STL-Style Generic Programming with Images

Ullrich Köthe
Fraunhofer-Institute for Computer Graphics
Joachim-Jungius-Str. 9, D-18059 Rostock, Germany
Email: koethe@egd.igd.fhg.de, Phone: +49-381-4024-114, Fax: -199

Introduction

Generic Programming has been introduced as a powerful mechanism to implement reusable algorithms that are intended to run in many different application contexts [MS89, MS94]. The Standard Template Library (STL) has impressively demonstrated this for fundamental algorithms such as sorting and searching [MS96] and has been incorporated into the official C++ standard [C++98]. However, the need for algorithm implementations that are independent of a specific application framework is not restricted to the fields covered by the STL.
In this paper we explore the problem as it is applied to the field of image processing. Image processing algorithms are applied in application areas as diverse as robotics, medical imaging, and video processing. In fact, most applications that deal with pixel graphics involve some form of image manipulation, for example resizing, and calculation of a color map. Therefore, reusable image processing algorithms would be very useful.
In the STL, reusability is achieved by three key design techniques: *Iterators* serve to de-couple algorithms from their underlying data structures by encapsulating the navigation and access functionality, *functors* allow to flexibly exchange parts of the computation, and *generic algorithms* implement an algorithm in terms of abstract iterators and functors. By using the template mechanism, generic algorithms can be instantiated for any combination of iterators and functors that conform to certain interface requirements.
To give a concrete example that leads the way to image processing, suppose we want to transform a sequence of RGB values into gray values. A RGB value is defined like this:

```
struct RGBValue {
    unsigned char red, green, blue;
};
```

Gray values are represented by `unsigned char`. A functor that performs the transformation by calculating a weighted average of the colors is implemented as follows:

```
struct RGBToGray {
   unsigned char
   operator()(RGBValue const & rgb) const {
        return 0.3*rgb.red + 0.59*rgb.green
                           + 0.11*rgb.blue;
   }
};
```

If the RGB and gray values are stored in vectors, we can apply the STL `transform()` algorithm to perform the transformation:

```
vector<RGBValue>      rgbsignal;
vector<unsigned char> graysignal;
//...

transform(rgbsignal.begin(),rgbsignal.end(),
          graysignal.begin(), RGBToGray());
```

However, in practice RGB values are usually stored in two-dimensional images rather than one-dimensional vectors. STL iterators and algorithms are not directly applicable to two-dimensional data structures.
Therefore, in the sequel we will extend the iterator philosophy of the STL to two dimensions and adapt the algorithm/functor concept to the needs of image processing. Note, however, that for the sake of clarity some of the implementation examples are sub-optimal in terms of performance.

Two-Dimensional Image Iterators

Navigation and Access

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

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

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

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

    MoveX x;  // x-view to navigation data
    MoveY y;  // y-view to navigation data
};
```

Now we can use the nested objects `x` and `y` to specify the direction to move along as if we had two one-dimensional iterators:

```
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 the STL. Since images are usually random access data structures[1] we do not need categories reflecting different navigation capabilities. The functionality to access pixel values also follows the conventions of the STL. For 2-D distance calculations and index operations we need a two-dimensional distance type defined like this:

```
struct Dist2D {
    Dist2D(int w, int h)
    : width(w), height(h)
    {}

    int width, height;
};

// create new iterator at distance (5, 19)
ImageIterator j = i + Dist2D(5,19);
```

As in the STL we distinguish mutable iterators that return the pixel values by reference, and constant iterators which return them by value.

Table 1 lists the required `ImageIterator` functionality. 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 too vague in this domain. Therefore we require that some navigation functions in x direction and equality check between iterators should not be slower than any other operation. These operations should be preferred in inner loops. In a typical array based implementation they should consist of just one addition respective comparison.

Image Boundaries

In contrast to a 1-D sequence, which has just 2 ends, an image of size $w*h$ has $2*(w+h+2)$ past-the-border pixels[2]. To completely mark the boundaries of this image, we need a set of iterators - one for the beginning and end of each row and column. Iterators that refer to past-the-border pixels will be called *boundary markers* (this generalizes the STL notion of past-the-end to two dimensions). According to their position we distinguish *left, right, top*, and *bottom* boundary markers. It is not desirable to pass a set of $2*(w+h+2)$ boundary markers to an algorithm. Therefore we specify rules how algorithms 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 dereferenceable (that is, we must not call `*iterator` there). Thus, we can transform an iterator into a boundary marker and a boundary marker into another one. Top boundary markers are defined as follows:

- Each column has a unique top boundary marker. (Of course, a program may hold many identical instances of this marker.)
- If iterator `i` marks the top boundary and you call `++i.y`, it is moved to the first pixel of the current column. A subsequent call of `--i.y` recreates the marker. Likewise, `i.y+=dy` may be applied if the target pixel is inside the image or past the bottom border ($dy \le h+1$). Subsequent `i.y-=dy` recreates the marker.
- If you call `++i.x`, `--i.x`, `i.x+=dx` or `i.x-=dx` on a top boundary marker, the top boundary markers of the respective target columns are created (if they exist).

Analogous definitions apply to the other boundaries. Examples for generic algorithms implemented on the basis of these definitions are given in section “Algorithms and Functors” below.

Implementation

Of course, the implementation of the iterators depends on the underlying image data format. Nevertheless it is worthwhile to discuss different options. Lets start with a simple implementation that can wrap an existing image format. Suppose, we want to implement an image iterator for an `AbstractGrayImage` defined like this:

```
class AbstractGrayImage {
  public:
    virtual unsigned char &
            pixel(int x, int y) = 0;
};
```

[1] Image implementations that do not allow random access to the data exist (e.g. regular grids represented as graphs), but are typically used with a different set of algorithms, so we do not consider them here.

[2] Past-the-border pixels are the outside pixels having at least one vertical, horizontal, or diagonal neighbor in the image.

Embedded Types	Semantics
`typename ImageIterator::value_type`	the iterator's pixel type
`typename ImageIterator::MoveX`	type of the x navigator
`typename ImageIterator::MoveY`	type of the y navigator

Operation	Result	Semantics
`++i.x; i.x++`	`void`	increment x coordinate (`*i.x++` not allowed)
`--i.x; i.x--`	`void`	decrement x coordinate (`*i.x--` not allowed)
`i.x += dx`	`ImageIterator::MoveX &`	add dx to x coordinate
`i.x -= dx`	`ImageIterator::MoveX &`	subtract dx from x coordinate
`i.x - j.x`	`int`	difference of x coordinates
`i.x = j.x`	`ImageIterator::MoveX &`	`i.x += j.x - i.x`
`i.x == j.x`	`bool`	`0 == j.x - i.x`
`i.x < j.x`	`bool`	`0 < j.x - i.x`
likewise for y-direction (`++i.y`, `i.y == j.y`, etc.)		
`ImageIterator i(k)`		copy constructor
`i = k`	`ImageIterator &`	copy assignment
`i += dist`	`ImageIterator &`	`(i.x += dist.width, i.y += dist.height)`
`i -= dist`	`ImageIterator &`	`(i.x -= dist.width, i.y -= dist.height)`
`i + dist`	`ImageIterator`	`{ImageIterator k(i); k += dist; return k;}`
`i - dist`	`ImageIterator`	`{ImageIterator k(i); k -= dist; return k;}`
`i - j`	`Dist2D`	`{Dist2D d(i.x-j.x, i.y-j.y); return d;}`
`i == j`	`bool`	`i.x == j.x && i.y == j.y`
`*i`	`value_type &`	access the pixel data (mutable iterator)
`*i`	`value_type`	read the pixel data (constant iterator)
`i[dist]`	`value_type &`	`*(i+dist)` (access data at offset `dist`, mutable iterator)
`i[dist]`	`value_type`	`*(i+dist)` (read data at offset `dist`, constant iterator)

Notation:
```
ImageIterator i, j; // must refer to the same image
int dx, dy;
Dist2D dist;
```

Table 1: Required functionality of an `ImageIterator *`

If we cannot modify the source code of this class, we can instead encapsulate a call to the virtual function `pixel()` in the iterator's `operator*()`:

```
struct AbstractGrayImageIterator {
    typedef unsigned char value_type;
    typedef int MoveX;
    typedef int MoveY;

    AbstractGrayImageIterator(
                  AbstractGrayImage * i)
: image_(i), x(0), y(0)
{}

bool operator==(
 AbstractGrayImageIterator const & i) {
    return x == i.x && y == i.y;
}

value_type & operator*() {
    return image_->pixel(x, y);
}
```

```
    // etc.

    int x, y;
  private:
    AbstractGrayImage * image_;
};
```

Of course, we can provide a much more efficient implementation if we have access to the internal representation of the image. Typically, images internally allocate an array of raw storage to hold the pixel values. Listings 1 shows an example iterator implementation that directly accesses the raw storage – many more possibilities exist. Although our implementation performs an extra address calculation within `operator*()`, it is very fast, because many compilers are able to highly optimize this code. You should perform some benchmarks with your compiler before you choose any particular image iterator implementation.

Existing Code is not Broken

Note that implementing the iterators for an existing data structure does not break any existing code. Only a simple extension in the interface of the images is required. For example, we can define factory functions `upperLeft()` and `lowerRight()` to create the corresponding iterators analogously to the `begin()` and `end()` functions in the STL. Thus, in an existing project, one can introduce generic programming at any point without the need to reimplement a huge code base. It is even possible to use the technique if the image data structures are simple C structs or raw memory, rather than a complete C++ class – the iterator definition does not require the images to be C++ objects. For example, we could construct the iterator shown in listing 1 like this:

```
const int width = 100, height = 200;
int imagedata[width*height];

ImageIterator<int>
  upperleft(imagedata, width),
  lowerright = upperleft +
                   Dist2D(width,height);

// now execute an algorithm
...
```

Listing 1: possible implementation of ImageIterator

```
template <class PIXELDATA>
class ImageIterator
{
   typedef PIXELTYPE    value_type;
   typedef value_type * MoveX;

   struct MoveY {
     MoveY(int width)
     : width_(width), offset_(0)
     {}

     void operator++() {
        offset_ += width_;
     }

     bool
     operator==(MoveY const & o)const{
          return offset_ == o.offset_;
     }
     // etc.

     int offset_, width_;
   };

   MoveX x;
   MoveY y;

   ImageIterator(value_type * data,
                              int width)
   : x(data), y(width)
   {}

   ImageIterator &
   operator+=(Dist2D d) {
      x += d.width; y += d.height;
      return *this;
   }

   bool operator==(
      ImageIterator const & i) const {
      return (x == i.x) && (y == i.y);
   }

   value_type & operator*() {
     return *(x + y.offset_);
   }

   // etc.
};
```

Algorithms and Functors

The basic principles of generic image algorithms are also adopted from the STL. Algorithms are templates of iterator types, and a pair of iterators, called `upper_left` and `lower_right`, determines the range of iteration. The algorithms assume that `upper_left` corresponds to the first pixel inside the range, while `lower_right` denotes the first pixel *outside* the range (one pixel right and below of the last pixel inside the range). Hereby we extend the standard convention of C and the STL towards two dimensions. Note, that `upper_left` and `lower_right` may refer to arbitrary positions in the image, so that any algorithm can be applied to any rectangular sub-region of an image without modification.

Like in the STL, functors are used to make the computation in the inner loop exchangeable. An algorithm that transforms one

image into another (using a functor to determine the transformation) could be implemented like this:

```
template <class ImageIterator1,
          class ImageIterator2, class Functor>
void
transformImage(ImageIterator1 src_upperleft,
               ImageIterator1 src_lowerright,
               ImageIterator2 dest_upperleft,
               Functor const & f)
{
    int width = src_lowerright.x -
                               src_upperleft.x;
    int height = src_lowerright.y -
                               src_upperleft.y;

    if((width <= 0) || (height <= 0)) return;

    // create y iterators
    ImageIterator1 ys(src_upperleft);
    ImageIterator1 srcend(src_lowerright);
    ImageIterator2 yd(dest_upperleft);

    // go down first column
    for(; ys.y != srcend.y; ++ys.y, ++yd.y)
    {
        // create x iterators
        ImageIterator1 xs(ys);
        ImageIterator2 xd(yd);

        // go accros current row
        for(; xs.x != srcend.x; ++xs.x,++xd.x)
        {
           // transform  present pixel
            *xd = f(*xs);
        }
    }
}
```

This implementation is the two-dimensional analogue of the one-dimensional `transform()` function of the STL. We can now calculate gray levels from color values using ImageIterators instead of STL forward iterators:

```
transformImage(rgbimage->upperLeft(),
               rgbimage->lowerRight(),
               grayimage->upperLeft(),
               RGBToGray());
```

The advantage of using image iterators is that we can apply this algorithm, without modification, to an arbitrary rectangular subregion by simply moving the iterators inwards. Suppose the gray image's size is only 100x100, while the RGB image is larger. So we only transform a 100x100 rectangle in the upper left area of the RGB image:

```
transformImage(rgbimage->upperLeft(),
   rgbimage->upperLeft() + Dist2D(100,100),
   small_grayimage->upperLeft(),
   RGBToGray());
```

In addition, the function `copyImage()` is provided as a convenient abbreviation of the trivial transform that copies the pixels unchanged (except for type conversions). Similarly, we implement the functions `inspectImage()` to scan a single image (analogous to the STL `for_each()` function) and `combineTwoImages()` to perform a transformation that depends on two input images (analogous to the STL binary `transform()` algorithm). The interfaces of these functions look like this:

```
template <class ImageIterator1,
          class ImageIterator2>
void
copyImage(ImageIterator1 src_upperleft,
          ImageIterator1 src_lowerright,
          ImageIterator2 dest_upperleft);

template <class ImageIterator,
          class Functor>
void
inspectImage(ImageIterator upperleft,
             ImageIterator lowerright,
             Functor & f);

template <class SrcIterator1,
          class SrcIterator2,
          class DestIterator,
          class Functor>
void
combineTwoImages(SrcIterator1 upperleft1,
                 SrcIterator1 lowerright1,
                 SrcIterator2 upperleft2,
                 DestIterator upperleftd,
                 Functor f);
```

`inspectImage()` can, for example, be used to find the minimum and maximum gray levels in an image, or to calculate a histogram, while `combineTwoImages()` is typically used to calculate the (pixelwise) sum, difference etc. of two images (the STL functors `plus`, `minus` etc. can be used here).

To enable operation on *arbitrary* shaped regions-of-interest (ROI), we add corresponding functions that utilize a mask image to determine the ROI. A predicate functor is used to encapsulate the evaluation of the mask. Similar to the STL, this group of functions is marked by the suffix `If`. The function `inspectImageIf()` looks like this (note that the functor is given by reference so that it serves as a return value):

```
template <class ImageIterator,
          class MaskIterator,
          class Functor, class Predicate>
void inspectImageIf(ImageIterator upperleft,
                    ImageIterator lowerright,
                    MaskIterator mask,
                    Functor & f, Predicate p)
{
    int width = lowerright.x - upperleft.x;
    int height = lowerright.y - upperleft.y;

    // create y iterators
    ImageIterator yi(upperleft);
```

```
    MaskIterator ym(mask);

    // go down first column
    for(int y=0; y<height;
                      ++y, ++yi.y, ++ym.y)
    {
        // create x iterators
        ImageIterator xi(yi);
        MaskIterator xm(ym);

        // go accross current row
        for(int x=0; x<width;
                      ++x, ++xi.x, ++xm.x)
        {
           // aggregate statistics in functor
           // only if mask predicate is true
           if(p(*xm)) f(*xi);
        }
    }
}
```

By implementing these function templates and about 15 commonly used functor templates (including those already provided by the STL, e.g. `plus` and `minus`), we can provide a basic set of image processing functionality in roughly 10 Kbytes of source code. In a traditional system such as KHOROS[3] more than 100 Kbytes are needed to implement the same functionality, because each function must be implemented repeatedly for different image `value_types`.

Neighborhoods

Many image processing tasks involve a neighborhood or a window around the current pixel. This is another important reason for the introduction of truly two-dimensional iterators: finding the neighbors of a given pixel is much easier with two-dimensional iterators.

For example, to smooth an image and reduce noise, we can replace each pixel value with a (weighted) average of the gray values or colors in its neighborhood. Image iterators allow us to look at the neighborhood window as if it were a small image in itself.

A simple implementation of a smoothing algorithm based on this idea is given in listing 2. A window of the given size is successively placed over each pixel (the two outer loops). All pixels currently in the window are used to calculate an average which replaces the value of the window's center pixel (the two inner loops). If we are near the image border, we clip the window as to avoid accessing pixel values outside the image. Note that the iterators encapsulate all address calculations which are so difficult to get right in traditional, pointer-based implementations of the same algorithm.

Listing 2: generic implementation of smoothing algorithm

```
template <class ImageIterator1,
          class ImageIterator2>
void
smooth(ImageIterator1 source_upper_left,
       ImageIterator1 source_lower_right,
       ImageIterator2 dest_upper_left,
       int window_radius)
{
    // use traits to determine SumType as
    // to prevent possible overflow
    typedef typename
      ImageIterator1::value_type SrcType;
    typedef typename
     numeric_traits<SrcType>::Promote
     SumType;

    // calculate width and height of image
    int width = source_lower_right.x -
                        source_upper_left.x;
    int height = source_lower_right.y -
                        source_upper_left.y;

    // create y iterators
    ImageIterator1 ys(source_upper_left);
    ImageIterator2 yd(dest_upper_left);

    // down first column
    for(int y=0; y < height;
                    ++y, ++ys.y, ++yd.y)
    {
      // create x iterators
      ImageIterator1 xs(ys);
      ImageIterator2 xd(yd);

      // accross current row
      for(int x=0; x < width;
                    ++x, ++xs.x, ++xd.x)
      {
         // clip the window
         int x0 = min(x, window_radius);
         int y0 = min(y, window_radius);
         int winwidth  = min(width - x,
                   window_radius + 1) + x0;
         int winheight = min(height - y,
                   window_radius + 1) + y0;

         // create y and end iterators for
         // the clipped window
         ImageIterator1
              yw(xs - Dist2D(x0, y0));
         ImageIterator1 wend(yw +
              Dist2D(winwidth, winheight));

         // init the sum
         SumType sum = 0;
         // go down first column of window
         for(; yw.y != wend.y; ++yw.y)
         {
            // create x iterator for
            // current row in window
            ImageIterator1 xw(yw);

            // go accross row in window
            for(; xw.x != wend.x; ++xw.x)
            {
```

[3] See http://www.khoral.com/

```
                sum += *xw; // sum values
            }
        }

        // store average in destination
        *xd =
             sum / (winwidth * winheight);
    }
  }
}
```

Note also that a traits class [Meyers95] is used to determine the type of the intermediate variable `sum`. This is necessary to avoid overflow during the calculation of the sum of the values in the window: if the pixel type is `unsigned char` we should better store intermediate results in an `int`. The local type `numeric_traits::Promote` encodes the relevant type promotion rules. It is specialized for every pixel type, for example:

```
template <>
struct numeric_traits<unsigned char> {
    typedef int Promote;
};
```

Performance

To assess the claims about the performance of generic algorithms we conducted a number of benchmark tests. We tested 5 different implementations of the smoothing algorithm for a 2000x1000 float image with window size 7x7 on various systems. The image iterator based variants use the implementation shown in listing 2, the other variants are implemented analogously. These are the variants tested:

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, such as `AbstractGrayImage::pixel(x,y)` from subsection “Implementation”.

Variant 3: the smoothing algorithm as written in listing 2 using the `AbstractGrayImageIterator` from subsection “Implementation”.

Variant 4: the smoothing algorithm as written in listing 2 using the `ImageIterator` from listing 1.

Variant 5: the same algorithm as in variant 2, but using `ImageIterator::operator[](Dist2D)` instead of the virtual function of the abstract image.

The results of these tests are given in table 2. 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 (virtual function calls) are much slower (variants 2 and 3; this is mostly due to the lost optimization possibilities caused by the function call). Even in the worst case `ImageIterator` takes less than double the optimal time. In the light of the flexibility gained this should be acceptable for many applications. Moreover, there is no principal reason that an image iterator must be slower than the handcrafted version. Thus, with the continued improvement of compiler optimizers, we expect the performance of our iterator to increase further (note the better performance of the more modern compilers).

	Algorithm Variant				
System	1	2	3	4	5
1	2.75 s (100%)	12.6 s (460%)	7.5 s (270%)	3.3 s (120%)	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%)	17.0 s (150%)	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 6.3, SGI C++ 7.2 System 2: SGI INDY, IRIX 5.3, SGI C++ 4.0 (outdated compiler) System 3: Sun SparcServer 1000, Solaris 5.5, Sun C++ 4.2 (outdated compiler) System 4: PC Pentium 90, Windows NT 4.0, Microsoft VC++ 5.0				

Table 2: Performance measurements for different implementations of the averaging algorithm

Conclusions

The present paper introduced a generic programming approach to the development of two-dimensional, reusable algorithms in image processing. It is based upon the iterator design in the C++ Standard Template Library, extending the abstract iterator design to the domain of two-dimensional images. This approach overcomes a number of limitations of traditional approaches to designing reusable software:

- Programming is more efficient than with traditional methods since algorithms need not be repeated for different image data formats.
- There is only a small performance penalty compared to an optimized version of the algorithm. No image conversion is necessary.
- A smooth, incremental transition to generic programming is possible. Existing code is not broken while new iterators are incrementally implemented and adopted for existing data structures.
- Implementing reusable algorithms is no more difficult than traditional programming. No complex inheritance hierarchies need to be designed.

Although it might seem as if a lot of different iterators implementations are needed to cover all image types, our experience shows that a single iterator template suffices to implement iterators for many array based image data types.
Using the concepts described in this paper, reuse of image processing algorithms should become much easier. It might even be highly desirable to officially standardize a common set of iterators as to prevent any compatibility problems between implementations that may otherwise occur.
Many more advanced concepts can be built on top of the proposed image iterators, for example iterator adapters that select a linear substructure from an image (such as a line, a circle, or a spline). These concepts might be treated in detail in a follow-up paper.
The ideas put forward in this paper are realized in the image analysis library VIGRA, which is freely available on the WWW at “http://www.egd.igd.fhg.de/~ulli/vigra/”.

References

[C++98] *"The Programming Language C++"* (C++-Standard), ISO/IEC Document 14882, 1998

[KW97] D. Kühl, K. Weihe: *“Data Access Templates”*, C++-Report July/August 1997

[MS89] D. Musser, A. Stepanov: *"Generic Programming"*, 1st Intl. Joint Conf. of ISSAC-88 and AAEC-6, Springer LNCS 358, 1989

[MS94] D. Musser, A. Stepanov: *"Algorithm-Oriented Generic Libraries"*, Software - Practice and Experience, Vol. 24(7), 623-642, 1994

[MS96] D. Musser, A. Saini: *"STL Tutorial and Reference Guide"*, Addison-Wesley 1996

[Myers95] N. Myers: *"A New and Useful Template Technique: Traits"*, C++ Report, June 1995