Noms : LEMPEREUR

Prénoms : Noémie

Classe : CIR2

Module : C++

Responsable du module : Nils BEAUSSE

Titre du document : Rapport du TP4 de C++

Date et heure d'envoi : 24 Novembre 2020 – 8h00

Nombre de mots :

- ☑ En adressant ce document à l'enseignant, je certifie que ce travail est le mien et que j'ai pris connaissance des règles relatives au référencement et au plagiat.

Sommaire

Introduction

Nous allons manipuler et créer des images au format PGM et au format PPM.

Tout d'abord, un fichier PGM permet de stocker des données d'images. Sur la première ligne du fichier, on a P2 qui est écrit. P2 permet de préciser que nous sommes en présence d'une image PGM. Ensuite, on a la largeur et la hauteur en pixel de l'image. Puis à la ligne, on a la valeur maximale (nous prendrons 255 pour le tp). On a toutes les valeurs d'intensité de gris sous forme de tableau.

L'objectif est dans un premier temps de réussir à écrire, lire et modifier une image PGM, puis pour une image PPM.

Une image PPM a le même format qu'une image PGM mais elle sera en couleur. On aura donc à la place des valeurs d'intensité de gris, un tableau pour stocker les valeurs RVB.

Classe PGM

La classe PGM est composé de plusieurs attributs et méthodes :

```
class PGM {
private:
   int largeur;
   int hauteur;
   int valeur_max;
   int valeur_min;
   int **data;
   static int nbImages;
public:
   PGM();
   PGM(int haut, int larg, int max,int min);
   ~PGM();
   PGM(PGM const& copie);
   void setLargeur(int larg);
   void setHauteur(int haut);
   void setValeurMax(int max);
   void setData(int **tab);
   int getLargeur();
   int getHauteur();
   int getValeurMax();
   int** getData();
   void initImage(int haut, int larg);
   void supprImage();
   void creeImage(int minpix=0, int maxpix=255);
   int getValeurMin();
   void setValeurMin(int min);
   void afficherImage();
   void ecrireFichier(char* nom_fichier);
   void dessinRect(int x1, int y1, int x2, int y2, int val);
   void dessinLigne(int x1, int x2, int line, int val);
   void dessinCroix(int x, int y, int val);
   void lectureFichier(char *nom_fichier);
   static int getNbImages();
   void finProgramme();
   void seuil(int seuil);
   void flou(int larg);
   void filtrerImage(int l);
};
```

Les attributs sont des variables privées auxquelles on peut accéder grâce aux accesseurs.
Les attributs largeur et hauteur désignent le nombre de pixel sur la largeur ou sur la hauteur de l'image. Les attributs valeur_max et valeur_min désignent la valeur maximale et minimale que peut prendre un pixel. Nous prendrons une valeur minimale de 0 et une valeur maximale de 255.

Nous avons aussi **data qui est un tableau à deux dimensions d'entiers. Il va permettre de stocker chaque pixel de l'image. On peut accéder à la valeur de la nuance de gris d'un pixel en faisant data[hauteur_pixel][largeur_pixel].

Par exemple, si on initialise l'image avec une hauteur de 5 et une largeur de 10, on aura data comme suit :

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										

Ensuite les différentes cases de data prendront la valeur de l'intensité de gris (de 0 à 255) du pixel correspondant :

	0	1	2	3	4	5	6	7	8	9
0	2	189	190	16	89	89	92	108	132	250
1	22	187	191	16	9	8	92	98	132	251
2	27	18	199	14	8	8	92	118	132	234
3	28	19	197	16	87	12	91	128	132	246
4	8	19	190	12	83	27	92	178	132	250

Dans ce cas, la case data [0][0]=2, le pixel en haut à gauche sera donc noir.

Le dernier attribut, l'attribut nbImages est un attribut static. Il est global à toutes les classes PGM et permet de compter le nombre d'images créées. On l'initialise au début du fichier PGM.cpp : int PGM::nbImages=0; Pour qu'à la fin du programme, cette valeur soit à 0, on peut appeler une fonction : la fonction finProgramme :

```
void PGM::finProgramme() {
        nbImages=0;
}
```

En cas d'erreur (si le destructeur n'est pas appelé pour détruire les images), les images seront forcément à 0 à la fin du programme.

Accesseurs et mutateurs de la Classe PGM

Du fait du principe de l'encapsulation, les attributs sont des éléments privés. On ne peut donc pas accéder à ces éléments à partir des autres classes ou du main.cpp. C'est pour cela qu'on a recours à des accesseurs et des mutateurs. Les accesseurs permettent de renvoyer la valeur de l'attribut et un mutateur permet de modifier celle-ci.

Dans la classe PGM, nous avons six attributs. Nous avons donc fait six accesseurs et cinq mutateurs (la variable globale nbImages n'a pas de mutateurs). Pour plus de lisibilité, les accesseurs sont appelés getAttributs et les mutateurs setAttributs.

Les accesseurs :

Pour faire un accesseur, il suffit de renvoyer la valeur que l'on souhaite :

```
int getAttribut(){
        return attribut ;
}
```

Si on applique ce code à la classe PGM et aux différents attributs, on obtient :

Pour la largeur :

```
int PGM::getLargeur() {
        return largeur;
}
```

Pour la hauteur :

```
int PGM::getHauteur() {
        return hauteur;
}
```

Pour la valeur maximale (valeur_max) :

```
int PGM::getValeurMax() {
        return valeur_max;
}
```

Pour la valeur minimale (valeur_min) :

```
int PGM::getValeurMin() {
        return valeur_min;
}
```

Pour le tableau data :

```
int** PGM::getData() {
   return data;
}
```

Pour le nombre d'images :

```
int PGM::getNbImages() {
   return nbImages;
}
```

Les mutateurs :

Pour faire un mutateur, il suffit de remplacer la valeur que l'on souhaite par celle placée en paramètres :

```
void setAttribut(int value){
        attribut=value ;
}
```

Si on applique ce code à la classe PGM et aux différents attributs, on obtient :

Pour la largeur :

```
void PGM::setLargeur(int larg) {
   largeur=larg;
}
```

Pour la hauteur :

```
void PGM::setHauteur(int haut) {
   hauteur=haut;
}
```

Pour la valeur maximale (valeur_max) :

```
void PGM::setValeurMax(int max) {
   valeur_max=max;
}
```

Pour la valeur minimale (valeur_min) :

```
void PGM::setValeurMin(int min) {
   valeur_min=min;
}
```

Pour le tableau data :

```
void PGM::setData(int **tab) {
   data=tab;
}
```

Pour le nombre d'image :

Comme le nombre d'image est une variable globale qui compte le nombre d'images, on peut tout simplement incrémenter cette variable lorsqu'on crée une image et la décrémenter lorsqu'on en supprime une. Je n'ai donc pas créé de mutateurs pour cette variable car il est inutile d'augmenter le nombre d'image si on n'en crée pas une, or pour en créer une on utilise forcément un constructeur de la classe.

Constructeurs et destructeur

Les constructeurs

Le constructeur est une fonction appelée automatiquement lors de la création d'un objet, ici une image PGM. Cette fonction permet l'initialisation des variables, elle est donc la première à être exécutée. Il a une déclaration particulière car il a le même nom que la classe et ne comporte pas de void et ne renvoie rien: PGM();.
Nous avons fait trois constructeurs : un sans paramètre, un avec paramètres et un constructeur de copie. Les constructeurs appellent la fonction initImage.

```
void PGM::initImage(int haut, int larg) {
        data=new int*[haut];            //On alloue de la mémoire pour le tableau des valeurs des
        for(int i=0;i<haut;i++) {       //pixel
                data[i]=new int[larg];
        }
        setHauteur(haut);               //Grâce aux paramètres, on rentre les arguments de l'image
        setLargeur(larg);               // dans la classe
        setValeurMax(255);
        setValeurMin(0);
}
```

Constructeur sans paramètre :

```
PGM::PGM() {
        initImage(100,100);   //On initialise l'image vide, on lui met une valeur par défaut de
                              //100 pixels de haut et 100 pixels de large
        nbImages++;           //On vient de créer une image donc on incrémente la variable
                              //globale
}
```

Constructeur avec paramètres :

```
PGM::PGM(int haut, int larg, int max, int min) {
        initImage(haut,larg); //L'image a une hauteur de haut et une largeur de larg
        nbImages++;           //On vient de créer une image donc on incrémente la variable
                              //globale

}
```

Constructeur de copie :

Pour pouvoir créer une image à partir d'une autre image, il faut créer un constructeur de copie :

```
PGM::PGM(const PGM &copie) :
        nbImages++;                          //On vient de créer une image donc on
                                             //incrémente la variable globale
        this->largeur=copie.largeur ;        //on copie la largeur
        this->hauteur=copie.hauteur ;        //on copie la hauteur
        this->valeur_max=copie.valeur_max ;  //on copie la valeur maximale
        this->valeur_min=copie.valeur_min ;  //on copie la valeur minimale
        this->data=copie.data ;              //on copie le tableau data
}
```

Le destructeur

Le destructeur est une fonction appelée automatiquement lors de la suppression d'une image PGM. Cette fonction permet de libérer la mémoire utilisée pour le stockage des attributs. Par exemple, avec les images PGM, il faut avoir un tableau d'entiers. Ce tableau doit être détruit à l'aide d'un « delete » lors de la suppression de l'image. Tout comme le constructeur, le destructeur a une déclaration particulière : on doit mettre un tilde devant le nom de la classe: ~PGM();.

Le destructeur va faire appel à la fonction supprImage :

```
void PGM::supprImage() {
        for(int i=0;i<hauteur;i++) {
                delete[](data[i]);           //On supprime la mémoire utilisée pour le tableau
        }                                    //des valeurs de pixels
        delete[](data);
         setHauteur(0);                      //on remet la hauteur et la largeur à 0
        setLargeur(0);
}
```

Le destructeur de la classe PGM est :

```
PGM::~PGM() {
        supprImage();                        //On appelle la fonction de suppression
        nbImages--;                          //On vient de supprimer une image donc on
                                             //décrémente la variable globale
}
```

Fonctions de création des images PGM

Création d'une image aléatoire

Pour créer une image avec des valeurs de pixels aléatoire, il faut parcourir toutes las cases du tableau des valeurs des pixels et les remplacées par un nombre aléatoire (ici entre 0 et 255). Pour avoir des nombres aléatoires, on utilise la fonction rand(). Avant cela, il faut mettre « srand(time(NULL)) » qui permet d'initialiser le générateur de nombres aléatoires. Si on ne le met pas, à chaque fois que nous lancerons le programme, nous aurons les mêmes nombres aléatoires dans le même ordre.

```
void PGM::creeImage(int minpix, int maxpix) {
   srand (time(NULL));                  //Permet d'initialiser la suite des nombres aléatoires
   for(int i=0;i<hauteur;i++){          //Permet de parcourir le tableau dans sa hauteur
      for(int j=0;j<largeur;j++) {      //Permet de parcourir le tableau dans sa largeur
         data[i][j]= rand() % 256;      //Le pixel prend pour valeur un nombre aléatoire
                                        //(compris entre 0 et 255)
      }
   }
}
```

En exécutant cette fonction, on obtient une image avec des pixels ayant une valeur aléatoire, par exemple, on peut obtenir :

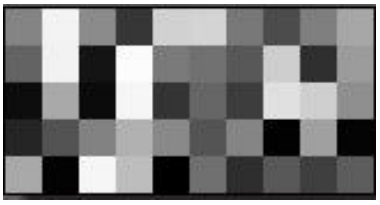
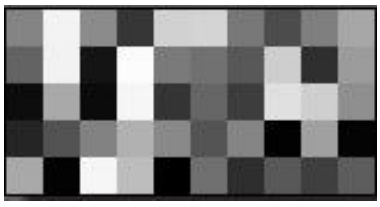


Image créée aléatoirement

Affichage dans la console

Pour afficher une image PGM, il faut le faire à partir de logiciels spéciaux (ex :Photo filtre, GIMP, photoshop…). Pour plus de facilité, on peut créer une fonction nous affichant un tableau de valeur dans lequel chaque valeur représente le nombre de l'intensité de gris pour un pixel.

En exécutant la fonction affichage, on obtient un tableau des valeurs de chaque pixel :



```
132 243 139 54  209 212 120 76  127 167
99  242 20  252 122 113 88  206 47  156
13  169 11  246 52  102 61  224 207 143
37  83  130 177 137 84  133 2   160 4
169 3   246 189 0   112 46  88  63  93
```

Image1 créée aléatoirement *Matrice de l'image1 grâce à la fonction affichage*

Pour créer cette fonction, il faut parcourir le tableau en hauteur et largeur, on utilise donc deux boucles « for ». Ensuite il faut afficher la valeur du pixel, on a donc cout << data[i][j] . A chaque fois que nous changeons de hauteur, il faut revenir à la ligne dans l'affichage. Pour cela, dans la boucle « for » de la hauteur on ajoute cout<<endl . Pour plus de lisibilité, il faut aligner les valeurs en ajoutant des espaces en fonction du nombre de chiffres à afficher.

```
void PGM::afficherImage() {
  for(int i=0;i<hauteur;i++){                    //On parcourt le tableau dans sa hauteur
    for(int j=0;j<largeur;j++) {                 //On parcourt le tableau dans sa largeur
      if (data[i][j]>99) {
        cout << data[i][j] << " ";               //On affiche la valeur du pixel + 1 espace
      }
      else{
        if(data[i][j]>9){
          cout << data[i][j] << "   ";           //On affiche la valeur du pixel + 2 espaces
        }
        else{
          cout << data[i][j] << "    ";          //On affiche la valeur du pixel + 3 espaces
        }
      }
    }
    cout<<endl;                                  //On retourne à la ligne
  }
}
```

Lecture et écriture d'images PGM

Lecture d'une image PGM:
Comme dit en introduction, le format PGM possède plusieurs lignes.
On va d'abord récupérer la première ligne qui contient P2. On récupère les informations pour les mettre en attributs (largeur, hauteur valeur maximale). On va aussi récupérer toutes les valeurs de nuances de gris pour les stocker dans data.

```
void PGM::lectureFichier(char* nom_fichier){
   ifstream monfichier;
   string ligne;
   stringstream s;
   monfichier.open(nom_fichier);
   if (monfichier.is_open()){
      getline(monfichier,ligne);
      s << monfichier.rdbuf();
      s >> largeur >> hauteur;          //On récupère la largeur et la hauteur
      s >> valeur_max;                  //On récupère la valeur maximale
      initImage(hauteur,largeur);       //On initialise l'image avec les paramètres
                                        //de largeur et de hauteur
      setValeurMax(valeur_max) ;        //On modifie la valeur maximale de l'image
                                        //initialisée par celle du fichier
      for(int i=0;i<hauteur;i++){
         for(int j=0;j<largeur;j++){
            s>>data[i][j];              //On modifie les valeurs de nuance de gris
                                        //de l'image par celles du fichier
         }
      }
   }
   monfichier.close();
}
```

Ecriture d'une image PGM:

Sur le même principe que la fonction de lecture, on peut développer la fonction d'écriture :

```
void PGM::ecrireFichier(char* nom_fichier) {
   ofstream myfile;
   myfile.open(nom_fichier);
   myfile<<"P2\n";                          //On écrit P2 au début du fichier pour indiquer que
                                            //l'image est une image de format PGM
   myfile<<largeur<<' '<<hauteur<<'\n';     //On écrit dans le fichier les différents attributs de
   myfile<<valeur_max<<'\n';                // la classe PGM : la hauteur, la largeur, la valeur
   for(int i=0;i<hauteur;i++){              // max et le tableau de valeurs des pixels
      for(int j=0;j<largeur;j++) {
         if (data[i][j]>99) {
            myfile<< data[i][j] << " ";     //1 espace
         }
         else{
            if(data[i][j]>9){
               myfile<< data[i][j] << "  ";  //2 espaces
            }
            else{
               myfile<< data[i][j] << "   ";  //3 espaces
            }
         }
      }
      myfile<<'\n';
   }
}
```

Fonctions de dessins

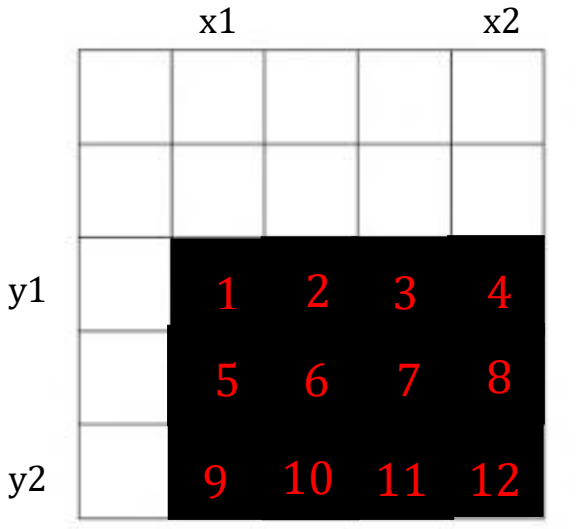
Nous avons faire plusieurs fonctions permettant de dessiner sur une image.

Dessin d'un rectangle :

La première fonction de dessin permet de dessiner un rectangle dans laquelle toutes les valeurs des pixels prendront une valeur donnée en paramètre.

Pour cela, il faut parcourir le tableau des valeurs en hauteur entre y1 et y2 et en largeur entre x1 et x2. Il faut modifier la valeur des pixels par la valeur passée en paramètre.

Voici un extrait de data, avec en noir les pixels à changer (le chiffre indique l'ordre de changement) :



On a donc

```
void PGM::dessinRect(int x1, int y1, int x2, int y2, int val){
  for(int i=y1;i<=y2;i++){          //On parcourt le tableau dans sa hauteur entre y1 et y2
    for(int j=x1;j<=x2;j++){        //On parcourt le tableau dans sa largeur entre x1 et x2
      data[i][j]=val;               //On modifie la valeur du pixel pour qu'elle soit égale à la
    }                               // valeur passée en paramètre (val)
  }
}
```

Par exemple, si on appelle la fonction dessinRect(200,210,300,400,6) sur l'image lena.pgm, on obtient :

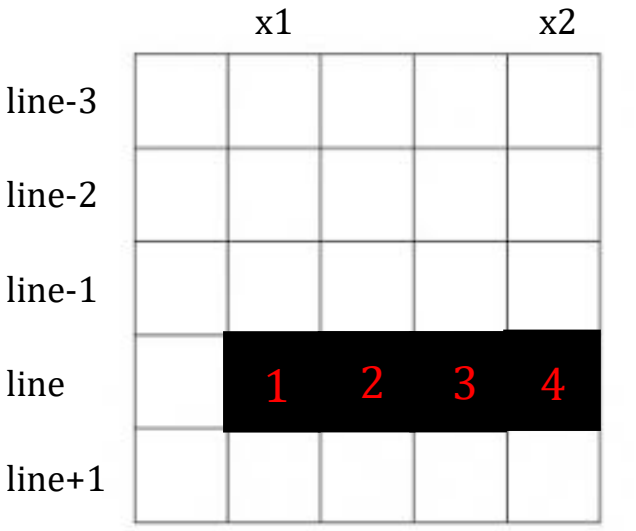
Image modifiée avec la fonction dessinRect

Dessin d'une ligne :

La seconde fonction de dessin permet de dessiner une ligne sur laquelle toutes les valeurs des pixels prendront une valeur donnée en paramètre.

Pour cela, il faut parcourir le tableau des valeurs en largeur entre x1 et x2. Il faut modifier la valeur des pixels par la valeur passée en paramètre. L'entier line passé en paramètre est la hauteur à laquelle la ligne horizontale a lieu.

Voici un extrait de data, avec en noir les pixels à changer (le chiffre indique l'ordre de changement) :



On a donc :

```
void PGM::dessinLigne(int x1, int x2, int line, int val){
        for(int i=x1;i<= x2;i++){              //On parcourt le tableau en largeur entre x1 et x2

                data[line][i]=val;             //On change la valeur du pixel par celle passée en
                                               //paramètre pour tout pixel se trouvant sur le
                                               //segment [x1;x2] à une hauteur line

        }
}
```

On peut aussi développer cette fonction en appelant simplement la fonction dessinRect :

```
void PGM::dessinLigne(int x1, int x2, int line, int val){
        dessinRect(x1,line,x2,line,val) ;
}
```

Par exemple, si on appelle la fonction dessinLigne(250,350,300,6) sur l'image lena.pgm, on obtient :

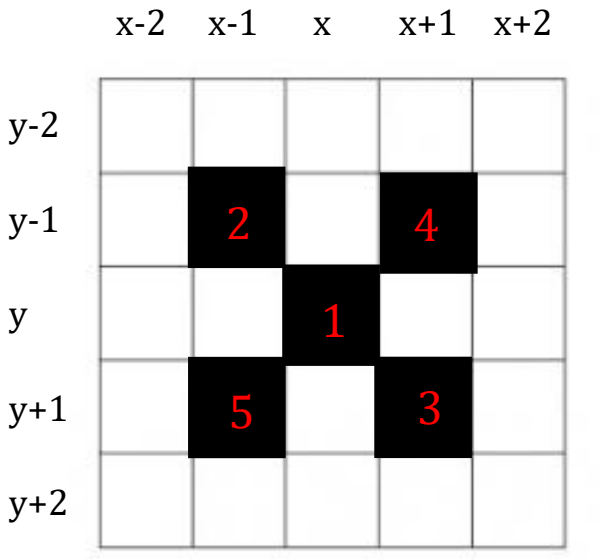
Image modifiée avec la fonction dessinLigne

Dessin d'une croix :

La troisième fonction de dessin permet de dessiner une croix de cinq pixels dans laquelle toutes les valeurs des pixels prendront une valeur donnée en paramètre.

Pour cela, il faut modifier les pixels en diagonale du pixel rentré en paramètre et le pixel passé en paramètre. Il faut modifier la valeur des pixels par la valeur passée en paramètre.

Voici un extrait de data, avec en noir les pixels à changer (le chiffre indique l'ordre de changement) :



On a donc :

```
void PGM::dessinCroix(int x, int y, int val){
   data[y][x]=val;          //On modifie la valeur du pixel à une position y en hauteur et une
                            //position x en largeur par val (correspondant au carré 1)
   data[y-1][x-1]=val;      //On modifie la valeur du pixel à une position y-1 en hauteur et une
                            //position x-1 en largeur par val (correspondant au carré 2)
   data[y+1][x+1]=val;        //On modifie la valeur du pixel à une position y+1 en hauteur et
                              //une position x+1 en largeur par val (correspondant au carré 3)
   data[y-1][x+1]=val;      //On modifie la valeur du pixel à une position y-1 en hauteur et
                            //une position x+1 en largeur par val (correspondant au carré 4)
   data[y+1][x-1]=val;      //On modifie la valeur du pixel à une position y+1 en hauteur et
                            //une position x-1 en largeur par val (correspondant au carré 5)
}
```

Par exemple, si on appelle la fonction dessinCroix(300,210,6) sur l'image lena.pgm, on obtient :

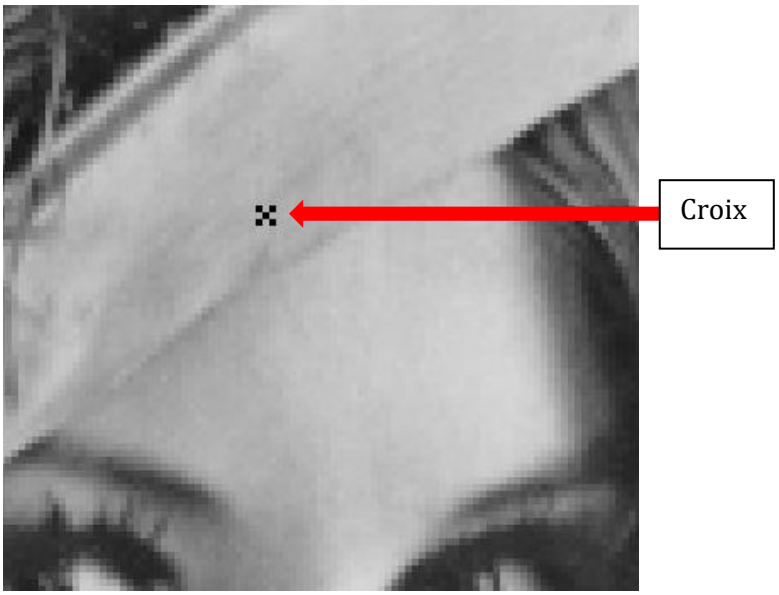


Image modifiée avec la fonction dessinCroix

Fonctions d'opération sur les images PGM

Il existe différentes fonctions d'opérations sur des images.

La fonction de seuil

La fonction de seuil prend une valeur en paramètre. Elle compare la valeur de chaque pixel par celle passée en paramètre. Si la valeur du pixel est plus faible que celle passée en paramètre, alors la valeur du pixel devient 0, sinon elle prend la valeur maximale (ici 255).
Par exemple, voici les différentes images obtenues avec différents seuils :

Image originale

Image avec un seuil de 80

Image avec un seuil de 120

Image avec un seuil de 150

On peut voir que plus le seuil est élevé, plus l'image sera noire et inversement. Le seuil enlève toute nuance de gris et fonctionne de façon binaire : c'est soit noir, soit blanc.

Le code de la fonction est le suivant :

```
void PGM::seuil(int seuil) {
   for(int i=0;i<hauteur;i++){          //On parcourt le tableau de valeurs des pixels dans sa hauteur
      for(int j=0;j<largeur;j++){       //On parcourt le tableau de valeurs des pixels dans sa largeur
         if(data[i][j]>seuil){
            data[i][j]=255;             //Si la valeur du pixel est supérieure au seuil alors elle
         }                              // prends la valeur 255
         else{
            data[i][j]=0;               //Sinon elle prend la valeur 0
         }
      }
   }
}
```

Les deux boucles « for » permettent de parcourir toutes les valeurs des pixels. Le « if » compare la valeur du pixel avec celle du seuil. Si elle est supérieure au seuil alors elle prend la valeur 255, sinon elle prend la valeur 0

La fonction de floutage

Le principe de la fonction de floutage, est de prendre un rayon de pixel autour de chaque pixel. Additionner la valeur des pixels autour puis diviser par le nombre pixels. On obtient la moyenne des pixels compris dans un carré de coté deux fois le rayon (passé en paramètre). Ensuite, on donne au pixel du centre le résultat de cette moyenne.

```
void PGM::flou(int larg) {
  for(int i=0;i<hauteur;i++){          //On parcourt le tableau dans sa hauteur
    for(int j=0;j<largeur;j++){        //On parcourt le tableau dans sa largeur
      int sum=0;                       //On initialise une variable qui contiendra la valeur de tous
                                       //les pixels autour
      int num=0;                       //On initialise une variable qui contiendra le nombre de
                                       //pixels autour
      for(int k=i-larg;k<i+larg;k++){      //On parcourt les pixels en hauteur appartenant au
                                           //carré de côté 2*larg
        if(k>=0&&k<hauteur){               //On vérifie que le pixel appartient bien au tableau
          for (int p =j-larg; p<j + larg; p++) {  //On parcourt les pixels en largeur appartenant au
                                                  //carré de côté 2*larg

            if(p>=0&&p<largeur){           //On vérifie que le pixel appartient bien au tableau
              sum=sum+data[k][p];          //Comme le pixel appartient au tableau, à l'aide de
                                           //la variable sum, on l'additionne aux autres pixels
              num=num+1;                   //On augmente le nombre de pixel additionné
            }
          }
        }
      }
      int val_moy=sum/num;             //On calcule la valeur moyenne des pixels
                                       //appartenant au carré de côté 2*larg
      data[i][j]=val_moy;              //Le pixel au centre du carré prend cette valeur
                                       //moyenne
    }
  }
}
```

Par exemple, voici les différentes images obtenues avec différentes valeurs de rayon de pixel :

Image originale

Image avec un flou de 5 pixels

Image avec un flou de 10 pixels

Image avec un flou de 20 pixels

On peut voir que plus le rayon de pixel flouté est élevé, moins l'image sera visible, dû au manque de netteté.

La fonction de filtrage

Le principe de la fonction de filtrage, a des similitudes avec la fonction de floutage. On va, comme dans la fonction de floutage, récupérer les valeurs des pixels. On va ensuite les stocker dans un tableau. Puis on va trier le tableau à l'aide de la fonction Tri_Selection. On obtient la moyenne des pixels compris dans un carré de côté deux fois le rayon (passé en paramètre). Ensuite, on donne au pixel du centre la valeur médiane du tableau, c'est-à-dire la valeur se trouvant au centre du tableau. On fait ça pour chaque pixel.
Par exemple, voici les différentes images obtenues avec différentes valeurs de rayon de pixel :

Image originale

Image avec un filtrage d'un carré de 1 pixel autour

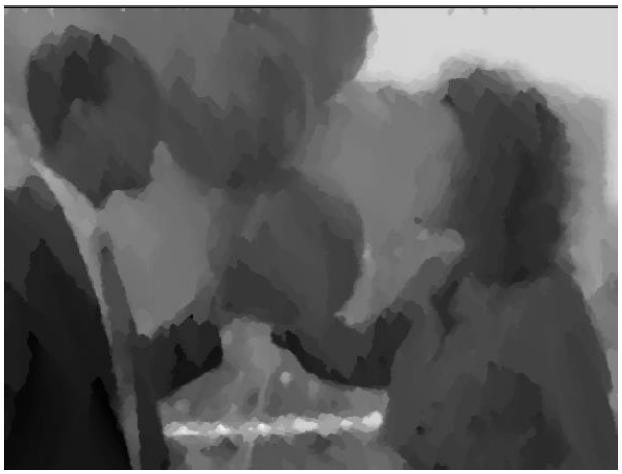


Image avec un filtrage d'un carré de 2 pixels autour

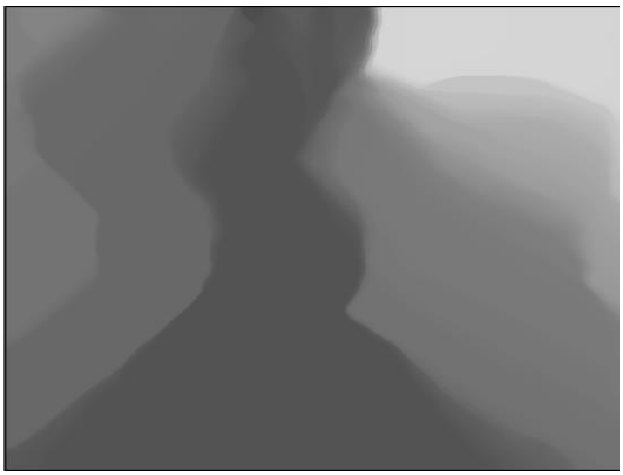


Image avec un filtrage d'un carré de 10 pixels autour

On peut voir que pour obtenir une image nette et précise, il faut un rayon d'un pixel autour. A partir de 2 pixels autour, l'image n'est pas précise. On peut voir que si on applique cette fonction avec un très grand nombre comme rayon, l'image tend vers la couleur médiane du tableau. On a peu de couleurs extrêmes.

Le «tri par sélection» est un algorithme de tri qui va d'abord trouver la valeur minimale du tableau (entrée en paramètre) en parcourant l'ensemble du tableau. Ensuite, il échange cette valeur avec le premier. Il effectue les mêmes opérations en ne traitant plus la première valeur puisqu'elle est déjà triée. Il continue jusqu'à ce qu'il ait fait toute la table.

```
void Tri_Selection(int table[10000], int taille){
   int c;
   for(int i=0;i<taille-1;i++){
      for(int j=i+1;j<taille;j++){
         if(table[i]>table[j]){
            c=table[i];
            table[i]=table[j];
            table[j]=c;
         }
      }
   }
}
```

```
void PGM::filtrerImage(int l) {
  for(int i=0;i<hauteur;i++){                  //On parcourt tout le tableau dans sa hauteur
    for(int j=0;j<largeur;j++){                //On parcourt tout le tableau dans sa largeur
      int sum[100000000]={0};                  //tableau permettant de stocker les valeurs des
                                               //pixels présent dans le rayon
      int num=0;                               //Correspond au nombre de valeurs dans le tableau
      for(int k=(i-l);k<=(i+l);k++){           //On parcourt les pixels en hauteur appartenant au
                                               //carré de côté 2*l
        if(k>=0&&k<hauteur){                   //On vérifie que le pixel appartient bien au tableau
          for (int p =(j-l); p<=(j + l); p++) {//On parcourt les pixels en largeur appartenant au
                                               //carré de côté 2*l
            if(p>=0&&p<largeur){               //On vérifie que le pixel appartient bien au tableau
              sum[num]=data[k][p];             //On ajoute la valeur du pixel au tableau
              num=num+1;                       //On a rajouté un nombre dans le tableau donc on
                                               //incrémente la variable num
            }
          }
        }
      }
      if(num!=0) {                             //Si le rayon est de 0, il n'y a pas besoin de trier le
                                               //tableau
        Tri_Selection(sum, num);               //On trie le tableau
        int val_med = sum[num / 2];            //On trouve la valeur médiane du tableau
        data[i][j] = val_med;                  //On remplace la valeur d'intensité de gris par la
                                               //valeur médiane
      }
    }
  }
}
```

Classe PPM: extension aux images PGM

La classe PPM est très semblable à la classe PGM. En effet, Elle comporte autant d'attributs. Le seul attribut modifié est le int** data qui devient un int*** data. Il faut coder les couleurs en RGB donc il faut trois valeurs par pixels et non une comme dans la classe PGM. Toutes les méthodes vont être similaires mais il faut prendre en compte le changement de l'attribut data. Les accesseurs et les mutateurs autres que celui de data ne vont pas changer.

Pour l'accesseur et le mutateur, seule la déclaration changera : l'accesseur retournera un int*** et le mutateur aura comme paramètre un int***.

La fonction d'initialisation va utiliser 3 « new » et la fonction de suppression 3 « delete » :

```
void PPM::initImage(int haut, int larg) {
   data=new int**[haut];
   for(int i=0;i<haut;i++) {
      data[i]=new int*[larg];
      for(int j=0;j<larg;j++) {
         data[i][j]=new int[3];
      }
   }
   setHauteur(haut);
   setLargeur(larg);
   setValeurMax(255);
   setValeurMin(0);
}

void PPM::supprImage() {
   for(int i=0;i<hauteur;i++) {
      for(int j=0;j<3;j++) {
         delete[](data[i][j]);
      }
      delete[](data[i]);
   }
   delete[](data);
   setHauteur(0);
   setLargeur(0);
}
```

Le constructeur de copie est légèrement modifié car on rajoute une boucle « for ».

```
PPM::PPM(const PPM &copie) {
   initImage(hauteur,largeur);
   for(int i=0;i<hauteur;i++){
      for(int j=0;j<largeur;j++){
         for(int k=0;k<3;k++) {
            data[i][j][k] = copie.data[i][j][k];
         }
      }
   }
   nbImages++;
}
```

La fonction d'affichage de la matrice dans la console contiendra elle aussi une boucle « for » supplémentaire pour permettre d'afficher les 3 valeurs du pixel.

```
void PPM::afficherImage() {
   for(int i=0;i<hauteur;i++){
      for(int j=0;j<largeur;j++) {
         for(int k=0;k<3;k++) {
            if (data[i][j][k] > 99) {
               cout << data[i][j][k] << " ";
            } else {
               if (data[i][j][k] > 9) {
                  cout << data[i][j][k] << "  ";
               } else {
                  cout << data[i][j][k] << "   ";
               }
            }
         }
         cout <<"    ";
      }
      cout<<endl;
   }
}
```

Il va aussi falloir trois nombres aléatoires pour les composantes rouges, vertes et bleues pour créer une image en couleur aléatoirement :

```
void PPM::creeImage(int minpix, int maxpix) {
   srand (time(NULL));
   int ran;
   for(int i=0;i<hauteur;i++){
      for(int j=0;j<largeur;j++) {
         ran = rand() % 256;
         data[i][j][0]=ran;
         ran = rand() % 256;
         data[i][j][1]=ran;
         ran = rand() % 256;
         data[i][j][2]=ran;
      }
   }
}
```

Par exemple, la fonction créer image peut donner :

Les fonctions de dessins, vont-elles aussi être modifié. On va devoir mettre trois valeurs pour coder la couleur en paramètre. Le principe ne change pas entre le PGM et le PPM.

```
void PPM::dessinRect(int x1, int y1, int x2, int y2, int valR,int valV,int valB){
  for(int i=y1;i<=y2;i++){
    for(int j=x1;j<=x2;j++){
      data[i][j][0]=valR;
      data[i][j][1]=valV;
      data[i][j][2]=valB;
    }
  }
}

void PPM::dessinCroix(int x, int y, int valR,int valV,int valB){
  data[y][x][0]=valR;
  data[y][x][1]=valV;
  data[y][x][2]=valB;
  data[y-1][x-1][0]=valR;
  data[y-1][x-1][1]=valV;
  data[y-1][x-1][2]=valB;
  data[y+1][x+1][0]=valR;
  data[y+1][x+1][1]=valV;
  data[y+1][x+1][2]=valB;
  data[y-1][x+1][0]=valR;
  data[y-1][x+1][1]=valV;
  data[y-1][x+1][2]=valB;
  data[y+1][x-1][0]=valR;
  data[y+1][x-1][1]=valV;
  data[y+1][x-1][2]=valB;
}
```

```
void PPM::dessinLigne(int x1, int x2, int line, int valR,int valV,int valB){
   for(int i=x1;i<= x2;i++){
      data[line][i][0]=valR;
      data[line][i][1]=valV;
      data[line][i][2]=valB;
   }
}
```

En appliquant les fonctions de dessins, on obtient :

Image originale

Image appelé avec la fonction dessinLigne(5,12,5,6,0,255)

Image appelé avec la fonction dessinRect(3,4,10,6,56,255,3)

Image appelé avec la fonction dessinCroix(3,4,10,6,0,255)

Pour les fonctions de lecture et d'écriture, il va aussi falloir faire trois boucles « for » pour pouvoir récupérer les valeurs des différentes composantes RGB du pixel. Tout comme autres fonctions, le principe ne change pas.

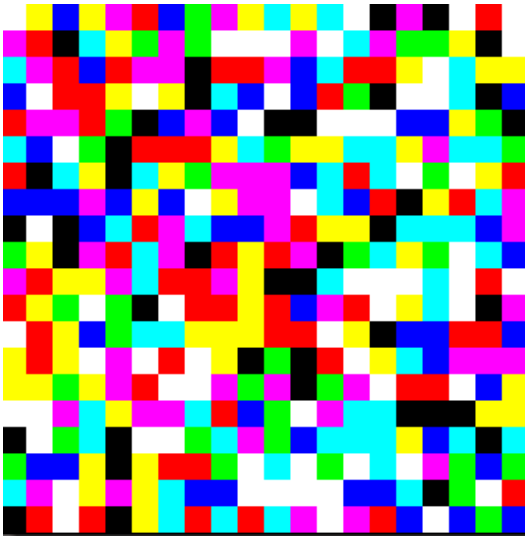
```
void PPM::lectureFichier(char* nom_fichier){
   ifstream monfichier;
   string ligne;
   stringstream s;
   monfichier.open(nom_fichier);
   if (monfichier.is_open()){
      getline(monfichier,ligne);
      s << monfichier.rdbuf();
      s >> largeur >> hauteur;
      s >> valeur_max;
      initImage(hauteur,largeur);
      cout<<hauteur<<endl;
      cout<<largeur<<endl;
      for(int i=0;i<hauteur;i++){
         for(int j=0;j<largeur;j++){
            for(int k=0;k<3;k++) {
               cout<<data[i][j][k]<<endl;
               s >> data[i][j][k];
            }
         }
      }
   }
   monfichier.close();
}
```

```
void PPM::ecrireFichier(char* nom_fichier) {
   ofstream myfile;
   myfile.open(nom_fichier);
   myfile<<"P3\n";                                  //On a une image PPM, il faut donc mettre P3
   myfile<<largeur<<' '<<hauteur<<'\n';
   myfile<<valeur_max<<'\n';
   for(int i=0;i<hauteur;i++){
      for(int j=0;j<largeur;j++) {
         for(int k=0;k<3;k++) {
            if (data[i][j][k] > 99) {
               myfile << data[i][j][k] << " ";
            } else {
               if (data[i][j][k] > 9) {
                  myfile << data[i][j][k] << "  ";
               } else {
                  myfile << data[i][j][k] << "   ";
               }
            }
         }
      }
      myfile<<'\n';
   }
}
```

Enfin, la fonction de seuil est aussi modifiée : il y a une troisième boucle « for » et on compare chaque valeur du pixel avec celle du seuil, pour ensuite lui donner la valeur 0 si elle est inférieure, ou la valeur 255 si elle est supérieure.

```
void PPM::seuil(int seuil) {
  for(int i=0;i<hauteur;i++){
    for(int j=0;j<largeur;j++){
      for(int k=0;k<3;k++) {
        if (data[i][j][k] > seuil) {
          data[i][j][k] = 255;
        } else {
          data[i][j][k] = 0;
        }
      }
    }
  }
}
```

Sur une image aléatoire, on obtient une image comme suit :



On peut voir que nous obtenons beaucoup plus de pixels différents par rapport au format PGM.

Pour le floutage et le filtrage, on exécute trois fois la fonction de floutage du format PGM, la première fois avec la composante rouge du pixel, puis avec la composante verte et enfin avec la composante bleue.

Conclusion

Pour conclure, durant ce TP, nous avons pu mettre en pratique tous les concepts vus en cours : les constructeurs, les destructeurs, les accesseurs et les mutateurs. Nous avons appris à créer une image PGM et une image PPM.
Nous avons dans un premier temps fait une classe PGM. Il y avait donc qu'une seule valeur par pixel. Nous avions des images en nuance de gris. Puis dans un second temps, nous avons adapté les fonctions pour qu'elles puissent être en couleur. Pour cela, nous avons coder une couleur à l'aide du code RGB.