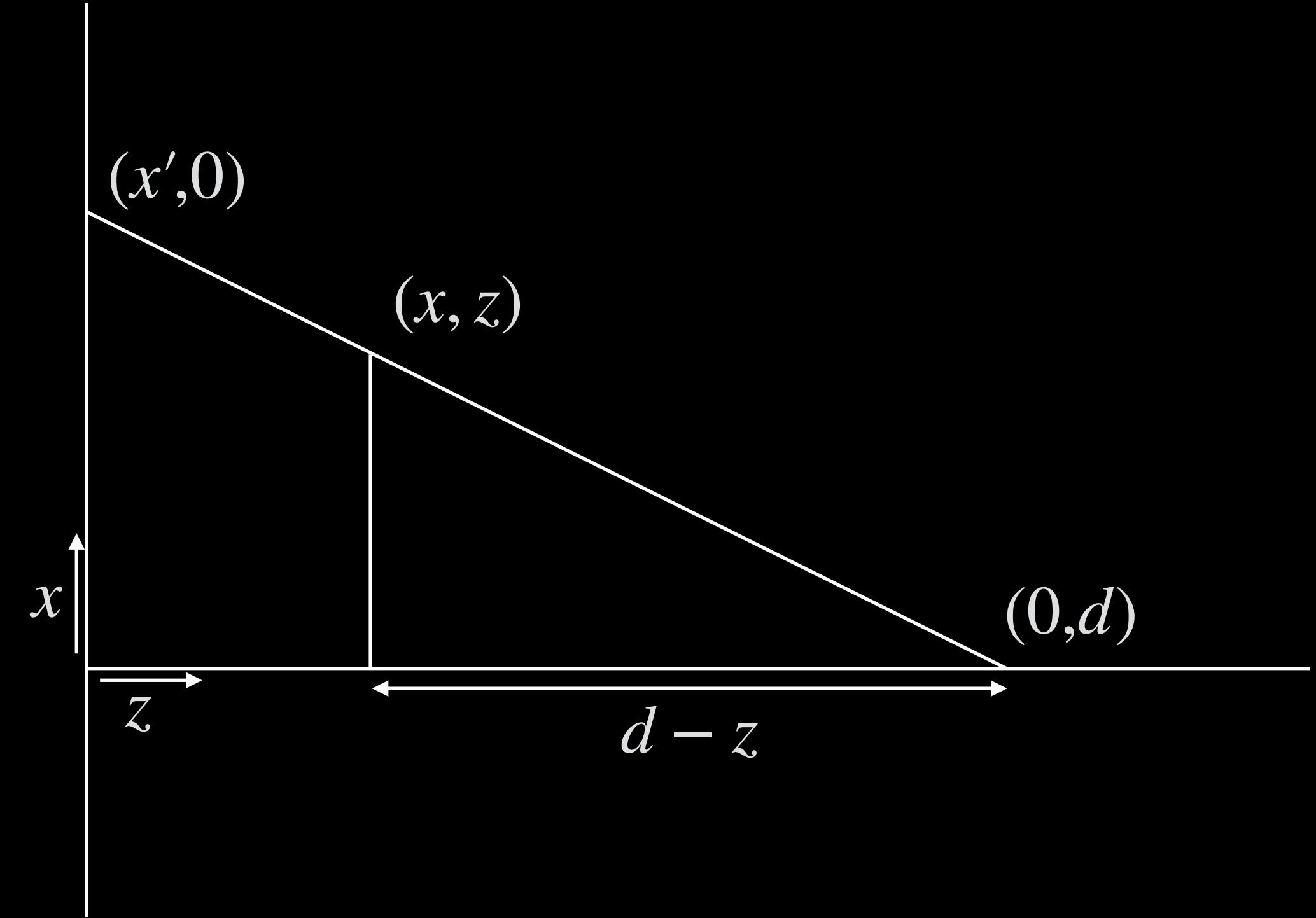
```
Vec3f xyz(VecH m) {
    Vec3f v;
    for(size_t i = 0; i < 3; ++i)
        v.raw[i] = m[i][0]/m[3][0];
    return v;
}
```

# 2ème application - projection

heig-vd



# Pour projeter de 2d en 1d...



$$\frac{x'}{d} = \frac{x}{d - z}$$

$$x' = \frac{x}{1 - \frac{1}{d} \cdot z}$$

# Pour projeter de 2d en 1d...

$$x' = \frac{x}{1 - \frac{1}{d} \cdot z}$$

$$x' = x \\ w' = 1 - \frac{1}{d} \cdot z$$

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -\frac{1}{d} & 1 \end{pmatrix} \begin{pmatrix} x \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ z \\ 1 - \frac{1}{d}z \end{pmatrix} = \begin{pmatrix} \frac{x}{1 - \frac{1}{d}z} \\ \frac{z}{1 - \frac{1}{d}z} \\ 1 \end{pmatrix}$$

# Pour projeter de 3d en 2d...

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -\frac{1}{d} & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 1 - \frac{1}{d}z \end{pmatrix} = \begin{pmatrix} \frac{x}{1 - \frac{1}{d}z} \\ \frac{y}{1 - \frac{1}{d}z} \\ \frac{z}{1 - \frac{1}{d}z} \\ 1 \end{pmatrix}$$

# Code fourni

- ♦ Rien de plus

## Votre mission

- ♦ Insérer la matrice de projection dans le calcul des coordonnées écran
- ♦ Générer l'image ci-contre en plaçant la caméra en  $z = 2$
- ♦ Question: qu'obtient-on si  
 $z = \text{numeric\_limits<} \text{float} \text{>} ::\text{infinity}();$



# Projection

```
Vec3f camera{0.f,0.f,2.f};  
  
Matrix4x4 projection = Matrix4x4::identity();  
projection[3][2] = -1.f / camera.z;  
  
...  
  
screen[j] = xyz(viewport * projection * xyzw(world[j]));
```

# 3ème application - bouger la caméra

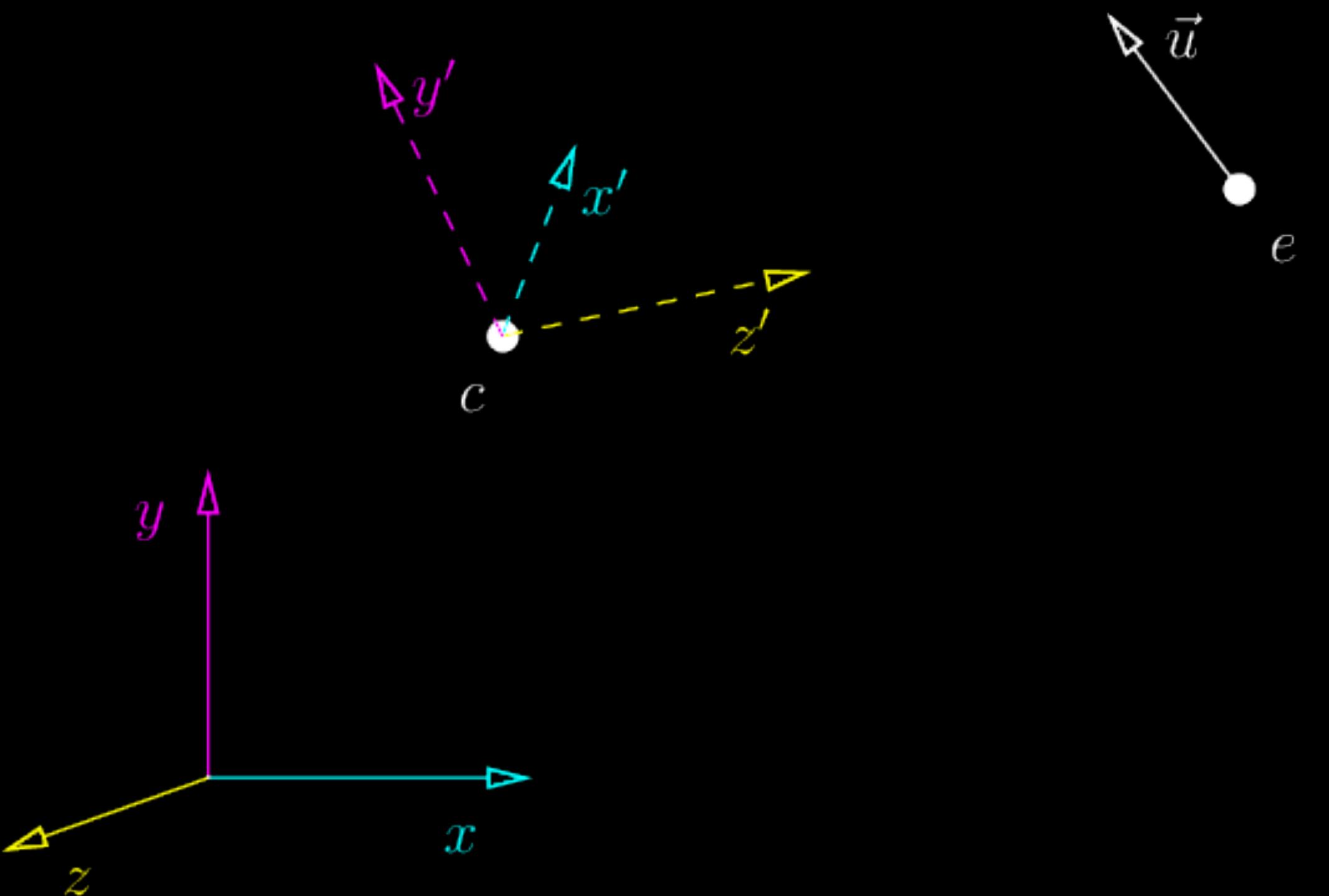
- ♦ Jusqu'à présent, nous avons toujours considéré la caméra dans la direction z
- ♦ Nous allons appliquer une transformation pour exprimer le monde dans les coordonnées de la camera
- ♦ La transformation correspondante s'appelle view et vient s'appliquer avant la projection

```
screen[j] = viewport * projection * view * world[j];
```



# View

- ◆ L'oeil de la caméra est au point  $e$  (eye)
- ◆ Elle regarde vers le centre de l'image au point  $c$  (centre)
- ◆ L'axe  $e-c$  définit la direction de  $z'$
- ◆ L'image est dans le plan  $(x', y')$  perpendiculaire à  $z'$
- ◆ La direction dans ce plan des vecteurs  $x'$  et  $y'$  est déterminée par le vecteur  $u$  (up) qui indique le haut de l'image.  $x'$  lui est perpendiculaire
- ◆ Pour passer des coordonnées monde aux coordonnées camera, il faut effectuer un changement de base



# Changement de base

- ♦ Les coordonnées  $(x, y, z)$  d'un point  $P$  dans une base définie par un point  $O$  et 3 vecteurs  $i, j$  et  $k$  sont telle que

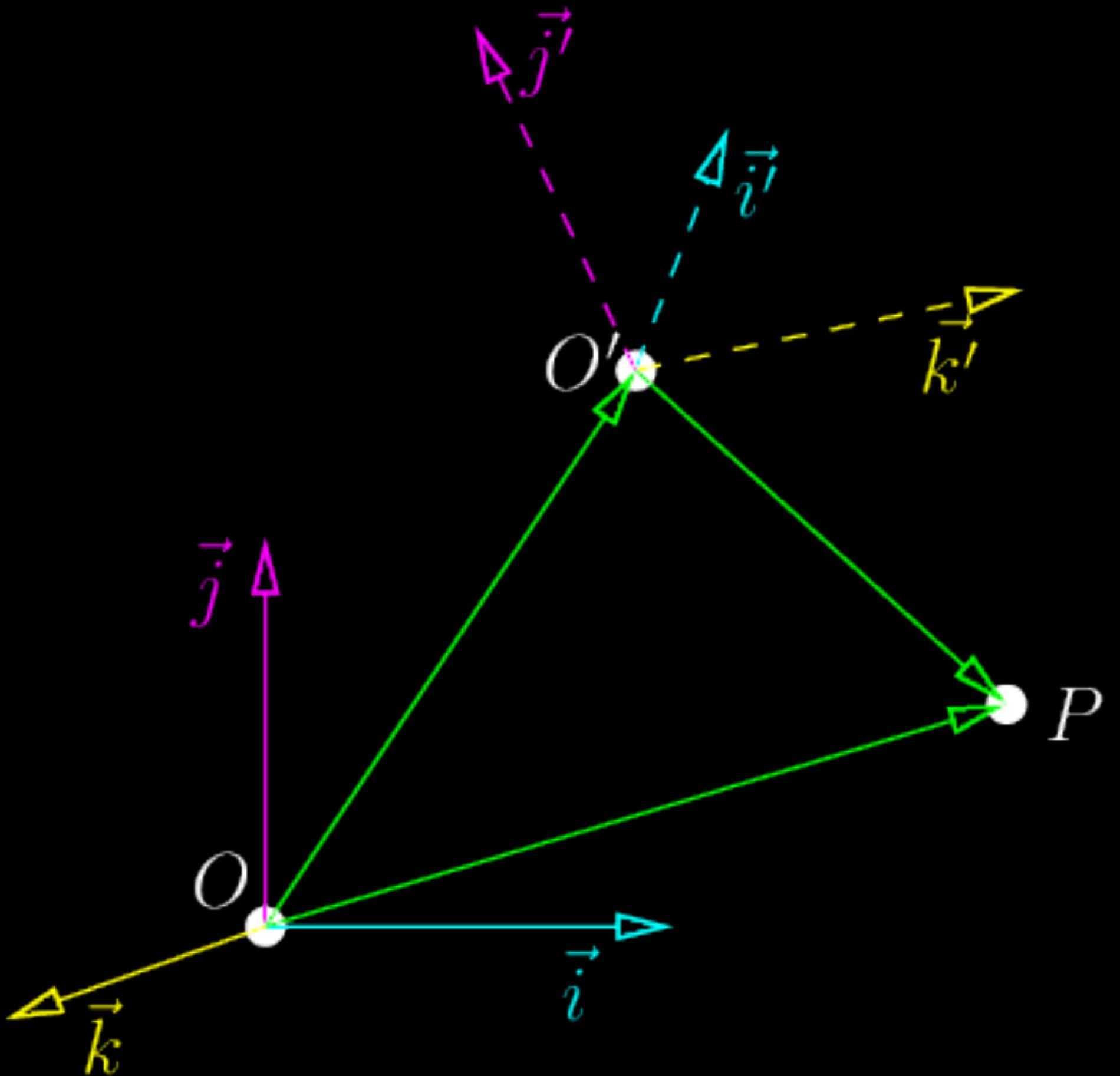
$$\overrightarrow{OP} = \vec{i}x + \vec{j}y + \vec{k}z = (\vec{i} \ \vec{j} \ \vec{k}) \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

- ♦ On veut pouvoir exprimer  $P$  dans la base  $O', i', j', k'$

$$\overrightarrow{O'P} = \vec{i}'x' + \vec{j}'y' + \vec{k}'z' = (\vec{i}' \ \vec{j}' \ \vec{k}') \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix}$$

- ♦ Les coordonnées de  $i', j', k'$  dans la base  $i, j, k$  nous donnent les colonnes de la matrice  $M$  telle que

$$(\vec{i}' \ \vec{j}' \ \vec{k}') = (\vec{i} \ \vec{j} \ \vec{k}) \times M$$



# Changement de base (2)

- ◆ Cela nous permet d'écrire  $\overrightarrow{OP} = \overrightarrow{OO'} + \overrightarrow{O'P}$

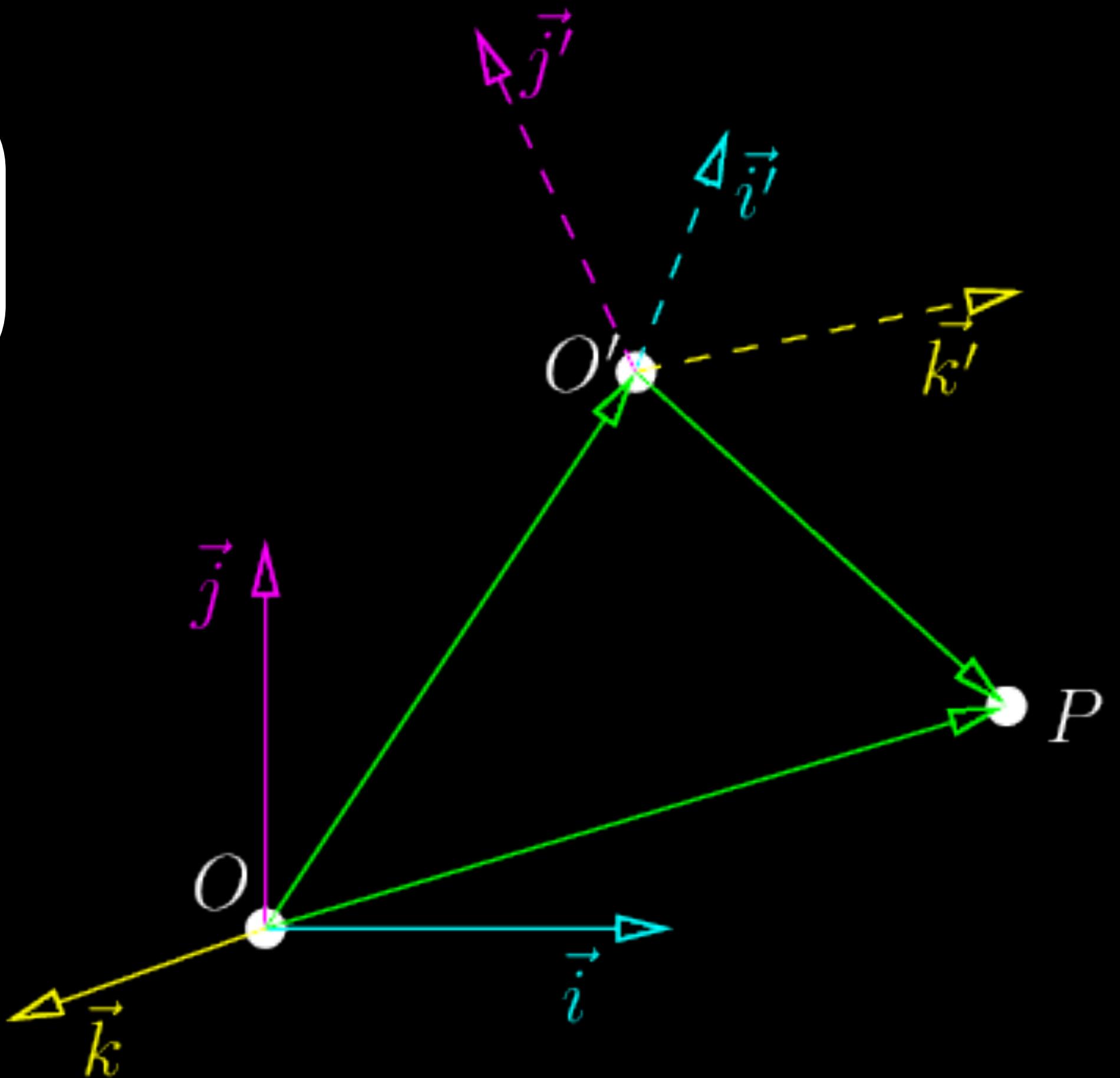
$$= (\vec{i} \vec{j} \vec{k}) \begin{pmatrix} O'_x \\ O'_y \\ O'_z \end{pmatrix} + (\vec{i}' \vec{j}' \vec{k}') \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix}$$

$$= (\vec{i} \vec{j} \vec{k}) \left( \begin{pmatrix} O'_x \\ O'_y \\ O'_z \end{pmatrix} + M \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} \right)$$

- ◆ Et donc pour les coordonnées

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} O'_x \\ O'_y \\ O'_z \end{pmatrix} + M \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix}$$

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = M^{-1} \left( \begin{pmatrix} x \\ y \\ z \end{pmatrix} - \begin{pmatrix} O'_x \\ O'_y \\ O'_z \end{pmatrix} \right)$$



# Code fourni

- ♦ Rien de plus

## Votre mission

- ♦ Ecrire une fonction `lookat(eye, center, up)` qui prend en entrée trois `Vec3f` et produit la `Matrix4x4` view correspondante
- ♦ Générer l'image ci-contre avec l'oeil en  $(1,1,2)$ , le centre en  $(0,0,0)$  et la direction up selon l'axe y.



# View

```
Vec3f      eye(1,1,2);
Vec3f      center(0,0,0);
Vec3f      up(0,1,0);

Matrix4x4 view = lookat(eye,center,up);
```

```
Matrix4x4 lookat(Vec3f eye, Vec3f center, Vec3f up)
{
    Matrix4x4 view = Matrix4x4::identity();

    Vec3f z = (eye-center).normalize();
    Vec3f x = (up^z).normalize();
    Vec3f y = (z^x).normalize();

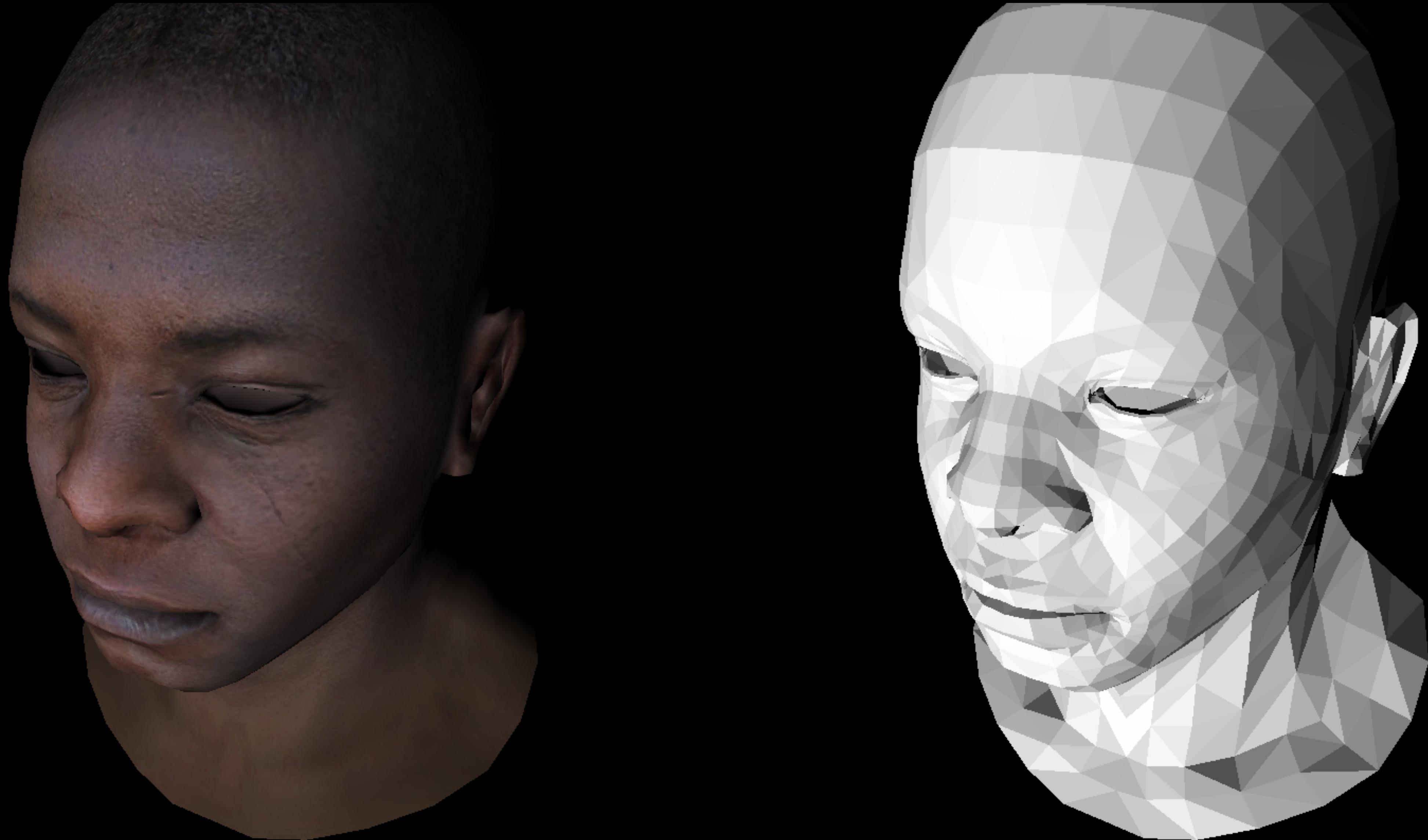
    for (int i=0; i<3; i++) {
        view[0][i] = x.raw[i];
        view[1][i] = y.raw[i];
        view[2][i] = z.raw[i];
        view[i][3] = -center.raw[i];
    }

    return view;
}
```

# Refactor

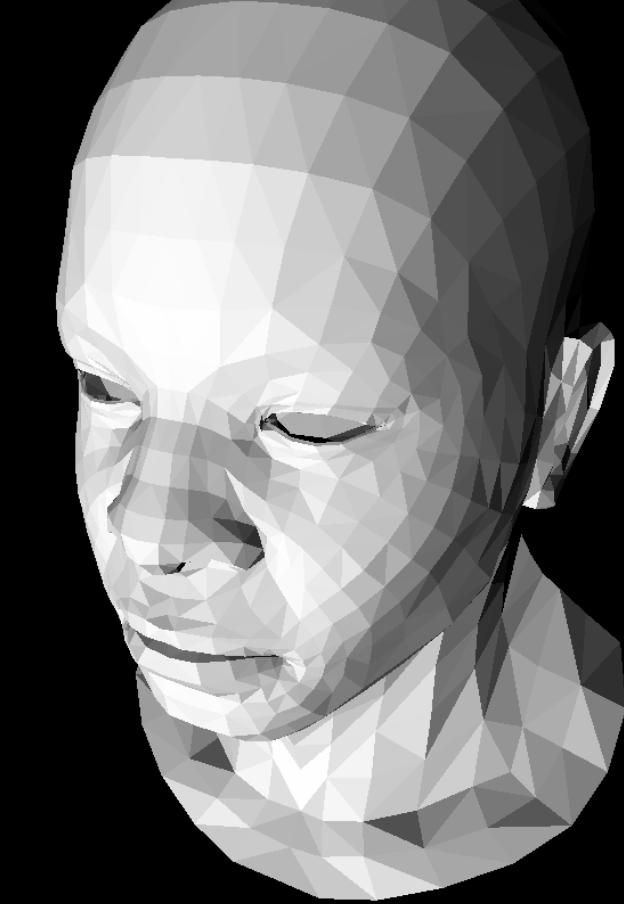
Débarrassons-nous des duplications ...

# En quoi les codes générant ces images diffèrent-ils ?



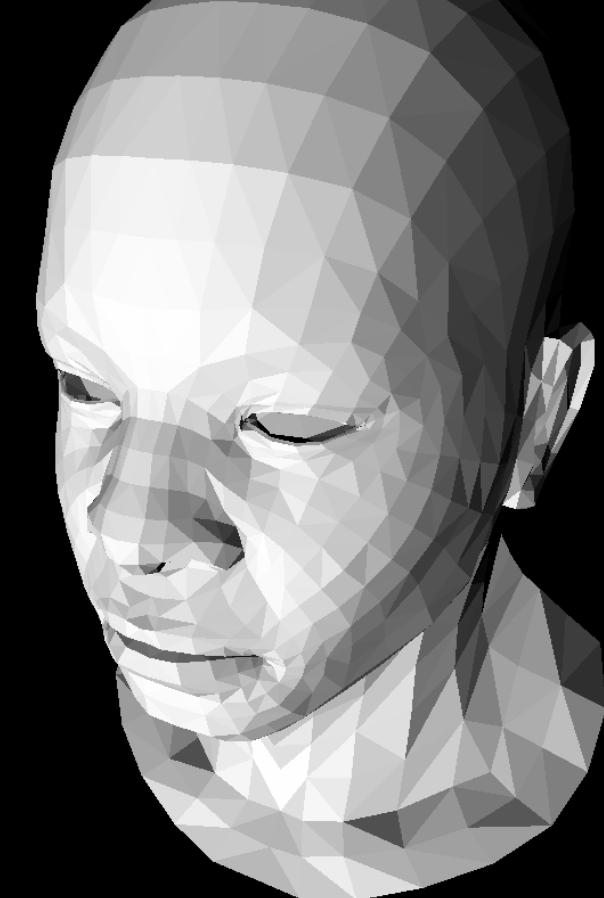


# main()



```
for (int i = 0; i < model.nfaces(); i++) {    for (int i = 0; i < model.nfaces(); i++) {  
    Vec3f screen[3];    Vec3f screen[3];  
    Vec3f world[3];    Vec3f world[3];  
    Vec3f text[3];  
    Vec3f intensity;  
  
    for (int j = 0; j < 3; j++) {  
        world[j] = model.vert(model.face(i)[j]);  
        screen[j] = xyz(viewport * projection *  
        text[j] = model.texture(model.face_tex);  
        Vec3f n = model.normal(model.face_norm);  
        intensity.raw[j] = n * light;  
    }  
    triangle(screen, text, zbuffer, image, tex  
}  
triangle(screen, zbuffer, image, texture, int
```

# triangle()



```

void triangle(Vec3f t[3], Vec3f tc[3], float* zbuffer, T
{
    auto bbox = boite_englante(image, t);
    for(int y = bbox[0].y; y <= bbox[1].y; ++y) {
        for(int x = bbox[0].x; x <= bbox[1].x; ++x) {
            Vec3f p { float(x), float(y), 0 };
            Vec3f b = barycentriques(t,p);
            if ( b.x >= 0 and b.y >= 0 and b.z >= 0 ) {
                p.z = b.x*t[0].z + b.y*t[1].z + b.z*t[2].z;
                if(zbuffer[x + image.get_width() * y] <
                   zbuffer[x + image.get_width() * y + 1])
                    zbuffer[x + image.get_width() * y + 1] = p.z;
                Vec3f ptc = tc[0]*b.x + tc[1]*b.y + tc[2]*b.z;
                Vec2i pc ( ptc.x * texture.get_width(),
                           ptc.y * texture.get_height());
                TGAColor color = texture.get(pc.x, pc.y);
                float gouraud = intensity * b.z;
                if(gouraud < 0) gouraud = 0;
                for(int i = 0; i < 3; ++i) color.raw[i] = gouraud;
                image.set(p.x, p.y, color);
            }
        }
    }
}

```

```
d triangle(Vec3f t[3], float* zbuffer, TGAImage &image,
auto bbox = boite_englante(image, t);
for(int y = bbox[0].y; y <= bbox[1].y; ++y) {
    for(int x = bbox[0].x; x <= bbox[1].x; ++x) {
        Vec3f p { float(x), float(y), 0};
        Vec3f b = barycentriques(t,p);
        if ( b.x >= 0 and b.y >= 0 and b.z >= 0 ) {
            p.z = b.x*t[0].z + b.y*t[1].z + b.z*t[2].z;
            if(zbuffer[x + image.get_width() * y] < p.z)
                zbuffer[x + image.get_width() * y] = p.z;
            TGAColor color = TGAColor(255,255,255,255);
            for(int i = 0; i < 3; ++i) color.raw[i] = image.get(p.x,p.y,color);
        }
    }
}
}
```

# Comportement commun

- ◆ main()
- ◆ Boucle sur toutes les faces du modèles
- ◆ Boucle sur tous les sommets de chaque face
- ◆ Pour chaque sommet, extrait des infos du modèle ou calcule des attributs
- ◆ triangle( )
- ◆ Boucle sur tous les pixels de la bounding box
- ◆ Vérifie si le pixel doit être colorié
- ◆ Interpole sa couleur à partir des attributs des sommets / de la face
- ◆ Les paramètres de triangle() transmettent ces infos

# Shaders

```

struct Shader {
    Vec3f light;
    Matrix4x4 view;
    Matrix4x4 projection;
    Matrix4x4 viewport;
    Model* model;

    virtual ~Shader() {};
    virtual Vec3f vertex(int iface, int nthvert);
    virtual bool fragment(Vec3f bary, TGAColor &color);
};

for (size_t f = 0; f < model.nfaces(); ++f) {
    Vec3f screen[3];
    for(size_t v = 0; v < 3; ++v) {
        screen[v] = shader.vertex(f,v);
    }
    triangle(screen, shader, zbuffer, image);
}

```

```

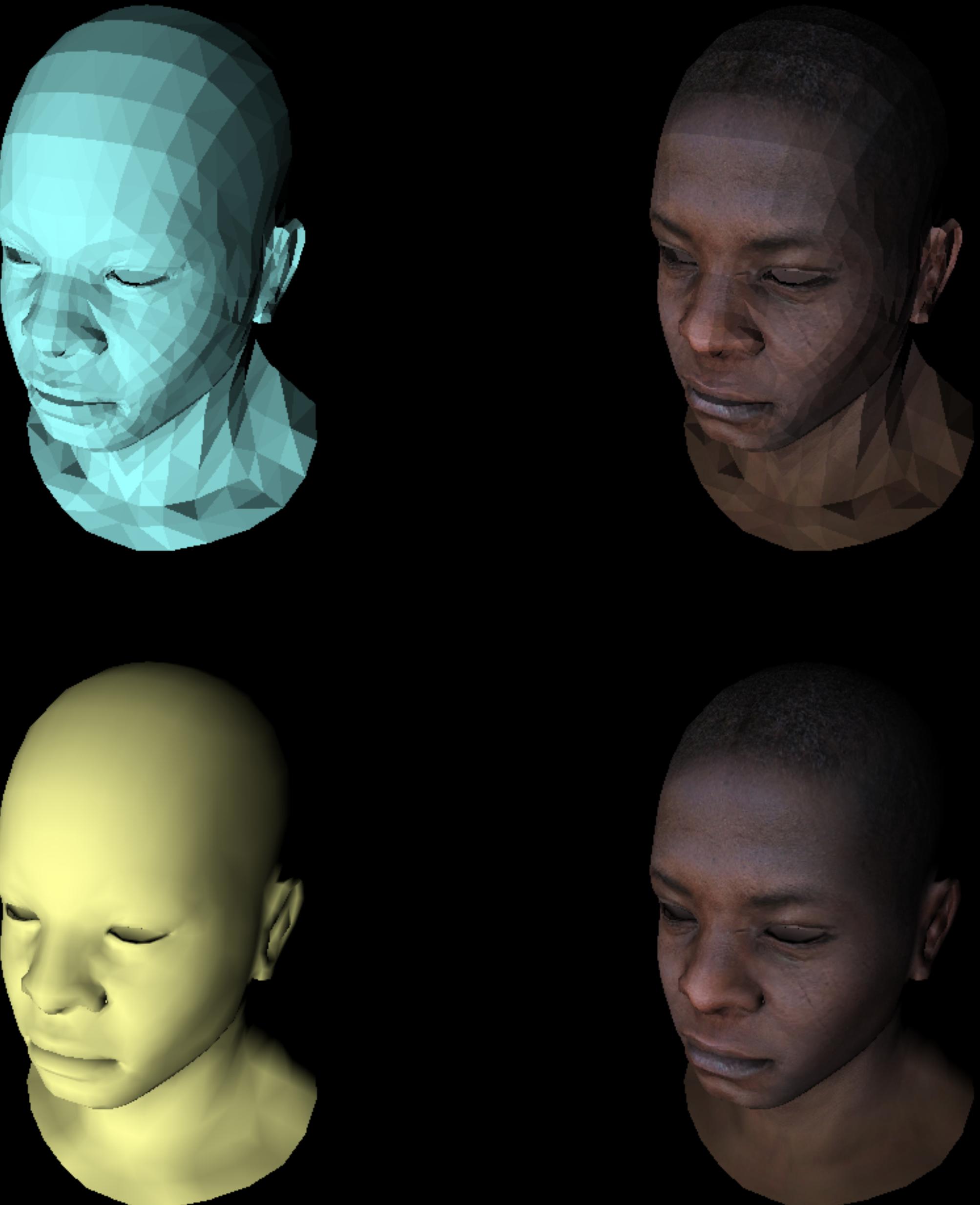
void triangle(Vec3f screen[3], Shader& shader, float
    auto bbox = boite_englante(image, screen);
    for(int y = bbox[0].y; y <= bbox[1].y; ++y) {
        for(int x = bbox[0].x; x <= bbox[1].x; ++x) {
            Vec3f p { float(x), float(y), 0};
            Vec3f b = barycentriques(screen,p);
            if ( b.x >= 0 and b.y >= 0 and b.z >= 0 ) {
                p.z = b.x*screen[0].z + b.y*screen[1].z +
                if ( zbuffer[x + image.get_width() * y] < |
                    TGAColor color;
                    if( shader.fragment(b, color) ) {
                        zbuffer[x + image.get_width() * y] = p.z;
                        image.set(p.x,p.y,color);
                    }
                }
            }
        }
    }
}

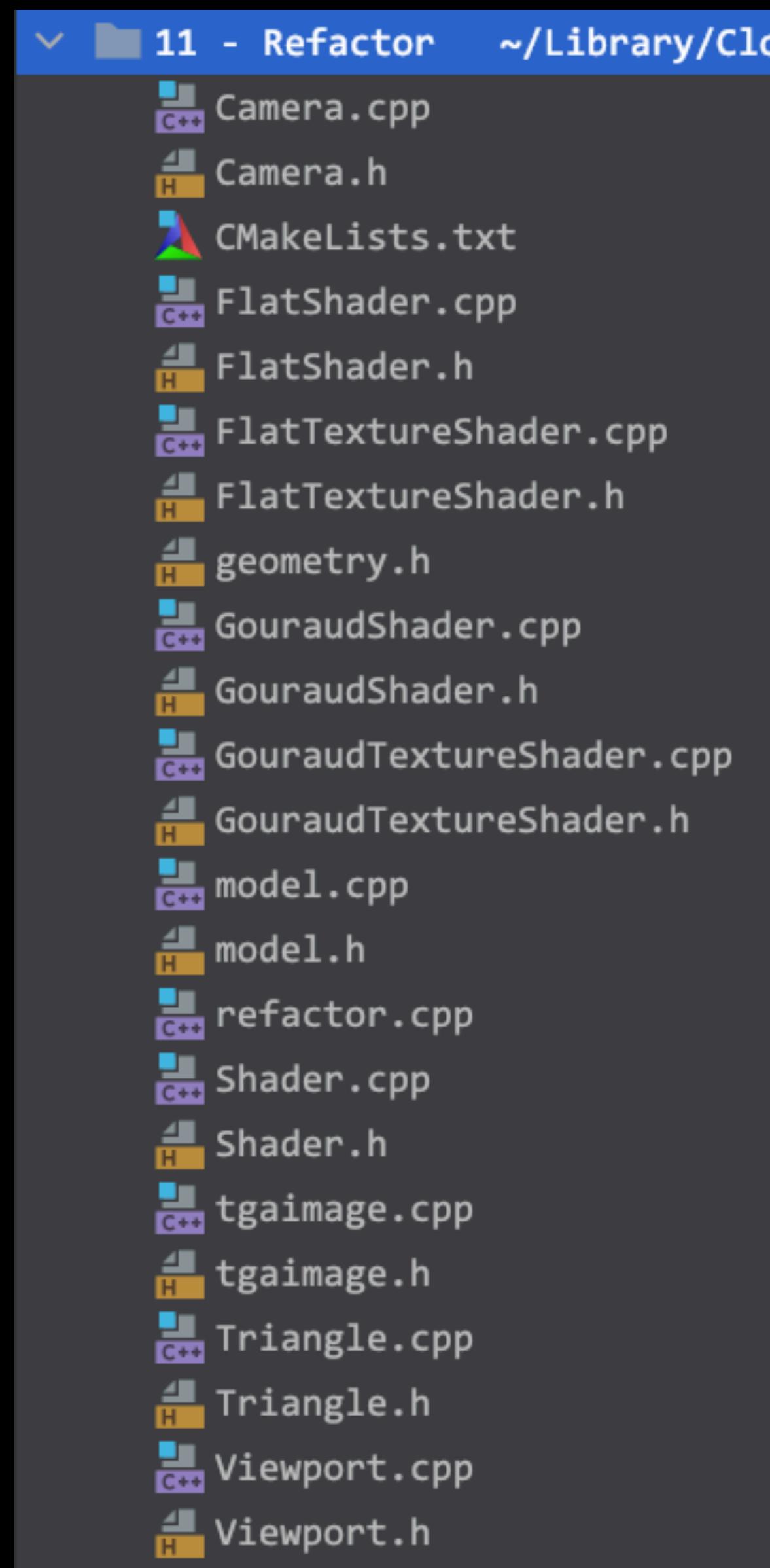
```

# Votre mission

- ♦ Un programme avec le moins de code dupliqué possible qui génère l'image ci-contre en fournissant 4 shaders

Flat Shader	Flat Texture Shader
Gouraud Shader	Gouraud Texture Shader

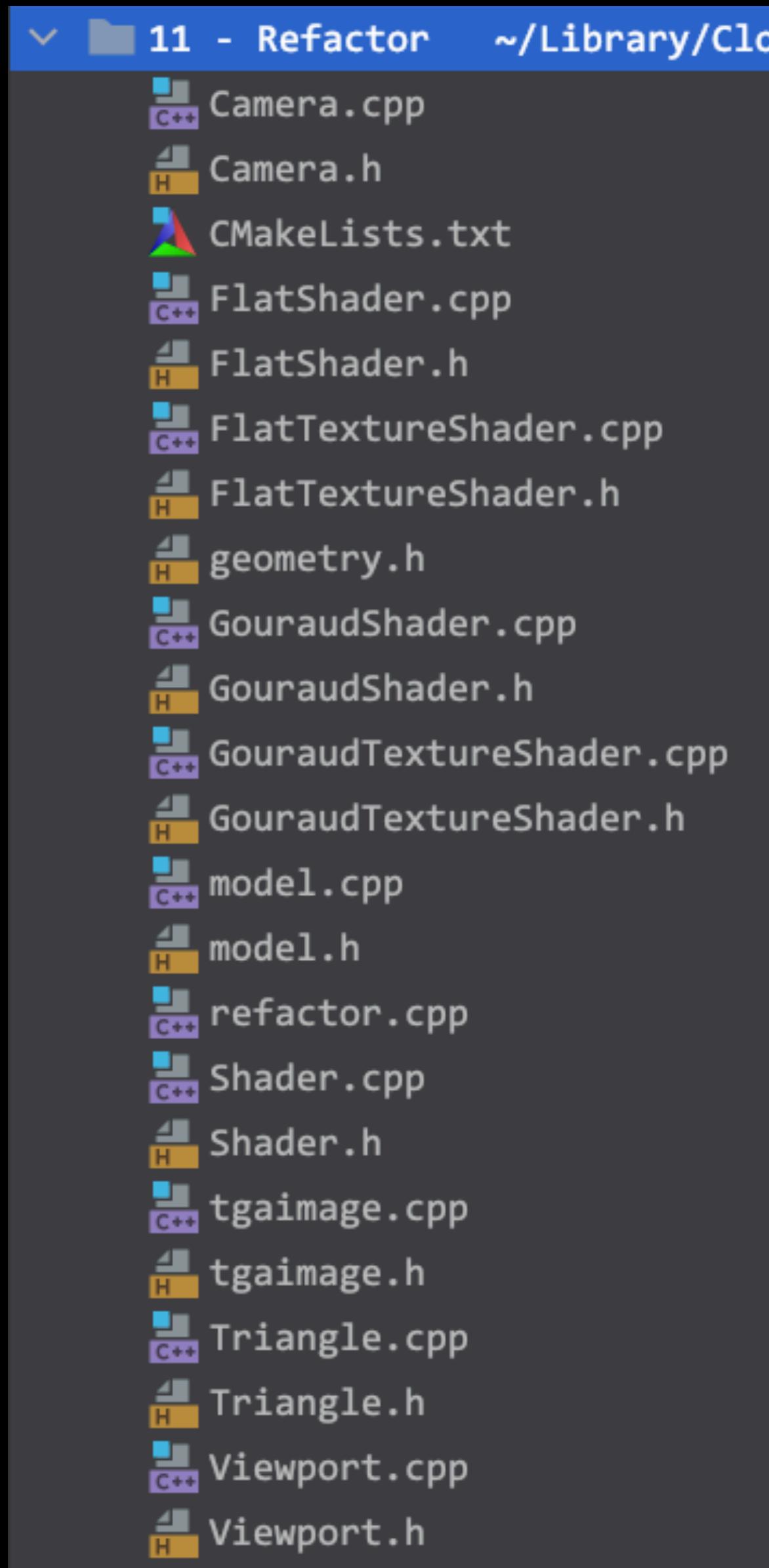




# Code fourni

```
#include ...  
  
#include "Camera.h"  
#include "Viewport.h"  
#include "Triangle.h"  
  
#include "FlatShader.h"  
#include "FlatTextureShader.h"  
#include "GouraudShader.h"  
#include "GouraudTextureShader.h"  
  
using namespace std;  
  
int main() {  
    ...  
    vector<Shader*> shaders = { &gouraudShader, &gouraudTextureShader,  
                                &flatShader, &flatTextureShader };  
  
    for(size_t i = 0; i < shaders.size(); ++i) {  
        ...  
    }  
    ...  
}
```

# Code fourni



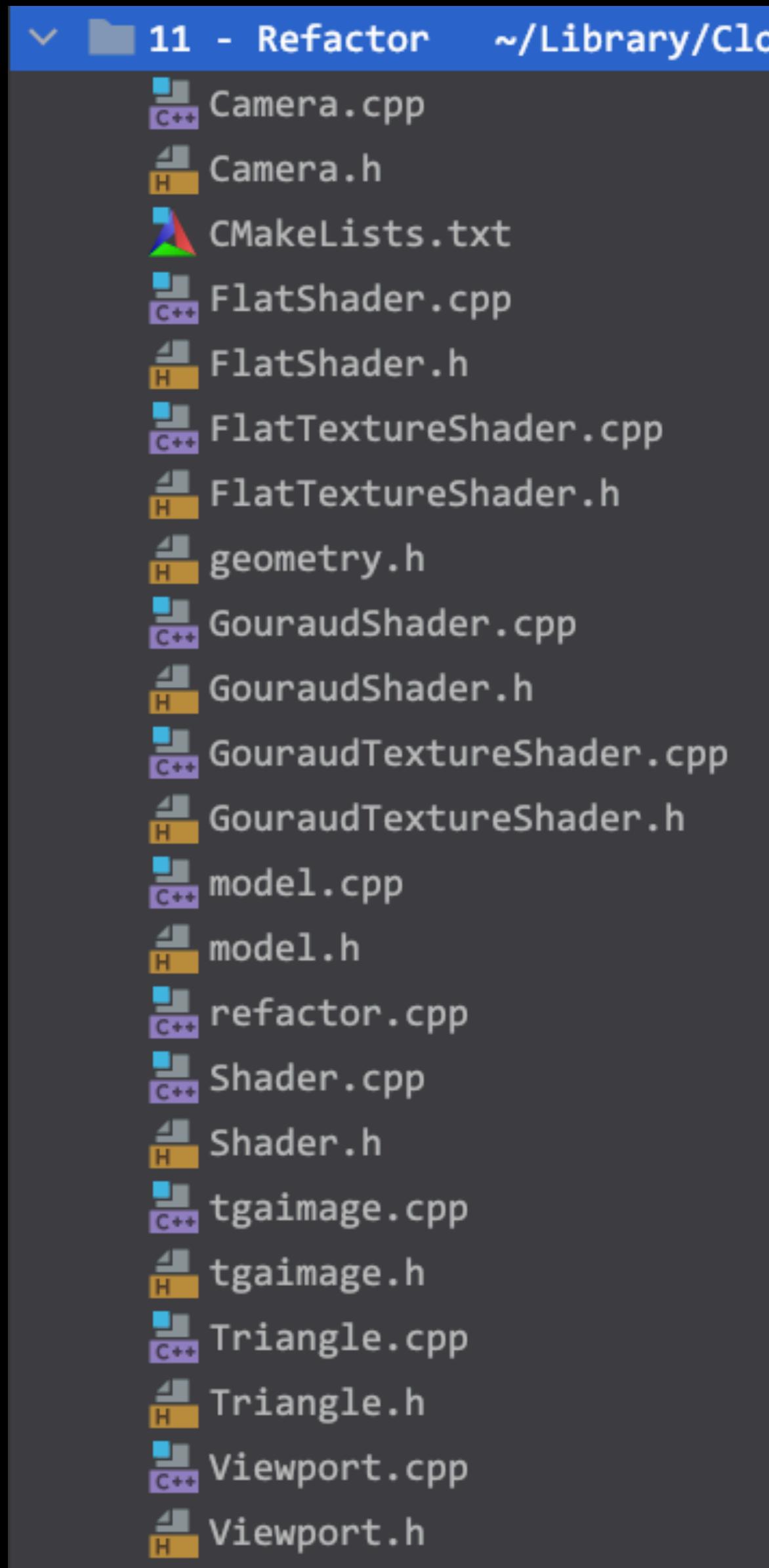
```
#ifndef CPPRENDERER_CAMERA_H
#define CPPRENDERER_CAMERA_H

#include "geometry.h"

struct Camera {
    Vec3f eye, center, up;
    float focalDistance() const;
    Matrix4x4 view() const;
    Matrix4x4 projection() const;
};

#endif //CPPRENDERER_CAMERA_H
```

# Code fourni



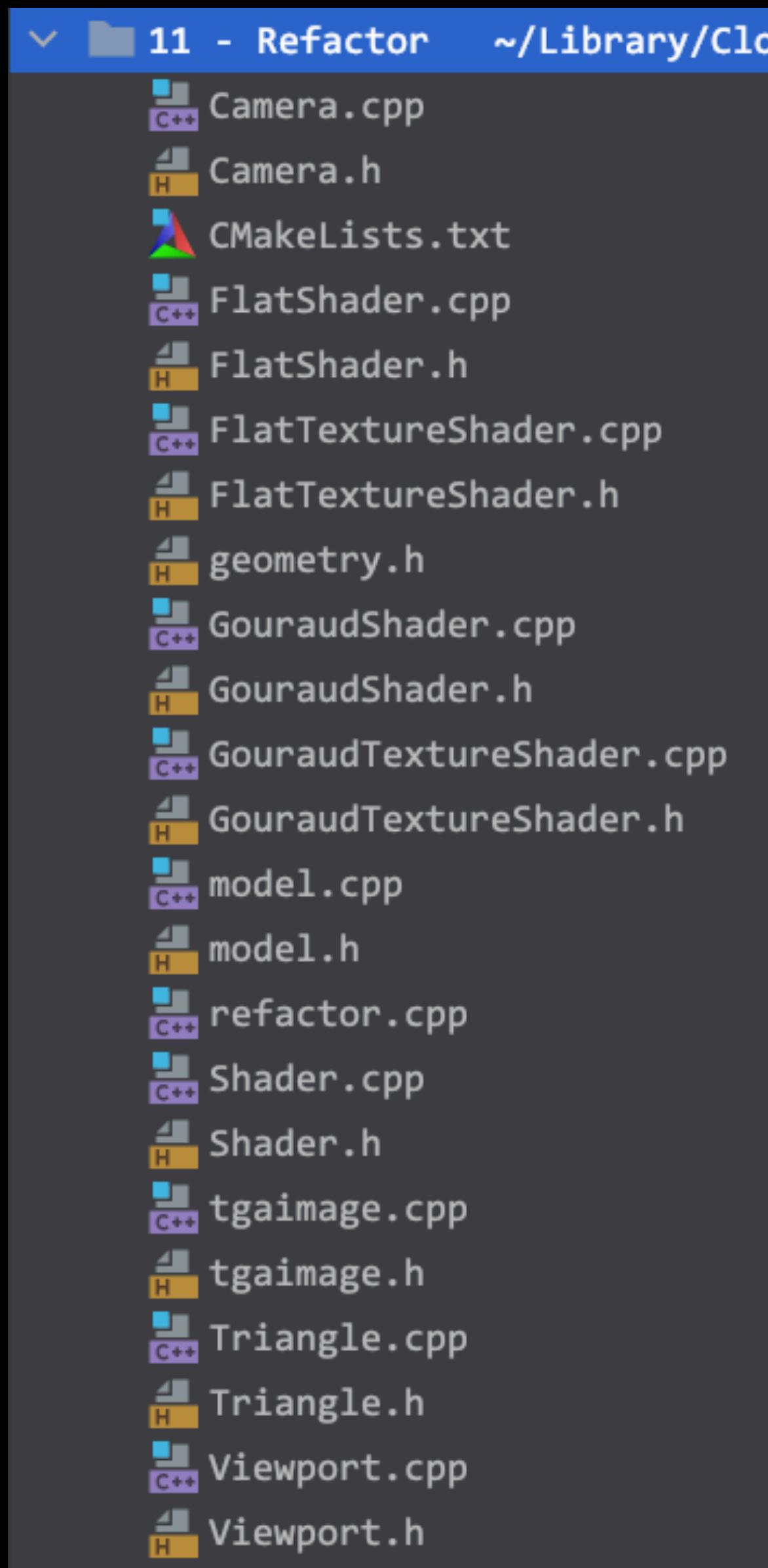
```
#ifndef CPPRENDERER_VIEWPORT_H
#define CPPRENDERER_VIEWPORT_H

#include "geometry.h"

Matrix4x4 make_viewport(int x, int y, size_t w, size_t h,
                        float zoom = 0.4f);

#endif //CPPRENDERER_VIEWPORT_H
```

# Code fourni

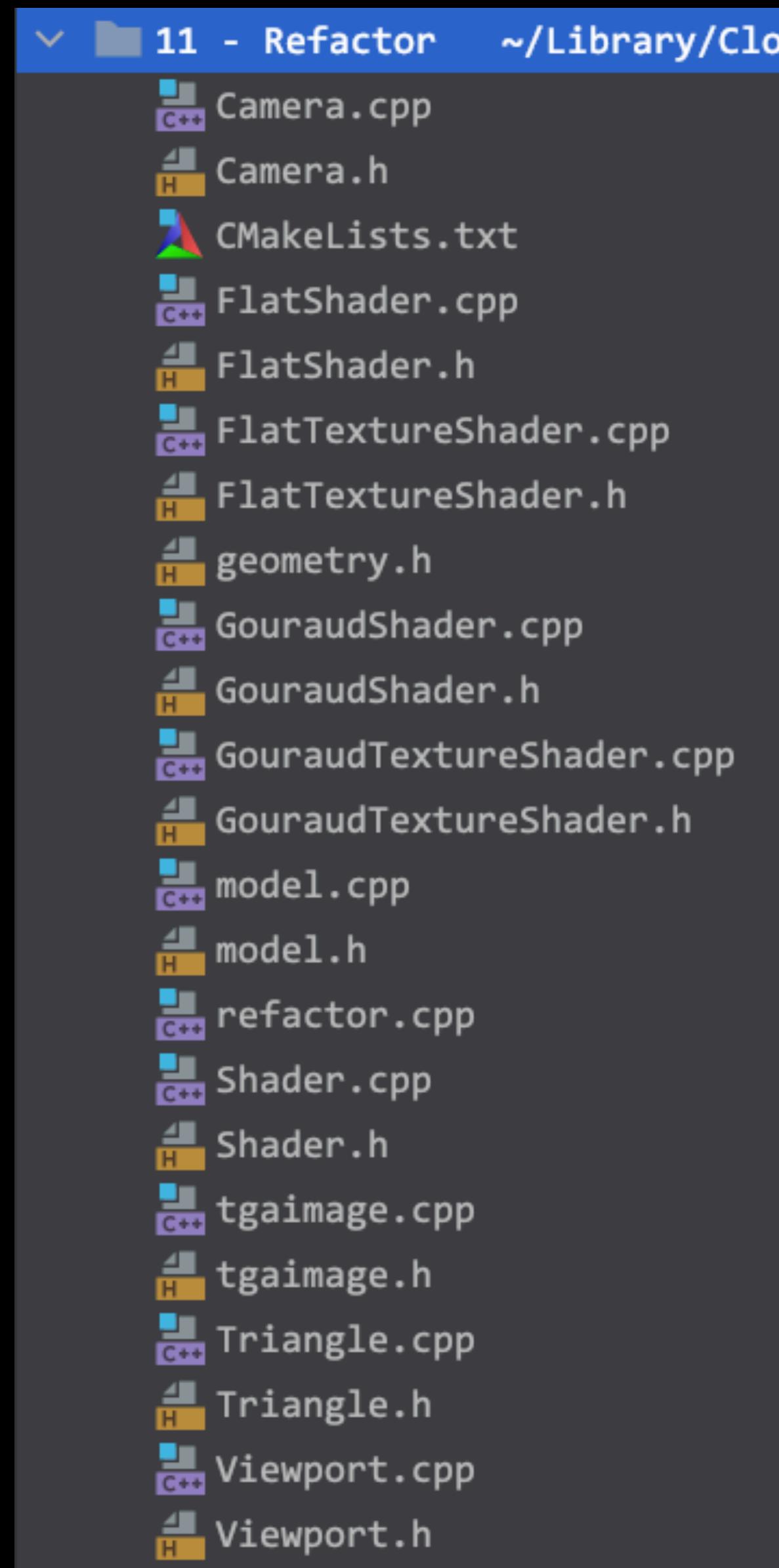


```
#ifndef CPPRENDERER_TRIANGLE_H
#define CPPRENDERER_TRIANGLE_H

#include "geometry.h"
#include "tgaiimage.h"
#include "Shader.h"

void triangle(Vec3f screen[3], Shader& shader,
              float* zbuffer, TGAIImage &image);

#endif //CPPRENDERER_TRIANGLE_H
```



# Code fourni

```
#ifndef CPPRENDERER_SHADER_H
#define CPPRENDERER_SHADER_H

#include ...

struct Shader {
    virtual ~Shader() = default;
    virtual Vec3f vertex(size_t face, size_t vert);
    virtual bool fragment(Vec3f bary, TGAColor& color);

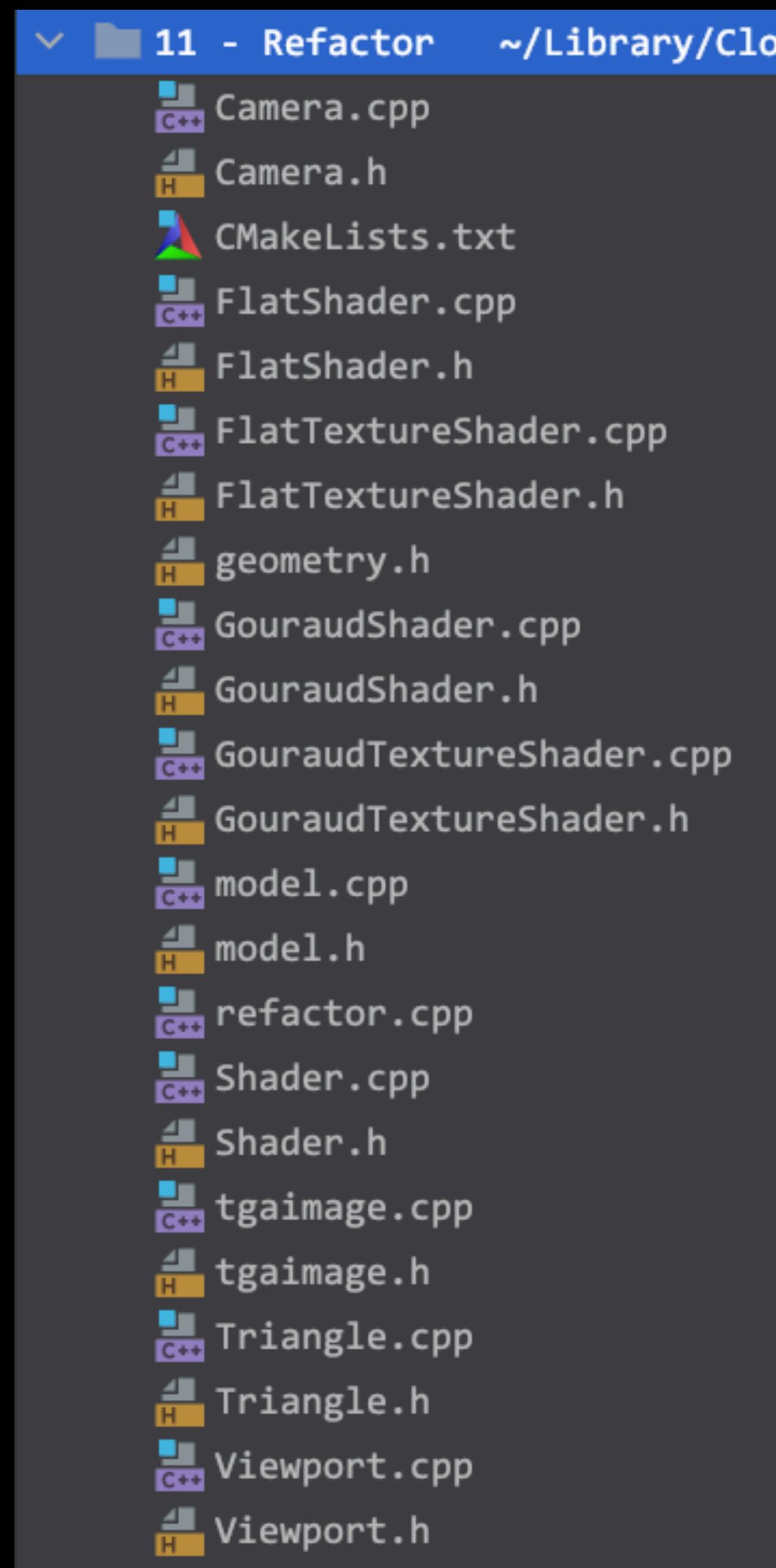
    Vec3f light;
    Matrix4x4 view;
    Matrix4x4 projection;
    Matrix4x4 viewport;
    Model* model = nullptr;
};

#endif

Vec3f Shader::vertex(size_t face, size_t vert) {
    Vec3f world = model->vert(model->face(face)[vert]);
    return viewport * projection * view * VecH(world);
}

bool Shader::fragment(Vec3f bary, TGAColor& color) {
    color = TGAColor(128,128,128,255);
    return true;
```

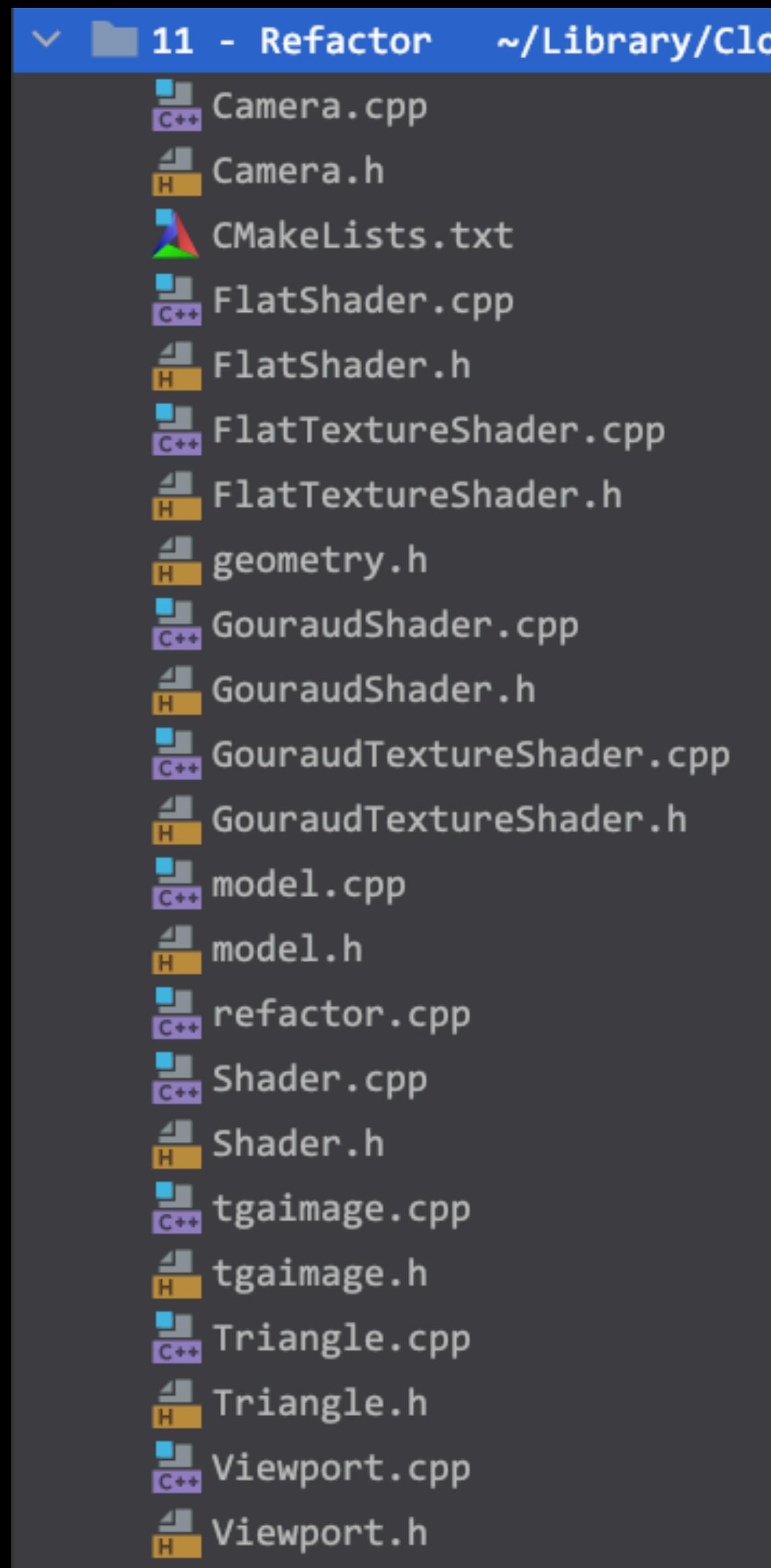
# Code fourni : sortie actuelle



```
11 - Refactor ~/Library/Clouds
  Camera.cpp
  Camera.h
  CMakeLists.txt
  FlatShader.cpp
  FlatShader.h
  FlatTextureShader.cpp
  FlatTextureShader.h
  geometry.h
  GouraudShader.cpp
  GouraudShader.h
  GouraudTextureShader.cpp
  GouraudTextureShader.h
  model.cpp
  model.h
  refactor.cpp
  Shader.cpp
  Shader.h
  tgimage.cpp
  tgimage.h
  Triangle.cpp
  Triangle.h
  Viewport.cpp
  Viewport.h
```



# Code à modifier



```
#ifndef CPPRENDERER_FLATSHADER_H
#define CPPRENDERER_FLATSHADER_H

#include "Shader.h"

struct FlatShader : public Shader {
    TGAColor color;
private:
    float intensity;
};

#endif //CPPRENDERER_FLATSHADER_H


#ifndef CPPRENDERER_GOURAUDSHADER_H
#define CPPRENDERER_GOURAUDSHADER_H

#include "Shader.h"

struct GouraudShader : public Shader {
    TGAColor color;
private:
    Vec3f varying_intensity;
};

#endif //CPPRENDERER_GOURAUDSHADER_H


#ifndef CPPRENDERER_FLATTEXTURESHADER_H
#define CPPRENDERER_FLATTEXTURESHADER_H

#include "Shader.h"

struct FlatTextureShader : public Shader {
    TGAIImage* texture;
private:
    Vec3f varying_uv[3];
    float intensity;
};

#endif //CPPRENDERER_FLATTEXTURESHADER_H


#ifndef CPPRENDERER_GOURAUDTEXTURESHADER_H
#define CPPRENDERER_GOURAUDTEXTURESHADER_H

#include "Shader.h"

struct GouraudTextureShader : public Shader {
    TGAIImage* texture;
private:
    Vec3f varying_intensity;
    Vec3f varying_uv[3];
};

#endif //CPPRENDERER_GOURAUDTEXTURESHADER_H
```

# Shaders (.h)

```
#ifndef CPPRENDERER_FLATSHADER_H
#define CPPRENDERER_FLATSHADER_H

#include "Shader.h"

struct FlatShader : public Shader,
    public ColorShaderBase,
    public FlatShaderBase {
    virtual Vec3f vertex(size_t face, size_t vert);
    virtual bool fragment(Vec3f bary, TGAColor &color);
};

#endif //CPPRENDERER_FLATSHADER_H


#ifndef CPPRENDERER_GOURAUDSHADER_H
#define CPPRENDERER_GOURAUDSHADER_H

#include "Shader.h"

struct GouraudShader
    : public Shader,
    public ColorShaderBase,
    public GouraudShaderBase {
    virtual Vec3f vertex(size_t face, size_t vert);
    virtual bool fragment(Vec3f bary, TGAColor &color);
};

#endif //CPPRENDERER_GOURAUDSHADER_H


#ifndef CPPRENDERER_FLATTEXTURESHADER_H
#define CPPRENDERER_FLATTEXTURESHADER_H

#include "Shader.h"

struct FlatTextureShader : public Shader,
    public TextureShaderBase,
    public FlatShaderBase {
    virtual Vec3f vertex(size_t face, size_t vert);
    virtual bool fragment(Vec3f bary, TGAColor &color);
};

#endif //CPPRENDERER_FLATTEXTURESHADER_H


#ifndef CPPRENDERER_GOURAUDTEXTURESHADER_H
#define CPPRENDERER_GOURAUDTEXTURESHADER_H

#include "Shader.h"

struct GouraudTextureShader
    : public Shader,
    public TextureShaderBase,
    public GouraudShaderBase {
    virtual Vec3f vertex(size_t face, size_t vert);
    virtual bool fragment(Vec3f bary, TGAColor& color);
};

#endif //CPPRENDERER_GOURAUDTEXTURESHADER_H
```

# Shaders (.cpp)

```
#include "FlatShader.h"

Vec3f FlatShader::vertex(size_t face, size_t vert) {
    FlatShaderBase::vertex(model, light, face, vert);
    return Shader::vertex(face, vert);
}

bool FlatShader::fragment(Vec3f bary, TGAColor &color) {
    ColorShaderBase::fragment(color);
    return FlatShaderBase::fragment(bary, color);
}
```

```
#include "GouraudShader.h"

Vec3f GouraudShader::vertex(size_t face, size_t vert) {
    GouraudShaderBase::vertex(model, light, face, vert);
    return Shader::vertex(face, vert);
}

bool GouraudShader::fragment(Vec3f bary, TGAColor &color) {
    ColorShaderBase::fragment(color);
    return GouraudShaderBase::fragment(bary, color);
}
```

```
#include "FlatTextureShader.h"

Vec3f FlatTextureShader::vertex(size_t face, size_t vert) {
    TextureShaderBase::vertex(model, face, vert);
    FlatShaderBase::vertex(model, light, face, vert);
    return Shader::vertex(face, vert);
}

bool FlatTextureShader::fragment(Vec3f bary, TGAColor &color) {
    TextureShaderBase::fragment(bary, color);
    return FlatShaderBase::fragment(bary, color);
}
```

```
#include "GouraudTextureShader.h"

Vec3f GouraudTextureShader::vertex(size_t face, size_t vert) {
    TextureShaderBase::vertex(model, face, vert);
    GouraudShaderBase::vertex(model, light, face, vert);
    return Shader::vertex(face, vert);
}

bool GouraudTextureShader::fragment(Vec3f bary, TGAColor &color) {
    TextureShaderBase::fragment(bary, color);
    return GouraudShaderBase::fragment(bary, color);
}
```

# Bases - couleur sous-jacente

```
struct ColorShaderBase {
    TGAColor color = TGAColor(255, 255, 255, 255);
    void fragment(TGAColor& color);
};

void ColorShaderBase::fragment(TGAColor& color) {
    color = this->color;
}

struct TextureShaderBase {
    TGAImage* texture = nullptr;
    void vertex(Model* model, size_t face, size_t vert);
    void fragment(Vec3f bary, TGAColor& color);
protected:
    Vec3f varying_uv[3];
};

void TextureShaderBase::vertex(Model* model, size_t face, size_t vert) {
    varying_uv[vert] = model->texture(size_t(model->face_texts(face)[vert]));
}

void TextureShaderBase::fragment(Vec3f bary, TGAColor &color) {
    Vec3f ptc = varying_uv[0] * bary.x + varying_uv[1] * bary.y + varying_uv[2] * bary.z;
    Vec2i pc( int(round(ptc.x * float(texture->get_width()))),
              int(round(ptc.y * float(texture->get_height()))));
    color = texture->get(pc.x, pc.y);
}
```

# Bases - ombrage

```

#define World(vertIdx) model->vert(size_t(model->face(face)[(vertIdx)]))
void FlatShaderBase::vertex(Model* model, Vec3f light,
                             size_t face, size_t vert) {
    if(vert == 0) {
        Vec3f n = (World(1)- World(0)) ^ (World(2) - World(0));
        n.normalize();
        intensity = n * light;
    }
}

bool FlatShaderBase::fragment([[maybe_unused]] Vec3f bary, TGAColor& color) {
    if(intensity < 0) return false;
    for(int i = 0; i < 3; ++i)
        color.raw[i] = (unsigned char)(float(color.raw[i]) * intensity);
    return true;
}

struct GouraudShaderBase {
    void vertex(Model* model, Vec3f light,
                size_t face, size_t vert);
    bool fragment(Vec3f bary, TGAColor& color);
private:
    Vec3f varying_intensity;
};

void GouraudShaderBase::vertex(Model* model, Vec3f light,
                               size_t face, size_t vert) {
    Vec3f n = model->normal(size_t(model->face_normals(face)[vert]));
    varying_intensity.raw[vert] = n * light;
}

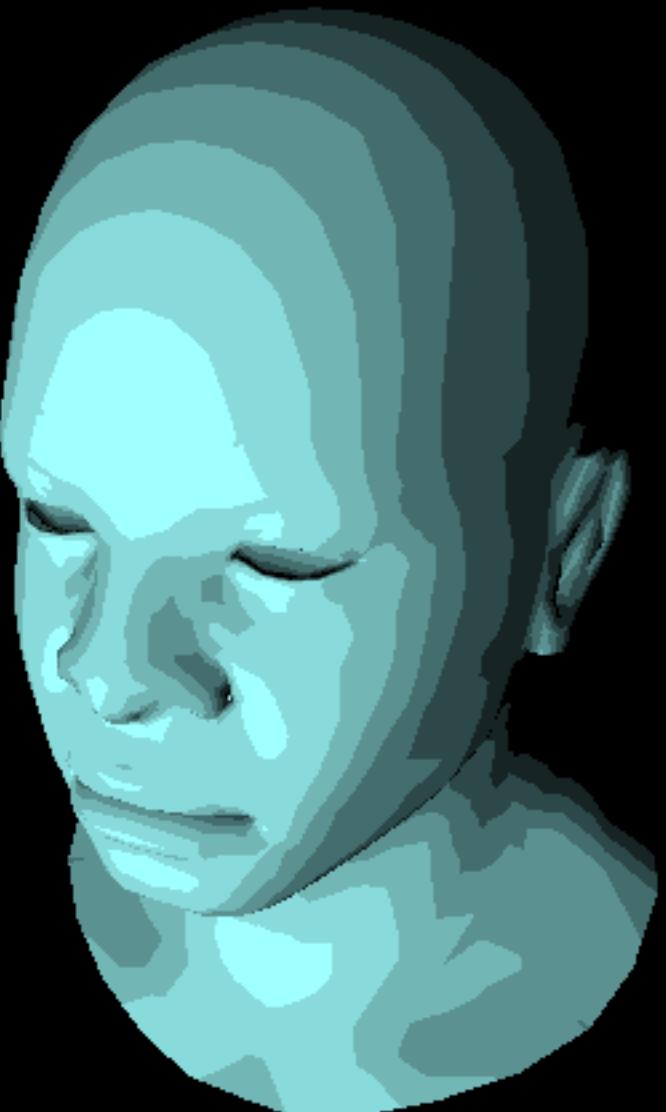
bool GouraudShaderBase::fragment(Vec3f bary, TGAColor& color) {
    float pixelIntensity = varying_intensity * bary;
    if (pixelIntensity < 0) return false;
    for (int i = 0; i < 3; ++i)
        color.raw[i] = (unsigned char) (float(color.raw[i]) * pixelIntensity);
    return true;
}

```

# Quantized Gouraud Shader

# Votre mission ...

- ❖ Créez un QuantizedGouraudShader qui limite discrétise l'intensité de Gouraud sur QuantizedGouraudShader::quants niveaux



# QuantizedGouraudShader

```
#ifndef CPPRENDERER_QUANTIZEDGOURAUDSHADER_H
#define CPPRENDERER_QUANTIZEDGOURAUDSHADER_H

#include "GouraudShader.h"

struct QuantizedGouraudShader
    : public GouraudShader {
    unsigned quants = 5;
    virtual bool fragment(Vec3f bary, TGAColor &color);
};

#endif //CPPRENDERER_QUANTIZEDGOURAUDSHADER_H

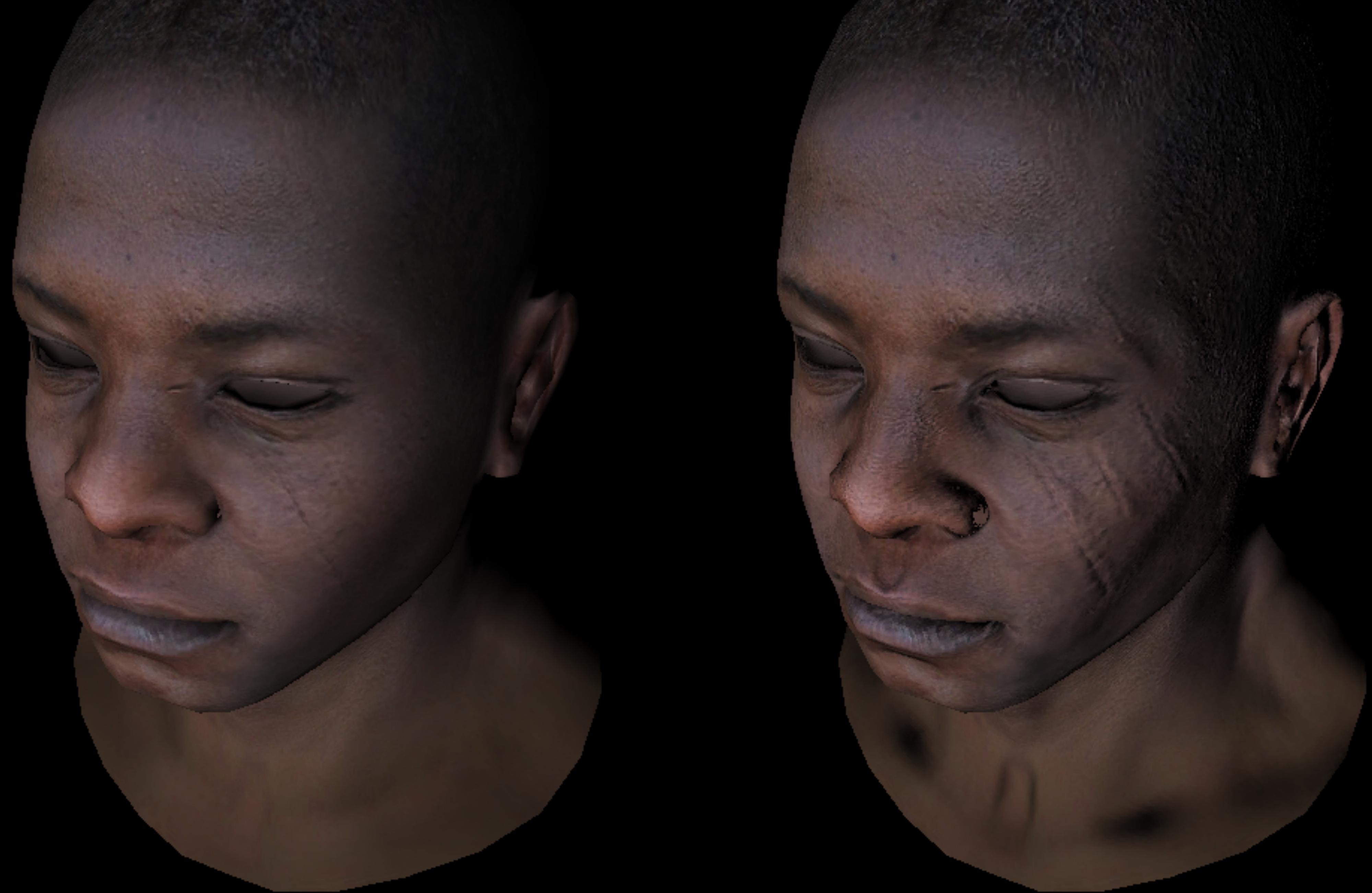
#include "QuantizedGouraudShader.h"

bool QuantizedGouraudShader::fragment(Vec3f bary, TGAColor& color) {
    ColorShaderBase::fragment(color);

    float pixelIntensity = varying_intensity * bary;
    if (pixelIntensity < 0) return false;
    pixelIntensity = round(pixelIntensity * quants) / quants;
    for (int i = 0; i < 3; ++i)
        color.raw[i] = (unsigned char) (float(color.raw[i]) * pixelIntensity);

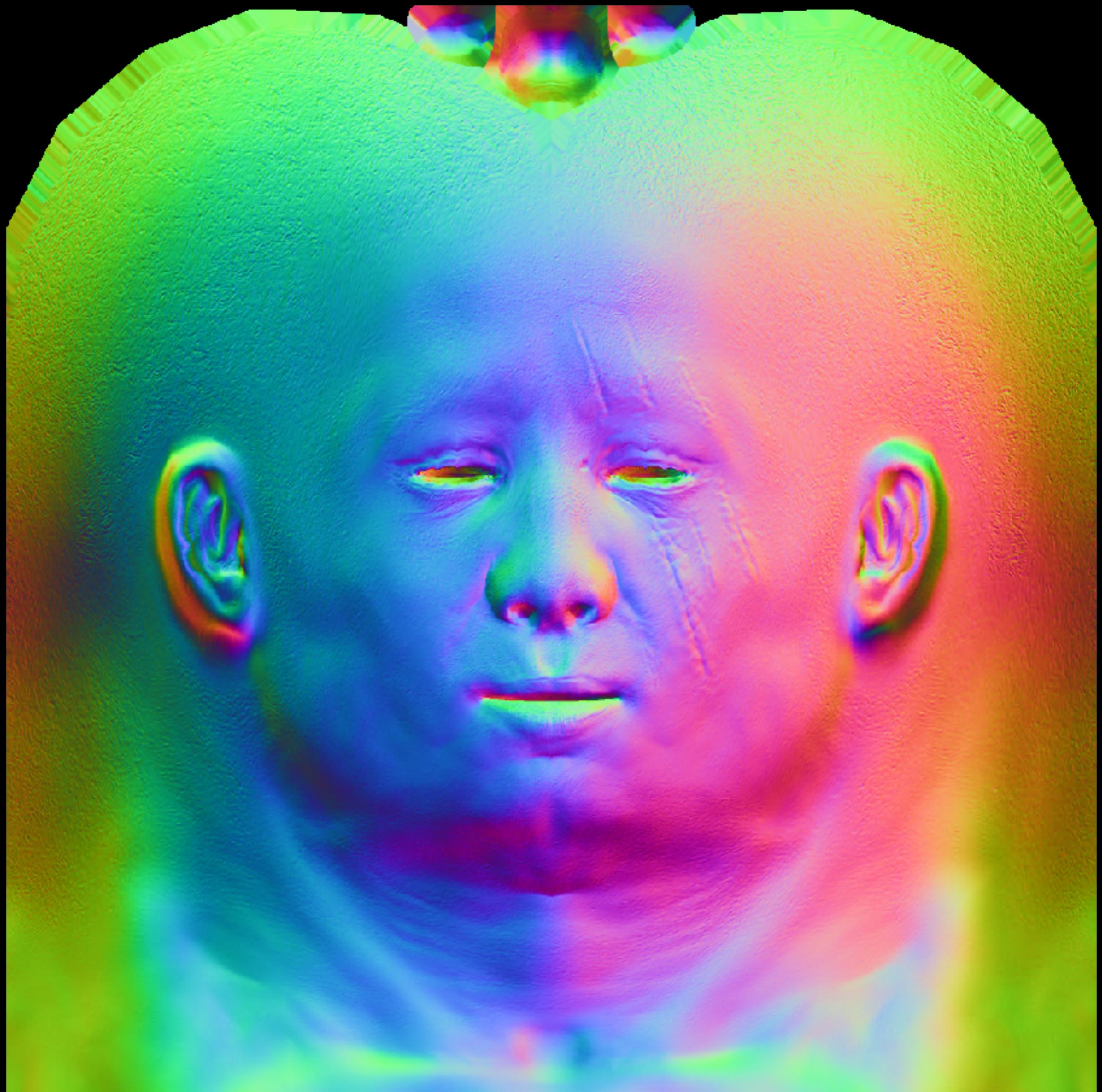
    return true;
}
```

# Normal Texture Shader



# african\_head\_nm.tga

- ◆ Plutôt que d'interpoler la normale à partir des normales aux sommets (Gouraud)
- ◆ On lit la normal dans une image de texture à plus haute résolution
- ◆ Votre mission : écrire NormalTextureShader



# NormalTextureShader (.h) (.cpp)

```
#ifndef CPPRENDERER_NORMALTEXTURESHADER_H      #include "NormalTextureShader.h"

#define CPPRENDERER_NORMALTEXTURESHADER_H

#include "GouraudTextureShader.h"

struct NormalTextureShader :           bool NormalTextureShader::fragment(Vec3f bary, TGAColor &color) {
    public GouraudTextureShader {             TextureShaderBase::fragment(bary, color);

        TGAImage* normals;                 Vec3f ptc = varying_uv[0] * bary.x + varying_uv[1] * bary.y + varying_u
        virtual bool fragment(Vec3f bary,          Vec2i pc(int(round(ptc.x * float(normals->get_width()))),int(round(ptc
            TGAColor& color);                  normal = normals->get(pc.x, pc.y);

    };                                         auto n = Vec3f{ float(normal.raw[2]), float(normal.raw[1]),
                                                               float(normal.raw[0]) } * (2.f/255.f) - Vec3f{1.f,1.f,1
#endif //CPPRENDERER_NORMALTEXTURESHADER_H                                         n.normalize();

                                         float pixelIntensity = n * light;
                                         if (pixelIntensity < 0) return false;
                                         for (int i = 0; i < 3; ++i)
                                             color.raw[i] = (unsigned char) (float(color.raw[i]) * pixelIntensity

                                         return true;
}
```