

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 TGAImage minimaliste.
- ♦ Constructeur définit la taille
- ♦ La méthode set permet d'écrire dans un pixel

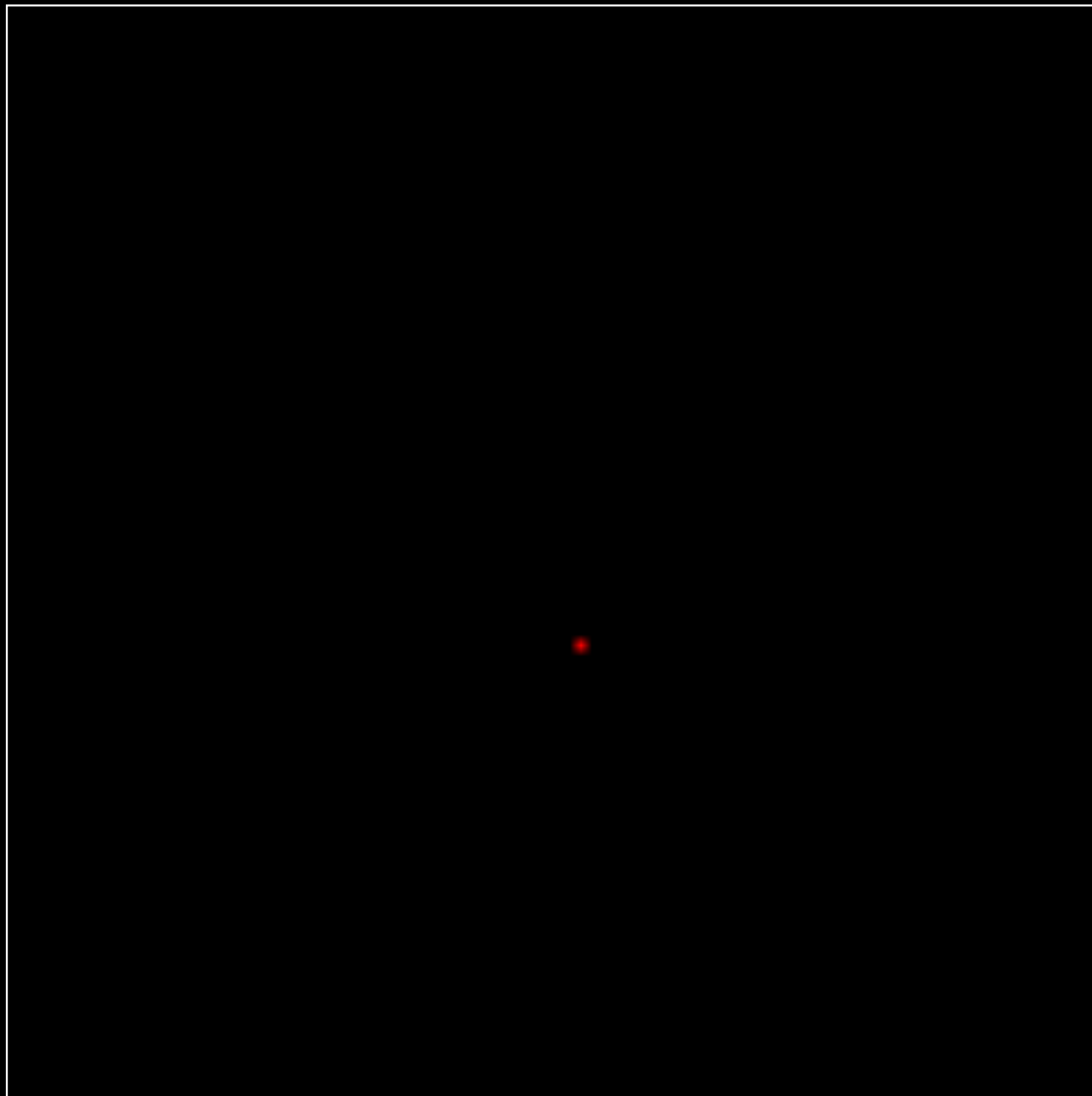
```
class TGAImage {
protected:
    unsigned char* data;
    int width;
    int height;
    int bytespp;

    bool load_rle_data(std::ifstream &in);
    bool unload_rle_data(std::ofstream &out);
public:
    enum Format {
        GRAYSCALE=1, RGB=3, RGBA=4
    };

    TGAImage();
    TGAImage(int w, int h, int bpp);
    TGAImage(const TGAImage &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);
    ~TGAImage();
    TGAImage & operator =(const TGAImage &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,  
          TGImage &image, TGAColor color);
```

```
int main(int argc, char** argv) {  
    TGImage image(100, 100, TGImage::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...

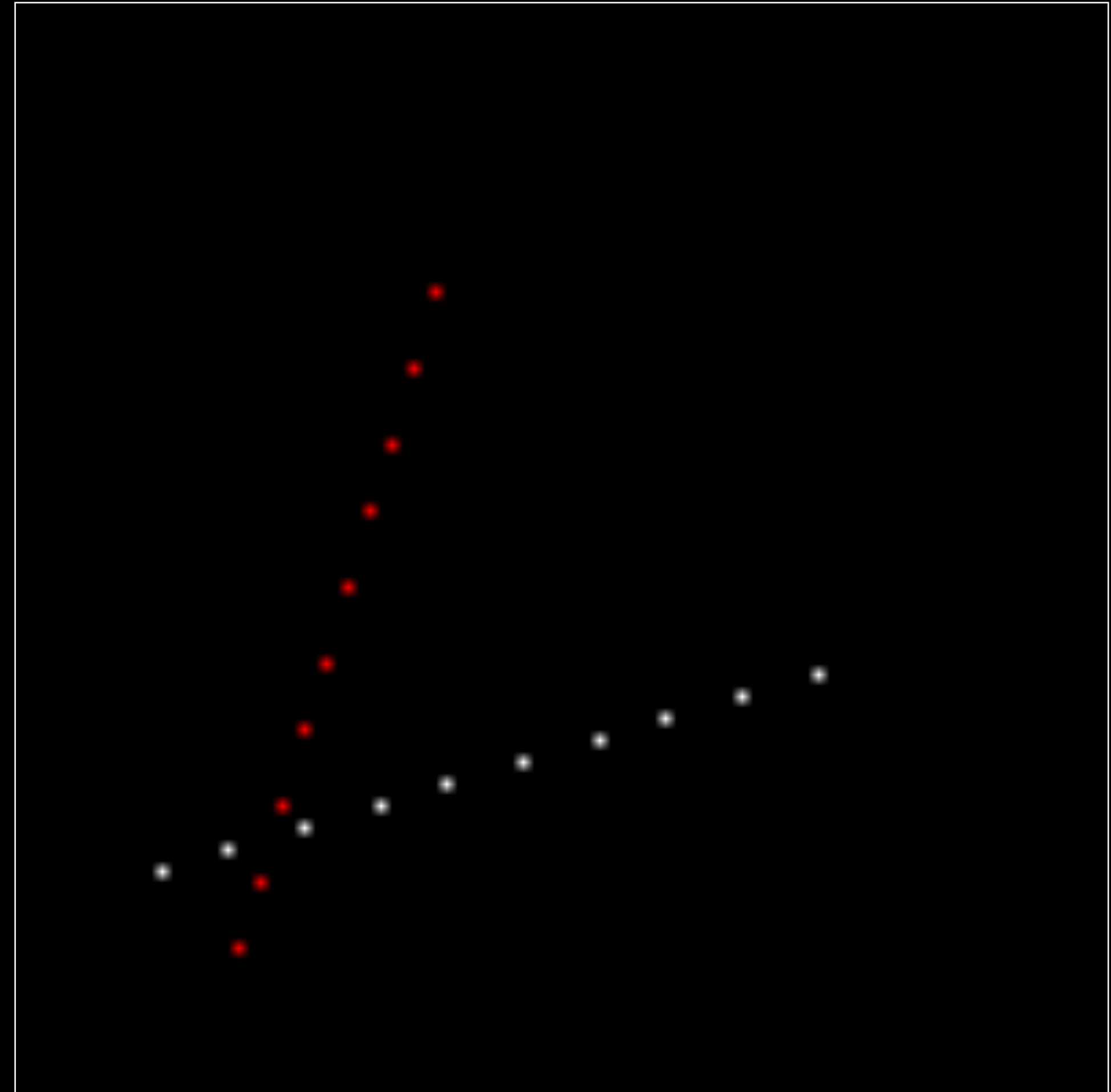
```
void line(int x0, int y0, int x1, int y1,  
          TGAImage &image, TGAColor color)  
{  
    for (float t=0.; t<1.; t+=.01) {  
        int x = x0 + (x1-x0)*t;  
        int y = y0 + (y1-y0)*t;  
        image.set(x, y, color);  
    }  
}
```

- ✦ Fonctionne ... apparemment
- ✦ Méchant nombre magique ...

t+=.01
- ✦ Pas efficace si le pas est trop petit
- ✦ Pas correct si le pas est trop grand

$t += .1$

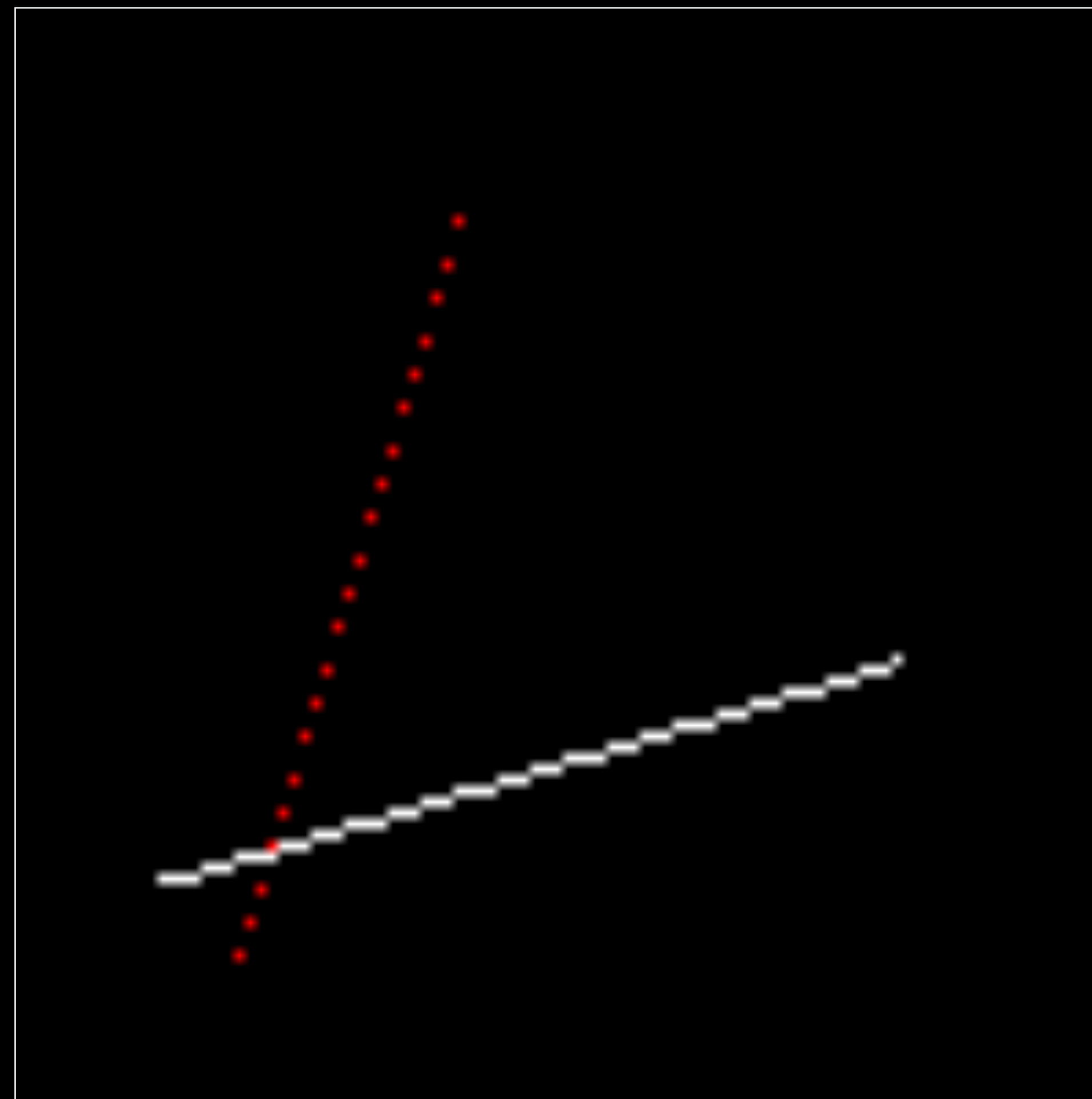
```
void line(int x0, int y0, int x1, int y1,
          TGAImage &image, TGAColor color)
{
    for (float t=0.; t<1.; t+=.1) {
        int x = x0 + (x1-x0)*t;
        int y = y0 + (y1-y0)*t;
        image.set(x, y, color);
    }
}
```



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 $(x1-x0)$ 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 $y = y_i + \text{sign}(y_1 - y_0) * y_p$

```
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);
}
```

```
float ystep = std::abs(y1-y0)/float(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);
}
```

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

```
float ystep = std::abs(y1-y0)/float(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 dx = (x1-x0);
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)
```

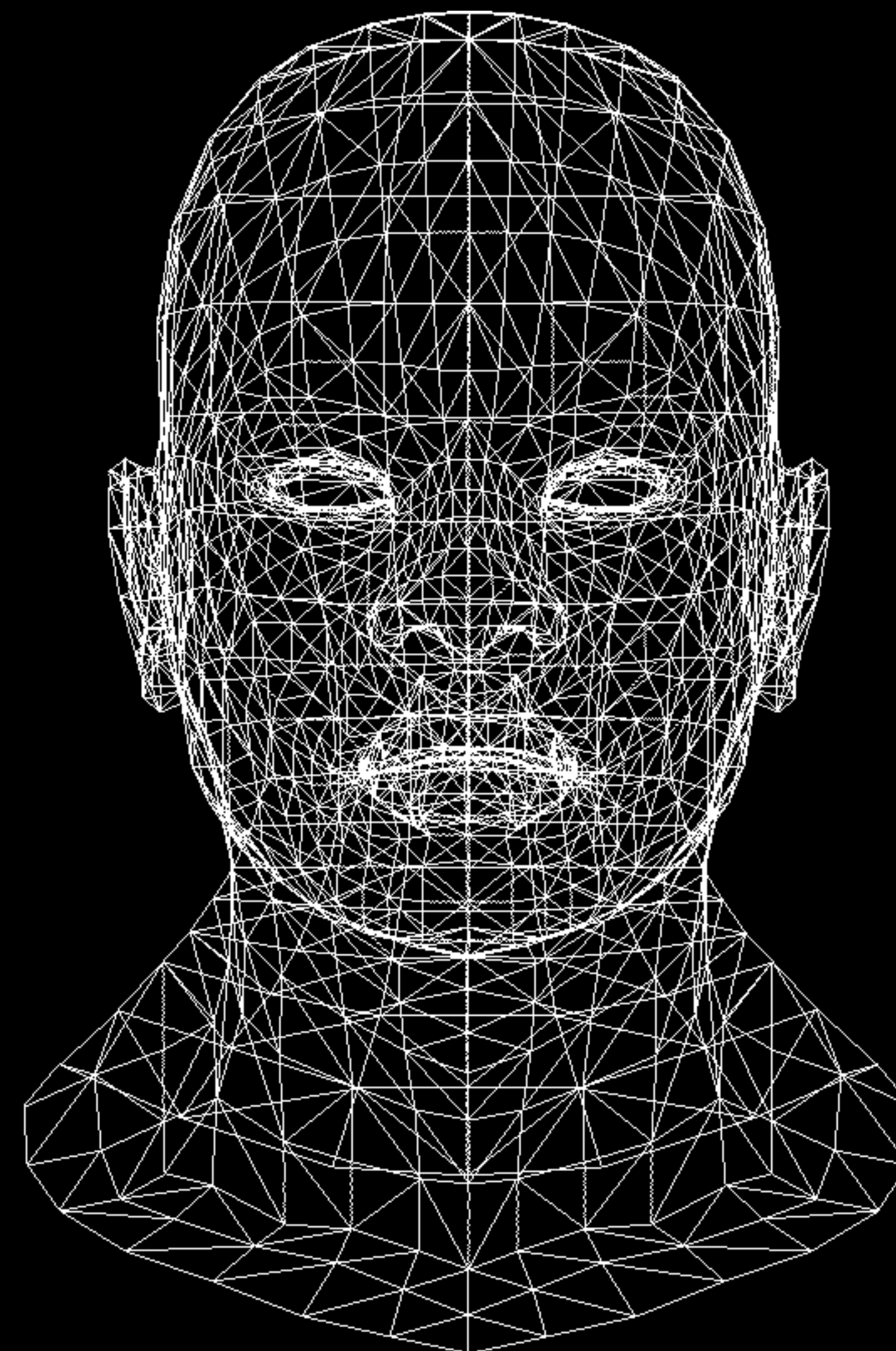
```
{  
    bool steep = false;  
    if (std::abs(x0-x1)<std::abs(y0-y1)) {  
        std::swap(x0, y0);  
        std::swap(x1, y1);  
        steep = true;  
    }  
    if (x0>x1) {        std::swap(x0, x1);  
        std::swap(y0, y1);  
    }  
}
```

```
    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 ...
...

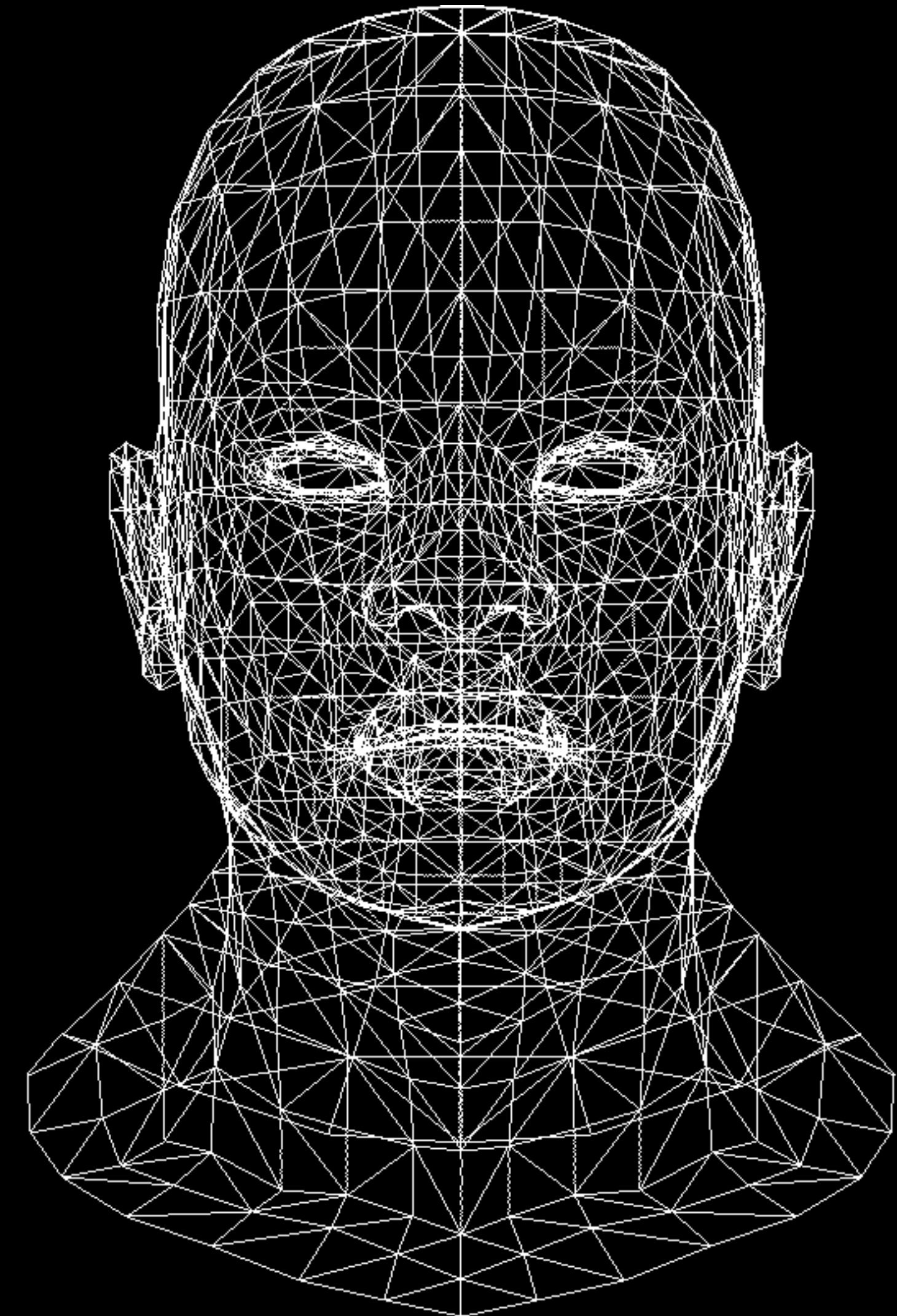
# Parameter space vertices in ( u [,v] [,w] ) form; free form
geometry statement ( see below )
vp 0.310000 3.210000 2.100000
vp ...
...

# 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
```

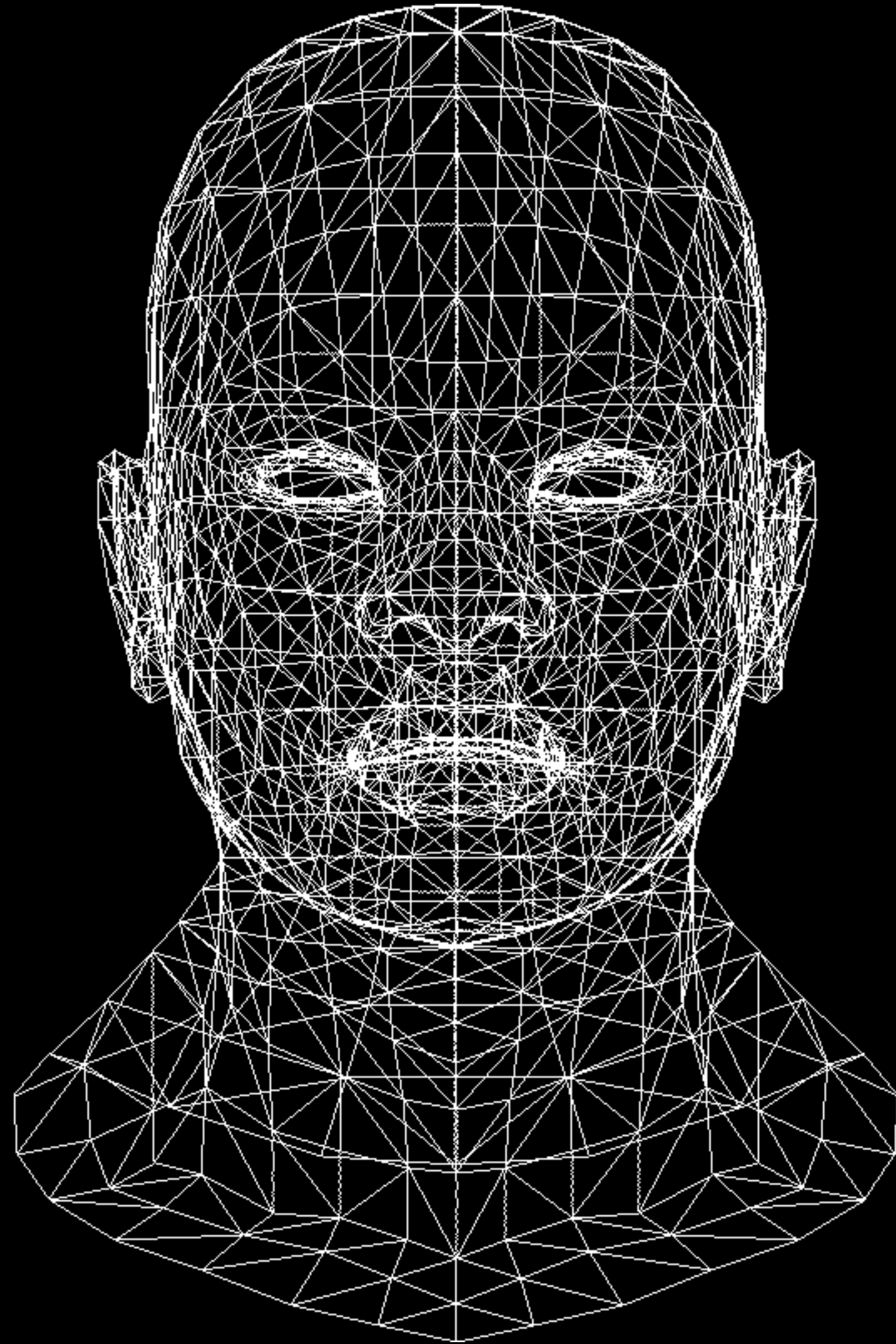

Votre but

- ♦ 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

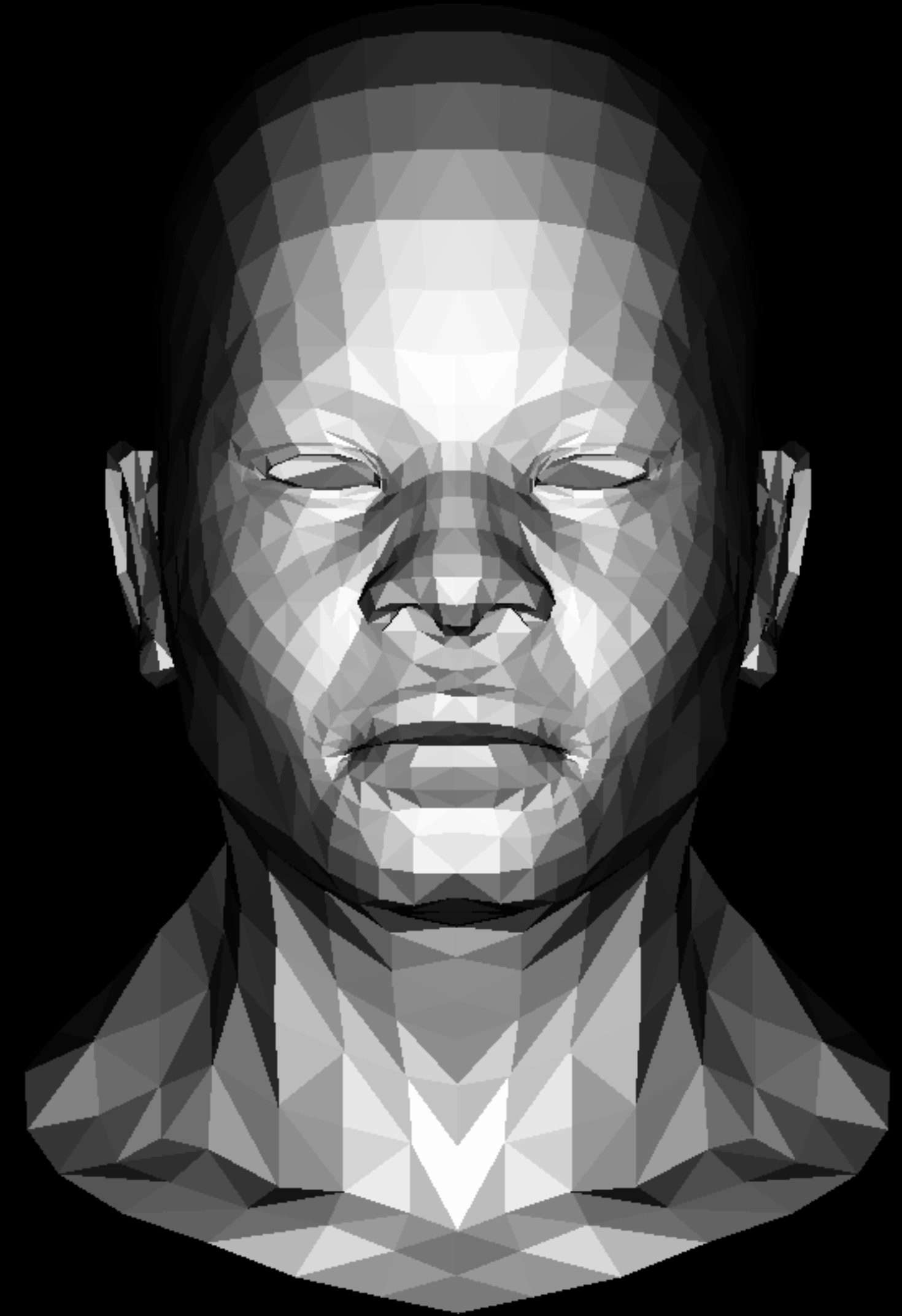


Dessiner un triangle...

Avant



Après



Code fourni

```
#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);
}
```

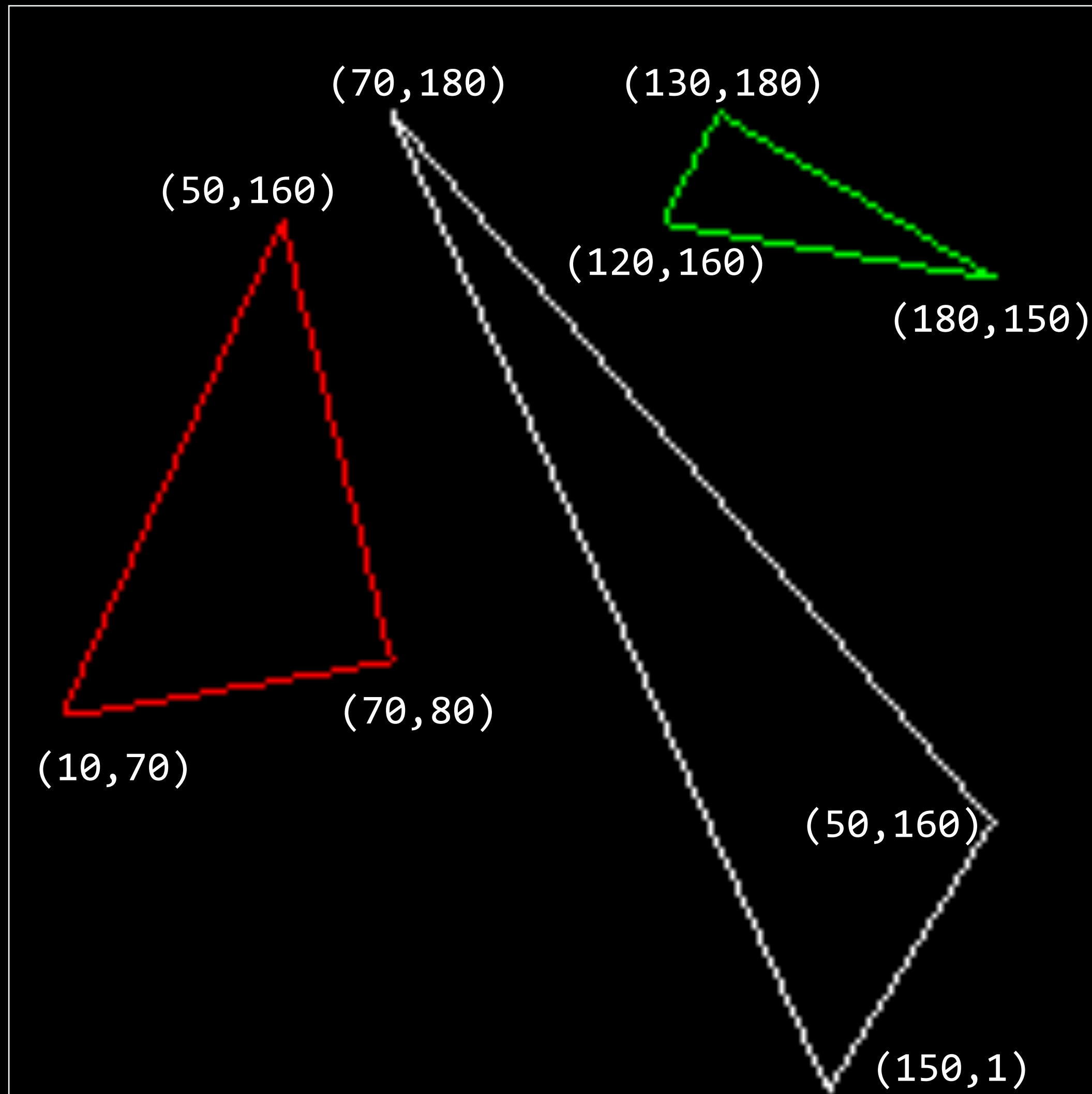
```
int main() {
    TGAIImage image(width, height, TGAIImage::RGB);

    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



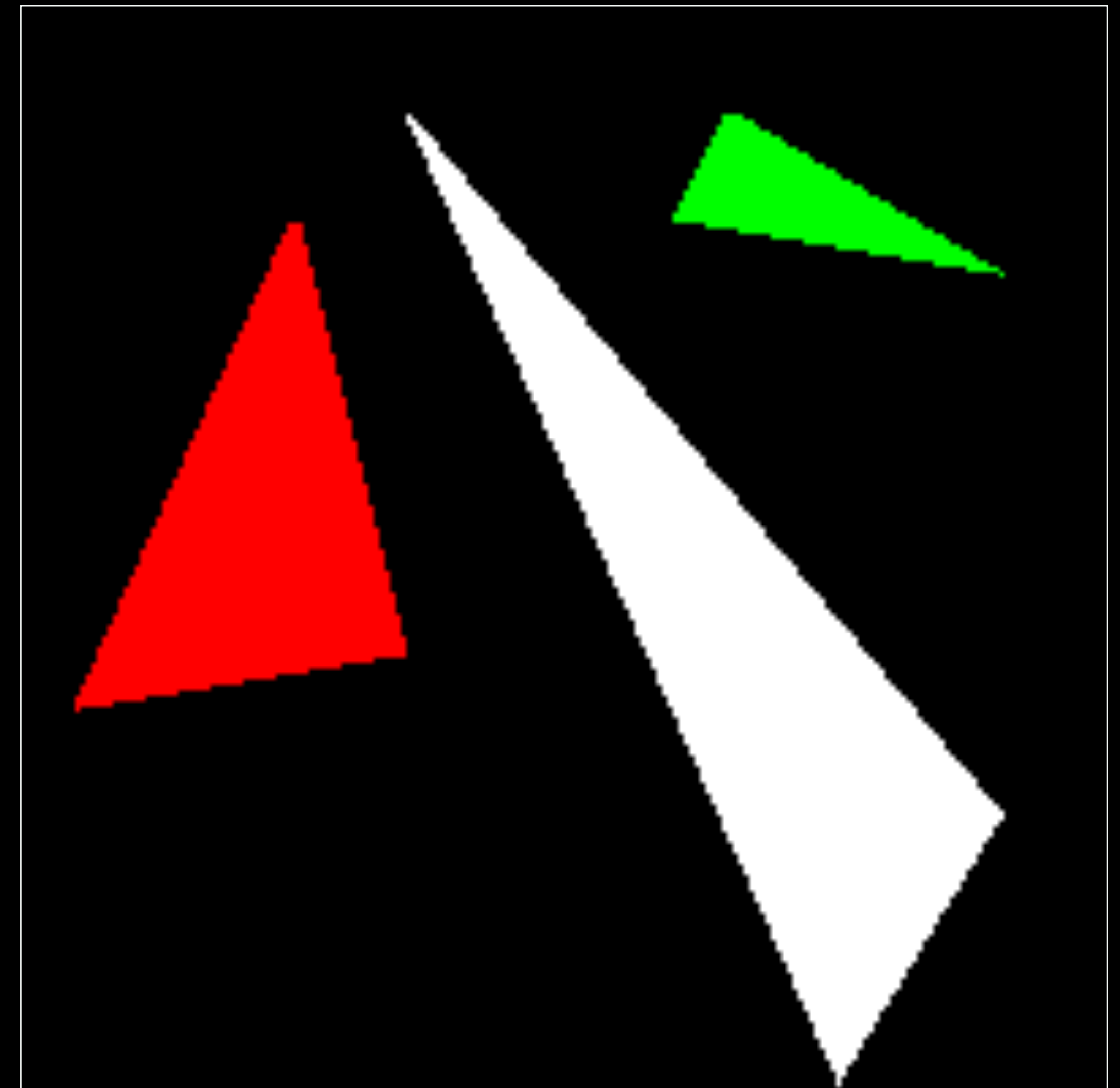
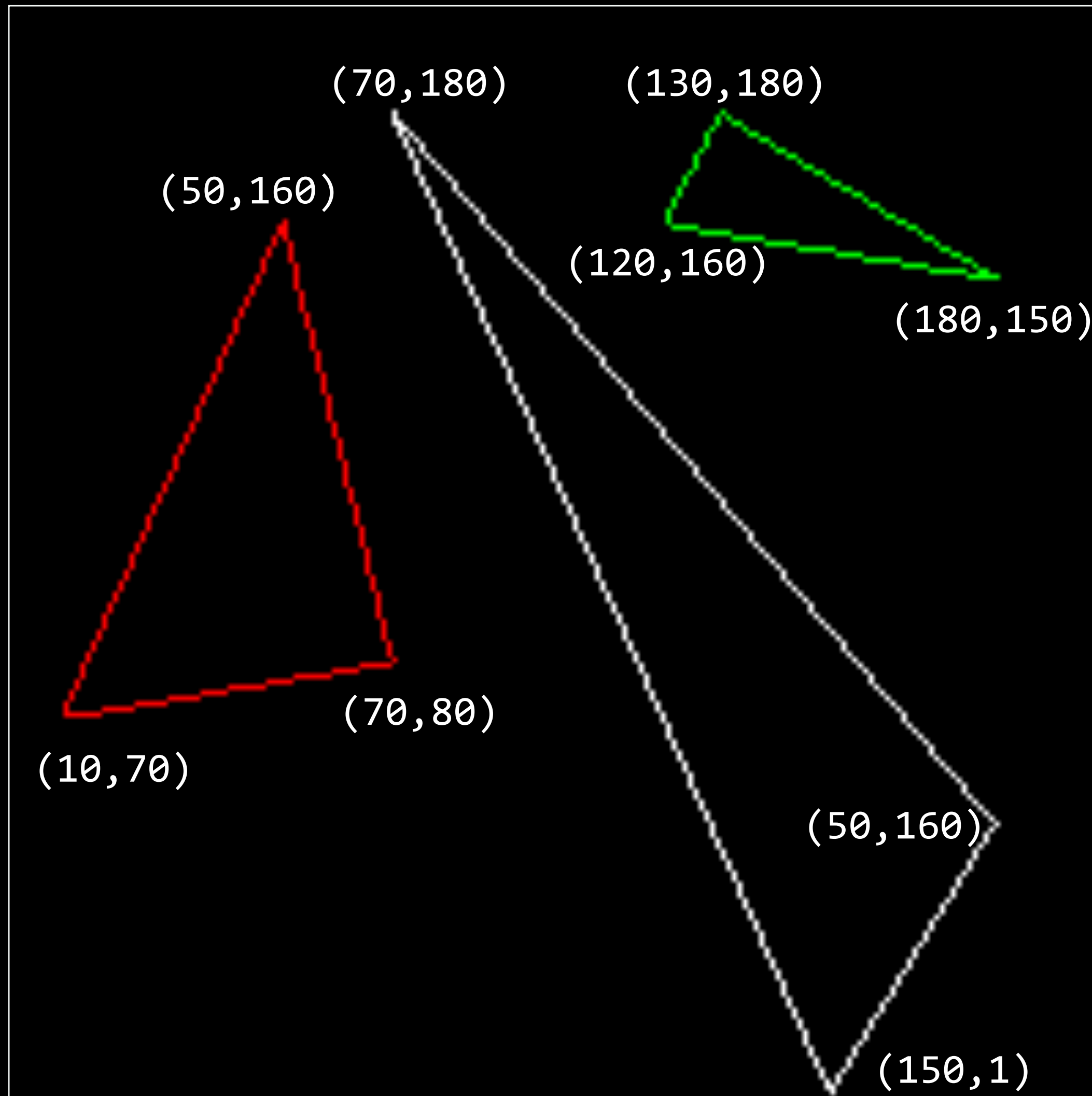
```
int main() {
    TGAImage image(width, height, TGAImage::RGB);

    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);

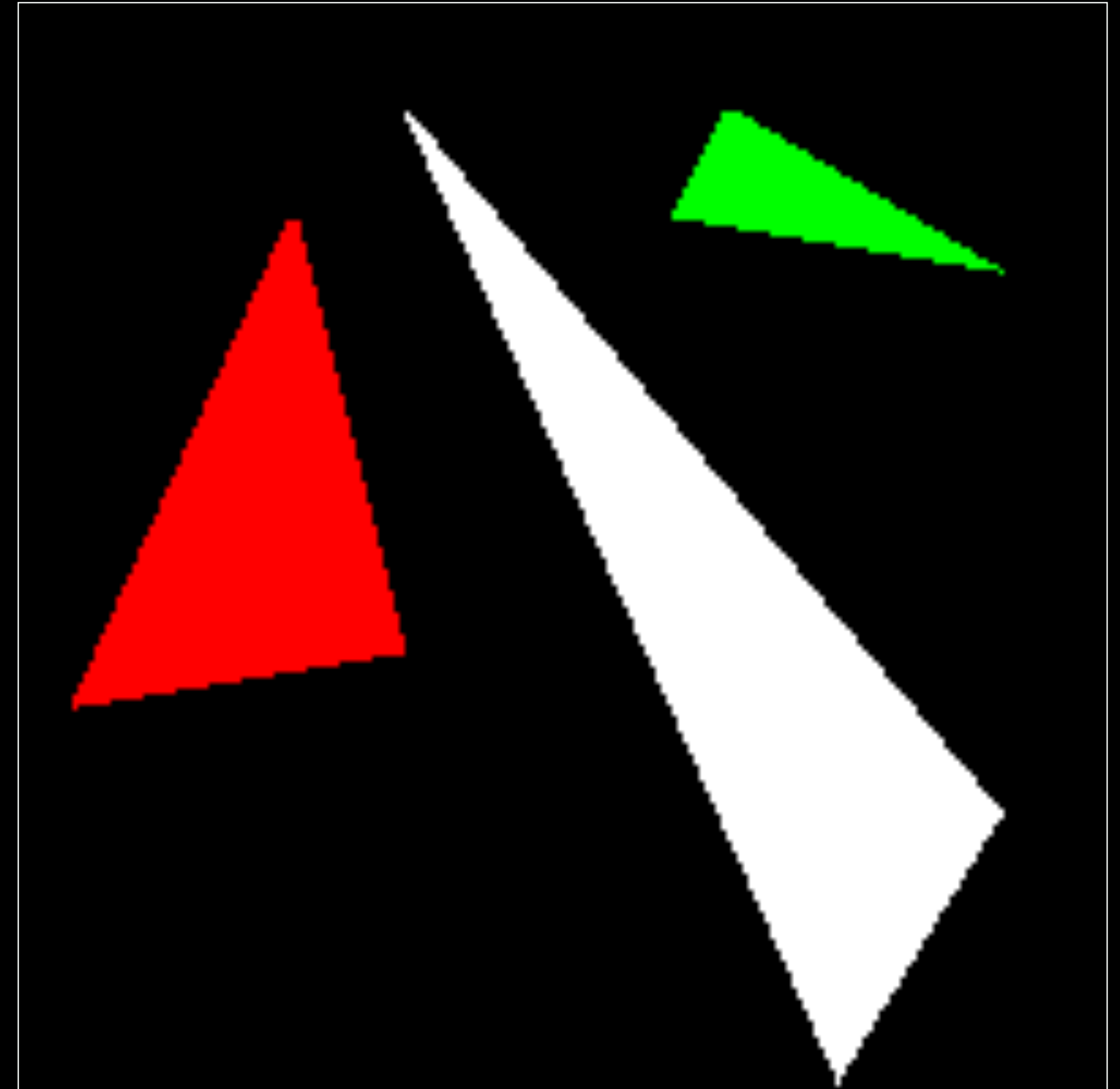
    image.flip_vertically();
    image.write_tga_file("triangle.tga");
    return 0;
}
```


Le résultat espéré



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



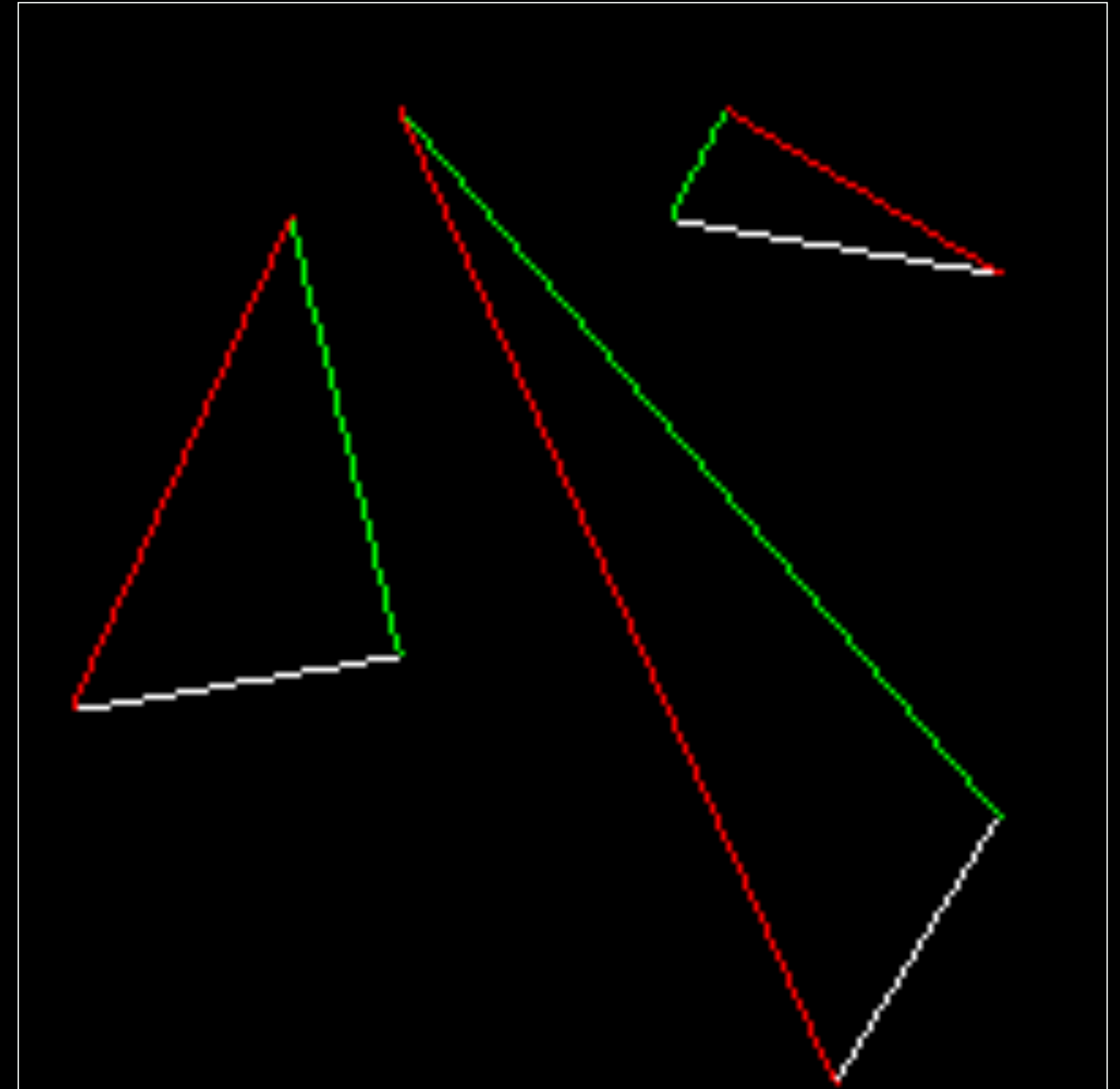
1ère approche: line sweeping

- ♦ 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 ces 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*,
              TGImage &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,
             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]);

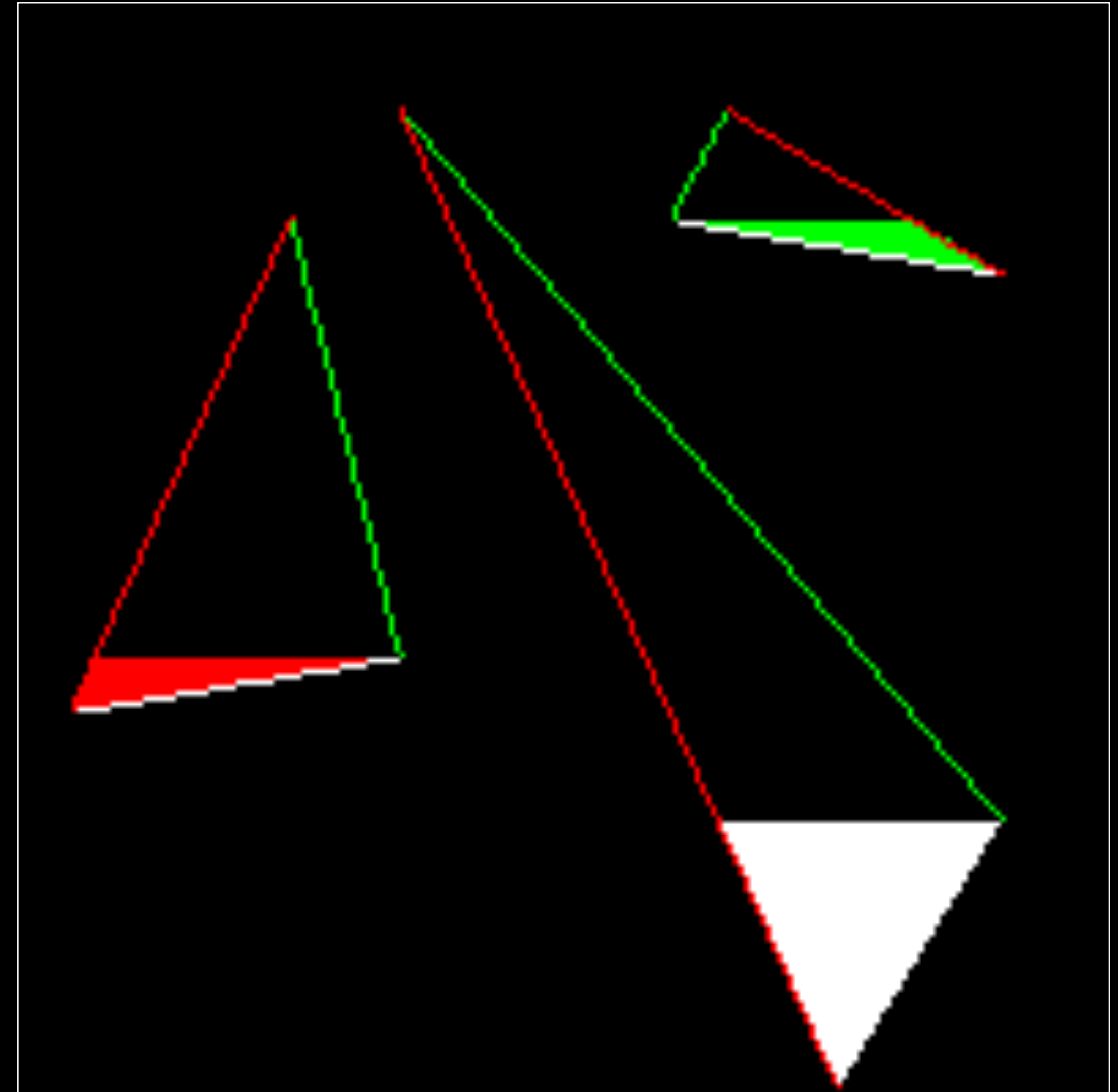
    for(int y = t[0].y; y < t1.y; ++y)
    {
        float t01 = ( y - t[0].y ) / float( t[1].y - t[0].y );
        int x01 = t[0].x + t01 * (t[1].x - t[0].x);

        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,
                TGImage &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);

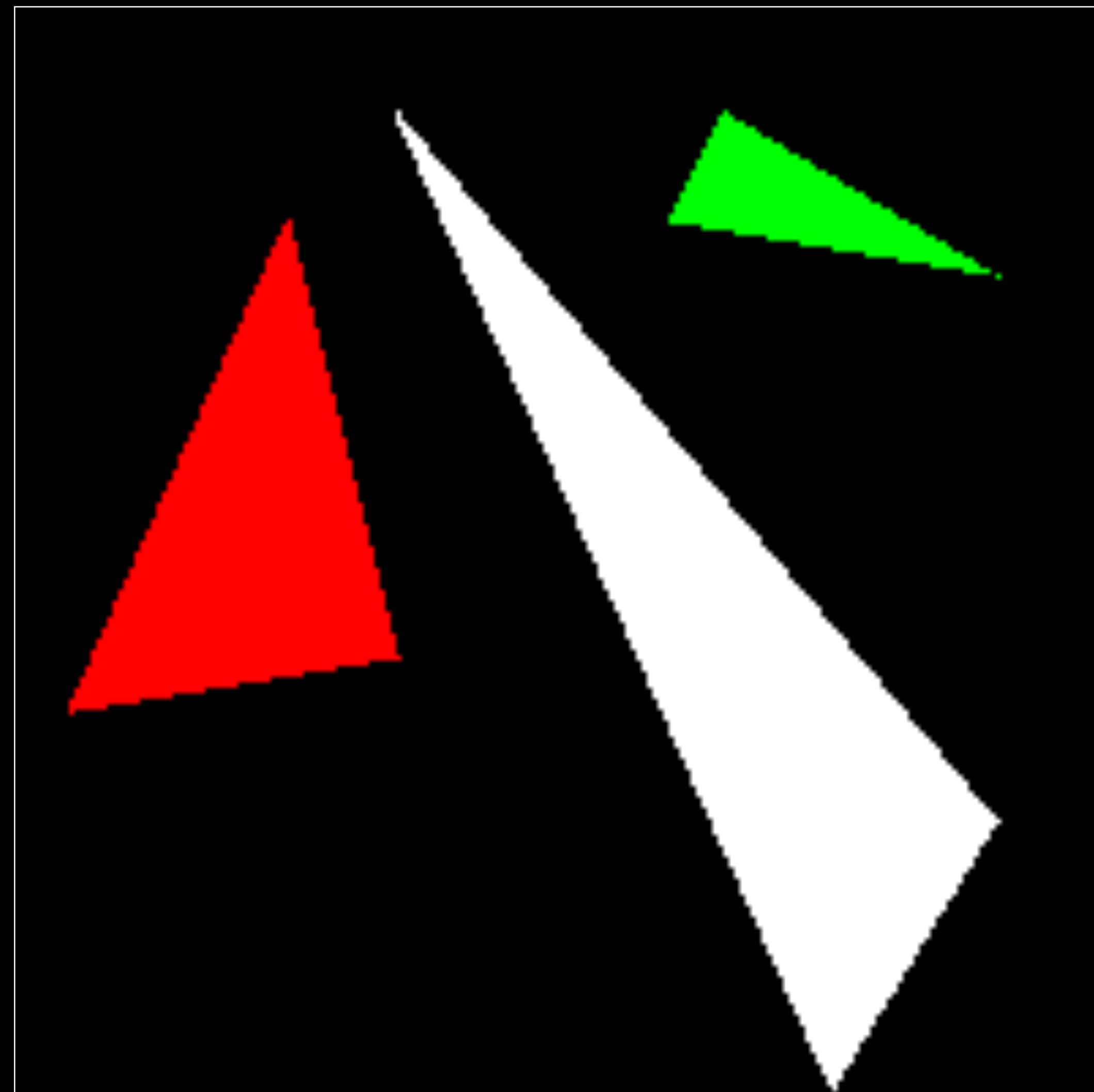
        if(x23 < x01) std::swap(x01,x23);
        for(int x = x01; x <= x23; ++x)
            image.set(x,y,color);
    }
}

void triangle(Vec2i* t,
             TGImage &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



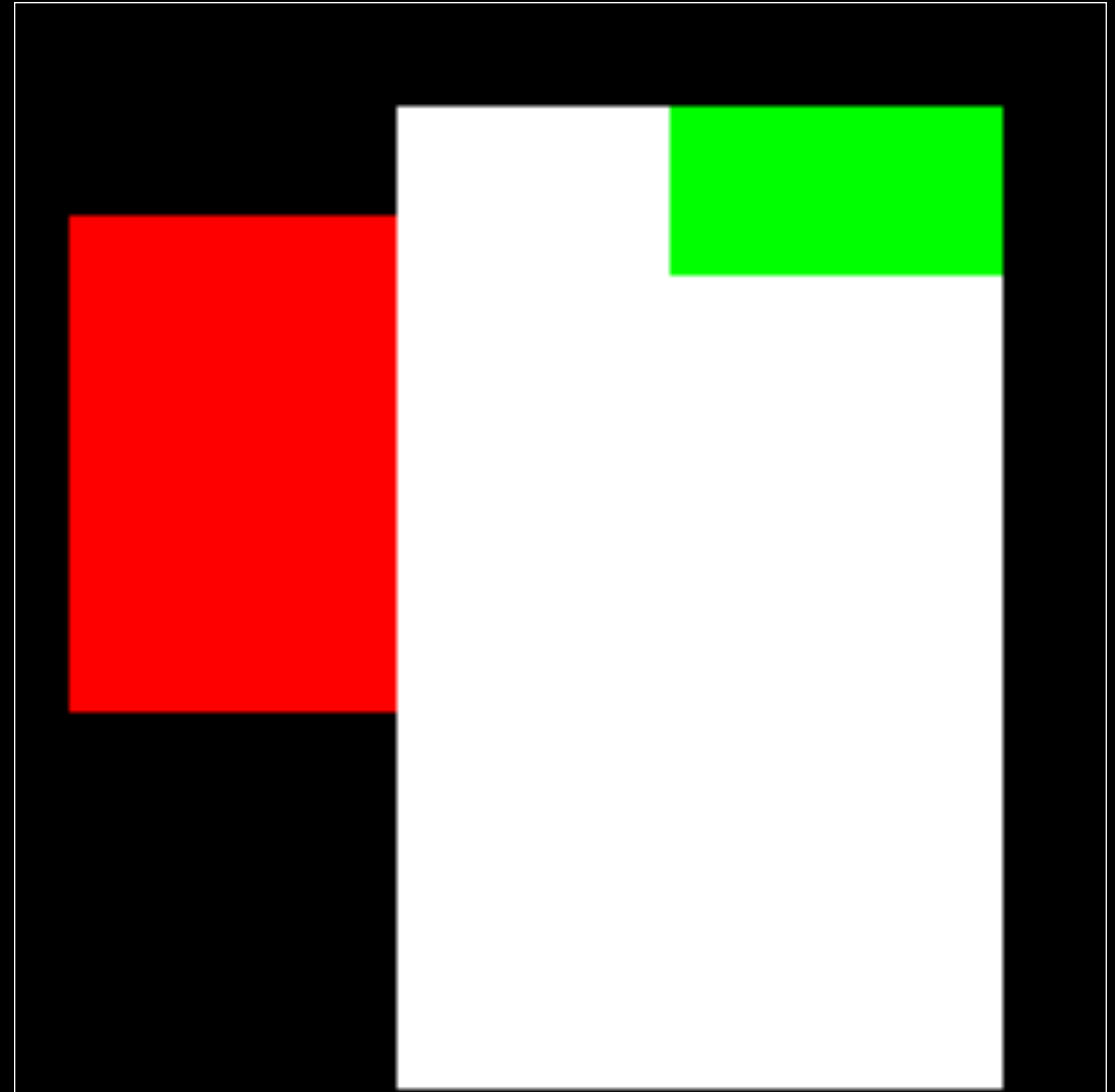
Et si on exécutait ce code sur GPU ?

```
std::array<Vec2i,2> boite_englobante(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_englobante(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);
        }
    }
}
```

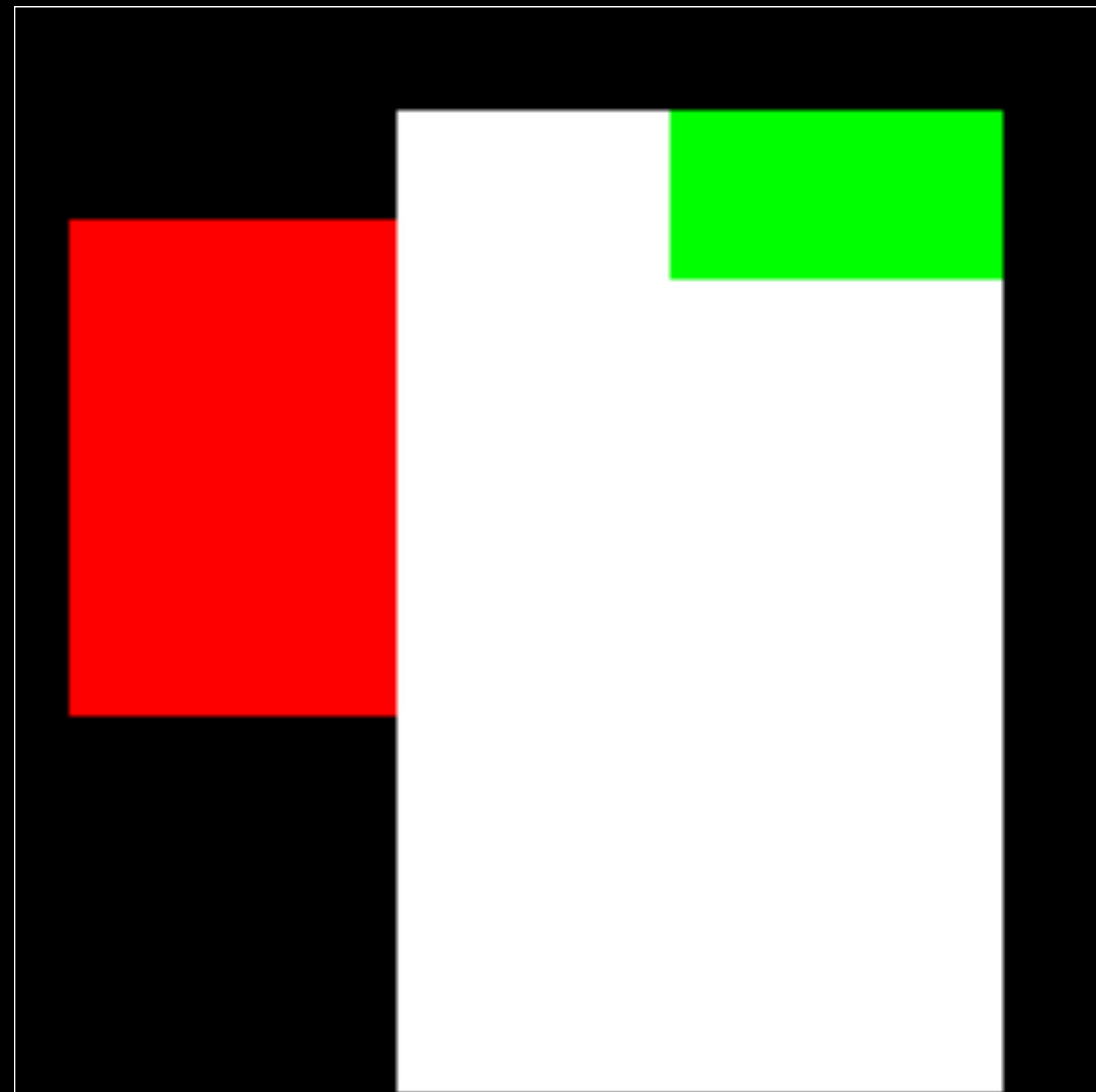
Bounding Box



Bounding Box

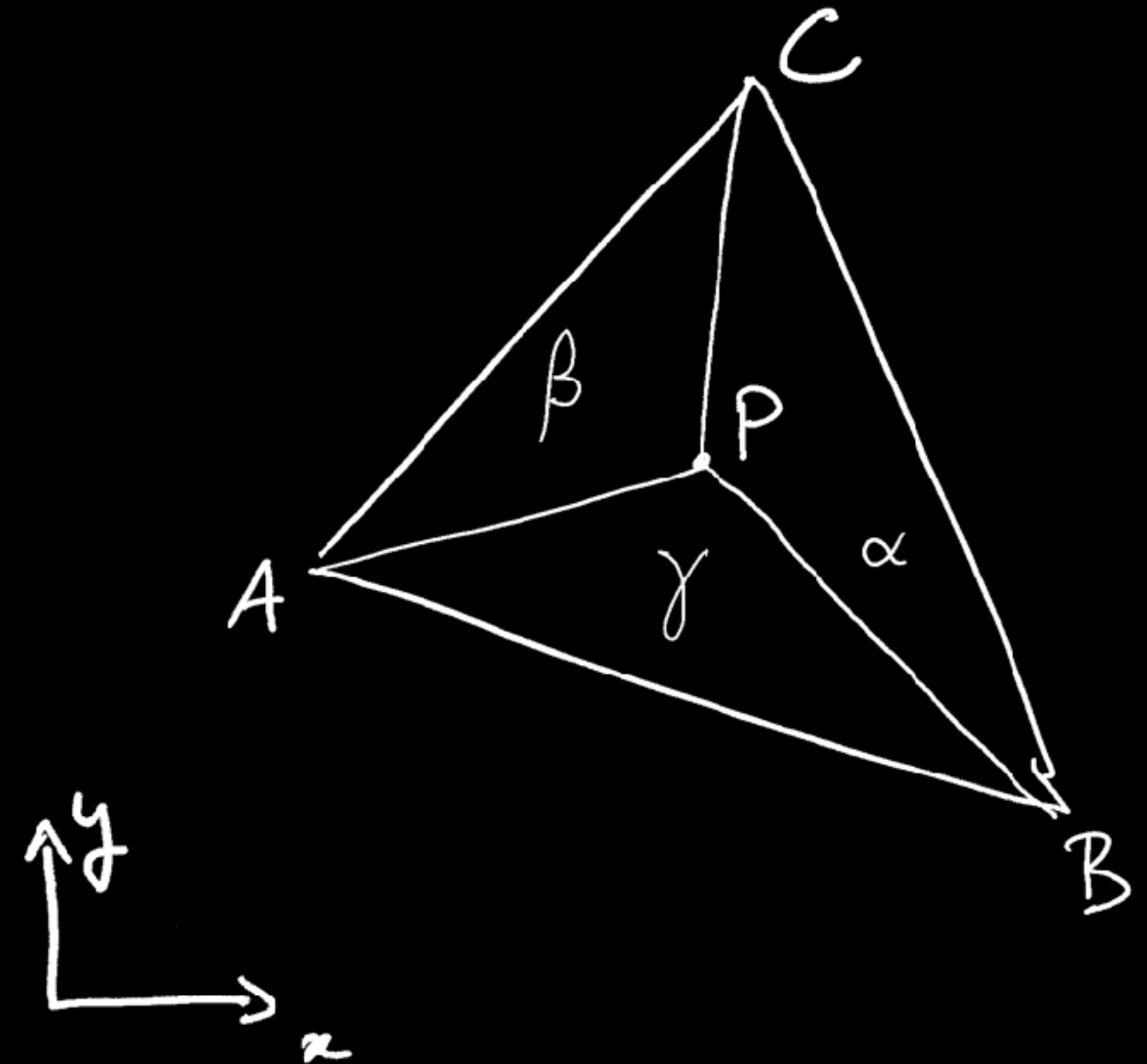
```
std::array<Vec2i,2>
boite_englobante(Vec2i* t, int n = 3)
{
    std::array<Vec2i,2> box { t[0],t[0] };
    for(int i = 1; i < n; ++i) {
        if ( t[i].x < box[0].x ) box[0].x = t[i].x;
        if ( t[i].y < box[0].y ) box[0].y = t[i].y;
        if ( t[i].x > box[1].x ) box[1].x = t[i].x;
    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



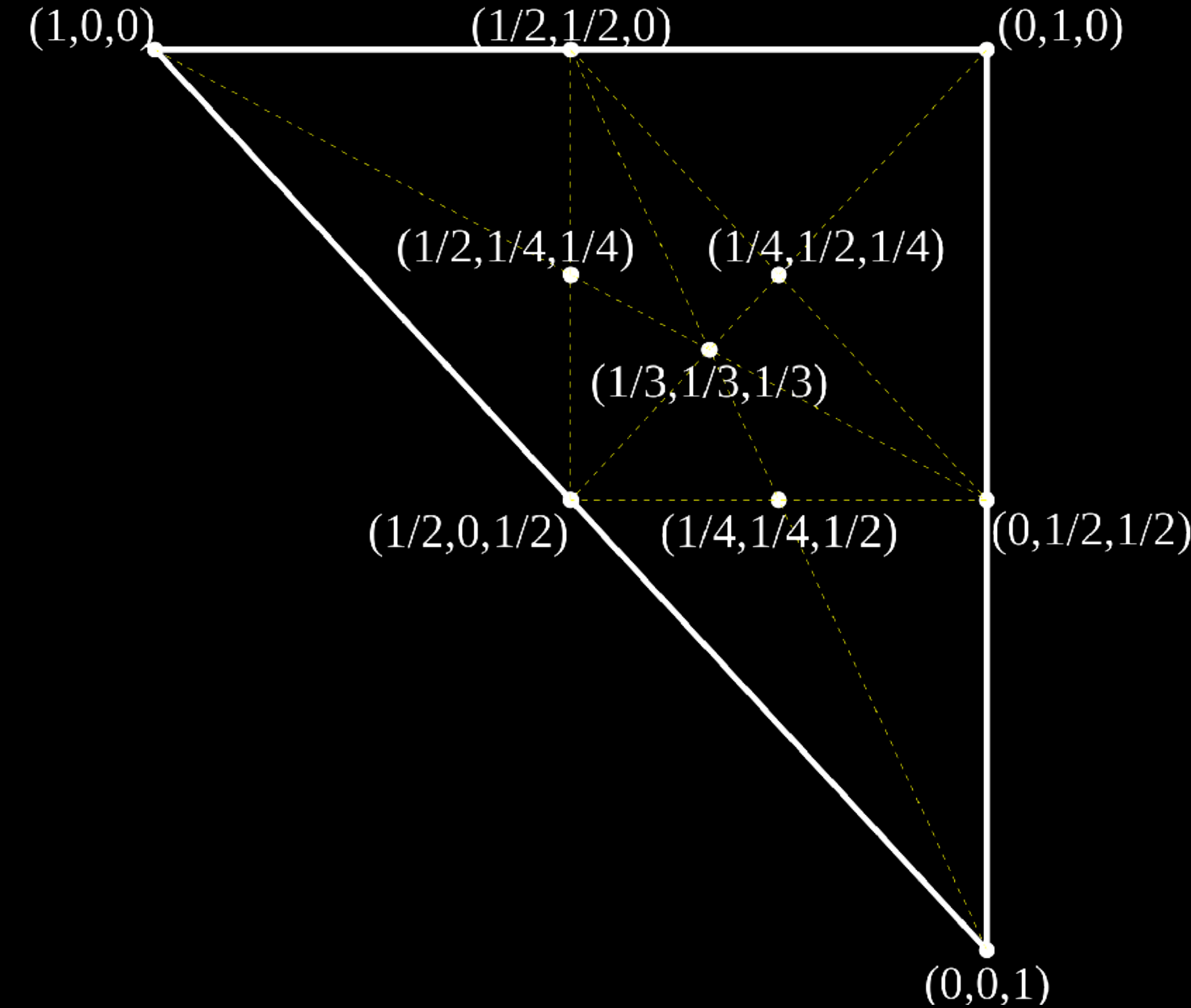
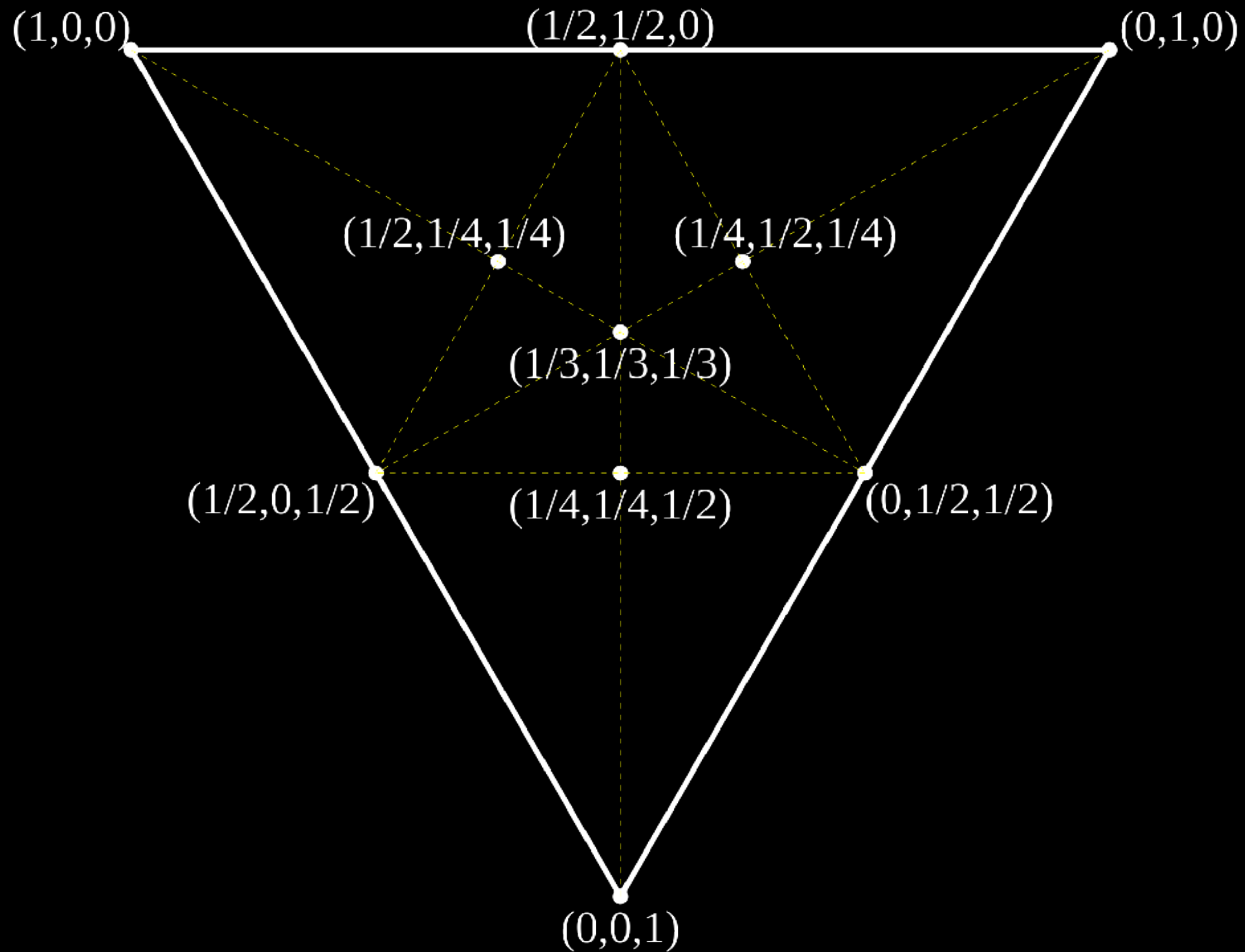
Vérifier si p est à l'intérieur d'un triangle ?

- ♦ 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

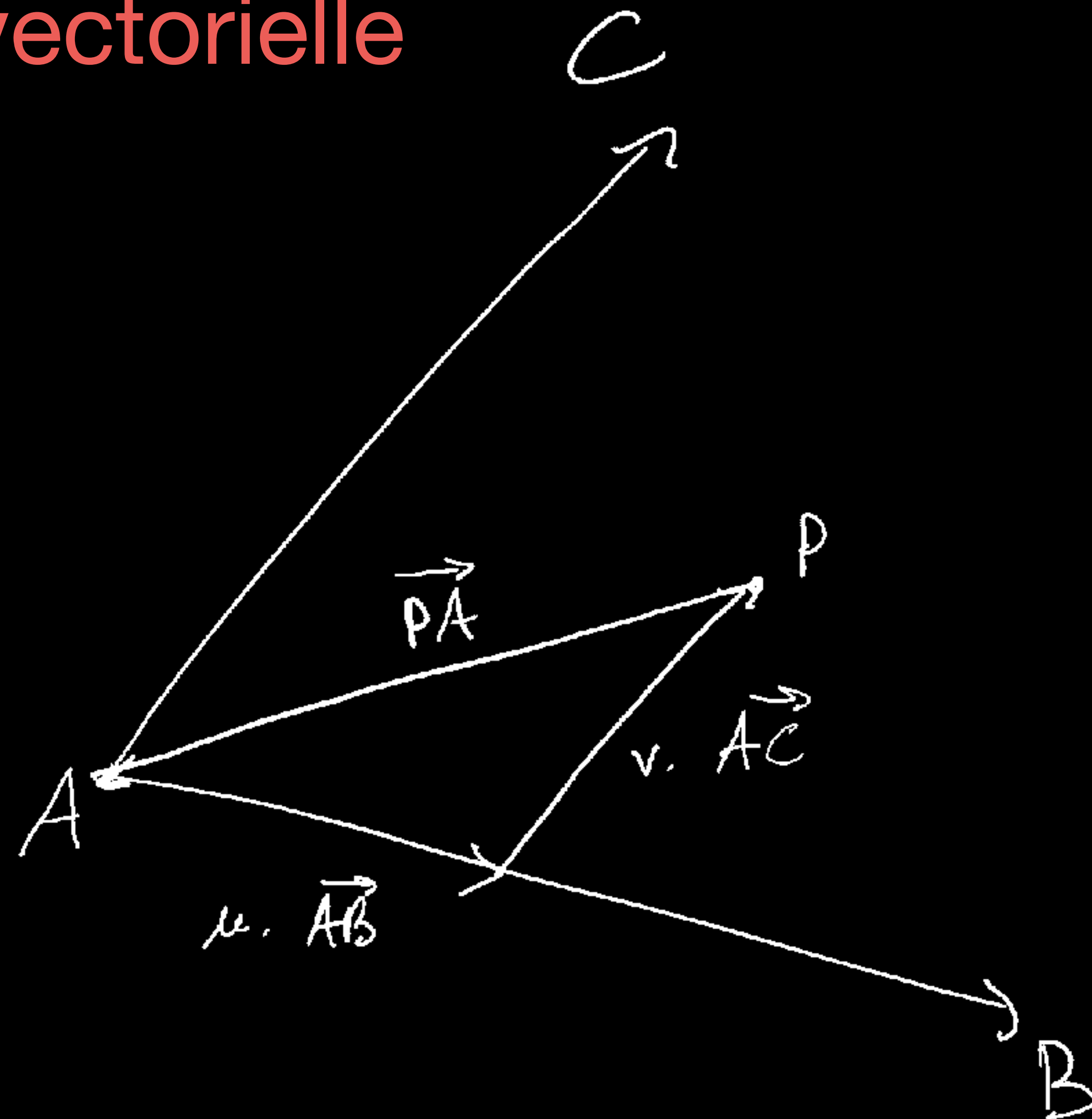


Sous forme vectorielle

$$\vec{P} = \vec{A} + \mu \cdot \vec{AB} + \nu \cdot \vec{AC}$$

$$1 \cdot \vec{A} + \mu \cdot (\vec{B} - \vec{A}) + \nu \cdot (\vec{C} - \vec{A})$$

$$(1 - \mu - \nu) \vec{A} + \mu \vec{B} + \nu \vec{C}$$



$$\mu \cdot \vec{AB} + \nu \cdot \vec{AC} + \vec{PA} = \vec{0}$$

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

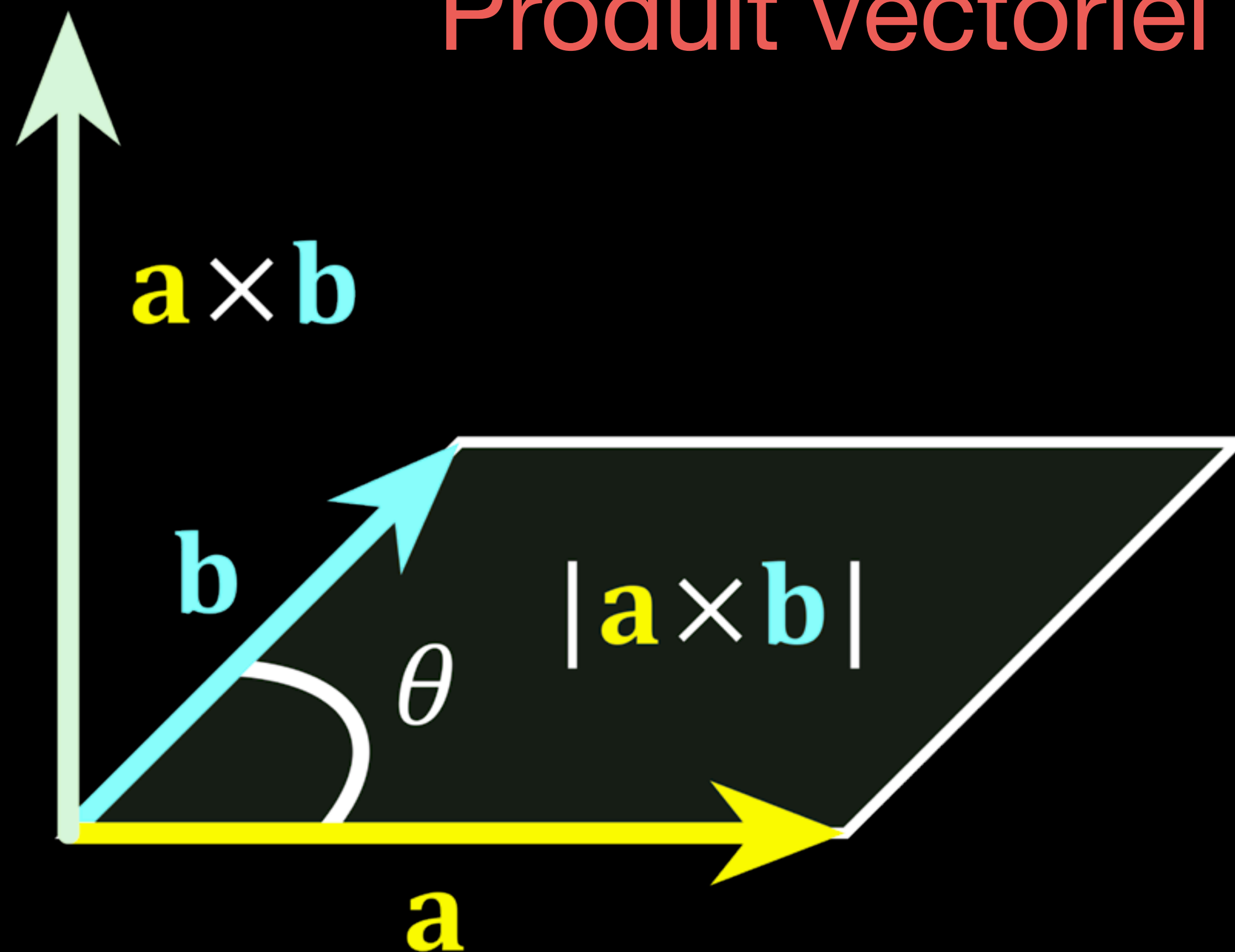
$$u \cdot (B_x - A_x) + v \cdot (C_x - A_x) + 1 \cdot (A_x - P_x) = 0$$

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

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

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

Produit vectoriel

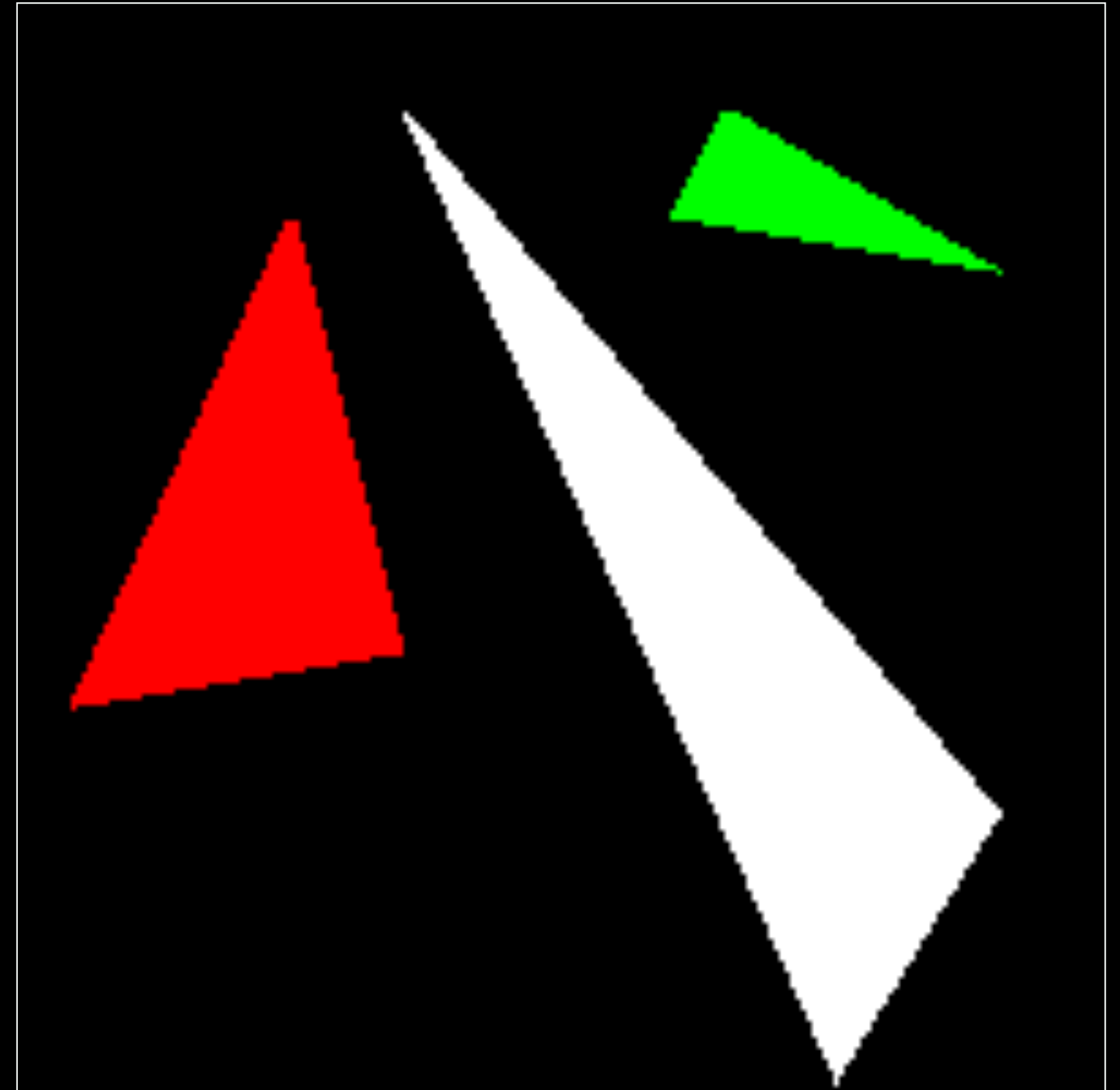


$$\overrightarrow{(u, v, 1)} = \overrightarrow{(B_x - A_x, C_x - A_x, A_x - P_x)} \times \overrightarrow{(B_y - A_y, C_y - A_y, A_y - P_y)}$$

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

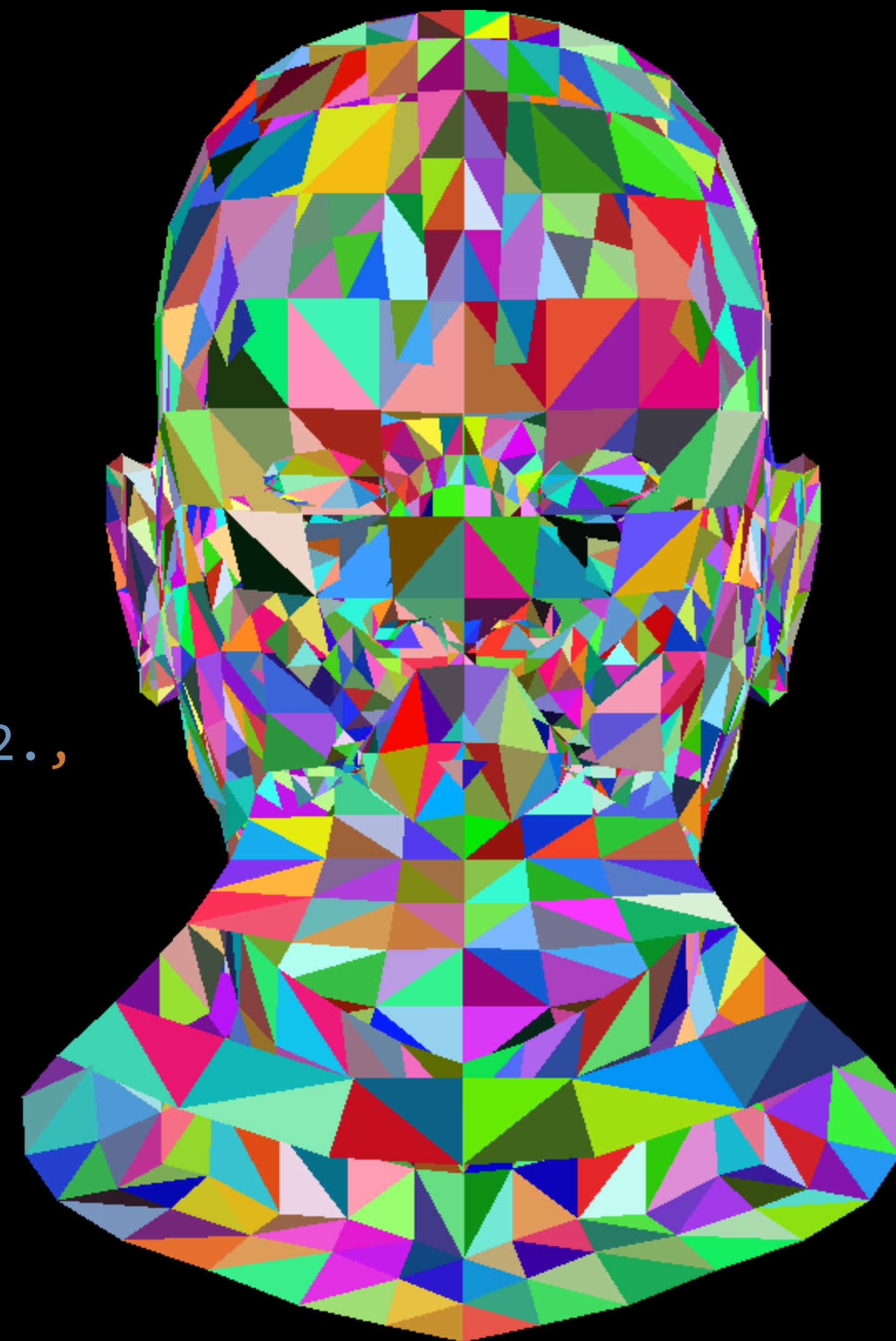
```
TGAImage image(width, height, TGAImage::RGB);

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);
}
```



Ombfrage plat

Principe général de l'ombrage (shading)

heig-vd



Ombrage plat (flat shading)

- ♦ Rendu en niveaux de gris
- ♦ Pour simplifier, on utilise $R = G = B =$ 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 ?

