IF112 - Projet d'Informatique

2023 - 2024

Yannick Bornat - yannick.bornat@enseirb-matmeca.fr
Guillaume Bourmaud - guillaume.bourmaud@enseirb-matmeca.fr
Clémence Gillet - clemence.gillet@enseirb-matmeca.fr
Rémi Giraud - remi.giraud@enseirb-matmeca.fr

TP2 : Débogage avec *GDB*
Construction de types personnalisés

Les objectifs de cette séance sont :
- Savoir manipuler l'outil de débogage *GDB*.
- Se familiariser avec les types personnalisés construits à partir des types de base.

Pour cela nous allons continuer d'explorer l'ensemble de Mandelbrot en reprenant des fonctions du TP1.

1 Débogage avec *GDB*

Pour le débogage de code C, on peut faire bien mieux qu'à coups de `printf` placés à des endroits clés du code. Un des outils principaux de débogage est *GDB* (GNU Debugger) qui permet notamment :
- De fixer des *breakpoints* et d'exécuter le code pas à pas.
- D'afficher les valeurs des variables au cours de l'exécution.
- D'afficher parfois directement la ligne qui cause une erreur de segmentation.

GDB permet donc de déboguer le côté algorithmique du code, en l'exécutant instruction par instruction ou morceau par morceau et en affichant les valeurs des variables. Une instruction provoquant une erreur de segmentation peut également être indiquée mais nous verrons dans le TP suivant un outil dédié à ce problème permettant aussi de détecter les fuites mémoires (ex. : oubl de `free`).

Ci-dessous un exemple d'utilisation de *GDB* avec les commandes principales :

`gcc -g code.c`	Compilation sans optimisation pour le débogage (préserve les n° de lignes)
`gdb a.out`	Exécution avec *GDB*
`break 10`	Crée un breakpoint à la ligne 10
`run`	Lance le programme - qui s'arrête à la ligne 10
`p x`	Affiche la variable x si elle existe avant la ligne 10
`n`	(next) Exécute la ligne suivante
`help`	Pour voir toutes les options existantes
`c`	(continue) Continue jusqu'au breakpoint suivant
`clear`	Efface tous les breakpoints
`run`	Relance le programme entièrement
`quit`	Sort de gdb et retourne au terminal

Testez le programme contenu dans *code_gdb.c* en le compilant d'abord simplement avec la commande `gcc code_gdb.c` puis exécutez-le avec `./a.out`. Vous devriez avoir une erreur de segmentation.

Compilez avec l'option `-g` et exécutez avec *gdb* pour localiser l'erreur.

Testez avec d'autres tailles de matrices. Est-ce que vous obtenez toujours un erreur de segmentation ? Est-ce que quand c'est le cas, *GDB* vous renvoie toujours la ligne causant l'erreur de segmentation ?

Utilisez les commandes ci-dessus, notamment celle permettant d'afficher les variables pour corriger facilement l'erreur.

2 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 composan te.

Définissez une structure de données appelée `color`, contenant trois champ de type `unsigned 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 $256 \leq c \leq 511$, la couleur sera (red=255, green=c-255, blue=0), pour $512 \leq c \leq 767$, la couleur sera (red=767-c, green=255, blue=0) etc. Vous remarquerez que ce cycle n'est pas limité.

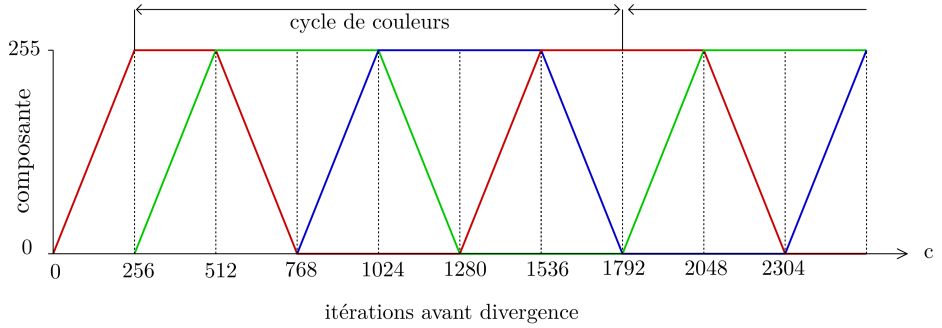

Figure 1: Ordre de codage des composantes de pixel

Écrivez 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. (Rappel : opérateur modulo `%`)

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.

3 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, height` : deux `int` qui codent respectivement la largeur et la hauteur en pixels
- `Xmin , Ymin` : deux `double` pour mémoriser les coordonnées du point en bas à gauche.
- `Xmax , Ymax` : deux `double` pour mémoriser les coordonnées du point en haut à droite.
- `scale` : un `double` définissant l'échelle de l'image. (1.0 correspond à Xmax-Xmin=3.0)
- `pixwidth` : un double définissant la largeur d'un pixel de l'image.
- `convrg` : un pointeur vers `int`, destiné au tableau de valeurs de convergence de chaque pixel.

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 :

- Xmax = Xmin + (scale * 3.0) (par définition de scale)
- Ymax = Ymin + (scale * 3.0 * height / width)
- pixwidth = scale * 3.0/width

À vous de définir comment initialiser `convrg`. Pour rappel, `convrg` 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 recoit 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()`.

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

4 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 (Xmin = -0.755232, Ymin = 0.121387, scale = 0.01) ou encore (Xmin = -0.743662, Ymin = 0.131810, scale = 0.000014) donnent des images intéressantes, mais leur calcul nécessite plus de ressources.

Testez ces points caractéristiques pour observer un effet de zoom. Combien de temps met le programme pour calculer une nouvelle image ?

Pour calculer les images successives d'une séquence de zoom, on peut réutiliser les calculs effectués pour l'image précédente : on accélère le calcul au prix d'une erreur plus ou moins négligeable. On peut d'abord calculer cette valeur 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 faut alors calculer avec la fonction `convergence()`).

Déboguez avec *GDB* 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`.

Comparez le temps d'exécution avec l'option de débogage `-g` et sans.

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, la fonction doit retourner -1. Ce système est particulièrement utile dans les zones ou la suite ne converge pas car 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.

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 peut être 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).

5 Structures et fonctions génériques de manipulation d'images

On souhaite à présent s'inspirer des codes précédents pour créer une structure et des fonctions de manipulations d'images génériques. Ces structures et fonctions nous serviront notamment dans les TP suivants.

Écrivez dans un nouveau fichier, la définition de la structure `picture` qui sera utilisée pour mémoriser votre image pendant sa création. Elle doit contenir les champs suivants :
- width : la largeur de l'image en pixels.
- height : la hauteur de l'image en pixels.
- pixels : une représentation du contenu de l'image sous forme d'un tableau de composantes de couleur.

Écrivez la fonction `new_pic()` qui récupère deux entiers, une largeur et une hauteur exprimées en pixels, et qui renvoie un `struct picture` correspondant aux dimensions souhaitées. Cette fonction est très proche de la fonction `new_mandel()` du TP2.

Écrivez la fonction `save_pic()` qui reçoit un `struct picture` et un nom de fichier sous forme de chaîne de caractères, et qui crée le fichier image correspondant. Cette fonction peut être écrite en simplifiant la fonction `save_mandel()` du TP2.

Écrivez la fonction `set_pixel()` qui reçoit un `struct picture`, deux entiers (des coordonnées x et y en pixels) et une couleur. Cette fonction assigne la couleur donnée au pixel correspondant aux coordonnées x et y dans l'image fournie. Pour suivre la convention des formats d'images, le pixel de référence (0,0) sera en haut à gauche. Selon votre choix d'utiliser ou non le type `struct color`, vous entrerez votre couleur sous forme d'un `struct color` ou de trois `char` codant les trois composantes rouge, verte et bleue.

Testez les différentes fonctions que vous avez produites jusqu'ici en créant une image de taille 10x10, dont tous les pixels sont noirs à l'exception des quatre pixels correspondant aux quatre coins de l'image qui prendront la couleur de votre choix (à condition de ne pas choisir noir).

6 Bonus : Création d'une série d'images

Retour à Mandelbrot, 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 pouvoir 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 définit 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.

7 Bonus : 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) entre eux.

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 le sous-programme et attendre la fin de son exécution.

- `system("./sous_programme &");` va lancer le sous-programme et le 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.