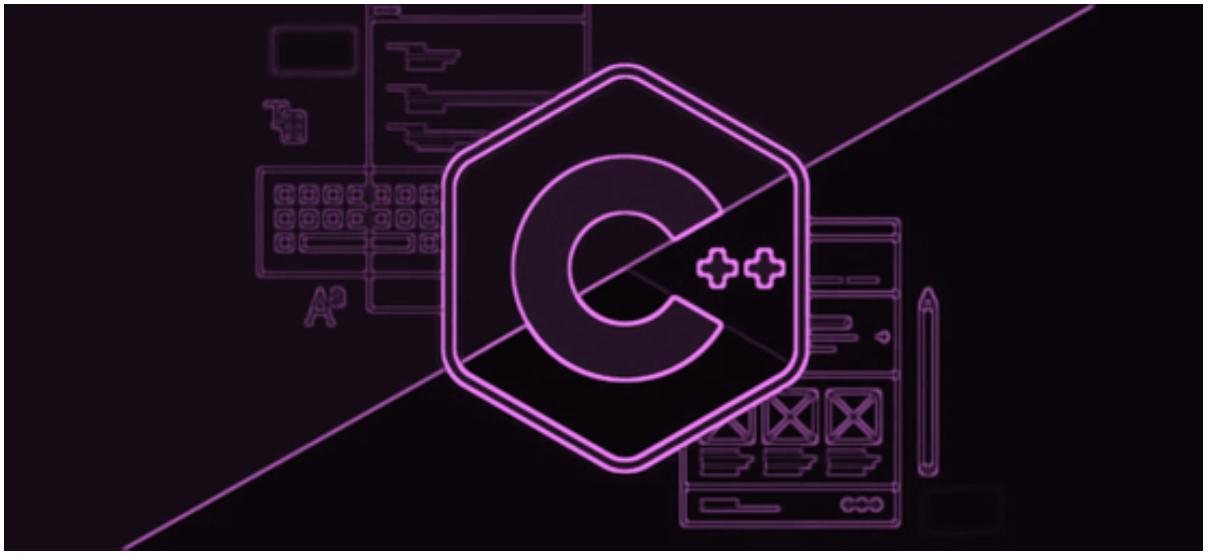
Embedded C++

Lab 4
Lab Mandelbrot Set

18 Mars 2022



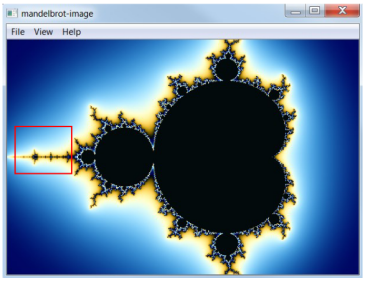
Lab Objectives:

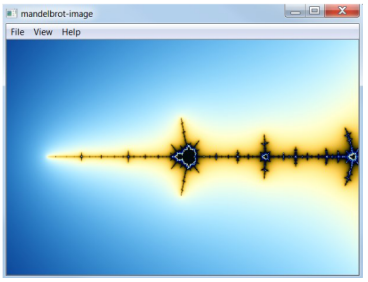
- Draw an image of the Mandelbrot set
- Learn new C++ language features such as friend functions, STL complex numbers and multi-threaded programming.

Authors : GHOBRIAL Sara, HERAU Léa, JONAS Audrey, MOFID Océane

Partie 1 : Mandelbrot Set

Consigne : Dessiner les deux images présentées dans le sujet :





Plusieurs variables sont imposées :

- xc, yc Le centre du rectangle,
- d La demi-hauteur du rectangle,
- La largeur du rectangle 1.5d,
- La largeur et la hauteur de la fenêtre mesuré en pixel w and h.

---

1. Introduction

Dans notre ***mandelbrotimage.cpp***, nous déclarons deux function object:

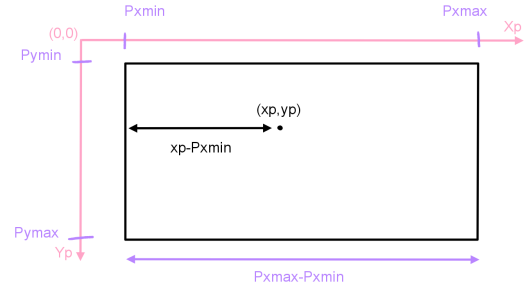
➔ ***v_pixel2rect(...)*** et ***h_pixel2rect(...)***

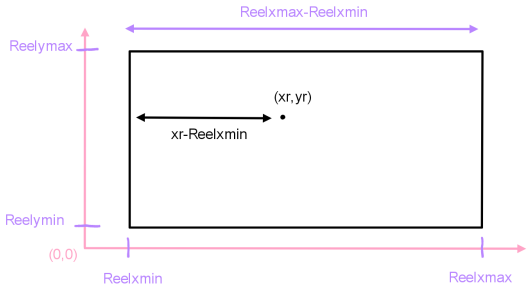
qui vont renvoyer les coordonnées x et y dans le monde réel à partir des coordonnées x ou y dans le domaine des pixels.

Pour cela, nous avons créé un ***pixel2rect.h*** qui ne contient que son constructeur et une méthode ***operator()***.

```
class Pixel2rect_Converter{
private:
    double pix_min, pix_max, real_min, real_max ;

public:
    Pixel2rect_Converter(double p_min, double p_max, double r_min, double r_max) : pix_min(p_min), pix_max(p_max),
        real_min(r_min), real_max(r_max) {}

    double operator()(double p){

        return (p-pix_min)/(pix_max-pix_min)*(real_max-real_min)+real_min;
    }

};
```

Le constructeur prend en paramètre la valeur minimale des lignes ou colonnes en pixel , la valeur maximale, la valeur minimale des coordonnées réelles selon l'axe concerné et la valeur maximale. Le calcul de conversion s'effectue ensuite dans l'opérateur () et se base sur la figure ci-contre grâce à un produit en croix.





Ici, pour retrouver nos coordonnées dans le monde réel à partir de nos coordonnées en pixel nous avons utilisé un produit en croix.

Prenons un exemple : Nous avons un point de coordonnées pixel (xp,yp) qui correspondra au point (xr,yr) dans le monde réel.

Pour retrouver xr nous nous basons sur les différences
(x_p - P_{xmin}), (x_r - $Reel_{xmin}$), ($Reel_{xmax}$ - $Reel_{xmin}$) et (P_{xmax} - P_{xmin}).

Pixel	Réel
x_p - P_{xmin}	x_r - $Reel_{xmin}$
P_{xmax} - P_{xmin}	$Reel_{xmax}$ - $Reel_{xmin}$

En effet, le produit en croix ci-dessus. Nous donne le calcul suivant :

$$xr = \frac{(xp - Pxmin) * (Reelxmax - Reelxmin)}{(Pxmax - Pxmin)} + Reelxmin$$

De la même manière nous retrouvons :

$$yr = \frac{(yp - Pymin) * (Reelymax - Reelymin)}{(Pymax - Pymin)} + Reelymin$$

2. First Step: A Two-Color Image

Maintenant que nous savons convertir nos coordonnées d'un repère dans à l'autre, nous souhaitons dans un premier temps créer une fractale de type Mandelbrot. Pour cela, nous instancions notre constructeur de classe :

```
MandelbrotImage::MandelbrotImage(const int width, const int height): QImage (width, height, QImage::Format_RGB32){
    //create the converters
    Pixel2rect_Converter h_pixel2rect(0, width,cx-1.5*d, cx+1.5*d);
    Pixel2rect_Converter v_pixel2rect(0, height, cy+d, cy-d);

    for (int py = 0; py <height; py++) {
        double ry = v_pixel2rect(py);
        for (int px = 0; px < width; px++) {
            double rx = h_pixel2rect(px);
            QRgb rgb_color = calc_in_out(rx, ry);
            setPixel(px, py, rgb_color);
        }
    }
```

Il va parcourir notre future image d'abord selon les coordonnées ***pixel*** verticalement, convertir la coordonnée en coordonnée ***réelle*** puis parcourir les coordonnées ***pixel*** horizontalement et faire de même pour cette dernière. Nous faisons ensuite appel à notre fonction ***calc_in_out*** qui va calculer si le pixel est à l'intérieur ou à l'extérieur de la fractale.

```
/*
 * Role : returns the QRgb color of the pixel of coordinates(rx,ry)
 *        depending on if it's inside the MandelbortImage.
 */
QRgb MandelbrotImage::calc_in_out(double rx, double ry)
{
    //c0= x0 +iy0
    //z0 = 0
    std::complex c0(rx,ry);
    std::complex z(0.0,0.0);
    QRgb color= qRgb(0, 0, 0);//black


    for(int n=0; n<512; n++){
        z = z*z+c0;
        double module= abs(z);
        if(module>2){
            color=qRgb(255,218,103);
        }
    }
    return color;
}
```

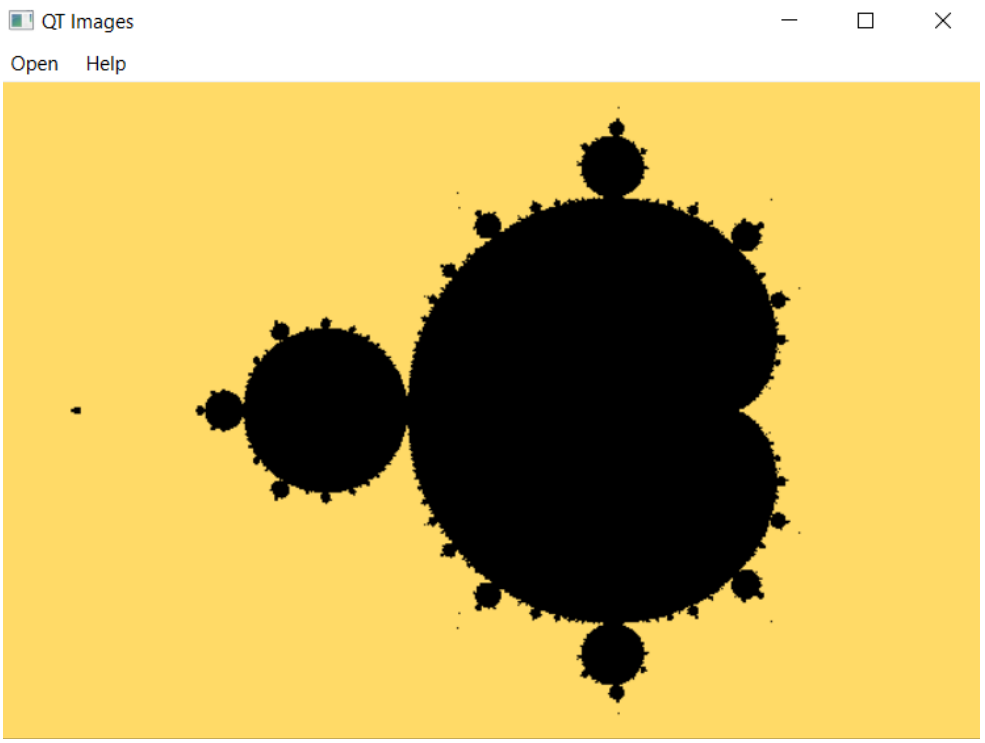
Cette fonction renvoie une **QRgb color** qui va être déterminée d'après l'algorithme de Mandelbrot. Nous avons décidé de considérer par défaut que le pixel appartient à la figure et donc, de lui attribuer la couleur noir. On réalise ensuite la série mathématique $z_{n+1} = z_n^2 + c_0$ avec $c_0 = x_0 + y_0$ et en prenant $n_{MAX} = 512$. Pour chaque itération, on calcule le module du complexe et on regarde si ce dernier est inférieur à 2; si c'est le cas, la couleur du pixel est fixée à (255, 218, 103) et on arrête les itérations.

Voici les déclarations réalisées dans notre ***mandelbrotimage.h***, nous avons ajouté les paramètres de la fractale voulue, les dimension de notre image ainsi que la fonction ***calc_in_out*** :

:

```
class MandelbrotImage : public QImage
{
public:
  MandelbrotImage(const int width, const int height);


private:
  //image parameters
  double cx = -0.5;
  double cy = 0;
  double d = 1.0;

  double px_min = 0.0;
  double px_max = 599.0;
  double py_min = 0.0;
  double py_max = 399.0;

  //methods
  QRgb calc_in_out(double rx, double ry);
```

Voici le résultat obtenu, identique à ce qui était demandé :



Partie 2 : Colored Mandelbrot Set

Consigne : Générer une fractale de Mandelbrot présentant un gradient de couleurs prédéfinies et mesurer le temps d'exécution de notre programme.

---

1. Color Palette

Dans un premier temps, nous créons dans notre fichier ***mandelbrotimage.h*** un vecteur de ***QRgb*** qui contiendra nos 2048 couleurs :

```
//vector of 2048 RGB colors
std::vector<QRgb> tab_colors;
```

Nous appelons ***create_gradient_colors()*** au début de notre constructeur :

```
/*
 * Role : Constructs a MandelbrotImage of size width x height
 */
MandelbrotImage::MandelbrotImage(const int width, const int height): QImage (width, height, QImage::Format_RGB32){
    //create the vector of 2048 colors
    create_gradient_colors();
    //create the converters
    Pixel2rect_Converter h_pixel2rect(0, width,cx-1.5*d, cx+1.5*d);
    Pixel2rect_Converter v_pixel2rect(0, height, cy+d, cy-d);

    for (int py = 0; py <height; py++) {
        double ry = v_pixel2rect(py);
        for (int px = 0; px < width; px++) {
            double rx = h_pixel2rect(px);
            QRgb rgb_color = calc_in_out(rx, ry);
            setPixel(px, py, rgb_color);
        }
    }
```

Cette nouvelle méthode ***create_gradient_colors()*** va venir insérer nos ***QRgb*** dans le vecteur. Nous commençons par instaurer le nombre de couleur avec ***max_color*** et après avoir alloué assez d'espace dans le tas, nous itérons sur ***max_color*** et nous venons calculer notre index. Nous allons ensuite faire appel à ***interpolColors*** qui va retourner la couleur RGB selon l'interpolation réalisée avec ***xs***, ***yr***, ***yg*** et ***yb***.

```
/*
 * Role:Create the vector of nb_color colors
 */
void MandelbrotImage::create_gradient_colors(){
    //nb of colors we want
    double max_color= 2048;
    tab_colors.reserve(max_color);
    //we iterate to calculate the 2048 RGB colors
    for(int i=0; i<max_color; i++){
        //we add to the vector the new color
        double color = static_cast<double>(i)/max_color;
        tab_colors.emplace_back(interpolColors(color));
    }
}
```

La méthode ***interpolColors()*** va renvoyer un qRGB qui contient les valeurs des composantes R, G et B après interpolation. Etant donné que le type qRgb est composé de trois entiers et que ***get_value()*** renvoie un double, nous devons convertir le type en entier grâce à ***static_cast()***.

```
/*
 * Role: create an QRgb object with the R G B values after interpolation
 */
QRgb MandelbrotImage::interpolColors(double x) {
    //compute our R,G,B components
    int R = static_cast<int>(interpolation_R.get_value(x));
    int G = static_cast<int>(interpolation_G.get_value(x));
    int B = static_cast<int>(interpolation_B.get_value(x));
    return qRgb(R,G,B);
}
```

Nous venons ajouter à notre fichier ***mandelbrotimage.h*** les vecteurs de points donnés qui nous permettent de réaliser l'interpolation, nous ajoutons également les objets Interpolator de notre classe Interpolator pour chaque composante RGB.

```
  //color points
  std::vector<double> xs_{ 0., 0.16, 0.42, 0.6425, 0.8575};
  std::vector<double> yr_{ 0., 32. , 237. , 215. , 0. };
  std::vector<double> yg_{ 7., 107. , 255. , 170. , 10. };
  std::vector<double> yb_{ 100., 183. , 235. , 40. , 15. };

  //we create 3 interpolators objects (R, G, B)
  Interpolator interpolation_R{xs_,yr_};
  Interpolator interpolation_G{xs_,yg_};
  Interpolator interpolation_B{xs_,yb_};
```

Nous créons la classe ***Interpolator*** qui aura pour variables membres ***ai_*** et ***bi_*** deux vecteurs de doubles correspondant, respectivement, à la pente et à l'ordonnée à l'origine des segments formés par les vecteurs ***xs_*** (aussi une variable membre) et ***y*** qui sera par la suite passé en paramètre dans le constructeur.

```
class Interpolator : public QImage
{
private:
    std::vector<double> ai_;
    std::vector<double> bi_;
    std::vector<double> xs_;
```

Le constructeur de la classe ainsi qu'une méthode ***get_value*** sont également ajoutés.

```
public:
   /*
    * Role: Constructs an Interpolator object which sets up the ai and bi coefficients
    */
    Interpolator(std::vector<double> xs, std::vector<double> &y) : xs_(xs) {
       y.push_back(y.front());
       bi_ = y;
       xs_.push_back(1.0);
        for (unsigned int i = 0; i < xs_.size() - 1; i++) {
            ai_.push_back((y[i + 1] - y[i]) / (xs_[i + 1] - xs_[i]));
         }
    }

    /*
     * Role : Returns the @x interpolation with the current interpolator object
     */
    double get_value(const double x) const {
        unsigned long i = 0;
        while (i< xs_.size()-1 && x > xs_[i+1]) i++;
        return std::clamp(ai_[i] * (x - xs_[i]) + bi_[i], 0., 255.);
    }

};
```

Dans la liste d'initialisation du constructeur on associe la variable membre ***xs_*** et le vecteur ***xs*** passé en paramètre, après cela nous itérons sur la taille du vecteur ***xs_*** et nous calculons les coefficients de pente que nous insérons dans le vecteur ***ai_***.

Pour la méthode ***get_value***, elle va prendre un double ***x*** en paramètre et tester dans quel intervalle de ***xs_*** ce dernier se trouve et va retourner l'ordonnée correspondant à l'équation de droite de l'intervalle. Nous utilisons ensuite la méthode ***std::clamp*** afin d'éliminer toute valeur qui serait en dehors de [0;255], cette méthode prend en paramètre une valeur, une borne inférieure puis une borne supérieure et va retourner la valeur passée en paramètre si elle appartient à l'intervalle ou l'une des bornes si elle sort de l'intervalle.

2. Link between Points and Color

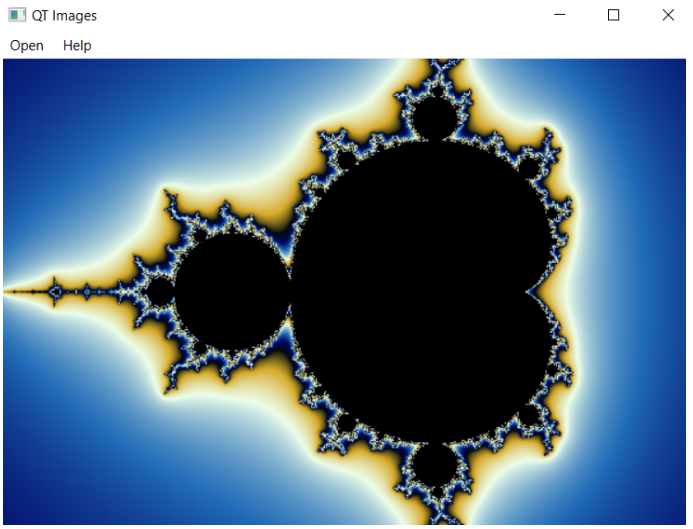
Nous devons maintenant calculer l'index du tableau ***tab_color*** auquel le pixel va prendre sa couleur. Pour ce faire, nous reprenons notre fonction ***calc_in_out*** auquel nous rajoutons un ***if*** :

```
/*
 * Role : returns the QRgb color of the pixel of coordinates(rx,ry)
 *        depending on if it's inside the MandelbortImage.
 */
QRgb MandelbrotImage::calc_in_out(double rx, double ry)
{
    //c0= x0 +iy0
    //z0 = 0
    std::complex c0(rx,ry);
    std::complex z(0.0,0.0);
    QRgb color= qRgb(0, 0, 0);//we initialise with black
    bool is_inside =true;
    int i = 0.0;


    for(int n=0; n<512; n++){
        z = z*z+c0;
        double module= abs(z);
        if(module>2){
            is_inside=false;
         }
         if(!is_inside and module>=256){
                 double v = log2(log2(module*module));
                 i = static_cast<int>(1024*sqrt(n+5-v))%2048;
                 color = tab_colors[i];
                 return color;
         }
        }
       return color;
    }
```

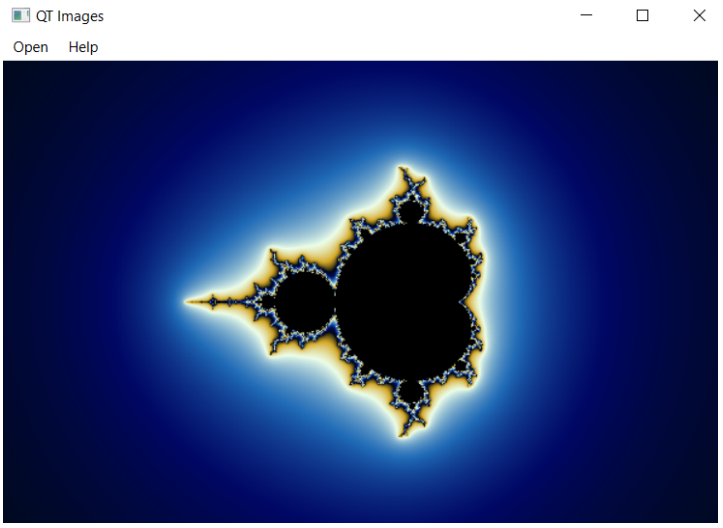
Comme l'explique l'énoncé, notre index ***i*** est calculé par une formule basée sur le nombre d'itérations réalisées afin d'avoir $|z_n| \geq 256$. Dès que le module d'un des z_n est >2 une variable booléenne ***is_inside*** initialisée à ***true*** passe à ***false*** et ne changera plus de valeur. A chaque itération, si le pixel a été détecté ***outside*** et que l'itération courante engendre ***module>=256***, alors on doit attribuer la couleur au pixel. Nous implémentons ensuite la formule et affectons la couleur à l'index ***i*** de ***tab_colors*** à ***color*** que nous retournons.

Voici notre fractale de Mandelbrot avec un gradient de couleurs :

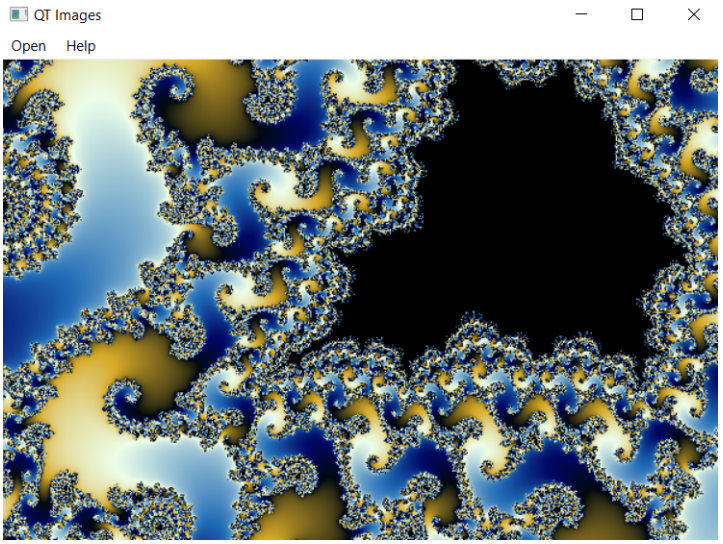
avec d = 1



avec d = 2



Pour cx = -0.7436447860; cy = 0.1318252536; d = 0.0000029336



3. Measuring Time

Pour mesurer le temps d'exécution de notre programme pour dessiner une image, nous importons la library ***chrono*** et nous utilisons l'horloge ***high_resolution_clock*** pour mesurer l'intervalle. Nous relevons le temps avant l'exécution à l'aide de la fonction ***now()*** que nous stockons dans ***start*** puis nous relevons le temps de nouveau après l'appel au constructeur ***MandelbrotImage*** que nous affections à ***end***.
Nous réalisons la différence entre les deux et nous utilisons la méthode de cast de la library afin d'obtenir des microsecondes et nous affichons sur la sortie standard le résultat.

```
//recording the time before the execution
auto start = std::chrono::high_resolution_clock::now();

//create the Mandelbrotimage
MandelbrotImage mandelbrot_image(mandelbrot_width, mandelbrot_height);

//recording the time after the execution
auto end = std::chrono::high_resolution_clock::now();
auto interval = end-start;
int64_t t = std::chrono::duration_cast<std::chrono::microseconds>(interval).count();

//prints the execution time
std::cout << "INFO: image calculated in  " << t << " us\n";
```

```
INFO: image calculated in  2883175 us
```

NB : la capture réalisée ici et plus bas (avec le nouvel affichage) ont été réalisées après exécution du programme sur un ordinateur assez peu performant, elles ne sont là que pour montrer la différence d'affichage.

4. Friend functions

Afin d'améliorer la lecture du résultat, nous créons la classe ***Commify*** où le constructeur va faire correspondre la valeur passée en paramètre de l'objet à sa variable membre de type ***int64_t***.
Comme ce que nous souhaitons améliorer est l'affichage, nous allons faire une surcharge de l'opérateur **<<** qui va venir transformer l'objet en ***string*** puis insérer une virgule tous les trois caractères à l'aide de ***insert***. Nous retournons ensuite un ***ostream*** qui a récupéré la ***string*** :

```
class Commify {
 private:
    int64_t duration_;
 public:
    explicit Commify(int64_t value) : duration_{value} {}

    friend std::ostream &operator << (std::ostream &os, const Commify &c){
        std::string s = std::to_string(c.duration_);
        int comma_pos = s.length() - 3;
        while (comma_pos > 0) {
            s.insert(comma_pos, ",");
            comma_pos -= 3;
        }
        os << s;
        return os;
    }
};
```

Voici le code mis à jour avec cette classe :

```
    //recording the time before the execution
    auto start = std::chrono::high_resolution_clock::now();

    //create the Mandelbrotimage
    MandelbrotImage mandelbrot_image(mandelbrot_width, mandelbrot_height);

    //recording the time after the execution
    auto end = std::chrono::high_resolution_clock::now();
    auto interval = end-start;
    int64_t t = std::chrono::duration_cast<std::chrono::microseconds>(interval).count();
    Commify exe_time(t);

    //prints the execution time
    std::cout << "INFO: image calculated in  " << exe_time << " us\n";
```

Et la sortie avec le nouvel affichage :

```
INFO: image calculated in  2,843,097 us
```

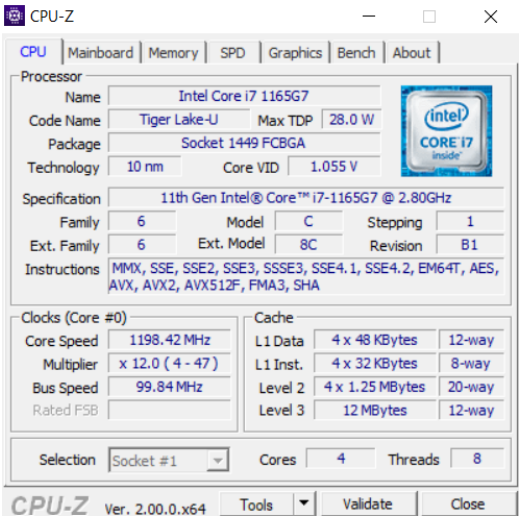
Partie 3 : Improving Performances

Consigne: Optimisation du code en réduisant les opérations qui prennent du temps par l'exploitation de l'exécution en parallèle.

---

1. Know Your Hardware

A l'aide de ***CPU-Z***, nous trouvons des informations clés qui vont nous aider à améliorer la performance de notre programme, ici une capture d'écran de l'exécution de l'outil sur l'un de nos ordinateurs personnels :



On apprend que l'ordinateur possède un processeur ***Intel*** de ***11ème génération*** qui fonctionne à une fréquence de ***2,8 GHz***. Enfin, ce qui nous intéresse c'est le nombre de coeurs et donc le nombre de threads car le nombre de threads est égal à 2 fois le nombre de coeurs, cet ordinateur possède ***4 coeurs → 8 threads***, il peut donc exécuter ***8 instructions simultanément***.

2. Using « threads » for Parallel Computation

Q. The double loop we have done so far has always been with the outer loop on each line and the inner loop on each pixel of the line, but the result would be equivalent if you try to swap these two loops. Can you justify why our first choice is far better?

➔ Pour le cas normal :
On peut associer notre image à un tableau à 2 dimensions et il s'avère que ces derniers sont stockés dans la mémoire ligne par ligne. Ainsi, chaque élément d'une ligne possède des adresses consécutives, rendant le parcours d'un tableau plus facile car il ne faut qu'incrémenter l'adresse de la taille d'un élément.
Si on échange les boucles, c'est-à-dire qu'on boucle d'abord sur les colonnes puis sur les lignes, les éléments n'auront plus des adresses consécutives et le processeur devra donc calculer à chaque itération l'adresse du prochain élément qui se trouve "*nbr d'éléments d'une ligne * taille d'un élément*" plus loin en terme d'adresse.

Comme demandé, une méthode ***process_sub_image*** a été ajoutée, qui réalise la double boucle mais sur les pixels d'une sous image calculée à partir du nombre de threads.

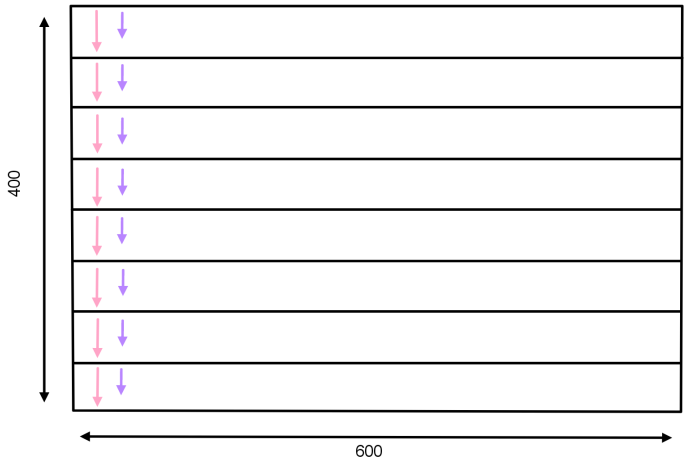
```
void MandelbrotImage::process_sub_image(int current_thread, int max_threads){

    int sub_image_height = height() / max_threads;
    int yi = current_thread * sub_image_height;
    Pixel2rect_Converter h_pixel2rect(0, width(),cx-1.5*d, cx+1.5*d);
    Pixel2rect_Converter v_pixel2rect(0, height(), cy+d, cy-d);


    for (int py = yi; py < sub_image_height + yi; py++) {
        double ry = v_pixel2rect(py);
        for (int px = 0; px < width(); px++) {
            double rx = h_pixel2rect(px);
            QRgb rgb_color = calc_in_out(rx, ry);
            setPixel(px, py, rgb_color);
        }
    }
}
```

Ceci nous a amené à apporter des modification sur notre constructeur afin d'implémenter les threads et leur exécution simultanée :

```
/*
 * Role : Constructs a MandelbrotImage of size width x height
 */
MandelbrotImage::MandelbrotImage(const int width, const int height): QImage (width, height, QImage::Format_RGB32){
    //create the vector of 2048 colors
    create_gradient_colors();
    //allows the parallelism in multi-thread
    std::vector<std::thread> threads;
    const int max_threads = 4;

    for(int i = 0; i < max_threads; i++) {
        threads.emplace_back([=]() {
            process_sub_image(i, max_threads);
        });
     }
    for(auto &thread_elem :threads){
        thread_elem.join();
    }
}
```

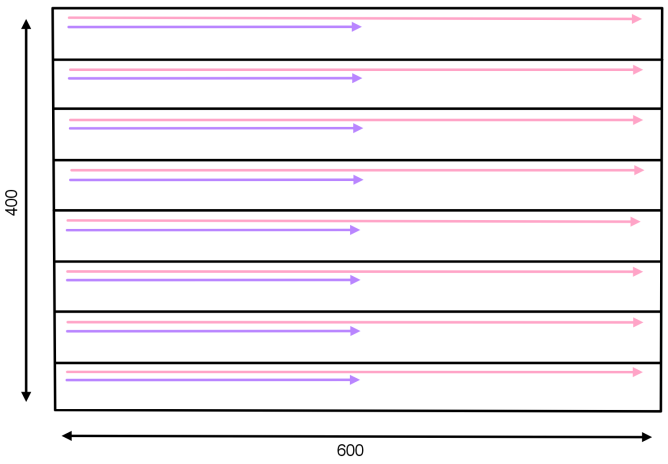
Nous créons donc un vecteur de threads dans lequel à la fois nous insérons et créons les sous images à l'aide de ***emplace_back*** dans un for itéré sur le nombre max de threads ici à 4. Nous avons ensuite une autre boucle qui va appeler ***join()*** pour chaque thread ce qui va mettre en attente le thread principal, c'est-à-dire le constructeur, en attendant que les threads passés en objet courant se terminent.

Ci-dessous 2 schémas expliquant comme nous itérons dans nos boucles à l'aide des threads : à gauche lorsqu'on boucle sur les ***x*** (pixel de chaque ligne) en premier, à droite sur les ***y*** (chaque ligne)



Avec threads = 8,
En tout 600/8 =75 boucles

Exécuté en même temps boucle 1
Exécuté en même temps boucle 2



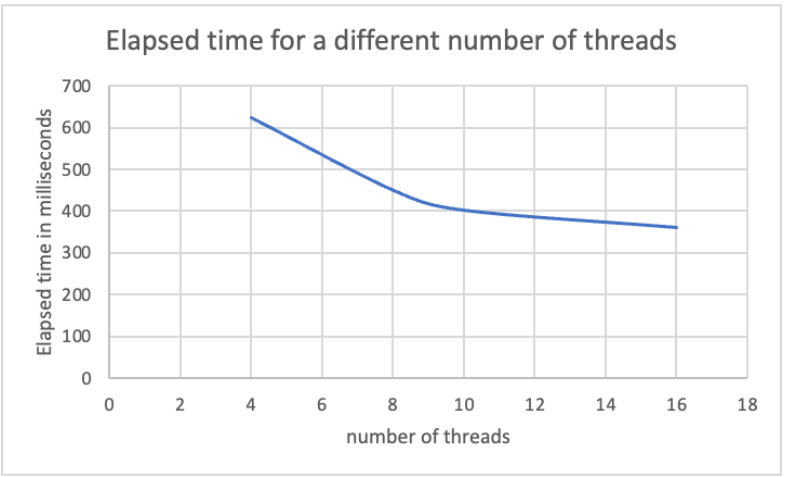
Avec threads = 8,
En tout 400/8 = 50 boucles

Exécuté en même temps boucle 1
Exécuté en même temps boucle 2

➔ Pour le cas des threads :
Dans la première situation nous allons boucler dans une première boucle de 75 itérations tandis que dans la deuxième situation ou la première boucle concerne les y, la première boucle n'aura que 50 itérations.

Q. Plot a simple graph of the elapsed time as a function of the number of threads, comment your results.

On peut observer qu'à partir de ***4 threads***, le temps d'exécution se stabilise aux alentours de ***0,3 s***. On peut en conclure que pour une telle utilisation, un grand nombre de threads utilisés en parallèle n'est pas utile. Le processus de multithreading est très utile pour exécuter des tâches de grande envergure en parallèle en maximisant les processeurs de nos ordinateurs.



Pour max_thread=4

```
INFO: image calculated in  624,419 us
```

Pour max_thread=8

```
INFO: image calculated in  449,779 us
```

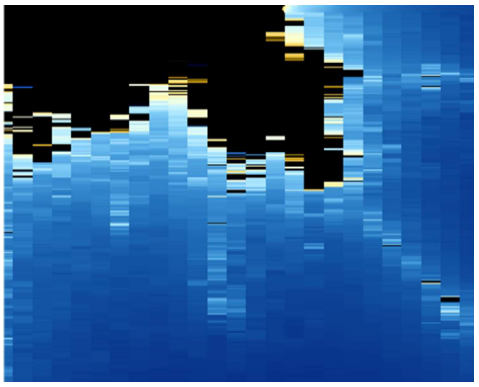
Pour max_thread=10

```
INFO: image calculated in  401,569 us
```

Pour max_thread=16

```
INFO: image calculated in  359,973 us
```

3. Image quality



Q. Explain why the image quality is so low?

L'image est de très faible qualité car en zoomant c'est-à-dire en diminuant ***d***, nous réduisons la dimension réelle de la figure et donc des valeurs prises par ***x*** et ***y.*** Ainsi pour une dimension d'image inchangée, un nombre plus conséquent de pixels appartiendront au même intervalle en coordonnées réels et se verront donc affecter la même couleur par la fonction d'interpolation.

Partie 4 : Extra credit

User interactions and Julia set

Nous avons adapté notre programme afin de pouvoir alterner entre les deux images: **Julia** et **Mandelbrot**, ceci en utilisant la touche “T”.

Pour cela nous utilisons les éléments suivants :

- Surcharge de la fonction **keyPressEvent** de Qt (voir image ci-dessous) afin d’écouter les différents appuis de boutons (“T” pour le toggle, mouvements haut/bas/droite/gauche et zoom +/-).

```
private:
  void create_menus();
  void create_actions();
  void keyPressEvent(QKeyEvent *event) override;
```

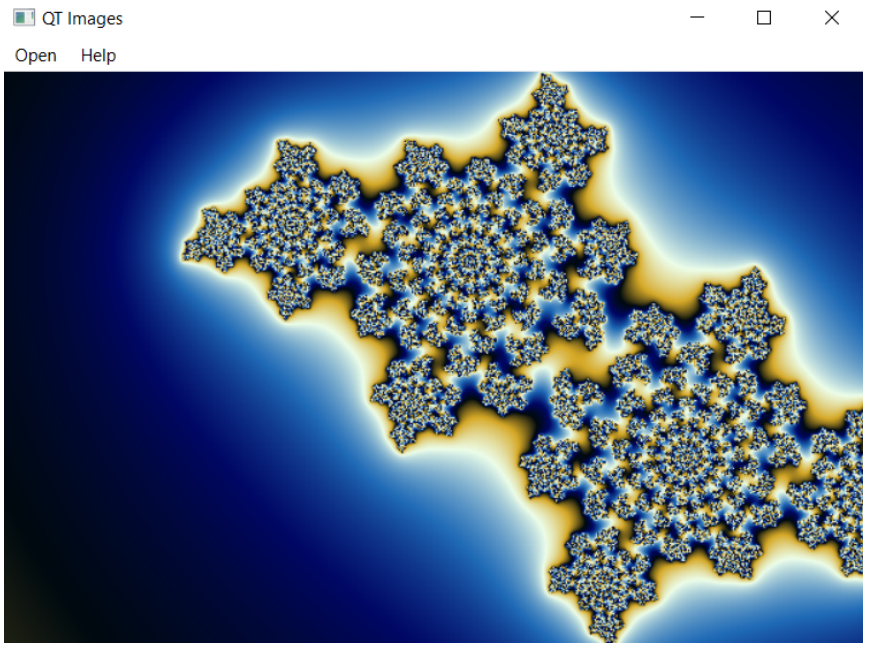
```
void MainWindow::keyPressEvent(QKeyEvent *event) {
  bool updateNeeded = false;

  switch (event->key()) {
  case Qt::Key_T:
      imageType_ = imageType_ == ImageType::Julia ? ImageType::Mandelbrot : ImageType::Julia;
      updateNeeded = true;
      break;

  case Qt::Key_Left:
      xc_ -= 0.1 * d_;
      updateNeeded = true;
      break;

  case Qt::Key_Right:
      xc_ += 0.1 * d_;
      updateNeeded = true;
      break;

  case Qt::Key_Up:
      yc_ += 0.1 * d_;
      updateNeeded = true;
      break;

  case Qt::Key_Down:
      yc_ -= 0.1 * d_;
      updateNeeded = true;
      break;

  case Qt::Key_Plus:
      d_ = d_ / 1.5;
      updateNeeded = true;
      break;

  case Qt::Key_Minus:
      d_ = d_ * 1.5;
      updateNeeded = true;
      break;

  default:
      QMainWindow::keyPressEvent(event);
  }

  //Update image only if needed
  if (updateNeeded) {
      slot_load_fractal_image();
  }
}
```

- Une classe mère **FractalImage** et deux classes filles: **MandelbrotImage** et **JuliaImage**. Ces deux dernières surchargent la fonction ***calc_in_out***: qui calcule si un pixel est à l'intérieur ou à l'extérieur de la fractale selon son équation.

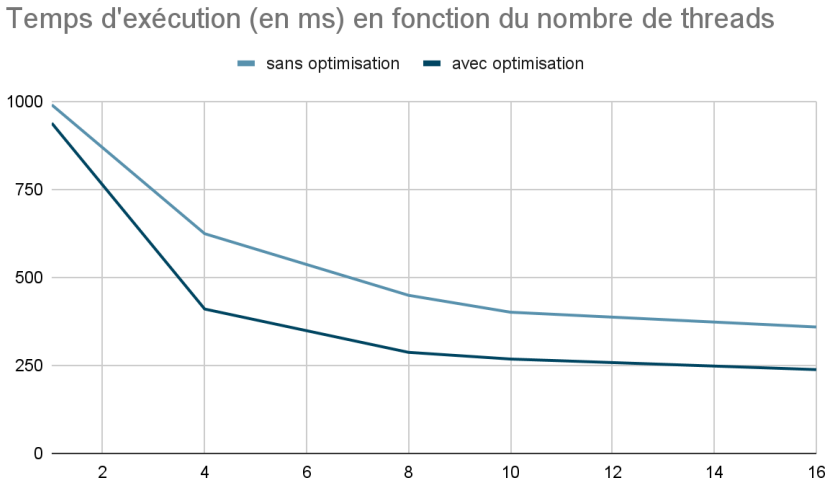
- Pour distinguer les deux fractales nous avons créé un type **ImageType** à l'aide d'un enum class. Cette information est sauvegardée dans la classe **MainWindow** avec les autres informations relatives au centre et zoom de l'affichage.

```
  enum class ImageType {
      Mandelbrot, Julia
  };

  ImageType imageType_ = ImageType::Mandelbrot;
  double xc_ = -0.5;
  double yc_ = 0.0;
  double d_ = 1.0;
};
```

Voici ce qui s'affiche lorsqu'on utilise la touche T :



Nous avons effectué des optimisations de code en utilisant des foncteurs afin d'éviter de répéter les calculs des courbes d'interpolations des couleurs (R,G,B), ainsi que les calculs de conversion du pixel au monde réel. Les résultats en terme de temps d'exécution sont les suivants :



Exemples d'optimisations :

Le foncteur ***H_Pixel2Rect*** est utilisé dans process_sub_image, et est instancié une fois en tant que variable membre de la classe *FractalImage*.

```
H_Pixel2Rect hPixel2Rect{cx_, d_, px_min_, px_max_};
V_Pixel2Rect vPixel2Rect{cy_, d_, py_min_, py_max_};
```

```
void FractalImage::process_sub_image(int current_thread, int max_threads){
    int sub_image_height = height() / max_threads;
    int yi = current_thread * sub_image_height;

    for (int py = yi; py < sub_image_height + yi; py++) {
        double ry = vPixel2Rect(py);
        for (int px = 0; px < width(); px++) {
            double rx = hPixel2Rect(px);
            QRgb rgb_color = calc_in_out(rx, ry);
            setPixel(px, py, rgb_color);
        }
    }
}
```

```
class H_Pixel2Rect{
private:
    double a_;
    double b_;
public:
    H_Pixel2Rect(double cx, double d, double px_min, double px_max){
        //linear interpolation with x
        //px: 0        ---          599
        //rx: xc - 1.5d      ---       xc + 1.5d
        //a , b
        a_ = (3.0 * d / (px_max - px_min));
        b_ = cx - 1.5 * d;
    }
    double operator()(double px) {
        return a_ * px + b_;
    }
};
```