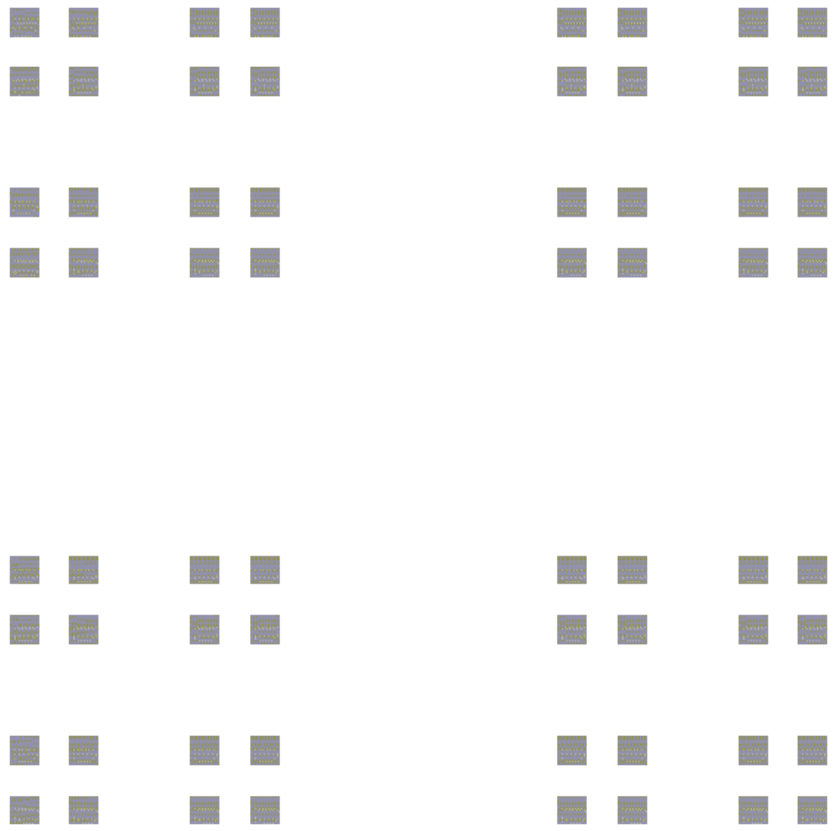
Génération de fractale

En mathématiques, l'ensemble de Cantor (ou ensemble triadique de Cantor, ou poussière de Cantor), est un sous-ensemble remarquable de la droite réelle construit par le mathématicien allemand Georg Cantor.

Il s'agit d'un sous-ensemble fermé de l'intervalle unité [0, 1], d'intérieur vide. Il sert d'exemple pour montrer qu'il existe des ensembles infinis non dénombrables [...]. C'est aussi le premier exemple de fractale (bien que le terme ne soit apparu qu'un siècle plus tard), et il possède une dimension non entière.

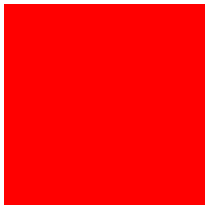
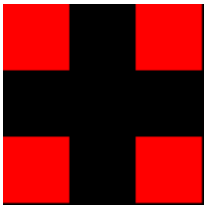
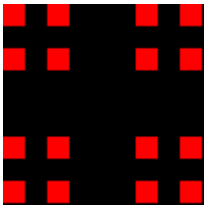
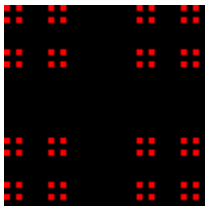
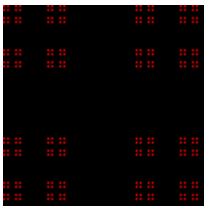
Référence: https://fr.wikipedia.org/wiki/Ensemble_de_Cantor

Pendant ce TP nous allons générer le carré de cantor:



https://fr.wikipedia.org/wiki/Ensemble_de_Cantor#/media/Fichier:Carre_cantor.png

En fonction du nombre d’itérations, nous pouvons avoir plus ou moins de carrés:

 0 itérations = 1 carré	 1 itération = 4 carrés	 2 itérations = 16 carrés
 3 itérations = 64 carrés	 4 itérations = 256 carrés	

Partie 1

Dans un premier temps, nous allons générer l'ensemble de Cantor sous un format statique de votre choix (`png`, `jpeg`, `svg`, etc.).
L'utilisation de librairies externes pour générer les images est autorisée. Au choix:

- https://crates.io/crates/image
- https://crates.io/crates/plotters
- https://crates.io/crates/svg
- ou autre…

On doit pouvoir paramétrer la taille du carré initial et le nombre d'itérations. Passer ces paramètre soit:

- en ligne de commande
- en variables d'environnement
- de façon intéractive

Performance: pour la partie suivante, si vous commencez à rencontrer des problèmes de lenteur d'exécution, c'est normal car on commence à s'attaquer à un domaine de calcul intensif. Nous pouvons évidemment et idéalement chercher des optimisations algorithmiques, cependant, il est bon de savoir que par défaut, le compilateur et la toolchain Rust émettent un exécutable en mode "debug".
Ce mode fait que l'exécutable a plein de symboles supplémentaires attachés, qui réduisent la performance, mais le rendent plus facilement "debuggable". Ces symboles font que par exemple, en cas de crash, nous avons toute la "stack trace" qui est émise.
Lorsqu'une application est prête, nous pouvons la compiler en mode "release". Cela la rend beaucoup plus performante mais moins debuggable. Pour utiliser le mode "release" nous passons en paramètre `--release` à la commande `cargo`
Référence:
https://doc.rust-lang.org/book/ch01-03-hello-cargo.html?highlight=--release#building-for-release

Partie 2

Pour cette partie, nous allons donner vie au carré de Cantor. Déjà sur la partie précédente, nous pouvons zoomer manuellement sur l'image générée. Nous allons faire ça en 2 temps

Générer une animation

Utiliser le format GIF pour créer un zoom "infini" sur le carré de Cantor. Nous pouvons utiliser la librairie suivante pour générer le GIF: https://crates.io/crates/gif.

Résultat possible joint à l'énoncé.

Une fois ce résultat atteint, quels sont les axes d'amélioration ?

- Comment boucler au bon moment pour rendre le zoom le plus fluide possible ?
- Comment rendre les couleurs plus agréables ?

Utiliser une librairie graphique

Enfin, nous allons utiliser une librairie graphique pour faire un zoom intéractif sur le carré de Cantor.
L'utilisateur peut alors paramétrer avec des widget (slider par exemple) les entrées du générateur du carré de Cantor – à savoir, le zoom et le nombre d'itérations.
Librairie graphique au choix

- https://crates.io/crates/egui
- https://crates.io/crates/iced
- https://crates.io/crates/kiss3d
- Ou autre…

Résultat possible joint à l'énoncé.

Une fois ce résultat atteint, quels sont les axes d'amélioration ?

- Peut-on choisir quelle zone explorer de l'ensemble du carré ?
- Comment peut-on zoomer avec la souris au lieu d'un slider ?
- Peut-on rendre le nombre d'itérations dynamique en fonction du zoom ?
- Peut–on, en fonction du zoom, ne dessiner que les carrés qui sont visibles à l'écran ?