Projet informatique

L'objectif de ce projet est de développer un logiciel de compression d'images en niveaux de gris. La création de la version compressée est basée sur la construction d'un arbre dont les feuilles sont des rectangles qui pavent un domaine rectangle. Chaque nœud de l'arbre contient soit une feuille, soit quatre branches menant aux nouveaux nœuds. On appelle cette structure un *quadtree*. La figure 1 représente l'image originale et la figure 2 représente l'image obtenue après compression. C'est l'objectif principal de ce projet.

Méthode

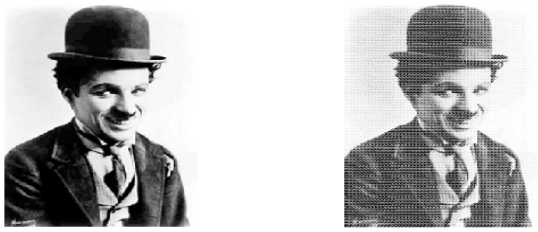

FIGURE 1 – À gauche l'image originale, à droite le maillage colorié.

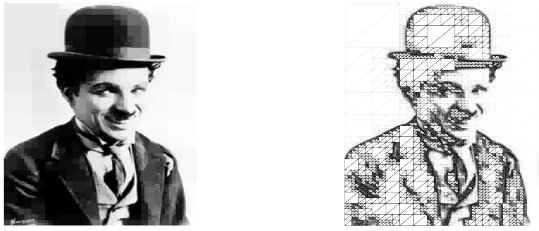

FIGURE 2 – À gauche l'image compressée, à droite le maillage colorié.

La méthode de compression choisie fonctionne de la manière suivante.

On plonge l'image dans un *quadtree* qui contient au plus $2^n \times 2^n$ feuilles carrées. n étant calculé de sorte que 2^n est plus grand que la plus grande des dimensions de l'image (en pixels).

Une fois tous les pixels insérés dans l'arbre, on va effectuer la compression. On parcours alors toutes les branches dont les nœuds sont tous des feuilles. Pour chacune de ces branches, on calcule la moyenne des couleurs des feuilles. Si le plus grand des écarts à cette moyenne est plus petit qu'un paramètre δ fixé, alors on remplace cette branche par une feuille de couleur moyenne. On itère le procédé jusqu'à ce qu'il n'y ait plus de modification à apporter à l'arbre.

Besoins

Besoin - 1 *Classes utilitaires à écrire*

Afin de mener à bien ce projet, un certain nombre de classes sont à écrire.

On prendra bien soin d'utiliser la compilation séparée lorsque c'est possible et de ne déclarer qu'une classe par fichier. Un **`Makefile`** aussi complet que possible devra être utilisé.

Besoin-1-1 **:** **`template <typename T> class Tableau`**

On écrira une classe conteneur à accès direct et extensible (une fonction de retaillage doit être programmée).

Besoin-1-2 **:** **`template <typename T, int N> class TinyVector`**

On aura besoin de cette classe utilitaire vue en cours pour la suite du projet. Cette classe doit surcharger les opérateurs nécessaires pour fournir une algèbre raisonnable. Des opérateurs d'accès surchargeant l'opérateur **`[]`** doivent être programmés pour accéder aux données de manière sécurisée.

On surchargera l'opérateur chevron (**`<<`**) pour permettre l'écriture de **`TinyVector`** sur la console.

Besoin-1-3 **:** **`N2`**

En se servant de la classe précédente, on définira, dans le fichier `N2.hpp`, le type **`N2`** qui modélisera $\mathbb{N}^2$.

Besoin-1-4 **:** **`R2`**

On procédera de la même manière pour définir le type **`R2`** qui modélisera $\mathbb{R}^2$. Ceci dans le fichier `R2.hpp`.

Besoin-1-5 **:** **`class Pixel : public N2`**

Cette classe servira à décrire un pixel de l'image originale. Ce seront des pixels qui seront stockés dans le *quadtree*. Cette classe servira aussi à décrire le maillage servant à l'affichage.

La classe mère **`N2`** permettra de stocker les coordonnées du pixel et on ajoutera un membre entier qui contiendra la couleur.

On définira les fonctions membres **`x()`** et **`y()`** qui permettront d'accéder aux coordonnées ainsi qu'une fonction **`couleur()`** qui donnera accès à la couleur du **`Pixel`**. On surchargera l'opérateur de copie et on définira les constructeurs.

Enfin, on permettra l'écriture de pixels sur la sortie standard.

Besoin-1-6 : `class Quadrangle`

Nous définissons une classe **`Quadrangle`** un peu particulière. Pour des raisons pratique d'affichage, les sommets seront définis comme des pixels (des points de $\mathbb{N}^2$ avec une couleur).
- Tout d'abord on surchargera l'opérateur **`()`** pour l'accès aux quatre pixels définissant les quadrangles.
- On définira l'opérateur de copie.

Besoin-1-7 : `class Maillage`

Cette classe ne sert qu'à l'affichage. Cette classe contiendra un tableau de **`Quadrangle`**s et un tableau de couleurs.

On définira les fonctions d'accès aux deux tableaux (une version constante et un permettant de modifier les listes).

On définira aussi une fonction patron qui mettra en œuvre le *design pattern* visiteur. Cette fonction permet à la classe **`DessinateurDeMaillage`** (fournie) de faire son travail. Elle a la signature suivante :

```
template <typename ManipulateurDeMaillage>
void accepteManipulateur(const ManipulateurDeMaillage& m);
```

Besoin - 2 Quadtree *et manipulation*

On définit maintenant les classes permettant la programmation du *quadtree*. On a choisi ici de définir une classe générique qui pourrait donc servir dans un contexte différent.

Besoin-2-1 : `template <typename Contenu> class Noeud`

La classe **`Noeud`** est la classe de base pour les feuilles et les branches. C'est dans cette classe qu'on définit les types énumérés (**`const Type`**) **`feuille`** et **`branche`** qui permettront de distinguer les feuilles des branches. Elle ne contient que le membre **`m_type`** qui décrit le type et une fonction d'accès en lecture seule à ce type.

Le seul constructeur autorisé prend en argument le type de la classe dérivée et est protégé afin d'interdire la création d'instance de **Noeud** qui ne soient ni des **`Feuilles`** ni des **`Branches`**.

Besoin-2-2 : `template<typename Contenu> class Feuille:public Noeud<Contenu>`

Cette classe contient le **`Contenu`** et sa position (**`R2`**). On fournit deux fonctions d'accès à ces membres. Le seul constructeur à définir prend en argument la position et le contenu.

Besoin-2-3 : `template<typename Contenu> class Branche:public Noeud<Contenu>`

Cette classe contient 4 données :
- un tableau de taille fixe 4 qui contient des pointeurs vers les nœuds fils,
- le centre de la cellule (**`R2`**),
- la taille de la cellule (**`R2`** car on pourrait imaginer le cas non carré),
- Un pointeur vers la **`Branche`** mère.

On prévoit des fonctions d'accès aux trois derniers membres. On surchargera l'opérateur **`[]`** pour accéder aux nœuds.

On définira la fonction

`void ajoute(const R2& position, const Contenu& contenu);`

Son rôle est d'ajouter une feuille à l'arbre (voir explications en cours).

Le constructeur de **`Branche`** prend en paramètres le centre, la taille et le pointeur vers la mère. Ils ne peuvent plus être modifiés par la suite. Les pointeurs vers les nœuds fils doivent être mis à 0 à l'initialisation.

Besoin-2-4 : `template <typename Contenu> class QuadTree`

L'unique donnée membre de cette classe est un pointeur vers la branche initiale (tronc).

Cette classe contient une sous-classe **`feuille_iterator`** décrite plus bas.

Elle contient les fonctions suivantes :
- **`feuille_iterator beginFeuille() const;`** Renvoie un itérateur vers la première feuille de l'arbre.
- **`feuille_iterator endFeuille() const;`** Renvoie un itérateur indiquant qu'on n'appartient pas à l'arbre.
- **`int nombreDeFeuilles() const`**
- **`void ajoute(const R2& position, const Contenu& contenu);`** Ajoute une feuille à l'arbre.

Besoin-2-5 : `class feuille_iterator`

Cette classe est une sous-classe de la classe **`QuadTree`**.

Pour cette classe un peu complexe, on utilisera deux classes patrons de la bibliothèque standard C++. La classe **`pair`** et la classe **`stack`**. L'idée est de créer une pile permettant de décrire l'adresse de l'itérateur dans l'arbre. On définira donc la position locale par une paire faisant intervenir un pointeur sur une branche et le numéro du nœud fils utilisé. On définira ensuite l'adresse comme l'empilement de ces positions locales partant du tronc de l'arbre.

Les fonctions suivantes seront à définir :
- une fonction d'accès constante à la feuille pointée par l'itérateur.
- une fonction d'accès constante à la branche dont est issue la feuille.
- une fonction retournant le **centre de la case** occupée par la feuille,
- une fonction donnant la taille (**`R2`**) de la case occupée par la feuille,
- une surcharge de l'opérateur **`++`** afin de passer à la feuille suivante de l'arbre,
- des surcharges des opérateurs **`!=`** et **`==`** pour comparer deux itérateurs.

Besoin-2-6 : `class SimplificateurDeQuadtree`

Cette classe fonction ne contient qu'une fonction dont le rôle est de simplifier le *quadtree* comme on l'a décrit dans la section Méthode.

Besoin-2-7 : `class ConvertisseurQuadtreeMaillage`

Cette autre classe fonction a pour rôle de convertir l'ensemble des feuilles d'un *quadtree* en un maillage de quadrangles qu'on pourra alors afficher.

Utilitaires

Utilitaire-1 : Conversion d'une image au format `xpm`

Pour convertir une image au format **`xpm`** en niveau de gris, on pourra utiliser l'outil **`convert`** sous Unix.

Pour créer l'image de l'exemple, on a utilisé la commande suivante :
`convert charlie-chaplin.jpg -colorspace Gray charlie-chaplin.xpm`

Utilitaire-2 : Lecteur de fichier au format `xpm`

Les fichiers sources **`LecteurXPM.hpp`** et **`LecteurXPM.cpp`** sont accessibles au lieu habituel. En guise d'illustration, voici le contenu du fichier **`LecteurXPM.hpp`**.

fichier:LecteurXPM.hpp

```
#ifndef LECTEURXPM_HPP
#define LECTEURXPM_HPP

#include <string>
#include <map>
#include <fstream>

#include "Tableau.hpp"

class LecteurXPM
{
private:
  typedef std::map<std::string,int> Dictionnaire;

  Tableau<int> m_pixels;

  int m_nombre_de_colonnes;
  int m_nombre_de_lignes;

  int m_nombre_de_couleurs;
  int m_nb_caracteres_par_pixel;

  void _construitDictionnaire(std::ifstream& fin,
                              Dictionnaire& dictionnaire) const;

  void _litPixels(std::ifstream& fin,
                  const Dictionnaire& dictionnaire);

  void _lit(const std::string& nom_fichier);

  explicit LecteurXPM(const LecteurXPM&  lecteur);
public:
  const Tableau<int>& pixels() const
  {
    return m_pixels;
  }

  const int& nbLignes() const
  {
    return m_nombre_de_lignes;
  }
  const int& nbColonnes() const
  {
    return m_nombre_de_colonnes;
  }

  LecteurXPM(const std::string& nom_fichier);
  ~LecteurXPM();
};

#endif // LECTEURXPM_HPP
```

Utilitaire-3 : Le *design pattern* visiteur

Ce motif de programmation est mis en oeuvre dans la représentation graphique du maillage.

Le maillage dispose alors d'une fonction membre *template*, paramètrée par le type du visiteur, et définie comme suit :

fichier:tableau.hpp

```
1   template<typename ManipulateurDeMaillage>
2   void accepteManipulateur(const ManipulateurDeMaillage& manip){
3       manip.manipule(this);
4   }
```

Un visiteur potentiel doit satisfaire le concept requis, à savoir, disposer d'une fonction membre constante **`manipule`**, de prototype :

fichier:tableau.hpp

```
void manipule(MaillageQuadrangle * maillage) const;
```

Une illustration en est la classe **`DessinateurDeQuadtree`**, qui vous est aussi fournie. Il s'agit en fait d'un dessinateur du maillage de quadrangles (ensemble des feuilles) extrait du **`QuadTree`**. Mais par soucis de clarté, nous avons convenu de la dénommée **`DessinateurDeQuadtree`** et non **`DessinateurDeMaillageQuadrangles`** comme on aurait pu s'y attendre. Nous avons choisi délibérément de ne pas y inclure de commentaire ; cette tâche vous revenant, afin de vous assurer de la compréhension des fondements de cette classe.

fichier:DessinateurDeQuadtree.hpp

```
#ifndef DESSINATEURDEQUADTREE_HPP
#define DESSINATEURDEQUADTREE_HPP
/**
 * @file   DessinateurDeQuadtree.hpp
 * @author Jean-Baptiste APOUNG KAMGA <jean-baptiste.apoung@math.u-psud.fr>
 * @date   Sat Oct 29 11:00:25 2011
 *
 * @brief
 *
 */

/**
 * @brief
 *
 * @class DessinateurDeQuadtree
 */

class MaillageQuadrangle;
class DessinateurDeQuadtree
{
  mutable MaillageQuadrangle * m_pmaillage;
                                  /**< TODO */
public:
/**
 * @brief
 *
 * @fn DessinateurDeQuadtree
 */
  DessinateurDeQuadtree()
  :m_pmaillage(0)
  {
  }

/**
 * @brief
```

```
 *
 * @fn ~DessinateurDeQuadtree
 */
  ~DessinateurDeQuadtree()
  {
  }

/**
 * @brief
 *
 * @fn manipule
 * @param maillage
 */
  void manipule(MaillageQuadrangle * maillage) const;

private:
  class Affiche;
  friend class Affiche;
};

#endif
```

fichier:DessinateurDeQuadtree.cpp

```
#include "DessinateurDeQuadtree.hpp"
#include "MaillageQuadrangle.hpp"
#include "Quadrangle.hpp"

#include <mgl/mgl_qt.h>
#include <mgl/mgl_fltk.h>
#include <mgl/mgl_data.h>
#include <mgl/mgl_glut.h>
#include <cmath>

class DessinateurDeQuadtree::Affiche:public mglDraw
{
private:
  const MaillageQuadrangle & m_maillage;

public:
  int Draw(mglGraph * graph)
  {

    const Tableau < Quadrangle > & quadrangles = m_maillage.quadrangles();
    if(quadrangles.taille() == 0) {
      return 0;
    }

    const Tableau < double > & couleurs = m_maillage.couleurs();


    Tableau < double > triangles_serialise;
    Tableau < double > sommets_serialise;
    Tableau < double > couleurs_serialise;

    int num_sommet = 0;

    for(int i = 0; i < quadrangles.taille(); ++i) {
      const Quadrangle & t = quadrangles[i];
      const Pixel& p0 = t(0);const Pixel& p1 = t(1);
      const Pixel& p2 = t(2);const Pixel& p3 = t(3);

      int i0 = num_sommet,
          i1 = num_sommet+1,
          i2 = num_sommet+2,
          i3 = num_sommet+3;

      sommets_serialise.ajoute(p0.x());
      sommets_serialise.ajoute(p0.y());
      sommets_serialise.ajoute(0.);

      sommets_serialise.ajoute(p1.x());
      sommets_serialise.ajoute(p1.y());
      sommets_serialise.ajoute(0.);

      sommets_serialise.ajoute(p2.x());
      sommets_serialise.ajoute(p2.y());
      sommets_serialise.ajoute(0.);
```

```
      sommets_serialise.ajoute(p3.x());
      sommets_serialise.ajoute(p3.y());
      sommets_serialise.ajoute(0.);

      triangles_serialise.ajoute(i0);
      triangles_serialise.ajoute(i1);
      triangles_serialise.ajoute(i2);

      triangles_serialise.ajoute(i0);
      triangles_serialise.ajoute(i2);
      triangles_serialise.ajoute(i3);

      couleurs_serialise.ajoute(couleurs[i]);
      couleurs_serialise.ajoute(couleurs[i]);

      num_sommet += 4;
    }

    mglData data_s(sommets_serialise.taille() / 3,
                   3, &(sommets_serialise[0]));
    mglData data_t(triangles_serialise.taille() / 3,
                   3, &(triangles_serialise[0]));
    mglData data_v(couleurs_serialise.taille(), &(couleurs_serialise[0]));

    graph->SubPlot(2, 1, 0);

    graph->SetRanges(data_s.SubData(0).Minimal(),
                     data_s.SubData(0).Maximal(),
                     data_s.SubData(1).Minimal(),
                     data_s.SubData(1).Maximal());
    graph->RecalcBorder();
    graph->CRange(0, 255);

    graph->TranspType = 1;
    graph->TriPlot(data_t,
                   data_s.SubData(0),
                   data_s.SubData(1), data_s.SubData(2), data_v, "kw");

    graph->SubPlot(2, 1, 1);

    graph->SetRanges(data_s.SubData(0).Minimal(),
                     data_s.SubData(0).Maximal(),
                     data_s.SubData(1).Minimal(),
                     data_s.SubData(1).Maximal());
    graph->RecalcBorder();
    graph->CRange(0, 255);

    graph->TranspType = 1;

    graph->TriPlot(data_t,
                   data_s.SubData(0),
                   data_s.SubData(1), data_s.SubData(2), data_v, "kw#");
    return 0;
  }

  Affiche(const DessinateurDeQuadtree & visit)
    :  m_maillage(*visit.m_pmaillage){
  }

  static int sdraw(mglGraph* graph, void* p){
   static_cast<Affiche*>(p)->Draw(graph);
   return 0;
  }
};

void DessinateurDeQuadtree::manipule(MaillageQuadrangle * maillage) const{
    m_pmaillage = maillage;
    Affiche affiche(*this);
    char** argv=0;
    /*
    mglGraphQT graph; //commneté car QT ne fonctionne pas !!
    graph.Window(0, 0, &affiche, "Titre");
    mglQtRun();
    */
    mglGraphFLTK graph;
    graph.Window(0, argv, Affiche::sdraw, "Titre",&affiche);
    mglFlRun();
}
```