

TP2 : construction de types personnalisés

Pour ce TP, l'objectif est de vous familiariser avec les types personnalisés construits à partir des types de base. Le support choisi est d'explorer l'ensemble de Mandelbrot en produisant des images de cet ensemble à différentes échelles.

Nous allons repartir du programme de tracé de l'ensemble de Mandelbrot du TP1, mais gardez une copie du programme d'origine.

1) Utilisation d'une palette de couleurs plus complexe

La première étape va être d'utiliser un plus grand nombre de couleurs pour représenter les différents niveaux de convergence. Lors du TP1, la représentation des couleurs faisait simplement correspondre une composante à la valeur de convergence (codage linéaire sur une composante). Nous allons maintenant utiliser un codage linéaire par morceaux sur plusieurs composantes. Bien que ce codage soit simple, il nécessite beaucoup de code, il faut donc utiliser une fonction spécifique que nous appellerons *palette()*. En C, une fonction ne peut renvoyer qu'une seule valeur. La solution est alors de définir un type de donnée comme étant l'association de trois entiers codant chacun une composante.

Définissez une structure de données appelée *color*, contenant trois champs de type *char* s'appelant *red*, *green* et *blue*. C'est ce type de donnée que renverra la fonction *palette()*.

Le schéma de la figure 1 représente le codage pour chacune des composantes en fonction du nombre d'itérations calculées (cf TP1) notée *c*. Par exemple, pour $0 \leq c \leq 255$, la couleur sera (*red*=*c*, *green*=0, *blue*=0), pour $255 \leq c \leq 510$, la couleur sera (*red*=255, *green*=*c*-255, *blue*=0), pour $510 \leq c \leq 765$, la couleur sera (*red*=765-*c*, *green*=255, *blue*=0) etc. Vous remarquerez que ce cycle n'est pas limité.

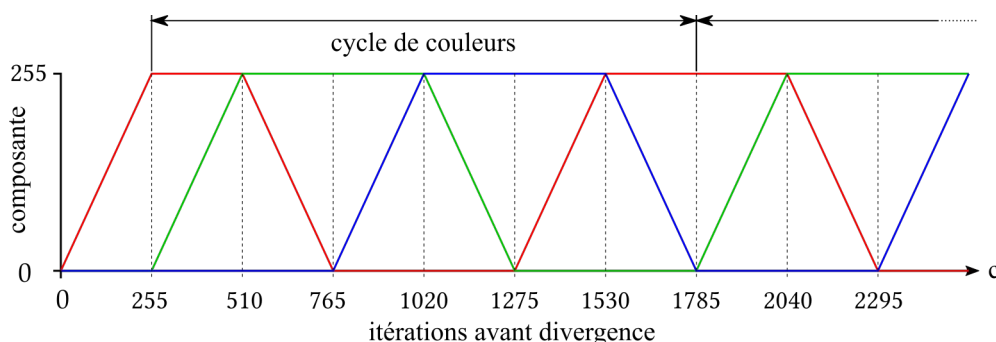


Figure 1: représentation des composantes de couleur selon le nombre d'itérations calculé

Ecrivez la fonction *palette()* dont l'entrée sera la valeur entière *c*, et dont le retour est un *struct color* correspondant au codage de la figure 1.

Modifiez votre programme pour utiliser *palette()* lors de l'écriture de l'image. Pour améliorer le rendu, vous pouvez temporairement multiplier la valeur de *convergence()* par 5 ou 10.

2) Structure de stockage d'une vue en mémoire

Nous allons maintenant rendre le programme plus facile à modifier. Pour cela, il est nécessaire de séparer le calcul de l'image et la création du fichier. Ce qui signifie qu'il faudra temporairement stocker l'image en mémoire.

Écrivez une structure de données appelée *mandel_pic* contenant les champs suivants :

- *width* et *height* : deux *int* qui codent respectivement la largeur et la hauteur en pixels
- *Xmin*, *Ymin* : deux *doubles* pour mémoriser les coordonnées du point en bas à gauche.
- *Xmax*, *Ymax* : deux *doubles* pour mémoriser les coordonnées du point en haut à droite.
- *scale* : un *double* définissant l'échelle de l'image. (1.0 correspond à $X_{max} - X_{min} = 3.0$)
- *pixwidth* : un *double* définissant la largeur d'un pixel de l'image.
- *converge* : un pointeur vers *int*, destiné à identifier le tableau de valeurs de convergence. Les pixels sont agencés dans le même ordre que pour le fichier image.

Écrivez la fonction *new_mandel()* qui crée un *struct mandel_pic* dont le prototype est :
struct mandel_pic new_mandel(int width, int height, double Xmin, double Ymin, double scale);

Les arguments sont destinés à être directement stockés dans la structure. Les autres éléments de la structure doivent être calculés à partir des arguments :

- $X_{max} = X_{min} + (scale * 3.0)$ (par définition de *scale*)
- $Y_{max} = Y_{min} + (scale * 3.0 * height / width)$
- $pixwidth = scale * 3.0 / width$

A vous de définir comment initialiser *converge*. Pour rappel, *converge* étant de type *int **, il peut être utilisé comme un tableau.

Modifiez votre programme pour qu'il remplisse une structure *mandel_pic* au lieu de créer un fichier. Utilisez au maximum les champs de la structure pour que votre programme soit le plus générique possible.

Écrivez la fonction *save_mandel()* qui reçoit un *struct mandel_pic* et un nom de fichier sous forme de chaîne de caractères, et qui crée le fichier image correspondant en utilisant *palette()*.

A ce stade, vous devriez retrouver le comportement de la première partie du TP.

3) Utilisation d'une vue précédente pour accélérer le calcul

L'aspect intéressant de l'ensemble de mandelbrot réside dans sa structure fractale (la répétition d'un même motif à des échelles différentes), mais pour bien l'observer, il faut zoomer sur la figure et utiliser une valeur d'itération maximale bien supérieure. Par exemple en montant à 1000 itérations maximum, les valeurs ($X_{min} = -0.755232$, $Y_{min} = 0.121387$, $scale = 0.01$) ou encore ($X_{min} = -0.743662$, $Y_{min} = 0.131810$, $scale = 0.000014$) donnent des images intéressantes, mais leur calcul nécessite plus de ressources.

Pour calculer les images successives d'une séquence de zoom, l'idée est donc de réutiliser les calculs effectués pour l'image précédente : on accélère le calcul au prix d'une erreur plus ou moins négligeable sur le rendu final.

Écrivez la fonction *interpolate()* qui, pour un *struct mandel_pic*, et deux *double* *x* et *y*, donne une estimation de la convergence au point (*x*,*y*) au format *int*. Cette valeur sera calculée en retournant la valeur du point le plus proche. Si le point (*x*,*y*) est en dehors de l'image déjà calculée, la fonction renvoie -1, ce qui signifie qu'aucune valeur valide ne peut être retournée (il faudra alors calculer avec la fonction *convergence()*).

Testez *interpolate()* en utilisant d'abord un calcul normal sur les coordonnées (*Xmin* = -0.755232, *Ymin* = 0.121387, *scale* = 0.01), puis en utilisant *interpolate* pour calculer l'image aux coordonnées (*Xmin* = -0.752914, *Ymin* = 0.123475, *scale* = 0.00738). Il n'y a pas de risque de récupérer la valeur -1, car la seconde image est entièrement dans la première.

La technique précédente est rapide, mais inutile pour augmenter le niveau de détails dans l'image. L'idée est donc d'utiliser *interpolate()* pour tester les quatre points déjà calculés qui entourent le point (*x*,*y*). Si ces quatre points sont identiques, on peut raisonnablement croire que (*x*,*y*) leur sera égal.

Modifiez *interpolate()* pour qu'il vérifie l'égalité entre les 4 points. S'ils ne sont pas égaux, *interpolate* doit retourner -1. Ce système est particulièrement utile dans les zones où la suite ne converge pas. En effet, ces zones sont les plus lentes à calculer à cause du grand nombre d'itérations nécessaires.

Modifiez également le reste du programme pour qu'il utilise *convergence()* lorsque *interpolate()* a renvoyé -1.

Testez cette nouvelle méthode.

Pour éviter que des erreurs de calcul ne s'accumulent, vous pouvez également renvoyer -1 de temps en temps, sur une base aléatoire. Cela forcera le nouveau calcul de certains pixels, même si *a priori*, ce n'était pas nécessaire.

Pour cela, le test est de vérifier si la fonction *random()* renvoie une valeur inférieure à *RAND_MAX*/100 (pour ne forcer le nouveau calcul que pour 1 % du temps).

4) (optionnel) Création d'une série d'images

Tout comme au TP 1, vous pouvez, à loisir, créer une série d'images pour ensuite les assembler en une vidéo. Pour créer un grand nombre d'images, il faut être capable de créer des noms de fichiers en fonction de variables. Pour cela, la fonction *sprintf* est très utile : elle se contente de créer une chaîne de caractères dans un format défini comme pour *printf*. Pour plus de précisions : *man 3 sprintf*.

Maintenant qu'il est possible d'observer des phénomènes vers les grandes valeurs d'itération, un point intéressant est (*x*= -0.743643887037151 *y*= 0.13182590420533) (source Wikipedia). Par contre, le calcul peut être long. Faites d'abord des tests sur des images de petites dimensions.

5) (optionel) Le multitâche du pauvre...

Il est possible d'encore augmenter la vitesse de calcul en utilisant plusieurs processeurs en même temps. Il n'est pas question d'utiliser des fonctions particulières de parallélisation (qui ne sont pas au programme de la filière elec), mais il reste assez simple de lancer plusieurs processus indépendants.

Modifiez votre programme pour qu'il récupère ses paramètres depuis la ligne de commande. Notamment : Xmin, Ymin, Scale, le nombre de fichiers à générer, et le numéro que portera le premier fichier.

Vous pouvez alors exécuter quatre fois votre programme en parallèle, sur quatre séries d'images successives. Les optimisations ne fonctionneront pas sur la première image de chaque série, mais si votre ordinateur possède quatre processeurs physiques, les programmes ne se ralentiront pas (trop) les uns les autres.

Pour exécuter un programme depuis un autre programme, la fonction C à utiliser est *system()*. Elle reçoit la commande UNIX à exécuter sous forme de chaîne de caractère.

- *system("./sous_programme");* va exécuter votre sous-programme et attendre la fin de son exécution.
- *system("./sous_programme &");* va lancer votre sous-programme et votre programme continuera normalement en parallèle du sous programme.

Le top du top, c'est si votre sous-programme et votre programme ne font qu'un :). En effet, grâce aux paramètres de ligne de commande, vous pouvez définir comment votre programme doit se comporter.

- *./programme* (sans argument) peut lancer le calcul complet
- *./programme -2.0 -1.0 1.0 10 0* peut uniquement faire une série de 10 images commençant à xmin=-2.0, ymin=-1.0, scale=1.0 et dont la première porte le numéro 0.

Avertissement : Si vous générez de trop grandes séries d'images de grande résolution, vos fichiers occuperont beaucoup de place sur le disque et risquent de vous faire dépasser votre quota. Supprimez vos images dès que vous les avez vérifiées.