Accepté au 17ème Colloque GRETSI sur le Traitement du Signal et des Images, Vannes, France, 13-17 septembre 1999.

Vers une réutilisabilité totale des algorithmes de traitement d'images

Thierry GÉRAUD[1], Yoann FABRE[1], Dimitri PAPADOPOULOS-ORFANOS[2], Jean-François MANGIN[2]

[1]Laboratoire de Recherche et Développement d'EPITA
14-16, rue Voltaire, 94276 Le Kremlin-Bicêtre cedex, France

[2]Service Hospitalier Frédéric Joliot, CEA
4, place du Général Leclerc, 91401 Orsay cedex, France
thierry.geraud@epita.fr, mangin@shfj.cea.fr

Résumé – Cet article présente l'évolution des techniques de programmation d'algorithmes de traitement d'images et discute des limites de la réutilisabilité de ces algorithmes. En particulier, nous montrons qu'en C++ un algorithme peut s'écrire sous une forme générale, indépendante aussi bien du type des données que du type des structures de données sur lesquelles il peut s'appliquer. Une réutilisabilité totale des algorithmes peut donc être obtenue ; mieux, leur écriture est plus naturelle et elle n'introduit pas de surcoût significatif en temps d'exécution.

Abstract – This paper presents the evolution of algorithm computation in image processing and discusses the reusability limits of these algorithms. In particular, it shows that an algorithm can have a generic writing in C++ and be therefore independent of both data type and data structure type. Moreover, this total reusability can be obtained with a very comprehensive writing and without leading to a significant additional cost of execution time.

1 Introduction

De nombreux efforts ont été fournis pour la réalisation de bibliothèques dédiées au traitement d'images afin qu'elles soient fédératrices d'outils développés par différents laboratoires. Cependant, une des principales difficultés de tels projets est la prise en compte de la diversité des problèmes abordés en traitement d'images, et en particulier de la diversité des structures de données (images 2D ou 3D, isotropes ou non, séquences ou pyramides d'images, régions ou graphes d'adjacences de régions, collections, etc.) et de la diversité des données elles-mêmes (scalaires, booléennes, entières ou flottantes, complexes, composées comme RVB, etc.)

Dans cet article, nous nous intéressons aux techniques de programmation d'algorithmes de traitement d'images sous le jour de la réutilisabilité de ces algorithmes vis-à-vis du type des données et de celui des structures de données qui peuvent supporter ces algorithmes. Nous allons illustrer nos propos sur deux exemples très simples de traitement : l'ajout d'une constante à la valeur des éléments d'un agrégat (par exemple les points d'une image) et un filtre moyenneur sur un agrégat. Nous verrons finalement qu'une écriture totalement générale de ces algorithmes est possible en C++ grâce à la généricité et au polymorphisme générique ; nous verrons également que cette écriture n'entraîne pas de surcoût significatif en temps d'exécution et qu'elle est proche de la description en langage naturel de ces algorithmes.

Le code que nous citons dans cet article[1] est volontairement simplifié afin d'augmenter la compréhensibilité des techniques de programmation que nous présentons.

1. disponible à partir de l'adresse `http://www.epita.fr/~lrde`

TAB. 1: Définition impérative d'une structure.

```
typedef struct
{
   int     nRows;
   int     nColumns;
   type_t  type;
   void*   data;
}
   Image2D;
```

2 Ecriture impérative

Dans le cadre d'algorithmes écrits dans un langage comme le C, un type de structures de données est défini par une entité structurée, où le champ des données `data` n'est pas typé et où le champ `type` renseigne le type des données ; ce style de définition est illustré en table 1.

L'avantage de cette écriture est de fournir une unique définition par type de structure de données, et ce quel que soit le type des données [2] ; en conséquence, un algorithme de traitement sur une structure de données se traduit par une unique fonction. Pour que l'addition soit supportée pour les images 2D dont le type des données est codé en `char`, une routine spécifique à ce type doit être écrite dans la fonction *ad hoc*, comme le montre la table 2.

T traitements, S types de structures et D types de données peuvent donner lieu à l'écriture de $T \times S \times D$ routines différentes. Afin de réduire cette combinatoire, de nombreuses bibliothèques limitent le nombre de types

TAB. 2: Addition en langage impératif.

```
void add( Image2D* image, void* val )
{
   int iRow, iColumn;
   switch ( image->type )
   {
     case CHAR:
        char** dataChar = toChar2D( image );
        char   valChar = toChar( val );
        for ( iRow = 0; iRow < image->nRows; ++iRow )
        for ( iColumn = 0; iColumn < image->nColumns; ++iColumn )
           dataChar[iRow][iColumn] += valChar;
        break;
     //...
   }
}
```

TAB. 3: Définition générique d'une structure.

```
template<class Data>
class Image2D
{
   int     nRows;
   int     nColumns;
   Data**  data;
   // ...
};
```

de données et de structures de données manipulables par l'utilisateur, et restreignent l'applicabilité de la plupart des traitements à un petit nombre de types de structures et de données. Ainsi, pour les types de structures de données, la plupart des bibliothèques gèrent les images 2D, quelquefois les images 3D, et rarement les graphes d'adjacence de régions. On trouve pour les types de données, du plus classique au moins fréquent, les entiers non signés et signés sur 8 et 16 bits, les flottants, les booléens, les couleurs en rouge vert bleu avec des composantes entières non signées sur 8 et 16 bits.

L'écriture traditionnelle en programmation procédurale entraîne donc une réutilisabilité presque nulle : introduire dans une telle bibliothèque un nouveau type de données ou de structure de données, ou un nouveau traitement impose un effort de réécriture qui se traduit la plupart du temps par des "copier-coller-modifier" manuels.

3 Ecriture classique par objets

3.1 Généricité

Pour s'affranchir du type des données dans l'écriture de routines, certains langages par objets comme le C++ permettent de définir les structures de données à l'aide de classes paramétrées. Cette technique, utilisée dans les bibliothèques IAC++ [6] et IUE [4], se traduit par des définitions telles que celle de la table 3, où le paramètre `Data` est le type des données.

Les traitements sont paramétrés de la même façon, Cf. table 4. A partir de ce code, dit générique, le compilateur construit automatiquement les déclinaisons nécessaires. La déclaration `Image2D<char> image` et l'appel `add( image, 3 )` va ainsi forcer la compilation de la classe `Image2D<char>` et de la fonction `add( Image2D<char>&, char )` ; la routine de la section 2, dédiée à un type de données particulier, est donc produite automatiquement.

TAB. 4: Addition en programmation générique.

```
template<class Data>
void add( Image2D<Data>& image, Data val )
{
   int iRow, iColumn;
   for ( iRow = 0; iRow < image.nRows; ++iRow )
   for ( iColumn = 0; iColumn < image.nColumns; ++iColumn )
      image.data[iRow][iColumn] += val;
}
```

La programmation générique présente deux avantages : le typage est fort (tout `void*` disparaît) et l'écriture de traitements devient indépendante du type des données. Pour le programmeur, la combinatoire est ainsi réduite à $T \times S$ routines ; en revanche, toute fonction est encore dépendante du type de structures de données (ici, image 2D).

Les considérations que nous allons porter dans la suite de cet article sont à notre connaissance originales, l'ensemble des outils de traitement d'images que nous avons trouvés sur Internet n'utilisant pas les techniques que nous allons présenter.

3.2 Polymorphisme classique

Afin que tout type de structures puisse être employé avec un traitement écrit une unique fois, il faut abstraire les détails d'implantation liés à chaque type de structures. Cela est réalisable en présentant une interface unique pour manipuler ces différents types de structures et en faisant intervenir la notion de redéfinition de méthodes via l'héritage.

Dans [3], une modélisation des itérations sur des agrégats est donnée ; nous l'adaptons ici à notre problématique. Deux classes abstraites sont définies : `Aggregate<Data>` et `Iterator<Data>`, qui représentent respectivement un agrégat d'éléments de type `Data` et un itérateur sur cet agrégat. La première classe déclare une méthode abstraite `createIterator()` pour l'instantiation d'un itérateur et la seconde un jeu de méthodes pour la manipulation d'un itérateur : `void first()` pour initialiser sa position sur un premier élément de l'agrégat, `bool isDone()` pour savoir s'il a terminé de parcourir l'agrégat, `Data& value()` pour récupérer la valeur de l'agrégat correspondant à sa position courante, `void next()` pour le déplacer à l'élément suivant de l'agrégat. De ces deux classes abstraites dérivent respectivement les classes concrètes et spécialisées : `Image2D<Data>` et `Image2DIterator<Data>`. Un itérateur sur les points d'une image 2D possède trois attributs : les indices de ligne et de colonne du point courant et la référence à l'image 2D sur les points de laquelle il itère.

Avec cette modélisation, l'écriture de l'addition ne fait intervenir que les classes abstraites comme le montre la table 5. Un procédé de liaison dynamique se charge à l'exécution d'appeler les méthodes spécifiques aux types effectifs, dits types dynamiques. Ainsi, avec la déclaration `Image2D<char> image` et l'appel `add(image,3)`, la méthode `createIterator()` exécutée est celle de la classe `Image2D<char>`, le type dynamique de `iter` est `Image2DIterator<Data>` et les méthodes d'itération exécutées sont effecti-

TAB. 5: Addition par polymorphisme classique.

```
template<class Data>
void add( Aggregate<Data>& aggregate, Data val )
{
   Iterator<Data>& iter = aggregate.createIterator();
   for ( iter.first(); ! iter.isDone(); iter.next() )
      iter.value() += val;
}
```

TAB. 6: Moyenneur par polymorphisme classique.

```
template<class Data>
void mean( Aggregate<Data>& inputAggregate,
           Aggregate<Data>& outputAggregate )
{
   Iterator<Data>
     & inputElt    = inputAggregate.createIterator(),
     & neighborElt = inputElt.createNeighborIterator(),
     & outputElt   = outputAggregate.createFollowerIterator(
                                                  inputElt );
   for_each( inputElt )
   {
      Data sum = 0;
      for_each( neighborElt )
         sum += neighborElt.value();
      outputElt.value() = sum / neighborElt.getCard();
   }
}
```

vement celles d'un itérateur sur une image 2D.

Considérons maintenant l'exemple d'un filtre moyenneur. Nous pouvons définir deux nouveaux types d'itérateurs : un itérateur suiveur et un itérateur de voisinage. Avec une macro `for_each` simplifiant l'écriture de la boucle `for` de la table 5, une écriture par polymorphisme classique est donnée en table 6. Ici, `inputElt` itère sur les éléments de l'agrégat en entrée, `neighborElt` itère sur le voisinage de l'élément courant de l'agrégat en entrée et `outputElt` itère sur l'agrégat en sortie de façon équivalente à l'agrégat en entrée.

Cette technique, qui recourt à l'héritage et à la redéfinition de méthodes, est le polymorphisme que nous qualifierons de classique. Dans nos exemples, il permet de déduire du type de l'agrégat, argument des fonctions de traitement, les itérateurs adéquats. Ainsi, nous nous affranchissons à la fois du type de données et du type de structures de données. La combinatoire est donc minimale (T), un traitement étant programmé une fois pour toutes. De plus, ce code est relativement proche d'une description algorithmique en langage naturel.

3.3 Inconvénients

Les solutions fondées sur le polymorphisme classique présentent cependant quatre inconvénients, énumérés dans [1]. Le plus pénalisant dans le cadre de traitements numériques est le coût lié à chaque appel à une fonction polymorphe. En effet, un tel appel se traduit par deux indirections : une pour l'accès aux méthodes de la classe effective de l'objet ciblé et une pour l'accès à la méthode considérée.

Afin d'estimer le coût de ces indirections, nous avons réalisé des tests de performance. Le premier test correspond à l'addition d'une constante sur une image 2D comportant 8 millions de points ; le second test correspond

TAB. 7: Performance comparée des différentes techniques de programmation.

	approche impérative	polymorphisme classique	polymorphisme générique
addition	0.5 s	1.5 s	0.8 s
moyenneur	3.5 s	11.4 s	3.9 s

TAB. 8: Déclaration de types déduits.

```
template<class Data>
class Image2D
{
public:
   typedef Data                            DataType;
   typedef Image2DIterator<Data>           Iterator;
   typedef Image2DFollowerIterator<Data>   FollowerIterator;
   typedef Image2DNeighborIterator<Data>   NeighborIterator;
   // ...
};
```

à un filtrage par la moyenne sur un 6-voisinage d'une image 3D de 8 millions de points. Les résultats[2] sont donnés par le tableau 7. L'utilisation du polymorphisme classique est rédhibitoire, la répétition des appels aux méthodes `first()`, `isDone()`, `value()` et `next()` étant très coûteuse.

Le polymorphisme classique est une solution intéressante pour s'affranchir du type de la structure de données lors de l'écriture de traitements ; cependant, dans le cadre du traitement d'images où le nombre de points peut être élevé, cette approche “classique” du paradigme objet est inadaptée.

4 Nouvelle approche par objets

Avec l'adoption en 1994 par le comité de normalisation du C++ de la *Standard Template Library* (STL) [5], bibliothèque de conteneurs et d'algorithmes associés, une nouvelle approche de programmation générique s'est révélée ; nous la qualifierons de polymorphisme générique. Depuis, deux bibliothèques utilisent cette technique, CGAL [1] et Blitz++ [7], respectivement pour du calcul géométrique et pour du calcul algébrique.

4.1 Polymorphisme générique

L'idée clef du polymorphisme générique est de paramétrer les algorithmes, non pas par le type des données mais par le type des structures : `Aggregate`. La déduction de types à partir d'un type de structures est réalisée par des définitions de type dans les classes d'agrégats. La table 8 montre pour la classe `Image2D<Data>` un exemple qui permet l'utilisation du type `Image2D<Data>::Iterator`, synonyme de `Image2DIterator<Data>`. La nouvelle écriture de l'algorithme moyenneur est donnée en table 9.

La définition des classes abstraites `Aggregate<Data>` et `Iterator<Data>` n'est plus nécessaire et tout héritage devient obsolète. Chaque méthode correspond ici à un type

2. obtenus avec un PC à 333 MHz sous Linux et avec le compilateur libre `egcs` (disponible pour de nombreuses plateformes à l'adresse `http://egcs.cygnus.com/`)

TAB. 9: Moyenneur par polymorphisme générique.

```
template<class Aggregate>
void mean( Aggregate& inputAggregate,
           Aggregate& outputAggregate )
{
   typename Aggregate::Iterator       inputElt( inputAggregate );
   typename Aggregate::NeighborIterator  neighborElt( inputElt );
   typename Aggregate::FollowerIterator
                              outputElt( outputAggregate, inputElt );
   for_each( inputElt )
   {
      typename Aggregate::DataType::CumulType sum = 0;
      for_each( neighborElt )
         sum += neighborElt.value();
      outputElt.value() = sum / neighborElt.getCard();
   }
}
```

TAB. 10: Code déduit de la table 9 par le compilateur.

```
void mean( Image2D<IntegerS8>& inputAggregate,
           Image2D<IntegerS8>& outputAggregate )
{
   int iRow, iColumn;                     // inputElt attributes
   int iNeighbor;                         // neighborElt attribute
   iRow = iColumn = 0;                    // inputElt.first()
   while ( iRow != inputAggregate.nRows )
                                          // ! inputElt.isDone()
   {
      IntegerS16 sum = 0;
      iNeighbor = 0;                      // neighborElt.first()
      while ( iNeighbor != 4 )            // ! neighborElt.isDone()
      {
         sum +=
         inputAggregate.data[iRow + delta[iNeighbor].iRow]
                          [iColumn + delta[iNeighbor].iColumn];
                                          // neighborElt.value()
         ++iNeighbor;                     // neighborElt.next()
      }
      outputElt.data[iRow][iColumn]       // outputElt.value()
         = sum / 4;                       // neighborElt.getCard()
      if ( ++iColumn == inputAggregate.nColumns )
         iColumn = 0; ++iRow;             // inputElt.next()
   }
}
```

d'objet concret et connu lors de la compilation ; le surcoût lié au polymorphisme classique est donc éliminé.

De plus, le langage C++ offre un mécanisme qui permet au compilateur de remplacer les appels de méthodes par le corps de ces méthodes lorsqu'elles ne sont pas polymorphes. Si la fonction `mean()` de la table 9 est utilisée pour le type d'agrégat `Image2D<IntegerS8>`, la compilation correspondante sera rigoureusement équivalente à celle du code de la table 10.

Au final, les temps d'exécution des algorithmes de traitement ne présentent que de très légers surcoûts par rapport à ceux d'écritures traditionnelles, comme en témoigne le tableau 7. Le polymorphisme générique est donc une solution adaptée à la problématique de l'écriture d'algorithmes généraux, appliquables à des types de données et de structures de données variés.

4.2 Raffinements

Le polymorphisme générique permet de différencier les particularités de certains types de structures et de données. En effet, si un algorithme peut ou doit s'écrire différemment pour un type spécifique de structures, il suffit de surcharger l'écriture générique en définissant une fonction de même signature mais avec un paramètre explicite.

A propos des types de données, un même type prédéfini du langage peut servir à implanter des types de données de sémantiques différentes (par exemple, un entier non signé codé sur 8 bits peut aussi représenter une valeur floue discrétisée) ; pour effectuer leur distinction, qui conduit à des variantes d'implémentations, il est nécessaire de définir les types de données sous forme de classes. Ainsi la classe `IntegerS8` remplace avantageusement le type `char` dans la table 9 ; en particulier, le type de la variable `sum` servant au cumul de valeurs s'en déduit.

Nous développons actuellement une plateforme de traitements d'images qui utilise de façon hybride, en fonction des performances attendues, polymorphismes classique et générique. Sans modifier l'écriture simple des traitements, nous avons également obtenu deux formes supplémentaires de généricité n'entraînant pas de surcoût significatif à l'exécution : l'application de tout traitement peut être restreinte aux éléments d'un agrégat qui vérifient un prédicat donné et peut concerner uniquement un champ des données (par exemple la composante rouge de RVB).

5 Conclusion

Dans cet article, nous avons montré que l'évolution des techniques de programmation permet aujourd'hui d'écrire sous une forme générale des algorithmes de traitements d'images. Ces algorithmes sont alors totalement réutilisables : ils acceptent l'introduction de nouveaux types de données et de structures de données. Leur mode d'écriture, le polymorphisme générique, dont nous n'avons présenté que le principe, est adapté à des traitements de natures variées, par exemple des filtrages, des méthodes markoviennes ou de l'espace échelle.

Références

[1] A. Fabri *et al.*, *On the design of CGAL, the computational geometry algorithms library*, Rapport de recherche num. 3407, INRIA, 1998.

[2] M.R. Dobie and P.H. Lewis, *Data structures for image processing in C*, Pattern Recognition Letters, vol. 12, num. 8, pp. 457–466, 1991.

[3] E. Gamma *et al.*, *Design Patterns – Elements of reusable object-oriented software*, Addison-Wesley, 1994.

[4] C. Kohl and J. Mundy, *The development of the Image Understanding Environment*, in proc. Int. Conf. on Computer Vision and Pattern Recognition, pp. 443–447, 1994.

[5] A. Stepanov and M. Lee, *The standard template library*, Rapport, Hewlett-Packard, 1994.

[6] G.X. Ritter, J.N. Wilson and J.L. Davidson, *Image Algebra: an overview*, Computer Vision, Graphics, and Image Processing, vol. 49, num. 3, pp. 297–331, 1990.

[7] T.L. Veldhuizen, *C++ templates as partial evaluation*, Rapport technique num. 519, Indiana University Computer Science, 1996.