Reusable Implementations are Necessary to Characterize and Compare Vision Algorithms

Position Paper for Workshop "Performance Characteristics and Quality of Computer Vision Algorithms", Braunschweig, 18.9.97

Ullrich Köthe
Fraunhofer-Institut für Graphische Datenverarbeitung Rostock, Germany
Email: koethe@egd.igd.fhg.de

Abstract

This paper argues that the difficulties in implementing computer vision algorithms are a major reason for the lack of research into algorithm comparison. We conclude that it is important to represent algorithms in form of *reusable code*. Since current vision systems do not fulfill all requirements we must pose on reusable implementations, we propose to solve the reuse problem by applying *generic programming*. We define two dimensional iterators, which mediate between image processing algorithms and their underlying data structures, so that the same algorithm implementation can be applied to any number of different image formats. The elegance and efficiency of this approach is illustrated by a number of useful examples.

Keywords: reusable algorithms, low-level image processing

1 Introduction

If you want to test an algorithm, you must first implement it. This sounds trivial, but unfortunately it isn't – implementing vision algorithms correctly and efficiently is often difficult and time consuming. In fact, I believe that these difficulties are a major reason for the lack of activity in the field of algorithm characterization and comparison. This becomes apparent if we take a closer look at the relationship between the implementation of algorithms and their comparison.

Typically, the implementation of a complicated vision algorithm takes several weeks. At least four steps are involved: getting familiar with the general approach of the solution, understanding the algorithm, coding and, finally, testing for correctness. If we consider that one and the same problem can be solved by methods as diverse as differential geometry, statistics, knowledge based methods and neural networks (to name just a few), we immediately

realize that understanding an algorithm is not an easy task. Matters are further complicated by the fact that any publication describing an algorithm has to omit important detail which significantly influences the algorithms performance. This includes issues like the numerical precision during calculations, the method to solve a particular equation, filter sizes and so on. Good choices for details like these can only be obtained by experimentation which again takes time and requires an even deeper understanding of the algorithm.

When it finally comes to testing, one encounters a real dilemma: If the algorithm does not perform as advertised, is this an inherent property of the algorithm, or is it a consequence of an incorrect or suboptimal implementation? It is not uncommon that bugs in the implementation are discovered only after several months of algorithm usage. If a bug is discovered to late, all algorithm characterizations done so far have to be repeated. It takes a long time until one develops the necessary confidence that the implementation is correct and the algorithm is wrong.

In practice these difficulties have severe consequences. Most publications introducing a new algorithm only compare its performance with one or two simple reference algorithms. For example, new edge detectors are typically compared with the Canny detector, although more than ten years of further research have produced much better algorithms that should be used as references instead. In cases where a more sophisticated reference algorithm was actually used, I often got the impression that it was not implemented as carefully as the new algorithm, so that the comparisons did not really prove anything. But in light of the difficulties in implementing algorithms, this comes at no surprise: Few projects have the resources to implement several algorithm variants just to throw away all but the best after the tests have been completed.

To overcome these problems, many people, especially from industry, request academia to carry out projects dedicated to algorithm comparison. If standard data sets were used for these tests, and results carefully documented, a data base could be established which served as a guide to select appropriate algorithms for new applications. Such a database were certainly a big step forward, but I don't believe that this alone would solve the problem, because objective, reproducible criteria to characterize input data and document test results are still lacking for most image analysis tasks. So it is unclear, whether and how tabulated test results could be successfully transferred to new applications.

Consequently, even if the database existed, many people would still wish to test algorithms on their own data to get reliable comparisons. Since most cannot afford implementing all candidate algorithms themselves, they would like to *reuse* proven algorithm implementations. Hence, to facilitate algorithm characterization and comparison, we must also devote our attention to the question of reusable implementations.

2 Current Approaches to Reuse

Implementing reusable code seems to be more difficult in computer vision than in other fields of computer science. In part, this stems from the fact, that the vast amount of image data to be processed makes computational speed one of the highest priorities. Until recently, most programming had to respect the speed requirements of the underlying hardware. Since the hardware was continually changing, code reuse was virtually impossible. Due to the increasing performance and widespread application of standard computers code reuse has now become a realistic option, but speed still plays an important role. Lets analyse the current situation by looking at three representative examples.

The first successful attempt to standardize computer vision algorithms was the Programmer's Imaging Kernel System (PIKS), which was approved by ISO in 1994 as part of the IPI standard [1]. It defines a set of basic image data structures and some 120 low-level image processing algorithms. Unfortunately, the standard was defined using the structured programming paradigm which has a number of severe drawbacks, including:

- In structured programming one first defines data structures and than builds algorithms which directly access these data structures' attributes. This ensures very fast computation, but also results in a close coupling between algorithms and data structures. Consequently, whenever a data structure changes all algorithms using that data structure must be modified. In practice this means that the standard's algorithms can not run on top of your existing data structures, nor can you modify the standard's data structures to fit into your environment. In effect, a smooth transition from the old environment to the standard or an integration of the two is virtually impossible.

- The static nature of a structured design makes it difficult to define data structures for intermediate and high-level vision tasks where a lot of flexibility is needed. This was probably one reason that the standard was confined to low-level image processing. PIKS therefore did not sufficiently reflect the state of the art in computer vision and provided little beyond what most vision groups already had implemented.

Consequently, the PIKS standard did not gain widespread acceptance. Meanwhile, the object-oriented programming paradigm emerged, promising the elimination of the drawbacks of structured programming. Among others, the Image Understanding Environment (IUE) [2] was defined to become a new, object-oriented, de-facto standard for computer vision applications.

By using object-oriented modeling techniques the IUE provides flexible data structures (classes) and associated algorithms for all phases of image

analysis – low-level image processing, geometric and topologic feature representation, high-level object reconstruction, along with sensor models, coordinate systems and data exchange facilities. Using inheritance, the existing functionality can, to a certain extend, be adapted to new environments. Thus the IUE eliminates many of the drawbacks of the PIKS standard and provides a much wider range of solutions.

The power of the IUE comes, however, at the price of extremely high complexity. The core functionality alone accounts for more than 400 classes and several dozen Megabytes of library code. As more and more algorithms are added, the size of the system continues to grow. Unfortunately, the design approach chosen for the IUE has the tendency to make all parts of the huge system depend upon each other. It is not possible to split the system into independent parts – the IUE has to be used and understood as a whole.

Moreover, if we take a closer look at how the IUE implements algorithms, we see that they are still pretty closely coupled to the data structures. This results from the fact that virtual functions, the standard mechanism to achieve flexibility in (statically typed) object-oriented systems, are to slow for many computer vision tasks, because they add a lot of processing overhead. The IUE works around this by implementing algorithms as class member functions (so that the objects' data can be accessed directly) or using iterators and data accessors which buffer data for faster access. This complicates the system design significantly and gives away some of the flexibility gained by the object-oriented approach. In particular, it is still impossible to run algorithms on top of data structures defined outside the IUE.

A completely different road to reuse is taken by the KHOROS system [3]. KHOROS has probably enjoyed the greatest amount of reuse among all available vision systems. This is due to its component based approach: KHOROS consists of a collection of small, independent image processing modules which can be plugged together freely to accomplish very complicated tasks. Since all modules are independent of each other it is quite easy to write new modules or transform existing vision algorithms into a KHOROS conforming form. An intuitive and flexible user interface makes KHOROS an ideal platform for rapid prototyping and solution exploration.

The first version of KHOROS used a structured programming approach to data modeling. Therefore it was, like PIKS, mostly restricted to low-level image processing. In KHOROS version 2, the data model was extended by geometric and graph data structures using object-oriented design principles. Unfortunately, this new design was not implemented in an object-oriented language so that many advantages of the object-oriented approach were lost.

The ideal approach to reusable programming would unify the speed of structured programming with the modeling power of the object-oriented approach and the flexibility of a component based system. It should meet the following design requirements:

- low overhead, high performance
- powerful data representation
- portability to many systems
- high flexibility (independent modules, low coupling)
- separation between data structures and algorithms – algorithms can run on top of different data structures
- ease of use

Fortunately, such a design approach does already exist: *Generic programming* was invented by A. Stepanov and D. Musser with the explicit goal of meeting the above requirements [4]. Meanwhile, generic programming has become the underlying design methodology of much of the emerging C++ standard library. In the sequel, we will explain the basic principles of this new approach and apply it to the development of reusable algorithms in computer vision.

3 Definitions

In this paper images are defined as 2 dimensional[1] rectangular arrays with range $x = 0, ..., w-1$ and $y = 0, ..., h-1$. The point (0, 0) is in the upper left corner of the array, x increases from left to right and y from top to bottom. The grid positions are referred to as *pixels*, and the data types stored in a pixel as *pixel types*. A pixel type may be any type supported by the programming language.

Pixels outside the valid range that have at least one 8-neighbor inside the image are referred to as *past-the-border* pixels. Pixels at which data can be accessed are called *dereferencable*. Normally all inside pixels are dereferencable. Pixels outside the image can be dereferencable in certain situations.

4 Generic Algorithms

The need for reusable algorithms and better separation between them and the underlying data structures lead to the idea to abstract the logical behaviour of data structures, i.e. their navigation and access functionality, into separate classes called *iterators*[2]. Algorithms do not longer access data directly but

[1] Extension of the results to arbitrary dimensions is straightforward.

[2] Iterators are not new. Image iterators are, for example, present in the IUE. But they have been used quite differently from what will be described here.

always via mediating iterators which adapt different data structure implementations to the algorithms' expectations. By treating iterators as formal parameters of the algorithms, such algorithms can run on top of many different data structures as long as the iterators conform to the assumptions on which the algorithm is based. This philosophy of *generic programming* was first proposed by Stepanov and Musser [4],[5] and became popular with the inclusion of their Standard Template Library (STL) into the C++ standard [8],[6]. In C++, the template mechanism is used to realize generic algorithms by means of *compile-time polymorphism* so that the performance overhead is very small, if not completely eliminated. The present paper introduces iterators for 2-dimensional data structures that conform as closely as possible to the style of the C++ standard library.

Implementing a generic algorithm starts by identifying the essential access and navigation functions it needs. The algorithm is then written entirely in terms of a suitable abstract iterator that provides this functionality. It can now be instantiated for any concrete iterator type that translates the functionality of some concrete data structure into whatever the algorithm expects. The generic approach has a number of important advantages:

- The iterator must conform to the abstract interface, but it need not be derived from a specific abstract base class. Thus, we have more flexibility than in an inheritance based approach.
- It is much easier to implement an iterator than to port an algorithm to a new environment.
- Using inline functions and templates, a reusable algorithm in C++ runs almost as fast as an algorithm tailored to a specific data structure.

It is important to note that iterators can be build for existing data structures, even if these are not objects, without affecting any existing code. So you can switch to generic programming at any time without the need to redevelop a huge code base.

Lets look at a simple example: converting a RGB image to a gray valued image. For this task we need an iterator that can read the red, green and blue components of the RGB image, another that can write the results into the gray image, and a third that marks the end of the iteration. This gives the following generic algorithm:

```
template <class SequentialIterator, class RGBSequentialIterator>
void rgbToGray(SequentialIterator dest, SequentialIterator end,
         RGBSequentialIterator source)
{
    for(; dest != end; ++dest, ++source)
    {
```

```
        *dest = 0.3 * source.red() +
                0.59 * source.green() +
                0.11 * source.blue();
    }
}
```

Now assume that we have an image `ByteImage` that is able to create a `ByteImageIterator` and an RGBImage that creates an `RGBImageIterator` Then the above algorithm may be used like this:

```
void rgbToGray(ByteImage & result, RGBImage & src) {
    ByteImageIterator i = result.begin();
    ByteImageIterator end = result.end();
    RGBImageIterator rgb = src.begin();
       // instantiate generic algorithm
    rgbToGray(i, end, rgb);
}
```

Of course, the `ByteImage` rounds-off the exact result of the conversion to an integer. If we want to keep the accuracy, we can use a `FloatImage`:

```
void rgbToGray(FloatImage & result, RGBImage & src) {
    FloatImageIterator i = result.begin();
    FloatImageIterator end = result.end();
    RGBImageIterator rgb = src.begin();
        // the same generic algorithm, now using FloatImages
    rgbToGray(i, end, rgb);
}
```

The basic functions the gray image iterator must provide are `operator!=()`, `operator++()`, and `operator*()`. Every data structure for which these operators can be implemented may be used in the algorithm. Let the `ByteImage` be a simple class that stores the image data in a standard C array:

```
class ByteImage {
  public:
    // ...
    ByteImageIterator begin();
    ByteImageIterator end();

  private:
    int width, height;
    unsigned char * the_pixels;
};
```

In this case a standard C pointer can be used as an iterator, because the necessary operators are automatically defined for pointers:

```
typedef unsigned char * ByteImageIterator;

ByteImageIterator ByteImage::begin() { return the_pixels; }

ByteImageIterator ByteImage::end() {
    return the_pixels + width*height;
}
```

Now assume that the pixels in a `FloatImage` can only be accessed via a function `pixel(x,y)` that returns a reference to the specified pixel. Then we can implement an iterator that stores its current coordinates, but delegates data access to the image's `pixel(x,y)` function:

```
class FloatImageIterator {
  public:
    // ...
    // delegate access
    float & operator*() { return image->pixel(x, y); }

  private:
    FloatImage * image;  // the image to delegate to
    int x, y;            // the iterator's current position
};
```

The RGB iterator must provide functions that return the current pixel's red, green, and blue channels. A few examples should suffice to show how this can be implemented:

- A common form of RGB images stores each color channel as a separate band. Then the RGB iterator can simply encapsulate 3 standard sequential iterators:

  ```
  class RGBByteImageIterator1 {
    public:
      // ...
      unsigned char red() { return *red_; }
      unsigned char green() { return *green_; }
      unsigned char blue() { return *blue_; }

      void operator++() { ++red_; ++green; ++blue_; }

    private:
      ByteImageIterator red_, green_, blue_;
  };
  ```

- Another widely used form is to store all three values in a `long int`:

```
class RGBByteImageIterator2 {
  public:
    // ...
    unsigned char red() { return *current_ & 0xFF; }
    unsigned char green() { return (*current_ >> 8) & 0xFF; }
    unsigned char blue() { return (*current_ >> 16) & 0xFF; }

    void operator++() { ++current_; }

  private:
    LongIntImageIterator current_;
};
```

  A similar design results if a `struct RGBPixel` is used instead of `long int`.

- In a color-mapped image the iterator navigates the array of color indices, and the accessor funtions perform the color lookup using the current index.

5 Requirements for Image Iterators

In the STL [8],[6], iterators are classified in terms of the operations they can execute in *constant time*. Every iterator must have advancement to the next element (`++iter`, `iter++`), access of the current element (`*iter`), and check for equality with another iterator (`iter == other_iter`, except for the output iterator) defined and be executed in constant time. *Forward iterators*[3] provide exactly this functionality. *Bidirectional iterators* add backward iteration (`--iter`, `iter--`), and *random access iterators* also provide arbitrary offsets (`iter+=n`, `iter-=n`), relative access (`iter[n]`), distance calculation (binary `iter - other_iter`) and comparision (`iter < other_iter`). In addition, each iterator can be constant (`*iter` returns a value, i.e. is read-only) or mutable (`*iter` returns a reference, i.e. is read-write). The end of the sequence is always marked by a *past-the-end* iterator to which another iterator can be compared. These features ensure backward compatibility of iterators with standard C pointers which meet all requirements of random access iterators.

The iterators defined in the previous section are forward iterators. But it would not be difficult to extend them to random access iterators. However,

[3] In the sequel we do not consider input and output iterators.

it is important that the algorithms do not use more functionality than they actually need as to not unnecessarily restrict the classes of images they can operate on.

In this paper we want to built upon and extend this classification according to the specific needs of image processing. In doing so we need to find an optimal trade-off between simplicity and functionality. Some directions along which the STL model will be extended include:

- Navigation must be possible in two dimensions.
- The image border can not be marked by just one past-the-end iterator.
- Some iterators must provide easy access to certain 2D neighborhoods.
- Some iterators must know about their current coordinates.
- Access to the pixel data via `operator*()` is not always the best solution.

6 1-dimensional Image Iterators

The iterators introduced so far interpret the image as a sequence of pixels and hide its 2-dimensional structure. Thus, they are instances of *1-dimensional image iterators*. 1D image iterators are characterized by the fact that there is no way to change the iterator's position independently in x and y directions. The navigation functions like `++iter` always influence both coordinates or do not touch one coordinate at all. Several useful iterators that provide different 1-dimensional views onto the image data can be defined following the scheme of the STL, for example:

SequentialIterators are forward iterators that interpret the image as a 1-dimensional sequence of pixels.

LineIterators are random access iterators that iterate along an arbitrary straight line. They can, e.g., be used to extract a profile along this line.

ChainIterators are bidirectional iterators that iterate along an arbitrary curved line. Internally, the sequence of pixels visited may, for example, be specified by a chain code. They are useful for dealing with curved lines and region boundaries.

ROISequentialIterators are bidirectional iterators that successively visit the pixels of a region of interest of arbitrary shape. Use these iterators to calculate statistics on regions.

The latter three iterators must be able to tell their current coordinates.

7 2-dimensional Image Iterators

7.1 Navigation

To fully utilize the image data structure we need iterators that can be moved in all directions independently. This requires a mechanism to specify which coordinate a navigation command refers to. This is not directly possible using operator overloading. Instead, we could define navigation functions like `iter.incX()` or `iter.subtractY(dy)`, but in C++ there exists a more elegant solution using nested classes that define different views onto the same data. More specifically, consider the following iterator design:

```
class ImageIterator {
  public:
    // ...

    class AsX {
        // data necessary to navigate in X direction
      public:
        // navigation function applies to X-coordinate
        void operator++();
        // ...
    };

    class AsY {
        // data necessary to navigate in Y direction
      public:
        // navigation function applies to Y-coordinate
        void operator++();
        // ...
    };

    AsX x;                // x-view to the data
    AsY y;                // y-view to the data
};
```

Now we can use the nested objects x and y to specify the direction to move along:

```
ImageIterator i;

++i.x;  // move in x direction
++i.y;  // move in y direction
```

This way we can define 2D iterators in terms of the same constant time operations as 1D iterators. A categorisation similar to the STL iterator cat-

egories seems to be unneccessary as images are always random access data structures.

Table 1 lists the required `ImageIterator` navigation functionality. (In the table i, j, k are `ImageIterators` of identical types, with i and j referring to the same image, `dx` and `dy` are `int`'s, and s is of type `Size2D` which is defined as

```
struct Size2D {
    int width, height;
};
```

All navigation functions must commute with each other, i.e. the same location must be reached if a set of navigation functions (say `++i.x; ++i.y`) is applied in a different order (like `++i.y; ++i.x`).

Since speed is of utmost concern in image processing, requiring constant execution time is still to vague in this domain. Therefore we specify, that navigation functions in x direction and equality check between iterators should not be slower than any other operation. Thus, these operations should be preferred in inner loops. In a typical array based implementation they should consist of just one addition resp. comparison. Furthermore, iterators shall be lightweight objects so that copying an iterator is a cheap constant time operation.

7.2 Image Boundaries

In contrast to a 1D sequence, which has just 2 ends, an image of size $w * h$ has $2 * (w + h + 2)$ past-the-border pixels. An iterator that refers to one of the past-the-border pixels will be called *boundary marker* (no new type is required). According to their position we distinguish *left, right, top*, and *bottom* boundary markers. To completely mark the boundaries of the $w * h$ image, we need a set of $2 * (w + h + 2)$ boundary markers - one for the begin and end of each row and each column. It is not desirable to pass a set of so many boundary markers to an algorithm. Therefore we specify rules how the algorithm can create any boundary marker from just a few on known positions.

All 2D iterators must be able to navigate on past-the-border pixels even if these pixels are not dereferencable. (Navigation outside the past-the-border range is undefined.) Thus we can transform an iterator into a boundary marker and a boundary marker into another boundary marker or back into an iterator. Top boundary markers are defined as follows:

- For each column a unique top boundary marker exists. A program may hold many identical instances of this marker.

Table 1: Required navigation functionality of an `ImageIterator`

Operation	Result	Semantics	Remark
`++i.x; i.x++`	`void` (`*i.x++` not allowed)	increment x coordinate	use this in inner loop
`--i.x; i.x--`	`void` (`*i.x--` not allowed)	decrement x coordinate	use this in inner loop
`i.x += dx`	`ImageIterator::AsX &`	add dx to x coordinate	use this in inner loop
`i.x -= dx`	`ImageIterator::AsX &`	subtract dx from x coordinate	use this in inner loop
`i.x - j.x`	`int`	difference of x coordinates	result is undefined if i and j refer to different images
`i.x = j.x`	`ImageIterator::AsX &`	`i.x += j.x - i.x`	result is undefined if i and j refer to different images
`i.x == j.x`	`bool`	`j.x - i.x == 0`	result is undefined if i and j refer to different images
`i.x < j.x`	`bool`	`j.x - i.x > 0`	result is undefined if i and j refer to different images
`++i.y; i.y++`	`void` (`*i.y++` not allowed)	increment y coordinate	
`--i.y; i.y--`	`void` (`*i.y--` not allowed)	decrement y coordinate	
`i.y += dy`	`ImageIterator::AsY &`	add dy to y coordinate	
`i.y -= dy`	`ImageIterator::AsY &`	subtract dy from y coordinate	
`i.y - j.y`	`int`	difference of y coordinates	result is undefined if i and j refer to different images
`i.y = j.y`	`ImageIterator::AsY &`	`i.y += j.y - i.y`	result is undefined if i and j refer to different images
`i.y == j.y`	`bool`	`j.y - i.y == 0`	result is undefined if i and j refer to different images
`i.y < j.y`	`bool`	`j.y - i.y > 0`	result is undefined if i and j refer to different images
`ImageIterator i(k)`		copy constructor	
`i = k`	`ImageIterator &`	copy assignment	
`i += s`	`ImageIterator &`	add offset to x and y	
`i -= s`	`ImageIterator &`	subtract offset from x and y	
`i + s`	`ImageIterator`	add offset to x and y	
`i - s`	`ImageIterator`	subtract offset from x and y	
`i - j`	`Size2D`	difference in x and y	result is undefined if i and j refer to different images
`i == j`	`bool`	`i.x == j.x && i.y == j.y`	use this in inner loop, result is undefined if i and j refer to different images
`typename ImageIterator::AsX`		type of member `i.x`	
`typename ImageIterator::AsY`		type of member `i.y`	

- Applying `++i.y` will move the iterator to the first pixel of the current column, a subsequent `--i.y` recreates the boundary marker. Likewise `i.y+=dy` may be applied if the target pixel is inside the image or past the bottom border ($dy \leq h + 1$). Subsequent `i.y-=dy` recreates the marker.

- Applying `++i.x`, `--i.x`, `i.x+=dx` and `i.x-=dx` produces the top boundary markers of the respective target columns, if they exist.

Analogous definitions apply to the other boundaries.

7.3 Accessing the Pixel Data

In general, the functionality to access pixel values follows the conventions of the STL. However, we must use `operator()` as index operator because `operator[]` is not defined for multiple dimensions. As in the STL we distinguish mutable iterators which return the pixel values by reference, and constant iterators which return them by value. The result of calling an access function at a past-the-border position is undefined.

For images having vector pixels (e.g. multi-spectral images) this convention is not very convenient, especially if the image is implemented as a multi-band format. In this case, the vector at each pixel (the PixelType) does not physically exist as a C++ data structure since the image is implemented as a vector of scalar images rather than an image of vectors. Therefore we provide a different set of access functions for vector images and their most common incarnation, RGB images. An implementation is free to provide both the general and the vector interfaces.

Table 2 summarizes the proposed access functionality, where `i` is an ImageIterator, dx, dy, and b are of type `int`.

7.4 Implementation

Of course, the implementation of the iterators depends on the underlying image data format. Nevertheless it's worthwhile to give some examples for common cases. Lets start with an implementation that can wrap an existing image format of which we do not know or can not access the internal representation. Suppose, the image type in question (let's call it `ProtectedImage`) defines a pixel access function `pixel(int x, int y)`:

```
class ProtectedImage {
  public:
      typedef PixelType;
      PixelType & pixel(int x, int y);
// ...
};
```

Table 2: Required pixel access functionality of an `ImageIterator`

Operation	Result	Semantics	Remark
General Access Functionality			
`typename ImageIterator::PixelType`		the iterators pixel type	
`*i`	`PixelType &`	access the pixel data	mutable iterator only
`*i`	`PixelType`	read the pixel data	constant iterator only
`i(dx, dy)`	`PixelType &`	access data at offset (dx, dy)	mutable iterator only
`i(dx, dy)`	`PixelType`	read data at offset (dx, dy)	constant iterator only
Special Access Functionality for Vector Pixels			
`typename ImageIterator::ValueType`		type of the values in the vectors	
`i[b]`	`ValueType &`	access band b of current vector	mutable iterator only
`i[b]`	`ValueType`	read band b of current vector	constant iterator only
`i(dx, dy, b)`	`ValueType &`	access band b of vector at offset (dx, dy)	mutable iterator only
`i(dx, dy, b)`	`ValueType`	read band b of vector at offset (dx, dy)	constant iterator only
Special Access Functionality for RGB Pixels (in addition to vector access functionality)			
`i.red()`	`ValueType &`	access red channel of current pixel	mutable iterator only
`i.red()`	`ValueType`	read red channel of current pixel	constant iterator only
`i.red(dx, dy)`	`ValueType &`	access red channel of pixel at offset (dx, dy)	mutable iterator only
`i.red(dx, dy)`	`ValueType`	read red channel of pixel at offset (dx, dy)	constant iterator only
green and blue analogously			

Then the iterator can use this function to access pixels indirectly. Performance may thus be suboptimal, but the implementation is very simple and therefore a good choice during rapid prototyping. We implement the `AsX` and `AsY` data members as `int`'s so that the required operators in x and y directions are automatically defined. Three functions are shown for illustration:

```
struct ProtectedImageIterator {
    typedef ProtectedImage::PixelType PixelType;
    typedef int AsX;
    typedef int AsY;

    ProtectedImageIterator(ProtectedImage * i)
    : image_(i),    // remember the image
      x(0), y(0)
    {}

    // compare iterators by comparing their coordinates
    bool operator==(ProtectedImageIterator const & i) const {
        return x == i.x && y == i.y;
    }

    // access pixel indirectly via ProtectedImage member fct
    PixelType & operator*() {
        return image_->pixel(x, y);
    }

    // ... implement other functions accordingly

    int x, y;  // the current coordinates of the iterator
  private:
    ProtectedImage * image_;  // the wrapped image
};
```

Another typical case is an image that internally allocates raw storage to hold the pixel values. The appendix shows a possible iterator implementation for this case, called `ImageIterator`. This iterator directly accesses the pixels via pointers and is thus very efficient. It is the standard implementation we use for our own image data types. Our images have member functions that encapsulate the construction of the iterators so that users of the library need not know about raw storage and iterator initialization. Calling the `minPixelValue()` function to be defined in the next section might look like this:

```
OurImageType image(width, height); // and fill in pixel values ...
```

```
// construct iterators
OurImageType::Iterator upperleft = image.upperLeft();
OurImageType::Iterator lowerright = image.lowerRight();

// find the minimum of the entire image
OurImageType::PixelType min = minPixelValue(upperleft,lowerright);

// find the minimum in a smaller ROI (excluding the border pixels)
min = minPixelValue(upperleft + Size2D(1, 1),
                    lowerright - Size2D(1, 1));
```

Since the ImageIterator implementation only assumes that the pixels are stored as a contiguous array, we can apply it to a plain old C data structure, which can't have member functions, as well. For example, if we want to call a generic algorithm for a KHOROS VIFF image, we simply initialize the iterator by hand:

```
struct xvimage * viff_image; // and fill in data ...

if(viff_image->data_storage_type == VFF_TYPE_FLOAT)
{
    // we have a float image, init float iterator at upper left
    ImageIterator<float> viff_upper_left(
           (float *)(viff_image->imagedata), // pointer to first pixel
           viff_image->row_size);            // width of the image

    // lower right is Size2D(width, height) apart
    ImageIterator<float> viff_lower_right(viff_upper_left +
              Size2D(viff_image->row_size, viff_image->col_size);

    float min = minPixelValue(viff_upper_left, viff_lower_right);
}
```

Irrespective of the concrete iterator implementation choosen, we can use all our generic algorithms as long as the iterator complies to the requiremnts listed in table 1.

8 Example Algorithms

Based on the iterator definitions in the previous section we now give some examples to illustrate different implementation techniques for 2 dimensional generic algorithms. The benefit of reusing simple algorithms like those below

might not be apparent, but keep in mind that the techniques scale to large algorithms were reuse really pays off.

In the sequel we follow the convention customary in C++ that `upper_left` points to the first pixel *inside* the region of interest, while `lower_right` points to the first pixel *outside* the region (i.e. the last pixel inside the region would be reached by (`--lower_right.x`, `--lower_right.y`)). Note also, that `upper_left` and `lower_right` may refer to arbitrary coordinates in the image so that the algorithm can be applied to any rectangular subregion without modification.

The first algorithm calculates the minimal pixel value in the ROI. It uses simple integer based loops and accesses the pixels via the index operator (`operator()(int x, int y)`):

```
template <class ImageIterator>
ImageIterator::PixelType
minPixelValue(ImageIterator upper_left, ImageIterator lower_right)
{
    int width = lower_right.x - upper_left.x;   // width and ...
    int height = lower_right.y - upper_left.y;  // height of the ROI
    ImageIterator::PixelType min = *upper_left; // init minimum

    for(int y=0; y<height; ++y)       // use integer coordinates ...
    {
        for(int x=0; x<width; ++x)    // ... in the loops
        {
            // access pixel via index operator
            if(min > *upper_left) min = upper_left(x, y);
        }
    }
    return min;
}
```

This variant is very clear and can immediately be understood. Due to the address calculation involved in accessing the pixels it may, however, perform suboptimally, especially if each pixel is accessed several times. The address calculation is eliminated in the next variant which uses the navigation functions of the iterators to control the loops. The algorithm calculates the sum of all pixel values in the ROI:

```
template <class ImageIterator, class SumType>
SumType
sumOfPixelValues(ImageIterator upper_left, ImageIterator lower_right,
                 SumType initial_sum)

{
```

```
    // do nothing if lower_right is above or left of upper_left
    if((lower_right.x - upper_left.x <= 0) &&
       (lower_right.y - upper_left.y <= 0))     return;

    ImageIterator y(upper_left);  // iterator at begin of ROI
    ImageIterator yend(y);        // boundary marker at ...
    yend.y = lower_right.y;       // end of first column of ROI

    // iterate down the first column using iterator directly
    for(; y != yend; ++y.y)
    {
        ImageIterator x(y);  // iterator at begin of current row
        ImageIterator xend(x);        // boundary marker at ...
        xend.x = lower_right.x;       // ... end of current row

        // iterate across current row using iterator directly
        for(; x != xend; ++x.x)
        {
            // access pixel via dereferencing operator
            initial_sum += *x;
        }
    }
    return initial_sum:
}
```

The next example is a little more complicated since it involves two images. The algorithm smoothes an image by averaging over a square window of arbitrary size (the size is given as `window_radius` in chess-board metric). For each source pixel, the average of all pixels in the surrounding window is calculated and the result written into the destination pixel. If the window is partially outside the image, it is clipped accordingly. Note that the traits technique [7] is used to determine the type of the variable to hold the sum of the pixel values.

```
template <class ImageIterator>
void average(ImageIterator dest_upper_left,
             ImageIterator source_upper_left,
             ImageIterator source_lower_right,
             int window_radius)
{
    // use traits to determine SumType as to prevent possible overflow
    typedef ImageIterator::PixelType PixelType;
    typedef numeric_traits<PixelType>::Tpromote SumType;

    // calculate width and height of the image
```

```
    int w = source_lower_right.x - source_upper_left.x + 1;
    int h = source_lower_right.y - source_upper_left.y + 1;

    // create y iterators
    ImageIterator yd(dest_upper_left);
    ImageIterator ys(source_upper_left);

    for(int y=0; y < h; ++y, ++ys.y, ++yd.y)
    {
        // create x iterators
        ImageIterator xd(yd);
        ImageIterator xs(ys);

        for(int x=0; x < w; ++x, ++xs.x, ++xd.x)
        {
            // how much of the window fits into the image ?
            int x0 = min(xs.x - source_upper_left.x, window_radius);
            int y0 = min(xs.y - source_upper_left.y, window_radius);
            int x1 = min(source_lower_right.x - xs.x, window_radius + 1);
            int y1 = min(source_lower_right.y - xs.y, window_radius + 1);

            int ww = x0 + x1 + 1;  // window width
            int wh = y0 + y1 + 1;  // window height

            // init the sum
            SumType sum = 0;

            // call the sumOfPixelValues() function for window
            sum = sumOfPixelValues(xs - Size2D(x0, y0),
                                   xs + Size2D(x1, y1), sum);

            // store average in destination pixel
            *xd = sum / (ww * wh);
        }
    }
}
```

This algorithm is no more complicated to write than a pointer based version and runs almost as fast.

9 Performance

To assess the claims about the performance of generic algorithms we conducted a number of benchmark tests. We tested 5 different implementations

Table 3: Benchmarks (averaging of 2000x1000 float image, 7x7 window)

	Algorithm Variant				
System	1	2	3	4	5
1	2.75 s (100%)	12.6 s (460%)	7.5 s (270%)	5.0 s (180%)	2.9 s (105%)
2	9.9 s (100%)	42.2 s (425%)	35.6 s (360%)	18.5 s (190%)	12.9 s (130%)
3	11.5 s (100%)	64.8 s (560%)	55.7 s (480%)	21.0 s (180%)	46.0 s (400%)
4	8.1 s (100%)	38.4 s (470%)	13.5 s (170%)	9.1 s (112%)	9.0 s (111%)
	System 1: SGI O2, IRIX 6.2, SGI C++ 7.1 System 2: SGI INDY, IRIX 5.3, SGI C++ 4.0 System 3: Sun SparcServer 1000, Solaris 5.5, Sun C++ 4.1 System 4: PC Pentium 90, Windows NT 4.0, Microsoft VC++ 4.0				

of the averaging algorithm for a 2000x1000 float image with window size 7x7 on various systems:

Variant 1: a hand-optimized pointer-based version

Variant 2: a traditional object-oriented variant using an abstract image interface with a virtual access function:

```
class AbstractImage {
   public:
     virtual float & pixel(int x, int y) = 0;
};
```

Variant 3: the averaging algorithm as written in the previous section using the `ProtectedImageIterator` from section 7.4

Variant 4: the averaging algorithm as written in the previous section using the `ImageIterator` from the appendix

Variant 5: the same algorithm as in variant 2, but using the `ImageIterator`'s `operator()(int x, int y)` instead of the virtual function of the abstract image (cf. function `minPixelValue()`)

The results of these tests are given in table 3. The table shows that with the best compilers the proposed `ImageIterator` (Variants 4 and 5) comes close to the performance of a hand-crafted algorithm (Variant 1), whereas the versions using the abstract image interface are much slower. Even in the worst case our standard iterator takes less then double the optimal time. In the light of the flexibility gained this should be acceptable for many applications. As compiler optimizers are permanently improved, we expect the numbers to further decrease during the next years, the more so as the C++ Standard Library itself has adopted the iterator concept.

10 Conclusions

The present paper introduced a generic programming approach to the development of reusable algorithms in image processing. It is based on ideas made popular by the C++ Standard Template Library and extends its abstract iterator design to the domain of 2-dimensional images. This approach overcomes a number of limitations of traditional ways to design reusable software in image processing:

- There is only a small performance penalty compared to an optimized version of the algorithm. No image conversion is necessary.
- Iterators can be implemented for existing data structures. Switching to generic programming does not influence any existing code.
- Implementing reusable algorithms is no more difficult than traditional programming. No complex inheritance hierarchies need to be designed.

Using the proposed approach, reuse of image processing algorithms should become much easier. Thus it is an ideal basis for intensified efforts in algorithm characterization and comparison.

We plan to continue our generic programming effort by defining iterators for higher level image analysis functionality, in particular different types of graph algorithms.

References

[1] "Image Processing and Interchange Standard", ISO/IEC Document 12087, 1994

[2] "Image Understanding Environment Documentation", Amerinex Applied Imaging Inc. 1996 (http://www.aai.com/AAI/IUE/IUE.html)

[3] "Khoros Documentation", Khoral Research Inc. 1996 (http://www.khoros.unm.edu/)

[4] D. Musser, A. Stepanov: "Generic Programming", 1st Intl. Joint Conf. of ISSAC-88 and AAEC-6, Springer LNCS 358, 1989

[5] D. Musser, A. Stepanov: "Algorithm-Oriented Generic Libraries", Software - Practice and Experience, Vol. 24(7), 623-642, 1994

[6] D. Musser, A. Saini: "STL Tutorial and Reference Guide", Addison-Wesley 1996

[7] N. Myers: "A New and Useful Template Technique: Traits", C++ Report, June 1995

[8] "The Programming Language C++" (C++-Standard, Commitee Draft), ISO/IEC Document CD 14882, 1996

Appendix

This appendix shows our standard iterator implementation which is applicable to any image format that stores the pixel data as an array of raw memory. The idea of this implementation is that the AsX and AsY members share one set of navigation data (which is achieved by making ImageIterator::y a reference to ImageIterator::x) but interpret it differently - AsX does never change the y coordinate and vice versa. The ImageIterator itself is a friend of both AsX and AsY, so that it can also access the data directly - see for example the implementation of `operator*()`. The operators intended for use in the inner loop (see table 1) need only one atomic operation which makes this iterator theoretically equally efficient than a pointer (in praxis, a small speed penalty is observed because current C++ compilers optimize the iterator less well than a pointer). Due to space limitations not every line could be commented, but the implementation should be easily understandable for most C++ programmers.

```
template <class PIXELTYPE>
class ImageIterator
{
  public:

    typedef PIXELTYPE PixelType;

    // Let navigation operations act in Y direction
    class AsY
    {
      public:
        // construct fom raw memory and width
        AsY(PIXELTYPE * base, int width)
        : pixel_(base), line_(base), offset_(width)
        {}

        // assign y coordinate of rhs, let x unchanged
        AsY & operator=(AsY const & rhs) {
            if(this != &rhs)
            {
                int dy = operator-(rhs);
```

```
            operator-=(dy);
        }
        return *this;
    }

    // increment and decrement y
    void operator++() {
        pixel_ += offset_; line_ += offset_;
    }
    void operator++(int) {
        pixel_ += offset_; line_ += offset_;
    }
    void operator--() {
        pixel_ -= offset_; line_ -= offset_;
    }
    void operator--(int) {
        pixel_ -= offset_; line_ -= offset_;
    }

    // random offset in y direction
    AsY & operator+=(int dy) {
        pixel_ += dy * offset_; line_ +=  dy * offset_;
        return *this;
    }
    AsY & operator-=(int dy) {
        pixel_ -= dy * offset_; line_ -=  dy * offset_;
        return *this;
    }

    // comparision of y coordinates
    bool operator==(const AsY & rhs) const {
        return (line_  == rhs.line_);
    }
    bool operator<(const AsY & rhs) const {
        return (line_  < rhs.line_);
    }

    // Difference of y Coordinates
    long operator-(const AsY & rhs) const {
        return (line_ - rhs.line_) / offset_;
    }

protected:

    friend class ImageIterator<PIXELTYPE>;
```

```
    AsY(AsY const & rhs)
    : pixel_(rhs.pixel_), line_(rhs.line_), offset_(rhs.offset_)
    {}

    PIXELTYPE * line_, * pixel_;
    long offset_;
};

// Let navigation operations act in X direction
class AsX : public AsY
{
  public:
    // construct from raw memory and width
    AsX(PIXELTYPE * base, int width)
    : AsY(base, width)
    {}

    // assign x coordinate of rhs, let y unchanged
    AsX & operator=(AsX const & rhs) {
        if(this != &rhs)
            pixel_ = line_ + (rhs.line_ - rhs.pixel_);
        return *this;
    }

    // increment and decrement x
    void operator++() {
        ++pixel_;
    }
    void operator++(int) {
        ++pixel_;
    }
    void operator--() {
        --pixel_;
    }
    void operator--(int) {
        --pixel_;
    }

    // random offset in x direction
    AsX & operator+=(int dx) {
        pixel_ += dx;
        return *this;
    }
    AsX & operator-=(int dx) {
```

```
        pixel_ -= dx;
        return *this;
    }

    // Comparision of x Coordinates
    bool operator==(const AsX & rhs) const {
        return (pixel_ - line_) == (rhs.pixel_ - rhs.line_);
    }
    bool operator<(const AsX & rhs) const {
        return (pixel_ - line_) < (rhs.pixel_ - rhs.line_);
    }

    // Difference of x Coordinates
    long operator-(const AsX & rhs) const {
        return (pixel_ - line_) - (rhs.pixel_ - rhs.line_);
    }

  protected:
    friend class ImageIterator<PIXELTYPE>;

    AsX(AsX const & rhs)
    : AsY(rhs)
    {}
};


// Construct form raw memory and width
ImageIterator(PIXELTYPE * base, int width)
: x(base, width), y(x)
{}

// Copy constructor
ImageIterator(const ImageIterator & rhs)
: x(rhs.x), y(x)
{}

// Copy assignment
ImageIterator & operator=(const ImageIterator & rhs) {
    if(this != &rhs)
    {
        x.line_ = rhs.x.line_;
        x.pixel_ = rhs.x.pixel_;
        x.offset_ = rhs.x.offset_;
    }
    return *this;
```

```
}

// Add and subtract random offset in x and y via Size2D object
ImageIterator & operator+=(Size2D const & s) {
    x += s.width;
    y += s.height;
    return *this;
}
ImageIterator & operator-=(Size2D const & s) {
    x -= s.width;
    y -= s.height;
    return *this;
}

// Create new iterator at offset in x and y via Size2D object
ImageIterator operator+(Size2D const & s) const {
    ImageIterator ret(*this);
    ret += s;
    return ret;
}
ImageIterator operator-(Size2D const & s) const {
    ImageIterator ret(*this);
    ret -= s;
    return ret;
}

// Comparision of Iterators
bool operator==(const ImageIterator & rhs) const {
    return x.pixel_ == rhs.x.pixel_;
}
bool operator!=(const ImageIterator & rhs) const {
    return x.pixel_ != rhs.x.pixel_;
}

PixelType & operator*() {
    return *(x.pixel_);
}

PIXELTYPE & operator()(int dx, int dy) {
    return *(x.pixel_ + x.offset_*dy + dx);
}


// refer to x coordinate
AsX x;
```

```
    // refer to y coordinate
    AsY & y;
};

// subtract two iterators
template <class PixelType>
Size2D
operator-(ImageIterator<PixelType> i, ImageIterator<PixelType> j) {
    return Size2D(i.x - j.x, i.y - j.y);
}
```