Image processing with C / C++

Marc Schlipsing* and Jan Salmen†
Research group: Real-time Computer Vision (RTCV)
Institut für Neuroinformatik, Ruhr-Universität Bochum

version 2
November 15, 2017

Preface

This tutorial is directed towards students that are not familiar with image processing or C / C++. Part I introduces some *basic images processing techniques*, it explains how to handle digital images and perform basic operations. Part II covers *advanced programming teqchniques* related to the programming language C++. The two parts do not depend on each other.

For both parts of the tutorial, example code is provided and explained in detail. Additional exercises are left for you so that you can apply practically what you have learned and deepen your knowledge.

Additional image processing and computer vision techniques can be found in lecture notes [Win09]. For any further information concerning programming techniques we refer to [Cli, BBRS].

If you have any comments or suggestions concerning this tutorial, do not hesitate to contact us!

*marc.schlipsing@ini.rub.de
†jan.salmen@ini.rub.de

Contents

Part I.
Basic techniques

1. Introduction

The first part of our tutorial will follow the example code from a demo program available with this script. Once a new issue comes up in the code it will be addressed, so that this document is ordered by the image operations implemented in the code and not by C++ programming topics.

In order to make sure you learn anything new from this tutorial you might want to have a look at the example code first (see section 9). This code will be discussed chunk by chunk in the oncoming sections.

2. Image representation in memory and function calls

In general, two-dimensional images are stored in a linear memory buffer, so that all image rows are placed in memory consecutively. Depending on the size of the image and the bit-resolution for each pixel one will need to allocate an adequate amount of memory. The position of the image in memory is stored in a pointer which refers to the first pixel. In the case of an 8-bit gray-value image one may use a pointer of the unsigned 8-bit type. Beginning with the `main()`, in line 64 we create a pointer and initialize it with `0`.

```
unsigned char * pImage = 0;
```

We call the previously defined function

```
bool readPGM( const char * fileName, unsigned char ** ppData, int & width, int & height )
```

to receive an image from a .pgm-file. Given a file name, the method will try opening the file, allocate the right amount of memory, return the image position by setting our pointer `pImage` and the size parameters. This call shows two ways in which a function can write on the input parameters. For the memory position we pass the address to our pointer. The sizes `width` and `height` are handed over *by-reference*. In both cases one can write the outside variable from inside the function.

```
int width, height;
bool readOk = readPGM( "eyes_dark.pgm", &pImage, width, height );
```

You might want to take a look at the implementation of this function in `PpmIO.h`. It uses `realloc()` (similar to `malloc()`) to adjust the amount of memory allocated at `pImage`. Passing the address of the pointer enables the function to write the memory location to `pImage`, otherwise it would only receive a copy of the pointer.

If we want to avoid writing external parameters from within a function one could either use a constant reference (`const int & width`) that does not allow writing or a *call-by-value* (`int width`) where the parameter is given as a local copy. Be aware that the last case may produce unintended overhead, especially when passing complex objects (e. g. vectors, lists). If you do not have special requirements, use constant reference for parameters of complex objects (not built-in types).

After checking the return value we can start working with the image data. If the image was not read properly we report to the console and exit. The *boolean* expression `( ! readOk )` is the same as `( false == readOk )`.

```
if( ! readOk )
{
    printf( "Reading image failed!\n");
    printf( "Press any key to exit.\n" );
    getchar();
    return -1;
}
```

The image we work with in this example is shown in figure 1.

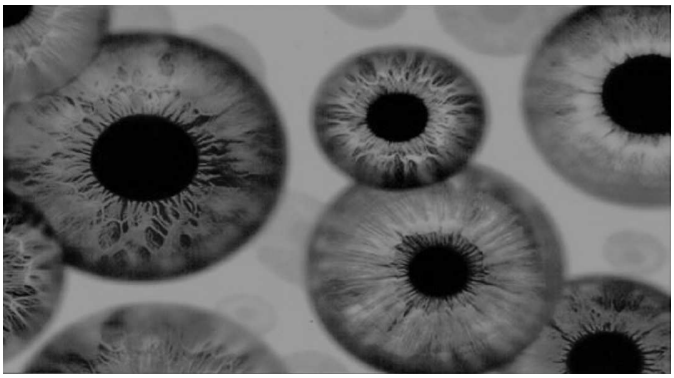


Figure 1: The image loaded from `eyes_dark.pgm`.

3. Accessing an image

How to access and manipulate image pixels is demonstrated by an easy example of scaling down our image to the half width and half height of the original. Therefor we pre-compute the new image sizes `widthScl` and `heightScl`.

```
const int widthScl = width / 2;
const int heightScl = height / 2;
```

There are two interesting things here: First, we declare these two new variables to be `const` in order to make clear we do not want the value to be changed, which is ensured by the compiler. Using `const` for constants makes your code easier to read and understand by others and will avoid errors. Second, the operator `/` represents an *integer division*

here because all values involved are of type `int`. The result of an integer division `a/b` is $\lfloor a/b \rfloor$, for instance, $1/2 = 0$. So be careful! The remainder can be received from `a%b`.

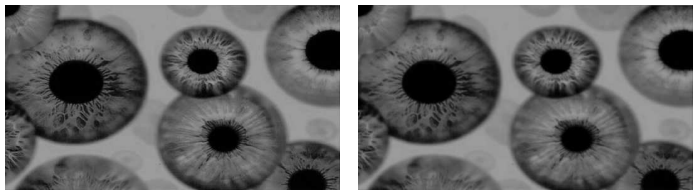
As we calculated the size for the scaled image, we can now define a new pointer variable and directly allocate memory using `new` (line 83).

```
unsigned char * pScaledImage = new unsigned char[ widthScl * heightScl ];
```

The scaled image is calculated by simply copying every second pixel from every second row of the original image to the new one. This is the easiest and fastest possible way to do the desired scaling. Although it does not yield the best possible results, it allows to show nicely how image pixels are accessed.

```
for( int y = 0; y < heightScl; y++ )
{
    for( int x = 0; x < widthScl; x++ )
    {
        pScaledImage[ x + y*widthScl ] = pImage[ 2*x + 2*y*width ];
    }
}
```

We see that a *nested* `for` loop is used to process the images. We iterate over the pixels of our new scaled image row by row (outer loop) and pixel by pixel inside each row (inner loop). As mentioned in section 2, the image is stored in a linear memory buffer. Therefore, the pixel at position (x, y) is located at the linear position $x + y \cdot widthScl$. This *index* calculated in line 89 can be used to address the pixel relative to `pScaledImage` (pointing at the beginning of the image) using the `[]` operator. To illustrate this further: `pScaledImage[0]` is the value of the first pixel (which is an `unsigned char` in this example), `pScaledImage[1]` the second, and `pScaledImage[100]` the 101st pixel value. You could even use negative indices inside the brackets to access values prior to a pointer. Nevertheless, you have to ensure that the memory you are reading or writing was allocated for you. In this example, we can safely use values from `pScaledImage[0]` ($x = 0$, $y = 0$, pixel in the upper left corner) up to `pScaledImage[widthScl * heightScl - 1]` ($x = widthScl - 1$, $y = heightScl - 1$, pixel in the lower right corner). In order to do the desired rescaling, we access every second pixel in the original image in a similar way.



(a) (b)

Figure 2: (a) Image scaled by 0.5 (b) after smoothing filter.

4. Filtering

The scaled image may look hard-edged or grainy. We can make use of a popular *smoothing filter* to improve this: The *Gaussian* filter performs a *convolution* using weights approximating a 2-dimensional Gaussian (*bell-shaped*) distribution. This function is called in line 98, passing the image pointer and size. Note that the function will overwrite the original image.

```
filterGaussian3x3( pScaledImage, widthScl, heightScl );
```

The implementation of the algorithm (see lines 43–56) make use of the *separability* of the filter kernel.

$$\frac{1}{16}\begin{pmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{pmatrix} = \frac{1}{4}\begin{pmatrix}1\\ 2\\ 1\end{pmatrix} \cdot \frac{1}{4}\begin{pmatrix}1 & 2 & 1\end{pmatrix}$$

Generally, it is more efficient to filter in vertical direction first followed by a horizontal filtering on the result. But what does *filtering* actually mean in this context? Each pixel here is assigned a new value depending on its neighborhood (3×3 here). A filter matrix defines that linear calculation. It contains a weight mask for all pixels in the neighborhood. The resulting *weighted sum* is assigned to the center pixel. Other popular filter masks follow:

$\begin{pmatrix}\frac{1}{9} & \frac{1}{9} & \frac{1}{9}\\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9}\\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9}\end{pmatrix}$

3×3 mean filter (blurring)

$\frac{1}{25}\begin{pmatrix}1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1 & 1\end{pmatrix}$

5×5 mean filter (blurring)

$\begin{pmatrix}-1 & -1 & -1\\ -1 & 9 & -1\\ -1 & -1 & -1\end{pmatrix}$

3×3 sharpen

$\begin{pmatrix}1 & 2 & 1\\ 0 & 0 & 0\\ -1 & -2 & -1\end{pmatrix}$

3×3 Sobel edge filter (vertical component)

In section 6, we will see some more examples for filtering. Back to our code example. In `filterGauss3x3` temporary image memory is allocated for storing intermediate results first.

```
unsigned char * pFltY = new unsigned char[ width * height ];
```

Then, the subroutine `filterGauss1x3` is called. It processes the image pixel-wise, starting in the second and ending in the before-last row. This is managed by the outer loop:

```
for (int y = 1; y < height-1; ++y)
{
    ...
}
```

The implementation calculates two new pointers (i.e. positions in memory) `pDst` and `pSrc` in each iteration of the loop:

```
unsigned char * pDst = pImgDst + y * width;
const unsigned char * pSrc = pImgSrc + y * width;
```

The so called *pointer arithmetic* is an important issue in C. Starting at `pImgDst` we add an integer value `y * width`, resulting again in a pointer, `pDst` specifying the $(y \cdot width)$-th element position behind `pDst`. This is the first pixel in image row y. Thus, we are in line $1 \leq y < height - 1$ and defined pointers to the first values in these rows in both the input and the output image. The inner loop calculates the filtered response for each pixel within the current row (`y`):

```
for (int x = 0; x < width; ++x)
{
    const unsigned int sum = (int)pSrc[-width] + 2*(int)pSrc[0] + (int)pSrc[width];
    *pDst = sum / 4;
    ++pDst;
    ++pSrc;
}
```

Again, we make use of value access by index here. As we know that there is always at least one image row above and below the current pixel (this is why the outer loop was written that way), it is safe to access `pSrc[-width]` and `pSrc[width]` here. These are the pixel values from adjacent rows that are needed for the filter response. Note that we cast the values to `int`, as the `unsigned char` domain is exceeded during the computations made (e.g., negative values). The resulting value `sum` is divided by 4 (one could also use a shift operation here) and assigned to the destination pixel, where `pDst` is pointing at. `*pDst` is the same as `pDst[0]`. We increment both pointers by 1 (`++pDst` and `++pSrc`) in order to prepare processing of the next pixel within the row, which is another typical example for pointer arithmetic.

As we did not do *border handling*, that is, we did not process the first and last row, respectively, the original image values are simply copied there (`filterGaussian3x3`, line 51)

```
memcpy( pFltY, pImg, width );
memcpy( pFltY+width*(height-1), pImg+width*(height-1), width );
```

The routine `memcpy()` used here is quite low-level, but best suited for copying a block of memory. We simply pass pointers to the starting positions for reading and writing, respectively, and the number of bytes to be copied. Note that we want to copy `unsigned char` values which are single bytes in fact. If one wanted to copy values of type `int` using `memcpy()`, the corresponding number of items had to be multiplied by `sizeof(int)` to receive the correct size.

Now the second part of our filter kernel is calculated – taking the intermediate image as input and the original for writing. Finally, we clean up by unblocking the memory that has been allocated.

```
filterGauss3x1( pImg, pFltY, width, height );
delete pFltY;
```

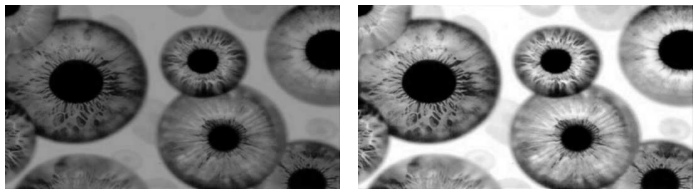
After the function returns to our `main()` we save the filter response image.

```
writePGM( "halfFiltered.pgm", pScaledImage, widthScl, heightScl );
```

to a file. Figure 2b shows how it should look like.

5. Histograms and image enhancement

For many applications it makes sense to improve the quality of the image before starting further computations. One common method is to calculate a histogram and modify the image such that the pixels make use of the full range of gray-values $\{0 \dots 255\}$. Depending on the used method we will receive higher contrast and an appropriate illumination.



(a) (b)

Figure 3: Our image (a) before and (b) after histogram stretching. The left image is the same as in figure 2b.

Before modifying the image by a histogram stretch, we compute a histogram with the full resolution of 8 bit. One may declare an array of 256 `unsigned int` values and initialize them to 0. The most efficient way of doing this is to use the C-function `memset()` that writes a specified byte-value to a memory block, given by the starting position and a length in bytes. Thus, we insert $256 \cdot 4$ bytes of zeros, starting at the position of the `histogram`:

```
unsigned int histogram[256];
memset( histogram, 0, 256*sizeof(unsigned int) );
```

The expression `sizeof(unsigned int)` is used for safety reasons, but will evaluate to 4 in most C/C++implementations.

Computing the actual histogram is as easy as counting all occurring gray-values. We iterate over all pixels and increment the corresponding histogram item.

```
for( int i = 0; i < widthScl*heightScl; i++ )
    ++histogram[ pScaledImage[i] ];
```

For the *stretching* we need to determine two bounds l and u for the mapping, which is a simple linear function $[l, u] \to [0, 255]$. We decide to cut off 5% of the lowest and highest histogram values in order to be robust to noise, so that two additional constants

`lowerPercentile` and `upperPercentile` are defined. They express how many low and high pixels are left outside the bounds.

```
const float          cutOffPercentage = 0.05;
unsigned char        lowerBound, upperBound;
unsigned int         histAccu = 0;
const unsigned int   lowerPercentile = cutOffPercentage * widthScl*heightScl;
const unsigned int   upperPercentile = (1-cutOffPercentage) * widthScl*heightScl;
```

Iterating through the histogram, the entries are accumulated in `histAccu`. As long as the accumulator is smaller than 5% of the pixels, we store the index as the lower bound and `continue` with the next iteration. Reaching the `lowerPercentile` we keep on iterating until `histAccu` exceeds the `upperPercentile`, so that the upper bound can be stored before `break`ing the loop.

```
for( int h = 0; h < 256 ; h++ )
{
    histAccu += histogram[h];
    if( histAccu <= lowerPercentile )
    {
        lowerBound = h;
        continue;
    }
    if( histAccu >= upperPercentile )
    {
        upperBound = h;
        break;
    }
}
```

Knowing the two bounds, the scale factor for the linear mapping can be determined as `histScale`. The resulting function is then applied to all image pixels receiving `newVal`. To make sure all values lie within $[0, 255]$ we limit them using the template functions `std::min()` and `std::max()`. Note that we perform these computations in integer type because the `unsigned char` domain is left in between.

```
const float histScale = 255. / (upperBound - lowerBound);
for( int i = 0; i < widthScl*heightScl; i++ )
{
    const int newVal = histScale * ( (int)pScaledImage[i] - lowerBound );
    pScaledImage[i] = std::min<int>( 255, std::max<int>( 0, newVal ) );
}
writePGM( "histogram.pgm", pScaledImage, widthScl, heightScl );
```

The resulting image is saved to `histogram.pgm`, see figure 3b.

6. Gradients, energy, and segmentation

A very important feature in images are *contours* or *edges* which result form visible object borders for instance. In order to detect edges, we have to find small image regions where "something changes", i.e., brightness in grayscale images. In this section we show how to compute a measure for edge strength based on simple filter operations (see sec. 4 also).

First, we allocate memory for the energy image and initialize it by setting all pixel values to 0 ('black') using `memset`. See line 147 in our code:

```
unsigned char * energy = new unsigned char[ widthScl * heightScl ];
memset( energy, 0, widthScl*heightScl );
```

The energy e of one pixel is calculated as $e = \sqrt{dx^2 + dy^2}$ where dx and dy are the strength of the local gradients in horizontal and vertical direction, respectively. We iterate over the image (ignoring a border of 1 pixel at each side) in a nested loop:

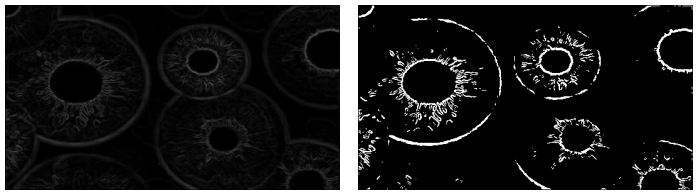
```
for (int y = 1; y < heightScl-1; ++y)
{
    const int rowOffset = y * widthScl;
    for (int x = 1; x < widthScl-1; ++x)
    {
        const int gradX = pScaledImage[rowOffset+x+1] - pScaledImage[rowOffset+x-1];
        const int gradY = pScaledImage[rowOffset+x+widthScl]
                        - pScaledImage[rowOffset+x-widthScl];
        energy[rowOffset+x] = sqrt( (float)(gradX * gradX + gradY * gradY) ) / sqrt(2.f);
    }
}
```

We can calculate the offset `rowOffset` of the current row in memory relative to `pScaledImage` at the beginning to the outer loop. Using this row offset in combination with the inner loop variable `x`, it is easy to access the four adjacent pixels that are needed for gradient computation: To calculate the energy of pixel (x, y), we calculate `gradX` based on $(x-1, y)$ and $(x+1, y)$ and `gradY` based on $(x, y-1)$ and $(x, y+1)$. For calculating the energy, we use `float` (not `double`) precision because it is sufficient here. The maximum possible energy is $e_{\max} = \sqrt{2 * 255^2} = \sqrt{2} \cdot 255$, therefore we make use of division by `sqrt(2.f)` in order to normalize to `unsigned char` range.

Finally, we write the resulting energy image to the disk,

```
writePGM( "energy.pgm", energy, widthScl, heightScl );
```

it should look like the image in figure 4a.



(a) (b)

Figure 4: (a) The energy image and (b) a binary version of it obtained by thresholding.

If one wants to have a *segmentation* showing whether a pixel belongs to an edge or not, a binary version of the energy image can be calculated. We did this by *thresholding*. The result is the simplest form of a so called *edge image*. We iterate over the whole image with a single `for` loop, reading and writing pixel-wise. The result is set to `true` (255) or `false` (0) here depending on the energy being higher or lower than 30 (hard-coded here), see line 167–173.

```
for( int i = 0; i < widthScl*heightScl; i++ )
{
    if( energy[i] > 30 )
        energy[i] = 255;
    else
        energy[i] = 0;
}

writePGM( "energyThresh.pgm", energy, widthScl, heightScl );
```

The resulting image is written to `energyThresh.pgm`. Figure 4b shows this binary edge image.

Now we are done with all examples concerning grayscale images. Therefore, we should de-allocate the memory we assigned in our code.

```
delete energy;
delete pScaledImage;
free( pImage );
```

Note that those pointer variables still exist and even point to the earlier locations. Nevertheless, after the next calls, it is not safe to access this memory. That's why you might see code where pointers are set to `0` after memory is released – not a bad idea in order to allow the detection of the pointer's state afterwards.

7. Color images

So far, we described the handling of grayscale images. If you want to consider color images instead, some differences emerge regarding image representation and access. With color many more image processing operations become possible!

In color images, one pixel cannot be described by a single intensity value. Instead, one typically uses a 3-tupel to define the color of one point. A very popular *color space* is *RGB* where the color of one pixel is defined by a red, a green and a blue component. Other color spaces are *HSV* and *YUV*, for instance.

From a programming point of view, it is desirable to allocate an even number of bytes per pixel in order to have efficient memory access on today's PCs. Therefore, colors should be represented by 32 bit *integers* in memory. However, we want to pack three 8-bit color components in such an integer and have easy access. That's why we define an appropriate data structure first:

```
union rtcvRgbaValue
{
    int m_Val;
    struct
    {
        unsigned char m_b;
        unsigned char m_g;
        unsigned char m_r;
        unsigned char m_a;
    };
    unsigned char byte[4];

    ...
};
```

This *union* data structure allows to define exactly what we wanted to have: a new data type with size of an integer where we can access the four single bytes comfortably by name (`m_r`, `m_g`, ...) or index (`byte[0]`, `byte[1]`, ...). Here, we use one byte for red, green and blue, respectively. The fourth byte is reserved to represent the so called *alpha* channel coding the transparency of the pixel. We will not make use of alpha values in this tutorial.

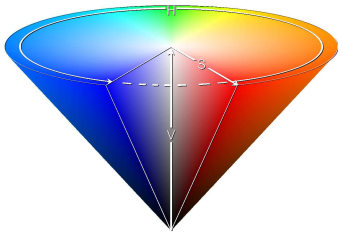


Figure 5: An illustration of the *HSV* color space. You could also process this `HSV_cone.ppm` to see what the following methods are doing in *HSV space*.

Now that we have a convenient data type, we simply write

```
rtcvRgbaValue * pRgbImage = 0;
```

in order to declare a pointer `pRgbImage` that can address a color image in memory. Again (see sec. 2), a predefined function is called in order to read from a file:

```
readOk = readPPM( "eyes_color.ppm", &pRgbImage, width, height );
```

Note that PPM is for color images what PGM is for grayscale images – namely a very simple data format, storing raw (uncompressed) image data with minimum overhead. We can check if reading the image was successful

```
if( ! readOk )
{
    printf( "Reading color image failed!\n");
    printf( "Press any key to exit.\n" );
    getchar();
    return -1;
}
```

and, if it was, we can now perform some exemplary operations on our color image (see fig. 6). Let's start by compiling some statistics. One could ask what the *dominant*, i.e. most frequent, color is. Unfortunately, the *RGB* color space is not very well suited for answering questions like this. Therefore, we build a histogram based on *hue* values, a component from the *HSV* color space. First, a histogram with eight bins is created and initialized (see sec. 5):

```
unsigned int hueHist[8] = {0,0,0,0,0,0,0,0};
```

Figure 6: The image loaded from `eyes_color.ppm`.

We could have used `memset()` here as well or even a loop. Now we iterate over all pixels of our image (in a single loop) and collect the statistics:

```
for( int i = 0; i < width*height; i++ )
    ++hueHist[ pRgbImage[i].getHue() >> 5 ];
```

This might look difficult – so first things first. `getHue()` is a function of our *RGB* datatype, calculating the *hue* value from r, g, and b. You might want to have a look at the implementation of this function in `PpmIO.h` or on Wikipedia[1]. The resulting *hue* value is *shifted* 5 bits to the right. This is the same as an integer division by $2^5 = 32$ but might be faster depending on hardware and compiler used. This division is done in order to calculate the appropriate bin $\{0, \ldots, 7\}$ in which the *hue* value $\{0, \ldots, 255\}$ goes into. Finally, the number of pixels in the related bin is increased by one with the pre-increment operator `++` (post-increment will do as well here).

In the next step (lines 206–215), we search our histogram for the bin `maxHue` containing the maximum number of contributing pixels. As each bin represents an interval of hue values, the index of the best filled bin allows to calculate the dominant color in our image.

```
int maxHist = 0;
int maxHue = -1;
for( int h = 0; h < 8; h++ )
{
    if( hueHist[h] > maxHist )
    {
        maxHist = hueHist[h];
        maxHue = h;
    }
}
```

In order to see what we found, let's create an image showing which pixels belong to the dominant color (see sec. 6). In the lines 220 to 230 we create a new image `hueSeg`, iterate over our input image, set the gray-values to 255 (white) if the pixel matches the description we found, and 0 (black) otherwise.

[1] `http://en.wikipedia.org/wiki/HSL_and_HSV`

```
unsigned char * hueSeg = new unsigned char[ width*height ];
for( int i = 0; i < width*height; i++ )
{
    const unsigned char hue = pRgbImage[i].getHue() >> 5; // in bins
    const unsigned char sat = pRgbImage[i].getSat();
    const unsigned char val = pRgbImage[i].getV();
    if( hue == maxHue && sat > 100 && val > 100 )
        hueSeg[i] = 255;
    else
        hueSeg[i] = 0;
}
```

In addition to the *hue*-check it is also necessary to exclude pixels that are *achromatic* by constraining the *saturation* and the v value. Try to do the same segmentation without these constraints and it will give you worse results. The image showing the segmentation is saved to a file (see fig. 7)

```
writePGM( "hueSegmentation.pgm", hueSeg, width, height );
```

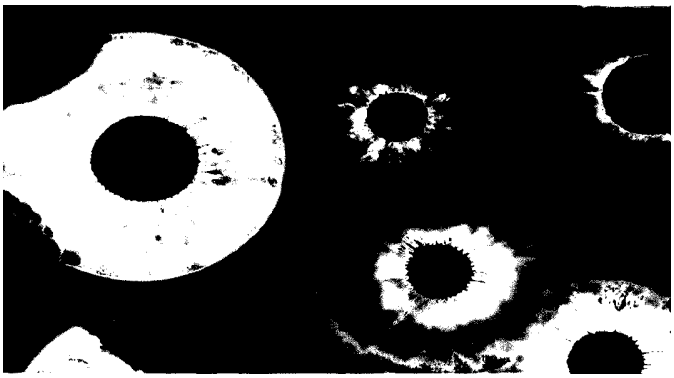


Figure 7: Segmentation of dominant color in our image.

Finally, we have to clean up and end our program. As the memory at `hueSeg` was allocated with `new` and memory at `pRgbImage` was allocated with `malloc()`, we have to use `delete` and `free()` respectively.

```
delete hueSeg;
free( pRgbImage );

printf( "Finished! Press any key.\n" );
getchar();

return 0;
```

We output some messages to the console, wait for the user to confirm, and return `0` which stands for "Everything went well".

8. Final remarks

In this part of the tutorial, we showed how to perform basic image processing tasks with C / C++. In order to teach many different things, we did basically the same in different

possible ways (allocating and accessing memory, for instance). This is not good practice for all day programming – there, you should be as consistent as possible.

The examples introduces in this first part can all be realized in plain C (with minimal adaptions). The second part is considering more advanced programming techniques.

9. Example program code

```
1 #include "PgmIO.h"
2 #include "PpmIO.h"
3
4 static void filterGauss3x1( unsigned char * pImgDst, const unsigned char * pImgSrc,
5                             const int width, const int height )
6 {
7     for (int y = 0; y < height; ++y)
8     {
9         unsigned char * pDst = pImgDst + y * width + 1;
10        const unsigned char * pSrc = pImgSrc + y * width;
11
12        for (int x = 1; x < width-1; ++x)
13        {
14            const unsigned int sum = (int)pSrc[0] + 2*(int)pSrc[1] + (int)pSrc[2];
15            *pDst = sum / 4;
16            ++pDst;
17            ++pSrc;
18        }
19    }
20 };
21
22 static void filterGauss1x3( unsigned char * pImgDst, const unsigned char * pImgSrc,
23                             const int width, const int height )
24 {
25    for (int y = 1; y < height-1; ++y)
26    {
27        unsigned char * pDst = pImgDst + y * width;
28        const unsigned char * pSrc = pImgSrc + y * width;
29
30        for (int x = 0; x < width; ++x)
31        {
32            const unsigned int sum = (int)pSrc[-width] + 2*(int)pSrc[0] + (int)pSrc[width];
33            *pDst = sum / 4;
34            ++pDst;
35            ++pSrc;
36        }
37    }
38 };
39
40 ////////////////////////////////////////////////////////////////////////
41 // Filter image with 3x3 Gaussian kernel (separated in 3x1 and 1x3)
42 ////////////////////////////////////////////////////////////////////////
43 static void filterGaussian3x3( unsigned char * pImg, const int width, const int height )
44 {
45    unsigned char * pFltY = new unsigned char[ width * height ];
46
47    filterGauss1x3( pFltY, pImg, width, height );
48
49    // copy border
50    memcpy( pFltY, pImg, width );
51    memcpy( pFltY+width*(height-1), pImg+width*(height-1), width );
52
53    filterGauss3x1( pImg, pFltY, width, height );
54
55    delete pFltY;
56 };
57
58
59 int main( int argc, char* argv[] )
60 {
61    ////////////////////////////////////////////////////////////////////////
62    // Read gray-value image
63    ////////////////////////////////////////////////////////////////////////
64    unsigned char * pImage = 0;
65    int width, height;
66
67    bool readOk = readPGM( "eyes_dark.pgm", &pImage, width, height );
68
69    if( ! readOk )
70    {
71        printf( "Reading image failed!\n");
72        printf( "Press any key to exit.\n" );
73        getchar();
74        return -1;
75    }
76
77    ////////////////////////////////////////////////////////////////////////
78    // Scale image by 1/2
79    ////////////////////////////////////////////////////////////////////////
80    const int widthScl = width / 2;
81    const int heightScl = height / 2;
```

```
82
83      unsigned char * pScaledImage = new unsigned char[ widthScl * heightScl ];
84
85      for( int y = 0; y < heightScl; y++ )
86      {
87          for( int x = 0; x < widthScl; x++ )
88          {
89              pScaledImage[ x + y*widthScl ] = pImage[ 2*x + 2*y*width ];
90          }
91      }
92
93      writePGM( "half.pgm", pScaledImage, widthScl, heightScl );
94
95      ////////////////////////////////////////////////////////////////////////
96      // Filter image with 3x3 Gaussian kernel
97      ////////////////////////////////////////////////////////////////////////
98      filterGaussian3x3( pScaledImage, widthScl, heightScl );
99
100     writePGM( "halfFiltered.pgm", pScaledImage, widthScl, heightScl );
101
102     ////////////////////////////////////////////////////////////////////////
103     // Compute histogram / cut upper and lower 5% of gray-values
104     ////////////////////////////////////////////////////////////////////////
105     unsigned int histogram[256];
106     memset( histogram, 0, 256*sizeof(unsigned int) );
107
108     // calculate histogram
109     for( int i = 0; i < widthScl*heightScl; i++ )
110         ++histogram[ pScaledImage[i] ];
111
112     // determine lower and upper bound for histogram stretch
113     const float          cutOffPercentage = 0.05;
114     unsigned char        lowerBound, upperBound;
115     unsigned int         histAccu = 0;
116     const unsigned int   lowerPercentile = cutOffPercentage * widthScl*heightScl;
117     const unsigned int   upperPercentile = (1-cutOffPercentage) * widthScl*heightScl;
118
119     for( int h = 0; h < 256 ; h++ )
120     {
121         histAccu += histogram[h];
122         if( histAccu <= lowerPercentile )
123         {
124             lowerBound = h;
125             continue;
126         }
127         if( histAccu >= upperPercentile )
128         {
129             upperBound = h;
130             break;
131         }
132     }
133
134     // assign new gray-values from linear mapping between lower and upper bound
135     const float histScale = 255. / (upperBound - lowerBound);
136     for( int i = 0; i < widthScl*heightScl; i++ )
137     {
138         const int newVal = histScale * ( (int)pScaledImage[i] - lowerBound );
139         pScaledImage[i] = std::min<int>( 255, std::max<int>( 0, newVal ) );
140     }
141
142     writePGM( "histogram.pgm", pScaledImage, widthScl, heightScl );
143
144     ////////////////////////////////////////////////////////////////////////
145     // Compute image energy from gradients
146     ////////////////////////////////////////////////////////////////////////
147     unsigned char * energy = new unsigned char[ widthScl * heightScl ];
148     memset( energy, 0, widthScl*heightScl );
149
150     for (int y = 1; y < heightScl-1; ++y)
151     {
152         const int rowOffset = y * widthScl;
153         for (int x = 1; x < widthScl-1; ++x)
154         {
155             const int gradX = pScaledImage[rowOffset+x+1] - pScaledImage[rowOffset+x-1];
156             const int gradY = pScaledImage[rowOffset+x+widthScl]
157                               - pScaledImage[rowOffset+x-widthScl];
158             energy[rowOffset+x] = sqrt( (float)(gradX * gradX + gradY * gradY) ) / sqrt(2.f);
159         }
160     }
161
162     writePGM( "energy.pgm", energy, widthScl, heightScl );
163
164     ////////////////////////////////////////////////////////////////////////
165     // Segment high energy areas by Thresholding
```

```
166     ////////////////////////////////////////////////////////////////////////
167     for( int i = 0; i < widthScl*heightScl; i++ )
168     {
169         if( energy[i] > 30 )
170             energy[i] = 255;
171         else
172             energy[i] = 0;
173     }
174     writePGM( "energyThresh.pgm", energy, widthScl, heightScl );
175
176     delete energy;
177     delete pScaledImage;
178     free( pImage );
179
180
181     ////////////////////////////////////////////////////////////////////////
182     // Read color (RGB) image
183     ////////////////////////////////////////////////////////////////////////
184     rtcvRgbaValue * pRgbImage = 0;
185     readOk = readPPM( "eyes_color.ppm", &pRgbImage, width, height );
186     //readOk = readPPM( "HSV_cone.ppm", &pRgbImage, width, height ); // proof of concept ;)
187
188     if( ! readOk )
189     {
190         printf( "Reading color image failed!\n");
191         printf( "Press any key to exit.\n" );
192         getchar();
193         return -1;
194     }
195
196     ////////////////////////////////////////////////////////////////////////
197     // Determine the dominant color from a Hue-histogram
198     ////////////////////////////////////////////////////////////////////////
199
200     // Compute hue-histogram with 8 bins
201     unsigned int hueHist[8] = {0,0,0,0,0,0,0,0};
202     for( int i = 0; i < width*height; i++ )
203         ++hueHist[ pRgbImage[i].getHue() >> 5 ];
204
205     // Find maximum in histogram
206     int maxHist = 0;
207     int maxHue = -1;
208     for( int h = 0; h < 8; h++ )
209     {
210         if( hueHist[h] > maxHist )
211         {
212             maxHist = hueHist[h];
213             maxHue = h;
214         }
215     }
216
217     ////////////////////////////////////////////////////////////////////////
218     // Segment dominant color (and neighbors) in HSV color space
219     ////////////////////////////////////////////////////////////////////////
220     unsigned char * hueSeg = new unsigned char[ width*height ];
221     for( int i = 0; i < width*height; i++ )
222     {
223         const unsigned char hue = pRgbImage[i].getHue() >> 5; // in bins
224         const unsigned char sat = pRgbImage[i].getSat();
225         const unsigned char val = pRgbImage[i].getV();
226         if( hue == maxHue && sat > 100 && val > 100 )
227             hueSeg[i] = 255;
228         else
229             hueSeg[i] = 0;
230     }
231
232     writePGM( "hueSegmentation.pgm", hueSeg, width, height );
233
234     delete hueSeg;
235     free( pRgbImage );
236
237     printf( "Finished! Press any key.\n" );
238     getchar();
239
240     return 0;
241 }
```

10. Exercises

Here are some more examples to show what we can do with images. You may want to try some of these exercises in order to visualise what you learned from the tutorial.

The exact *signature* of the functions is not defined in most exercises here. Often, you may either pass one pointer and manipulate the image directly or pass a `const` pointer and a second one for the result image. You could even use pointer to pointers or make use of the image class from the second part of the tutorial.

1. Write a function `convertToGrayscale(...)` that converts a color image (`const rtcvRgbaValue *`) to grayscale (`unsigned char *`). The intensity i of a pixel with color values (r, g, b) can be calculated as $i = 0.3 \cdot r + 0.6 \cdot g + 0.1 \cdot b$.
2. Write a function `resize_2(...)` that creates an image scaled by a factor of 2. Make use of adequate filtering in order to enhance the appearance.
3. Write a function `saveRoi(...)` accepting an image and a rectangular region of interest (ROI). The function should make use of `writePGM` in order to save only the selected region of the image to a file.
4. Write a function `filterMean(..., const int sxFilter, const int syFilter)` [`filterMedian`] that performs filtering of an image with a mean [median] filter of arbitrary size $sxFilter \times syFilter$.
5. Write a function `mirrorImageHorizontally(...)` [`mirrorImageVertically`] that creates a mirrored version of a grayscale [color] image.
6. Write a function `rotateImage_45(...)` that rotates a grayscale images by 45°. Note that the rotated image has a larger size and therefore you have to re-allocate the memory (backup the original image first!) or create a new image.
7. Write a function `rotateImage(..., double angle)` that rotates a grayscale images by *angle* degrees. Read up on *bilinear interpolation* – you can need it.
8. Extract from a color image three grayscale images, each storing one color channel separately (e.g., R, G, and B). Perform some operations (filtering, histogram stretching, ...) on the channels independently and then assemble the three parts to a single color image again.

Part II.
Advanced programming techniques

11. Introduction

The second part of our tutorial will follow the example code in section 14. We introduce a simple C++ template class in order to handle images. The focus of this section lies on some advanced programming techniques, so there are no further image processing examples.

12. A new template class

In this section, we show how a new class is declared and implemented. We develop the class `rtcvImage` that can be used to handle images of arbitrary pixel type and size. Therefore, we need to build a *template* class.

First, we include some stuff that is needed here.

```
#include <string.h>
#include "assert.h"
```

Then we start the declaration of our class using the template parameter `T` representing the type of data that the class actually handles. This will be resolved at compile-time: If someone writes code declaring `rtcvImage<unsigned char> imgGrayScale` and `rtcvImage<rtcvRgbaValue> imgRgb`, two different specialized implementations of our class will be created. The declaration starts in line 9.

```
template <typename T>
class rtcvImage
{
public:

    rtcvImage<T>(const int sx = 0, const int sy = 0, const T * pImg = 0);
    rtcvImage<T>(const rtcvImage<T> & img);
    ~rtcvImage<T>();

    rtcvImage<T> & operator=(const rtcvImage<T> & img);
```

We have two different *constructors* here. As the first one has *default values* for all parameters, it serves as *default constructor*. The second one is the so called *copy constructor* used to create a new object as a copy of a another object that already exists. We *have to* implement the copy constructor here because one of our members is a pointer and this would cause unwanted effects here. Note that the compiler automatically creates a copy constructor which is fine for many other cases.

The destructor `rtcvImage` is called when the object of our class is destroyed. We implement this as we have to release memory that has been allocated. If we don't, *memory leaks* will be caused. The assignment operator `operator=` has to be implemented here for the same reason as the copy constructor.

Note that up to now, the functions only were *declared* and not yet *implemented*.

Our class has three members storing the image data and the current size of the image (see line 59 in the source code). All three members are declared to be *private*, therefore they cannot be accessed from outside our class directly. We will provide member functions for access in order to ensures that we have full control, e.g. concerning memory allocation.

```
private:
    T * pData;

    int sx;
    int sy;
```

So let's continue with some functions allowing to use our new image class, see line 21–41.

```
    void changeSize(const int sx, const int sy);
    void fill( const T & val );
    void setData( const T * pDatNew );

    inline T * getDataPointer() const { return pData; }
    inline const T * getDataPointerConst() const { return pData; }
    inline int getSx() const { return sx; }
    inline int getSy() const { return sy; }
    inline int getSz() const { return sx*sy; }

    inline T getValue( const int idx ) const
    {
        assert( (idx >= 0) && (idx < sx*sy) );
        return ( pData[idx] );
    }

    inline T getValue( const int x, const int y ) const
    {
        assert( (x >= 0) && (x < sx) && (y >= 0) && (y < sy) );
        return ( pData[y * sx + x] );
    }
```

The first three functions are declared and will be implemented later like all others before. The following seven functions are typical *get*-functions ("getters") allowing others to read values of our private members. They are declared to be `inline` and `const` and are implemented directly. The `inline` keyword is a hint for the compiler not to perform a function call here but to execute the code directly. The declaration of a function (not a variable!) as `const` means that calling such a function cannot (must not) change the object (i.e. its members) the function belongs to.

We provide two different functions to access our pointer `pData` itself, one returning a `const` pointer ("read-only access"). Note that making the data pointer accessible outside the class involves some risk – somebody could deallocate our memory, for instance. So these functions contradict the idea of the private/getter/setter concept in some way. Nevertheless, we accept this in order to allow any processing of the image data outside the class efficiently.

We provide two different `getValue` functions allowing to access single pixel values by coordinates or index respectively. Note that we make use of `assert` statements here in order to make sure that only memory that has been allocated by us is accessed. These checks are only performed when the code is compiled in *debug* mode. Therefore, in *release* mode, we do not waste runtime on the one hand but have to rely on correct usage on the other hand.

The counterpart of get-functions are, of course, set-functions ("setters"). We implement such functions in line 43–55 for manipulation of single pixel values. Again, assert statements support debugging.

```
    inline void setValue( const int idx, const T & val ) const
    {
        assert( (idx >= 0) && (idx < sx*sy) );
        pData[idx] = val;
        return;
    }

    inline void setValue( const int x, const int y, const T & val ) const
    {
        assert( (x >= 0) && (x < sx)&& (y >= 0) && (y < sy) );
        pData[y * sx + x] = val;
        return;
    }
```

The declaration of our new image class is complete but we still have to implement some functions. The constructors are implemented as follows (line 67–82).

```
template<typename T> rtcvImage<T>::rtcvImage(const int sx, const int sy, const T * pImg)
{
    this->pData = 0;
    this->sx = 0;
    this->sy = 0;

    this->changeSize(sx, sy);

    if( pImg )
        setData( pImg );
}

template<typename T> rtcvImage<T>::rtcvImage(const rtcvImage<T> & img) : pData(0), sx(0), sy(0)
{
    this->changeSize( img.sx, img.sy );
    this->setData( img.getDataPointer() );
}
```

We have to initialize our members which is done in form of a initializer list in the second case. Have a look at the first constructor: if a concrete size of our image is given, we make use of our own function `setData` handling that. If furthermore image data is provided, `setData` will copy that to our own memory buffer. The second (copy) constructor works in a very similar way. However, here we get another image and have to assure becoming a copy of that.

The destructor releases memory that was allocated by any class member function. Note that calling `delete pData` is no problem even if the pointer is `0`. That is why all pointers are initialized to `0`.

```
template<typename T> rtcvImage<T>::~rtcvImage()
{
    delete pData;
}
```

The next function to be implemented is the assignment operator. It is necessary to perform the check if the left hand and right hand side are different: `this != &img`. This is a typical issue when implementing operators like this: As the first operation in `changeSize` (see below) is to deallocate our memory, *self-assignment* would destroy our own data.

```
template<typename T> rtcvImage<T> & rtcvImage<T>::operator=(const rtcvImage<T> & img)
{
    if ( this != &img )
    {
```

```
        this->changeSize( img.sx, img.sy );
        this->setData( img.getDataPointer() );
    }

    return *this;
}
```

The function `changeSize` manages the size of our image and therefore the amount of memory allocated. It does not scale the image content – previous image data is thrown away when calling `changeSize`.

```
template<typename T> void rtcvImage<T>::changeSize(const int sx, const int sy)
{
    assert( (sx >= 0) && (sy >= 0) );

    if ( (sx != this->sx) || (sy != this->sy) )
    {
        delete pData;
        pData = 0;

        if ( (sx > 0) && (sy > 0) )
        {
            pData = new T[sx*sy];
            this->sx = sx;
            this->sy = sy;
        }
        else
        {
            this->sx = 0;
            this->sy = 0;
        }
    }

    return;
}
```

Our next small function `fill` uses the very fast `memset` function to set all pixels' values to `val`. Therefore it can be used for initialization of images, for instance. Unfortunately, `memset` only works byte-wise so that we can only use this implementation for `unsigned char` or `char` images. For all other cases we provide a default implementation using `setValue`.

```
void rtcvImage<unsigned char>::fill( const unsigned char & val )
{
    memset( getDataPointer(), val, sx*sy );

    return;
}

void rtcvImage<char>::fill( const char & val )
{
    memset( getDataPointer(), val, sx*sy );

    return;
}

template<typename T> void rtcvImage<T>::fill( const T & val )
{
    const int size = sx*sy;
    for( int i=0; i<size; i++ )
        setValue( i, val );

    return;
}
```

Finally, the function `setData` allows to store image data (that has been created somewhere else, e.g., read from a file) in an object of `rtcvImage`. The data is simply copied to our own memory. We assume that we are allowed to read $sx \times sy$ values there. We can make use of the very fast function `memcpy` here.

```
template<typename T> void rtcvImage<T>::setData( const T * pDatNew )
{
    assert( ((sx == 0) && (sy == 0)) || (pDatNew != 0) );

    memcpy( pData, pDatNew, sx*sy*sizeof(T) );

    return;
}
```

13. Using our new class

Now that we have completed `rtcvImage`, we can create image objects like

```
rtcvImage<unsigned char> img;
rtcvImage<unsigned char> imgGray;
rtcvImage<int> imgInt(1, 1);
rtcvImage<double> imgDouble(100, 100);
rtcvImage<rtcvRgbaValue> imgCol(640, 480);
```

The size of the images can be changed later.

```
imgGray.changeSize(320, 240);
imgCol.changeSize(320, 240);
```

Let's perform some initializations.

```
imgGray.fill( 0 );
imgDouble.fill( 0. );
imgCol.fill( rtcvRgbaValue(255,0,0) );
```

The next short line of code will cause two things: the memory allocated by the object `img` is resized (320×240 elements) and the data from `imgGray` is copied.

```
img = imgGray;
```

The following ***does not work*** because the images are of different types. Nevertheless, one could write some conversion routines that handle the different cases (see exercises in sec. 15).

```
imgGray = imgCol;
```

Finally, let's read and write some pixel values.

```
imgDouble.setValue( 50, 50, 5. );  // access by coordinates
imgDouble.setValue( 1000, -0.5 );  // access by index

const int sy = imgDouble.getSy();
double * pData = imgDouble.getDataPointer();
pData[ 10 * sy + 5 ] = 7.5;  // index-based access at coordinates (x, y) = (5, 10)
pData[ 5 * sy + 10 ] = 10.;  // index-based access at coordinates (x, y) = (10, 5)
```

14. Example program code

```
#ifndef _RTCV_IMAGE_H_
#define _RTCV_IMAGE_H_


#include <string.h>
#include "assert.h"


template <typename T>
class rtcvImage
{

public:

    rtcvImage<T>(const int sx = 0, const int sy = 0, const T * pImg = 0);
    rtcvImage<T>(const rtcvImage<T> & img);
    ~rtcvImage<T>();

    rtcvImage<T> & operator=(const rtcvImage<T> & img);

    void changeSize( const int sx, const int sy );
    void fill( const T & val );
    void setData( const T * pDatNew );

    inline T * getDataPointer() const { return pData; }
    inline const T * getDataPointerConst() const { return pData; }
    inline int getSx() const { return sx; }
    inline int getSy() const { return sy; }
    inline int getSz() const { return sx*sy; }

    inline T getValue( const int idx ) const
    {
        assert( (idx >= 0) && (idx < sx*sy) );
        return ( pData[idx] );
    }

    inline T getValue( const int x, const int y ) const
    {
        assert( (x >= 0) && (x < sx)&& (y >= 0) && (y < sy) );
        return ( pData[y * sx + x] );
    }

    inline void setValue( const int idx, const T & val ) const
    {
        assert( (idx >= 0) && (idx < sx*sy) );
        pData[idx] = val;
        return;
    }

    inline void setValue( const int x, const int y, const T & val ) const
    {
        assert( (x >= 0) && (x < sx)&& (y >= 0) && (y < sy) );
        pData[y * sx + x] = val;
        return;
    }


private:
    T * pData;

    int sx;
    int sy;
};

//=========================================================

template<typename T> rtcvImage<T>::rtcvImage(const int sx, const int sy, const T * pImg)
{
    this->pData = 0;
    this->sx = 0;
    this->sy = 0;

    this->changeSize(sx, sy);

    if (pImg) setData( pImg );
}

template<typename T> rtcvImage<T>::rtcvImage(const rtcvImage<T> & img) : pData(0), sx(0), sy(0)
{
    this->changeSize( img.sx, img.sy );
    this->setData( img.getDataPointer() );
```

```
82 }
83
84 template<typename T> rtcvImage<T>::~rtcvImage()
85 {
86     delete pData;
87 }
88
89 template<typename T> rtcvImage<T> & rtcvImage<T>::operator=(const rtcvImage<T> & img)
90 {
91     if ( this != &img )
92     {
93         this->changeSize( img.sx, img.sy );
94         this->setData( img.getDataPointer() );
95     }
96
97     return *this;
98 }
99
100 template<typename T> void rtcvImage<T>::changeSize(const int sx, const int sy)
101 {
102     assert( (sx >= 0) && (sy >= 0) );
103
104     if ( (sx != this->sx) || (sy != this->sy) )
105     {
106         delete[] pData;
107         pData = 0;
108
109         if ( (sx > 0) && (sy > 0) )
110         {
111             pData = new T[sx*sy];
112             this->sx = sx;
113             this->sy = sy;
114         }
115         else
116         {
117             this->sx = 0;
118             this->sy = 0;
119         }
120     }
121
122     return;
123 }
124
125 template<typename T> void rtcvImage<T>::fill( const T & val )
126 {
127     const int size = sx*sy;
128     for( int i=0; i<size; i++ )
129         setValue( i, val );
130
131     return;
132 }
133
134 void rtcvImage<unsigned char>::fill( const unsigned char & val )
135 {
136     memset( getDataPointer(), val, sx*sy );
137
138     return;
139 }
140
141 template<typename T> void rtcvImage<T>::setData( const T * pDatNew )
142 {
143     assert( ((sx == 0) && (sy == 0)) || (pDatNew != 0) );
144
145     memcpy( pData, pDatNew, sx*sy*sizeof(T) );
146
147     return;
148 }
149
150
151 #endif  // #ifndef _RTCV_IMAGE_H_
```

15. Exercises

1. Write a function like `convertRtcvImage( const rtcvImage<int> & imgInt, rtcvImage<unsigned char> & img )` that accepts an integer image, scales its values to `unsigned char` range, and writes the results to the a new image `img`. This function could be used in order to visualize gradient images, for instance.
2. Write a new template class `rtcvVolume` that represents 3-dimensional images.
3. Choose some of the functions from part I and integrate them into `rtcvImage` as member functions.
4. Read about the C++ keyword `friend`. Write a new class `rtcvGrayScaleImage` that inherits from `rtcvImage<unsigned char>`. So the new class is specialized just for the one datatype. One could implement a `writePgm` function there which only makes sense for grayscale images.
5. Write a new class `rtcvColorImage` that inherits from `rtcvImage<rtcvRgbaValue>`. You could implement `writePpm` here (the *PPM* is for color images what *PGM* is for grayscale images – look it up).

References

[BBRS] Peter Becker, Andreas Bruchmann, Dirk F. Raetzel, and Manfred Sommer. C++ – Eine Einführung. http://www.mathematik.uni-marburg.de/~cpp.

[Cli] Marshall Cline. C++ faq lite. http://www.parashift.com/c++-faq-lite.

[Win09] Susanne Winter. Digitale Bildverarbeitung – Skript zur Vorlesung. http://www.neuroinformatik.ruhr-uni-bochum.de/thbio/group/medical/lecture, 2009.