Travaux Pratiques
Programmation Multi-Paradigme
Licence 3 Informatique

Julien BERNARD et Arthur HUGEAT

Table des matières

Consignes communes à tous les projets

Au cours de cette UE, vous avez **trois** projets à réaliser à raison d'un projet pour deux séances de trois heures de travaux pratiques encadrées. Les projets sont à faire et à rendre dans l'ordre du présent sujet.

Pour chaque projet, vous devrez implémenter une interface donnée dans un fichier d'en-tête, ainsi qu'un ensemble de tests unitaires pour cette interface. Les tests serviront à montrer que votre implémentation est correcte et complète.

Il est attendu que vos codes sources et les commentaires soient rédigés en anglais et uniquement en anglais.

Projet n°1 : Image

En informatique, il existe de nombreux formats pour stocker les images : *Portable Network Graphics (PNG)*, *Joint Photographic Experts Group (JPEG)*, *Graphics Interchange Format (GIF)*, *Truevision Targa (TGA)*... Tous ses formats ont une structure qui diffère les unes des autres. Les données peuvent être dites compressées s'il est nécessaire d'appliquer un algorithme pour lire les données. Au final, les données seront lues sous forme d'une matrice de pixel. Les pixels peuvent être représentés de plusieurs façons : *Hue Saturation Lightness (HSL)* ou *Red Green Blue (RGB)* pour ne citer que les plus célèbres. Là encore, ces représentations de couleurs ont des variantes ce qui complexifie la gestion des images.

Le but de ce projet sera de proposer une interface permettant de manipuler des images non compressées. La bibliothèque sera capable de lire des images non compressées ou de les modifier. Vous devrez également gérer la conversion entre différentes variantes de couleurs *RGB*.

Pour ce projet, vous avez **l'interdiction absolue** d'utiliser une classe conteneur de la librairie standard (par exemple `std::vector`) et de pointeurs intelligents (par exemple `std::unique_ptr`). En cas de doute, n'hésitez pas à poser la question à votre encadrant de travaux pratiques.

Étape 1 : Gestion des pixels

La couleur d'un pixel sera représentée par la classe templatée `Color` qui stockera les quatre composantes de la couleur : le niveau de rouge, le niveau de vert, le niveau de bleu et l'opacité. Le type du template sera quant à lui le type utilisé pour représenter une composante de la couleur. Une couleur de type *RGB* peut être stockée en mémoire de plusieurs façons. Dans le cadre de ce projet, voici les formats que vous allez devoir gérer :

- RGB : la composante rouge suivie de la verte puis de la bleu
- BGR : la composante bleu suivie de la verte puis de la rouge
- RGBA : la composante rouge suivie de la verte puis de la bleu et de la valeur d'opacité
- BGRA : la composante bleu suivie de la verte puis de la rouge et de la valeur d'opacité

Lorsqu'une couleur est exprimée avec un entier alors la valeur maximale est celle de l'entier. Par exemple, si le type est `uint8_t` alors la valeur maximale sera 255. Dans le cas où le type serait un nombre flottant, alors la valeur maximale sera 1.0.

Afin de passer de cette représentation mémoire à notre classe `PixelRGB`, nous utiliserons les classes templatées `PixelRGB`, `PixelBGR`, `PixelRGBA` et `PixelBGRA`. Chacune de ces classes a deux attributs constants :

- `PlaneCount` qui donne le nombre de composantes que le pixel contient
- `Max` qui donne la valeur maximale pour une composante

Elles définiront également deux méthodes :

- `fromRaw` permet de passer de la valeur en mémoire à notre classe `Color`
- `toRaw` permet de passer de notre classe `Color` a sa représentation en mémoire

Et pour finir, les classes `Pixel*` définissent le type `DataType` qui est le type utilisé pour représenter une composante de la couleur.

Toutes ces classes utilitaires, nous permettrons de pouvoir convertir facilement n'importe quelle couleur de pixel vers sa représentation en mémoire et inversement.

Étape 2 : Les images

Pour représenter une image, nous utiliserons la classe templatée `Image`. Le type du template sera l'une des structures `Pixel*` que nous avons défini lors de l'étape précédente. Cette classe sera chargée garder en mémoire les données de l'image et elle nous permettra d'y accéder. Une image sera également définie par sa largeur et sa hauteur donnée en nombre de pixels. Les données des couleurs doivent être celle de la représentation mémoire et non une liste de `Color`. Elles devront être stockées ligne par ligne dans un tableau contiguë. Par exemple, pour une image de type *BGR*, le tableau devra contenir une succession de composante bleu, puis verte, puis rouge ainsi de suite jusqu'au dernier pixel de l'image.

Pour rappel, vous n'avez pas le droit d'utiliser de classes conteneurs issue de la librairie standard ou de pointeurs intelligents. Cela veut donc dire que vous devez gérer vous-même l'allocation de la mémoire, donc, bien que ce soit une mauvaise pratique en temps normal, vous avez le droit de faire appel aux opérateurs `new` et `delete`.

Afin de simplifier l'écriture du code, la classe `Image` définira les types suivants :
- `DataType` est le type utilisé pour représenter la valeur d'une composante de la couleur
- `ColorType` est le type de notre classe

Nous pouvons construire l'image de plusieurs façons :
- Par défaut, l'image sera vide et aura pour dimension 0 par 0
- En donnant sa dimension et on initialisant chaque pixel à la couleur bleu (la composante bleu devra être à la valeur maximale)
- À partir d'un buffer mémoire (vous n'avez pas à vérifier que ce buffer est valide)

Les méthodes `getWidth` et `getHeight` nous donnerons respectivement la largeur et la hauteur de l'image exprimées en nombre de pixels. La méthode `getData` retournera le pointeur vers la valeur mémoire des pixels. Pour finir les méthode `getColor` et `setColor` vous permettrons de lire ou de modifier la valeur d'un pixel précis.

La classe `Image` permet aussi de passer d'un format de pixel à l'autre que ça soit à la construction d'un objet ou lors d'une affectation. La conversion est possible entre pixels de type différent (*RGB* vers *BGRA* par exemple) mais aussi dans le cas où les `DataType` sont différents.

Étape 3 : Représentation en niveau de gris

Pour cette dernière étape, vous allez de voir permettre la gestion des images en niveau de gris. Jusqu'à présent, les couleurs étaient exprimées avec au moins trois composantes. Dans le cas du niveau de gris, chaque pixel n'a plus qu'une seule composante. L'ajout de ce nouveau type de couleur se fera en ajoutant la structure `PixelGray`. À l'instar des autres structures `Pixel*`, `PixelGray` disposera des mêmes attributs et méthodes. Pour convertir une couleur en niveau de gris, vous utiliserez la formule suivante issue de la recommandation CCIR 601 [1] :

$$Gris = 0.299 \times Rouge + 0.587 \times Vert + 0.114 \times Bleu$$

La conversion d'un niveau de gris en couleur est une opération largement plus complexe qui dépasse de loin la portée de ce projet. Pour nos besoins, nous allons simplement exprimer ce niveau de gris sans interpréter la couleur :

$$Rouge = Gris, Vert = Gris, Bleu = Gris$$

1. `https://en.wikipedia.org/wiki/Rec._601`

Exemple d'utilisation

```cpp
#include "Image.h"

int main() {
  img::ImageRGB imageUInt8(256u, 256u);
  for (int row = 0; row < imageUInt8.getHeight(); ++row) {
    for (int col = 0; col < imageUInt8.getWidth(); ++col) {
      img::ImageRGB::ColorType color;
      color.alpha = 0xFF;

      if (row < 128 && col < 128) {
        color.red = 0xFF;
        color.green = 0x00;
        color.blue = 0x00;
      } else if (row < 128 && col >= 128) {
        color.red = 0x00;
        color.green = 0xFF;
        color.blue = 0x00;
      } else if (row >= 128 && col < 128) {
        color.red = 0xFF;
        color.green = 0xFF;
        color.blue = 0xFF;
      } else if (row >= 128 && col >= 128) {
        color.red = 0x00;
        color.green = 0x00;
        color.blue = 0x00;
      }

      imageUInt8.setColor(col, row, color);
    }
  }

  img::Image<img::PixelRGB<float>> imageFloat(imageUInt8);
  const auto color1 = imageFloat.getColor(0, 0);
  // R=1.0f, G=0.0f, B=0.0f

  const auto color2 = imageFloat.getColor(imageFloat.getWidth() - 1, 0);
  // R=0.0f, G=1.0f, B=0.0f

  const auto color3 = imageFloat.getColor(0, imageFloat.getHeight() - 1);
  // R=1.0f, G=1.0f, B=1.0f

  const auto color4 = imageFloat.getColor(
    imageFloat.getWidth() - 1,
    imageFloat.getHeight() - 1
  );
  // R=0.0f, G=0.0f, B=0.0f

  return 0;
}
```

Projet n°2 : Tuple

Un `tuple` est un objet classique dans les langages de programmation. Il s'agit d'une classe utilitaire qui contient un nombre quelconque de valeurs dont les types peuvent être différents.

Le but de ce projet sera d'implémenter une classe `Tuple`, librement inspirée de la classe `std::tuple` que propose la librairie standard.

Afin de pouvoir fonctionner correctement, les types stockés dans un `tuple` doivent être au moins déplaçables et avoir un constructeur par défaut. Si ce n'est pas le cas, cela entraînera une erreur de compilation. *Ce n'est pas à vous de gérer ce cas de figure.* Dans ce projet, beaucoup de cas d'erreur seront gérés par le compilateur lui-même et ne relève pas de l'implémentation de la classe.

Si les messages d'erreur de `gcc` vous paraisse peu explicite, vous pouvez tester votre projet avec `clang`. Voici la commande pour configurer `cmake` avec `clang`.

```
CXX=clang++ C=clang cmake -B build-clang
```

Étape 1 : Création et récupération

La première étape est donc de pouvoir instancier un `Tuple` et d'accéder à ces éléments. La classe `Tuple` dispose donc d'un constructeur par défaut et d'un second constructeur qui initialise tous les éléments du `tuple`.

Une autre façon de créer un `tuple` est de faire appel à la fonction `t::makeTuple()`. La fonction doit déduire automatiquement le type du `tuple` en fonction des paramètres qu'on lui passe.

Pour récupérer les valeurs d'un `tuple`, on utilise la méthode `Tuple::get()`. Cette méthode est templatée, et elle s'attend à avoir au minimum le numéro de l'élément à récupérer. La méthode existe en deux versions. La première qui est `const` et qui doit renvoyer une référence constante sur l'objet stocké. La seconde qui n'est pas `const` et qui doit renvoyer une référence sur l'objet stocké.

Étape 2 : Opérateurs de comparaison

La classe `Tuple` dispose également de tous les opérateurs de comparaison : `==`, `!=`, `<=`, `<`, `>=`, `>`. Pour comparer deux `tuples`, on compare les éléments deux à deux en partant du premier jusqu'au dernier (ordre lexicographique).

Étape 3 : Opérateurs arithmétiques

Notre classe `Tuple` dispose des opérateurs arithmétiques habituels (`+`, `-`, `*` et `/`) ainsi que leur équivalant d'affectation (`+=`, `-=`, `*=` et `/=`). L'utilisation de ces opérateurs signifie qu'on applique élément par élément l'opérateur correspondant. Cela implique que les deux `tuples` doivent avoir le même nombre d'éléments et que ceux-ci doivent être compatibles entre eux. Il est possible, par exemple, d'additionner un `int` et `double` mais pas un `float` et un `std::string`.

Si les types ne sont pas compatibles ou qu'il n'y a pas le bon nombre d'éléments entre les `tuples`, c'est le compilateur qui retourne une erreur et non l'implémentation de la classe (exception). Ce n'est

donc pas à vous de faire cette vérification.

Étape 4 : Concaténation

Il est possible de concaténer deux `tuples`. Le résultat est un troisième `tuple` qui contient tous les éléments des deux `tuples` d'origine. La concaténation s'appelle via l'opérateur `Tuple::operator|()`.

Exemple d'utilisation

```cpp
#include <iostream>

#include "Tuple.h"

int main() {
  tpl::Tuple<int, double, std::string> t(42, 3.14, "The cake is ");

  std::cout
    << t.get<0>() << ", "
    << t.get<1>() << ", "
    << t.get<2>() << std::endl;
  // 42, 3.14, The cake is

  tpl::Tuple<int, double, std::string> t2(-42, -3.14, "a lie");
  t += t2;

  std::cout
    << t.get<0>() << ", "
    << t.get<1>() << ", "
    << t.get<2>() << std::endl;
  // 0, 0, The cake is a lie!

  auto t3 = tpl::makeTuple(10, 10.0f);
  tpl::Tuple<std::size_t, double> t4(2lu, 2.0);
  auto t5 = t3 * t4;
  std::cout << t5.get<0>() << ", " << t5.get<1>() << std::endl;
  // 20, 20

  auto t6 = std::move(t) | std::move(t5);
  std::cout
    << t6.get<0>() << ", "
    << t6.get<1>() << ", "
    << t6.get<2>() << ", "
    << t6.get<3>() << ", "
    << t6.get<4>() << std::endl;
  // 0, 0, The cake is a lie!, 20, 20

  return 0;
}
```