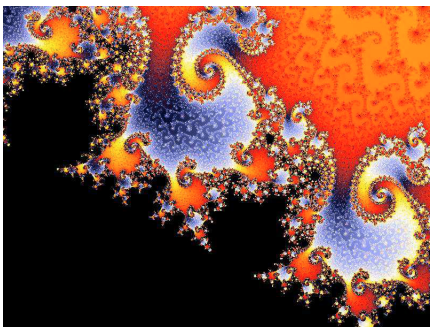
TP 2 : les Fractales, c'est Fatal

notions : Classes, objets, encpasulation, constructeurs, accesseurs, modifieurs



Remarque introductive : nous travaillerons sous Atom et en ligne de commande (Terminal sous Mac ou Linux ou PowerShell sous Windows), plutôt que sous Eclipse.

1 Une classe Image

Cette première partie met en place une classe `Image` qui permet de représenter et manipuler une image en mémoire.

Une image est essentiellement un tableau 2D de pixels, chaque pixel étant encodé par sa couleur.

Vous utiliserez le codage nommé ARGB (en anglais Alpha Red Green Blue) pour coder les couleurs. Dans ce codage, la couleur d'un pixel est faite sur un entier 32 bits (`int`), qui regroupe 4 valeurs sur 8 bits : l'opacité (ou "canal alpha"), le rouge, le vert, le bleu.

Chacun des 4 octets de l'entier code donc l'intensité d'un canal de couleur.

Les 8 bits de poids faible codent le Bleu, les bits 8 à 15 codent le vert et les bits 16 à 23 codent le rouge.

Les bits de poids fort sont réservés à l'opacité (canal Alpha), qu'on règlera toujours à 255, soit 0xFF en hexadécimal, qui signifie opaque.

Par exemple :
- Dans le canal rouge, 0 signifie pas de rouge et 255 un tout rouge qui fait mal aux yeux, aie, ça fait mal.
- Le noir opaque s'écrit `0xFF000000` en hexadécimal ou encore 255 << 24.
- Le rouge Phelma se code 255 (ou `0xFF` en hexadécimal) en canal alpha, 191 (ou `0xBF`) en Rouge et rien dans les autres composantes, soit `0xFFBF0000` en hexadécimal ou encore 255 << 24|191 << 16.
- pour stocker dans un entier la valeur encodant un bleu cyan (mélange de vert et bleu saturés), on pourra écrire :

```
int couleur = 0xFF00FFFF ;
```

ou : `int couleur = 255<<24 | 255<<8 | 255;`

1.1 La classe `ImageToolkit` fournie

Dans l'archive à télécharger sur le site, nous fournissons :
— Une classe `Complexe.java`
— Un paquetage (sous-répertoire) `phelma`, qui contient une classe `ImageToolkit` (fichier `ImageToolkit.java`)
— Un programme de test de la classe `ImageToolkit`.

Les méthodes utiles de la classe `ImageToolkit` sont les suivantes :
— `public static void displayPixelMatrix(int [][]im)` : crée une nouvelle fenêtre à l'écran et affichage le tableau 2D de pixels `im` dans cette fenêtre.
— `public static boolean savePixelMatrix(String nom, int [][]im)` : sauve le tableau 2D de pixels `im` dans un fichier de nom `nom` au format png. L'extension `".png"` est automatiquement ajoutée au nom. Cette méthode retourne `true` si le fichier a bien été sauvé, `false` sinon.

Pour utiliser le paquetage `phelma`, donc pour utiliser la classe `ImageToolkit`, il faut écrire :

```
import phelma.* ;
```

au début de votre fichier.

Exemple d'utilisation de la classe `ImageToolkit` :

```
 // la classe ImageToolkit fait partie d'un package Java nommé phelma, donc :
import phelma.ImageToolkit;

public class TestImageToolkit {
    public static void main(String[] args) {
        // Definition des dimensions
        int nbl=256, nbc=512;
        // Creation du tableau 2D
        int [][] image = ..... ; // a vous de voir...

        // On dessine un rectangle bleu dans l'image
        for (int i=20; i<nbl-20; i++)
            for (int j=20; j<nbc-20; j++)
                image[i][j]=255<<24|255;  // Couleur Bleue en codage ARGB

        // Utilisation de la classe ImageToolkit
        // Les méthodes de cette classe sont des méthodes de classe (static) :
        // => il n'est pas nécessaire (et pas pertinent) de créer un objet de
            type ImageToolkit.
        // => on utilise directement le nom de la classe.

        // Affichage de l'image par la fenetre
        ImageToolkit.displayPixelMatrix(image);
        // Sauvegarde dans un fichier nommé "jolirectangle.png"
        ImageToolkit.savePixelMatrix("jolirectangle", image);

    }
}
```

Pour compiler et exécuter cet exemple en ligne de commande :

— `cd **le répertoire des sources du TP java**`
— `javac TestImageToolkit.java`
— `java TestImageToolkit`

1.2 La classe `Image`

Ecrire une classe `Image` représentant une image stockée en mémoire.

Votre classe comportera les attributs suivants :
— `nbl` et `nbc`, qui représentent respectivement les nombres de lignes et de colonnes de l'image
— `data` : le tableau 2D d'entiers 32 bits contenant les données (pixels).
Les attributs seront déclarés `private` (... bien sûr ...).

Votre classe comportera les méthodes suivantes :
— un constructeur comportant 2 paramètres : le nombre de lignes et de colonnes. L'image est initialement noire (255 << 24).
— un constructeur de copie.
— les accesseurs sur les valeurs des attributs `nbl`, `nbc`
— un accesseur `int getPixel(int i, int j)` qui retourne la valeur du pixel `(i, j)` de l'image
— un modifieur qui modifie la valeur du pixel `(i, j)` de l'image,
— une méthode `void drawRect(int x, int y, int H, int L)` qui dessine dans l'image un rectangle blanc de taille LxH et dont le coin supérieur gauche est à la position x,y. Note : la valeur entière pour le blanc est : `0xFFFFFFFF` en hexadécimal, ou encore 255 << 24|255 << 16|255 << 8|255 (tous les canaux sont saturées).
— une méthode `void afficher()` qui affiche l'image en utilisant la classe `ImageToolkit`
— une méthode `void sauver(String nom)` qui sauvegarde l'image dans le fichier `"nom.pgn"`

Quelques questions...
— les attributs `nbl` et `nbc` sont ils vraiment nécessaires dans cette classe ? Ne pourrait-on pas supprimer ces attributs, tout en conservant les accesseurs sur le nombre de lignes et de colonnes de l'image ?
— pourquoi définir un accesseur direct sur le tableau `data` romprait-il le principe d'encapsulation en POO ?
— pourquoi ne pas définir des modifieurs sur les attributs `nbl` et `nbc` ?

1.3 Programme :

Ecrire un programme qui réalise les actions suivantes :

1. Crée une image im1,
2. Dessine un rectangle blanc dans l'image im1.
3. Affiche l'image im1.
4. Sauve cette image sur disque sous le nom "test.png".
5. Crée une nouvelle image im2, par **copie** de im.
6. Dessine un second rectangle blanc dans l'image im2.
7. Affiche l'image im2 : deux rectangles sont visibles.
8. Réaffiche l'image im1. Elle ne doit pas avoir été modifiée (un seul rectangle visible) !

.../...

2 Principe de la génération d'une forme fractale et classe `Complexe` fournie

La suite de ce TP est consacrée à l'écriture de classes qui permettent de dessiner de jolies fractales

Une forme fractale s'obtient à partir d'une suite de nombres complexes Z_{z_p}, définie en tout point du plan complexe d'affixe z_p par une relation de récurrence $Z_{n+1} = F(Z_n)$.

La fonction F détermine le type de la fractale. Suivant les cas, c'est le premier terme de la suite et/ou la fonction F qui dépend de z_p.

Le principe est alors le suivant :
— Si au point du plan complexe d'affixe z_p la suite Z_{z_p} est convergente, ce point fait partie de la forme fractale.
— Si au contraire la suite est divergente, ce point ne fait pas partie de la fractale.

Pour calculer une forme fractale, il faut donc manipuler des nombres complexes. Nous fournissons une classe `Complexe`, disponible sur le site du cours.
— Télécharger la classe `Complexe` et ajouter-la à votre projet Eclipse, dans le paquetage par défaut (sous `src`).
— Lire le code de cette classe pour comprendre comment l'utiliser.
Pour vous faciliter la vie, générez la javadoc !
En ligne de commande : `javadoc -charset utf8 Complexe.java -d doc`, puis `firefox doc/index.html`.

3 La fractale de Mandelbrot

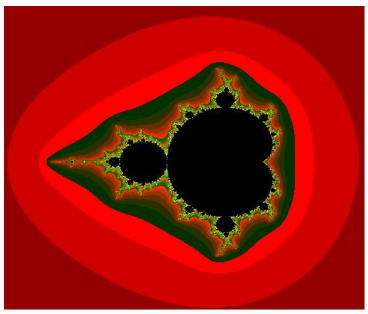


FIGURE 1 – Mandelbrot

3.1 L'ensemble de Mandelbrot

Pour l'ensemble de Mandelbrot M, la fonction F est de la forme $F(z) = z^2 + z_p$, où z_p est le point du plan complexe qui nous intéresse. Le premier terme de la suite est nul.

On a donc la suite $U(z_p) : z_0 = 0; z_{n+1} = z_n^2 + z_p$.

Pour chaque point de coordonnées (x, y) et d'affixe $x + i * y$ du plan cartésien, on définit $z_p = x + i * y$ et on étudie la convergence de la suite $U(z_p)$.

Si la suite diverge, on dit que le point d'affixe $x + i * y$ n'appartient pas à l'ensemble M. Dans le cas contraire, alors le point d'affixe $x + i * y$ appartient à M et on le colorie par exemple en noir.

Exemples :

— Pour le point de coordonnée $(0,0)$, $z_p = 0 + 0 * i$: la suite est définie par $z_0 = 0$ et $z_{n+1} = z_n^2$
La suite est toujours nulle, elle converge (vers 0), donc le point de coordonnées $(0,0)$ appartient à l'ensemble de Mandelbrot

— Pour le point de coordonnée $(1,1)$, $z_p = 1 + i$: la suite est définie par $z_0 = 0$ et $z_{n+1} = z_n^2 + 1 + i$
On a donc :
$z_1 = 1 + i$
$z_2 = 1 + 3 * i$
$z_3 = -7 + 7 * i$
$z_4 = 1 - 97 * i$
$z_5 = -9407 - 193 * i$
$z_6 = 88454401 + 3631103 * i$
On voit que suite tend rapidement vers plus l'infini. Elle diverge. En conséquence, le point de coordonnées $(1,1)$ n'appartient pas à l'ensemble de Mandelbrot.

3.2 Construction d'une image fractale

Pour construire l'image fractale, on parcourt tous les pixels de coordonnées x, y de l'image et on étudie la convergence de la suite précédente avec $z_p = x + i * y$.

Si la suite converge, la couleur du pixel de coordonnées x, y est mis à 0 car il appartient à la fractale. Sinon, on choisit une autre couleur pour ce pixel.

Remarque 1 : En pratique, on décidera que la suite diverge quand son module dépasse une constante $maxModule$, par exemple 100, et on se limite en général à calculer un certain nombre de termes $maxTermes$ (quelques centaines ou milliers de termes au maximum) pour détecter la convergence.

Remarque 2 : Pour obtenir de jolies images, pour les pixels (x, y) auxquels on détecte que la suite diverge, une ruse consiste choisir une couleur qui dépend du rang auquel la divergence est détectée (c'est à dire le rang auquel le module de la suite dépasse $maxModule$).

Pour une fractale en niveaux de gris, on peut, par exemple, choisir pour chacun des canaux R, G et B le carré du numéro de l'itérations auquel on a détecté que la suite divergeait, modulo 255.

Une loi de colorisation plus complexe peut donner une fractale plus jolie encore. Essayez !

Remarque 3 : en réalité la partie intéressante de l'ensemble de Mandelbrot se situe autour de l'origine, pour des points de module inférieur à 1. Il est donc plus pertinent de changer l'échelle de l'image et de la centrer.

Par exemple, on posera $z_p = -1 + (2 * x)/nbl + (-1 + (2 * y)/nbc) * i$.

Dans ce cas, le complexe z_p parcourt l'espace $[-1, 1[\text{x}[-1, 1[$ dans le plan complexe, quand les valeurs de x et y varient sur $[0, nbl - 1]\text{x}[0, nbc - 1]$.

.../...

3.3 La Classe Mandel

Ecrire une classe `Mandel` qui représente une instance de la fractale de Mandelbrot.

Cette classe `Mandel` comportera les attributs `private` suivant ;
— un attribut de type `Image`, dans lequel la fractale sera dessinée
— un attribut `double maxModule` pour le test de divergence : si le module de z_i dépasse cette valeur, on considère que la suite diverge.
— un attribut `int maxTermes` : le nombre de termes qui sera calculé au maximum pour tester si la suite diverge

La classe comportera les méthodes suivantes :
— un constructeur comportant 3 paramètres : la taille de l'image n (l'image sera carrée de taille nxn), la valeur de `maxModule` et de `maxTermes`. Le constructeur génèrera immédiatement l'image en appelant la méthode `genererImage()`, définie ci-dessous.
— les accesseurs aux attributs `nbl`, `nbc` de l'image
— une méthode `private int calculer(Complexe zp);` qui calcule pas à pas les valeur de la suite complexe pour détécter la divergence. `maxTermes` seront calculés au maximum. Cette méthode retourne `maxTermes` si la suite converge, et sinon le numéro du pas, inférieur à `maxTermes`, auquel la la divergence a été détectée.
Pourquoi est-il pertinent que cette méthode soit privée ?
— une méthode `private void genererImage()` qui effectue le calcul de la fractale pour chaque pixel de l'image, c'est-à-dire pour chaque nombre complexe z_p correspondant à la partie $[-1, 1[x[-1, 1[$ du plan complexe. Cette méthode utilise la méthode précédente.
Pourquoi est-il adéquat que cette méthode soit privée ?
— une méthode `void afficher()` qui affiche l'image.
— une méthode `void sauver(String nom)` qui sauvegarde l'image dans le fichier nommé `nom.png"`

3.4 Le programme

Ecrire une classe `TestFractale` pour tester cette classe.

.../...

4 Autres Fractales

4.1 L'ensemble de Julia

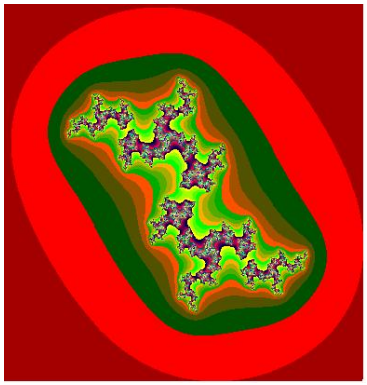


FIGURE 2 – Julia

L'ensemble de Julia est défini presque de la même manière que celui de Mandelbrot. Soit la suite :

$$\begin{aligned} z_0 &= x + i * y \\ z_{n+1} &= z_n^2 + c \end{aligned}$$

Pour l'ensemble de Julia, c est un paramètre fixé pendant tout le calcul de l'image et c'est le premier terme z_0 de la suite qui dépend du pixel. Le point z_0 de coordonnées (x, y) et d'affixe $x + i * y$ appartient à la fractale de Julia si et seulement si la suite z_n converge, où c est un complexe fixé. A chaque c correspond donc un ensemble de Julia particulier, noté $J(c)$.

Voici quelques valeurs de c, donnant des résultats visuels intéressants :
— $c = -0.5 + i * 0.5$
— $c = -0.3 + i * 0.5$
— $c = -0.73 + i * 0.19$
— $c = - - 0.156844471694257101941 + i * -0.649707745759247905171$
— $c = 0.285 + i * 0.01$
— $c = -1.417022285618 + i * 0.0099534$

La classe Julia Sur le même principe que précédemment, écrire une classe `Julia` pour construire une fractale de Julia.

Remarque : La classe Julia aura un nouvel attribut `Complexe c`, qu'il convient d'initialiser dans le constructeur. Pour pouvoir tester plusieurs valeurs de `c`, il serait aussi utile que la classe ait un modifieur `void setC(Complexe c);`. Notez que, quand `c` est modifié, il convient de mettre à jour l'image possédé par l'objet de type `Julia`...

Quels sont les points communs et les spécificités de cette classe `Julia` par rapport à la classe `Mandel` ?

4.2 Les fractales de Newton

La fractale de Newton est l'ensemble de Julia d'une fonction complexe $p(x)$ définie par :

$$\begin{aligned} z_0 &= x + y * i \\ z_{n+1} &= z_n - \frac{p(z_n)}{p'(z_n)} \end{aligned}$$

La fractale classique de Newton est obtenu pour la fonction $p(x) = x^3 - 1$, qui est généralisée facilement avec $p(x) = x^k - 1$

$$\begin{aligned} z_0 &= x + y * i \\ z_{n+1} &= z_n - \frac{z_n^k - 1}{k * z_n^{k-1}} = \frac{(k-1) * z_n^k + 1}{k * z_n^{k-1}} \end{aligned}$$

Pour construire la fractale, on étudie la convergence de la suite z_n pour chaque point z_0 de coordonnées (x, y) et d'affixe $x + i * y$. Dans le cas des fractales de Newton, la suite converge généralement vers une des racines de $p(x) = 0$.

On pourra considérer que la suite à convergé lorsque le module de la différence $z_{n+1} - z_n$ est inférieur à une certaine valeur *epsilon*.

Pour choisir la couleur d'un pixel, on colorie le point soit en fonction du nombre d'itérations nécessaire à établir la divergence (comme pour Julia), soit en attribuant une couleur différente à chaque racine atteinte à la convergence.

FIGURE 3 – Newton coloriée par racine à gauche, par nombre d'itérations à droite

La classe Newton Sur le même principe que précédemment, écrire une classe `Newton` pour construire une fractale de Newton.

Quels sont les points communs et les spécificités de cette classe par rapport aux classes `Mandel` et `Julia` ?

Comment peut-on utiliser cette classe `Newton` pour créer des fractales pour des fonctions différentes de la fonction classique $p(x) = x^3 - 1$ sans voir à modifier cette classe `Newton` ?