

17 Dessin et fractales

Afin d'éviter des temps de calcul trop long, il est conseillée de ne pas excéder 25 millions⁵¹ de pixels par image.

17.1 Formes simples

L'utilisation du package `2htdp/image` permet de dessiner en utilisant des formes simples. Les fonctions de base sont `circle`, `ellipse`, `line`, `add-line`, `add-curve`, `add-solid-curve`, `text`, `text/font` et `empty-image`.

```
1 (require 2htdp/image)
2 (circle 30 "outline" "red")
3 (circle 20 "solid" "blue")
4 (circle 20 100 "blue")
```



Les fonctions `circle`, `ellipse` et `line` permettent de dessiner des formes du même nom; `circle` attend un rayon, un mode de tracé et une couleur; le mode de tracé peut être une valeur de transparence variant de 0 à 255; le mode de tracé `solid` est équivalent à la valeur 255; pour un mode de tracé de 0, le tracé est invisible; `ellipse` attend une largeur, une hauteur, un mode de tracé et une couleur; la largeur et la hauteur sont les dimensions du rectangle englobant l'ellipse; `line` attend une valeur en x, une valeur en y et une couleur; pour une valeur nulle en x, la ligne est verticale; pour une valeur nulle en y, la ligne est horizontale; pour une valeur négative en y, la ligne est vers le haut; la fonction `add-line` permet de surperposer des lignes sur un dessin en cours (l'image sur laquelle est superposée la ligne est passée en premier paramètre, les paramètres suivants concernant la ligne en question).

```
1 (require 2htdp/image)
2 (add-line (square 40 "solid" "gray")
3           0 40 80 0 "red")
```



Les fonctions `square` et `rectangle` permettent respectivement de tracer un carré et un rectangle.

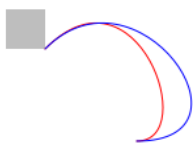
51. Ce qui avoisine les 100 mo. Pour une résolution 4K-UHD1, l'image est de 3840 par 2160, soit 8.3 millions de pixels. Pour une résolution de 1024 par 768, l'image contient un peu moins de 0.8 millions de pixels.

La fonction `add-curve` permet de tracer des courbes ; elle requiert 10 paramètres qui sont : l'image concernée par le tracé, 4 coordonnées de départ, 4 coordonnées de fin et une couleur de tracé ; les ensembles de 4 coordonnées sont x , y , l'angle en degré et le temps de conservation de cet angle.

```

1 (require 2htdp/image)
2 (add-curve (add-curve (square 30 "solid" "gray")
3               30 30 45 1 100 100 180 0.5 "red")
4               30 30 45 1 100 100 180 1 "blue")

```

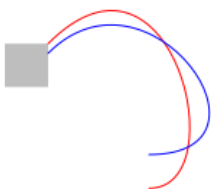


Les coordonnées de départ placent les tracés à partir des dimensions du dessin en cours ; ainsi l'ajout d'un tracé peut modifier les dimensions du dessin en cours et placer à des positions différentes des tracés ayant les mêmes coordonnées de départ ; dans le dessin ci-dessous, la courbe rouge agrandit la taille du dessin et modifie la position de départ de la courbe bleue.

```

1 (require 2htdp/image)
2 (add-curve (add-curve (square 30 "solid" "gray")
3               30 0 45 1 100 100 180 0.5 "red")
4               30 30 45 1 100 100 180 1 "blue")

```

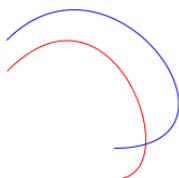


La fonction `empty-image` correspond au tracé d'un rectangle blanc de côté 0 ; ainsi superposer les deux courbes précédentes avec des coordonnées de départ en (0,0) équivaut à placer l'ordonnée de départ de la deuxième courbe au point le plus haut de la première.

```

1 (require 2htdp/image)
2 (add-curve (add-curve empty-image
3               0 0 45 1 100 100 180 0.5 "red")
4               0 0 45 1 100 100 180 1 "blue")

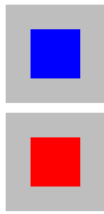
```



17.2 Superpositions de dessins

Les fonctions `overlay` et `underlay` permettent plus généralement de superposer des dessins de toutes formes; les deux dessins considérés sont superposés centrés; avec `underlay`, le premier paramètre est une sous-couche du second; avec `overlay`, le premier paramètre est une surcouche du second.

```
1 (require 2htdp/image)
2 (underlay (square 80 "solid" "gray") (square 40 "solid" "blue"))
3 (overlay (square 40 "solid" "red") (square 40 "solid" "gray"))
```



Avec les fonction `overlay/xy` et `underlay/xy`, il est possible de préciser des décalages entre les dessins superposés.

```
1 (require 2htdp/image)
2 (underlay/xy (square 80 "solid" "gray")
3   60 0
4   (square 40 "solid" "blue"))
5 (overlay/xy (square 40 "solid" "red")
6   60 0
7   (square 80 "solid" "gray"))
8 (overlay/xy (square 80 "solid" "gray")
9   60 0
10  (square 40 "solid" "red"))
```



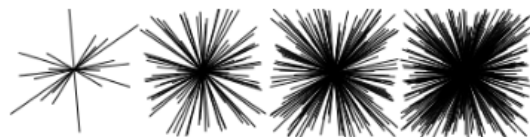
Les valeurs `x` et `y` définissent le décalage du second par rapport au premier; avec `underlay/xy`, le premier est une sous-couche, donc le second peut cacher partiellement le premier; avec `overlay/xy`, le premier est une sur-couche, donc le premier peut cacher partiellement le second.

Le programme suivant présente un autre exemple d'utilisation des fonctions de superposition.

```

1 (require 2htdp/image)
2 (define (rline n)
3   (if (= n 0) empty-image
4       (underlay
5         (line (random 100) (- (random 200) 100) "black")
6         (rline (- n 1))
7       )))
8 (underlay/xy (underlay/xy (underlay/xy
9                           (rline 10)
10                          100 0 (rline 50))
11               200 0 (rline 100))
12             300 0 (rline 200))

```



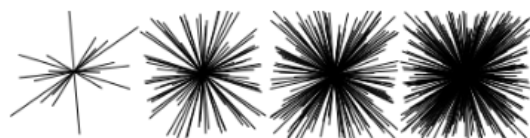
On utilise `underlay/xy` (ligne 8) pour placer les figures les unes à côté des autres; comme les images tracées sont sans intersection, utiliser `overlay/xy` avec les mêmes coordonnées donnerait le même résultat.

Il est également possible de placer les images les unes par rapport aux autres à l'aide des fonctions `above` (*i.e.* verticalement vers le bas⁵²), `above/align` (*i.e.* verticalement vers le bas avec un alignement à gauche ou à droite), `beside` (*i.e.* horizontalement vers la gauche) et `beside/align` (*i.e.* idem avec alignement); ces fonctions prennent en argument une liste d'objets à dessiner; avec l'alignement, il faut ajouter une string qui précise l'alignement "`right`", "`left`", "`baseline`", "`top`" ou "`bottom`"; ainsi le programme précédent s'écrit comme suit :

```

1 (require 2htdp/image)
2 (define (rline n)
3   (if (= n 0) empty-image
4       (underlay
5         (line (random 100) (- (random 200) 100) "black")
6         (rline (- n 1))
7       )))
8 (beside (rline 10) (rline 50) (rline 100) (rline 200))

```



52. Littéralement (`above A B`) équivaut à `A` est au dessus de `B`; ces fonctions de placement acceptent en réalité des listes de figures, permettant ainsi d'indiquer dans le cas de `above`, `A` au dessus de `B` au dessus de `C`...

Le programme suivant présente les différences entre placements avec et sans alignement :

```

1 (require 2htdp/image)
2 (beside (above (square 40 "solid" "blue")
3             (square 80 "solid" "gray")))
4       (square 60 "solid" "red"))

```



```

1 (require 2htdp/image)
2 (beside/align "bottom"
3   (above/align "right"
4     (square 40 "solid" "blue")
5     (square 80 "solid" "gray")))
6   (square 60 "solid" "red"))

```

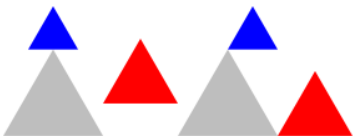


Les placements sont définis par rapport au rectangle englobant de chaque figure ; le programme ci-dessous reprend les placements des programmes précédents en utilisant des triangles :

```

1 (require 2htdp/image)
2 (beside
3   (beside
4     (above (triangle 40 "solid" "blue")
5             (triangle 80 "solid" "gray")))
6     (triangle 60 "solid" "red"))
7   (beside/align "bottom"
8     (above/align "right"
9       (triangle 40 "solid" "blue")
10      (triangle 80 "solid" "gray")))
11      (triangle 60 "solid" "red"))
12 )

```

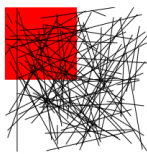


Le programme suivant présente un autre exemple d'utilisation des fonctions de superposition ; la fonction `rline2`, présentée ci-dessous, superpose aléatoirement n lignes dans un rectangle de dx sur dy par appel récursif de `rline2` empilant les appels à la fonction `add-line` ; le programme trace un dessin plaçant 100 lignes aléatoires tenant dans un carré de 200 par 200 au dessus d'un carré rouge de 100 par 100 ; comme les coordonnées (initiales et finales) des lignes varient de 0 à dx et de 0 à dy et selon la fonction `random`, les positions des lignes varient.

```

1 (require 2htdp/image)
2 (define (rline2 input n dx dy)
3   (if (= n 0)
4       input
5       (add-line (rline2 input (- n 1) dx dy)
6                 (random dx) (random dy)
7                 (random dx) (random dy) "black"))
8   ))
9 (overlay/xy (rline2 empty-image 100 200 200) 0 0
10  (square 100 "solid" "red"))

```

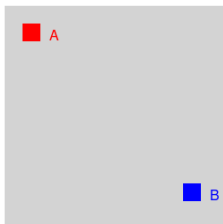


La fonction `text` permet d'ajouter du texte⁵³ dans un dessin, comme présenté dans le programme suivant :

```

1 (require 2htdp/image)
2 (underlay/xy (underlay/xy (underlay/xy (underlay/xy
3   (square 250 "solid" "lightgray")
4     20 20 (square 20 "solid" "red"))
5       50 20 (text "A" 16 "red"))
6     200 200 (square 20 "solid" "blue"))
7     230 200 (text "B" 16 "blue"))

```



53. La fonction `text/font` permet de préciser taille de font, famille et style, comme par exemple avec `(text/font "A" 16 "red" "times" 'script 'italic 'bold #t)`.

17.3 Polygones et couleurs

Les polygones sont définis par une séquence de paramètres : *données*, *mode* et *couleur* ; la partie *données* peut être composée d'une ou plusieurs valeurs ; la partie *mode* définit le remplissage du dessin qui est plein (*i.e.* pour “**solid**” ou **'solid**) ou filiforme (*i.e.* pour “**outline**” ou **'outline**) ; la fonction **rectangle** utilise comme *données* les valeurs de largeur et de hauteur ; la fonction **triangle** définit un triangle équilatéral avec pour unique donnée une taille de côté ; la figure 25 présente la liste des principaux polygones⁵⁴ possibles ; la figure 26 présente la liste des mots-clés⁵⁵ possibles les plus courants pour la partie *couleur*.

(triangle side-length mode color)	pour un triangle équilatéral
(right-triangle side-length1 side-length2 mode color)	pour un triangle rectangle
(isosceles-triangle side-length angle mode color)	pour un triangle isocèle
(square side-len mode color)	pour un carré
(rectangle width height mode color)	pour un rectangle

Fig. 25 – Principaux polygones possibles.

Tomato, Red, Pink, Brown, Orange, Gold, Yellow, Green, Turquoise, Cyan, Blue, Indigo, Purple, White, LightGray, Silver, Gray, DarkGray, Black

Fig. 26 – Principaux mots-clés possibles pour la partie *couleur*.

⁵⁴. Il est également possible de spécifier les formes des triangles en combinant angle et taille de côté à l'aide des fonctions **triangle/sss**, **triangle/ssa**, **triangle/aas**, **triangle/asa**, **triangle/saa**.

⁵⁵. Dont une liste plus complète est donnée ici : OrangeRed, **Tomato**, DarkRed, **Red**, Firebrick, Crimson, DeepPink, Maroon, IndianRed, MediumVioletRed, VioletRed, LightCoral, HotPink, PaleVioletRed, LightPink, RosyBrown, **Pink**, Orchid, LavenderBlush, Snow, Chocolate, SaddleBrown, **Brown**, DarkOrange, Coral, Sienna, **Orange**, Salmon, Peru, DarkGoldenrod, Goldenrod, SandyBrown, LightSalmon, DarkSalmon, **Gold**, **Yellow**, Olive, Burlywood, Tan, NavajoWhite, PeachPuff, Khaki, DarkKhaki, Moccasin, Wheat, Bisque, PaleGoldenrod, BlanchedAlmond, MediumGoldenrod, PapayaWhip, MistyRose, LemonChiffon, AntiqueWhite, Cornsilk, LightGoldenrodYellow, OldLace, Linen, LightYellow, SeaShell, Beige, FloralWhite, Ivory, **Green**, LawnGreen, Chartreuse, GreenYellow, YellowGreen, MediumForestGreen, DarkOliveGreen, DarkSeaGreen, Lime, DarkGreen, LimeGreen, ForestGreen, SpringGreen, MediumSpringGreen, SeaGreen, MediumSeaGreen, Aquamarine, LightGreen, PaleGreen, MediumAquamarine, **Turquoise**, LightSeaGreen, MediumTurquoise, Honeydew, MintCream, RoyalBlue, DodgerBlue, DeepSkyBlue, CornflowerBlue, SteelBlue, LightSkyBlue, DarkTurquoise, **Cyan**, Aqua, DarkCyan, Teal, SkyBlue, CadetBlue, DarkSlateGray, LightSlateGray, SlateGray, LightSteelBlue, LightBlue, PowderBlue, PaleTurquoise, LightCyan, AliceBlue, Azure, MediumBlue, DarkBlue, MidnightBlue, Navy, **Blue**, **Indigo**, BlueViolet, MediumSlateBlue, SlateBlue, **Purple**, DarkSlateBlue, DarkViolet, DarkOrchid, MediumPurple, MediumOrchid, Magenta, Fuchsia, DarkMagenta, **Violet**, Plum, Lavender, Thistle, GhostWhite, **White**, WhiteSmoke, Gainsboro, **LightGray**, **Silver**, **Gray**, **DarkGray**, DimGray, **Black**.

17.4 Sauvegarder un dessin dans un fichier

La fonction `save-image` permet de sauvegarder un dessin dans un fichier au format PNG⁵⁶ (le format SVG⁵⁷ est également accessible en utilisant la fonction `save-svg-image`); en réutilisant la fonction `rline2` de la section 17.2, le programme ci-dessous présente la sauvegarde de deux images : la première est sauvegardée telle que et produit une image avec un fond partiellement transparent⁵⁸, la deuxième est sauvegardée en ajoutant un fond gris-clair sur l'ensemble de l'image; les images résultantes sont présentées en figure 27 et figure 28; un appel à la fonction `save-image` retourne `true` en fin d'exécution.

```
1 (require 2htdp/image)
2 (save-image
3   (overlay/xy (rline2 empty-image 100 200 200)
4     0 0
5     (square 100 "solid" "red"))
6   "image1.png")
7 (save-image
8   (overlay/xy
9     (overlay/xy (rline2 empty-image 100 200 200)
10      0 0
11      (square 100 "solid" "red"))
12     0 0
13     (square 200 "solid" "lightgray"))
14   "image2.png")
```

```
#t
#t
```

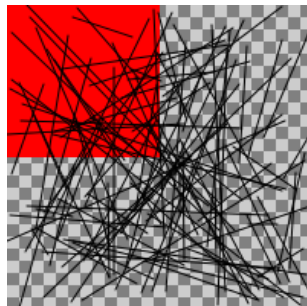


Fig. 27 – image1.png.

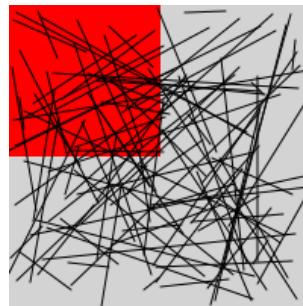


Fig. 28 – image2.png.

56. Portable Network Graphics, format ouvert orienté image numérique.

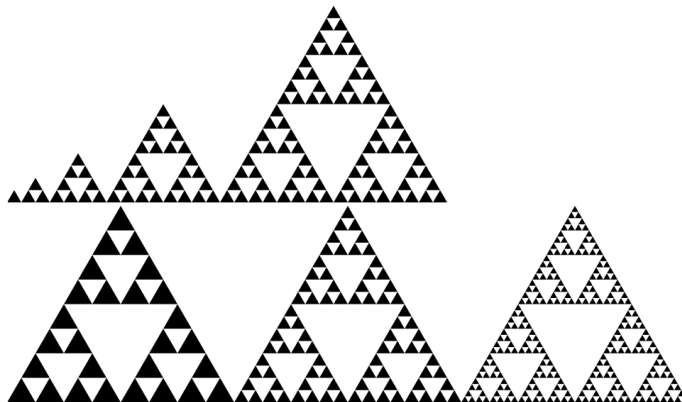
57. Scalable Vector Graphics, format orienté dessin vectoriel.

58. Un fond transparent est usuellement représenté par un pavage carré alternant gris foncé et gris clair.

17.5 Triangle et tapis de Sierpiński

Le triangle de Sierpiński est une fractale obtenue par empilement de triangles ; son principe est de considérer le triangle formé par l'ensemble de la figure, de diviser sa taille par deux, et d'empiler trois de ces triangles réduits pour former un nouveau triangle contenant un creux en son centre ; la répétition de ce principe produit un triangle de plus en plus creux ; le premier alignement de dessins présente des empilements de triangles de côté 20 ; le deuxième alignement de dessins présente des triangles de plus en plus creux

```
1 (require 2htdp/image)
2 (define (sierpinski0 d) (triangle d "solid" "black"))
3 (define (sierpinski n d)
4   (if (= n 0) (sierpinski0 d)
5       (above (sierpinski (- n 1) d)
6              (beside (sierpinski (- n 1) d) (sierpinski (- n 1) d))
7              )))
8 (beside/align "bottom"
9   (sierpinski 0 20) (sierpinski 1 20)
10  (sierpinski 2 20) (sierpinski 3 20) (sierpinski 4 20))
11 (beside (sierpinski 3 40) (sierpinski 4 20) (sierpinski 5 10))
```

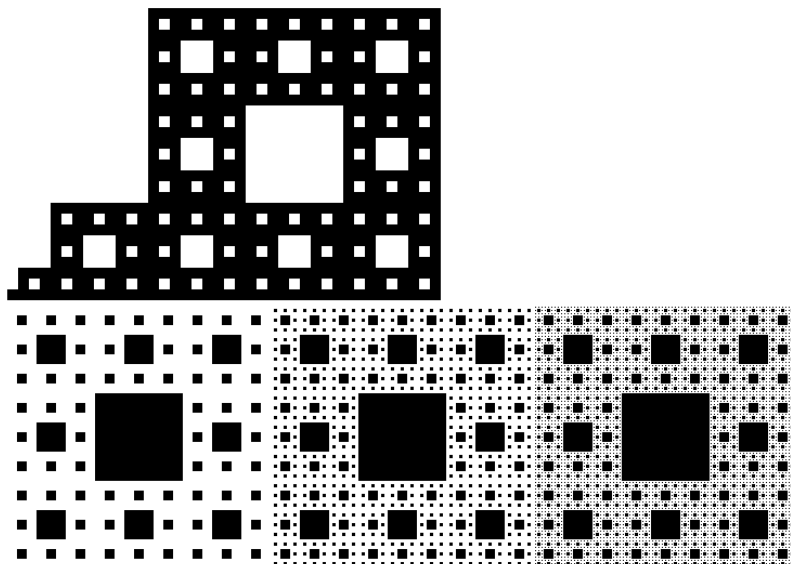


Le tapis de Sierpiński est une fractale obtenue par empilement de carrés; son principe est de considérer un carré comme une grille 3×3 dont on supprime la case centrale, et de répéter ce principe sur les autres cases; la répétition de ce principe produit un carré de plus en plus creux; le premier alignement de dessins présente des empilements de carrés de côté 10; le deuxième alignement de dessins présente des carrés de plus en plus creux.

```

1 (require 2htdp/image)
2 (define (sierpinski0 d c) (square d "solid" c))
3 (define (sierp-line0 n d col0 col1)
4   (beside (sierpinski n d col0 col1)(sierpinski n d col0 col1)
5     (sierpinski n d col0 col1)))
6 (define (sierp-line1 n d col0 col1)
7   (beside (sierpinski n d col0 col1)(sierpinski n d col1 col1)
8     (sierpinski n d col0 col1)
9   ))
10
11 (define (sierpinski n d col0 col1)
12   (if (= n 0) (sierpinski0 d col0)
13     (above (sierp-line0 (- n 1) d col0 col1)
14       (sierp-line1 (- n 1) d col0 col1)
15       (sierp-line0 (- n 1) d col0 col1)
16     )))
17 (beside/align "bottom" (sierpinski 0 10 "black" "white")
18   (sierpinski 1 10 "black" "white") (sierpinski 2 10 "black" "white")
19   (sierpinski 3 10 "black" "white"))
20 (beside (sierpinski 3 9 "white" "black")
21   (sierpinski 4 3 "white" "black") (sierpinski 5 1 "white" "black"))

```



17.6 Dessiner des images pixel par pixel

L'utilisation du package `2htdp/image` permet de manipuler des images sous forme matricielle; une image est ici une matrice de pixels; pour chaque pixel, on a quatres composantes `rgba` (*i.e.* rouge, vert, bleu et alpha pour l'opacité de la couleur); la fonction `bitmap` permet de charger une image `png` ou `jpg`.

```
1 (require 2htdp/image)
2 (bitmap "/home/n/nuages.png")
```



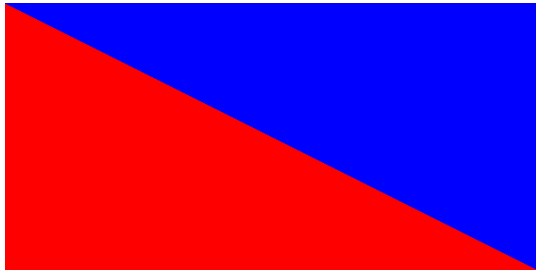
Les fonctions `image-width` et `image-height` permettent de connaître largeur et hauteur d'une image; la fonction `image->color-list` retourne la liste des pixels d'une image; la fonction `color-list->image` permet de définir une image à partir d'une liste de pixels; pour chaque élément de la liste de type `color-list`, on obtient les valeurs `r`, `g`, `b` et `a` à l'aide des fonctions `color-red`, `color-green`, `color-blue`, `color-alpha`; considérant que passer une image en niveaux de gris équivaut à avoir sur chacune des composantes la moyenne des composantes `rgb` de l'image de couleur, on peut définir la fonction `image-color->gray` comme suit :

```
1 (require 2htdp/image)
2 (define (image-color->gray img)
3   (define (f L)
4     (if (empty? L) empty
5         (let* ([pix (first L)]
6               [sum_pix (+ (color-red pix)
7                           (color-green pix)(color-blue pix))]
8               [gpix (round (/ sum_pix 3))])
9             (cons (color gpix gpix gpix (color-alpha pix)) (f (rest L))))))
11  (color-list->bitmap
12    (f (image->color-list img))
13    (image-width img) (image-height img)
14  ))
15 (image-color->gray (bitmap "/home/n/nuages.png"))
```



On pourra définir une fonction de construction d'une image conformément à une fonction `f` dépendant de `x` et `y` comme suit :

```
1 (require 2htdp/image)
2 (define (mk-image-line y fun width)
3   (define (f x)
4     (if (= x width) empty
5         (cons (fun x y) (f (add1 x)))))
6   (f 0)
7 )
8
9 (define (mk-image fun width height)
10  (define (f y L)
11    (if (= y height) (color-list->bitmap L width height)
12        (f (add1 y) (append L (mk-image-line y fun width)))))
13  (f 0 empty)
14 )
15
16 (define (rgb-fun x y)
17   (if (= y 0) (color 0 0 255)
18       (if (< (/ x y) 2)
19           (color 255 0 0)
20           (color 0 0 255))))
21
22 (mk-image rgb-fun 600 300)
```



Il est également possible de charger une image à partir d'une url avec la fonction `bitmap/url` ou de charger une image à partir d'un fichier postscript avec la fonction `bitmap/file`.

On pourra également utiliser le prédicat `image?` pour savoir si un élément est une image et le prédicat `image-color?` pour savoir si un élément est un pixel.

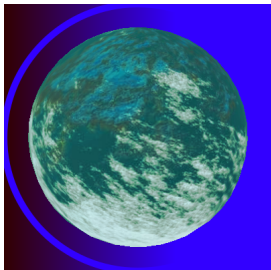
On pourra superposer une image vectorielle et les primitives de dessin précédemment vues.

```
1 (overlay (circle 145 "solid" "gold") (mk-image rgb-fun 600 300))
```



On pourra superposer un dégradé du noir vers le bleu horizontalement, des primitives de dessin et une image de la planète Namek.

```
1 (define (clamp pix-val) (max 0 (min 255 pix-val)))
2 (define (grad1 x y) (clamp x))
3 (define (rgb-fun x y) (color 0 0 (grad1 x y)))
4 (underlay
5   (underlay
6     (mk-image rgb-fun 400 400)
7     (circle 200 "outline"
8       (make-pen (color 50 0 255) 10 "solid" "round" "round")
9     ))
10  (bitmap "/home/n/namek.png"))
11 )
```

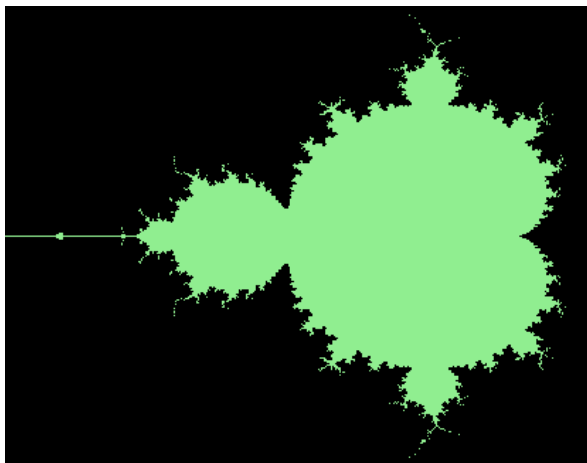


La fonction `clamp` permet de borner les valeurs dans l'intervalle $[0; 255]$; la fonction `grad1` réalise un dégradé en fonction de `x` ; la fonction `rgb-fun` retourne une couleur dégradée dans le bleu ; la fonction `make-pen` permet de grossir le tracé du cercle bleu.

17.7 Dessiner des fractales

Avec la fonction `mk-image`, on peut dessiner des fractales en utilisant les nombres complexes ; l'ensemble de Mandelbrot est défini par itération d'un polynôme de la forme $z^2 + c$; partant d'un nombre complexe z_0 , on calcule un complexe z_n tel que $z_n = z_{n-1}^2 + c$; après n itérations, on regarde la distance à l'origine ; selon une distance max m , les positions de départ z_0 ayant un z_n situé à plus de m de l'origine sont d'une couleur et celles ayant un z_n situé à moins de m de l'origine sont d'une autre couleur.

```
1 (define maxite 20)
2 (define maxdist 2.0)
3 (define (dist z)
4   (sqrt (+
5     (* (real-part z) (real-part z))
6     (* (imag-part z) (imag-part z))
7   )))
8 (define (f1 z zi) (+ (* zi zi) z))
9 (define (ite-in x fun)
10  (define (f xi ite)
11    (if (> ite maxite) ite
12      (let ([nx (fun x xi)])
13        (if (> (dist nx) maxdist) ite
14          (f nx (add1 ite))
15        ))
16    ))
17  (f x 0))
18 (define (f x y)
19  (let ([nx (/ (- x 320) 160.0)]
20        [ny (/ (- y 240) 140.0)])
21    (if (< (ite-in (make-rectangular nx ny) f1) maxite)
22      "black" "lightgreen")
23  ))
24 (mk-image f 640 480)
```

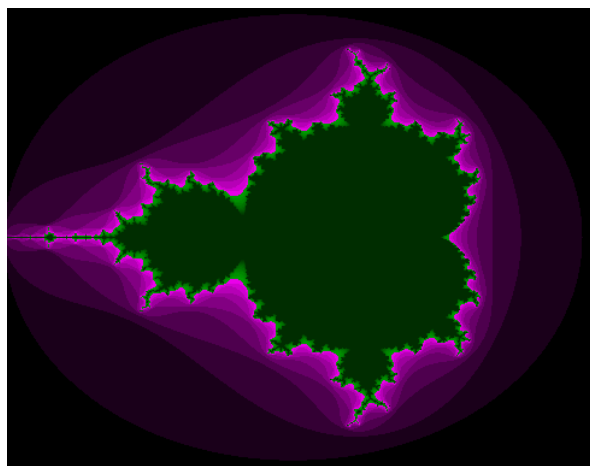


Pour ajouter un dégradé sur les valeurs inférieures à 10 et un dégradé sur les valeurs supérieures à 10, on modifie la fonction `f` en testant le nombre d'itérations pour être au-delà de la distance `maxdist`.

```

1 (define (f x y)
2   (let* ([nx (/ (- x 320) 160.0)]
3         [ny (/ (- y 240) 140.0)]
4         [nite (ite-in (make-rectangular nx ny) f1))])
5     (if (< nite 10)
6         (color (* nite 26) 0 (* nite 26))
7         (color 0 (- 255 (* nite 10)) 0))
8   ))
9 (mk-image f 640 480)

```



En modifiant la fonction `f1` (ligne 8 page 134), on obtient d'autres fractales :

- Le lapin de Douadi (figure 29)
- Une fractale de l'ensemble de Julia (figure 30)
- Une fractale de l'ensemble de Julia (figure 31)
- Un dragon (figure 32)
- Un dragon (figure 33)
- Des îles (figure 34)
- Un trèfle à quatre feuilles (figure 35)
- Une dendrite (figure 36)

Pour éviter de réécrire la fonction `ite-in` (lignes 9 à 17, page 134) qui utilise une fonction `(fun x xi)` correspondant à un appel de `(f z zi)`, on réutilise le prototype `(f1 z zi)` dans les fonctions `f1` suivantes même si ces fonctions `f1` ne dépendent que de `zi` et ignorent `z`.

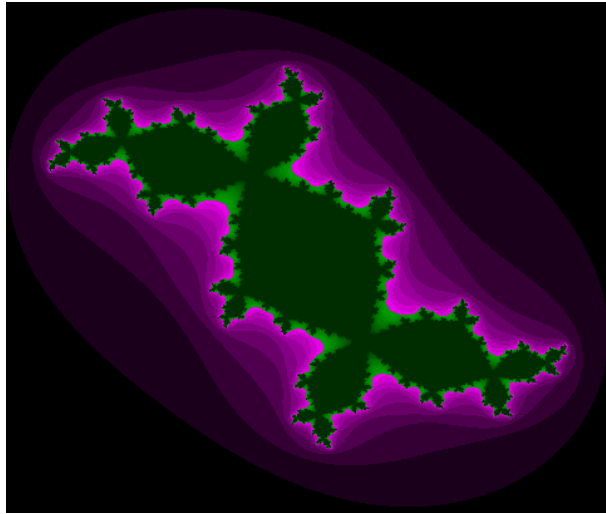


Fig. 29 – Un lapin avec `(define (f1 z zi) (+(* zi zi) -0.125-0.758i))`.

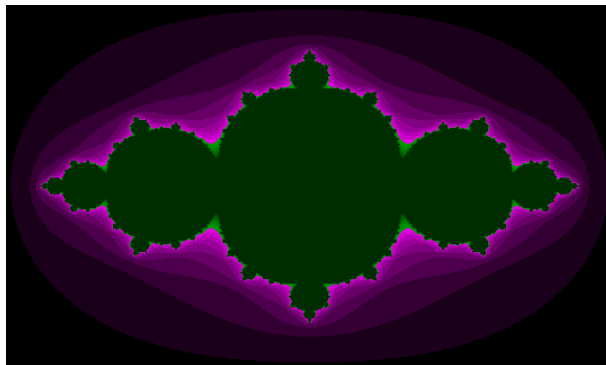


Fig. 30 – Une Julia avec `(define (f1 z zi) (+(* zi zi) -0.75+0.0i))`.

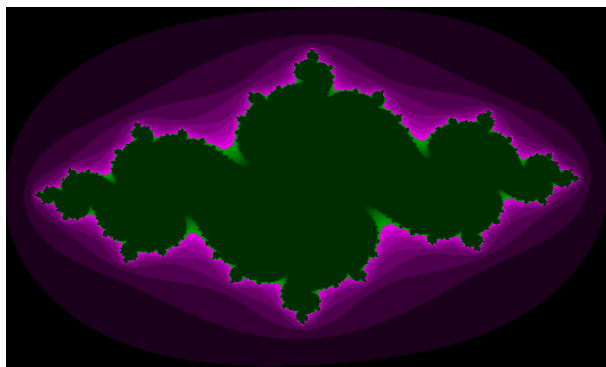


Fig. 31 – Une Julia avec `(define (f1 z zi) (+(* zi zi) -0.75+0.1i))`.

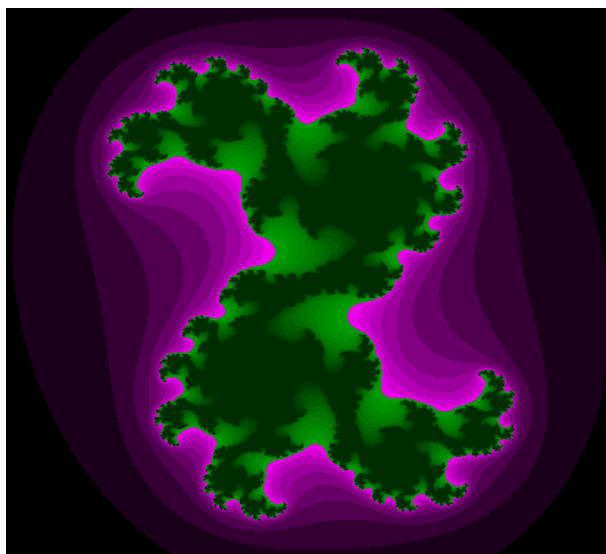


Fig. 32 – Un dragon avec `(define (f1 z zi) (+(* zi zi) 0.4-0.192i))`.

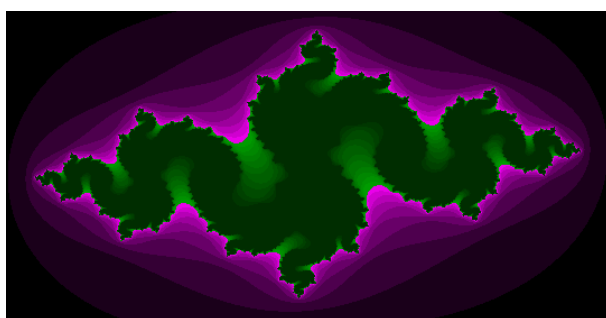


Fig. 33 – Un dragon avec `(define (f1 z zi) (+(* zi zi) -0.8+0.168i))`.

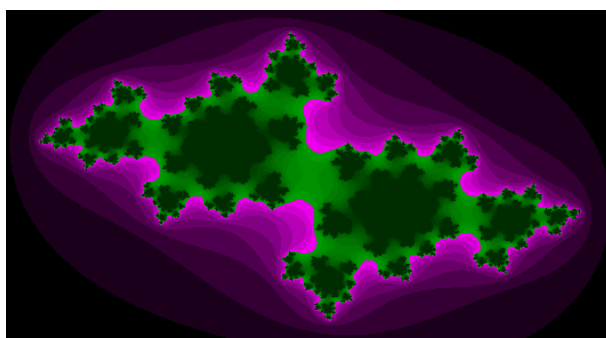


Fig. 34 – Des iles avec `(define (f1 z zi) (+(* zi zi) -0.683-0.408i))`.

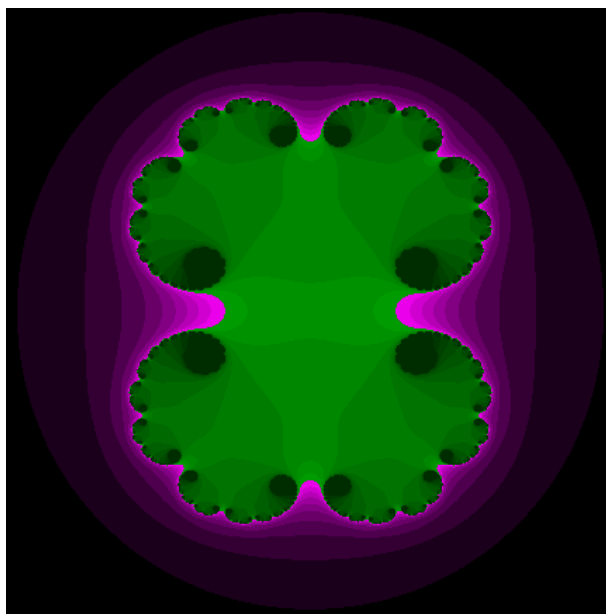


Fig. 35 – Un trèfle avec `(define (f1 z zi) (+(* zi zi) 0.3))`.

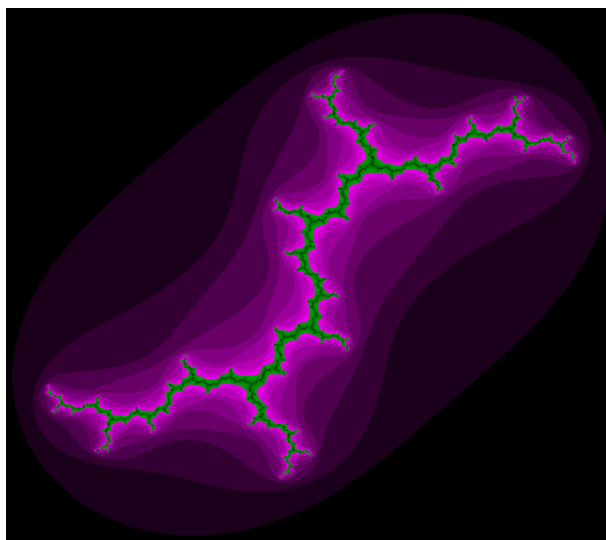


Fig. 36 – Un dendrite avec `(define (f1 z zi) (+ (* zi zi) 0.0+i))`.

17.8 Dessiner avec des tortues

Les tortues présentent une interface ludique de dessin au travers d'un animal qu'il s'agit de mouvoir⁵⁹ ; la figure 37 présente la liste des principales fonctions de contrôle des tortues⁶⁰ ; initialement les tortues sont centrées et orientées à droite dans la fenêtre ; le triangle orange du dessin montre la position finale de la tortue.

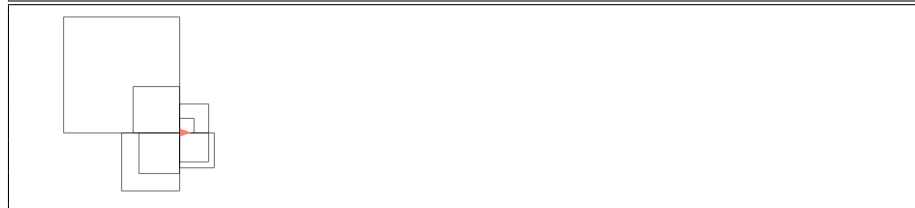
(turtles #t)	Activer et dessiner les tortues
(draw n)	Tracer un trait de n pixels
(erase n)	Effacer n pixels
(move n)	Déplacer la tortue sans tracé
(turn theta)	Tourner de θ degrés
(turn/radians theta)	Tourner de θ radians
(clear)	Tout effacer
(home)	Replacer la tortue à sa position initiale
(save-turtle-bitmap "img.png" "png")	Sauvegarder le dessin dans le fichier <code>img.png</code>

Fig. 37 – Principales fonctions de contrôle des tortues.

La fonction `begin` permet de définir une séquence⁶¹ de fonctions à appeler, permettant ainsi de définir des séquences de déplacements.

```

1 (require graphics/turtles)
2 (turtles #t)
3 (define M move)
4 (define T turn)
5 (define D draw)
6 (define (draw-sqr side) (begin (D side)(T 90)(D side)(T 90)
7                               (D side)(T 90)(D side)))
8 (draw-sqr 50)(draw-sqr 60)(draw-sqr 70)(draw-sqr 80)
9 (draw-sqr 25)(draw-sqr 50)(draw-sqr 100)(draw-sqr 200)
```



59. L'utilisation des tortues est présentée par S. Papert dans les années 60 au MIT pour enseigner la programmation aux enfants.

60. Concernant la sauvegarde, les formats jpeg, xbm, xpm et bmp sont également disponibles ; la cohérence entre nom et extension est à l'attention du programmeur ; pour un nom du fichier global dans le système de fichiers, on écrira `/tmp/img.png` sous Linux.

61. Plus généralement (`equal? '(0) '(1)`) `'(1)` affiche `#t` ; elle permet ici d'exécuter les deux fonctions `'(0)` et `'(1)` ; la valeur retournée par `begin` est la valeur retournée par la dernière fonction appelée, ici `'(1)` et c'est pourquoi étant égale à `'(1)`, il s'affiche `#t`.

17.9 Flocon et île de Koch

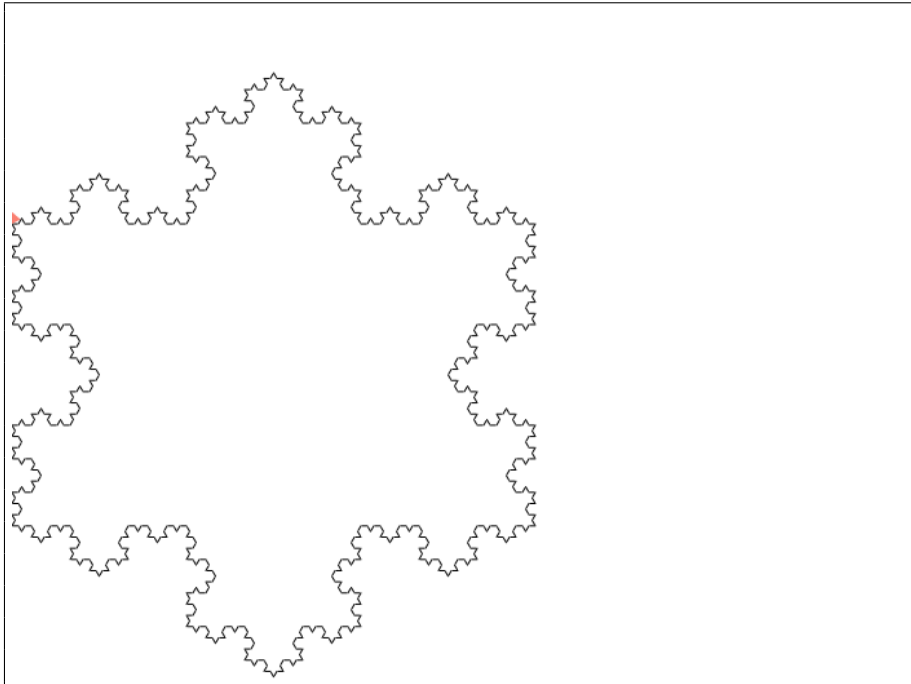
Les tortues permettent de dessiner des fractales, telles que le flocon de Koch qui se construit en partant d'un triangle et en subdivisant les segments en pics ; le programme suivant présente le principe de cette subdivision en pics ; pour dessiner les segments du flocon de Koch, il faut dessiner un segment, tourner de 60° , dessiner un segment, tourner de -120° , dessiner un segment, tourner de 60° et dessiner un dernier segment (ce qui crée des pics).

```
1 (require graphics/turtles)
2 (turtles #t)
3 (define M move)
4 (define T turn)
5 (define D draw)
6 (define (draw-koch side) (begin (D side)(T 60)(D side)(T -120)
7                                (D side)(T 60)(D side)))
8 (define (draw-koch2 side n)
9   (local ([define (sub-draw) (draw-koch2 (/ side 2) (- n 1))])
10    (if (= n 0)
11        (draw-koch side)
12        (begin (sub-draw) (T 60)
13                (sub-draw) (T -120)
14                (sub-draw) (T 60)
15                (sub-draw))
16    )))
17 (M (* -1 (/ turtle-window-size 2)))
18 (draw-koch2 20 0)
19 (draw-koch2 20 1)
20 (draw-koch2 15 2)
21 (draw-koch2 15 3)
22 (draw-koch2 15 4)
```



Ainsi pour dessiner un flocon de Koch complet, il faut intégrer le dessin d'un triangle (en réutilisant les fonctions `draw-koch` et `draw-koch2`).

```
1 (require graphics/turtles)
2 (turtles #t)
3 (define (draw-koch3 side n)
4   (begin (draw-koch2 side n)(T -120)
5         (draw-koch2 side n)(T -120)
6         (draw-koch2 side n)))
7 (M (* -1 (/ turtle-window-size 2)))
8 (draw-koch3 40 3)
```

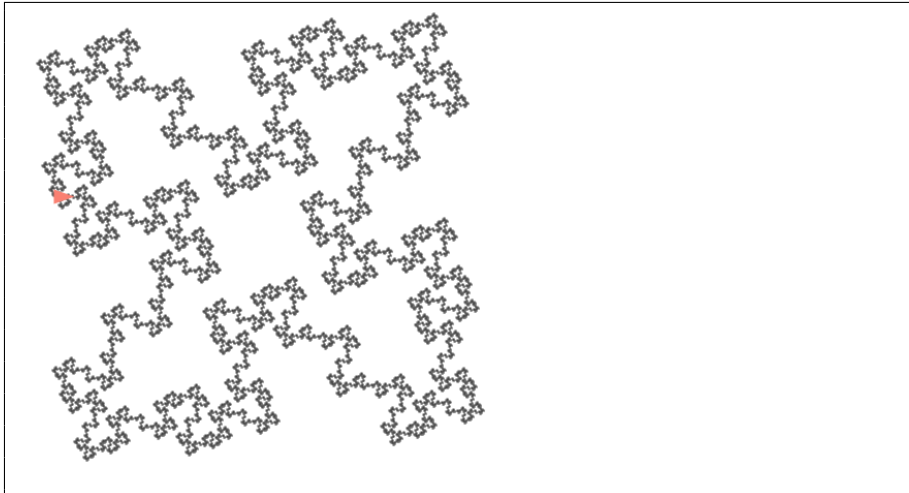


L'île de Koch est la seconde variante quadratique⁶² du flocon de Koch.

```

1 (require 2htdp/image)
2 (require graphics/turtles)
3 (turtles #t)
4 (define M move)
5 (define T turn)
6 (define D draw)
7 (define (koch-island-step d)
8   (begin (D d)(T 90)(D d)(T -90)(D d)(T -90)
9     (D (* 3 d))(T 90)(D d)(T 90)(D d)(T -90)(D d)))
10 (define (koch-island-loop d n)
11   (define (sub-koch) (koch-island-loop d (- n 1)))
12   (if (= n 0)
13     (koch-island-step d)
14     (begin (sub-koch) (T 90)
15       (sub-koch) (T -90)
16       (sub-koch) (T -90)
17       (sub-koch) (sub-koch)
18       (sub-koch) (T 90)
19       (sub-koch) (T 90)
20       (sub-koch) (T -90)
21       (sub-koch)))
22   ))
23 (define (koch-island d n)
24   (koch-island-loop d n)(T 90)(koch-island-loop d n)(T 90)
25   (koch-island-loop d n)(T 90)(koch-island-loop d n)(T 90))
26 (M (* -0.8 (/ turtle-window-size 2)))
27 (koch-island 1 3)

```



62. L'assimilation à une équation du second degré vient de la croissance de la somme des longueurs des segments de la fractale qui est égale au carré de sa valeur à la génération précédente; autrement dit, la longueur de son contour est égale au carré de sa longueur à la génération précédente; pour obtenir une île de Koch de taille fixe, on pourra fixer la longueur d'un segment à $l_0/4^n$ pour une longueur initiale l_0 à la génération 0 et pour une génération n .

17.10 Interprètes et L-systèmes

Les fractales et les L-systèmes⁶³ ont donné lieu à un système particulier de réécriture synthétique; ils permettent de produire des suites de symboles qui sont interprétables par des déplacements de tortues; la figure 38 présente les symboles couramment utilisés dans les L-systèmes.

F	pour avancer d'un pas
+	pour tourner dans le sens horaire d'un angle prédéfini
-	pour tourner dans le sens anti-horaire
 	pour faire demi-tour
[pour sauvegarder sa position courante
]	pour restaurer la dernière position sauvegardée

Fig. 38 – Symboles utilisés dans les L-systèmes.

A partir de ces symboles, deux ensembles définissent les L-systèmes :

- une suite initiale de symboles, notée w
- des règles de réécriture des symboles permettant de faire évoluer la suite initiale de génération en génération, notées p

Pour reprendre l'exemple du flocon de Koch présenté dans la section 17.9, il est défini par le L-système suivant :

- $w : \mathbf{F+F+F+F}$
- $p : \mathbf{F} \leftarrow \mathbf{F+F-F-FF+F+F-F}$

Avec n le numéro de la génération résultante, on a :

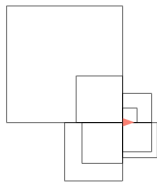
- pour $n = 0$, $\mathbf{F+F+F+F}$
- pour $n = 1$, on obtient la suite de 60 symboles suivante
 $\mathbf{F+F-F-FF+F+F-F} +$
 $\mathbf{F+F-F-FF+F+F-F} +$
 $\mathbf{F+F-F-FF+F+F-F} +$
 $\mathbf{F+F-F-FF+F+F-F}$
- pour $n = 2$, on obtient une suite de 475 symboles commençant par
 $\mathbf{F+F-F-FF+F+F-F} +$
 $\mathbf{F+F-F-FF+F+F-F} -$
 $\mathbf{F+F-F-FF+F+F-F} -$
 $\mathbf{F+F-F-FF+F+F-F} \mathbf{F+F-F-FF+F+F-F} + \dots$

Après construction de la suite à la génération désirée, il s'agit d'interpréter la suite créée en tracés réalisés à l'aide de déplacements de tortues; il s'agit donc de considérer les éléments à tracer comme des listes d'éléments et de définir une fonction **interprete** qui réalisera le tracé correspondant à un élément et une fonction **process** qui appelle la fonction **interprete** pour chacun des éléments de la liste.

⁶³. Les systèmes de Lindenmayer modélisent le développement des plantes et des bactéries; Leur système de réécriture est une forme de grammaire générative permettant de produire des dessins graphiquement complexes et réalistes.

Pour reprendre l'exemple de la section 17.8, il s'agit d'avoir des listes de taille de côté de carré dont on réalise le tracé en séquence ; la fonction **process** prend les éléments de cette liste un à un (*i.e.* soit des tailles de côté de carré) et réalise les dessins correspondants en appelant la fonction **interprete** pour chaque élément ; pour une valeur passée en paramètre, la fonction **interprete** dessine ainsi un carré avec cette valeur pour taille de côté ; ainsi le programme de la section 17.8 se présente comme suit :

```
1 (require 2htdp/image)
2 (require graphics/turtles)
3 (turtles #t)
4 (define (interprete d)
5   (draw d)(turn 90)(draw d)(turn 90)(draw d)(turn 90)(draw d))
6 (define (process L)
7   (if (= (length L) 1) (interprete (first L))
8       (begin (interprete (first L)) (process (rest L)))))
9   ))
10 (define L '(50 60 70 80 25 50 100 200))
11 (process L)
```



Pour reprendre l'exemple de la section 17.9, il s'agit d'un L-système défini par :

- $w : F$
- $p : F \leftarrow F+F--F+F$

Il faut modifier la fonction `process`, redéfinir la fonction `interprete` et définir les 3 fonctions génératives `develop`, `develop-elt` et `generate`; la fonction `process` appelle la fonction `interprete` en passant le symbole et une taille (utile dans le cas du symbole `F`) ; si le symbole `e` à interpréter est inconnu (*i.e.* différent de `F`, `+` et `-`), la tortue ne se déplace pas (car exécutant (`turn 0`)); la fonction `develop` appelle `develop-elt` pour chaque symbole de la liste; la fonction `develop-elt` applique la transformation défini par p pour le symbole `F` et recopie simplement les autres symboles; la fonction `generate` produit les générations d'indice n .

```

1 (require graphics/turtles)
2 (turtles #t)
3 (define (interprete e d)
4   (cond
5     [(equal? e #\F) (draw d)]
6     [(equal? e #\+) (turn 60)]
7     [(equal? e #\-) (turn -60)]
8     [else (turn 0)])
9   ))
10 (define (process L d)
11   (if (= (length L) 1) (interprete (first L) d)
12     (begin (interprete (first L) d) (process (rest L) d)))
13   ))
14 (define (develop L)
15   (if (empty? L) L
16     (append (develop-elt (first L)) (develop (rest L))))
17   ))
18 (define (develop-elt e)
19   (cond
20     [(equal? e #\F) (list #\F #\+ #\F #\- #\- #\F #\+ #\F)]
21     [else (list e)])
22   ))
23 (define (generate L n)
24   (if (= n 0) (develop L)
25     (develop (generate L (- n 1))))
26   ))
27 (move (* -1 (/ turtle-window-size 2)))
28 (define start (list #\F))
29 (process (generate start 0) 20)
30 (process (generate start 1) 10)
31 (process (generate start 2) 5)
32 (process (generate start 3) 2)

```



Pour reprendre l'exemple de la section 17.9, il s'agit d'un L-système défini par :

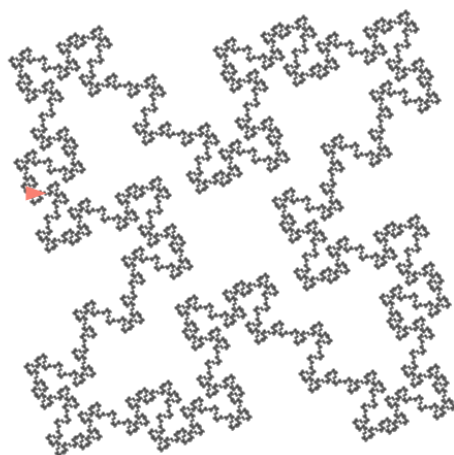
- $w : F+F+F+F+$
- $p : F \leftarrow F+F-F-FFF+F+F-F$

Par soucis de réutilisation pour les deux exemples suivants, la fonction `interprete` est redéfinie pour interpréter le nouveau symbole `f` ; les angles sont également modifiés (de 60 degrés dans l'exemple précédent à 90 degrés ici) ; la fonction `develop-elt` est redéfinie conformément à p et une liste `start` est définie conformément à w .

```

1 (require graphics/turtles)
2 (turtles #t)
3 (define (interprete e d)
4   (cond
5     [(equal? e #\F) (draw d)]
6     [(equal? e #\f) (move d)]
7     [(equal? e #\-) (turn -90)]
8     [(equal? e #\+) (turn 90)]
9     [else (turn 0)]
10  ))
11 ;; ... (define (process L d) ...
12 ;; ... (define (develop L) ...
13 (define (develop-elt e)
14   (cond
15     [(equal? e #\F)
16      (list #\F #\+ #\F #\+ #\F #\+ #\F #\+ #\F #\+ #\F #\+
17            #\F #\+ #\F #\+ #\F)]
18     [else (list e)]
19   ))
20 ;; ... (define (generate L n) ...
21 (move (* -0.8 (/ turtle-window-size 2)))
22 (define start (list #\F #\+ #\F #\+ #\F #\+ #\F #\+))
23 (process (generate start 3) 1)

```



Avec le symbole **f**, les fractales discontinues apparaissent et il s'agit de maintenir une distance appropriée à chaque génération entre les éléments⁶⁴ comme pour le L-système définit par :

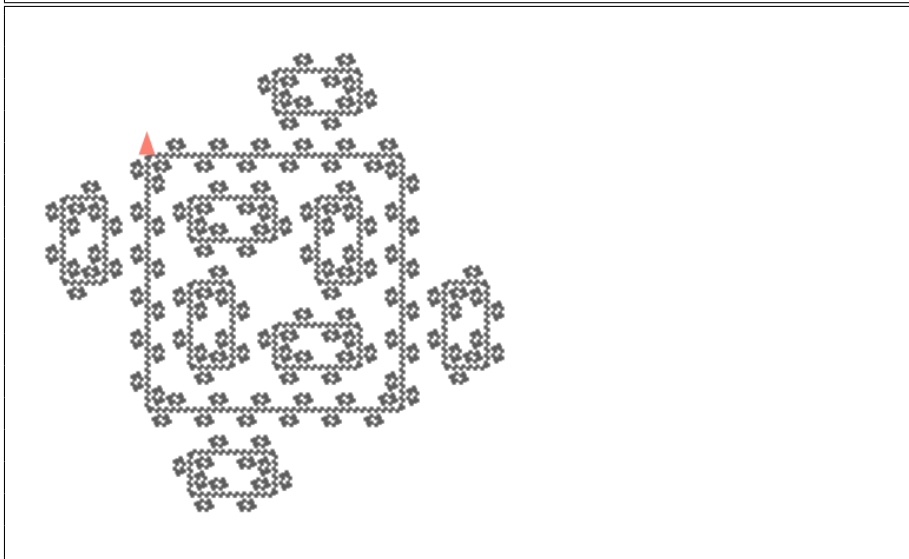
- $w : F-F-F-F$
- $p1 : F \leftarrow F-f+FF-F-FF-Ff-FF+f-FF+F+FF+Ff+FFF$
- $p2 : f \leftarrow fffff$

Il suffit maintenant de redéfinir la fonction **develop-elt** conformément à $p1$ et $p2$ et la définition de la liste **start** conformément à w .

```

1 (require graphics/turtles)
2 (turtles #t)
3 ;; ... (define (interprete e d) ...)
4 ;; ... (define (process L d) ...)
5 ;; ... (define (develop L) ...)
6 (define (develop-elt e)
7   (cond
8     [(equal? e #\F)
9      (list #\F #\- #\f #\+ #\F #\F #\- #\F #\- #\F #\F
10            #\- #\F #\f #\- #\F #\F #\+ #\f #\- #\F #\F
11            #\+ #\F #\+ #\F #\F #\+ #\F #\f
12            #\+ #\F #\F #\F)]
13     [(equal? e #\f)
14      (list #\f #\f #\f #\f #\f #\f)]
15     [else (list e)])
16   ))
17 ;; ... (define (generate L n) ...)
18 (move (* -0.8 (/ turtle-window-size 2)))
19 (define start (list #\F #\- #\F #\- #\F #\- #\F))
20 (process (generate start 2) 1)

```



64. Dans le cas de ce L-système, nommé des îles et des lacs par B. Mandelbrot.

En ajoutant des symboles impliqués dans la phase de génération et sans conséquence dans la phase d'interprétation (ici l et r , il est possible de définir un L-système correspondant à la courbe du dragon :

— $w : Fl$
 — $p1 : l \leftarrow l+rF+$
 — $p2 : r \leftarrow -Fl-r$

Dans la phase d'interprétation, les symboles l et r sont traduits par (turn 0).

```

1 (require graphics/turtles)
2 (turtles #t)
3 ;; ... (define (interprete e d) ...)
4 ;; ... (define (process L d) ...)
5 ;; ... (define (develop L) ...)
6 (define (develop-elt e)
7   (cond [(equal? e #\l)
8         (list #\l #\+ #\r #\F #\+)]
9         [(equal? e #\r)
10        (list #\- #\F #\l #\- #\r)]
11        [else (list e)])
12   ))
13 ;; ... (define (generate L n) ...)
14 (define start (list #\F #\l))
15 (process (generate start 13) 2)

```

