Empilements de cercles

Christophe Deleuze

26 août 2007

Résumé

La rubrique “Formes mathématiques” d’un récent numéro de la revue *Découverte* (revue du palais de la découverte) [1] présente les figures mathématiques appelées “empilements de cercles”. Ce programme, écrit en *Objective Caml* [2], permet de réaliser de telles figures.

Il a été rédigé dans le style “programmation littéraire” *(literate programming)*, qui intégre la documentation et le code source en un seul document. L’outil utilisé est Ocamlweb [3], qui permet de mêler code en Caml et documentation en LaTeX.

1. Un empilement de cercles est un ensemble de cercles tangents ou disjoints, construits de manière systématique comme sur la figure 1.

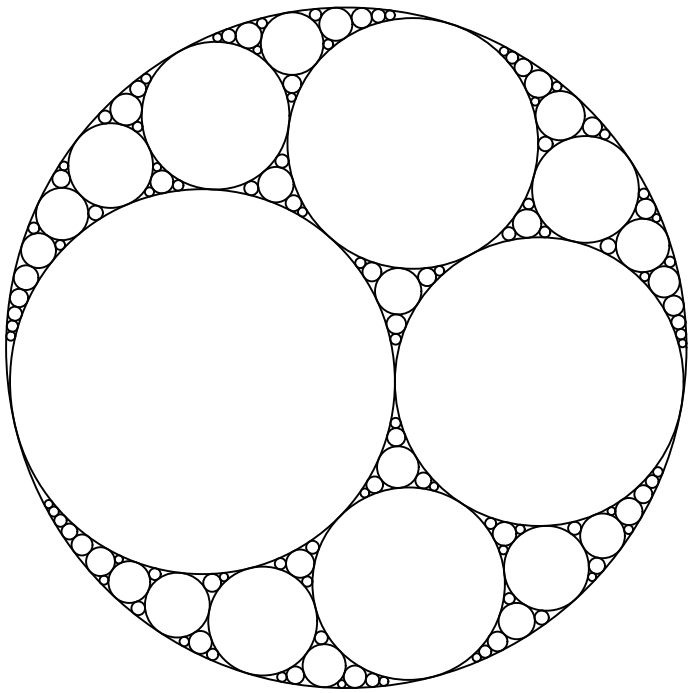

FIGURE 1 – Un empilement de cercles

2. Un théorème de René Descartes énonce que le carré de la somme des courbures (la courbure d'un cercle est définie comme l'inverse de son rayon) de quatre cercles deux à deux tangents est égal au double de la somme de leurs carrés :

$$2(e^2+f^2+g^2+h^2) = (e+f+g+h)^2$$

Ainsi, si nous connaissons les courbures e, f et g de trois cercles tangents deux à deux, nous devons résoudre une simple équation du second degré pour déterminer la courbure d'un quatrième cercle tangent aux trois autres. Il y a en fait toujours deux cercles possibles, il y aura donc toujours des solutions (deux distinctes ou une double dans le cas où les deux cercles solutions ont la même courbure). Une courbure peut être négative : cela signifie que le cercle en question englobe les autres, comme c'est le cas du grand cercle de la figure 1. La fonction ci-dessous calcule les deux courbures en question.

```
let solve_real e f g =
  let delta = 16. *. (e *. f +. e *. g +. f *. g) in
  let sqd = sqrt delta
  in
  (2. *. (e +. f +. g) -. sqd) /. 2., (2. *. (e +. f +. g) +. sqd) /. 2.
```

3. Une généralisation de cette relation de Descartes s'applique aux nombres complexes $c_i z_i$, produits de la courbure c_i et du complexe représentant le centre du cercle z_i.

$$2((c_e z_e)^2+(c_f z_f)^2+(c_g z_g)^2+(c_h z_h)^2) = (c_e z_e + c_f z_f + c_g z_g + c_h z_h)^2$$

En la résolvant de la même façon, on obtient le produit complexe courbure fois position du centre pour chacun des deux cercles solutions. Pour rendre les expressions lisibles les opérateurs arithmétiques sur les entiers sont redéfinis provisoirement pour manipuler les complexes.

```
let solve_complex e f g =
  let c16 = { Complex.re = 16.; Complex.im = 0.}
  and c2 = { Complex.re = 2.; Complex.im = 0.}
  in
  let ( * ) = Complex.mul and ( / ) = Complex.div
  and ( + ) = Complex.add and ( - ) = Complex.sub
  in
  let delta = c16 * (e*f + e*g + f*g) in
  let sqd = Complex.sqrt delta
  in
  (c2*(e+f+g) - sqd) / c2, (c2*(e+f+g) + sqd) / c2
```

4. Nous pouvons donc calculer d'une part les deux courbures *ca* et *cb*, d'autre part les deux produits courbure fois position du centre *cz1* et *cz2*. Il nous reste à associer les deux pour connaître les deux cercles solutions.

Par définition, la distance de chacun des deux centres au centre de l'un quelconque des trois cercles initiaux, le cercle de référence (zr, cr), est égale au rayon du cercle de référence plus le rayon du cercle construit (c'est à dire l'inverse de sa courbure).

La phrase précédente est vraie si les deux courbures sont positives. Si l'une est négative (cercle "contenant" l'autre) les rayons doivent être soustraits. Si nous considérons que les rayons peuvent être négatifs (inverse de la courbure), l'opération reste une addition (par contre le résultat peut être négatif et il faudra en prendre la valeur absolue).

Nous calculons les deux points *z1a* et *z2a* correspondant aux deux choix de centres possibles pour la courbure *ca*, calculons les distances avec *zr* (ce sont les modules des complexes $z1a-zr$ et $z1b-zr$) et y soustrayons la distance théorique $|\frac{1}{ca}+\frac{1}{cr}|$. La différence devrait être nulle pour l'un des deux points. Pour tenir compte des erreurs d'arrondis nous choisissons celui pour lequel elle est la plus faible en valeur absolue.

Nous pouvons renvoyer les valeurs des deux cercles correspondants. Dans la suite, un cercle sera toujours représenté par un couple (complexe, réel) représentant son centre et sa courbure.

```
let cdiv { Complex.re = re; Complex.im = im } r =
  { Complex.re = re /. r; Complex.im = im /. r }

let associate_center_and_curve cz1 cz2 ca cb (zr, cr) =
  let z1a = cdiv cz1 ca
  and z2a = cdiv cz2 ca
  and ( - ) = Complex.sub
  in
  let d1 = abs_float((Complex.norm (z1a - zr)) -. (abs_float (1./. cr +. 1./. ca)))
  and d2 = abs_float((Complex.norm (z2a - zr)) -. (abs_float (1./. cr +. 1./. ca)))
  in
  if d1 < d2 then (z1a, ca), (cdiv cz2 cb, cb)
  else (z2a, ca), (cdiv cz1 cb, cb)
```

5. Étant donnés trois cercles tangents deux à deux, nous avons maintenant tout ce qu'il nous faut pour déterminer les deux cercles qui sont tangents à ces trois cercles.

```
let cmul { Complex.re = re; Complex.im = im } r =
  { Complex.re = re *. r; Complex.im = im *. r }

let get_circles (z1,c1) (z2,c2) (z3,c3) =
  let ca, cb = solve_real c1 c2 c3
  and cz1, cz2 = solve_complex (cmul z1 c1) (cmul z2 c2) (cmul z3 c3)
  in
  associate_center_and_curve cz1 cz2 ca cb (z1,c1)
```

6. Nous avons dit qu'il y a deux cercles solutions. Si une seule nous intéresse, comment les distinguer ? Considérons d'abord le cas où les courbures des cercles initiaux C_1, C_2 et C_3 sont toutes positives. L'un des deux cercles solutions est *intérieur*, "coincé" à l'intérieur des trois cercles, l'autre est *extérieur*. Dans la figure 2 à gauche, le cercle extérieur (C_e) englobe les trois autres et a donc une courbure négative, mais ce n'est pas forcément le cas, comme le montre la

figure 2 à droite. Par contre, on peut facilement se convaincre (et démontrer si nécessaire) que le cercle intérieur (C_i) a toujours le rayon le plus petit, et donc que sa courbure est la plus grande.

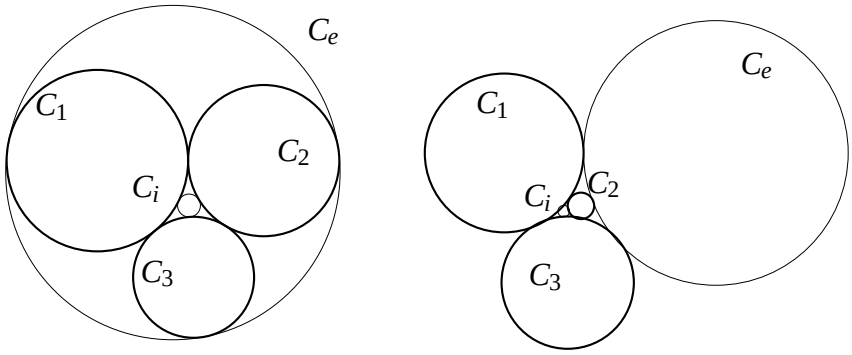

FIGURE 2 – Position des solutions si les trois cercles ont une courbure positive

Dans la suite du programme *ci* désignera la courbure du cercle C_i alors que *c_i* désignera le cercle lui même, c'est à dire le couple (zi, ci).

```
let out_circle c_1 c_2 c_3 =
  let (za,ca), (zb,cb) = get_circles c_1 c_2 c_3
  in
  if ca < cb then (za,ca) else (zb,cb)

let in_circle c_1 c_2 c_3 =
  let (za,ca), (zb,cb) = get_circles c_1 c_2 c_3
  in
  if ca > cb then (za,ca) else (zb,cb)
```

7. Un deuxième cas est celui où l'un des cercles initiaux a une courbure négative (C_3 sur la figure 3). Les deux solutions C_g et C_d ont alors toujours une courbure positive, comme le montre la figure. Nous ferons en sorte que le cercle à courbure négative soit toujours le troisième et, en posant par convention que C_2 est au dessus de C_1, nous pourrons dire que l'une des deux solutions (C_d) est "à droite" et l'autre (C_g) "à gauche" des cercles C_1 et C_2.

Nous commençons par définir une fonction qui nous indique si un point de coordonnées z_m est à droite des deux points de coordonnées z_a et z_b. C'est le cas si l'angle $\widehat{ABM}$ est compris entre 0 et π, c'est à dire de sinus positif. Cet angle est la différence entre les angles des vecteurs $\overrightarrow{BM}$ et $\overrightarrow{BA}$. Nous l'utilisons ensuite dans la fonction *right_circle* qui nous renvoie donc le cercle "de droite".

```
let pi = acos(-1.)

let on_right zm za zb =
  let th_bm = Complex.arg (Complex.sub zm zb)
  and th_ba = Complex.arg (Complex.sub za zb)
  in
  sin (th_bm -. th_ba) ≥ 0.
```

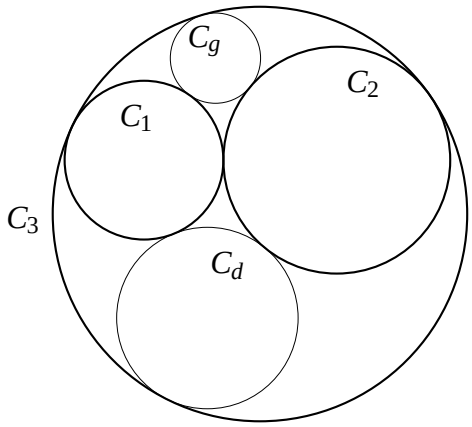

FIGURE 3 – Position des solutions si l'un des trois cercles (C_3) a une courbure négative

```
let right_circle (z1,c1) (z2,c2) (z3,c3) =
  let (za,ca), (zb,cb) = get_circles (z1,c1) (z2,c2) (z3,c3)
  in
  if on_right za z1 z2 then (za,ca) else (zb,cb)
```

8. Définissons maintenant comment construire récursivement notre figure. Nous partons de trois cercles C_1, C_2 et C_3 tangents deux à deux et d'un rayon minimal *min* en deçà duquel nous stopperons la récursion. Considérons tout d'abord que tous trois sont à courbure positive. Nous voulons construire la liste des cercles qui prennent place dans l'espace délimité par C_1, C_2 et C_3.

Nous savons calculer le cercle C_4, que nous avons défini précédemment comme le cercle *intérieur*. Notons que ce cercle est le plus grand de tous les cercles que pourra contenir notre espace. Une fois ce cercle construit, il reste à remplir les trois espaces délimités respectivement par C_1, C_4, C_3, par C_4, C_2, C_3 et par C_1, C_2, C_4. Ces espaces sont notés A, B et C sur la figure 4.

Nous pouvons donc définir notre liste récursivement :

- cas de base : si l'espace est trop petit (le rayon de C_4 est inférieur au rayon *min*), il ne contiendra aucun cercle, la réponse est donc la liste vide,
- sinon l'espace contiendra le cercle C_4, ainsi que les cercles remplissant les espaces délimités par C_1, C_4, C_3, par C_4, C_2, C_3 et par C_1, C_2, C_4.

Nous devons aussi traiter le cas des espaces délimités par trois cercles dont l'un a une courbure négative. Considérons qu'il s'agit de C_3. Comment remplir l'espace "de droite" ? De manière très similaire au cas précédent : nous calculons d'abord C_d le cercle "de droite" puis devons calculer les cercles entrant dans les espaces délimités par C_1, C_d et C_3 (A sur la figure 5), par C_d, C_2 et C_3 (B) et C_1, C_2 et C_d (C). Les zones A et B sont traitées en constuisant le cercle "de droite" alors que la zone C est traitée en construisant le cercle "intérieur".

Ne mais doit-on pas remplir aussi l'espace "de gauche" ? Pas maintenant, remarquez que le cercle de gauche de C_d, C_1 et C_3 existe déjà, c'est C_2, de même que C_1 pour C_2, C_d et C_3.

Nous pouvons fusionner les deux cas. Étant donnés trois cercles C_1, C_2 et C_3, avec C_3 ayant éventuellement une courbure négative nous calculons un quatrième cercle C_4 (l'intérieur si C_3 a une courbure positive, celui de droite dans le cas contraire) et procédons récursivement sur les

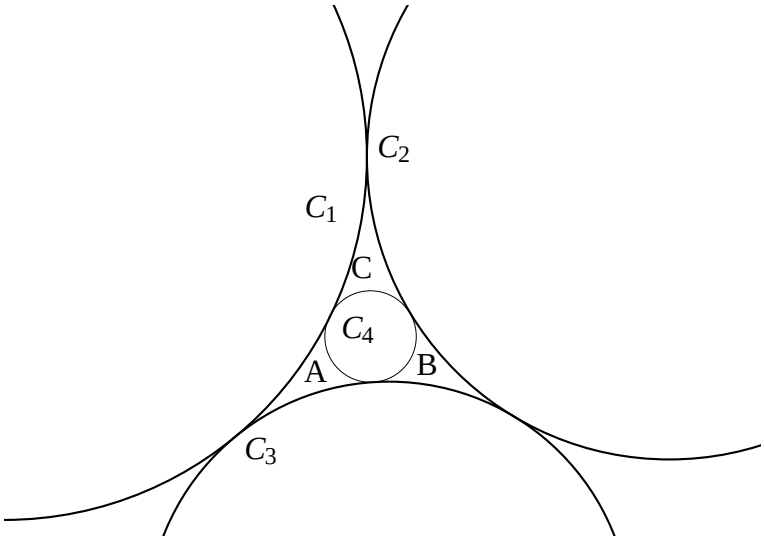

FIGURE 4 – Construction récursive pour trois cercles à courbure positive

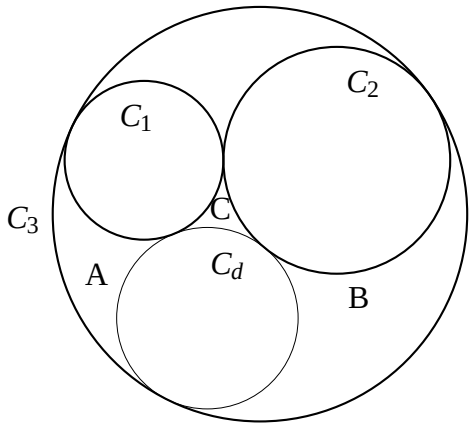

FIGURE 5 – Construction récursive dans le cas où C_3 a une courbure négative

triplets de cercles :

- C_1, C_4 et C_3,
- C_4, C_2 et C_3,
- C_1, C_2 et C_4.

Notons qu'ici nous posons qu'au départ seul C_3 est susceptible d'avoir une courbure négative, ce qui sera maintenu dans la récursion puisqu'il se trouve toujours en dernière position dans les appels récursifs.

```
let rec recurse c_1 c_2 c_3 min =
  let z3, c3 = c_3 in
  let c_4 = if c3 > 0. then in_circle c_1 c_2 c_3
  else right_circle c_1 c_2 c_3
  in
```

```
let r = match c_4 with (_,c) → 1./.c
in if r > min then c_4 ::
   recurse c_1 c_4 c_3 min @
   recurse c_4 c_2 c_3 min @
   recurse c_1 c_2 c_4 min
else
   []
```

9. Et voici comment amorcer la récursion. Nous avons trois cercles C_1, C_2 et C_3 tangents deux à deux, tous à courbure positive et nous voulons construire l'empilement de cercles déterminé par ces trois cercles.

Il nous faut tout d'abord déterminer le cercle C_e englobant ces trois. S'il n'existe pas (le cercle externe est à courbure positive) nous abandonnons. [1]

Sinon notre empilement est constitué de ces quatre cercles, plus ceux qui seront construits dans les espaces délimités par (figure 6) :

- C_1, C_2 et C_3 (A),
- C_1, C_3 et C_e (B),
- C_3, C_2 et C_e (C),
- C_2, C_1 et C_e (D).

Notons que le cercle de courbure négative (C_e) est toujours le dernier dans les appels à *recurse*. Le fait que nous n'ayons pas traité de cercle "de gauche" au paragraphe précédent est maintenant justifié : un espace à gauche est aussi un espace à droite pour une autre combinaison de cercles.

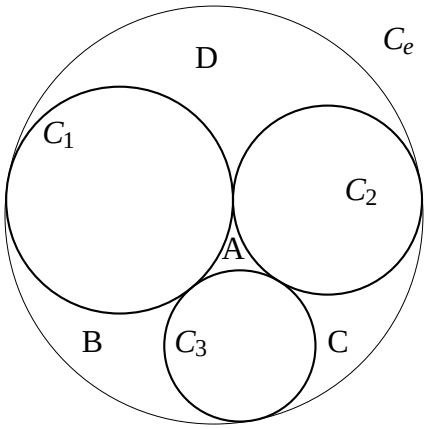

FIGURE 6 – Construction complète à partir de trois cercles

```
let start c_1 c_2 c_3 min =
   let c_e = out_circle c_1 c_2 c_3 in
   let (ze,ce) = c_e in
   if ce > 0. then failwith "Giving up!"
```

1. Nous pourrions en fait construire le cercle englobant les trois plus gros, qui sera lui de courbure négative.

```
else
  c_e :: c_1 :: c_2 :: c_3 ::
  recurse c_1 c_2 c_3 min @
  recurse c_1 c_3 c_e min @
  recurse c_3 c_2 c_e min @
  recurse c_2 c_1 c_e min
```

10. Pour commencer la construction, il faut tout d'abord disposer de trois cercles tangents deux à deux, tous à courbure positive. Voici comment construire trois tels cercles C_1, C_2 et C_3, de rayons quelconques r_1, r_2 et r_3.

Dans ce paragraphe nous raisonnons en rayons, les courbures ne nous étant d'aucun intérêt. Centrons C_1 en O, on a donc $C_1 = (O, r_1)$. Nous pouvons centrer C_2 en $A(r_1 + r_2, 0)$, C_2 sera bien tangent à C_1. Reste à construire un cercle C_3 de rayon r_3, tangent à C_1 et C_2. Par définition, son centre B sera à la distance $r_1 + r_3$ de O et $r_2 + r_3$ de A. Il y a deux positions possibles pour ce point B, qui sont les deux intersections des cercles $(O, r_1 + r_3)$ et $(B, r_2 + r_3)$.

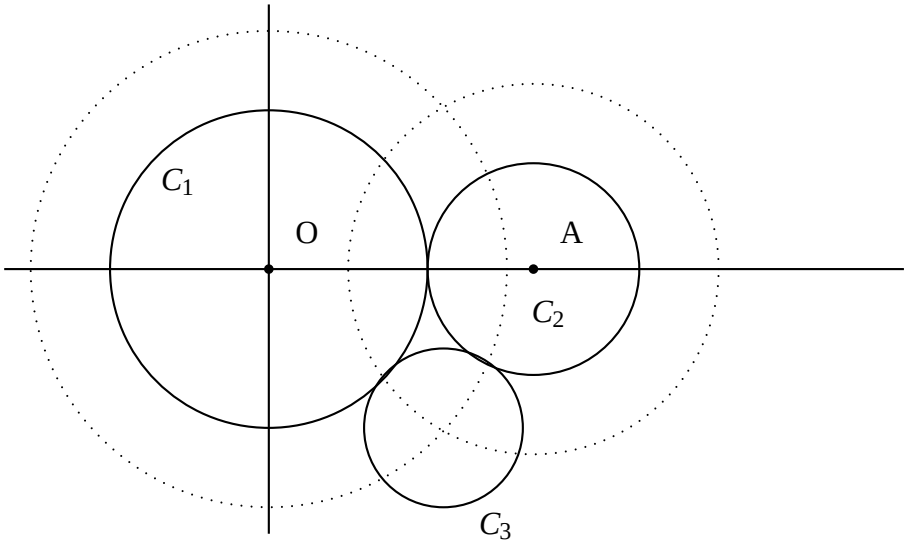

FIGURE 7 – Calcul du centre du troisième cercle

Pour rester cohérent avec ce que nous avons fait jusqu'à présent, nous choisirons pour C_3 la position "à droite" de C_1 et C_2, c'est à dire avec une ordonnée négative (figure 7).

On peut calculer analytiquement les coordonnées de ces points d'intersection à partir des équations des cercles dans le cas général. Cependant, vues les positions choisies pour les centres, notre cas particulier est bien plus simple. Nous laisserons au lecteur le plaisir de vérifier le détail du calcul effectué par la fonction *third_center*, qui calcule donc la position du cercle C_3. Dans le cas où la valeur de *delta* serait négative suite aux erreurs d'arrondis nous levons une exception pour le signaler.

```
let third_center r1 r2 r3 =
  let ra = r1 +. r3
  and rb = r2 +. r3
  and x2 = r1 +. r2 in
  let ra2 = ra **2.
  in
  let x = (-. (x2 **2.) -. ra2 +. rb **2.) /. (2. *. (-. x2)) in
  let delta = -. 4. *. (x **2. -. ra2)
  in
  if delta < 0. then failwith "third_center"
  else { Complex.re = x; Complex.im = -. (sqrt delta) /. 2. }
```

11. Nous pouvons maintenant, à partir des rayons des trois cercles initiaux et du rayon minimal, construire la liste des cercles de l'empilement. Il suffit pour cela de construire les trois cercles initiaux puis d'appeler la fonction *start*.

```
let compute_circles r1 r2 r3 min =
  let c1 = { Complex.re = 0.; Complex.im = 0. }, 1./. r1
  and c2 = { Complex.re = r1 +. r2; Complex.im = 0.}, 1./. r2
  and c3 = third_center r1 r2 r3, 1./. r3
  in
  start c1 c2 c3 min
```

12. Tant que nous y sommes, nous pouvons dessiner un empilement de cercles *fractal*. Pour cela nous calculons notre liste de cercles, puis remplisson chacun de ces cercles avec une version réduite de l'empilement. En fait ce que nous obtenons ici n'est pas vraiment fractal, car nous ne reproduisons la figure qu'une fois dans chaque cercle initial (figure 8). Le premier cercle de la liste est le cercle extérieur de la figure, nous l'utilisons pour calculer le facteur de réduction à appliquer à la liste pour chacun des cercles.

```
let almost_fractal_circles ((ze,ce) :: circles) cmax =
  List.flatten
    (List.map
       (fun (zi,ci) ->
          let scl = abs_float (ci /. ce)
          in
          List.map
            (fun (z,c) -> Complex.add (cdiv (Complex.sub z ze) scl) zi, c *. scl)
            (List.filter (fun (z,c) -> c *. scl < cmax) circles))
       circles)
```

13. Pour obtenir une vraie figure fractale, nous devons reproduire la figure *aussi* dans les nouveaux cercles que nous ajoutons, tant que leur courbure est inférieure au maximum fixé. Ça fait pas mal travailler l'ordinateur (il y a beaucoup plus de cercles), et le résultat n'est pas très

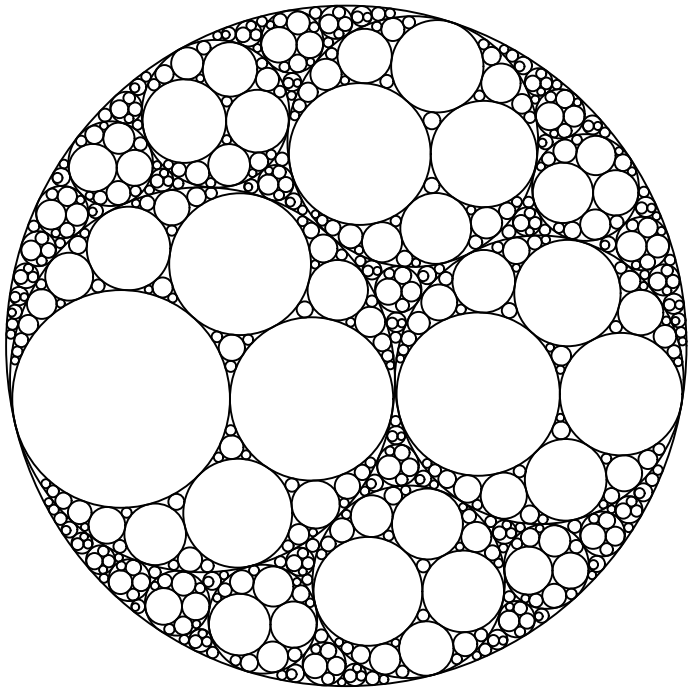

FIGURE 8 – Version presque fractale

joli. Pour alléger la figure 9 nous avons choisi un rayon limite plus élevé que dans les exemples précédents.

```
let fractal_circles ((ze,ce) :: circles) cmax =
  let smallest = List.fold_left (fun sml (z,c) -> min sml c) max_float circles
  in
  let rec help acc candidates =
    match candidates with
    | [] -> acc
    | (z,c) :: others ->
        if (c/.ce) *. smallest > cmax then
          (* this candidate has too big a curve, skip it *)
          help acc others
        else
          let newcircles =
            let scl = abs_float (c /. ce) in
            List.map
              (fun (zi,ci) -> Complex.add (cdiv (Complex.sub zi ze) scl) z, ci *. scl)
              (List.filter (fun (z,c) -> c *. scl < cmax) circles)
          in
          (* new circles are added to acc but are also new candidates *)
          (* to receive a copy of the original figure *)
```

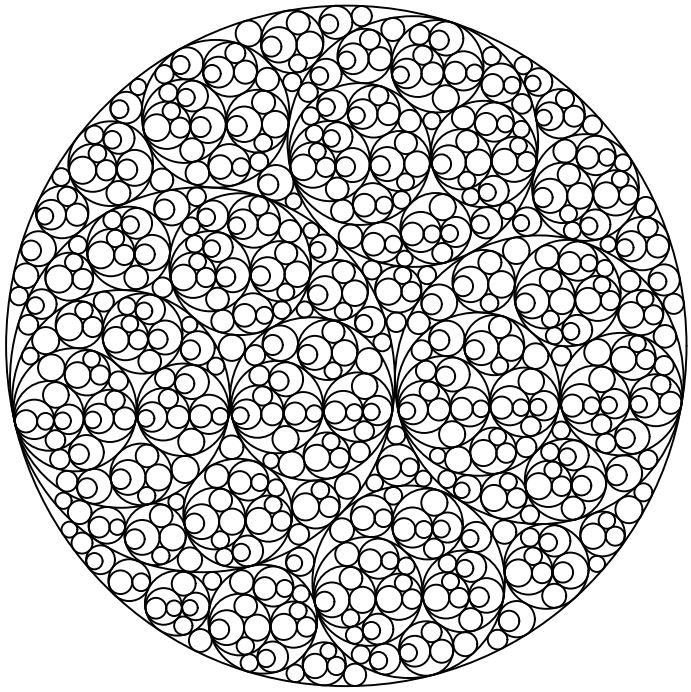

FIGURE 9 – Version fractale

```
        help (acc @ newcircles) (others @ newcircles)
in
help [] circles
```

14. Il nous reste à présenter l'information calculée, c'est à dire convertir notre liste de cercles codés sous la forme d'un couple (complexe, réel) en une représentation pour l'utilisateur. Nous laisserons le choix entre quatre possibilités :

1. un affichage graphique direct (pour voir immédiatement l'aspect de l'empilement calculé),
2. une liste textuelle des position, rayon et courbure des cercles (données qui pourront être stockées dans un fichier pour être ensuite utilisées par un autre programme),
3. une représentation graphique dans un fichier PostScript encapsulé,
4. idem mais avec un texte mis à l'échelle dans chaque cercle.

Pour les trois derniers cas, le résultat sera stocké dans un fichier (la sortie standard par défaut).

15. Il n'y a pas grand chose à dire sur la fonction *to_text* qui génère la représentation textuelle sur la sortie standard.

```
let to_text fout circles r1 r2 r3 min =
  Printf.fprintf fout "# Empilement de cercles r1=%f r2=%f r3=%f min=%f\n"
    r1 r2 r3 min;
  List.iter
    (fun ({Complex.re = x; Complex.im = y}, c)
      -> Printf.fprintf fout "%f %f %f %f\n" x y (1./. c) c)
    circles
```

16. Pour les deux premières représentations graphiques, nous utiliserons une structure de données graphiques fournie par le module Pictures [4]. Il permet de manipuler des figures composées d'une liste d'objets graphiques élémentaires et fournit des fonctions permettant d'afficher directement la figure à l'écran ou de générer un fichier postscript encapsulé. Le seul type d'objets dont nous aurons besoin est $Circle(x,y,r)$ où x et y sont les coordonnées du centre du cercle et r son rayon (tous trois de type *float*). La fonction *make_picture* retourne une valeur de type *pict*.

```
let make_pict circles =
  let pict_of_vc ({Complex.re = x; Complex.im = y}, c) =
    [ Picture.Circle(x, y, abs_float(1./. c)) ]
  in
  Picture.make_picture (List.flatten (List.map pict_of_vc circles))
```

17. Le dessin illustrant l'article [1] contient dans chaque cercle un nombre représentant sa courbure. Ce nombre est centré et d'une taille remplissant son cercle. Ceci n'est pas possible avec le module Pictures, mais peut facilement être réalisé en PostScript. Nous générons donc directement du code PostScript [5] que nous ne commenterons pas ici. La fonction *to_eps* génère le fichier sur la sortie standard. Le résultat est une figure du type de la figure 10. Le nombre affiché dans chaque cercle est la partie entière de la courbure multipliée par le paramètre *sc*. *lw* fixera l'épaisseur des traits.

```
let version = 0.96

let to_eps fout (c0 :: circles) r1 r2 r3 min sc lw =
  Printf.fprintf fout "%%!PS-Adobe-2.0 EPSF-2.0
   %%%%Title: Empilement de cercles r1=%f r2=%f r3=%f min=%f
   %%%%Creator: circles.ml %f\n"
    r1 r2 r3 min version;
  let x,y,r = match c0 with ({Complex.re = x;Complex.im = y},c) -> x,y,-1./.c in
  Printf.fprintf fout "%%%%BoundingBox: %f %f %f %f\n" (x-.r) (y-.r) (x+.r) (y+.r);
  Printf.fprintf fout "%s"
"%%DocumentFonts: Helvetica-Bold
   %%EndComments
   /textincircle { % x y r t
       /t exch def
       /r exch def
       /y exch def
```

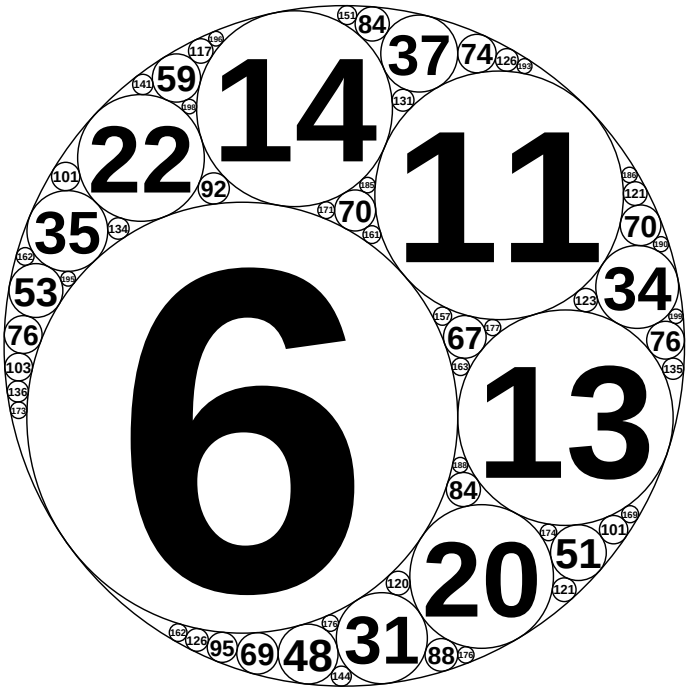

FIGURE 10 – Empilement avec indication des courbures

```
    /x exch def
    gsave
    % compute text size
    newpath 0 0 moveto
    t true charpath flattenpath pathbbox
    2 index sub /height exch def
    2 index sub /width exch def
    pop pop

    % gs BUG? pathbbox gives wrong width for strings with odd numbers of chars
    t stringwidth pop /width exch def

    % draw circle
    x y translate
    newpath 0 0 r 0 360 arc stroke
    % set text scale
    height width atan cos r mul 2 mul width div dup scale
    % draw centered text
    width 2 div neg height 2 div neg moveto t show
    grestore
} def
/c { % x y r c
```

```
    (mystring) cvs textincircle
} def
%%EndProlog
/Helvetica-Bold findfont 1 scalefont setfont
";
```

```
Printf.fprintf fout "%f setlinewidth\n" lw;
Printf.fprintf fout "newpath %f %f %f 0 360 arc stroke\n" x y r;
List.iter
  (fun ({Complex.re = x; Complex.im = y}, c)
    → Printf.fprintf fout "%f %f %f %i c\n"
        x y (1./. c) (int_of_float (c ×. sc)))
  circles;
Printf.fprintf fout "%s" "%%Trailer\n"
```

18. Pour finir, il nous faut traiter les arguments du programme, lancer les calculs et sauver ou afficher les résultats. Le programme attend comme arguments quatre nombres décimaux (les rayons des trois cercles initiaux et le rayon minimal) et accepte les options `-t` pour générer la représentation texte, `-e` pour générer la représentation postscript encapsulé et `-E` pour celle avec les nombres dans les cercles. Si aucune de ces trois options n'est donnée, l'empilement de cercles sera affiché à l'écran. La côté de la fenêtre peut alors être spécifiée par l'option `-w`. Les options `-f` et `-F` sélectionnent les versions fractales. Enfin `-s` et `-l` fixent, en conjonction avec `-E`, le facteur multiplicateur appliqué à la courbure pour déterminer le nombre affiché dans chaque cercle et l'épaisseur des traits. Le nom du fichier de sortie (par défaut la sortie standard) sera fixé par `-o`.

```
let text = ref false
and eps = ref false
and eps_nb = ref false
and output = ref ""

and fractal = ref false
and almostf = ref false
and wsize = ref 500
and scale = ref 1000.
and lw = ref 1.

let all_rads = ref false
and rads = Array.make 4 0.

let get_rads =
  let cpt = ref 0
  in
  fun s → rads.(!cpt) ← float_of_string s;
    incr cpt; if !cpt = 4 then all_rads :=true
```

```
let spec = [
  ("-t", Arg.Set text, "text representation");
  ("-e", Arg.Set eps, "eps file");
  ("-E", Arg.Set eps_nb, "eps file with numbers");
  ("-o", Arg.Set_string output,"output file (default stdout)");
  ("-w", Arg.Set_int wsize, "<size> window size for on screen display "^
   "(default "^string_of_int !wsize^")");
  ("-s", Arg.Set_float scale, "<scale> for eps, multiply label values "^
   "(default "^string_of_float !scale^")");
  ("-l", Arg.Set_float lw, "<linewidth> for eps "^
   "(default "^string_of_float !lw^")");
  ("-f", Arg.Set almostf, "almost fractal figure");
  ("-F", Arg.Set fractal, "fractal figure")
]
and usage_msg = "circles r1 r2 r3 rmin (draw on screen by default)"

let main r1 r2 r3 min =
  (* compute *)
  let cl =
    let circles = compute_circles r1 r2 r3 min in
    circles @ (if !fractal then fractal_circles circles (1./.min) else
    if !almostf then almost_fractal_circles circles (1./.min) else [])
  in
  Printf.fprintf stderr "Computed %i circles.\n" (List.length cl);
  flush stdout;

  let fout = if !output ="" then stdout else open_out !output
  in

  (* display *)
  if !text then to_text fout cl r1 r2 r3 min (* text *)
  else if !eps then Picture.to_eps fout (make_pict cl) (* eps *)
  else if !eps_nb then to_eps fout cl r1 r2 r3 min !scale !lw (* eps with nbs *)
  else begin
    let ws = string_of_int !wsize in (* screen *)
    Graphics.open_graph (":0 "^ws^"X"^ws);
    Picture.to_screen (make_pict cl) 0 0 !wsize !wsize;
    Graphics.read_key (); ()
  end;
  if fout <> stdout then close_out fout;

  in
  Arg.parse spec get_rads usage_msg;
  if !all_rads then main rads.(0) rads.(1) rads.(2) rads.(3)
  else Arg.usage spec usage_msg
```

19. Ce programme a également été traduit en PostScript [**?**]. Les paramètres (tailles des trois cercles de départ et taille minimale) sont à éditer directement dans le fichier qui peut être ouvert avec un logiciel de visualisation de documents PostScript.

La traduction n'est pas difficile (pour quelqu'un familier avec PostScript) mais l'écriture directe aurait été certainement plus délicate. En effet, PostScript ne fournit pas de manière directe la récursion ni les structures de données. Il n'est pas bien difficile de les mettre en œuvre, mais cela constitue une distraction du but initial pour le programmeur. Un point plus fondamental est que le programmeur doit alors formuler une solution en terme de concepts qui ne sont pas directement fournis par le langage, c'est à dire qu'il a un langage pour penser la solution et un langage pour la mettre en œuvre. Il est bien plus confortable de pouvoir penser directement une solution dans le langage dans lequel elle sera mise en œuvre.

Références

[1] Attal (Romain). – Empilements de cercles. *Découverte*, no344-345, janvier-février 2007, pp. 8–9.

[2] INRIA. – The Caml language. Page web : `http://caml.inria.fr`.

[3] Filliâtre (Jean-Christophe) et Marché (Claude). – ocamlweb: a literate programming tool for objective caml. Page web : `http://www.lri.fr/~filliatr/ocamlweb/`.

[4] Deleuze (Christophe). – Ocaml Pictures module. – Logiciel non publié.

[5] Adobe (édité par). – *PostScript language tutorial and cookbook*. – Addison-Wesley, 1985.