Software and Hardware Architectures for Image Processing

An Image Processing Library based on Abstract Image Data-types in C++*

Dennis Koelma and Arnold Smeulders

Faculty of Mathematics and Computer Science, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
tel.: +31 20 525 7516, e-mail: koelma@fwi.uva.nl

Abstract. The paper presents a library for image processing based on abstract image data-types. The library finds a balance between programmability and efficiency of implementation of image processing operations. Operations are designed using abstract concepts for images and pixel manipulation to be independent of the image data-type. Operations may be optimized in an iterative way by replacing abstract concepts with data-type specific operations, reaching the level of efficiency of operations designed from scratch for a specific image data-type.

1 Introduction

The emerging of abstract data types and the object-oriented (OO) paradigm in good programmer's practice makes it worth while to reconsider the design of image processing libraries. In this paper we will review the design considerations of an image processing library and discuss an implementation of such a library in C++.

Over the years it has proven difficult to reach consensus on the functionality of an image processing library [1]. In fact, not more than 25 % of the 22 libraries surveyed in the framework of the ANSI proposed imaging standard coincide on the bulk of supported operations. The driving force behind the diversity are the conflicting design criteria to which every specific library is a compromise. The conflicting criteria include efficiency, applicability, extensibility, programmability, maintainability, and portability. As they are important in the design of any library we review them briefly.

Efficiency in image processing operations on standard hardware is achieved through decent programming and the use of specialized image data structures [2] or assembler and microcode [3]. In a library optimized for efficiency, images are typically represented in specific form, e.g. two-dimensional grey-value images in [2]. The focus on efficiency in the design reduces the possibility to extend the library with other image data-types. In general, designing a software library with efficiency as the primary target reduces its applicability, programmability, extensibility, and maintainability.

* This research was supported by STW project AWI92-1691

The lack of consensus in software libraries for image processing implicates the need for extensibility. Extension of a library with new operations may be facilitated through a high level of programmability. Math-like languages [4] and image algebras [5] use abstract specifications to facilitate implementation of operations. We feel that the high level of programmability in these approaches enhances extensibility but reduces efficiency. The loss of efficiency is due to neighbourhood addressing by indexing, requiring an additional function to translate a logical address to a physical address. Producing efficient code is even more difficult in case specialized data structures are used to store images, e.g. bitmaps for binary valued images [6].

Our aim in designing a software library is to find a balance between programmability and efficiency of image processing operations. The library is based on concepts designed to provide a high level of programmability and a simple optimization process. Operations implemented using the abstract concepts are valid for all existing and future image data-types. Once implemented, operations can be made more efficient in an iterative way by replacing the abstract concepts with data-type specific methods.

A remaining requirement to influence the design of our library is the need for portability. To assure portability, the library is implemented in a standard programming language.

In section 2 of the paper the design of the abstract concepts in the library is described. The implementation of some commonly used image processing operations is demonstrated in section 3. Section 4 describes the optimization of operations.

2 Concepts

Following the OO methodology we distinguish the following important concepts in the library: images, operations on images, and pixel manipulation.

2.1 Images

Image types are usually ordered in a class hierarchy to obtain a clear correspondence between operations and image types. A reasonable class hierarchy for image types is based on the following defining characteristics of images:

- number of dimensions of an image
- form of the pixel representation
- semantics of the contents of an image

The number of dimensions has a large influence on addressing of (neighbouring) pixels in an image. We consider images to be two- or three-dimensional observations of the real world. The number of dimensions of an image is therefore two or three. We do not represent scale spaces or time series in an equal dimension as they are essentially different from images. They are considered to be time or

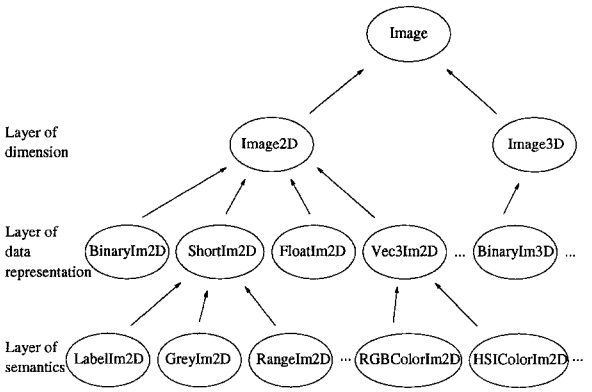

Fig. 1. Image class hierarchy

scale sets of two- or three-dimensional images with their own set of admissible operations in that dimension.

The representation of the pixel data is an important aspect in the definition of the class hierarchy as it determines how operations have to manipulate the actual pixel values. The pixel value representation is covered by introduction of a concept to facilitate future extensions with new pixel forms.

The semantics of the contents of an image gives an interpretation to the pixel values. In most operations, semantics will not influence manipulation of pixel values but in some operations it makes a difference.

We have adopted a hierarchical image class structure (see Fig. 1) avoiding multiple inheritance. Operations that are easily specified for all image types are implemented at the `Image` class level and inherited by all other classes. Operations can be optimized for a specific image type by re-implementing them at a lower level of abstraction in the hierarchy.

2.2 Operations on Images

Operations on images can be expressed in a number of ways. Consider the following list:
(i) `c=a+b`, (ii) `add(a, b, c)`, (iii) `a.add(b, c)`, and (iv) `c.assign(a).add(b)`.

The first way is a very elegant solution. However, the mathematical syntax is difficult to incorporate in a general purpose programming language. It also has an undesired side effect on memory allocation in that temporary objects are created in longer expressions: `c=a+b-d+7` has to be interpreted as `c=((a+b)-d)+7`.

The second approach is the classical way of expressing an operation and its arguments in a function-call. In this case, extension to larger expressions is not elegant. It can only be done through creation of temporary variables or introduction a new function for every new number of arguments.

Translation of the function-call approach to an OO approach results in the third expression. This approach has the same problem as the previous one when it is extended to larger expressions. Furthermore, it implies that objects create new objects which is not consistent with good OO practice [2].

The last approach suits the OO paradigm in the sense that each object hides its data from the outside world and functions are expressed as methods of a class. Also, creation of temporary objects or unnecessary copying is avoided. The approach yields an orthogonal function interface that is easily extensible for the specification of larger expressions: `c.assign(a).add(b).sub(d).add(7)`. The approach is adopted in our library.

2.3 Pixel Manipulation

In a general form, a pixel holds an N-dimensional data vector. Using a class to capture the concept of a pixel introduces a layer of abstraction that leads to a considerable loss of performance. The use of a separate pixel class also implies that storage of pixels cannot be optimized by the image class. For efficiency reasons we do not use a separate pixel class but defer storage of pixels to the image class.

There are two ways of addressing pixels in an image: by indices or by pointers. Indices imply a higher level of programmability than pointers because they offer more degrees of freedom in addressing pixels. The additional degrees of freedom also make operations using indices more difficult to optimize. We have chosen to use pointers for efficiency.

Pixel pointers are based on a common pixel data type. Pixel values expressed in the common pixel data type can be read-from or written-to the location indicated by a pixel pointer.

3 Image Processing Operations

To give an impression of the functionality of the concepts in the library we demonstrate the implementation of some typical image processing operations like point and neighbourhood operations. The example operations are given here without sanity checks to save text.

The first example is pixel by pixel addition of two images, called by `a.add(b)`:

```
Image &Image::add(const Image &op) {
  int npix = size();
  PixelPtr &p1 = pixel_ptr_clone(); PixelPtr &p2 = op.pixel_ptr_clone();
  while (--npix ≥ 0) { p1.write(p1.read() + p2.read()); p1++; p2++; }
  delete &p1; delete &p2; return *this;    //this image contains the result
}
```

Neighbourhood operations can be expressed on an abstract level through the generalized-template operation [5]:

```
Image &Image::g_template(const Image &templ, PixelIter &titer,
                                                PCDF g_mul, PCDF g_add) {
 Image &res = clone(0); PixelPtr &rptr = res.pixel_ptr_clone();
 PixelPtr &sptr= pixel_ptr_clone(); PixelPtr &tptr= templ.pixel_ptr_clone();
 PixelIter &siter = titer.clone(); siter.image(this); siter.check(1);
 for (k=0 ; k<depth() ; k++) {
   for (j=0 ; j<height() ; j++) {
     for (i=0 ; i<width() ; i++) {
       Common val; siter.center(Point(i, j, k)); titer.reset();
       titer.next(tptr); siter.next(sptr); val = g_mul(tptr.read(), sptr.read());
       while (titer.next(tptr)) {
           siter.next(sptr); val = g_add(val, g_mul(tptr.read(), sptr.read()));}
       rptr.write(val); rptr++; } } }
 delete &rptr; delete &tptr; delete &sptr; delete &siter;
 eat(res); return *this;          //cannibalism: steal data and destroy res
}
```

The operation combines weights of a template (`templ`) with pixel values in the neighbourhood (defined through an auxiliary class `PixelIter`) of a pixel by means of the `g_mul` operation. The resulting set of values is reduced to a single value by applying the `g_add` operation to the elements of the set in an iterative way. The operation will, for example, perform a convolution in case `titer` is set equal to `IterArea2D` (iterate over pixels in a rectangular, two-dimensional area in a row-wise order), `g_mul` is multiplication, and `g_add` is addition.

4 Optimization

The function `add` is optimized for the short-valued two-dimensional images by adding a function `add` to the interface of class `ShortIm2D`. The virtual function mechanism of C++ results in a call to `ShortIm2D::add` instead of `Image::add` whenever the object is an instance of the `ShortIm2D` class.

Optimization is straightforward in case pixels are actually stored in an array as assumed implicitly by the image concept. For example, optimizing the function `add` for the image class `ShortIm2D` consists of replacing the abstract `PixelPtr`'s with pointers to `short`'s and replacing the `read` and `write` functions with the pointer dereferencing operation.

Optimization is more difficult in case pixels are not stored in an array-like data structure. The efficiency of binary mathematical morphological operations on binary images can be improved significantly by representing images as bitmaps [6]. Direct translation of an operation defined for the `Image` class to an operation for the `BinaryIm2D` class will not produce the desired efficiency. In this case, operations must be redesigned on an algorithmic level to take optimal advantage of the bitmap structure.

To give an impression of the loss of efficiency caused by introducing abstract concepts we compare the execution speed of abstract operations with their optimized counterparts. In the abstract operations two different common pixel

representations are used: a vector of three doubles (`Common`) and a single double (`Single`).

As expected, optimized point operations execute at the same speed as C-functions performing the same operations on continuous memory blocks. Abstract point operations using `Common` are typically 4-8 times slower. Using `Single` reduces the factor to 3-6.

Abstract neighbourhood operations have a larger overhead factor because they require more pixel operations. A Sobel operation using `Common` is typically 15-20 times slower, using `Single` yields a factor of 5-10.

5 Discussion

We have presented an image processing library based on abstract data-types that finds a new balance between programmability and efficiency. The library is based on abstract concepts for images, operations on images, and pixel manipulation. Operations based on the abstract concepts are valid for all existing and future image data-types, reducing the amount of software to be maintained. For a library with more than 10 different types of images, as exemplified in Fig. 1, the source code will collapse by a factor over 10. Even greater factors can be reached by using multi-functional operations like the generalized template operation but such only at the expense of a severe loss of performance. The high level of programmability provided by the concepts allows for fast prototyping of new image processing operations, making the library easily extensible.

The improved programmability, maintainability, and extensibility of the library is payed for by a significant loss of efficiency. However, with current workstations a typical operation will still provide interactive response and thus not hamper prototyping the operation. When needed, an operation may be optimized for efficiency upon completion of its development. Our approach of optimizing individual operations allows us to reach the same level of efficiency as operations designed from scratch for a specific image data-type.

References

1. T. Butler and P. Krolak. An overview of the programmer's imaging kernel (PIK) proposed standard. *Computers and Graphics*, 15(4):465–472, 1991.
2. J. Piper and D. Rutovitz. Data structures for image processing in a C language and Unix environment. *Pattern Recognition Letters*, 3:119–129, March 1985.
3. A. C. Sleigh and P. K. Bailey. DIPOD: an image understanding development and implementation system. *Pattern Recognition Letters*, 6:101–106, July 1987.
4. D. Hatfield, R. A. Miner, and C. T. Wilkes. A mathematical expression language for imaging applications. *Computers and Graphics*, 15(4):495–506, 1991.
5. G. X. Ritter, J. N. Wilson, and J. L. Davidson. Image algebra: an overview. *Computer Vision, Graphics, and Image Processing*, 49:297–331, 1990.
6. R. van den Boomgaard and R. van Balen. Methods for fast morphological image transforms using bitmapped binary images. *CVGIP: Graphical Models and Image Processing*, 54(3):252–258, May 1992.