AYMONIN Maxime
GRIMA Olivier
DERAIN Victor

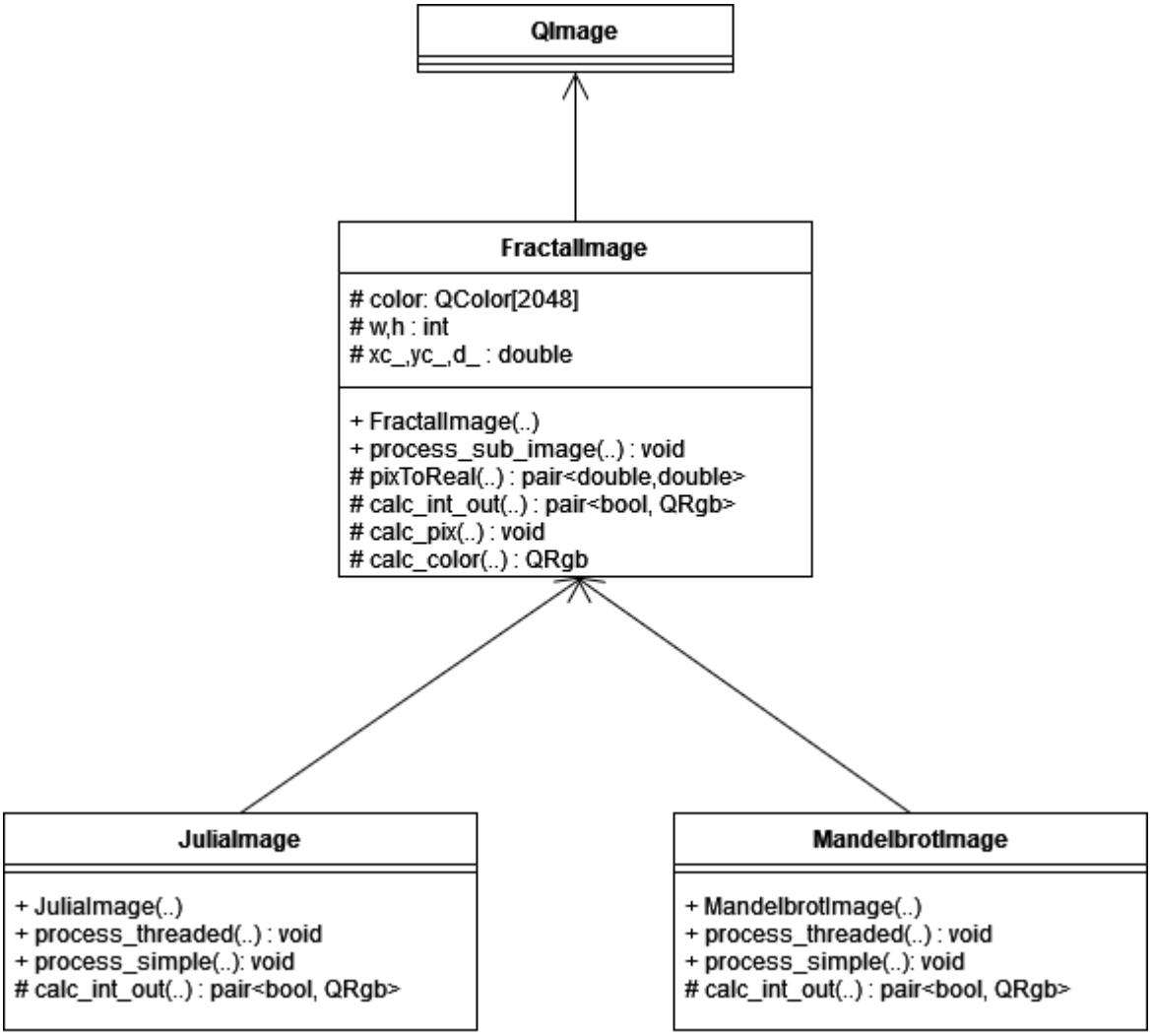
Lab 4 : Fractal

Objectif :

L'objectif de ce TP est de "dessiner" une image de l'ensemble de Mandelbrot. Nous allons pour cela manipuler des nombres complexes et découvrir la programmation multi-threaded nous permettant d'accélérer le temps de calcul de l'image à afficher.

I) Code

1) Architecture générale



Ci-dessus le diagramme de classes de notre solution finale. A la base de tout il y a la classe QImage qui nous permet d'avoir une PixMap. Ensuite nous avons décidé d'implémenter une class abstraite qui représente une image de fractale. Cette architecture a été faite en prenant en compte la partie bonus. C'est pour cela que la classe abstraite est utile car grâce au polymorphisme et à la liaison dynamique nous pourront ensuite appeler les méthodes

process_threaded ou process_simple pour calculer l'image indépendamment du type de fractal. Puisque nous avons deux classes qui héritent de FractalImage : MandelbrotImage et JuliaImage pour l'implémentation des calculs des deux set.

2) Classe mère

```
class FractalImage : public QImage
{
public:
   FractalImage(int width,int height,double xc,double yc,double d);
   virtual void process_sub_image(int i, int m);
   virtual void process_threaded(int i)=0;
   virtual void process_simple()=0;
protected:
   std::pair<double,double> pixToReal(int x,int y);
   virtual std::pair<bool, QRgb> calc_in_out(double rx, double ry)=0;
   void calc_pix(int px,int py);
   QRgb calc_color(int i);

   QRgb color[2048];
   int w,h;
   double xc_,yc_,d_;
};
```

Cette classe est déclarée comme virtuel pure et sera la classe mère des classes Mandelbrot et Julia. Elle hérite elle-même de la classe QImage qui nous sera utile par la suite pour afficher l'image.

FractalImage est le constructeur, il permet d'initialiser la taille de la fenêtre ainsi que les coordonnées et la taille de la fenêtre du "monde réel".
La méthode *process_sub_image* nous sera utile lorsque l'on utilise plusieurs thread. En effet, elle permet de découper l'image en plusieurs morceaux pour que chaque thread traite un morceau spécifique de l'image. Les paramètres "i" et "m" correspondent respectivement au numéro du thread appelant la méthode et au nombre maximum de threads. Cela nous permet de savoir comment découper notre image et quelle partie doit être calculée par ce thread.
Suivent ensuite deux méthodes virtuelles pure qui servent respectivement à gérer l'exécution mutli-threaded et l'exécution "simple". Ces deux méthodes étant virtuelles pures, leur corps est détaillé dans les classes filles **"MandelbrotImage"** et **"JuliaImage"**.
Sous la portée protected, nous avons la méthode *pixToReal* qui nous permet de faire l'interpolation linéaire entre les coordonnées des pixels vers celles du "monde réel".
Nous avons ensuite la méthode virtuelle pure *calc_in_out* qui permet de savoir si le pixel en question fait parti de l'ensemble de Mandelbrot ou non et quelle est sa couleur. La méthode retourne une paire Booléen/couleur (QRgb).
La méthode *calc_pix* fait appel aux méthodes *pixToReal* et *calc_in_out* pour déterminer si le pixel dont les coordonnées données en paramètre doit être coloré ou s'il doit rester noir.

Enfin, la méthode *calc_color* retourne la couleur à afficher pour un pixel. Cette méthode est appelée dans la méthode *calc_in_out*.

Explication section 2 :

Le calcul des couleurs est réalisé en deux parties. La première correspond au calcul du tableau des couleurs.

Nous avons créé une classe Linear qui, en prenant deux vecteurs de doubles, permet de calculer à l'aide d'une interpolation linéaire n'importe quel f(x) pour tout x appartenant de 0 à 1.

Nous instancions alors 3 objets de type linéaire qui correspondent aux différentes valeurs RGB.

Ensuite, nous avons préalablement créé un tableau de QRgb de 2048 éléments. Celui-ci est créé dans la classe FractalImage.
Ici nous parcourons chacun de ses éléments afin de le remplir avec qRgb() qui est une méthode renvoyant un élément du type QRgb.
Nous utilisons une méthode qui permet d'utiliser l'interpolation linéaire pour obtenir le bon y.
En parcourant notre tableau à l'aide du for, i varie de 0 à 2047, en le divisant par 2047 nous obtenons bien un x variant de 0 a 1.
Le double obtenu est ensuite converti en int a l'aide de static_cast<int>.
Une fois cette opération réalisée pour chacune des trois couleurs, nous pouvons stocker le QRgb et passer à l'itération suivante jusqu'à compléter entièrement notre tableau.

```
vector<double> xs{ 0., 0.16, 0.42, 0.6425, 0.8575};
vector<double> yr{ 0., 32. , 237. , 215. , 0. };
vector<double> yg{ 7., 107. , 255. , 170. , 10. };
vector<double> yb{ 100., 183. , 235. , 40. , 15. };

Linear red_color (xs,yr);
Linear green_color (xs,yg);
Linear blue_color (xs,yb);

for (int i = 0; i < 2048; i++) {
  color[i] = qRgb(static_cast<int>(red_color(i/2047.)),
                  static_cast<int>(green_color(i/2047.)),
                  static_cast<int>(blue_color(i/2047.)));
}
```

La deuxième partie permet de calculer d'après la formule donnée dans l'énoncé, la valeur de l'index du tableau correspondant a une valeur | z | ≥ 256,

```
for (int j = 0 ; j<512 ; j++){
   zn=zn*zn+c0;
   if (abs(zn) >= 256.)
   {
       iteration = j;
       break;
   }
}

else{
   double v = log2(log2(abs(zn)*abs(zn)));
   int i = 1024 * sqrt(( iteration + 5 - v));
   i = i % 2048;
   rgb_color = calc_color(i);
}
```

Dans cette partie de code, nous pouvons observer la mise en code des équations donnés par l'énoncé.
Nous effectuons récursivement les calculs de zn jusqu'à obtenir |zn| ≥ 256, nous arrêtons alors la boucle for et gardons en mémoire la première itération pour laquelle l'équation s'est avérée juste. Après avoir effectué les calculs et le modulo 2048, nous récupérons la valeur du tableau correspondant à l'aide de la méthode calc_color(). Nous avons ainsi la couleur correspondant. Dans la fonction ou la partie du code ci-dessus est implémenté, la couleur est renvoyée à l'aide d'une paire. Comme demandé dans l'énoncé.

Calcul de l'image entière :

```
for(int py=0;py< h;++py) {
         for(int px=0;px< w;++px) {
           calc_pix(px,py);
         }
    }
```

L'ordre des boucles a de l'importance à notre avis car suivant la structure de donnée utilisée par QImage, si elle stocke dans une grande suite tous les pixels ligne après ligne alors quand nous parcourons aussi ligne après ligne nous pouvons profiter de ce qui avait été chargé dans le cache en mode page et donc ne pas avoir à faire de requête dans la mémoire plus loin car les données recherchées sont à cotés donc chargée ensemble dans le cache qui y accède par bloc de données.

3) Thread

Pour utiliser le multi-threading disponible sur nos ordinateurs nous avons utilisé le patron donné dans l'énoncé. Nous créons un vecteur de thread, chaque thread appelant la méthode process_sub_image.

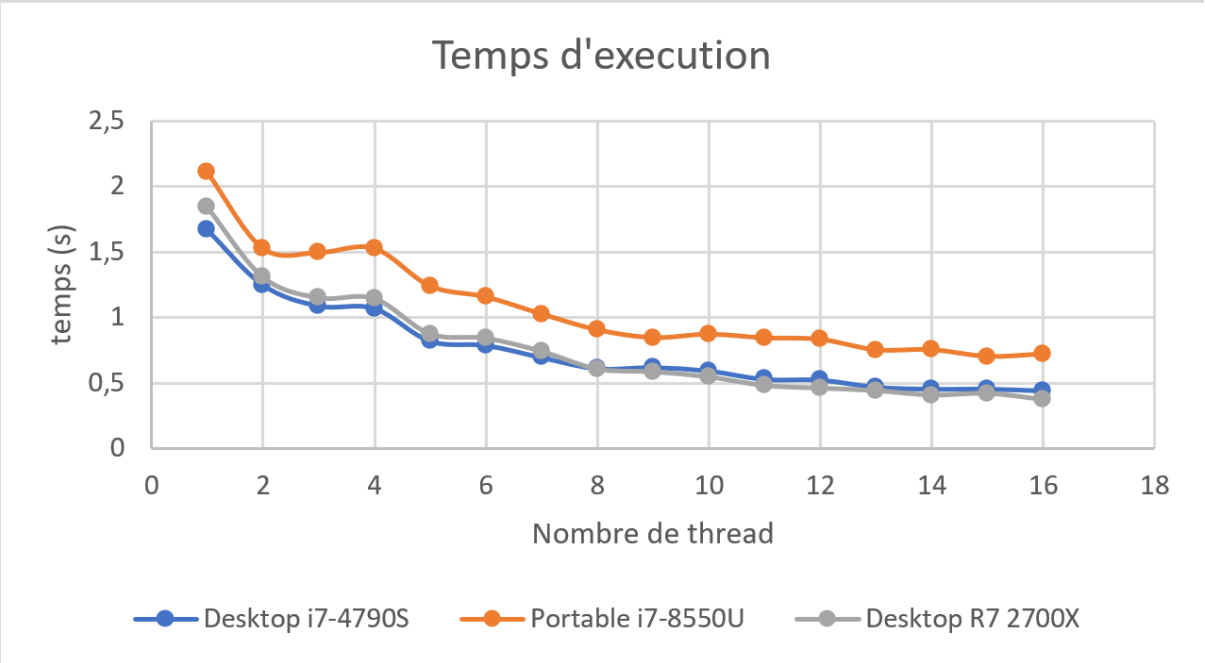
```
std::vector<std::thread> threads;
int max_threads = i;    // number of threads
for (int i = 0; i < max_threads; i++) {
       threads.emplace_back([=]() {
           process_sub_image(i, max_threads);
       });
}
for (auto &thread_elem : threads)   // joins threads
       thread_elem.join();
```

La méthode process_sub_image (ci dessous) permet de ne rendre qu'un bande verticale de l'image. Pour cela, en connaissant le nombre maximal de thread nous pouvons en déduire la largeur de chaque bande : w/m. Et ne parcourir que les pixels qui la composent pour les calculer.

```
void FractalImage::process_sub_image(int i, int m)
{
   for(int py=0;py< h;++py) {
         for(int px=(w/m)*i;px< (w/m)*(i+1);++px) {
           calc_pix(px,py);
         }
    }
}
```

II) Résultats

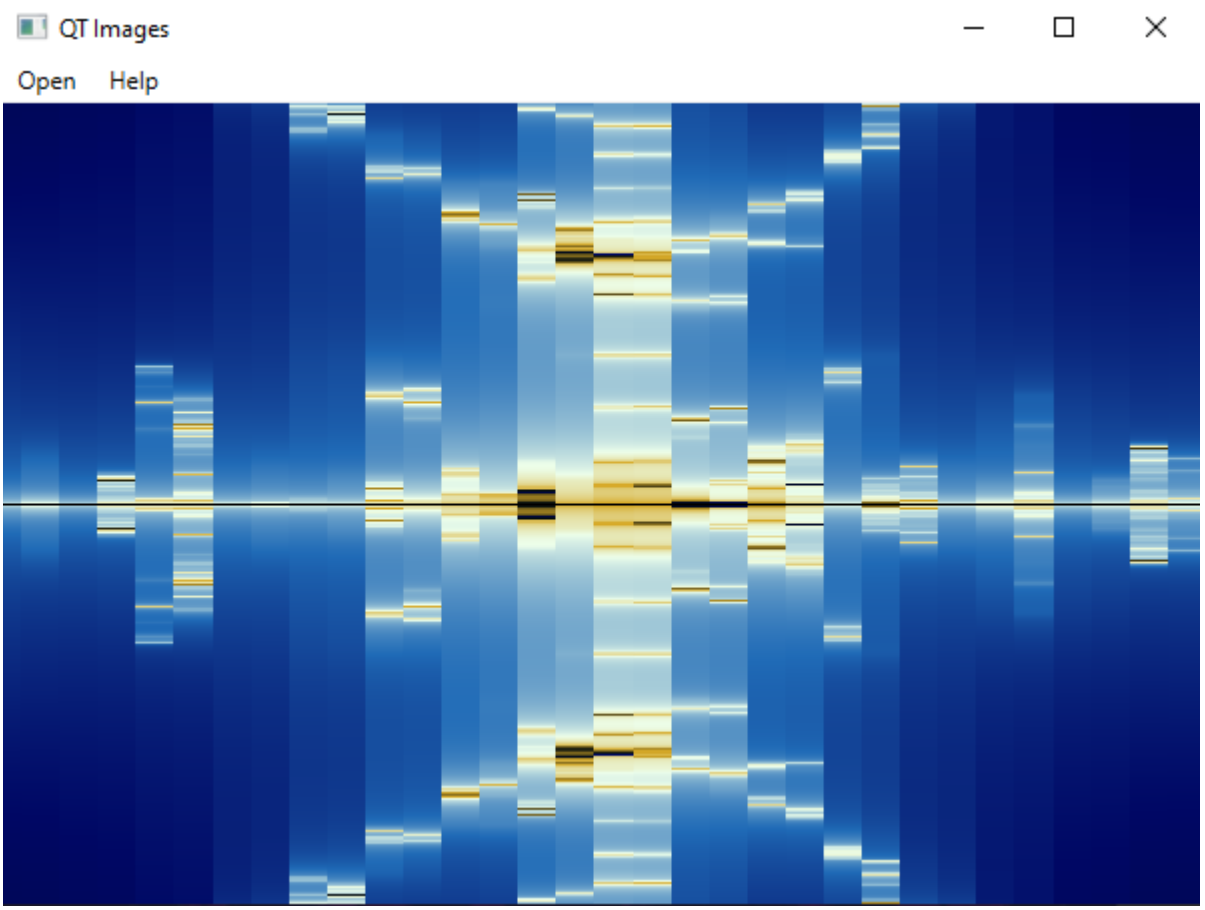
Voici les résultats après avoir exécuté notre programme et mesuré les temps d'exécutions (en secondes) en fonction du nombre de threads sur différents processeurs :



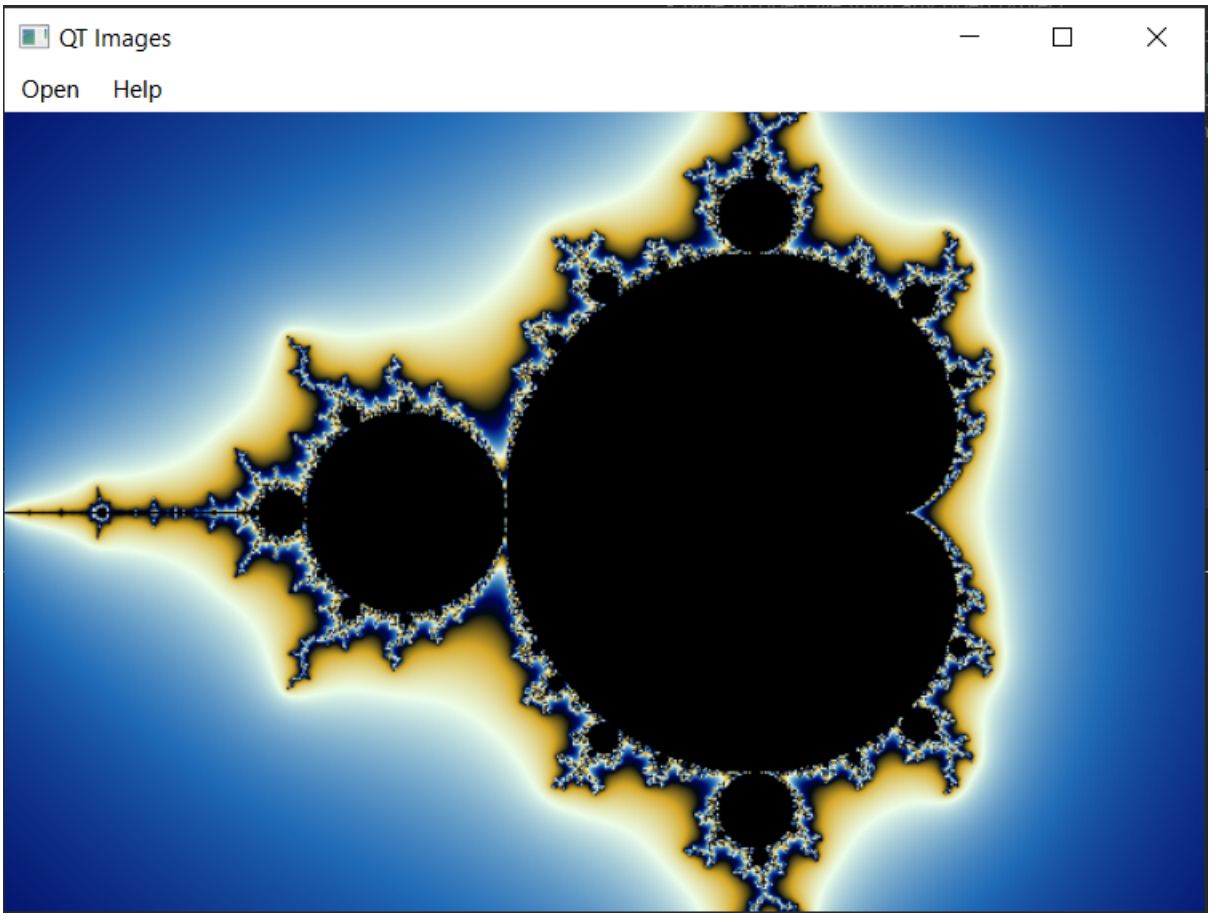
Nous remarquons qu'entre l'exécution sans thread et avec 2 threads nous améliorons bien le temps d'exécution. En revanche pour 2,3 et 4 thread le temps n'évolue que très peu, ce que nous trouvons étrange. Puis le temps décroît jusqu'à ne plus décroitre beaucoup au-delà de 12 threads. Cela nous semblait logique pour les processeurs i7-4790S et i7-8550U qui n'ont que 8 threads mais il apparaît que même le processeur Ryzen qui a 16 threads n'améliore pas de beaucoup le temps d'exécution passé 12 threads.
Nous avons aussi noté que lorsque l'on exécutait notre programme , dans le gestionnaire des tâches nous voyons qu'il n'utilisait pas 100% des ressources processeur.

Qualité de l'image :

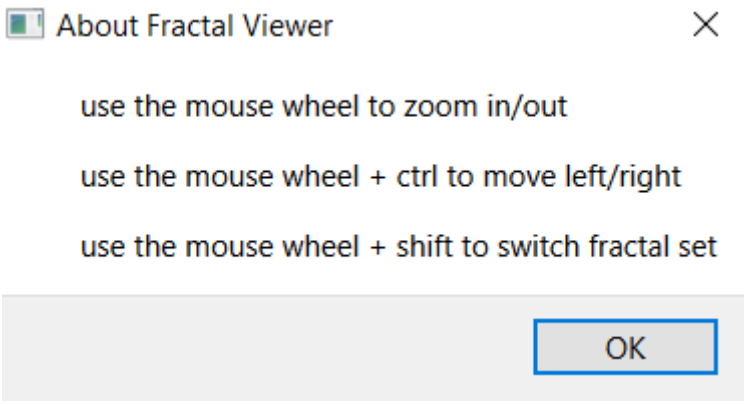
Q3) L'image une fois zoomée apparaît comme "pixelisée", cela est dû à la limite de précision du type double. Les nombres calculés sont si petits qu'il n'est pas possible de zoomer indéfiniment dans l'image. Il arrive un moment où les nombres sont approximés de part la précision de la machine. C'est pourquoi on voit apparaître des zones uniformes qui nous donnent une impression de faible qualité comme sur la capture d'écran ci-dessous.


QT Images
Open
Help

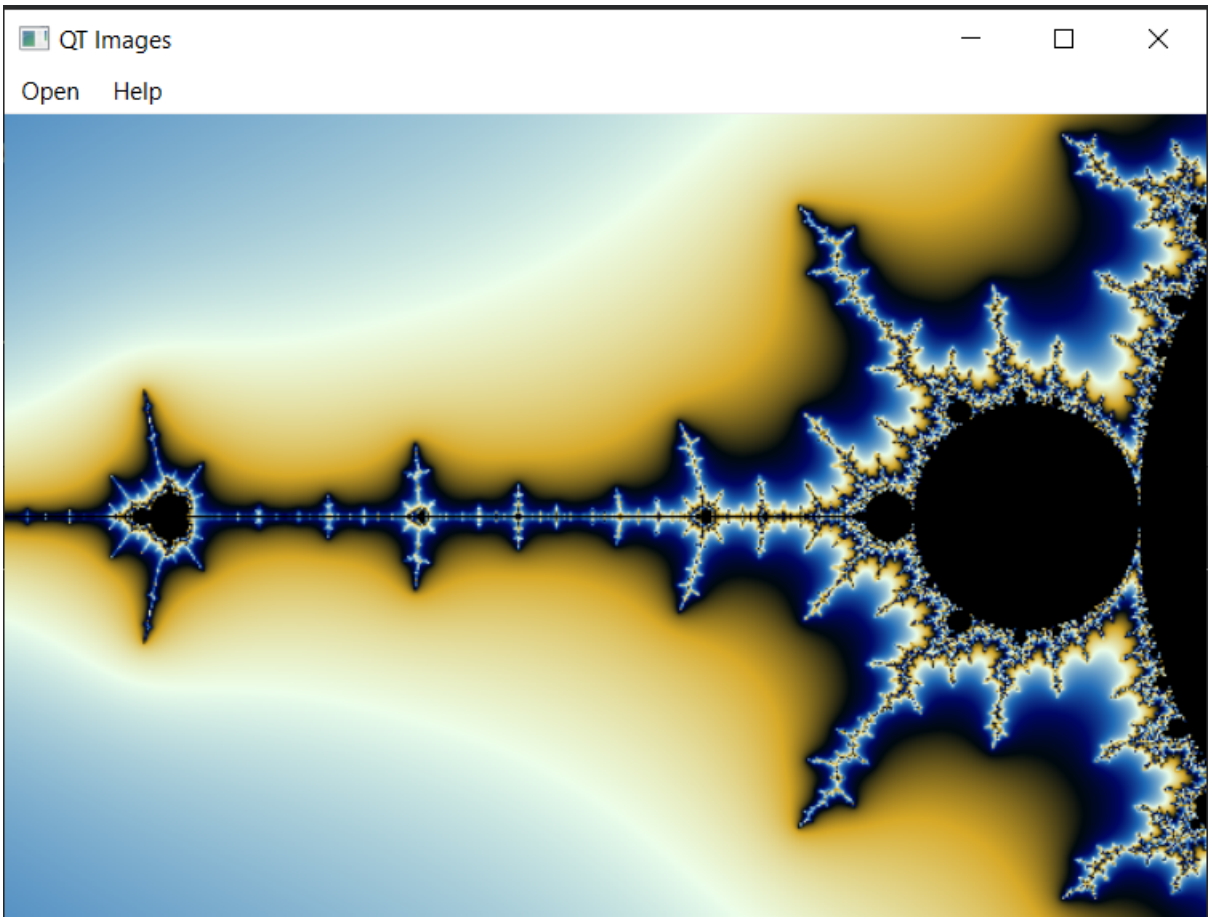
Résultat pour le set de MandelBrot



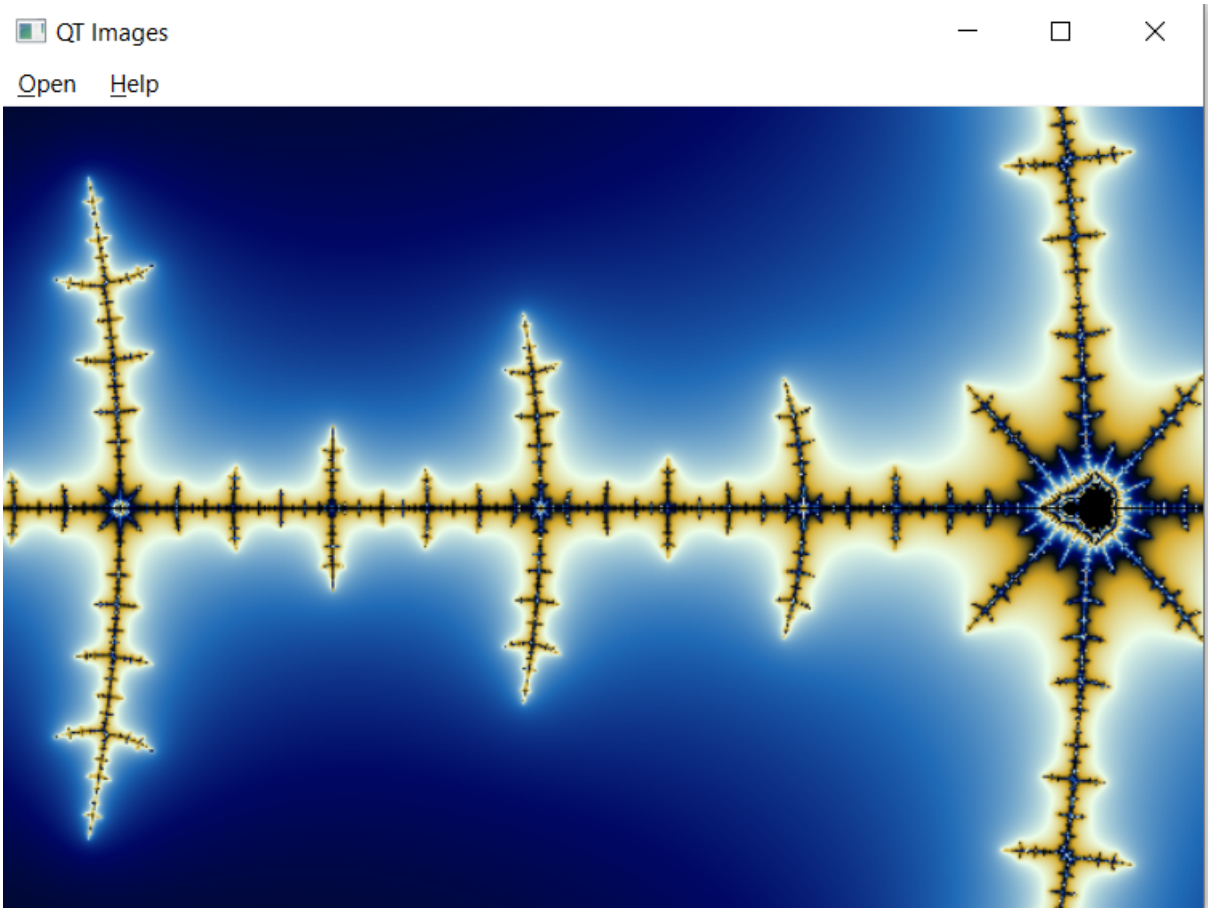
Nous avons implémenté un menu help permettant d'expliquer les différentes commandes pour naviguer dans l'image. Pour cela nous avons simplement override la méthode wheelEvent héritée de QMainWindow dans notre class MainWindow.



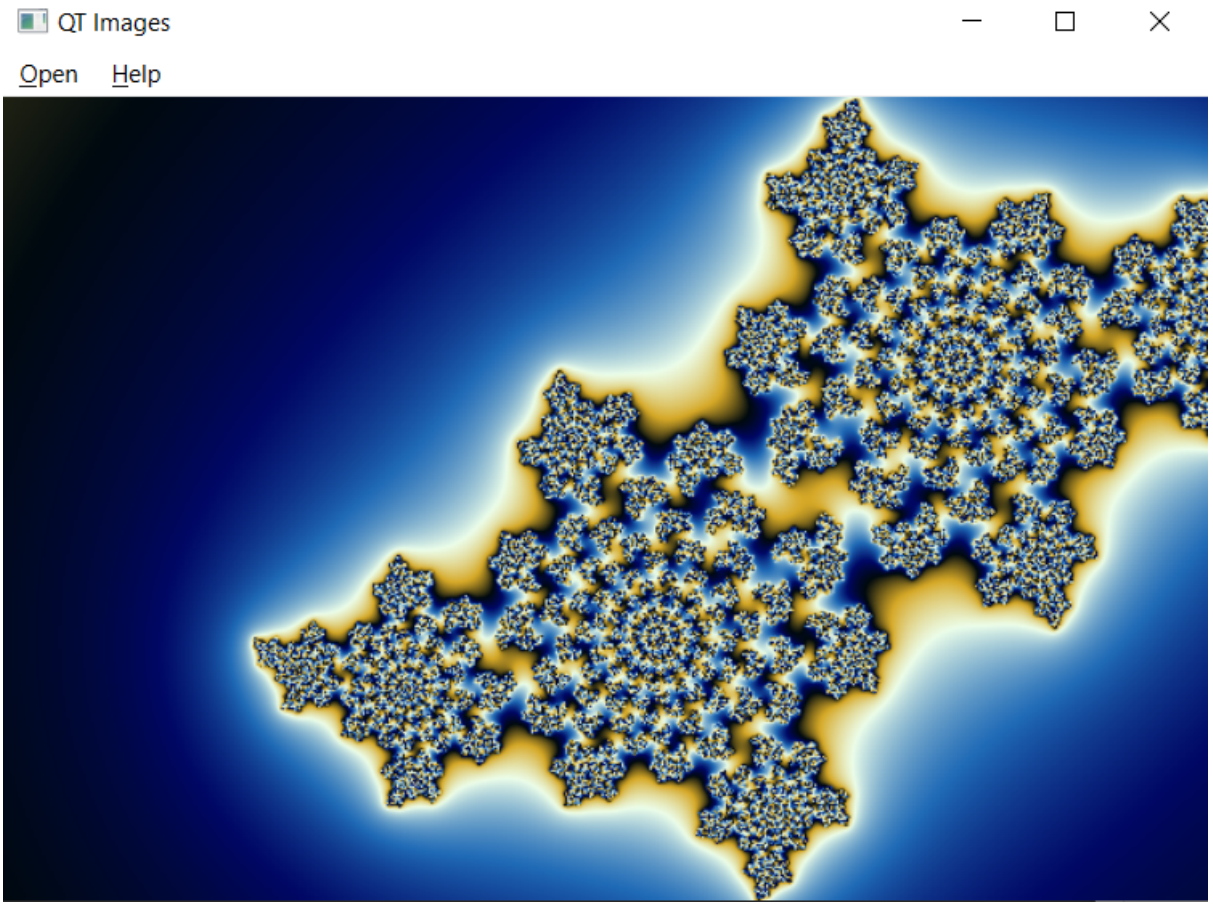
Résultat pour le set de Mandelbrot avec un zoom plus important



Résultat pour le set de Mandelbrot avec un zoom plus important



Résultat pour le set de Julia



Résultat pour le set de Julia avec un zoom plus important