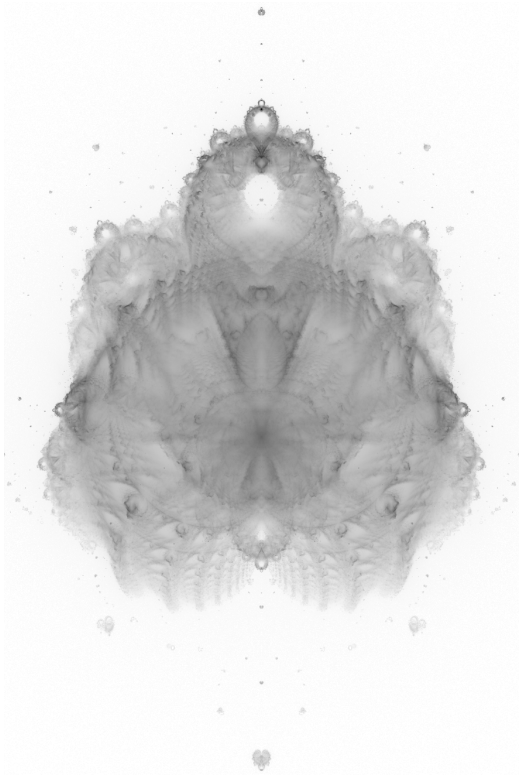
BuddhaBrot



Objectifs pédagogiques:

- Lecture / écriture de fichiers en C (§3, §5)
- Communication par sockets (§4)
- Benchmark et analyse de performance (§5)
- Structure de données dynamique et malloc (§5)
- Design de code et encapsulation (§6 – optionnel)

Présentation. L'image ci-contre, nommée BuddhaBrot, est une représentation originale de la fractale de Mandelbrot bien connue. Le nom de cette représentation vient de la forme obtenue, qui peut faire penser à la représentation d'un Boudha en position du lotus.

Le but de ce mini-projet est de dessiner cette fractale. Comme cela nécessite beaucoup de calcul, c'est l'excuse pour programmer en C avec un peu de réseau pour répartir les calculs.

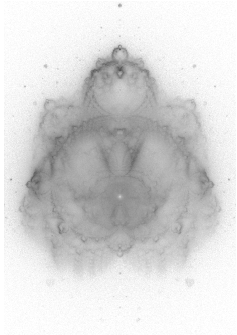
Principe. Pour obtenir l'image ci-contre, on calcule des milliers de trajectoires d'une fonction mathématique particulière, et le niveau de gris de chaque pixel dans l'image est en fonction du nombre de trajectoires passant par ce point. L'image est donc une sorte d'histogramme en deux dimensions.
Malheureusement, les trajectoires remplissant les conditions nécessaires sont rares. Il faut donc considérer des millions de trajectoires composées chacune de milliers de points pour le rendu.

1 Mandelbrot et BuddhaBrot

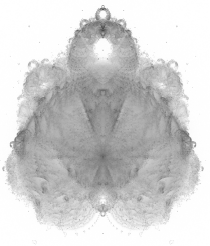
L'ensemble de Mandelbrot, dont nous allons explorer une variante, est l'ensemble des points c du plan complexe pour lesquels la suite z_n est bornée quand n tend vers l'infini. $\begin{cases} z_0 = 0 \\ z_{n+1} = z_n^2 + c \end{cases}$

Cet ensemble est une fractale: sa représentation graphique est auto-similaire. Cela signifie que l'on retrouve la forme générale quand on zoom sur la frontière de l'ensemble. Visitez la page Wikipédia pour quelques représentations notables de l'ensemble de Mandelbrot. On représente souvent l'ensemble lui-même en noir, entouré de points dont la couleur indique la vitesse de divergence de la trajectoire issue de chaque point.

Pour la représentation du BuddhaBrot, on ne s'intéresse pas aux points faisant partie de l'ensemble, mais plutôt à certains points c du plan complexe pour lesquels la suite z_n ci-dessus n'est pas bornée. Plus précisément, on s'intéresse aux orbites de la suite pour lesquelles il existe un entier l tel que $\forall n < l, |z_n| < 2$ et $|z_l| \geq 2$. On dit alors que l est la longueur de l'orbite issue du point c. Les points à l'intérieur de l'ensemble donnent des orbites de longueur infinie tandis que les points à l'extérieur donnent des orbites de longueur finie.



La longueur des orbites sélectionnées a un impact sur le rendu visuel du BuddhaBrot. La figure de gauche, relativement bruitée, combine 50 000 orbites de longueur comprise entre 100 et 200 seulement. La figure en haut de la page combine quant à elle environ 100 000 orbites de longueur comprise entre 10^4 et $2 \cdot 10^4$, tandis que la figure de droite combine des quelques milliers d'orbites de longueur comprise entre 10^6 et 10^7 pour un résultat beaucoup plus net.



Malheureusement, ces orbites de longueur finie mais importante sont très rares, ce qui oblige à explorer encore plus de candidates pour les trouver. Mon algorithme a un hit ratio de l'ordre de 30% quand je cherche des orbites de longueur 100, tandis que ce ratio tombe à moins de 0.02% malgré tous mes efforts quand je cherche des orbites de longueur entre 10^6 et 10^7...

2 Le code fourni

Le template de code fourni contient un programme C cherchant des orbites de la longueur demandée et dessinant BuddhaBrot. Le programme est conçu sur une boucle infinie, cherchant sans fin de nouvelles orbites, et redessinant la fenêtre toutes les 10 orbites trouvées (ou toutes les 1000 tentatives infructueuses).

Pour le compiler et le tester sur votre machine, vous aurez besoin de la bibliothèque SDL2, qui s'installe très facilement sur tous les systèmes d'exploitation.

Le programme est très simple, voire relativement rudimentaire pour laisser le plaisir de l'améliorer. Beaucoup de paramètres du programme sont définis en haut du code source. On retrouve la taille de la fenêtre d'affichage, la longueur attendue des orbites et les coordonnées de la zone de recherche : $Re \in [-1,8; 1,4]$; $Im \in [-1,1; 1,1]$. Vient ensuite une variable globale `orbits` où l'on décompte les orbites passant en chaque point. On trouve enfin quatre fonctions principales avant le main:

- `void pick_candidate(double*, double*)` choisit un candidat potentiellement intéressant dans le plan complexe. Dans cette version, on fait un tirage uniforme dans la zone possible, et on refait le tirage si on tombe sur un point dont on peut déterminer facilement qu'il est dans l'ensemble de Mandelbrot. Voir la page Wikipédia pour plus de détails sur la formule utilisée. Le résultat est stocké dans les pointeurs passés en paramètre (car le C ne permet pas de renvoyer plusieurs valeurs).
- `int orbit_length(double, double)` calcule la longueur de l'orbite issue du point passé en paramètre.

  La version donnée implémente l'algorithme de Brent pour détecter les cycles au plus vite: on maintient un pointeur vers une un point vu précédemment dans la trajectoire, et on coupe l'exploration dès qu'on repasse par le même point. L'ancien pointeur est déplacé vers la position courante chaque fois que la longueur courante de l'orbite est une puissance de deux (notez la ruse pour calculer déterminer efficacement si un nombre est une puissance de deux). Cette optimisation est extrêmement efficace, comme vous pouvez le vérifier en commentant cette partie du code.
- `void orbit_add(double, double)` modifie la matrice des orbites connues pour y ajouter l'orbite issue du point indiqué. Comme on ne sait pas à priori si l'orbite explorée sera de la bonne longueur, on ne peut pas modifier la matrice lors du premier parcours de l'orbite. Il est plus efficace de parcourir l'orbite deux fois que de devoir parfois annuler les changements apportés à la matrice.
- `void compute_pixels(uint32_t*)` calcule la couleur de chaque pixel dans la matrice d'après le décompte passant par chaque case de `orbits`. Plusieurs façons de calculer la couleur sont fournies, mais libre à vous d'inventer la vôtre. N'hésitez pas à m'envoyer vos galeries de beaux rendus ;)

3 Mise en place d'un cache

Étant donné la difficulté à trouver des points valides, la première étape est probablement de mettre en place un cache de résultats sur disque. Chaque fois qu'on trouve un nouveau point, il faut l'écrire dans un fichier dédié que l'on relira à chaque redémarrage. Le plus simple est d'utiliser un fichier texte pour cela. Un fichier binaire serait plus efficace, mais également plus difficile à debugguer. Attention cependant à bien écrire les flottants avec toute la précision, car les erreurs d'arrondi peuvent changer la trajectoire du tout au tout. Le mieux est probablement d'écrire chaque point de la façon suivante. Ajouter la longueur de l'orbite évite de la recalculer au rechargement pour vérifier que le point est intéressant pour la longueur recherché dans cette exécution.

```
fprintf(output, "%.60lf %.60lf %d\n", x, y, length);
```

Peut-on faire mieux que `.60` ? Pourquoi ?

4 Parallélisation

L'objectif principal de ce projet est de modifier le code fourni pour le découper en un client et un serveur. Le serveur écoute sur une socket et affiche les orbites issues des points reçus par TCP, tandis que le client cherche des points intéressants. L'idée est évidement de lancer de nombreux clients pour accélérer la visualisation.

Dans un premier temps, vous pouvez ouvrir une nouvelle connexion pour chaque nouveau point à envoyer, mais il est bien plus efficace de garder ces connexions ouvertes, même si cela oblige le serveur à utiliser `select(2)` pour trouver la prochaine socket à servir.

5 Optimisation

Avant de tenter la moindre optimisation, il faut afficher le nombre moyen d'orbites tentées et celui d'orbites réussies par seconde. Sans cela, on ne peut pas mesurer les avantages respectifs de chaque technique. Ensuite,

la première étape est d'activer les optimisations du compilateur en ajoutant le paramètre `-O3` à la liste `CFLAGS` dans le fichier `CMakeLists.txt`.

Au lieu de tirer des points au hasard dans l'espace, il est beaucoup plus efficace de précalculer des zones d'intérêt et de ne tirer que des points dans ces zones. Pour cela, on peut partir d'un maillage uniforme puis déterminer les cases du maillage qui contiennent des points d'intérêt. On élimine les cases dont les quatre angles sont du même coté de la limite pour ne garder que les cases dont au moins un angle donne une orbite trop longue et un autre angle donne une orbite trop courte. Cet algorithme élimine certaines cases contenant des points d'intérêt (par exemple quand l'ensemble de Mandelbrot fait une incise dans la case sans toucher les angles), mais il reste intéressant.

L'intérêt ici est de réfléchir à comment stocker en mémoire la liste des cases sélectionnées (spoiler: ce n'est PAS un tableau de taille fixe), puis d'implémenter la petite structure de données correspondante. Réaliser un tirage uniforme sur la zone ainsi décrite demande aussi un peu réflexion.

De plus, le calcul de la grille est une opération très coûteuse, et il serait intéressant d'avoir un cache sur disque du résultat de ce calcul. Mais il est important d'invalider le cache en cas de changement des paramètres comme la taille des cases ou la longueur d'orbite recherchée.

On peut aussi chercher à paralléliser le calcul de cette grille en utilisant des threads, ou imaginer un maillage plus intelligent de l'espace qu'une grille uniforme comptant énormément de petites cases, mais cette extension demande peut-être trop de travail.

6 De jolies extensions

Une méthode de rendu donnant de très beaux résultats consiste à combiner plusieurs matrices de décompte sur les différents canaux RGB, en mettant par exemple les orbites de courte longueur sur le canal R et des orbites plus longue sur le canal G. Cela nécessite cependant de passer du code fourni, très procédural avec la matrice `orbits` en globale, à un code plus encapsulé où cette matrice est passée en paramètre des fonctions l'utilisant. Cette extension, très intéressante d'un point de vue design de code, semble nécessaire pour pouvoir utiliser différentes matrices pour les différentes longueurs d'orbite.

On peut aussi vouloir augmenter l'application SDL afin de pouvoir zoomer sur les différentes parties de l'espace. Cela demande d'utiliser des matrices d'orbite bien plus grandes, et modifier la fonction `compute_pixels` pour afficher la partie voulue. Cette extension est attirante, mais elle ne présente pas vraiment d'intérêt d'un point de vue programmation.

Au lieu de chercher des orbites de longueur toujours plus grande, une autre approche est d'appliquer un algorithme de traitement d'image pour lisser le résultat. Un algorithme particulièrement adapté est donné dans l'article "Robust Denoising using Feature and Color Information", de Fabrice Rousselle, Marco Manzi et Matthias Zwicker que l'on trouve sur internet. Cette extension présente probablement un intérêt marginal d'un point de vue programmation.

Enfin, on peut vouloir appliquer cette approche à d'autres fractales, comme par exemple le tri-Mandelbrot défini par $z_{n+1} = z_n^3 + c$, le quadri-Mandelbrot défini par $z_{n+1} = z_n^4 + c$ ou peut-être d'autres ensembles définis d'après la fractale de Julia. Cette extension est à réserver aux plus curieux, ceux qui ont beaucoup de temps libre.