Master Ingénierie Mathématique (M2) — D5MA2J1 Informatique
Université Paris–Sud — Centre d'Orsay
AN-EDP — Année 2010 − 2011

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 maillage de Delaunay par insertion de sommets. La figure 1 illustre l'objectif du projet. Bien entendu, bien que cette méthode soit une véritable méthode de compression, les techniques utilisée sont simplifiées dans le cadre de ce projet.

Besoin - 1 *Objectif*

FIGURE 1 – A gauche l'image originale, à droite un maillage obtenu.

Une triangulation de Delaunay est un maillage de qui vérifie la propriété de la boule vide. C'est à dire que le cercle circonscrit à tout triangle ne contient aucun autre triangle. En fait, il suffit que toutes les paires de triangles adjacents par une arête vérifient cette propriété pourqu'elle soit vérifiée globalement. De plus, si aucune paire de triangles adjacents ne forme un trapèze, la triangulation de Delaunay est unique. Enfin, la propriété qui nous intéresse le plus pour ce projet est que partant d'une triangulation de Delaunay, pour insérer un sommet s, il suffit de supprimer les triangles dont les cercles circonscrits contiennent s puis d'étoiler la cavité obtenue à partir de s.

L'algorithme de compression d'image est le suivant. On choisit comme maillage initial le rectangle défini par les 4 coins de l'image et on le coupe par une de ses diagonales. Ensuite, on regarde la différence entre la valeur interpolée en certains points des triangles [1] et la valeur du pixel dans l'image originale. Si on juge cet écart trop important, on ajoute alors un sommet. On répète ce processus de manière itérative, jusqu'à obtenir satisfaction.

1. centre de gravité, milieu des arêtes, ...

Besoin - 2 *Classes à écrire*

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

On prendra bien soin à utiliser la compilation séparée et a ne déclarer qu'une classe par fichier. Un `Makefile` aussi complet que possible devra être utilisé.

Besoin-2-1 **:** **`template <typename T> class Tableau`**

On écrira une classe conteneur à accès direct, extensible et permettant de supprimer des éléments. Il s'agit de la classe décrite précisément en TP.

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

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-2-3 **:** **`N2`**

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

Besoin-2-4 **:** **`class Pixel : public N2`**

Cette classe servira à décrire un pixel de l'image originale. 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-2-5 **:** **`class Image`**

Cette classe servira à manipuler l'image originale. Elle ne contiendra pas les **`Pixel`**s mais les construira à la demande. Pour cela, seules les couleurs des points seront stockées dans un tableau d'entiers. De plus, les dimensions de l'image seront conservées dans des membres entiers.

On écrira les fonctions membres suivantes :
- **`int numero(const N2& X) const`** qui renverra le numéro du pixel de coordonnées **`X`**.
- **`Pixel pixel(const N2& X) const`** qui construira le **`Pixel`** de coordonnées **`X`**.
- **`Pixel pixel(const int& x, const int& y) const`** qui construira le **`Pixel`** de coordonnées (**`x`**, **`y`**).
- **`Pixel pixel(const int& i) const`** qui construira le i^e **`Pixel`** de l'image.

- **`const int& nbLignes() const`** qui permet d'accéder au nombre de lignes de l'image.
- **`const int& nbColonnes() const`** pour accéder au nombre de colonnes.
- **`Tableau<double> serialiseCoordonnees() const`** qui traduit dans un tableau de **`double`**s les coordonnées entières de tous les **`Pixel`**s pour l'affichage graphique. Pour chaque **`Pixel`** on donne le triplet $(x, y, 0)$.
- **`Tableau<double> serialiseValeurs() const`** qui traduit les couleurs entières de tous les **`Pixel`**s dans un tableau de **`double`**s. Cette fonction est nécessaire à l'affichage graphique.

Enfin, on écrira une fonction **`void affiche() const`** qui se servira de la classe **`Image::Affiche`** qui vous sera fournie.

Besoin-2-6 : **`class Triangle`**

Il s'agit là d'une classe **`Triangle`** un peu spéciale, puisque les sommets seront des pixels. On écrira les fonctions membres suivantes :

- tout d'abord on surchargera un opérateur pour l'accès aux trois Pixels définissant les triangles.
- On définira l'opérateur de copie.
- **`Pixel barycentre() const`** Cette fonction retourne le **`Pixel`** moyen. Les coordonnées sont les parties entières du barycentre du triangle et la couleur, la couleur moyenne. C'est cette couleur qui sera comparée à celle de l'image.
- **`bool cercleCirconscritContient(const N2& x) const`**. Cette fonction permet de déterminer si le cercle circonscrit du triangle contient le point x. Pour cela, on utilise la propriété suivante : le triangle (ABC) étant défini dans le sens direct, le point D est à l'intérieur du cercle circonscrit à (ABC), si et seulement si

$$\begin{vmatrix} A_x - D_x & A_y - D_y & A_x^2 - D_x^2 + A_y^2 - D_y^2 \\ B_x - D_x & B_y - D_y & B_x^2 - D_x^2 + B_y^2 - D_y^2 \\ C_x - D_x & C_y - D_y & C_x^2 - D_x^2 + C_y^2 - D_y^2 \end{vmatrix} > 0$$

  Attention, pour éviter les débordements des nombres entiers (si on ajoute un à l'**`int`** le plus grand, le résultat est négatif). Les calculs seront à effectuer avec des **`long long int`**.

Besoin-2-7 : **`class Cavite`**

Cette classe à pour rôle de stocker les numéros des **`Triangle`**s à supprimer lors d'une insertion de point ainsi que de stocker les **`Pixel`**s du bord en les ordonnant dans le sens direct autour du point à ajouter.

Pour effectuer ce tri, on utilisera la classe **`map`** de la bibliothèque standard. La clé sera un réel (l'angle) et la valeur, le **`Pixel`**. Pour calculer les angles, on utilisera la fonction **`atan2`**.

On accédera aux tableaux des triangles à supprimer par la fonction

```
const Tableau<int>& numerosAnciensTriangles() const
```

De même on obtiendra les **`Pixel`**s du contour par

```
const Tableau<Pixel>& pixelsDuContour() const
```

Besoin-2-8 : `class Maillage`

Cette classe contiendra une référence sur l'image originale, ainsi qu'un tableau de **`Triangle`**s. Par ailleurs pour éviter qu'un pixel ne soit inséré plusieurs fois, on utilisera un tableau de marqueurs booléens. Il contiendra *faux* pour les pixels pas encore utilisés.

On définira la fonction

```
void ajoute(const int& nombre,const int& precision)
```

qui ajoutera un **`nombre`** donné de sommets ne satisfaisant pas la **`precision`** demandée.

On définira aussi une fonction d'affichage utilisant la classe **`Maillage::Affiche`** qui sera fournie.

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`**.

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_colones;
  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& nbColones() const
  {
    return m_nombre_de_colones;
  }

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

#endif // LECTEURXPM_HPP
```

Utilitaire-3 : classe `Maillage::Affiche`

C'est une classe utilitaire de la classe **`Maillage`**. Elle est donc déclarée dans cette classe sans bien évidemment y afficher son contenue.

Sa définition est entièrement fournie dans le fichier **`Maillage.cpp`** comme suit

Définition de Maillage::Affiche

```
#include "Maillage.hpp"
#include "Cavite.hpp"

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

class Maillage::Affiche
  : public mglDraw
{
private:
  const Maillage& m_maillage;
public:
  int Draw(mglGraph* graph)
  {
    const Tableau<Triangle>& triangles = m_maillage.m_triangles;
    if (triangles.taille() == 0) {
      return 0;
    }

    const Image& image = m_maillage.m_image;
```

```
    // On met tous les sommets de l'image ...
    Tableau<double> sommets_serialise = image.serialiseCoordonnees();

    Tableau<double> triangles_serialise;
    for (int i=0; i<triangles.taille(); ++i) {
      const Triangle& t = triangles[i];
      for (int j=0; j<3; ++j) {
        triangles_serialise.ajoute(image.numero(t(j)));
      }
    }

    Tableau<double> couleurs = image.serialiseValeurs();;
    Tableau<double> constant(couleurs.taille());
    for (int i=0; i<couleurs.taille(); ++i) {
      constant.ajoute(0);
    }

    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.taille(), &(couleurs[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");


    return 0;
  }

  Affiche(const Maillage& maillage)
    : m_maillage(maillage)
  {
  }
};
```

Illustrations

Ci-dessous une illustration des images obtenues pour un certain nombre de pixels. A gauche l'image et à droite le maillage Delaunay associé.

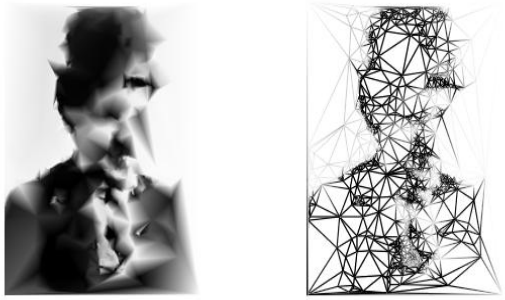

FIGURE 2 – Image compressée avec 1000 points.

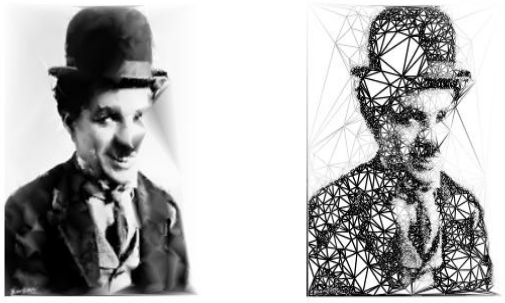

FIGURE 3 – Image compressée avec 5000 points.

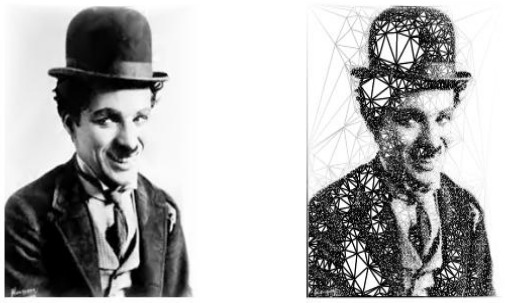

FIGURE 4 – Image compressée avec 100000 points.