Par MERIAUX Noël et TORRENS Lucas

Projet
Algorithmes de tracé de fractales en Xlib

« Le plus complexe des objets mathématiques,
l'ensemble de Mandelbrot,
est complexe au point d'en être incontrôlable par l'Homme
et descriptible comme étant le chaos. »
(Benoît Mandelbrot)

« La géométrie fractale vous fera tout percevoir différemment.
En lire davantage est dangereux.
Vous risquez de perdre de votre vision d'enfance des nuages,
forêts, fleurs, galaxies, feuilles, plumes, pierres, montagnes, torrents,
des tapis, des briques, et bien d'autres choses encore.
Votre interprétation de ces choses sera changée à tout jamais. »
(Michael F. Barnsley)

PROJET
ALGORITHMES DE TRACÉ DE FRACTALES EN XLIB

MERIAUX NOËL ET TORRENS LUCAS

Table des matières

1. Introduction

1.1. **Rappel sur la notion de fractales.**

Le terme “fractale” est inventé par Benoît Mandelbrot pour qualifier des objets géométriques aux propriétés particulières. Issus du latin “fractus” signifiant “irrégulier” ou “brisé”, ce mot est utilisé en anglais pour désigner certains nuages.

Les fractales sont des objets généralement connus pour être constitué d'un motif répété à l'infini et ce dans des positions différentes. Mandelbrot se propose de définir plus exactement leur nature à l'aide de l'une de leurs caractérisitiques : la dimension fractale, qui recouvre en fait plusieurs concepts de dimension distincts. Dans tous les cas, elle mesure le “taux d'irrégularité” de la figure : il s'agit dont d'un réel quelconque et non pas nécessairement d'un entier, comme on eût pu s'y attendre.

Si les modèles scientifiques donnent lieu à des représentations géométriques parfaites, il n'en va pas de même dans la nature. En effet, même sans plonger à l'échelle de l'atome, les conventions de tracé d'un contour quelconque rendent incertaine la position d'un hyperplan tangent en un point de la figure. La tangente changerait selon l'échelle considérée, mais existerait à toutes les échelles.

Pour illustrer ceci, Mandelbrot se propose de démontrer qu'une pelote de fil possède de façon latente plusieurs dimensions entières effectives. Sur une échelle de plusieurs mètres, la pelote n'est qu'un point sans dimension, mais sur l'échelle de quelques centimètres, il s'agit d'une boule tridimensionnelle. On peut continuer à plonger pour montrer que la pelote est unidimensionnelle (car constituée de fils), ou à nouveau sans dimension car constituée d'atomes ponctuels.

Les fractales constituent donc, même étant sujettes à ces variations, des zones de transition structurées.

1.2. **Objectifs.**

L'objectif de ce projet consiste à concevoir plusieurs algorithmes de tracé de fractales et ce selon diverses méthodes : de façon purement récursive ou selon un schéma itératif dépendant ou non de la structure déjà construite.

Nous commencerons par étudier un cas simple de fractale naturellement récursive que l'on dérécursifiera avant de complexifier légèrement le principe avec une fractale totalement paramétrée. Après cela, nous nous pencherons sur le cas d'une figure itérative intra-dépendante, avant de construire quelques fameux ensembles de Julia.

2. Une première fractale

Nous allons débuter en douceur avec une fractale composée d'un motif de base placé dans un premier temps au centre de notre fenêtre de travail, et que l'on reproduira par le biais d'homothéties, de translations, voire de rotations.

2.1. **Principe de l'algorithme.**

Ici, le motif de base est un carré que l'on souhaite reproduire en chacun des sommets des derniers carrés tracés, et ainsi obtenir, au bout de trois itérations, la figure suivante :

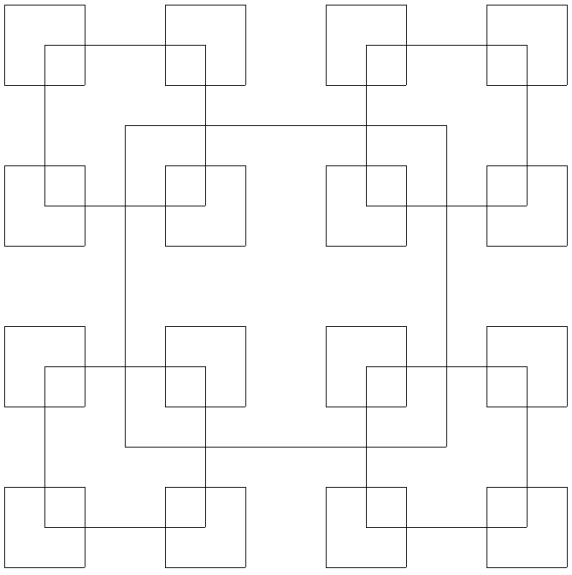

Fig. 1 : fractale récursive ; $n = 3$; $(x, y) = (672, 420)$; $l = 400$

On a opté sur la figure 1 pour une échelle de reproduction égalant 1/2 : on divise le côté par deux à chaque étape, il n'y aura donc pas d'intersections malheureuses entre les figures issues des différents sommets du motif originel (car la série $\sum_{n\in\mathbb{N}^*} \frac{1}{2^n}$ converge vers 1).

La fonction prend comme paramètres la profondeur de récursivité n, les coordonnées x et y du centre de la figure ainsi que le côté l du premier carré.

On suppose pour le moment que l'on dispose d'une fonction de tracé d'un carré, laquelle prend en paramètres les coordonnées du centre du carré et son côté.

Pour obtenir notre fractale, nous commencerons donc par tracer un carré au centre de notre fenêtre de travail, puis en chacun de ses sommets un carré de côté deux fois plus petit. Les coordoonnées des centres de ces carrés sont les suivantes :

$$(x - l/2; y - l/2)$$
$$(x - l/2; y + l/2)$$
$$(x + l/2; y + l/2)$$
$$(x + l/2; y - l/2)$$

2.2. **L'algorithme.**

```
void recursive (int n, int x, int y, int l)
{
   if (n > 0)
   {
      XDrawRectangle(dpy, win, gcontext, x - l / 2, y - l / 2, l, l) ;
      recursive (n - 1, x - l / 2, y - l / 2, l / 2) ;
      recursive (n - 1, x - l / 2, y + l / 2, l / 2) ;
      recursive (n - 1, x + l / 2, y + l / 2, l / 2) ;
      recursive (n - 1, x + l / 2, y - l / 2, l / 2) ;
   }
}
```

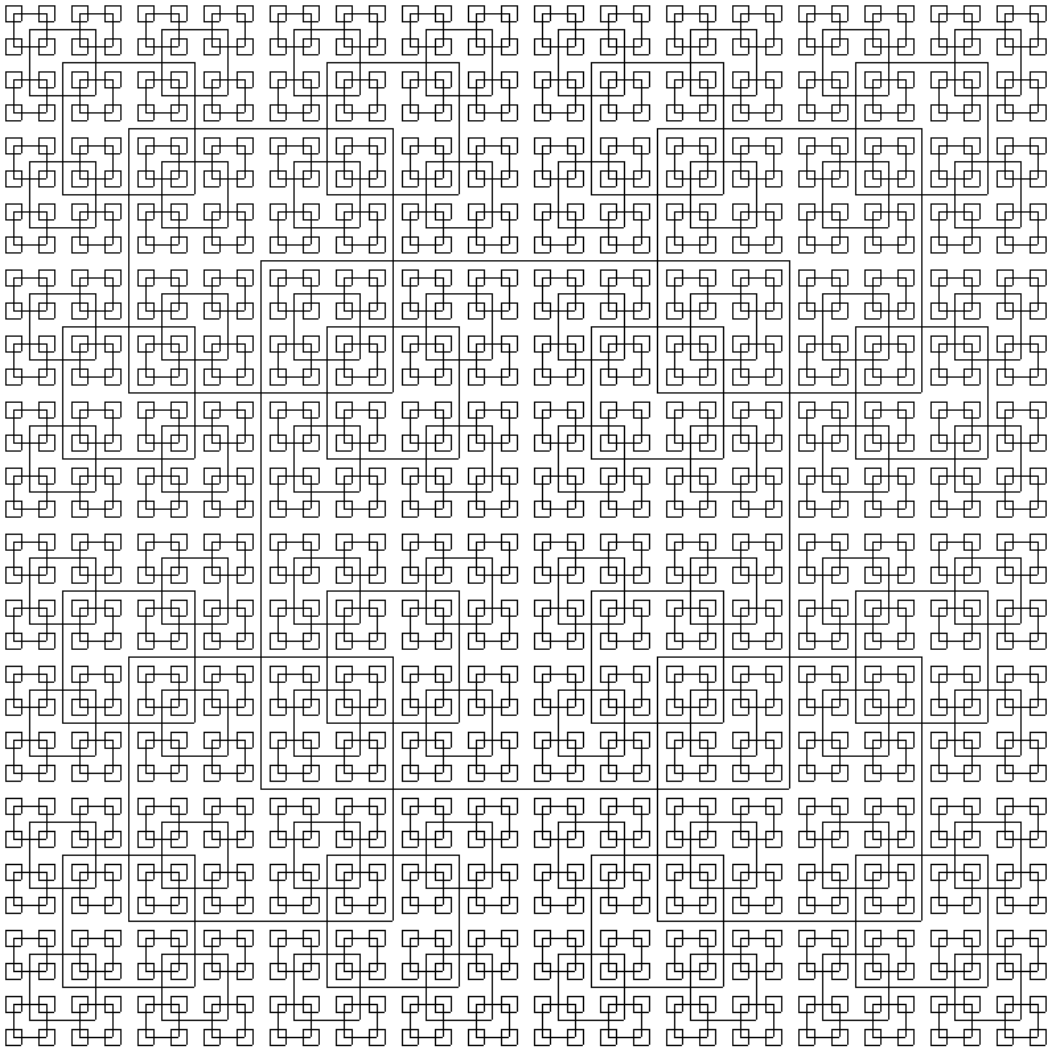

Fig. 2 : fractale récursive ; $n = 6$; $(x, y) = (672, 420)$; $l = 400$

2.3. **Algorithme dérécursifié.**

Pour la dérécursification, on utilise une pile dans laquelle on stocke à chaque étape la profondeur de récursivité courante, les coordonnées des nouveaux centres ainsi que le côté de chaque nouveau carré, donc douze données à chaque étape.

On eût pu se servir d'une file pour cette fonction, eûssions-nous tenu à procéder à l'affichage dans le sens de la profondeur de récursivité. Le principe du nouvel algorithme est simple : notons n une variable contenant initialement l'ordre de récursivité souhaité.

On commence par effectuer une boucle dans laquelle à chaque étape on trace un carré dans le coin supérieur gauche du carré précédent (et de côté deux fois plus petit) et on empile les données nécessaires au tracés des carrés depuis les trois autres coins, puis l'on décrémente n. Lorsque n devient nul, et si la pile n'est pas vide, on effectue un dépilement des donnés, et on reprend au début de la première boucle ; sinon, l'algorithme s'arête.

```
 void fracIter (int n, int x, int y, int l)
{

int stop = 0, err = 0 ;
Pile_t* p = creer_pile (12 * n) ;
int xCour = x, yCour = y, lCour = l, nCour = n ;

   while ( !stop)
   {
      while (nCour > 0)
      {

         XDrawRectangle(dpy, win, gcontext, xCour - lCour / 2, yCour
- lCour / 2, lCour, lCour) ;
         nCour-- ;

         if (nCour)
         {
            lCour /= 2 ;
            empiler_pile (p, nCour) ;
            empiler_pile (p, lCour) ;
            empiler_pile (p, xCour + lCour) ;
```

```
            empiler_pile (p, yCour - lCour) ;
            empiler_pile (p, nCour) ;
            empiler_pile (p, lCour) ;
            empiler_pile (p, xCour + lCour) ;
            empiler_pile (p, yCour + lCour) ;
            empiler_pile (p, nCour) ;
            empiler_pile (p, lCour) ;
            empiler_pile (p, xCour - lCour) ;
            empiler_pile (p, yCour + lCour) ;
            xCour = xCour - lCour ;
            yCour = yCour - lCour ;
        }
    }

    if ( !est_vide (p))
    {
        yCour = depiler_pile (p, &err) ;
        xCour = depiler_pile (p, &err) ;
        lCour = depiler_pile (p, &err) ;
        nCour = depiler_pile (p, &err) ;
    }
    else
        stop = 1 ;
  }

  supprimer_pile (p) ;
}
```

3. Paramétrisation de la récursivité

Dans l'exemple précédent, on a écrit une fonction dont l'unique but consiste à tracer des motifs d'une manière bien précise. Nous allons ici nous intéresser à une autre fonction récursive, laquelle est toutefois intégralement paramétrée, et permettrait donc, avec de mineures modifications, de tracer de multiples fractales.

3.1. **Principe de l'algorithme.**

Dans cet exemple, le motif à reproduire sera un simple segment, même si l'on peut se servir de toute fonction de tracé d'un motif donné à une échelle passée en paramètre. Il est par exemple possible d'obtenir, avec un paramétrage spécifique, d'obtenir la figure suivante en trois itérations :

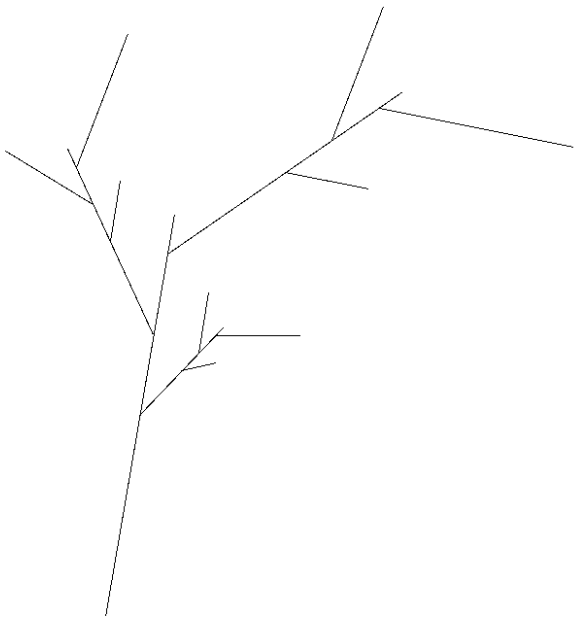

Fig. 3 : arbre paramétré ; $n = 3$; $(xd, yd) = (448, 840)$; $l = 420$; $(dd1, dg, dd2) = (0.9, 0.7, 0.5)$; $adep = 1.4rad$; $ad1 = 0.8rad$; $ag = 0.6rad$; $ad2 = 0.6rad$; $(pd1, pg, pd2) = (0.7, 0.5, 0.3)$

Dans cet exemple, on reproduit le motif de départ trois fois, et ce avec un paramétrage entièrement passé en paramètres.

La fonction prend en paramètre le niveau n de récursivité que l'on souhaite atteindre, les coordonnées xd et yd de tracé du premier motif, ainsi que trois paramètres par nouvelle occurrence du motif : l'angle, la distance et la proportion de reproduction.

Nous reproduirons dans cet exemple le motif trois fois à chaque appel de la fonction ; conséquemment, nous aurons besoin de paramètre que nous nommerons (respectivement à la précédente énumération) $ad1$, ag, $ad2$, $dd1$, dg, $dd2$, $pd1$, pg, $pd2$, de telle sorte qu'en usant d'angles entre 0 et $\pi/2$ radians, les motifs correspondant à (ag, dg, pg) soient reproduits à gauche du motif précédent et les autres à droite.

Puisque l'on ne trace que des segments dans nos exemples, cette fonction prend également en paramètre la longueur l du premier trait ainsi que l'angle $adep$ qu'il fait avec la verticale.

Le principe de l'algorithme reste en soi le même que précédemment : on commence par tracer une première fois le motif, puis, tant que l'on n'a pas atteint le niveau de récursivité souhaité, on poursuit les reproductions en appelant la fonction sur les coordonnées déduites des paramètres donnés (angles, distances, proportions ; l'angle et la distance pouvant être perçues comme des coordonnées cartésiennes par rapport à l'axe de tracé du motif générateur du motif courant).

Le terme de "distance" de reproduction est d'ailleurs légèrement abusif car, étant donné que celle-ci doit changer avec la profondeur de récursivité, on l'exprime dans notre programme comme proportion de la longueur du motif précédent.

3.2. **L'algorithme.**

Le but visé ici est de tracer un arbre en poursuivant les itérations de la figure 3.

```
void arbre (int n, int xd, int yd, int l, float dd1, float dg, float dd2,
float adep , float ad1 , float ag , float ad2 , float pd1 , float pg , float
pd2)
  {

    int x = xd , y = yd ;

    if (n > 0)
    {
        x = xd + l*cos(adep) ;
        y = yd - l*sin(adep) ;
        XDrawLine (dpy, win, gcontext, xd, yd, x, y) ;

        arbre (n-1 , xd + (l*dd1)*cos(adep) , yd - (l*dd1)*sin(adep)
, l*pd1 , dd1 , dg , dd2 , adep - ad1 , ad1 , ag , ad1 , pd1 , pg , pd2) ;
        arbre (n-1 , xd + (l*dd2)*cos(adep) , yd - (l*dd2)*sin(adep)
, l*pg , dd1 , dg , dd2 , adep + ag , ad1 , ag , ad2 , pd1 , pg , pd2) ;
        arbre (n-1 , xd + (l*dd3)*cos(adep) , yd - (l*dd3)*sin(adep)
, l*pd2 , dd1 , dg , dd2 , adep - ad2 , ad1 , ag , ad2 , pd1 , pg , pd2) ;
    }
  }
```

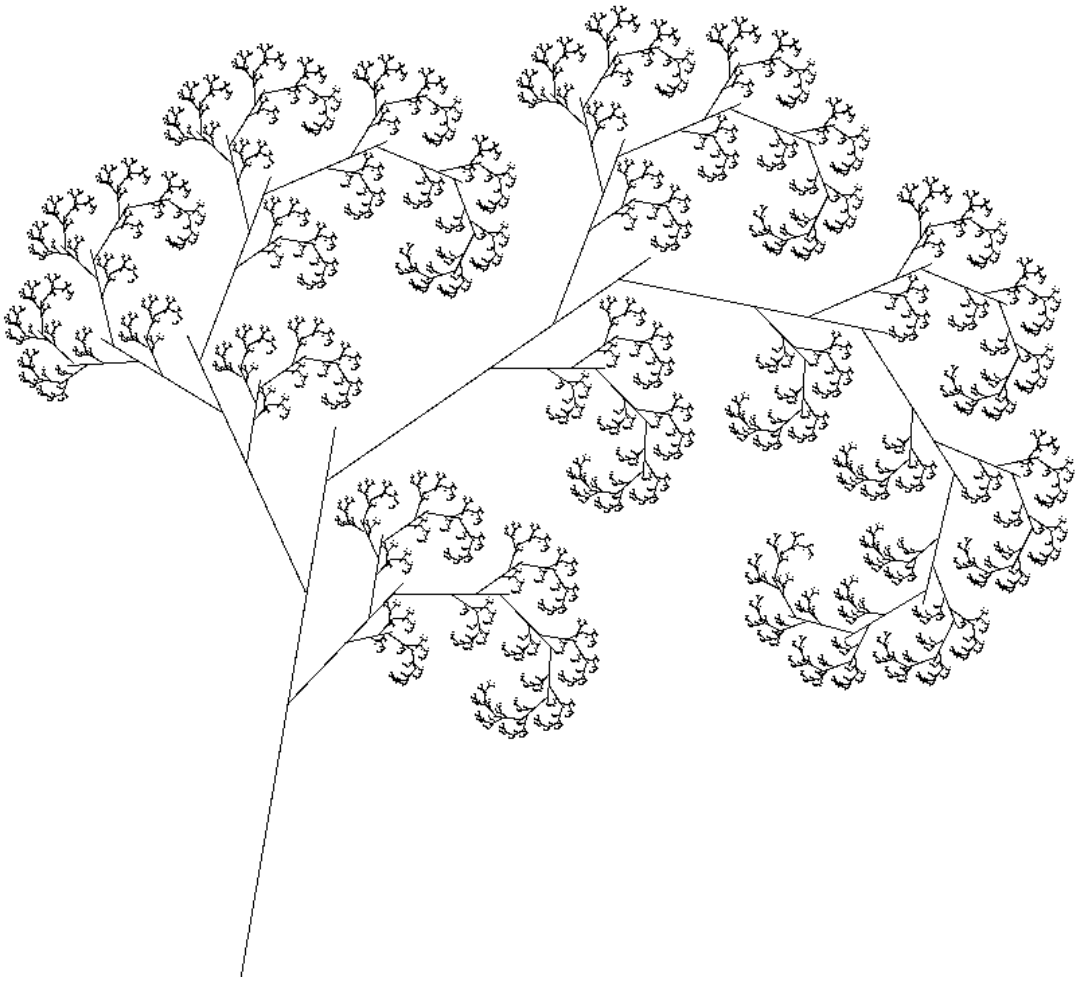

Fig. 4 :
Arbre paramétré, $n = 13$,
$(dd1, dg, dd2) = (0.9, 0.7, 0.5)$,
$(ad1, ag, ad2) = (0.8$ rad, 0.6 rad, 0.6 rad$)$
$(pd1, pg, pd2) = (0.7, 0.5, 0.3)$.

4. Tracé itératif

Après avoir tracé des fractales récursives, il semble naturel de se tourner vers un algorithme itératif. Nous allons ici construire une figure ligne par ligne, chaque ligne dépendant de la précédente, selon certaines règles qui peuvent nous mener à des variations du triangle de Sierpinski.

4.1. **Principe de l'algorithme.**

L'algorithme en lui-même est simple. Il faut d'abord définir la règle qui donnera à chaque étape le nouveau vecteur en fonction du précédent, ces vecteurs se composant de nombres correspondant à des couleurs. Ici, on se contentera d'associer 0 au noir et 1 au blanc.
Après cela, on fait une simple boucle dans laquelle on trace le vecteur courant, puis l'on calcule le vecteur suivant à patrir des règles mises en place.
Point important, la première colonne est toujours considérée comme intégralement blanche.

4.2. **L'algorithme.**

```
int regle_complete (int* v, int i)
{
   int res;
   if (v[i - 1])
   {
      if (v[i])
         res = v[i+1] ? 0 : 0;
      else
         res = v[i+1] ? 1 : 1;
   }
   else {
      if (v[i])
         res = v[i+1] ? 1 : 1;
      else
         res = v[i+1] ? 0 : 0;
   }
   return res; }
```

```
void suivant (int* v1, int* v2, int n)
{
   int i;

   v2[0] = 0;
   for (i = 1; i < n - 2; i++)
      v2[i] = regle_complete (v1, i);
   v2[n - 1] = 0;
}

void iterative (int* vInit, int hauteur)
{
   int* vCour = (int*) malloc ((size_X + 2) * sizeof (int));
   int* vSuiv = (int*) malloc ((size_X + 2) * sizeof (int));
   int k, l;

   for (k = 0; k < size_X + 1; k++)
      vCour[k] = vInit[k];


   for (k = 0; k < hauteur; k++)
   {
      for (l = 0; l < size_X - 1; l++)
      {
         if (vCour[l + 1])
            XDrawPoint (dpy, win, gcontext, l, k);
      }
      suivant (vCour, vSuiv, size_X + 1);

      for (l = 0; l < size_X + 1; l++)
         vCour[l] = vSuiv[l];
   }

   free (vCour);
   free (vSuiv);
}
```

La première de ces fonctions est celle qui donne la règle d'obtention de la $i$ème composante du vecteur suivant à partir du vecteur v précédent.

La deuxième permet de calculer le vecteur suivant $v2$ à partir de $v1$, connaissant leur taille n, conformément à ces règles et la troisième effectue le tracé grâce aux deux précédentes, en prenant comme paramètres un vecteur initial $vInit$ et la hauteur *hauteur* de la figure souhaitée.

Si les tests peuvent paraître étonnament compliqués dans la fonction *regle_complete* — on peut très résumer toute la fonction en une courte ligne — c'est pour faciliter le passage d'une figure à l'autre en ne changeant que les valeurs de retour dans chacun des tests. Cela est mis en application un peu plus tard dans la section 4.3

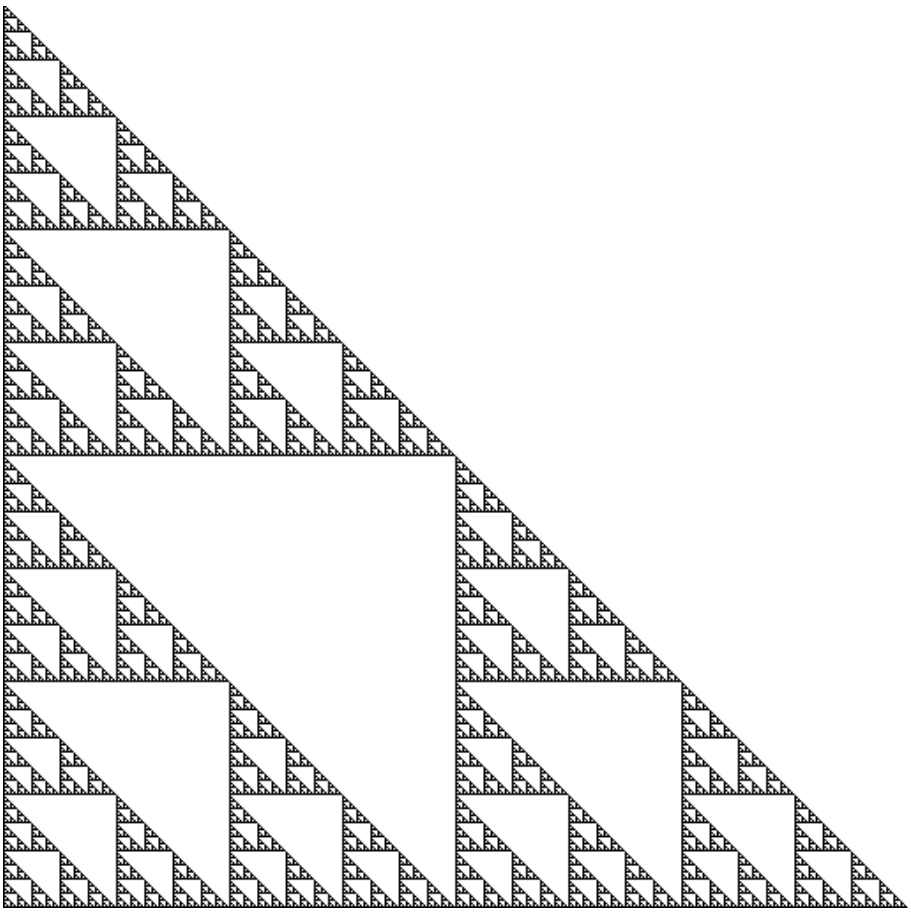

Fig. 5 : Triangle de Sierpinski,
$vInit = (0, 0, ...0, 1, 0, 0)$; $hauteur = 1344$

4.3. **Autres figures.**

Avec ce même algorithme et des règles légèrement distinctes on peut obtenir d'intéressantes variations dans la figure. En voici deux exemples.

La première règle que nous utiliserons est la suivante :

```
int regle_complete (int* v, int i)
{
   int res;
   if (v[i - 1])
   {
      if (v[i])
         res = v[i+1] ? 1 : 0;
      else
         res = v[i+1] ? 0 : 1;
   }
   else
   {
      if (v[i])
         res = v[i+1] ? 0 : 1;
      else
         res = v[i+1] ? 1 : 0;
   }
   return res;
}
```

On n'a ici changé que les résultats attenant au test sur v[i] dans le cas ou v[i-1] n'est pas nul. On obtient alors la figure 6.

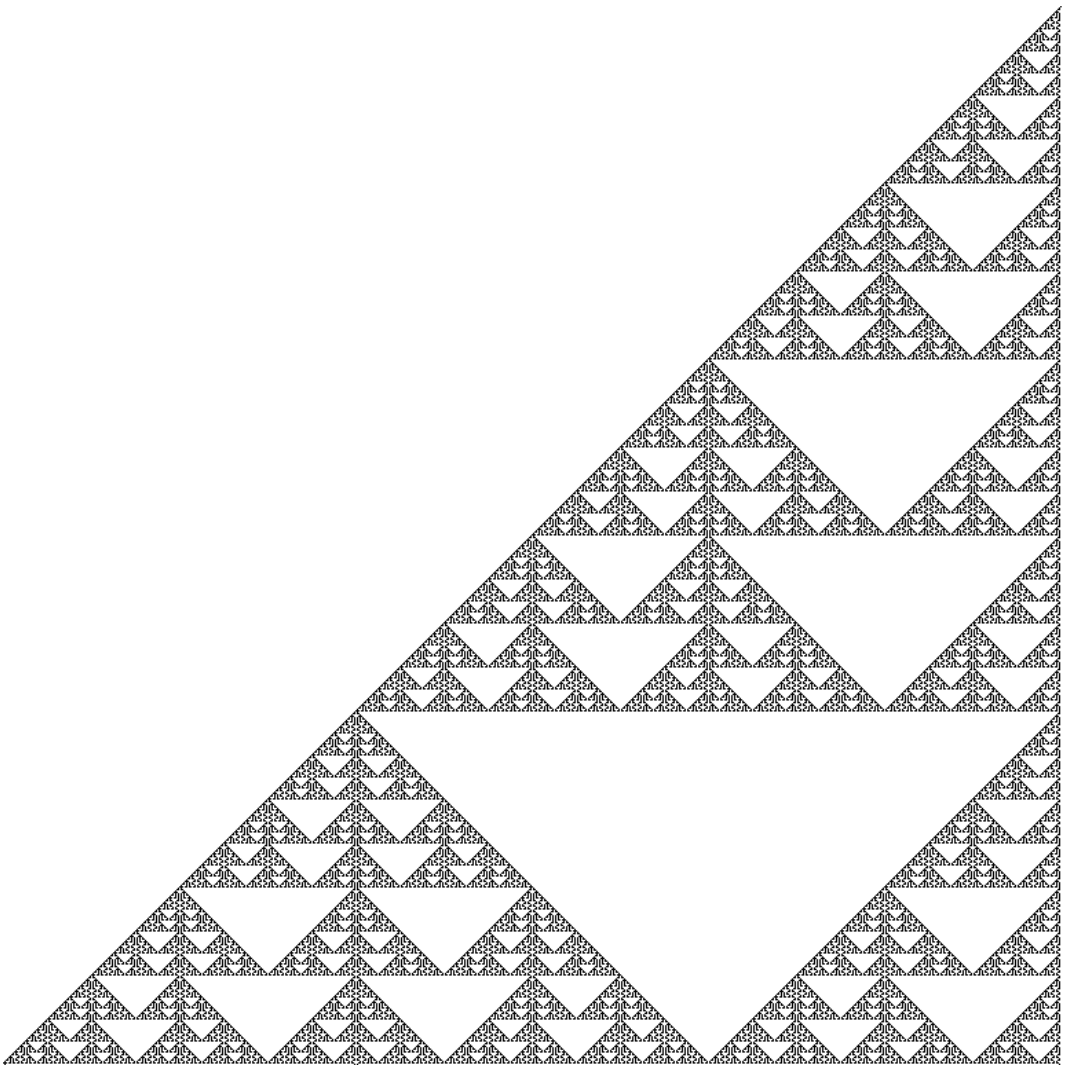

Fig. 6 : Variante du Triangle de Sierpinski,
$vInit = (0, 0, ...0, 1, 0, 0)$; $hauteur = 1344$

La seconde règle que nous utiliserons est la suivante :

```
int regle_complete (int* v, int i)
{
   int res;
   if (v[i - 1])
   {
      if (v[i])
         res = v[i+1] ? 1 : 0;
      else
         res = v[i+1] ? 1 : 1;
   }
   else
   {
      if (v[i])
         res = v[i+1] ? 0 : 1;
      else
         res = v[i+1] ? 1 : 0;
   }
   return res;
}
```

On n'a ici changé que les résultats attenant au test sur v[i] dans le cas ou v[i-1] n'est pas nul. On obtient alors la figure 7.

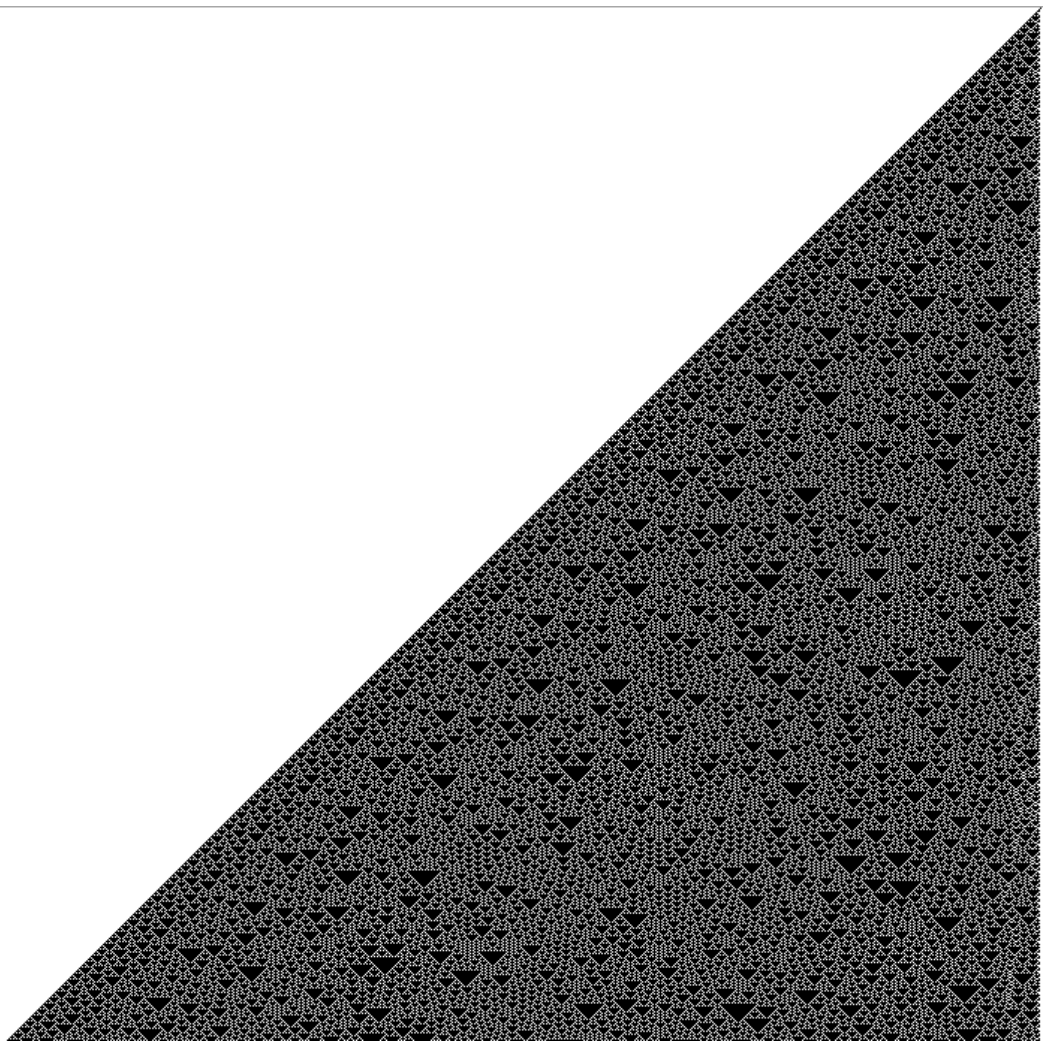

Fig. 7 : Variante du Triangle de Sierpinski,
$vInit = (0, 0, ...0, 1, 0, 0)$; $hauteur = 1344$

5. Ensemble de Julia

Un autre type de tracé fractal consiste à se baser sur un objet mathématique bien spécifique : on prend ici l'exemple de l'ensemble de Julia, lequel est l'ensemble des nombres complexes z_0 tels que la suite $(z_n)_{n\in\mathbb{N}},\ z_{n+1} = z_n + c,\ c \in \mathbb{C}$ soit convergente.

5.1. **Principe et Algorithme.**

Cet algorithme repose sur la vitesse de convergence de la suite selon son premier terme. Cette vitesse sera distincte selon les points considérés ; pour bien marquer l'évolution de cette vitesse, il serait appréciable de tracer la figure correspondante en couleurs.

De combien de couleurs aurons-nous besoin ? Il est d'abord nécessaire de se demander comment nous allons tester la convergence de la suite. Le test couramment adopté pour ce problème consiste à détecter le moment où le module de z_n dépasse 2.

Ce test sera donc effectué dans une boucle d'itérations du calcul de la suite, boucle dont il faut bien sûr majorer le nombre d'itérations pour les cas où le module ne dépasse pas 2. Notons ce nombre maximal *iter*.

Le nombre de couleurs nécessaires ne peut pas excéder *iter*. Si l'on choisit de tracer cette fractale en jaune sur fond noir, on peut alors user d'une fonction qui alloue les différentes teintes de couleurs dans un tableau, telle que la fonction suivante :

```
unsigned long * alloue_couleurs (int iter)
{
   int r, g, b ;
   int i ;
   unsigned long * Tab ;
   Tab = (unsigned long *) malloc (iter * sizeof(unsigned long)) ;

   if ( !Tab)
   {
      fprintf(stderr , "Diantre et ventrebleu, l'allocation, n'a point pu
avoir lieu.") ;
      exit(1) ;
   }

   for (i = 1 ; i < iter+1 ; i++)
   color.blue = 0 ;
   {
      color.red = 65535*i/iter ;
      color.green = 65535*i/iter ;
      XAllocColor (dpy, colormap, &color) ;
      Tab[i] = color.pixel ;
   }
   return Tab ;
}
```

Il est ensuite indispensable, pour un point donné du plan, de déterminer la couleur dans laquelle il faut le tracer, et donc au bout de combien de temps se produit l'éventuel dépassement de 2 par le module de z_n. Pour ce faire, on utilise une nouvelle fonction, qui prend en argument les coordonnées (r_z0, i_z0) dudit point, les parties réelle et imaginaire (r_c, i_c) du complexe c (cf formule de récurrence permettant de définir l'ensemble de Julia) le nombre maiximum d'itérations *iter* ainsi que le tableau de couleurs Tab.

On ajoute également un indicice, noté *kbis*, permettant de savoir si le nombre d'itérations atteint est le même que lors du précédent appel de la fonction. Si tel est le cas, on évite de redéfinir la couleur de tracé. Ainsi, la fonction finale est la suivante :

```
void coul (float r_c , float i_c , float r_z0 , float i_z0 , int * kbis ,
int iter , unsigned long * Tab)
{
   float r_z = r_z0 , i_z = i_z0 , save;
   int k = 0;
   while ((r_z*r_z + i_z*i_z) < 4 && k < iter)
   {
      save = r_z;
      r_z = r_z*r_z - i_z*i_z + r_c;
      i_z = 2*save*i_z + i_c;
      k++;
   }
   if (k != *kbis)
   {
      XSetForeground(dpy , gcontext , Tab[k]);
   }
}
```

Il ne reste plus qu'à mettre en place la fonction principale, laquelle prend en arguments le complexe $c = (r\_c, i\_c)$, la taille de la fenêtre de travail $(size\_X, size\_Y)$, le nombre d'itérations maximales $iter$ pour les tests de convergence, le facteur d'échelle $facteur$, et le tableau de couleurs allouées Tab.

Tout repose sur une double boucle pendant laquelle on appelle la fonction précédente et l'on trace le point traité à cette étape. Les coordonnées du point susnommées sont celles du point de la fenêtre divisées par le facteur d'échelle. Ainsi, si ce facteur vaut 10, le point (5 ; 6) de la fenêtre sera le point (0,5 ; 0,6) du plan de tracé.

```
 void Julia (float r_c , float i_c , int size_X , int size_Y , int iter ,
float facteur , unsigned long * Tab)
{
   int i , j , k;
   float size = (size_X < size_Y ? size_X : size_Y);
   for (i = -size/2; i < size/2; i++)
   {
      for (j = -size/2; j < size/2; j++)
      {
         coul(r_c , i_c , ((float) i) / (facteur) , ((float) j) / (facteur),
&k , iter , Tab);
         XDrawPoint (dpy, win, gcontext, i + size_X / 2, j + size_Y
/ 2);
      }
   }
}
```

5.2. **Quelques figures.**

Toutes ces figures ont été obtenues avec
$(size\_X, size\_Y) = (1344, 840)$, $iter = 1000$, et $facteur = 300$

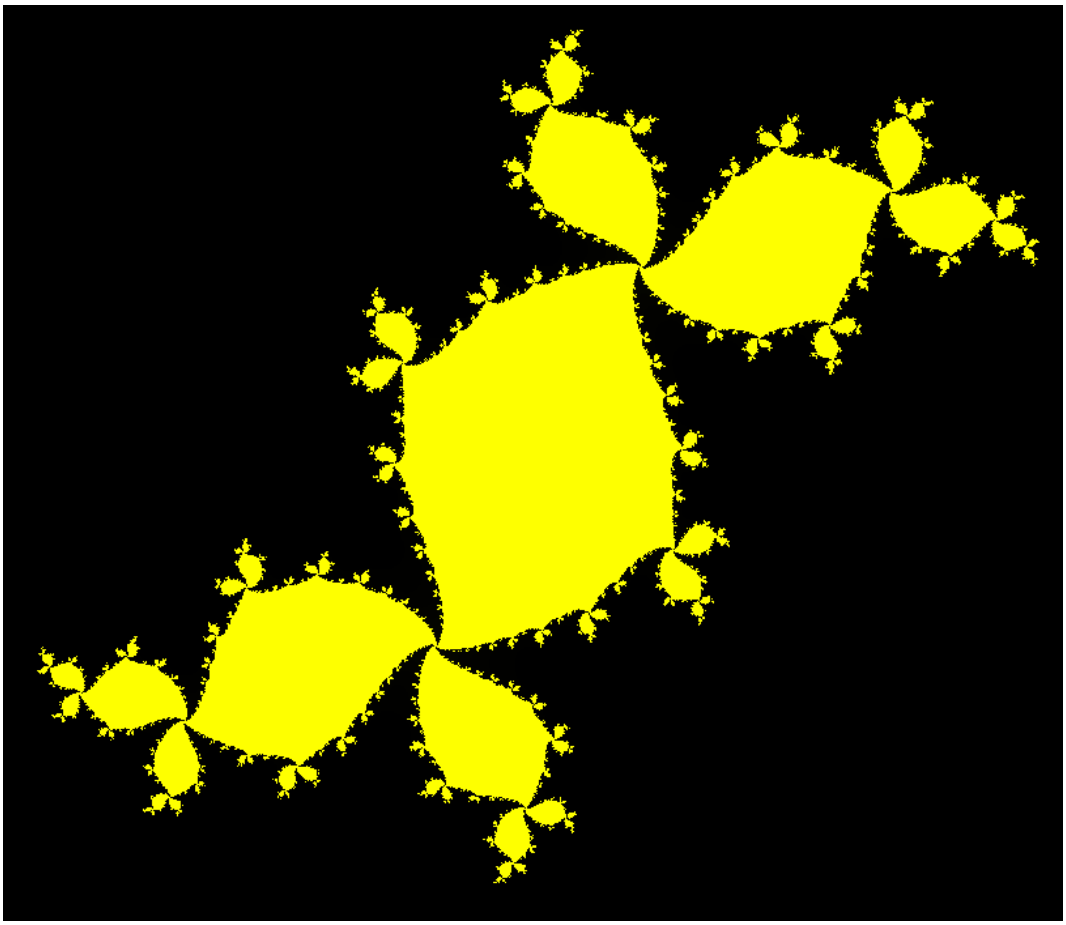

Fig. 8 :
Ensemble de Julia
Pour $C = -0,0958 + 0,735i$

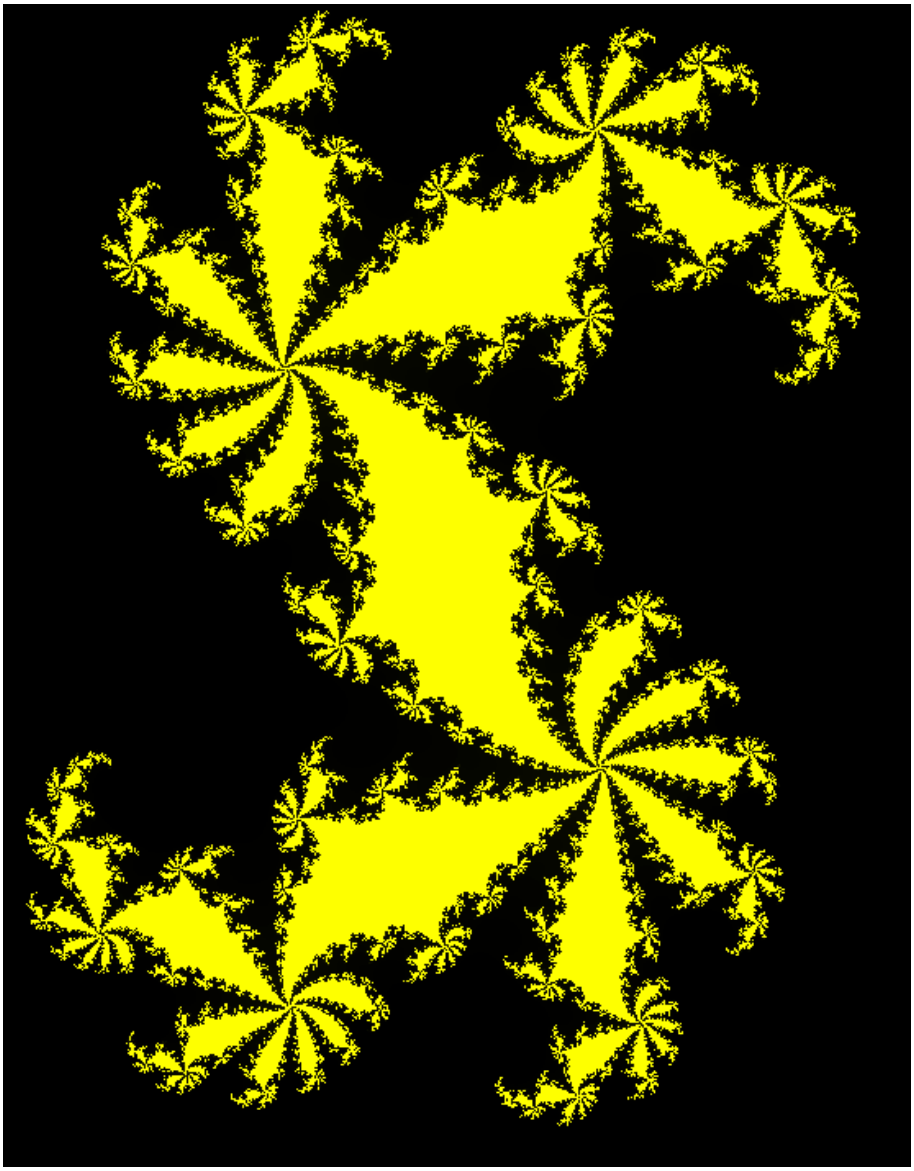

Fig. 9 :
Ensemble de Julia
Pour $C = 0,382 + 0,147i$

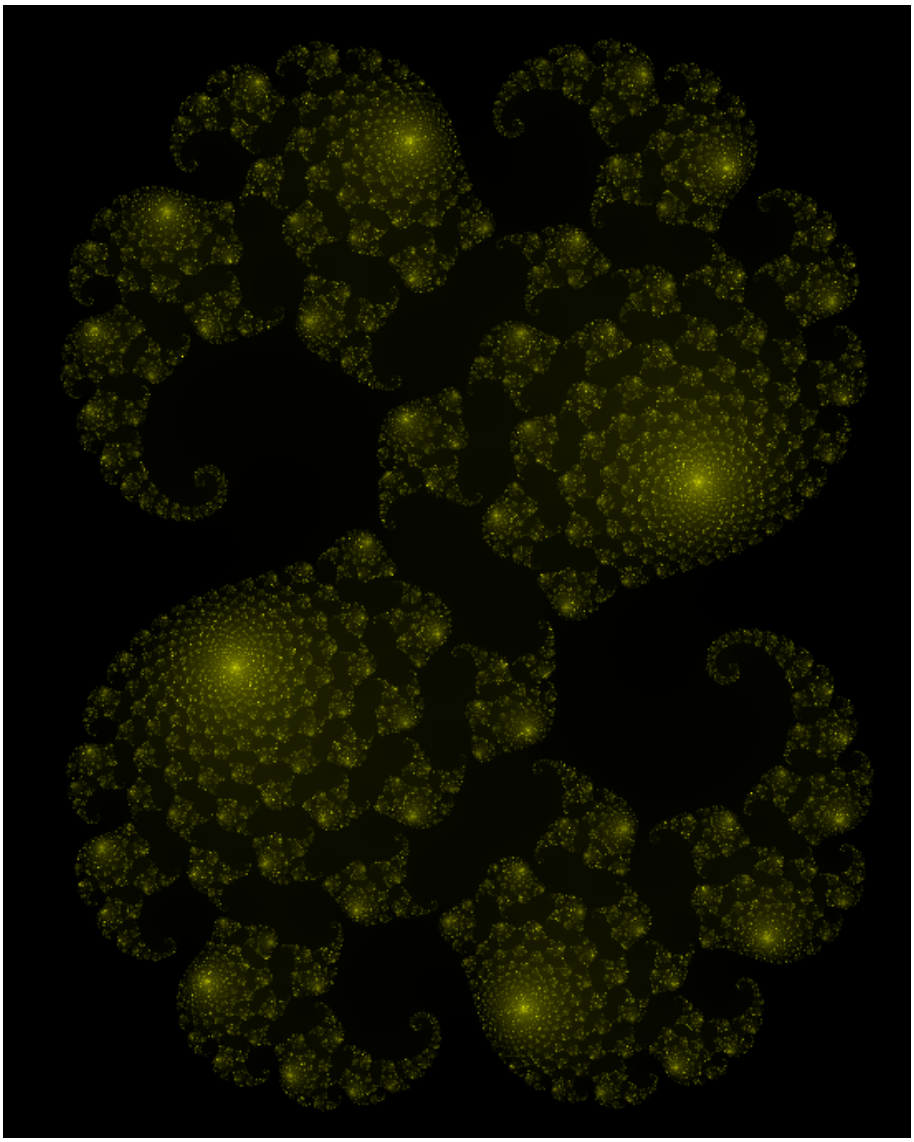

Fig. 10 :
Ensemble de Julia
Pour $C = 0,284 - 0,0122i$

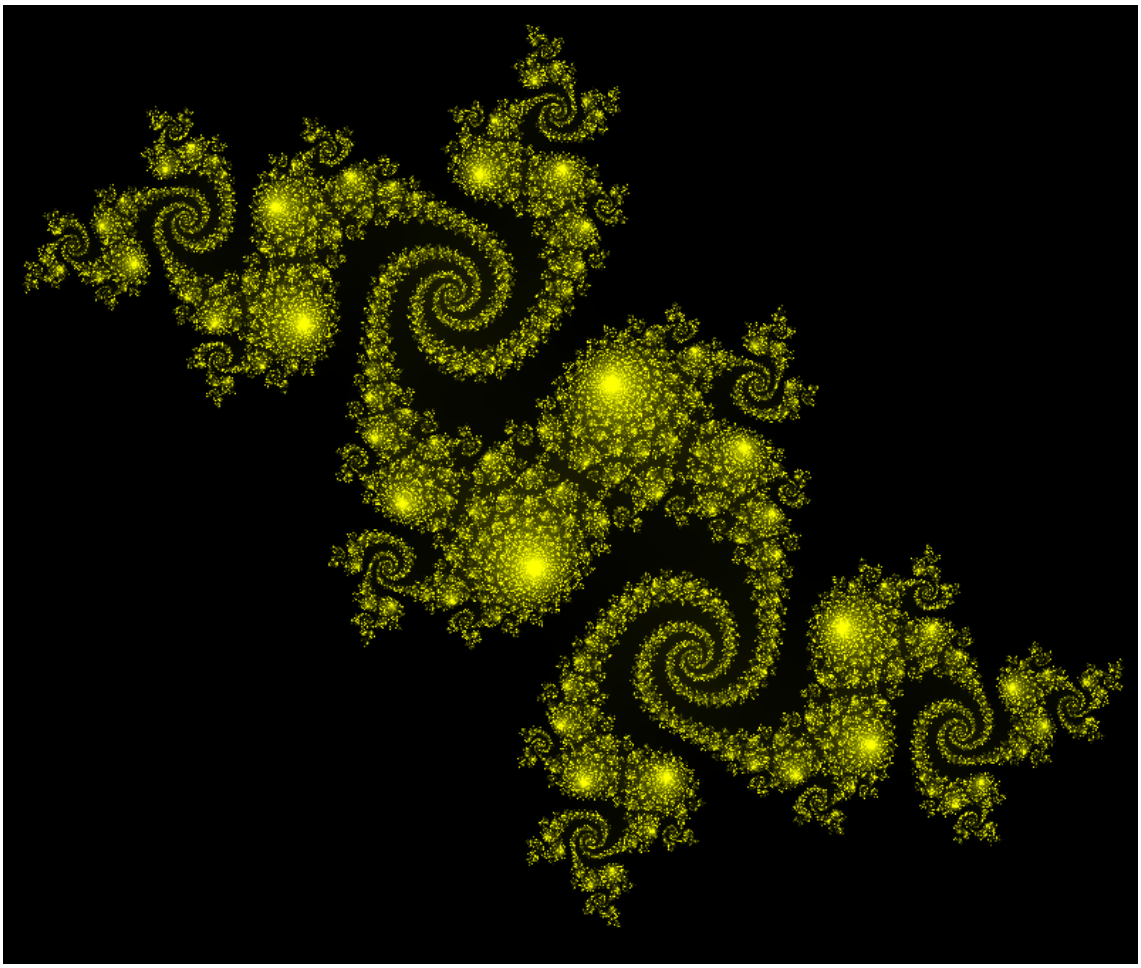

Fig 11 :
Ensemble de Julia
Pour $C = -0,181 - 0,667i$

6. Conclusion

A travers ce travail, nous avons donc pu constater que les méthodes de tracé de figures fractales sont nombreuses et variées : on peut utiliser une récursivité simple (Figures 1 et 2) ou lourdement paramétrée (Figures 3 et 4), ou bien se tourner vers des algorithmes itératifs, dont certains peuvent dépendre de la structure déjà construite (Figures 5 à 7) ou être inspirées d'objets mathématiques plus ou moins complexes (Ensembles de Julia : figures 8 à 11).

Il faut de plus noter que chacune de ces méthodes peut très bien s'appliquer au tracé de n'importe quel fractal (la construction du triangle de Sierpinski est originellement récursive) et qu'il est souvent possible de passer de l'une à l'autre (cf 2.3). Il eût même été possible d'envisager la programmation du tracé des ensembles de Julia sous forme récursive.

Cela posé, on pourrait parfaitement s'astreindre à n'user que d'algorithmes itératifs — ceux-ci étant plus rapidement traités par nos machines — afin d'uniformiser les méthodes de création de figures fractales et ainsi rendre leurs tracés informatiques plus efficaces.

7. Bibliographie

Les objets fractals : forme, hasard et dimension ; Flammarion 2010

La magie des fractales ; mag. Tangente, Hors-Série 18 (Mars 2004)