Library design is an exercise in compromise. The ideal library is small, fast, powerful, flexible, extensible, intuitive, universally available, well supported, free of use restrictions, and bug free.

It is also nonexistent!

— Scott Meyers, More Effective C++ 1996
Abstract

This tutorial is aimed at new users of the Image Processing Library (IPL98) but can also be used as a reference for more experienced users. Every feature in the library is explained in detail accompanied with examples. This tutorial, the source code and other related information can be found in digital form at http://www.mip.sdu.dk/ipl98.
## Contents

1 Preface .......................................................... 1

2 Introduction ...................................................... 2

3 Purpose of the library ........................................ 4
   3.1 Conventions for new releases .......................... 4

4 Structure .......................................................... 5
   4.1 Basic types .................................................. 7
   4.2 Image representation in RAM ......................... 7
   4.3 File names and placement .............................. 8

5 Installing and compiling IPL98 ............................ 11
   5.1 Hardware dependency .................................. 11
   5.2 Using the C only part of the library ................ 12
   5.3 Using the C++ part of the library .................... 12
      5.3.1 MS Visual C++ 6.0 .............................. 13
      5.3.2 MS Visual C++ .Net ............................ 14
      5.3.3 Borland C++ Builder 6.0 ..................... 14
      5.3.4 GNU G++ .......................................... 15
   5.4 Why build libraries? ................................... 15
      5.4.1 Library files and linker errors ............... 16

6 General information ........................................... 18
   6.1 The streaming operators << and >> ................... 18
   6.2 Two and three dimensional points and vectors .... 20
   6.3 Timer and date ........................................... 22
   6.4 Error handling .......................................... 23
      6.4.1 Errors produced ................................. 23
   6.5 History .................................................. 24

7 Using IPL98 to represent and interact with images ... 27
   7.1 Creating an image in RAM .............................. 28
   7.2 Borders .................................................... 31
   7.3 Save, load and show an image .......................... 32
   7.4 Manipulating an image in RAM ...................... 34
   7.5 Working with pixels .................................... 35
      7.5.1 The palette ....................................... 35
      7.5.2 Pixel access .................................... 36
      7.5.3 Pixel access with iterators .................. 37
      7.5.4 Simple image manipulation ................... 40
   7.6 Working with a pixelgroup .................... 40
   7.7 Container for several pixelgroups .................. 45
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Using the image algorithms of IPL98</td>
<td>47</td>
</tr>
<tr>
<td>8.1</td>
<td>Segmentation</td>
<td>47</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Blob algorithm</td>
<td>47</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Contour tracing</td>
<td>49</td>
</tr>
<tr>
<td>8.2</td>
<td>Coordinate transformation</td>
<td>51</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Scaling and rotation</td>
<td>51</td>
</tr>
<tr>
<td>8.3</td>
<td>Feature extraction</td>
<td>52</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Moments</td>
<td>53</td>
</tr>
<tr>
<td>8.4</td>
<td>Morphological operations</td>
<td>55</td>
</tr>
<tr>
<td>8.4.1</td>
<td>Dilate and erode</td>
<td>55</td>
</tr>
<tr>
<td>8.4.2</td>
<td>Thinning and skeletonizing</td>
<td>57</td>
</tr>
<tr>
<td>8.5</td>
<td>Perspective camera transformation</td>
<td>58</td>
</tr>
<tr>
<td>9</td>
<td>Utilities</td>
<td>67</td>
</tr>
<tr>
<td>9.1</td>
<td>Macros and small utility functions</td>
<td>67</td>
</tr>
<tr>
<td>9.2</td>
<td>Graphics in images</td>
<td>68</td>
</tr>
<tr>
<td>9.3</td>
<td>Corresponding 3D-2D points</td>
<td>69</td>
</tr>
<tr>
<td>10</td>
<td>Camera classes and IPL98</td>
<td>71</td>
</tr>
<tr>
<td>11</td>
<td>Adding code to IPL98</td>
<td>74</td>
</tr>
<tr>
<td>A</td>
<td>Appendix: C code from C++ examples</td>
<td>83</td>
</tr>
<tr>
<td>A.1</td>
<td>Create an image in RAM</td>
<td>83</td>
</tr>
<tr>
<td>A.2</td>
<td>Manipulate an image in RAM</td>
<td>84</td>
</tr>
<tr>
<td>A.3</td>
<td>Segmentation</td>
<td>85</td>
</tr>
<tr>
<td>A.4</td>
<td>Morphological operations</td>
<td>86</td>
</tr>
<tr>
<td>A.5</td>
<td>Corresponding 3D-2D points</td>
<td>87</td>
</tr>
</tbody>
</table>
1. Preface

This tutorial was written in the summer 1999 as part of a Ph.D. institute work at the Maersk Mc-Kinney Moller Institute for Production Technology (MIP). Many people have made contributions and comments in the developing phase of both the Image Processing Library 98 itself and this tutorial. At MIP I would like to say thank you to Ivar Balslev, Brian Kirstein Ramsgaard and Arnold K. Christensen. Discussions with them resulted in many ideas regarding the library and the structure of this tutorial.

The tutorial has been updated in January 2001 for IPL98 version 1.40, and in September 2003 for IPL98 version 2.14. It is not a complete update and there are still many new features which are not mentioned. We hope to be able to continuously update this tutorial in the future. Always have a look at the IPL98 home page to see the latest news.

The front page contains three pictures taken from the M.Sc. Student project made by Brian K. Ramsgaard. That project is a part of the MIP project at the Maersk Institute. The code put on top of the images is from the Image Processing Library 98. The pictures and data used in section 8.5 is also from Brians project.
2 Introduction

This tutorial will guide you through every phase of using the Image Processing Library 98 (IPL98). It is written using version 2.14, but should be usable with all new versions to come. The library has been tested with the following compilers:

- **Visual C++ 6.0** Must install the newest service pack for Visual C++, which is SP5 at the time of writing.

- **Visual C++ .Net 2003** All Visual C++ 6.0 projects supplied with the IPL98 distribution needs to be converted when using Visual C++ .Net. You will still get many warnings from the compiler, but I have tested most of the library with .Net and it seems to work.

- **GCC 3.2** IPL98 v.2.14 probably works with all version of GCC 3.0.1 or newer, but to be sure, install v. 3.2 or newer.

- **Borland C++ Builder 6.0** We would have liked to continue the support of v. 5.0, but it simply turned out to be too difficult to bypass all the errors in this compiler and at the same time make it work for the other compilers.

The tutorial will explain the structure of the library, the philosophy behind it and how to use it. It will not introduce every single method in each class but will give the overview needed to work with the library. Not all of the parameters are explained here in detail for the mentioned methods, it is recommended to look at the IPL98 home page found at http://www.mip.sdu.dk/ipl98 and then go to the “C++ Overview” or “C Overview” section to get the exact description.

To get started fast you may skip the first few sections and go directly to section 5 about installing the library. The sections following the installation will teach you how to use the IPL98. It is strongly recommended to read section 6 and 7 before using the library. Section 8 can be read as the need for image algorithms grows. If an application is to be connected directly with a camera connected to the computer read section 10 and check out for new versions of the camera add on library for IPL98 because there will be added much code in this area in new versions. Section 11 is about how to write new code to be added to the library.

Before starting it must be emphasized that this is not an introduction to C and C++ but with only little knowledge about the language it should be possible to use this tutorial. The reader is also expected to have knowledge about image processing and the terms used in this area, naming conventions used in the image topics are primary taken from the book Digital Image Processing by R. C. Gonzales & R. E. Woods [1]. The tutorial puts the focus on the C++ part of the library but will also give (hopefully) enough explanation about the C part to make the reader capable of programming with IPL98 using C only. Most examples are in C++ only but some of the examples are written in C too and can be found in the appendix. Including the C examples directly in the chapters simply turned out to be too much code.

The library is developed with the Open Source\(^1\) philosophy in mind and can be downloaded at http://www.mip.sdu.dk/ipl98. IPL98 is in continuous development and new versions are frequently released. The future success of IPL98 rely a lot on interest, contributions and

\(^1\)More about this subject can be found at http://www.opensource.org
2. Introduction

responds from the users. A big effort is put into removing errors and bugs from the library to make it reliable and stable. But because it is difficult for the programmers to discover all errors we encourage everyone who finds errors to report it to the person(s) responsible for the development of IPL98\textsuperscript{2}. 

\textsuperscript{2}See http://www.mip.sdu.dk/ipl98 for more information.
3. Purpose of the library

The purpose of the library is to be useful, for combining tailor-made image processing and interpretation with standard methods for acquisitions, processing, display and storage of image information. Emphasis is put on interactivity in projects made by students, as well as for advanced research and development.

The key issues in developing the library are:

1. Easy to use
2. Platform independent
3. Support for both C++ and Java
4. Easy to add new code and algorithms
5. Fast access to image data and fast algorithms

To fulfil both requirements for *easy to use* and *fast* ... the library often have more than one possible way of doing things. There are for example several different ways of accessing pixels in an image. To support both C++ and Java it was decided to implement the library in ANSI C and build C++ and Java classes on top of it. We later discovered that the extra work associated with implementing everything in ANSI C was overwhelming, so we now implement most new additions in ISO C++ only. C++ classes encapsulating everything in the C kernel are available. At the moment there are no Java classes available, they will only be build in the future if it turns out to be the *de facto* standard or simply if somebody volunteer to implement it.

A lot of thought have been put into the design issues of the library and the first versions (before version 1.00) changed a lot as bad design in parts of the library were discovered. Since version 1.00 everything is backward compatible so installing new versions of IPL98 should not affect old code.

3.1 Conventions for new releases

From version 1.00 there will be no changes to names, methods and functions already included in the library, unless a specific method or class has its own version number below 1.0. Always look for that on the home-page documentation for the classes (or C functions).

There are two reasons for the convention above:

- It is very convenient for users to know, that future versions will have the same interface, changes will cause way to much work for users, who would have to change all their code every time they want to update IPL98 to the newest version.

- On the other hand, it is desirable to publish new methods, classes etc. as soon as possible, but this results in possible undesired interfaces, simply because we do not have the experience in using it yet. Therefore we have decided to put version numbers on methods, classes etc. if we are not sure about the interface. The user must be aware, when using such methods/classes the interface might change in future versions.
4. Structure

In this chapter a short overview of the basic structure of IPL98 is given, this includes the relationship between the C and C++ part, the class-hierarchy, basic type definitions and the filenames and placement.

It is not essential to read this chapter in order to use the library. But as you get more experienced and use more features of the library it will sooner or later become worthwhile to read this chapter. So if you are in a hurry to get started programming with IPL98, I suggest you skip this chapter and return to it at a later point.

IPL98 consists of a kernel implemented in C which contain the most basic and essential code of the library. The C code is implemented as object oriented as possible with class attributes contained in one structure and all functions operating on it contained in one file. Details about the naming convention are given in section 11 and relationship between C and C++ are given later in this section. A C++ wrapper using the kernel C function is implemented on top of it. The C++ part of the library also contains much functionality not available in the C kernel.

The library is divided into two logical sections for both the C and C++ part: Container classes and algorithm classes as can be seen in figure 1. The container classes is the code needed to create or load an image in RAM and accessing the image pixels and other information regarding an image, such as dimensions, bit depth, origo etc. Different classes is available in this section, more details about this subject can be found in chapter 4.2 on page 7.
4. Structure

algorithms are contained in several classes where the classes are named after the different topics in image processing. Read more about this in section 8.

To make the library both hardware and software platform independent it was decided not to have any GUI related issues in IPL98. So to actually show an image on screen is something the user must implement on its own or find information about elsewhere. Complete GUI class and tutorial for using IPL98 with Visual C v. 6.0 and newer versions, can be found at the IPL98 home-page in the section IPL98 Exchange. A GUI for the Borland C++ Builder is available at the same place. At this time, the Builder GUI is not updated to work with the Borland C++ Builder 6.0, which is the version we support with the newest IPL98.

As mentioned before the library is divided into two parts: An ANSI C part and a ANSI/ISO C++ part. The C part has been developed as much as possible from an object oriented view point. It is therefore relatively easy to implement the C++ wrapper classes on top of it. The C++ classes only calls the C functions and keeps track of the information returned by these functions. Because of the limitations in C as an object oriented language a few conventions is needed:

1. To make it easy for the user to recognize functions belonging to IPL98 every C function begins with one of the following prefixes: \texttt{k} (k for kernel, this includes all algorithm functions and functions related to the most general image structure), \texttt{kB} (Kernel Byte image functions belonging to the byteimage functions), \texttt{kF} (kernel float, \texttt{kC} (kernel integer and \texttt{kI} (kernel complex image functions).

2. If a class has some attributes the corresponding C implementation will have a structure containing the attributes. The class will then contain an attribute of a type with that structure. The class name and the structure name is the same except the structure begins with a “T” and the class with a “C”. For instance the class CPixelGroup has the corresponding C structure name TPixelGroup.

3. Instead of a default constructor the user must remember to call a function named \texttt{kInitStructureName()}, where StructureName is the corresponding name of the structure.

4. Before leaving a scope where a structure is to be destroyed the user must remember to call a function name \texttt{kEmptyStructureName()}, where StructureName is the corresponding name of the structure.

Here is a simple example in C++ where an image with dimensions 100 times 80 with true color (24 bits per pixel) is created in RAM:

```cpp
#include <ipl98_cplusplus.h>
using namespace ipl;
void main()
{
    CImage Img(100,80,24);
}
```

In ANSI C the corresponding code is;
#include <ipl98.h>
void main()
{
    TImage Img;
    k_InitImage(&Img);
    k_AllocImage(100,80,24,&Img);
    k_EmptyImage(&Img);
}

A few methods have been implemented directly in C++ although they exist in the C
kernel. This is done in order to obtain maximum execution speed of programs. The overhead
of an extra function call to the C kernel can for some often used functions be severe. For
instance accessing pixels with GetPixel() and SetPixel(). In addition, such ambiguous C++
methods are in-lined to obtain further speed.

## 4.1 Basic types

When working with images it is important to have some basic types with a well defined
number of bytes in memory. This is due to the fact that the image file formats used on a
computer are created with a specific number of bits to represent each pixel. Because of the
lack of such well defined types in the C and C++ language it has been necessary to define
the following types:

<table>
<thead>
<tr>
<th>Type name</th>
<th>Description</th>
<th>Min. value</th>
<th>Max. value</th>
</tr>
</thead>
<tbody>
<tr>
<td>UINT8</td>
<td>unsigned 8 bit integer</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>INT8</td>
<td>signed 8 bit integer</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>UINT16</td>
<td>unsigned 16 bit integer</td>
<td>0</td>
<td>65535</td>
</tr>
<tr>
<td>INT16</td>
<td>signed 16 bit integer</td>
<td>-32768</td>
<td>32767</td>
</tr>
<tr>
<td>UINT32</td>
<td>unsigned 32 bit integer</td>
<td>0</td>
<td>4294967295</td>
</tr>
<tr>
<td>INT32</td>
<td>signed 32 bit integer</td>
<td>-2147483648</td>
<td>2147483647</td>
</tr>
<tr>
<td>FLOAT32</td>
<td>32 bit floating point</td>
<td>1E-37</td>
<td>1E+37</td>
</tr>
</tbody>
</table>

The naming convention is “U” for unsigned otherwise signed, followed by the type (integer
or float) and at the end the number of bits used to represent the number.

## 4.2 Image representation in RAM

The internal memory representation of an image is optimized to be fast to show on an x86
machine, figure 2 and 3 show how the pixels are stored in memory. The image pixels are stored
in a one-dimensional UINT8 array. Each scanline in this array are always zero-padded, so
that each scanline ends on a 32-bit (UINT32) boundary. The scanlines are stored in the array
beginning with the scanline in the bottom of the image. Because of the zero-padding and
the "bottom-up" storage of scanlines it is then possible relatively fast to show the image on
a Windows platform. All information about an image is stored in the "TImage" structure.
This includes three "different" pointers to the image data.

Accessing image data from the C++ part of the library does not involve direct access to
the TImage structure. Several methods are available for accessing image data, the most easy
4.3 File names and placement

Figure 2: Image representation in RAM. Two scanlines are shown. The first bytes in memory is the bottom line in the image. A zero-padding follows each scanline to make every scanline a multiple of 4 byte (i.e. double word aligned). This image is without border, see figure 3 below to see how a border is placed around the image. The pointer pPixelStream is a member of the C structure TImage which contains all the information for an image. In C++ the TImage structure is an attribute in the CStdImage class and the data is accessed through the methods in the class.

way is to use GetPixel(), SetPixel(), GetPixelFast(), or SetPixelFast(). If execution time is to be minimized, iterators can be used, they are available for all container classes in IPL98.

It is possible dynamically to set a variable sized border around the image. This makes it a lot easier to perform certain image manipulation routines. Then you do not have to worry about "edge" problems when accessing pixels near the edge of the image, for example when "running" a mask over the whole image. The border is included in the one-dimensional array together with the image data. The drawback of a border however is, that two representations of the image data are necessary, one for image analysis and one for showing the image on the screen.

4.3 File names and placement

When you download (see chapter 5) the IPL98 distribution, it may seem a bit overwhelming at first. To make it easy for beginners we have included complete projects for the three compilers that we support at the moment. This includes the projects, that creates library files from the IPL98 source code and example projects showing you how to use these library files. The complete structure can be seen in figure 4.

The source code itself is located in the folder source, projects building the library from the source code, are placed in the folder projects. They generate library files which will be created in the folder lib. Finally, some example projects using the generated libraries are found in the folder examples. A small readme.txt file is located in each project folder, it gives a short explanation on how to use the actual project. Two different library projects are available for the Borland C++ Builder and the Visual C++ compilers. One is named ipl98library and the other ipl98library_gui. The latter ensures, that IPL98 does not define the global function AddIPL98ErrorHistory(...), which takes care of outputting error message generated by IPL98. It must then be defined somewhere else. The reason for this is, that for projects containing a GUI front end, the creator of such a project wants to define a version of the error function redirecting the output to a window. The default version in IPL98 simply writes error output to the console window.

In the following, we will focus on the source folder, where the IPL98 source code is located.
4.3 File names and placement

![Image Memory Diagram]

Figure 3: The image memory in a "visual" form. The image is here shown with a border around it and the pointers available to the memory is also shown. They are all members of the C structure TImage which contains all the information for an image. In C++ the TImage structure is an attribute in the CStdImage class and the data is accessed through the methods in the class. Be aware that accessing a position (x,y) in the image with the ppMatrix pointer is done with the y-position first as the example ppMatrix[y][x] in the figure shows.

The files belonging to IPL98 is split into three folders (one of them contains several subfolders): ipl98, points and time, the hierarchy of the folders can be seen in figure 4 under the source folder. The folder ipl98 contains both the C and C++ code for the containers and image classes/functions. The points folder defines C++ classes giving two and three dimensional points and corresponding functions to work with the points, derived 2D and 3D vector classes are available in source/ipl98/cpp/vectors. The point classes can be used independently of the IPL98 library. The time folder contains C functions and a C++ class for using a timer and can also be used without using the rest of the library. All ANSI C code is placed in folders named kernelc in the relevant folders.

The ipl98 folder contains most of the code in the library. Some general include files are placed in this folder, the user should normally not use any of these files. The subfolders are: cpp_classes and kernelc. The cpp_classes contains all the C++ code. Here are all container classes, algorithm classes and a palette class. The folder algorithms contains classes for image processing algorithms. The file names containing classes should always have the same name as the class name except the leading letter "C". In the kernelc folder are the ANSI C code. Here the different matrix image classes are grouped into folders with relevant names.

---

3It still uses a few files in the ipl98 folder, so don’t delete this folder if you only want to use the timer functions.
4.3 File names and placement

Figure 4: The folder structure of the IPL98 distribution. The actual IPL98 source code is located in the source folder. Projects for building libraries from the source code, are found in the projects folder. Examples projects are located in the folder examples. Note, that project are available for all three supported compiles. A small file named readme.txt is available in each project folder, explaining how to use the actual project.
5 Installing and compiling IPL98

The library can be used as a C only library or as a C++ library. In both cases, download the latest distribution from http://ipl98.sourceforge.net/. Place the IPL98 files on your local drive, say \texttt{c:/}, in exactly the same structure as you received the IPL98 files in, you should get a folder named \texttt{ipl98} in the root of your c-drive. Then make sure your compiler have a path set to the location of the source files, which is \texttt{c:/ipl98/source}. Before explaining how to build the library and use the IPL98 code, you have to be aware of what hardware you are using.

5.1 Hardware dependency

In order to fulfill the desire of making a platform and hardware independent library two things must be taken into consideration. The first thing to check is if your hardware is of type BIG ENDIAN or LITTLE ENDIAN. The Intel x86 processors and compatible are LITTLE ENDIAN types whereas most RISC processors are BIG ENDIAN, that includes most (maybe all?) SGI and SUN machines. As default the library is set to use LITTLE ENDIAN, ie. when using it on a x86 architecture you do not have to make any changes.

In case of a BIG ENDIAN machine you must remove the comment-signs in the file \texttt{ipl98/ipl98_setup.h}:

```c
#include <ipl98/kernel_c/ipl98_types.h> // assumes that you have
  // set a general include path to the IPL98
  // source code as described in the beginning
  // of this chapter!
using namespace ipl; // only if you are using a C++ compiler,
  // otherwise remove this line!

int main(int argc, char* argv[]) {
  printf("size INT8 = %d (should be 1)\n",sizeof(INT8));
  printf("size UINT8 = %d (should be 1)\n",sizeof(UINT8));
  printf("size INT16 = %d (should be 2)\n",sizeof(INT16));
  printf("size UINT16 = %d (should be 2)\n",sizeof(UINT16));
  printf("size INT32 = %d (should be 4)\n",sizeof(INT32));
  printf("size UINT32 = %d (should be 4)\n",sizeof(UINT32));
  printf("size FLOAT32 = %d (should be 4)\n",sizeof(FLOAT32));
  return 0;
}
```
5.2 Using the C only part of the library

If the sizes are not correct you must change the type in the file *ipl98/kernel_c/ipl98_types.h*. On most machines the default types are correct so it should not be necessary to change these definitions.

5.2 Using the C only part of the library

In case of using the C only part of the library, include the files */ipl98/ipl98.h* and */ipl98/ipl98.c* in your project. These files include all other files needed. Then include the file */ipl98/ipl98.h* in your own files where you want to use the library. Here is an example on how to do it:

```c
#include <ipl98/ipl98.h>
int main()
{
  return 0;
}
```

This is not the most elegant way of using a library, but this is how we recommend to use the C only part of the library, to get started quickly. The drawback is that compilation time is unnecessary long, and if you have several projects, it gets even worse, read more about this subject in section 5.4. If you want to speed up things, you can either built a library, or you can manually find out exactly what *.c* files needs to be included in your project.

5.3 Using the C++ part of the library

The library has been compiled with several compilers and environments. First I will describe in general terms what needs to be done in order to compile and use the library. This will be followed by a detailed description of each compiler that we support.

Since v. 2.0 of IPL98 the installation procedure has changed. The IPL98 distribution now contains the source code and corresponding project building library (lib) files. This move has been necessary in order to reduce compilation time for the programmer using IPL98. We also moved away from including just one file, which would include the whole library, also done to reduce compilation time.

A little knowledge is needed in order to setup a compiler for working with library files. It may seem cumbersome to set up a compiler for using library files and making sure that the correct runtime libraries are used (read more about this subject in section 5.4.1), but when you (finally) make it work, I think you will appreciate it. Including a header file for each class you want to use, may also seem a bit cumbersome, but trust me on this: The compilation time can be considerably faster and this fact alone is worth the effort, especially when working with big projects!

First of all, you need to set up a path to the IPL98 source code, this can be done in a general include path for the compiler you are using, refer to your compiler manual for details about how to set this up. The path must point to *ipl98/source*, preceded by the path where you placed the IPL98 files. Say you placed the files at the root of the c-drive, your path will be *c:/ipl98/source*. Next, you need to compile the code and build the library file. In the folder *ipl98/projects*, you will find projects building the library file for each compiler that we officially support. You can add your own library project here, if you are using a different compiler. Finally, you must ensure that your project uses the library file, otherwise you will get linker errors. You need to setup the include path for your library files, for IPL98 this
5.3 Using the C++ part of the library

Figure 5: Setting the different paths for the Visual C++ compiler.

will be *ipl98/lib*, and then you need to tell your linker-part of your compiler, that the IPL98 library file is part of the code to be linked. Example projects using the IPL98 library can be found in *ipl98/examples*. Let’s now turn to a more specific description for each compiler.

5.3.1 MS Visual C++ 6.0

Let’s assume you installed the library at the root of the c-drive, you will then have a folder named: *c:/ipl98*. First you need to set up directories for include files, source files and library files. Open the Options dialog box by menu Tools — Options. Click the Directories tab. In the field ”Show directories for”, select Include files. Add the path *c:/ipl98/source*. Then select Library files and add the path *c:/ipl98/lib*. Finally select ”Source files” and add the path *c:/ipl98/source*. This sets up the paths to all resources. Figure 5 shows the dialog box.

You can now either open the project *c:/ipl98/projects/visualc* for building the library file. You will notice, that there are two library projects! The project *ipl98library_gui* is for the case, where you want to supply your own error-function used by IPL98, the project *ipl98library* defines the error function directly. Read more about this subject in section 4.3. You can also go directly to the example projects located in *ipl98/examples/visualc*. Actually, you will here find two workspaces, where each workspace contains two projects. One is the project building the library file and the other is an example project. The smart thing about workspaces in Visual C++ is, that they can setup dependencies between different projects. This ensures that the compiler will take care of building all your code form different projects automatically and in the right order! This is already done for you in these workspaces. If you for instance open the workspace ConsoleIPL, you will in the left pane (called the Workspace) see two projects. The first named ConsoleIPL and the second is the ipl98library project which is the project located in *ipl98/projects/visualc/ipl98library*. The ConsoleIPL project depends on the ipl98library project, so if the latter is not up to date, it will automatically be build when you compile and your ConsoleIPL project will be relinked to the IPL98 library file. Dependencies between the projects in a workspace can be seen by opening the menu Projects — Dependencies. A final notice is that you must ensure that the destination project (your
5.3 Using the C++ part of the library

Project which will actually build the executable file) is the active project. In the left pane you can right-click on the project you want to be active and select "Set as active project", the active project is written in bold. You build the project by choosing menu Build — Build ConsoleIPL (F7). If the IPL98 library file is not built yet, this will take quite a while, so be patient!

The other example project named show?7_educational is a complete GUI front end. To use this project, you need to understand the document-view structure of the Visual C++ environment, many books are available treating this subject.

The first time you try to debug into IPL98 code, the compiler will probably ask you for the location of that file. If you note what class you are debugging into, locate the file name for the header file by looking at the C++ documentation on the IPL98 home-page. On top of each documentation page for each class, you will see the exact #include statement. This is also where you will find the corresponding .cpp file for this class. Simply direct the compiler to this folder. Subsequent debugging into other classes should be automatic.

5.3.2 MS Visual C++ .Net

We are not yet supporting Visual C++ .Net. But I did try to convert the supplied Visual C++ project to the .Net version. I removed some errors reported by the .Net compiler and removed quite a few warnings. There are still many warnings, but none of them are severe! So everything should work if you simply let the compiler update the projects to the .Net version the first time you open them. The setup procedure is similar to the one described in the previous section, although dialog boxes and other things may have changed slightly.

5.3.3 Borland C++ Builder 6.0

With the release of IPL98 v. 2.14 we had to skip supporting Borland C++ Builder 5.0, it simply gave to many headaches trying to get around issues with the 5.0 compiler not being ISO C++ compliant.

First you need to create the IPL98 library file. You can either open one of the projects in the folder ipl98library or one of the projects in the folder ipl98library_gui, all located in the folder ipl98/projects/borlandbuilder, see section 4.3 and section 6.4.1 for more information about the difference between the GUI and non-GUI projects. Since the current IPL98 distribution does not include a GUI front end for the Borland C++ Builder, we will here focus on the projects in the ipl98library folder. The project ipl98_borland.bpr builds a library file ipl98_borland.lib without debug information, and places it in the folder ipl98/lib. The project ipl98_borland_d.bpr builds the library file ipl98_borland_d.lib including debug information and places it in the same folder as the non-debug version. Open both projects (one at a time) and compile them by choosing the menu Project — Build ipl98_borland(_d).

Now go to the folder ipl98/examples/borlandbuilder/ConsoleIPL and open the project. Compile it by choosing the menu Project — Build Project2. It should compile without any problems. You should also be able to debug into the IPL98 code, since the default settings for the project links with the debug version of the IPL98 library file.

To link with the non-debug version of the library (which will make the executable program run faster), go to the menu View — Project Manager. Remove the entry ipl98_borland_d.lib and insert the library file ipl98_borland.lib located in ipl98/lib. Figure 6 shows the dialog box. Compile the project again, this time you will not be able to debug into the IPL98 code.
5.4 Why build libraries?

Why build library files? At first, it may seem strange to build library files, with all the trouble it involves, see section 5.4.1 below. But it does make sense for two reasons:

- Compiled source code can be shared between different projects. In most cases you will end up having several different projects using the same library. Imagine if you have X projects using IPL98, changing some code in IPL98 would force you to rebuild the IPL98-object files X times, once for each project. It takes time and also takes up quite a lot of disc-space. With library files you only need to rebuild once, the other projects depending on the library file, will automatically include the new code without recompilation of source code.

You can contact Michael Landsiedl (michael.landsiedl@jku.at) if you have further questions regarding the Borland C++ Builder.

5.3.4 GNU G++

The g++ compiler v. 3.2 has been tested on a UNIX machine with IPL98 v. 2.14. The IPL98 v. 2.14 beta has also been used on Linux running on a Pentium PC. G++ versions from 3.0.1 and newer should work, but to be sure, use v. 3.2 or newer.

To create the IPL98 library file, go to the folder ipl98/projects/gcc and type "make". This should run the Makefile present in that folder. After this you need to type "make install", this will move the library file libipl98.a to ipl98/lib. You can now go to the folder ipl98/examples/gcc/ConsoleIPL and type "make". This will build the small example in the file main.cpp and link with the IPL98 library file.

For more information about using the g++ compiler with IPL98, please contact Anders Lau Olsen (alauo@mip.sdu.dk)

5.4 Why build libraries?

Figure 6: Remove the file ipl98_borland_d.lib and insert the file ipl98_borland.lib to run your program without IPL98 debug information.
5.4 Why build libraries?

- If you didn’t have library files, you would need to decide what cpp-files to include, and
  this can be quite a task, even if you build the library yourself. Say you want to use the
  CImage class in the IPL98 library, you will then include the file "ipl98/cpp/image.h". But
  this file have several methods using other container classes, for instance CByteImage. The
  image.h file will take care of including the header file for CByteImage, but you would have
  to include the corresponding cpp-file. And if you don’t want to use a CByteImage, you will
  not like to spend time figuring out what cpp-file to include in order to get the definition
  for the CByteImage class. If you look in the header file for image.h, you will realize that
  the class uses a lot of other classes and it will be a nightmare for you to figure out what
  cpp-files to include to get all the definitions. Even worse, the other files depend yet again
  on other classes! And don’t think the linker will save you, since there are no rules about
  file naming and class names, the linker cannot guess what file you needed to include to
  make the definitions available for the linker, so you will just get a lot of errors with some
  member names missing for different classes. On top of that, most linkers shows the mangled
  name for the missing methods, so you need to look close at the messages to figure out what
  class definition is missing.

5.4.1 Library files and linker errors

You will sooner or later discover this: The linker outputs dozens of errors reporting missing
definitions for functions and methods. Don’t be afraid, read this and you will have a good
idea about what is wrong. This is written for the MS Visual C++ compiler only, but if you
encounter similar problems with another compiler, this knowledge could be useful.

Visual C++ comes with several different run-time libraries. A run-time library is the core
code provided by your language, in C++ it is the old ANSI C library and the std-library.
For some reason (probably a good one, but I’m not an expert here), the different run-time
libraries are incompatible. It means you cannot link part of your code with one run-time
library and another part with another run-time library. Normally this is not a problem, since
one project can only link to one run-time library. But when you use library (LIB) files the
situation changes, you must make sure to use the same run-time library when compiling your
code as the one used with the LIB-file provided by someone else.

A typical error from Visual C++’s linker looks like this:

LIBCD.lib(crt0dat.obj) : error LNK2005: __cinit already defined in
LIBCMTD.lib(crt0dat.obj)

Here I used a library compiled with a single threaded run-time library but the actual
project was compiled with the multi threaded run-time library. So when you use a lib-file
make sure you know what run-time library it was compiled with (you can also try different
combinations with your own compilation, but it can be tedious!).

Microsoft has a naming convention to help you figure out what run-time library is in
use. The letters MT in the library file name means Multi Threaded, the letter D means
debug version, and RT means multi threaded Run Time, that is the library depends on a
corresponding DLL-file. A more detailed explanation of Visual C++’s different run-time
libraries can be found at MS C Run-time libraries.

I guess your next question is: So how do I choose the run-time library? This is easy. Open
"Project Settings" from the menu bar Project — Settings. Click the C/C++ tab and choose
"Code generation" in the Category field. Just below it to the right it will say "Use run-time
libraries".
5.4 Why build libraries?

Figure 7: Choosing the correct run time library to link with your program is important when working with pre-compiled library files.

library”. Here you can choose between the three different types of run-time libraries in both debug and release versions - a total of six combinations. The dialog box is shown in figure 7.
6 General information

This section describes some general behavior of the library which is common for most classes. It also describes the 2 and 3 dimensional point and vector classes.

Version

The version number of IPL98 can be found by the C definition IPL98_VERSION located in the file ipl98_types.h. Here is a simple example using it:

```cpp
#include <ipl98/ipl98_cplusplus.h>
using namespace ipl;

void main()
{
    if (IPL98_VERSION==2.14)
    {
        cout << "IPL98 Version " << IPL98_VERSION << " is in use" << endl;
    }
}
```

It may be a good idea in a final application to be able to see the version number of IPL98 which the application were compiled with.

6.1 The streaming operators << and >>

The streaming operators are available in most container classes and are used to write, and in some cases read, the most simple data encapsulated in a class to/from the given stream. In many cases not all data contained in a class is written when using the streaming operator, for this purpose a PrintInfo(...) method is available in most classes. The method often takes some parameters describing how to output all the data contained in a class. Below is given some simple examples of how this works, the description of the CPixelGroup class used can be found in section 7.6:

```cpp
CPixelGroup pg;
pg.AddPosition(1,4);
pg.AddPosition(3,7);

cout << pg << endl; // using the streaming operator
    // the positions are not written

pg.PrintInfo(true,1); // using the PrintInfo method and
    // choosing 1 position on each line
    // on the output
```

The output from the streaming results in this:

```
*************** PixelGroup info ***************
Positions: Top=(1,4) Bottom=(3,7) Left=(1,4) Right=(3,7)
NumberOfPixels=2 AllocatedPixels=2
```
6.1 The streaming operators << and >>

whereas the output produced from the PrintInfo(...) method is:

**************** PixelGroup info ****************
Positions: Top=(1,4) Bottom=(3,7) Left=(1,4) Right=(3,7)
NumberOfPixels=2 AllocatedPixels=2

pos 0=(1,4)
pos 1=(3,7)

All streaming and PrintInfo(...) for a class should begin with a line of "*" characters with the class name in the middle as shown in this example.

Some simple classes can stream their entire contents by the use of the streaming operator. This means that you can use this operator to save and load files containing such objects. One example is the CPoint2D<T> class (T is the template type). Here is a short example:

```cpp
#include <vector>
#include <fstream>
#include <points/point2d.h>

using namespace std;

int main(int argc, char *argv[]) {
    // create an array with a few points
    vector<CPoint2D<int> > Array;
    Array.push_back( CPoint2D<int>(1,4));
    Array.push_back( CPoint2D<int>(2,5));
    Array.push_back( CPoint2D<int>(3,6));

    ofstream ofst; // create an output file stream
    ofst.open("temp.txt");
    if (ofst)
    {
        for(int i=0; i<Array.size(); i++)
            ofst << Array[i] << endl;
        // read file again
        ifstream ifst;
        ifst.open("temp.txt");
        if (ifst)
        {
            CPoint2D<int> P;
            while(!ifst.eof())
            {
                P << ifst;
                cout << "P=" << P << endl;
            }
        }
        else
            cerr << "Failed reading file" << endl;
    }
```
If a class does not stream its entire contents by the use of the streaming operator and you need to store the information in a file, there will normally be a `Save()` method in the class. We may change this behavior slightly in future versions, since there are some advantages of always using the streaming operator (makes it easier to save several objects directly to one file).

### 6.2 Two and three dimensional points and vectors

The two template classes `CPoint2D<T>` and `CPoint3D<T>` are available for working with two or three dimensional points. The type `T` defines what type is to be used for the given point and can be any simple type such as short, float or integer, additional information about template classes can be found in standard C++ programming books. It is the goal of IPL98 to have all methods which takes a 2D point as parameter implemented in two versions: One taking a `CPoint2D<T>` and one taking two simple types `x,y`. And of course the same case with 3D points. This means the programmer can choose to work with the `CPoint2D` and `CPoint3D` classes which often gives a more nice looking code or work with the simple types which in some cases gives faster execution of the code (actually this should only happen in some very rare cases!). An example of a double implementation of a method is:

```cpp
GetPixel(const CPoint2D<int>& Pos)
GetPixel(int x, int y)
```

The first method works with a `CPoint2D` integer type and the second takes two simple types which constitutes a 2D point. Many operators are overloaded for the point classes, here is an example with the `CPoint2D` class used with integer position values giving an idea of how to work with points:

```cpp
CPoint2D<int> p1; // default constructor - integer point set to (0,0)
CPoint2D<int> p2(3,5); // integer point initialises to (3,5)
CPoint2D<int> p3(p2); // using the copy constructor to initialise p3=p2
p1=CPoint2D<int>(24,8); // using assignment operator and
// constructor on right side
p1=p3; // using the assignment operator
cout << "p1= " << p1 << endl; // streaming of a CPoint2D
p1 += (p2 + CPoint2D<int>(10,10)); // += and + operator
cout << "Distance between " << p2 << " and " << p1 << " is "
<< p2.GetDist(p1) << endl; // using the GetDist method
```
6.2 Two and three dimensional points and vectors

The CPoint2D and CPoint3D classes are only available in the C++ part of the library, in C there are only the simple types T2DInt, T2DFloat, T3DInt and T3DFloat and no functions is available for working directly with these types like the example given above.

Since version 1.40 of IPL98, we have added template conversions to the two point classes. It was a problem working with different types of points and not being able to convert the types on the fly. For instance working with both float and double types caused the problem. Here are some examples on how that works:

```cpp
void Test(const CPoint2D<int>& P)
{
    // value in P is (3,7) since PFLOAT32 has been truncated by the casting
    cout << "in Test():" << P << endl;
}

void Test2(CPoint2D<int>& P)
{
    cout << "Does nothing" << endl;
}

CPoint2D<int> Test3()
{
    return CPoint2D<int>(2,3);
}

int main()
{
    CPoint2D<double> Pdouble(1.8,2.8);
    CPoint2D<FLOAT32> PFLOAT32((FLOAT32)3.7,(FLOAT32)4.9);
    CPoint2D<UINT16> PUINT16(1,1);

    Test(PFLOAT32); // make an on the fly copy of PFLOAT32 object
                   // converted to an int - new feature!

    Test2(PFLOAT32); // error, cannot make a copy since it is by
                      // reference and type not the same

    PFLOAT32=Test3(); // ok, but normally produces a warning -
                       // new feature!

    PUINT16 += PFLOAT32; // ok, but normally produces a warning
                          // - the user must be aware if the conversion
                          // gives loss of data - new feature!

    PFLOAT32 = PUINT16; // ok - no problem going from UINT16 to
                         // FLOAT32 - new feature!

    PUINT16 = (UINT16)4*PUINT16; // ok, uses the friend function for
                                 // operator *
```
Two vector classes named `CVector2D<T>` and `CVector3D<T>` are derived from the `CPoint2D` and `CPoint3D` classes are also available. They add a few extra methods such as the dot product and modulus.

### 6.3 Timer and date

The class `CTimeDate` can be used for two purposes: As a timer or simply to get the time and date. When optimizing a program, this timer functionality is very useful for detecting where most time in the program is spend. The timer is started with the method `StartTimer()` and stopped with `StopTimer()`, split time can be taken with `GetTimer()`. The timer can be set on hold with `HoldTimer()` and continued with `ContinueTimer()`. When receiving the timer from either `StopTimer()` or `GetTimer()` the result is in seconds. The method `SecondsToHMS(unsigned long Seconds, string& str)` can convert the seconds to hours, minutes and seconds and return the result in the parameter `str`.

For getting the time in the format HH:MM:SS use the method `GetTime(string& str)`, for the date use `GetDate(string& str)`. The method `GetDateTimeYear(string& str)` gives all the year, date and time information in one string. Here is an example showing how to use the class:

```cpp
CTimeDate td1, td2;
string str;
unsigned int x, y;
ld1.StartTimer();
td1.GetDateTimeYear(str);
cout << "This program was run at: " << str << endl;

cout << "Doing first calculation..." << endl;
for(x=0;x<20000;x++){for(y=0;y<4000;y++){}}
cout << "Time elapsed after first calculation: " << td1.GetTimer() << " seconds" << endl << endl;

td2.StartTimer();
cout << "Doing second calculation..." << endl;
for(x=0;x<40000;x++){for(y=0;y<8000;y++){}}
cout << "Time elapsed for second calculation: " << td2.GetTimer() << " seconds" << endl;
cout << "Total time elapsed: " << td1.GetTimer() << " seconds" << endl;
```

---

4The class string is part of the standard C++ library (std).
6.4 Error handling

This section explains how the programmer can react on errors from IPL98 and how error messages produced by IPL98 looks. IPL98 does not yet support exception handling, we hope to find time to implement this in a future version.

Return values

All advanced functions and methods which can result in errors returns a boolean value. Methods and functions not categorized as advanced are the ones returning simple values such as UINT32 GetPixel(). The return values from the advanced functions and methods can be used by the programmer to recover from the error. A simple example is given here:

CImage Img; if (Img.Load("c:/temp/doggie.bmp")) {
   // do some image processing
} else {
   cout << "Could not find the file c:/temp/doggie.bmp" << endl;
}

This example prevents the image processing code to start on an empty object.

6.4.1 Errors produced

Error or warning messages are produced for all functions and methods if an operation for some reason is impossible. Only exception from this rule is a few optimized functions and methods where no error checking is done to speed up the execution time, for instance the GetPixelFast(...) which does not validate if the given position is inside the image dimensions. An error message should result in a false return value from the function/method. Warning messages are intended for situations where the function/method can still proceed but the user probably wanted something else to happen. Warning messages may or may not result in a false return value.

In some cases it is desirable to turn of warnings and/or errors. This can be done in the beginning of the file ipl98_setup.h located in the folder ipl98/. Find the area looking like this:

/* Enable or disable messages produced by IPL98 */

/*#define DISABLE_IPL_ERROR*/ /* ... */
/*#define DISABLE_IPL_WARNING*/ /* ... */
/*#define DISABLE_IPL_NORMAL*/ /*.... */
...

To disable errors simply remove the comment tokens /* and */ at the line defining DISABLE_IPL_ERROR. Do the same thing for warnings at the next line. The DISABLE_IPL_NORMAL definition is not used yet.

An example of the error/warning messages produced by IPL98 is shown in this example:
CImage Img(10,10,24);
CImage Img2;
Img2.CopySubImage(Img,0,0,10,20); // wrong dimension

This program tries to copy a sub image from the original image with dimension 10 times 10
but the sub image dimension is out of range. The resulting error message looks like this:

IPL_ERROR: k_CopySubImage() Illegal clipping rectangle
(d:\ipl98\source\ipl98\kernel_c\image\kernel_functions.c line
1491)

IPL_ERROR: CStdImage::CopySubImage() Failed copying subimage
(d:\ipl98\source\ipl98\cpp\std_image.cpp
line 263)

Normally two errors are produced when using the C++ methods: One from the kernel
function beginning with k_ and one from the method that the programmer used. The kernel
error is produced to show the programmer where the error checking was done and the second
is printed to show the programmer exactly what call in his code that caused the error. Errors
always begins with the prefix IPL_ERROR and warnings with IPL_WARNING. This is followed
by the functions/method name. In case of a method the class name is also included like the
example above where CStdImage:: is proceeding CopySubImage. After this the message text
is shown and often the parameters given are included in this text to give an idea about the
problem. In parentheses at the end of the error message is the exact location of the code
producing the error. This is very useful if the programmer want to take a closer look at the
IPL98 code. It is also recommended if the programmer finds a bug in IPL98 to send this error
information to the person responsible for developing the library.

Errors and warnings are as default written to the standard out device but it is sometimes
desirable to redirect the messages to somewhere else, for instance when working with
a GUI application. This is simply done by compiling the project ipl98library_gui located in
folder ipl98/projects/borlandbuilder/ipl98library_gui for the Borland C++ Builder compiler
and in folder ipl98/projects/visualc/ipl98library_gui for the Visual C++ compiler. These two
projects ensures that the function:

void AddIPL98ErrorHistory(const char* str)

is not defined by the library. You can then define this function in your own code and
redirect the provided string str. For the GCC compiler, you will need to redefine the existing
function found in the file ipl98/source/ipl98/ipl98_setup.c.

6.5 History

Unfortunately, the History part of the library is not very well supported yet. But here is an
introduction to it, we hope to find time to add history support from all algorithms at a later
time.

5Here the IPL98 library is placed in d:/ipl98
6Find information about this at the IPL98 home page at http://www.mip.sdu.dk/ipl98
6.5 History

When working with the image classes a history of type CText is present as an attribute in them and can be accessed as m_History. Every time an algorithm is performed on an image information about this is appended by the algorithm to the history attribute. This history is only saved when the programmer saves an image in the PGM format, see section 7.3 for more information. The history can also be saved manually by the method Save(...) in the CText class. The history can be disabled in the beginning of the file ipl98.h located in the folder ipl98/ which looks like this:

```cpp
/* Enable or disable adding to history from the ANSI C and C++ part of the library. To disable adding history
out comment the line "#define IPL_ADD_HISTORY". */
#define IPL_ADD_HISTORY
```

Simply outcomment the `#define IPL_ADD_HISTORY` line to disable the history, in that case all code involving the history will not be compiled.

Lines are added with the method `Append(const string& str)` or `AppendIPL(const string& str)`, the last mentioned adds the IPL98 version number to the string before insertion. The history can never be totally emptied, the method `Empty()` removes all lines but leaves one line with information about the number of lines removed. In this way it is always possible to see if some processing has been done on the image. The methods `AddTime(bool Value)` and `AddDate(bool Value)` can be used to disable or enable automatic appending of date and time information when adding lines to the history. The following example shows how the methods above may be used. In addition it show how to search through the history with the methods `Find(...)` and `Next(...)`. 

```cpp
CImage Img(10,10,8);  
CCoordinateTransform ct;  
ct.ScaleAndRotateAuto(Img,Img,2,4.7,PI/4.0,0);  
cout << Img.m_History << endl;  

// appending your own history  
Img.m_History.AddTime(false); // do not add time  
Img.m_History.AddDate(true); // add date  
string str("my own algorithm running with x=5");  
Img.m_History.Append(str);  
Img.m_History.AppendIPL(str); // append it with IPL version number
```

```cpp
cout << "The history now contains " << Img.m_History.TotalItems() << " lines:" << endl;  
cout << Img.m_History << endl;  

cout << "Searching for \"own\" in the history" << endl;  
string result;  
if (Img.m_History.Find(string("own"),result))  
{  
cout << "Found \"own\" in this line: " << endl;  
cout << result << endl;
```
cout << "Searching for other occurrences.... " << endl;
while(Img.m_History.FindNext(result))
{
    cout << "Found \"own\" in this line: " << endl;
    cout << result << endl;
}
}
Img.m_History.Empty();
cout << "This is how the history looks after emptying it: " << endl;
cout << Img.m_History << endl;
7 Using IPL98 to represent and interact with images

The container classes are the core of IPL98 used to hold image information in memory and giving the user an easy and fast access to the data contained in it. An overview of the container classes provided is given in figure 8.

![Diagram of container classes](image)

Figure 8: The hierarchy of container classes provided by IPL98. The CPixelGroup is a special class (and is tightly connected with another class named CPixelGroups) whereas the other classes are the normal matrix representation of images. The classes CGenericImage and CStdImage is virtual classes. CGenericImage gives a basic shared interface for all image classes whereas CStdImage gives an interface for images to be directly interpreted as “visual” images, that is each pixel value contains color information.

The image classes are all classes derived from CGenericImage, they all contain two dimensional image data. The most used image class is the CImage class, which contains normal images, i.e. images that can actually be shown on a screen. A more specialized class is the CByteImage, which is a bit more efficient but still holds normal image data. They are both derived from the base class CStdImage. Most methods operating on an image, will have the CStdImage as an argument, this means that you can use all algorithms no matter if you have an instance of type CImage or CByteImage. CIntImage, CFloatImage and CComplexImage are some special image classes. They hold different kinds of image data which will be explained in the next section.

All image classes are derived from a virtual base class named CGenericImage, i.e. you cannot instantiate an object of that type. This is done to ensure a common interface for the derived classes. Since these classes are not template based, the base class can only define a subset of the functions that are common for all the classes. All the image are built on top of a kernel ANSI C implementation.

The class CArray2D<T> is a template class working essentially as the other image classes, except it is template based. This mean that you can use any type T as the data contained
7.1 Creating an image in RAM

There are two different ways of representing image data in RAM. A class named CPixelGroup keeps a list of positions constituting the pixels of interest (not belonging to the background of the image) and each of these positions can optionally have a color associated with it, more details about this class are given in section 7.6. The other classes for working with an image have an internal matrix representation corresponding to how we normally think about an image. Different classes with this matrix representation is available and they have both advantages and draw backs. The major difference is the number of bits used to represent each pixel and if it is signed or unsigned values. Here is a table of the available image matrix classes:

<table>
<thead>
<tr>
<th>Class name</th>
<th>Pixel type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CImage</td>
<td>1, 8 or 24 bits per pixel unsigned</td>
</tr>
<tr>
<td>CByteImage</td>
<td>8 bits per pixel unsigned</td>
</tr>
<tr>
<td>CIntImage</td>
<td>16 bit signed integers</td>
</tr>
<tr>
<td>CFloatImage</td>
<td>32 bit floating point</td>
</tr>
<tr>
<td>CComplexImage</td>
<td>2 * 32 bit floating point</td>
</tr>
</tbody>
</table>

The first thing to consider when creating a new image in RAM is the type used to represent a pixel. CFloatImage (with floating point pixel values), CComplexImage (with two floating points for each pixel representing the real and imaginary part) and CIntImage (with signed integer values) are special classes available for holding pixel values which cannot be directly interpreted as colors. These classes are useable when doing internal image processing, for instance subtraction of one image with another may result in negative values and it is therefore desirable to put the result in a CIntImage class. The CFloatImage class can for instance be useful in case of some pixel transformations which results in non integer values. But normally a simple image with unsigned values and in some cases a corresponding palette will be the most convenient. These classes are also possible to show on screen and save to standard graphics file formats.

The most general image is the CImage class which can hold three different types of pixels, 1, 8 and 24 b/p\(^7\) (only one at a time though). The number of bits used to represent a pixel is called the bit depths or bits per pixel (b/p). A specialization of the CImage class is named CByteImage is also available. This class can only represent pixels with 8 bits. The advantage of CByteImage is the speed, it is faster to access pixels and the overall execution times will probably be considerably faster when working with a CByteImage. If you are not sure about what types of images you will be working on, it is recommended to begin with the CImage.

\(^7\)The class is prepared to include 4 bits per pixel if it turns out to be desirable in the future.
7.1 Creating an image in RAM

class and then at a later point replace it with a CByteImage if necessary. Because of the class
hierarchy most methods are common for the two classes and it is therefore an easy task to
replace a CImage type with a CByteImage type. When creating an image with 1 or 8 b/p a
gray tone palette will be associated with it, read more about this in section 7.5.

When creating an instance of a class (an object) you can choose to allocate memory for the
image area, copy another image into the new object or just initialize the object with no image
(i.e. no memory allocation for image data). Here is how its done in C++, the corresponding
C code can be found in the appendix on page 83:

```c++
#include <ipl98/cpp/image.h>
#include <ipl98/cpp/float_image.h>
#include <ipl98/cpp/int_image.h>
#include <ipl98/cpp/byte_image.h>
using namespace ipl;

void main()
{
    CImage Image1; // an empty image
    CImage Image2(768,512,8); // image with dimensions (768,512) and 8 b/p
    CImage Image3(768,512,8,0); // image with dimensions (768,512) and 8 b/p
    // image data initialized to 0
    CImage Image4(Image2); // the content of Image2 is copied into Image4

    CByteImage ByteImage1; // an empty image
    CByteImage ByteImage2(768,512); // image with dimensions (768,512),
    // for a CByteImage the pixel depths is 8
    CByteImage ByteImage3(768,512,0); // image with dimensions (768,512)
    // image data initialized to 0
    CByteImage ByteImage4(ByteImage2); // the content of ByteImage2 is
    // copied into ByteImage4

    CFloatImage FloatImage1(10,10); // float image with dimensions (10,10)
    CFloatImage FloatImage2(10,10,0); // image with dimensions (10,10)
    // image data initialized to 0
    CFloatImage FloatImage3(FloatImage1); // the content of FloatImage 1 is
    // copied into FloatImage3

    CIntImage IntImage1(10,10); // int image with dimensions (10,10)
    CIntImage IntImage2(10,10,0); // image with dimensions (10,10)
    // image data initialized to 0
    CIntImage IntImage3(IntImage1); // the content of IntImage1 is
    // copied into IntImage3
}
```

As can be seen from the example above the code for each type of image is almost the
same. For each class you need to include the correct header file. The easiest way to find
7.1 Creating an image in RAM

Figure 9: The exact include-statement is shown on each documentation page for each class (or function).

This file is from the code documentation pages found on the IPL98 home page by clicking the C++ overview button in the left pane (or the C overview button for the C kernel only). Now click on the class you need to include and simply copy the include statement found on top of the page. In figure 9 is shown an example for the CImage class. The path following the 
#include-statement is only correct, if you set up the compiler as described in chapter 5.

To simplify the examples given in the remaining part of this chapter main focus will be put on the CImage class, converting the examples to work with other image classes should be a relatively easy task. The #include and using namespace pre-compiler directives will also be excluded from the examples. Sometimes the main function name and the scope will also be omitted.

The memory allocated for the data stored in an image can be freed at any time by the method Empty(). Note that it is not necessary to empty an image when leaving a scope, this will be done by the class destructor. After using this method the dimensions of an image will be set to (0,0). An image can also be reallocated with different dimensions at any time with the method Alloc(...) which takes care of deleting old allocated image memory. To get information about an image, several methods are available, here is an example:

```c++
void main()
{
    CImage Img(768,512,24);

    // examples on accessing some specific information
    cout << "Image dimensions: Width=" << Img.GetWidth() << " Height=" <<
         Img.GetHeight() << endl; // image dimensions
    cout << "Bits per pixel (b/p): " << Img.GetBits() << endl; // b/p info
    cout << "Origo is set to: " << Img.GetOrigo() << endl; // origo info

    Img.Alloc(1024,768,8);
    // printing all relevant information for an image
    cout << endl << Img << endl;
}
```
7.2 Borders

As showed in the example specific information can be retrieved from an image. It is also possible with the streaming operator \texttt{\textless\textgreater{}} to get all image information with just one line of code. The output from the streaming operator looks like this:

*************** Image info ****************
Width=768 Height=512 ByteWidth=2304 SizeImage=1179648 Bits=24
PalEntries=0 BorderSize=0 Origo=(0,0) Origin=RAM

As showed in the example specific information can be retrieved from an image. It is also possible with the streaming operator \texttt{\textless\textgreater{}} to get all image information with just one line of code. The output from the streaming operator looks like this:

*************** Image info ****************
Width=768 Height=512 ByteWidth=2304 SizeImage=1179648 Bits=24
PalEntries=0 BorderSize=0 Origo=(0,0) Origin=RAM

With all image classes is a border associated which can be dynamically resized. A border is convenient for instance when using a filter mask on a whole image, if the border is set to half the size of the mask and has the same color as the background of the image the programmer does not have to take special care of the problem with accessing pixel positions outside the image when the mask is near the edge of the image. Here is an example on how to manipulate the border of an image:

```cpp
void main()
{
    CImage Img(768,512,8);
    Img.SetBorderSize(5); // set a border of size 5
    Img.SetBorderColor(0); // set color of border to 0 (black)

    // get the actual size and print it to stdout
    cout << "Size of border is " << Img.GetBorderSize() << endl;

    Img.SetBorder(10,255); // set border of size 10, color 255 (white)
}
```

---

8 Every scanline must be a multiple of 4 bytes, ie. 32 bit alignment is needed for each scanline.
7.3 Save, load and show an image

The two image classes CImage and CByteImage can be loaded from and saved in two standard file formats\(^9\): BMP (Windows Bitmap) and PGM (Portable Gray Map). The BMP is a standard format which is recognized by most image related applications and is build to be easy and fast for MS Windows to handle. It may include an associated palette (this is the case when working with 1 and 8 b/p images) and in that case the pixel values are indexes in this palette. If no palette is present the pixel values consists of four bytes where three of these bytes is the RGB value. The PGM format is supported for two reasons:

1. The image information and pixel data is stored in ASCII format and therefore easy to read for other applications such as MATLAB and Mathematica.

2. Comment lines can be included in the top of the file. This means that image information can be stored directly in the file. An example is information about time, date, camera settings and camera type used for acquiring the image.

The drawback of PGM is that it only supports gray tone images and at the moment only 8 b/p can be used with this format. More details about the graphics formats can be found on the World Wide Web at: http://www.wotsit.org.

To decide what file type an image is to be stored as simply use the correct extension in the given filename, the two methods Load("filename") and Save("filename") are used to handle the loading and saving of images:

```c
void main()
{
    CByteImage Img(768,512); // byte image, always 8 b/p
    CImage ImgDest(10,10,24); // image with 24 b/p

    Img.Save("c:/temp/testimage.bmp"); // save Img in BMP file format
    Img.Save("c:\temp\testimage.pgm"); // save Img in PGM file format

    cout << "ImgDest before loading: " << endl << ImgDest << endl;
    ImgDest.Load("c:/temp/testimage.bmp"); // old image destroyed
    cout << endl << "ImgDest after loading: " << endl << ImgDest << endl;
}
```

Both slash (/) and back slash (\) can be used in the path as shown in the example but remember to escape the back slash, otherwise C/C++ will treat the following character as a special symbol.

The two remaining CFloatImage and CIntImage classes cannot be saved in a normal graphic file type format. A simple ASCII format have been chosen and here is a small sample program showing a CIntImage file format:

\(^9\)More formats may be available in future versions
7.3 Save, load and show an image

```cpp
void main()
{
    CIntImage IntImg(5,3); // integer image
    IntImg.Save("c:/temp/testimage.txt"); // save image as ASCII text
    cout << IntImg << endl;
}
```

This program produces a file containing this:

```
# Int image written by IPL98 version 1.01
## WidthHeight 5 3
## RowsCols 3 5
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

Here is an example with the CFloatImage:

```cpp
void main()
{
    CFloatImage FloatImg(4,3); // integer image
    FloatImg.SetPixel(1,0,598694868);
    FloatImg.SetPixel(2,0,-21e-5);
    FloatImg.Save("c:/temp/testimage.txt"); // save image as ASCII text
    cout << FloatImg << endl;
}
```

This program produces a file containing this:

```
# Float image written by IPL98 version 1.01
## WidthHeight 4 3
## RowsCols 3 4
0.000000e+000 5.986948e+008 -2.100000e-004 0.000000e+000
0.000000e+000 0.000000e+000 0.000000e+000 0.000000e+000
0.000000e+000 0.000000e+000 0.000000e+000 0.000000e+000
```

Lines with comments must begin with a single "#" character and can be placed anywhere before the image data. Lines beginning with two "#" characters are treated as special information and must be followed by a space and a token. At present two tokens with the same information exists: WidthHeight and RowsCols. At least one of them must be present, if both are present IPL98 will produce an error if they do not contain the same dimensions. After these lines is the image data separated by white spaces and each row ended by a \n (newline) character. This format is intended to be easy to work with for other programs.

As mentioned before methods for showing images is not included in the library because it is hardware dependent, see section 4 on page 5 about where to obtain more information. But even without a method for showing images it is often enough to save the image after some processing and check the result in a standard image application.
7.4 Manipulating an image in RAM

One of the first things to do when programming with images is to move the contents of one image object to another. In IPL98 this is done with the “=” (assignment) operator and is very easy. Here is a simple example:

```csharp
CImage Img1(768,512,24);
CImage Img2;

Img2=Img1; // copy contents of Img1 to Img2
```

The assignment operator can only be used between images with the same pixel depth (b/p), it is possible to copy between a CImage and a CByteImage class as long as the pixel depth (b/p) is available in the destination image:

```csharp
CImage Img1(768,512,24);
CImage Img2(768,512,8);
CByteImage ByteImage;

ByteImage=Img1; // error, a CByteImage cannot contain a 24 b/p image
ByteImage=Img2; // OK, Img2 contains an image with 8 b/p

Img1=ByteImage; // OK to copy a CByteImage to a CImage object
```

To change the bit depths (b/p) of an image another method is available in both the CImage and CByteImage class: CopyConvert. This function takes two parameters, the first is the resulting bit depths and the second is the source image which can be both a CImage or CByteImage. Some information will be lost when converting to a lower bit depths. Conversion from 1 to 8 b/p assumes black as 0-values and 1 as white values in the source image. Conversion from 1 to 24 b/p and 8 to 24 b/p looks up the palette value in the source image and sets it in the destination - hence it works for both gray tones and color images. For 8 to 1 b/p a threshold of 128 is used (should be used for gray tone images only). For 24 to 8 b/p the mean value of the R, G and B component is used, i.e. the new image is a gray tone. For 24 to 1 b/p the method for 24 to 8 b/p is used followed by a threshold 128. When using the method with a CByteImage the resulting bit depths has to be 8. Here is an example of conversions, the corresponding C code can be found in the appendix on page 84:

```csharp
CImage Img1(768,512,24);
CImage Img2(768,512,8);
CByteImage ByteImage;

Img2.CopyConvert(1,Img1); // copy Img1 to Img2 and convert it from
// 24 b/p to 1 b/p

cout << "Contents of Img1" << endl << Img1 << endl << endl;

Img1.CopyConvert(1,Img1); // convert Img1 from 24 b/p to 1 b/p this is
// a bit slower

cout << "After converting Img1: " << endl << Img1 << endl;

ByteImage.CopyConvert(8,Img1); // copy Img1 to ByteImage and convert
// to 8 b/p.

ByteImage.CopyConvert(1,Img1); // error, CByteImage needs 8 b/p!
```
A method named `CopySubImage` copies an area of a given image to a new image as this example shows:

```cpp
CImage Img1(768,512,8);
CByteImage ByteImage;
ByteImage.CopySubImage(Img1,10,10,100,50); // Copy a rectangle with
// dimensions (90,40) from Img1 to ByteImage
```

When using `CopySubImage(...)` both the source and destination can be the same image but the method will be slower because an internal copy of the image has to be allocated.

### 7.5 Working with pixels

Accessing positions in an image and reading their color values is probably the most essential issue when working with image processing. A bit of knowledge is needed before describing the methods for pixel access.

Let's focus on the `CImage` class. The pixel values stored in the image can have two meanings: In case of 24 b/p the returned value consist of 3 bytes containing the RGB-value, i.e. the color is stored directly in this value. For 1 and 8 b/p images the pixel value contains indexes in a palette. The returned value is for a 1 b/p 0 or 1 and for 8 b/p in the range 0-255, the latter also valid for the `CByteImage` class. To get the corresponding color it is necessary to look this index up in a corresponding palette. When creating an image with 1 or 8 b/p a default palette is constructed. For 1 b/p index 0 is (R,G,B)=(0,0,0) which is black and index 1 is (R,G,B)=(255,255,255) which is white. For 8 b/p it is a gray tone palette where the index equals each R, G and B value in the palette, for instance palette index 76 has (R,G,B)=(76,76,76). So when working with a 8 b/p gray tone image the index can be used directly as the color information knowing that the R, G and B value is the same as the index. In the case of 1 b/p the value 1 can simply be interpreted directly as white and 0 as black.

#### 7.5.1 The palette

The palette is only present in the `CStdImage` class (that means accessible from both `CImage` and `CByteImage`) and can be accessed through the attribute `m_Pal` of type `CPalette`. The `CPalette` class contains a complete set of methods needed for manipulation of a palette. When accessing a pixel value in an image with a palette a look up in the palette attribute `m_Pal` at the returned index is necessary to get the RGB color value. The color value stored in the palette is always a `UINT32` type and is obtained with the method `GetColor(...)` to extract each R,G and B component use the static methods `GetRedVal(...)`, `GetGreenVal(...)` and `GetBlueVal(...)`. Here is how to do it:

```cpp
CImage Img1(768,512,8); // An image with a default gray palette is created
Img1.Flush(10); // set all pixels to palette index 10
UINT8 Index=Img1.GetPixel(0,0); // this is how to access pixels
UINT32 Color;
if (Img1.m_Pal.GetColor(Index,Color))
{
    cout << "Palette value at index " << (short)Index << endl;
}
```
7.5 Working with pixels

```cpp
cout << " Red=" << (short)(CPalette::GetRedVal(Color)) << endl;
cout << " Green=" << (short)(CPalette::GetGreenVal(Color)) << endl;
cout << " Blue=" << (short)(CPalette::GetBlueVal(Color)) << endl;
}
else
{
    cout << "Index=" << Index << " is out of range" << endl;
}
```

A default gray palette is created and the pixel values is set to 10 by the `Flush(...)` method. The method `GetPixel(...)` in the second line will be explained later. The `GetColor(...)` call is out of range in the palette. If it succeeds (which it should in this example) the R,G and B components is extracted from the Color variable. When using the streaming operator remember to convert the UINT8 type returned by `GetRedVal(...)`, `GetGreenVal(...)` and `GetBlueVal(...)` to an integer type. Otherwise it will be treated as a character (UINT8 is defined as unsigned char).

### 7.5.2 Pixel access

As default the coordinate system is placed with origo in upper left corner. This can be changed by the method `SetOrigo(x,y)` and is available in the classes CImage, CIntImage and CFloatImage but not in the CByteImage. Because of the possibility of changing the origo a set of methods is available for range checking, they are named: `GetMinX()`, `GetMinY()`, `GetMaxX()` and `GetMaxY()`. It is recommended to use these methods when iterating over a whole image, an example is given below.

At this point, I should also introduce the Region Of Interest (ROI). This is a rectangle in an image which can be set to be the active area. Alternatively, you can cut out this region and copy it into another image, but this takes time! If you set a ROI, the methods `GetMinX()`, `GetMinY()`, `GetMaxX()` and `GetMaxY()` returns boundary values according to this ROI. The iterators introduced later, will also be limited to work on the active ROI, if present.

There are several ways of reading and setting pixel values. The most general is `SetPixel(x,y,Color)` and `GetPixel(x,y)`. These functions produces an error if the position (x,y) is out of range. A faster set of methods named `SetPixelFast(x,y,Color)` and `GetPixelFast(x,y)` is available but they do not include a range check.

Here is an example involving the most used methods:

```cpp
CImage Img1(5,255,8); // An image with a default gray palette is created
cout << "Created image:" << endl << Img1 << endl;
Img1.SetOrigo(-10,5); // setting origo to (-10,5)
cout << "Origo set to: " << Img1.GetOrigo() << endl;
// the boundaries for the loops would be incorrect if not using
// the range methods GetMinX/Y and GetMaxX/Y. If a ROI is active
// these loops will take that into account
for(unsigned int y=Img1.GetMinY();y<Img1.GetMaxY();y++)
{
    for(unsigned int x=Img1.GetMinX();x<Img1.GetMaxX();x++)
    {
```
7.5 Working with pixels

```cpp
Img1.SetPixel(x, y, y); // setting each scanline to same
// color as y value
```
7.5 Working with pixels

// image with dim. (15,7), 8 b/p and initialized to 22
CImage Image(15,7,8,22);
int RowNr=1;
// do something to fill image data into Image ...
CImage::ConstRowIterator8bp it=Image.ConstRow8bp(RowNr);
// this example writes all pixel values in rownr
// to standard out stream
while(it!=it.End())
    // must convert to int, otherwise output is
    // written as char (the definition of UINT8)
    cout << (int)*it++ << " ";

To use an iterator you need to know what bit depth your image has, the iterators you can
choose from are:

- CImage::RowIterator1bp
- CImage::ConstRowIterator1bp
- CImage::ColumnIterator1bp
- CImage::ConstColumnIterator1bp
- CImage::RowIterator8bp
- CImage::ConstRowIterator8bp
- CImage::ColumnIterator8bp
- CImage::ConstColumnIterator8bp
- CImage::RowIterator24bp
- CImage::ConstRowIterator24bp
- CImage::ColumnIterator24bp
- CImage::ConstColumnIterator24bp

All these types have a corresponding method in the CImage class, which will return an iterator
of that type initialized to the beginning of the row (left) or column (top). The iterator it in
the example above, is a const row iterator. This means, that I cannot change the value of the
iterators contents. For instance:

CImage Image(15,7,8,22);
CImage::ConstRowIterator8bp it=Image.ConstRow8bp(0);
// the following gives a compiler error
// since it is const!
*it=1;

To make this work, we need to use a non-const iterator:
7.5 Working with pixels

CImage Image(15,7,8,22);
CImage::RowIterator8bp it=Image.Row8bp(0);
// the following gives a compiler error
// since it is const!
*it=1;

Iterators for the CImage class are more complicated than iterators for the other container classes. Since the other container classes holds only one type (for the CArray2D and CArrayOS2D they hold one type only after you instantiate an object), only one type of iterators are necessary. Here is an example using the ClutImage:

// image with dim. (15,7), initialized to 22
CIntImage Image(15,7,22);
tt RowNr=1;
// do something to fill image data into Image ...
CIntImage::ConstRowIterator it=Image.ConstRow(RowNr);
// this example writes all pixel values in rownr
// to standard out stream
while(it!=it.End())
    cout << *it++ << " ";

Here you only need to choose between the following iterators:

- CIntImage::RowIterator
- CIntImage::ConstRowIterator
- CIntImage::ColumnIterator
- CIntImage::ConstColumnIterator
- CIntImage::RowIterator

To iterate a whole image, you need to iterate one column or row at a time. Here is the example from page 36 rewritten to use iterators:

CImage Img1(5,255,8); // An image with a default gray palette is created
cout << "Created image:" << endl << Img1 << endl;
Img1.SetOrigo(-10,5); // setting origo to (-10,5)
cout << "Origo set to: " << Img1.GetOrigo() << endl;
// the boundaries for the loops would be incorrect if not using
// the range methods GetMinX/Y and GetMaxX/Y. If a ROI is active
// these loops will take that into account
CImage::RowIterator8bp it=Img1.Row8bp(0);
for(unsigned int y=Img1.GetMinY();y<Img1.GetMaxY();y++)
{
    it=Img1.Row8bp(y);
    while(it!=it.End())
    {
        *it=y; // setting each scanline to same color as y value
7.6 Working with a pixelgroup

```cpp
} }
cout << endl << "working with 24 b/p" << endl;
Img1=CImage(10,10,24);
Img1.Flush(ipl::CPalette::CreateRGB(10,20,30)); // set all image to
// (R,G,B)=(10,20,30)
// getting the R components directly from the image
cout << "Red component at (0,0): " <<
     (short)Img1.GetRedComponent(0,0) << endl;
Img1.SetRedComponent(0,0,100); // Setting the Red component directly
cout << "Red component after setting it: " <<
     (short)Img1.GetRedComponent(0,0) << endl;
```

The above example also shows how to iterate a whole image, which may not be very obvious when you start out using the iterator functionality.

We hope in the future to change all algorithms in the IPL98 library to use iterators. This will make it possible to set the ROI just before calling an algorithm. Unfortunately this will take some time, so I cannot promise when this will be fully implemented. At least new algorithm additions should work like this, but for further information, read the documentation for each class or method to see if it supports setting the ROI of an image.

7.5.4 Simple image manipulation

A few simple transformations of the image have been included directly in the image classes. More advanced methods are placed in the algorithm section. The transformations are:

1. **FlipHorizontal()** Mirrors an image around the x-axis with the x-axis placed in the vertical middle of the image.
2. **FlipVertical()** Mirrors an image around the y-axis with the y-axis placed in the horizontal middle of the image.
3. **Invert()** All pixels in the image is inverted, works with all pixel depths.
4. **Rotate90(int steps)** Rotates the image in steps of 90 degrees, for instance two steps turns the image upside down.

7.6 Working with a pixelgroup

The pixelgroup class named CPixelGroup is a different way of storing image information. It contains a list of positions and can optionally have an associated color stored for each position. It also keeps track of the positions defining the boundary rectangle of all the positions in the group. The advantage of storing image information in this class is:

- Only pixel values in interest uses memory.
- Faster access to the relevant pixels, i.e. iterating through non interesting pixels (pixels belonging to the background) is avoided, an example is the moment algorithm found in the class CFeatures explained in section 8.3 page 52.
7.6 Working with a pixelgroup

The drawback is that an image cannot be shown directly from a pixelgroup, it has to be converted to a CImage or CByteImage first.

A new empty CPixelGroup can be created by using the default constructor. If the programmer has an idea about the total number of positions that will be stored in the group it is faster to initialize the pixelgroup with the constructor CPixelGroup(InitialSize). In this way the pixelgroup can allocate memory in one chunk which saves time. If the number of positions exceeds this InitialSize the group will automatically expand the memory allocated. To do this as fast as possible the group always allocates twice the number of positions which was previously allocated. This may cause quite a lot of unused allocated memory in case of groups with many positions, for instance a group with 1038 positions in it may have allocated 2048 positions. This extra memory allocation can be freed again with the method AdjustSizeOfPositions(), after calling this method only 1038 positions will be allocated in the previous example.

Adding and removing positions can be done in two ways. The slow methods are InsertPosition(...) and RemovePositionSlow(...), they both preserve the ordering of the positions. The fast methods are AddPosition(...) which always appends a new position at the end and RemovePosition(...) which removes a position at a given index by moving the last position in the group to that index.

Adding colors to a pixelgroup

Adding colors are normally done by providing an image (typically the image from where the positions came) in the method AddColors(const CStdImage& Img). Colors can also be added manually with the methods AllocColors() followed by calls to AddPositionColor(...) or InsertPositionColor(...). They can be removed at any time by the method RemoveColors(). To copy a pixel group to an image, two methods are available: CopyToImage(const CPalette* pPalette, UINT32 BackGround, CStdImage& Dest) and AddToImage(...). The first allocates an images which is exactly the size of the blob and plots the blob in it, whereas the latter copies the group to an existing image. Using the method CopyToImage(...), there are three cases which decides the bit depth of the output image:

1. No colors available (i.e. the method AddColors() has not been called), then a black and white image with 1 b/p is produced and the positions are plotted in the image according to the "Color" member of the TPixelGroup-structure. If the Color-type is undefined the default is a white background with black pixels set at the given positions.

2. Colors available but no palette is given (i.e. Palette is a NULL pointer), then an image with a graytone palette is produced if the Bits pixel depth member of CPixelGroup is 1 or 8. If the pixel depts is 24 b/p an image with 24 b/p without a palette is produced.

3. Colors available and a palette is provided. An image with pixel depth given in CPixelGroup and the given palette is produced. The pallette must correspond with the pixel depth in the TPixelGroup, otherwise an error message will be produced.

Individual colors can be set and read in the associated colors (if they are available, use ColorsAvailable() to check) with the methods GetColor(...) and SetColor(...). Origo of the new image is set to (-Left.x,-Top.y) which is the pixelgroups top left boundary. This means that a point (x,y) in the original image will correspond to the same position (x,y) in an image created from a pixelgroup, that is of course only the case if the pixelgroup was derived
from the original image. The following example extract one blob\(^{10}\) from an image and create a new image consisting of that blob. Then a cross is drawn in both the original image and in the blob image to show that the position values used are the same in both image. Figure 10 shows the original image and the derived blob. Here is the code used to produce the images:

```cpp
CImage Img, ImgDest; // Images
CPixelGroups PixelGroups; // Holds information about blobs
CTimeDate Timer;
Timer.StartTimer(); // stop watch for time elapsed

if (Img.Load("c:/temp/three_segments.bmp"))
{
    cout << Img << endl;

    // Find the blobs in image
    CSegmentate Segm;
    Segm.DeriveBlobs(Img, PixelGroups, HIGHCOLOR, 128, EIGHTCONNECTED);

    // Copy blob number 2 to gray tone image
    PixelGroups.AddColors(Img); // first add colors
    CPixelGroup* pGrp = PixelGroups.GetGroup(2);
pGrp->CopyToImage(&Img.m_Pal, 128, ImgDest);

    int cnt;
    // draw rectangle in original image around blob area
    for (cnt = pGrp->GetLeft().GetX(); cnt < pGrp->GetRight().GetX(); cnt++)
    {
        Img.SetPixel(cnt, pGrp->GetTop().GetY(), 255);
        Img.SetPixel(cnt, pGrp->GetBottom().GetY(), 255);
    }
    for (cnt = pGrp->GetTop().GetY(); cnt < pGrp->GetBottom().GetY(); cnt++)
    {
        Img.SetPixel(pGrp->GetLeft().GetX(), cnt, 255);
        Img.SetPixel(pGrp->GetRight().GetX(), cnt, 255);
    }

    // draw cross in both original image and "blob" image
    int size = 5;
    for (cnt = 230 - size; cnt < 230 + size; cnt++)
    {
        Img.SetPixel(cnt, 220, 0);
        ImgDest.SetPixel(cnt, 220, 0);
    }
    for (cnt = 220 - size; cnt < 220 + size; cnt++)
    {
        Img.SetPixel(230, cnt, 0);
        ImgDest.SetPixel(230, cnt, 0);
    }

Read more about blobs in section 8.1.1.
```

\(^{10}\)Read more about blobs in section 8.1.1.
7.6 Working with a pixelgroup

```csharp
// save original image containing edge around the blob and a cross
Img.Save("c:/temp/test2.bmp");
// save blob 2 to disk (c:/temp/test.bmp)
ImgDest.Save("c:/temp/test.bmp");
cout << "Blob 2 saved as c:/temp/test.bmp" << endl;

cout << "Total time: " << Timer.StopTimer() << endl;
}
else
{
    cout << "Loading failed" << endl;
}
```

Figure 10: An example of a pixelgroup having the same position values as the original image. In a) the original image is shown with a rectangle surrounding the area from which the blob is derived. In b) is shown the pixelgroup copied to a new image. The cross shown in both images is drawn with center at the same position (230,220) in both images.

The streaming operator `<<` and the method `PrintInfo(...)` can be used to get information. From the `<<` operator this is written:

```
**************** PixelGroup info ****************
Positions: Top=(1,1) Bottom=(2,84) Left=(-1,19) Right=(30,12)
NumberOfPixels=5 AllocatedPixels=8
```

The boundary rectangle for this group is upper left (-1,1) and bottom right (30,80), note that the 2D positions defining the edge is stored, that is the leftmost, rightmost, uppermost and bottommost positions. If more than one position defines one of the edges one are picked randomly. `NumberOfPixels` is the total number of pixels store in the group and the `AllocatedPixels` is the total allocated memory for pixel positions, in this example the next three positions
Working with a pixelgroup

will be very fast to insert but the fourth will force the group to allocate a new area with room
for 8 positions. With the PrintInfo(...) method the positions in the group can be printed too.
The first parameter must be true to get positions printed and the second defines how many
positions on each line. The output below was produced by calling PrintInfo(true,3):

*************** PixelGroup info **********************
Positions: Top=(1,1) Bottom=(2,84) Left=(-1,19) Right=(30,12)
NumberOfPixels=8 AllocatedPixels=8

pos 0=(1,1) pos 1=(6,9) pos 2=(30,12)
pos 3=(2,84) pos 4=(-1,19) pos 5=(12,34)
pos 6=(5,1) pos 7=(10,48)

Here is a bigger example showing most of the methods explained:

// create an image with dimensions 10*10 and
// set every pixel to (x+y) for all color components
CImage Img(10,10,24);
for(int y=Img.GetMinY();y<Img.GetMaxY();y++)
{
    for(int x=Img.GetMinX();x<Img.GetMaxX();x++)
    {
        Img.SetPixelFast(x,y,CPalette::CreateRGB(x+y,x+y,x+y));
    }
}
CPixelGroup pg; // a new pixelgroup is created
// add some positions
pg.AddPosition(2,2);
pg.AddPosition(9,9);
pg.AddPosition(2,9);
pg.AddPosition(5,6);
pg.PrintInfo(true,5);

// inserting a position at a given place
// forces the group to allocate memory for a total of 8 positions
cout << endl << "inserting position at index 1 (slow method)" << endl;
pg.InsertPosition(CPoint2D<int>(9,2),1);
cout << "Now the group looks like this:" << endl;
pg.PrintInfo(true,5);

cout << "Adjusting the allocated memory in the pixelgroup" << endl;
pg.AdjustSizeOfPositions();
pg.PrintInfo(true,5);

pg.AddColors(Img);
if (pg.ColorsAvailable)
{
}
CImage Img2;
pg.CopyToImage(NULL, CPalette::CreateRGB(0,0,0), Img2);
// copy pixelgroup to new image (case 2 from // documentation) with black background
cout << "Pixelgroup copied to a new CImage object:" << endl;
cout << Img2 << endl << endl;

// setting a color for the last position stored in the group
if (pg.SetColor(CPalette::CreateRGB(100,0,255),
    pg.GetTotalPositions()-1))
{
    cout << "Color set at index " << pg.GetTotalPositions()-1
    << " in pixelgroup" << endl;
}

// searching for a position - index returned in "Index"
unsigned int Index;
if (pg.PosInGroup(CPoint2D<UINT16>(2,2),Index))
{
    cout << endl << "Position (2,2) found in group" << endl << endl;
}

pg.Empty(); // empty the pixelgroup the same way as with a CStdImage
cout << "Emptying the pixelgroup: " << endl;
cout << pg << endl;

The method PosInGroup(...) used at the end of the example shows how to ask if a given position is present in a group. Before that is shown how to manipulate a single color with the method SetColor(...).

7.7 Container for several pixelgroups

It is often neccesary to work with several CPixelGroup objects and for that purpose a CPixelGroups container class is available. The class works in many ways as the CPixelGroup class. The default constructor starts out allocating memory for 20 CPixelGroup objects and will adjust the size in the same way as the CPixelGroup class does for its positions. The constructor CPixelGroups(InitialSize) gives the possibility for initializing the size to save time. A group can be added and removed with the slow methods InsertGroupInGroups(...) and RemoveGroupSlow(...) which preserves the order. The fast methods are AddGroup(...) and RemoveGroup(...) just as the CPixelGroups methods for inserting and removing positions. A group with a specific position can be found with the method GetGroupWithPos(...). Here is a simple example on how this works:

CPixelGroup pg1,pg2; // define two pixel groups
// add some positions
pg1.AddPosition(2,5);
pg1.AddPosition(4,9);
pg2.AddPosition(109,901);
pg2.AddPosition(500,664);

CPixelGroups pgroups; // the container for the groups
pgroups.AddGroup(pg1);
pgroups.AddGroup(pg2);
cout << pgroups << endl;

// example of how to find a group with a given position
unsigned int GroupIndex;
unsigned int PosIndex;
if (pgroups.GetGroupWithPos(GroupIndex,CPoint2D<UINT16>(109,901),
    PosIndex))
{
    CPixelGroup* pPixelGroup=pgroups.GetGroup(GroupIndex);
    cout << "Group with position " << pPixelGroup->GetPosition(PosIndex)
    << " found at index " << GroupIndex << endl << endl;
}

pgroups.Empty(); // empty the pixelgroups the same way as with a CStdImage
cout << "Emptying the CPixelGroups variable: " << endl;
cout << pgroups << endl;
8. Using the image algorithms of IPL98

Not all algorithms are described in this section, you will need to look at the IPL98 documentation pages to find more details about algorithms not mentioned here. We hope to be able to include some examples in a future version of this tutorial. The following algorithms are not explained here:

- CHoughProbRHTLine
- CHoughRHTCircle
- CHoughRHTLine
- CHoughSHTLine
- CSpectral
- CTopography

The easiest way to get an overview of all available image processing algorithms is from the "C++ overview" web page\textsuperscript{11}. Click on the "modules" link at the top of the documentation page. Now click the link "Algorithm classes (C++)".

8.1 Segmentation

The segmentation of an image is a general expression for arranging pixels with some kind of connection into groups. In IPL98 such a group is to be stored in a CPixelGroup (see section 7.6 for more information). Two segmentation algorithms is available in the library: Blob and contour tracing.

8.1.1 Blob algorithm

The blob\textsuperscript{12} algorithm segmentates the image pixels not belonging to the background (the pixels in interest) into groups of pixels which are either 4 or 8 connected. The method is found in the CSegmentate class and looks like this:

\[
\text{DeriveBlobs(CStdImage\& Source, CPixelGroups\& PixelGroups, \text{COLORYPE Color, UINT8 Threshold, CONNECTIVITY Connected})}
\]

The Source parameter is the image to derive the blobs from and each blob is stored as a PixelGroup in PixelGroups. The Color defines whether white or black pixels are to be considered as foreground. The type COLORYPE can be either HIGHCOLOR (white) or LOWCOLOR (black). If the source image is a graytone image the Threshold parameter is used to set a global threshold in the image. The last parameter of type CONNECTIVITY can be either EIGHTCONNECTED or FOURCONNECTED. Here is a complete example on how to work with the blob algorithm and the resulting blobs stored in a CPixelGroups container, the corresponding C example can be found in the appendix at page 85:

\textsuperscript{11}Go to www.mip.sdu.dk/ipl98 and click on the C++ overview button in the left pane.
\textsuperscript{12}Blob is an abbreviation for Binary Large OBjects
void main()
{
    CImage Img, ImgDest; // Images
    CPixelGroups PixelGroups; // Holds information about blobs
    unsigned int BlobIndex;
    unsigned int LastPos;
    CTimeDate Timer;
    Timer.StartTimer(); // stop watch for time elapsed
    unsigned int PosIndex;

    if (Img.Load("c:/temp/three_segments.bmp")) // your image with 8 b/p
    {
        cout << Img << endl;

        // Find the blobs in image
        CSegmentate Segm;
        Segm.DeriveBlobs(Img, PixelGroups, HIGHCOLOR, 128, EIGHTCONNECTED);

        // Ask which blob pixel position (30,30) belongs to
        CPoint2D<unsigned short> pnt(30, 30);
        if (PixelGroups.GetGroupWithPos(BlobIndex, pnt, PosIndex))
        {
            LastPos = (PixelGroups.GetGroup(BlobIndex))->GetTotalPositions();
            cout << "Pixel at pos " << pnt << " in blob " << BlobIndex << endl;
            cout << "First pixel " <<
            (PixelGroups.GetGroup(BlobIndex))->GetPosition(0) << endl;
            cout << "Last pixel " <<
            (PixelGroups.GetGroup(BlobIndex))->GetPosition(LastPos - 1)
            << endl;
            cout << "Total pixels in blob: " <<
            (PixelGroups.GetGroup(BlobIndex))->GetTotalPositions()
            << endl;
        }

        // Remove a blob from the list of groups
        cout << "Total groups now: " << PixelGroups.GetTotalGroups()
        << endl;
        cout << "Removing one group" << endl;
        PixelGroups.RemoveGroup(0);
        cout << "Total groups now: " << PixelGroups.GetTotalGroups()
        << endl;
    }

    if (PixelGroups.GetTotalGroups() > 1)
    {
        // Copy blob number 1 to gray tone image and write to
8.1 Segmentation

// disk (c:/temp/test.bmp)
PixelGroups.AddColors(Img); // first add colors
(PixelGroups.GetGroup(1))->CopyToImage(&Img.m_Pal,128,ImgDest);
ImgDest.Save("c:/temp/test.bmp");
cout << "Blob 1 saved as c:/temp/test.bmp" << endl;

cout << "Total time: " << Timer.StopTimer() << endl;
}
}
else
{
  cout << "Loading failed" << endl;
}
}

8.1.2 Contour tracing

Two methods for collecting the pixels belonging to a contour is available:

- FindAndFollowLowContour(CStdImage& Source, CPoint2D<int> Start, DIRECTION SearchDirection, CONNECTIVITY Connected, CPixelGroup& PixelGroup, ROTATION& RotationDirection, COLORTYPE& SurroundedColor)

- FindAndFollowHighContour(CStdImage& Source, CPoint2D<int> Start, DIRECTION SearchDirection, CONNECTIVITY Connected, CPixelGroup& PixelGroup, ROTATION& RotationDirection, COLORTYPE& SurroundedColor)

The algorithm used for both methods searches from a given start point Start in the direction SearchDirection for an edge. The SearchDirection is of type DIRECTION and can be one of the values: NORTH, EAST, SOUTH and WEST. The parameter Connected of type CONNECTIVITY can be either EIGHTCONNECTED or FOURCONNECTED. The information about the surrounding color (returned in SurroundedColor) and the clockwise or anti clockwise search directions (returned in RotationDirection) gives information about wether the found contour is on the inner or outer edge of a figure. The RotationDirection parameter of type ROTAION can be one of the values: CLOCKWISE and ANTICLOCKWISE. If no edge is found in the search direction the method returns false. Otherwise the methods starts following this contour until it eventually returns to the first found edge. The result is placed in a CPixelGroup object. The example below shows how to use both methods and the results are plotted in an image with red and green for each contour type and saved to a file:

CImage Img,ImgDest; // Images
CPixelGroup pg1,pg2; // Holds information about the contour
CTimeDate Timer;
Timer.StartTimer(); // stop watch for time elapsed

// load your image with 1 or 8 b/p
if (Img.Load("c:/temp/three_segments.bmp"))
8.1 Segmentation

```
{ 
    cout << "Image loaded:" << endl << Img << endl;
    COLORTYPE ColType;
    ROTATION RotDirection;
    CPoint2D<int> start(200,200); // start position to search from

    // Find a low contour in image
    Img.SetBorder(1,0);
    CSegmentate Segm;
    Segm.FindAndFollowLowContour(Img,start,EAST,
                                   EIGHTCONNECTED,pg1,RotDirection,ColType);
    // Write information to cout
    cout << endl << "High contour info: " << endl << "RotDirection=";
    cout << ((RotDirection==CLOCKWISE) ? "CLOCKWISE":"ANTICLOCKWISE");
    cout << " ColType=";
    (ColType==HIGHCOLOR) ? cout << "HIGHCOLOR" : cout << "LOWCOLOR";
    cout << endl << pg1 << endl;

    // Find the corresponding high contour in image
    Segm.FindAndFollowHighContour(Img,start,EAST,
                                   EIGHTCONNECTED,pg2,RotDirection,ColType);
    // Write information to cout
    cout << endl << "Low contour info: " << endl << "RotDirection=";
    cout << ((RotDirection==CLOCKWISE) ? "CLOCKWISE":"ANTICLOCKWISE");
    cout << " ColType=";
    (ColType==HIGHCOLOR) ? cout << "HIGHCOLOR" : cout << "LOWCOLOR";
    cout << endl << pg2 << endl;

    // create image with dimensions to contain both contours
    ImgDest=CImage(pg1.GetRight().GetX()-pg1.GetLeft().GetX()+1,
                   pg1.GetBottom().GetY()-pg1.GetTop().GetY()+1,24);
    cout << "Created image to plot in:" << endl << ImgDest << endl;
    unsigned int count;
    CPoint2D<int> offset(pg1.GetLeft().GetX(),pg1.GetTop().GetY());
    // plot low contour as red
    for(count=0;count<pg1.GetTotalPositions();count++)
    {
        ImgDest.SetPixel(pg1.GetPosition(count)-offset,
                         CPalette::CreateRGB(255,0,0));
    }
    // plot high contour as green
    for(count=0;count<pg2.GetTotalPositions();count++)
    {
        ImgDest.SetPixel(pg2.GetPosition(count)-offset,
                         CPalette::CreateRGB(0,255,0));
    }
}
```
8.2 Coordinate transformation

Algorithms involving some kind of transformation for each pixel in one coordinate system to a new coordinate system is collected in the class CCoordinateTransform.

8.2.1 Scaling and rotation

Four methods are available for scaling and rotation, the effect of each method is shown in figure 11:

- **ScaleAndRotateAuto(...)** Scale and rotates the image at the same time. Auto is appended because the method resizes the image so no pixels are lost.

- **ScaleAndRotateFixed(...)** Same as above except the resulting image size is given as parameters. This may result in loosing some foreground pixels from the original image.

- **ScaleAuto(...)** Only scales the image, auto resizing of image size.

- **ScaleFixed(...)** As above but with resulting image size given as parameters.

The following example uses all four methods and writes the results in the files named: `ScaleAndRotateAuto.bmp`, `ScaleAndRotateFixed.bmp`, `ScaleAuto.bmp` and `ScaleFixed.bmp`, all of them saved in `c:/temp/`. The results are shown in figure 11.

```cpp
CImage Img, ImgDest; // Images
CTimeDate Timer;
Timer.StartTimer(); // stop watch for time elapsed

if (Img.Load("c:/temp/three_segments.bmp")) { // your image with 8 b/p
    cout << "Image loaded:" << endl << Img << endl;
    CCoordinateTransform ct;
    cout << endl << "Using ScaleAndRotateAuto()" << endl;
```
In figure 11 the result of the coordinate transformations are shown.

8.3 Feature extraction

The class CFeatures is used for algorithms extracting all kind of features from an image. For IPL98 version 1.01 the class only contains moment features.
8.3 Feature extraction

Figure 11: The results from coordinate transformations with scaling and rotation: a) Original image b) Method `ScaleAndRotateAuto(...)` used c) Method `ScaleAndRotateFixed(...)` used d) Method `ScaleAuto(...)` used e) Method `ScaleFixed(...)` used.

### 8.3.1 Moments

The following methods are available for calculating different moments. They all work on a `CPixelGroup` object.

- **DeriveMomentsGrayTone(...)** The weight of each pixel is the original pixel value minus a "Background" value which is given as a parameter. The results are stored in the `m_Moments` attribute. This attribute structure contains members named `m_{pq}` where (p+q) is the moment order as explained in Gozales & Woods page 514. The calculation is very fast, and a parameter named "MaxOrder" can be used to calculate the low order moments only, which makes the method even faster.

- **DeriveMomentsBinary(...)** Calculates the same moments as `DeriveMomentsGrayTone()`, except that each pixel has the value 1.

- **DeriveCentralMoments()** These central moments denoted \( \mu_{pq} \) in Gonzales & Woods are derived from the simple moments calculated by `DeriveMomentsBinary()` or `DeriveMomentsGrayTone()`. If they are not derived in advance the method will cause an error.

- **DeriveCentralNormMoments()** These normalized central moments denoted \( \eta_{pq} \) in Gonzales & Woods are derived from the simple moments calculated by `DeriveMomentsBinary()` or `DeriveMomentsGrayTone()`. If they are not derived in advance the method will cause an error.

- **DeriveHuFromCentralMom()** A total of 13 Hu moments are calculated, only the first seven are found in Gonzalez & Woods at page 516. More information is found in the article “Visual Pattern Recognition by Moment Invariants" by Ming-Kuei Hu [2].
method calculates Hu moments from the central moments calculated by `DeriveCentralMoments()`. If they are not derived in advance the method will cause an error. The result is placed in the public member `m_Hu` which is an array of 13 doubles. Indexing is done according to the numbering of the invariants, that is starting from number 1, index 0 is not used.

- **DeriveHuFromCentralNormMom()** Same as `DeriveHuFromCentrMom()`, except that the source moments are now the central normalized moments. The result is placed in the public member `m_Hu` which is an array of 13 doubles. Indexing is done according to the numbering of the invariants, that is starting from number 1, index 0 is not used.

An example showing how to extract moment features:

```cpp
CImage Img; // Images
CTimeDate Timer;
Timer.StartTimer(); // stop watch for time elapsed

if (Img.Load("c:/temp/three_segments.bmp")) // your image
{
    cout << "Image loaded:" << endl << Img << endl;
    CPixelGroups pgs;
    CSegmentate seg;
    CFeatures features;
    seg.DeriveBlobs(Img, pgs, HIGHCOLOR, 100, EIGHTCONNECTED);
    // iterate through the found groups and find moments
    for(int x=0; x<pgs.GetTotalGroups(); x++)
    {
        if (features.DeriveMomentsBinary(*pgs.GetGroup(x), 3))
        {
            cout << endl << *pgs.GetGroup(x) << endl; // group info
            cout << "Moment values found for segment " << x << endl;
            cout << "M00=" << features.m_Moments.m00 << " M10=" << features.m_Moments.m10 << " M01=" << features.m_Moments.m01 << endl;
            cout << "M20=" << features.m_Moments.m20 << " M02=" << features.m_Moments.m02 << " M11=" << features.m_Moments.m11 << endl;
            cout << "Deriving central moments..." << endl;
            features.DeriveCentralMoments();
            cout << "CM20= " << features.m_CentrMom.m20 << endl;
            cout << "Deriving central normalized moments..." << endl;
            features.DeriveCentralNormMoments();
            cout << "CNM20= " << features.m_CentrNormMom.m20 << endl;
            cout << "Deriving Hu moments..." << endl;
            features.DeriveHuFromCentralMom();
            cout << "Hu[0]= " << features.m_Hu[0] << endl;
        }
    }
}
```

8.3 Feature extraction
8.4 Morphological operations

The morphological class CMorphology includes the two simple operations erode and dilate. It also includes three variations on skeletonizing which are explained later.

8.4.1 Dilate and erode

The only morphological operations implemented at present is dilation and erosion, which are explained in Gonzalez & Woods page 518-521. Two versions of each type is available and their effects are shown in figure 12:

- **Dilate(...)** Uses a CStdImage as the mask and the active positions are pixel values different from 0.
- **DilateFast(...)** A speed optimized method working with a 3*3 mask only. The mask values are given as three bit combinations with values between 0 and 7.
- **Erode(...)** Uses a CStdImage as the mask and the active positions are pixel values different from 0.
- **ErodeFast(...)** A speed optimized method working with a 3*3 mask only. The mask values are given as three bit combinations with values between 0 and 7.

Here is a complete example on how to work with the simple erode and dilate operations, the result is shown in figure 12, the corresponding C example can be found in the appendix at page 86:

```cpp
string dty; // Holding the date, time and year
CTimeDate time;
CImage Img, ImgDest;
unsigned int count;

/* create and set a 3*3 mask */
CImage ImgMask(3, 3, 1);
ImgMask.Flush(1);

/* load image */
if (Img.Load("c:/temp/doggie.bmp"))
{
```

```cpp
cout << endl << "Total time: " << Timer.StopTimer() << endl;
}
else
{
    cout << "Loading failed" << endl;
}
```
8.4 Morphological operations

CMorphology morph;
cout << Img << endl;

/* an example of using the timer and date functions */
cout << "On a 400 MHz Pentium II this program takes"
  " 0.83 seconds to complete" << endl;
time.StartTimer();

/* convert to 1 b/p */
cout << "Converting image from " << Img.GetBits() << 
  " b/p to 1 b/p" << endl;
ImgDest.CopyConvert (1,Img);

/* invert image */
cout << "Inverting image" << endl;
ImgDest.Invert();

/* dilate */
cout << "Dilate image 5 times" << endl;
for(count=0; count<5; count++)
{
  morph.Dilate(ImgMask,ImgDest);
  /* replace the next line with the one above to make it faster */
  /*morph.DilateFast(ImgDest,7,7,7);*/
}

cout << "Erode image 5 times" << endl;
for(count=0; count<5; count++)
{
  morph.Erode(ImgMask,ImgDest);
  /* replace the next line with the one above to make it faster */
  /*morph.ErodeFast(ImgDest,7,7,7);*/
}

cout << "Saving image as \"c:/temp/test.bmp\"" << endl;
ImgDest.Save("c:/temp/test.bmp");

if (!time.GetDateTimeYear(dty))
  cout << "Failed reding date and time...." << endl;
else
  cout << "This program was run at: " << dty << endl;

cout << "Seconds elapsed: " << time.GetTimer() << endl;
else
{
  cout << "Loading failed" << endl;
8.4 Morphological operations

Try to replace the normal \texttt{Erode(...)} and \texttt{Dilate(...)} methods with the corresponding fast methods to get an idea of the decrease in execution time. In figure 12 the results of the dilation followed by erosion can be seen.

![Original image](doggie.bmp) \hspace{1cm} ![Dilation](doggie_dilate.bmp) \hspace{1cm} ![Erosion](doggie_erode.bmp)

Figure 12: The results from dilation and erosion. rotation: a) Original image (doggie.bmp) which is converted to 1 b/p and then inverted b) Method \texttt{Dilate(...)} used 5 times c) Method \texttt{Erode(...)} used 5 times.

8.4.2 Thinning and skeletonizing

Three methods are available for thinning and skeletonizing an object:

- \texttt{Thinning(...)} Reduces all black components in a binary image to single pixel wide branches and preserves the following properties: 1) Does not remove end points, 2) does not break connectedness, and 3) does not cause excessive erosion of the region. The algorithm is described in detail in Gonzales and Woods page 491-494. Note: A thinning algorithm is not the same as a medial axis transform (MAT).

- \texttt{Skeletonizing(...)} Transforms all black components in a binary image to a skeleton defined by the medial axis transform (MAT). Algorithm implemented with the method explained in Gonzales and Woods page 535-538.

- \texttt{SkeletonZhou(...)} A skeleton somewhere between the results of \texttt{Thinning(...)} and \texttt{Skeletonizing(...)} is produced. The resulting skeleton is more robust to noise on edges, i.e. it produces less small unwanted branches arising from noise on edges. But the skeleton is closer to a real MAT compared to the \texttt{Thinning()} algorithm. The algorithm is explained in detail in the article "A novel single-pass thinning algorithm and an effective set of performance criteria" by C. Quek & G. S. Ng. R. W. Zhou [3]. Pattern Recognition Letters, 16:1267-1275, 1995. The parameter \texttt{PreserveCorners} is an additional feature to the original algorithm. It prevents the algorithm from eroding corners and the result is then very close to a MAT.

The first two methods mentioned is the normal thinning and skeletonizing whereas \texttt{SkeletonZhou} is somewhere in between. Skeletonizing an object according to the MAT turns out to be very difficult to implement and the method \texttt{Skeletonizing(...)} is not always correct. In
most cases it also results in many small branches which needs to be pruned. The method `SkeletonZhou(...)` does not have the problems with the branches and are often a good alternative to skeletonizing. It is recommended to try all methods to get an idea about the results in a given task.

In the figures 13 to 20 a number of examples of each thinning/skeletonizing algorithm are shown on different shapes\(^\text{13}\).

![Figure 13: From left: 1. original shape, 2. Thinning(...), 3. Skeletonizing(...), 4. SkeletonZhou(...), 5. SkeletonZhou(...) with the parameter `PreserveCorners` set to true.](image)

### 8.5 Perspective camera transformation

This section is a bit outdated but most of it should still be correct. But it is recommended that you read the documentation page for the CPerspective class in order to get the latest news about the class.

The class CPerspective is used for calculating the standard camera matrix and for 3D and 2D calculations. A description of the camera model leading to the camera matrix is found in Gonzalez & Woods [1] page 52-68. Additional information of the camera model and the inverse perspective transformation can be found in “Machine Vision” by David Vernon [4]. The methods available are:

- **Calibrate(...)** Calibrates the standard camera matrix as given in Gonzalez & Woods page 67. At least six world coordinate points with corresponding image points\(^\text{15}\) is needed. The result is put into the C structure TCameraMatrix and each position in the matrix can be read with the method GetCamMatrix(i,j), where (i,j) is the position starting at (1,1) in upper left corner.

- **Calc3Dto2D(...)** Finds the correspondance between a given 3D point in space and the 2D image point.

- **Direction(...)** Calculates direction from a given 2D image point through Pinhole.

- **PinHole(...)** Caluculates the pinhole world coordinate.

The following example calculates the camera matrix from 11 measured points on a calibration plate. The plate can be seen in figure 21, and the camera image of the plate is shown in figure 22, note that the reference marks have to be in at least two different depths in order to calculate the camera matrix. When streaming the CPerspective class with the `<<` operator to cout the calculated camera matrix is shown. It is also possible to get the indexes written

\(^{13}\) If reading this tutorial as web pages it is in some cases impossible to see the details in these figures, see the printed manual instead.

\(^{14}\) Often called world coordinate point.

\(^{15}\) In this area often called the CCD points refering to the CCD chip in the camera.
8.5  Perspective camera transformation

Figure 14: From left: 1. original shape, 2. Thinning(...), 3. Skeletonizing(...), 4. SkeletonZhou(...), 5. SkeletonZhou(...) with the parameter \textit{PreserveCorners} set to true.

Figure 15: From left: 1. original shape, 2. Thinning(...), 3. Skeletonizing(...), 4. SkeletonZhou(...), 5. SkeletonZhou(...) with the parameter \textit{PreserveCorners} set to true.

Figure 16: From left: 1. original shape, 2. Thinning(...), 3. Skeletonizing(...), 4. SkeletonZhou(...), 5. SkeletonZhou(...) with the parameter \textit{PreserveCorners} set to true.

Figure 17: From left: 1. original shape, 2. Thinning(...), 3. Skeletonizing(...), 4. SkeletonZhou(...), 5. SkeletonZhou(...) with the parameter \textit{PreserveCorners} set to true.

Figure 18: From left: 1. original shape, 2. Thinning(...), 3. Skeletonizing(...), 4. SkeletonZhou(...), 5. SkeletonZhou(...) with the parameter \textit{PreserveCorners} set to true.
8.5 Perspective camera transformation

Figure 19: From left: 1. original shape, 2. Thinning(...), 3. Skeletonizing(...), 4. SkeletonZhou(...), 5. SkeletonZhou(...) with the parameter `PreserveCorners` set to true.

out using the method `PrintCameraMatrix(true)`. The last part of the code simply shows how to use the methods.

```cpp
CTimeDate Timer;
CPerspective Persp;
CPoint3D<float> pnts3d[11];
CPoint2D<float> pnts2d[11];
Timer.StartTimer(); // stop watch for time elapsed
// positions in world coordinates [mm]
pnts3d[0]=CPoint3D<float>(55,65,1);
pnts3d[1]=CPoint3D<float>(269,55,1);
pnts3d[2]=CPoint3D<float>(489,68,1);
pnts3d[3]=CPoint3D<float>(163,163,122);
pnts3d[4]=CPoint3D<float>(374,160,122);
pnts3d[5]=CPoint3D<float>(225,233,122);
pnts3d[6]=CPoint3D<float>(163,268,122);
pnts3d[7]=CPoint3D<float>(373,267,122);
pnts3d[8]=CPoint3D<float>(54,375,1);
pnts3d[9]=CPoint3D<float>(266,377,1);
pnts3d[10]=CPoint3D<float>(470,378,1);
// CCD coordinates [pixels]
pnts2d[0]=CPoint2D<float>(91.889694,62.717804);
pnts2d[1]=CPoint2D<float>(369.633486,48.792614);
pnts2d[2]=CPoint2D<float>(660.180781,64.659289);
pnts2d[3]=CPoint2D<float>(214.531214,173.732403);
pnts2d[4]=CPoint2D<float>(521.231441,170.783393);
pnts2d[5]=CPoint2D<float>(305.030159,271.006412);
pnts2d[6]=CPoint2D<float>(215.510610,319.824717);
pnts2d[7]=CPoint2D<float>(522.741073,318.324994);
pnts2d[8]=CPoint2D<float>(96.084059,446.294712);
pnts2d[9]=CPoint2D<float>(372.837662,448.715393);
pnts2d[10]=CPoint2D<float>(639.934707,448.745004);

Persp.Calibrate(6,pnts3d,pnts2d);
cout << "Camera matrix calculated: " << endl;
cout << Persp << endl;

// calculating the first 3D points corresponding CCD position
```

60
8.5 Perspective camera transformation

Figure 20: From left: 1. original shape, 2. Thinning(...), 3. Skeletonizing(...), 4. SkeletonZhou(...), 5. SkeletonZhou(...) with the parameter PreserveCorners set to true.

```cpp
CPoint2D<FLOAT32> pnt2D;
Persp.Calc3Dto2D(pnts3d[0], pnt2D);
cout << endl << "3D point " << pnts3d[0] << " in 2D CCD chip is: " <<
pnt2D << endl;
cout << "Original measured point is " << pnts2d[0] << endl;

// finding direction vector for pnt2D
CPoint3D<FLOAT32> pnt3D;
Persp.Direction(pnt2D, pnt3D);
cout << "Direction for point " << pnt2D << " from pinhole is "
<< pnt3D << endl;

// finding pinhole world coordinate
Persp.PinHole(pnt3D);
cout << endl << "Pinhole world coordinate is at " << pnt3D << endl;

cout << endl << "Total time: " << Timer.StopTimer() << endl;
```

The next example shows how to write a simple program that finds the center of each mark on the calibration plate. The program load the original image taken by the camera and converts it to 1 b/p. The CSegmentate class is then used to derive each white blob in the image with the method DeriveBlobs(...). By inspection of the binary image we know that each landmark must be a blob with more than 1000 pixels. Moments are then calculated on each blob containing more than 1000 pixels and the center of mass is found by:

\[
(x, y) = \left( \frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right)
\]

In figure 23 is shown the binary image derived from the original in figure 22. In figure 24 is shown the positions found for each mark. The accuracy of this method is probably within two pixels, it is possible with more sophisticated methods to find the center with subpixel accuracy. Here is the example code:

```cpp
CTimeDate Timer;
Timer.StartTimer();
CImage Img;
if (Img.Load("c:/temp/calibrationplate.bmp"))
```
8.5  Perspective camera transformation

Figure 21: The calibration plate used in the given example. The coordinate system is shown.

Figure 22: A camera image of the calibration plate shown in figure 21. The coordinate system is shown.
8.5 Perspective camera transformation

{
    Img.CopyConvert(1,Img);
    cout << "Image after converting\n" << Img << endl << endl;
    Img.Save("c:/temp/test.bmp");

    CSegmentate Segm;
    CPixelGroups Pgrps;
    CFeatures Features;

    Segm.DeriveBlobs(Img,Pgrps,HIGHCOLOR,1,EIGHTCONNECTED);
    cout << "Pixelgroups info:" << endl << Pgrps << endl;
    for(int count=0; count<Pgrps.GetTotalGroups(); count++)
    {
        CPixelGroup* pGrp=Pgrps.GetGroup(count);
        // comment out the next line to get less information on screen
        cout << "Pixelgroup number " << count << " info\n" <<
            *pGrp << endl;
        // by inspection of image we know that the marks
        // contains at least 1000 pixels, other small pixelgroups
        // are just skipped
        if(pGrp->GetTotalPositions()>1000)
        {
            Features.DeriveMomentsBinary(*pGrp,2);
            double xmean=Features.m_Moments.m10/Features.m_Moments.m00;
            double ymean=Features.m_Moments.m01/Features.m_Moments.m00;
            CPoint2D<double> pnt(xmean,ymean);
            cout << "Center of mass at " << pnt << endl;
        }
        else
        {
            cout << "Not enough pixels - skipping" << endl;
        }
    }
    cout << endl << "Total time: " << Timer.StopTimer() << endl;

    The center of mass found for each reference marks is:

    Center of mass at (369.288,49.1676)
    Center of mass at (91.5484,62.8519)
    Center of mass at (659.776,64.7859)
    Center of mass at (520.913,170.879)
    Center of mass at (214.196,173.837)
    Center of mass at (304.72,271.21)
    Center of mass at (522.402,318.474)
    Center of mass at (215.169,319.904)
8.5 Perspective camera transformation

Figure 23: The calibration image converted to one bit per pixel.

Figure 24: The center of mass found for each mark shown with a cross.
8.5 Perspective camera transformation

Center of mass at (95.7496,446.436)
Center of mass at (372.467,448.83)
Center of mass at (639.579,448.902)

A better method is to convert the original image to gray tones (8 b/p) and extract the center of mass from the gray tones. Here is the corresponding code for this:

CTimeDate Timer;
Timer.StartTimer();
CImage Img;
if (Img.Load("c:/temp/calibrationplate.bmp"))
{
    Img.CopyConvert(8,Img);
    cout << "Image after converting\n" << Img << endl << endl;
    Img.Save("c:/temp/test.bmp");

    CSegmentate Segm;
    CPixelGroups Pgrps;
    CFeatures Features;

    Segm.DeriveBlobs(Img,Pgrps,HIGHCOLOR,128,EIGHTCONNECTED);
    cout << "Pixelgroups info:" << endl << Pgrps << endl;
    for(int count=0; count<Pgrps.GetTotalGroups(); count++)
    {
        CPixelGroup* pGrp=Pgrps.GetGroup(count);
        // comment out the next line to get less information on screen
        cout << "Pixelgroup number " << count << " info\n" <<
             *pGrp << endl;
        // by inspection of image we know that the marks
        // contains at least 1000 pixels, other small pixelgroups
        // are just skipped
        if(pGrp->GetTotalPositions()>1000)
        {
            pGrp->AddColors(Img);
            Features.DeriveMomentsGrayTone(*pGrp,0,2);
            double xmean=Features.m_Moments.m10/Features.m_Moments.m00;
            double ymean=Features.m_Moments.m01/Features.m_Moments.m00;
            CPoint2D<double> pnt(xmean,ymean);
            cout << "Center of mass at " << pnt << endl;
        }
        else
        {
            cout << "Not enough pixels - skipping" << endl;
        }
    }
}


8.5  **Perspective camera transformation**

```cpp
    cout << endl << "Total time: " << Timer.StopTimer() << endl;
```

Compare this code with the first example and you will realize that only a few changes are necessary to get the gray tone center of mass values. Here is the results from this:

- Center of mass at (369.633, 48.7926)
- Center of mass at (91.8897, 62.7178)
- Center of mass at (660.181, 64.6593)
- Center of mass at (521.231, 170.783)
- Center of mass at (214.531, 173.732)
- Center of mass at (305.03, 271.006)
- Center of mass at (522.741, 318.325)
- Center of mass at (215.511, 319.825)
- Center of mass at (96.0841, 446.295)
- Center of mass at (372.838, 448.715)
- Center of mass at (639.935, 448.745)
9. Utilities

9.1 Macros and small utility functions

Two files in the library contain small functions which are useful in many places when programming. The files are \texttt{ipl98/kernel\_c\_general\_functions/ipl\_general\_functions\_c\_only.h} and \texttt{ipl98/cpp/ipl98\_general\_functions.h}. Here follows a description of the C++ macros only, they are all template based. For the involved type T it is written what operator need to be implemented to make the macro work with the type:

- \textbf{PI} This is simply the definition of PI (\( \sim 3.14 \ldots \)), not a template macro.
- \textbf{PI2} Defined as 2*PI. Not a template macro.
- \textbf{RadToDegree(Rad)} Converts a radian value to degrees.
- \textbf{DegreeToRad(Degree)} Converts a degree value to radians.
- \textbf{Sqr(x)} Calculates the square of x \((x \times x)\). The \texttt{*}-operator need to work for the type of x.
- \textbf{Swap(x,y)} Swaps the contents of x and y. The \texttt{=} -operator need to work for the type of x and y.
- \textbf{Max(x,y)} Returns the greatest value of x and y. The \texttt{>} -operator need to work for the type of x and y.
- \textbf{Min(x,y)} Returns the minimum value of x and y. The \texttt{<} -operator need to work for the type of x and y.
- \textbf{Round(x)} Round x to nearest integer value. The \texttt{(<,+,-)}-operators need to work for the type of x.
- \textbf{RoundUINT8(x)} Round x to nearest UINT8 value or if \( x > 255 \) then \( x = 255 \). The \texttt{(<,>,+)}-operators need to work for the type of x.

Some global functions for handling file names and paths are also available:

- \textbf{SplitFileNames(...)} Splits a file name including path given in "FilePathName" into a path (returned in "Path") and a filename including extension (returned in "FileName-Ext") and an extension (returned in "Ext").
- \textbf{ConvertBackslashToSlash(...)} Converts all back slash characters ‘/’ to slash ‘/’. Used to avoid problems with C++ reading a ‘/’ path separation as an escape character.
- \textbf{AddTrailingSlashToPath(...)} Adds a trailing ‘/’ character to the path if not present.
- \textbf{SkipSpaceAndComments(istream& s)} Used for in-streams. Skips all whitespace, newline, tab characters and comment lines, i.e. rest of line from a ‘#’ character. This method is useful if you create setup files, where you very often would like to be able to comment out some lines!
9.2 Graphics in images

The class CGraphics makes it possible to draw simple graphics directly in an image. This includes drawing:

- Text
- Lines
- Circles
- Ellipses

Three different bitmap fonts are available: CourierBold13x18, Courier8x12 and Terminal12x16. You choose them by calling the method SetFontActive(BITMAPFONT Font). Many other things can be adjusted, look in the class and the example below to see what options you have. The example also shows how to draw lines and circles, the resulting image in the variable Img is shown in figure 25. Here is the code example:

CImage Img(200,120,8,255);
CGraphics Graphics;
// Writing and aligning text in an image
Img.SetPixel(100,100,0);
GraphicsSetFontActive(CGraphics::Terminal12x16);
GraphicsSetFontColor(128);
GraphicsSetFontTransparent(false);
GraphicsSetFontBackgroundColor(200);
GraphicsSetFontAlignment(CGraphics::TOPRIGHT);
GraphicsPlotString("jg_RIGHT",CPoint2D<int>(100,100),Img);
// Following a line in an image
CPixelGroup Grp;

Figure 25: Example using different methods from the CGraphics class.

- ToUpperCase(...) Converts all characters in given string to upper case.
- CompareNoCase(...) Compares two strings case insensitive. Return values are like the ANSI cmp function: -1 if s<s2, 0 if s==s2 or 1 if s>s2, where s and s2 are the given strings.
- ConvertLinesToComments(...) Inserts ’ #’ in front of each line contained in given string. This is used for writing information to a file as comment lines.
9.3 Corresponding 3D-2D points

When working with machine vision and the relation ship between world coordinates and image coordinates, it is convenient to have a class for working with point sets containing the correspondence between 3D points and 2D points. This is possible with the container class CCorresponding3D2DPoints. The following example shows how to use the class:

```cpp
int main() {
    CCorresponding3D2DPoints PointSet(3);
    CCorresponding3D2DPoints PointSet2;
    for(int count=0; count<10; count++)
    {
        CPoint3D<float> P3d(count,count,count);
        CPoint2D<float> P2d(count,(float)7.65434);
        PointSet.AddPointSetAndId(P3d,P2d,count);
    }
    PointSet2=PointSet; /* copying to new object */
    PointSet.ReAlloc3D2DPoints(1); /* set the original object size to 1 */
    PointSet2.AdjustSizeOf3D2DPoints(); /* deallocate extra allocation */
    cout << PointSet2 << endl;
    if (getchar()){}
    return 0;
}
```

In addition, simple figures like a cross, rectangle and a plus can be drawn.

9.3 Corresponding 3D-2D points
9.3 Corresponding 3D-2D points

The corresponding code for the kernel C part of the library can be found in the appendix page 87. This class is very often used in conjunction with the CPerspective class explained in section 8.5.
10. Camera classes and IPL98

This chapter is unfortunately a bit outdated, I did not have time to rewrite it in the last update of this tutorial. I guess it is still worth something, if you need to use our camera library.

The integration of cameras and frame grabbers is not a part of IPL98 yet but are being developed at the same time as this tutorial is written. More details can be found on the IPL98 home page www.mip.sdu.dk/ipl98 by clicking the button ”Camera classes” in the left pane. Here follows a brief description of the decisions made so far.

The purpose of the camera classes is to make a standard interface for manipulating camera settings and acquiring images. The design should make it an easy task to put a new camera and framegrabber on a system by changing only a few lines in the code. A hierarchy of camera classes and one generic frame grabber class has been designed and is shown in figure 26.

The class CCameraGeneric will be a virtual class defining a number of methods which must be implemented in the inherited classes. Because all cameras do not have the same features there will be methods available for testing if a given feature is available, an example is:

```cpp
bool ShutterAvailable() = 0;
virtual double GetShutter();
virtual double SetShutter();
```

Only the method ShutterAvailable() is pure virtual whereas the other two methods have a dummy implementation. In that way the programmer of a camera class where shutter is not available do not have to implement a dummy functionality for the Get- and SetShutter methods. The two classes CCameraUSB and CCameraFireWire may be some generic classes.
or they may be implemented directly as inherited classes from the CCameraGeneric, more knowledge about this area is needed before a final decision can be done here.

The class CCameraFG is a generic class for all cameras working with a framegrabber. This class have an attribute of type CFrameGrabberGeneric which is a generic framegrabber class from which all framegrabber classes is to be inherited. The framegrabber class have a method Acquire(CstdImage Img) which the CCameraFG class’ method Acquire(CStdImage Img) uses. When initializing an object inherited from the CCameraFG class a reference to a CFrameGrabberGeneric object must be provided. This setup should make it very easy to change framegrabber and/or camera with only a small change in the code, here is an example:

Say we have two classes inherited from CCameraFG named CBasler1378 and CSonyAX123 and two framegrabber classes inherited from CFrameGrabberGeneric named CITIAMDIG and CDT3152. The programmer starts out with a setup using the Basler camera with an ITI framegrabber, setting it up in the code is done like this:

```cpp
CITIAMDIG fg; // framegrabber object
CBasler1378 camera; // create Basler camera object

// the following code should not be changed when
// using another camera or framegrabber
camera.Initialize(fg,0); // connect the camera object with the fg
// object and use port 0 on framegrabber

// simple example setting the shutter
if (camera.ShutterAvailable())
{
    cout << 'Shutter is: ' << camera.GetShutter() << endl;
}
CImage Img;
camera.Acquire(Img); // acquire image and store it in Img
```

Lets assume the user now changes the Basler camera with the sony camera, the only change in code will be:

```cpp
CSonyAX123 camera; // create Sony camera object
```

The CBasler1378 is simply changed with CSonyAX123. Changing the framegrabber to DT is done in a similar way:

```cpp
CDT3152 fg; // framegrabber object
```

The CITIAMDIG is replaced with CDT3152. No changes in the rest of the code should be neccesary.

The two methods AddCamera(PortNr, pCam) and RemoveCamera(pCam) are created to take framegrabbers with more than one camera port into account. The methods should be used by the method Initialize(...) in the classes inherited from CCameraFG.
Figure 27: The file structure and naming convention of the camera and framegrabber classes.

The filenaming convention can be seen in figure 27. Each camera class working with frame grabber and framegrabber class must be placed in a folder named “camera_brand_model” and “framegrabber_brand_model” placed in the folder camera_framegrabber.

More details about the design will be available in an updated version of this tutorial, until then look for information at the IPL98 homepage.
11 Adding code to IPL98

This chapter describes how to write code to be included in future versions of IPL98. It is desirable that the code is written in ANSI C but it is not a requirement, ANSI/ISO C++ code will also be accepted. This decision is made because in some cases it may be a big problem implementing it in C only, for instance if the Standard Template Library\textsuperscript{16} is used as an essential part of the algorithm. Before reading the rest of this section it is recommended to read section 4 Structure in order to fully understand how the library is build.

The first thing to consider is where the algorithm fits into the library. In most cases it should be natural to add it as a method in one of the algorithm classes:

- CCoordinateTransform
- CFeatures
- CLocalOperation
- CMaskOperation
- CMorphology
- CPerspective
- CSegmentate
- CSpectral
- CStatistic

Descriptions of each class can be found in previous sections. The following description will assume that the algorithm can be added to one of these classes, please contact the developer of IPL98 if it turns out to be difficult to decide where to place a method.

Code requirements

Here is a list of the things that must be fulfilled in order to have the code inserted into IPL98. It includes a nomenclature naming convention that also has to be followed:

- **Documentation** All public methods must be documentated in agreement with the DOC++\textsuperscript{17} standard. Every parameter and the return value must be commented. The “@author” and “@version” options must be included. In the C code the documentation must start with “@name” followed by the function name. Here is an example of the C and C++ documentation for k_DeriveBlobs (in C++ DeriveBlobs found in the CSegmentate class):

  ```c
  /** @name k_DeriveBlobs
   * Derive blobs from image and put results in TPixelGroups. If you want the original colors added to the found pixel groups call
   */
  ```

\textsuperscript{16}The Standard Template Library (STL) is now part of the standard C++ library and is therefore highly recommended to use.

\textsuperscript{17}Read more at http://www.zib.de/Visual/software/doc++/index.html.
11. Adding code to IPL98

the function k_AddColorsToGroups(...) found in the TPixelGroups
area.
@param pSource The source image to derive from, nothing is
changed in the image.
@param Color The color to segmentate into blobs, values are
HIGHCOLOR and LOWCOLOR.
@param Threshold A simple threshold value to separate background
from blobs in graytone images. not used if source image is 1
b/p.
@param Connected Decide if the blobs are to be eight- or
fourconnected, as described in Gonzalez and Woods page 41.
@return False if source image given to constructor is empty or
not 1 b/p or 8 b/p.
@version 1.00
@author Implementation by Ivar Balslev (ivb@mip.sdu.dk) and René
Dencker Eriksen (edr@mip.sdu.dk) */

bool k_DeriveBlobs(TImage* pSource, TPixelGroups* pPixelGroups,
                   COLORTYPE Color, UINT8 Threshold, CONNECTIVITY
                   Connected);

And the corresponding C++ documentation:

/** Derive blobs from image and put results in CPixelGroups. Old
data in "PixelGroups" are destroyed. If you want the
original colors added to the found pixel groups call the
method AddColors(...) found in the CPixelGroups class.
@param Color The color to segmentate into blobs, values are
HIGHCOLOR and LOWCOLOR.
@param Threshold A simple threshold value to separate background
from blobs in graytone images. not used if source image is 1
b/p.
@param Connected Decide if the blobs are to be eight- or
fourconnected, as described in Gonzalez and Woods page 41.
@return False if source image given to constructor is empty or
not 1 b/p or 8 b/p.
@version 1.00
@author Implementation by Ivar Balslev (ivb@mip.sdu.dk)
and René Dencker Eriksen (edr@mip.sdu.dk) */

bool DeriveBlobs(CStdImage& Source, CPixelGroups& PixelGroups,
                  COLORTYPE Color, UINT8 Threshold, CONNECTIVITY
                  Connected);

For more examples look in the header files in the library.

- Return values If possible let the method return a boolean value and test all possible
  sources for errors. If a method needs to return some other information do it in a
  reference parameter. In case of ANSI C use a pointer. Here is an example:
C++: Erode(const CStdImage& ImgMask, CStdImage& Img)  C: k_Erode(const TIntImage* pMask, TIntImage* pImg)

Both methods returns a boolean but and the result is placed in the parameter Img/pImg.

• **Nomenclature** The following rules must be followed:

1. Functions and macros written in ANSI C begins with the prefix "k_". Examples: k_Load(...), k_NewImage(...). Macro examples: k_PalGetRVal(...), k_PalCreateRGB(...). If a function is added to a container class the prefix is "k_" for CImage functions, "kB_" for CByteImage, "kF_" for CFloatImage, "kI_" for CIntImage, "kC_" for CComplexImage

2. Each word in multiple word names starts with upper case except in case of abstract names (x,y,cx1 etc). Examples: SearchDirection, FileName and FindAndFollowLowContour(...).

3. Structures begins with a T followed by a name. Examples: TFileInfo, TIntImage. NB: Only use structures when programming in ANSI C.

4. Type definitions (typedef) are upper case only. Examples: UINT8, INT32. Look in the file "ipl98 types.h" for more examples. NB: There is one exception from this rule: The type "bool" which is defined to comply with the C++ standard boolean type.

5. Class names starts with a "C" before the actual name. Examples: CImageGeneral and CImageMatrix.

6. Member variables starts with "m_". Examples: m_Pal and m_ImageInfo.

7. Two Hungarian notaions must be used for variables: "p" for pointers, "fp" for file pointer. Examples:

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UINT8* pPixelStream</td>
<td>“p” for pointer</td>
</tr>
<tr>
<td>FILE *fp</td>
<td>“fp” for file pointer</td>
</tr>
<tr>
<td>UINT8** ppMatrix</td>
<td>pointer pointer to UINT8</td>
</tr>
<tr>
<td>UINT8** ppPixel</td>
<td>member variable of type pointer pointer to UINT8</td>
</tr>
</tbody>
</table>

• **Error messages** All errors must be written by use of the function k_ShowMessage(...). The error type is split in three categories: IPL_ERROR, IPL_WARNING and IPL_NORMAL. The first is used for errors in the program where it is fatal to go on. The warning is used for errors which does not necessarily spoil further execution of the program. The last error type is for future use and can be used by programmer who adds their own code to IPL98 - at the moment no functions or methods uses IPL_NORMAL. An error message must begin with the function name or in case of C++ the class name followed by to "::"-signs and then the method name. It is optional if the argument values are to be printed. In case of C++ bot the class name and method name must be given. After the function name the error message must follow. Examples:

- From the kernel function k_AllocImage(...) where one argument value are written:
11. Adding code to IPL98

TString str;
_k_InitString(&str);
_k_AddFileAndLine(str);
_k_ShowMessage(IPL_ERROR,&str,"k_AllocImage() Bits=%d not supported",Bits);
_k_EmptyString(&str);
return false;

– From the method CopySubImage(...) in the CStdImage class:

TString str;
_k_InitString(&str);
_k_AddFileAndLine(str);
_k_ShowMessage(IPL_ERROR,&str,"CStdImage::CopySubImage: Failed copying subimage");
_k_EmptyString(&str);
return false;

The error handling in the C++ part will probably be changed in the future to take advantage of exception handling but until then this will be the method for error handling.

• **Big structures** If a parameter is a big structure or object let the parameter be transferred as a constant reference. Here is an example:

  C++: Erode(const CStdImage& ImgMask, CStdImage& Img)
  C: k_Erode(const TImage* pMask, TImage* pImg)

  The C++ method transfers the ImgMask parameter as a constant reference and the C function k_Erode transfers pMask as a constant pointer.

  If a function has a source and target image of the same type the code must check if they are the same image. In that case an internal target image should be created and at the end be copied to the image involved. Here is an example how this is done:

```c
bool k_ScaleAndRotateAuto(TImage* pSource,TImage* pDest,...)
{
    TImage* pTarget=pDest;
    bool CopyToSource=false;

    ...

    if (pDest==pSource)
    {
        pTarget=(TImage*) malloc(sizeof(TImage));
        k_InitImage(pTarget);
        CopyToSource=true;
    }
```
11. Adding code to IPL98

.../* all the code now works on pTarget and pSource */
...

if (CopyToSource==true)
{
   k_CopyImage(pDest,pTarget);
   k_EmptyImage(pTarget);
   free(pTarget);
}
...

return true;
}

The function checks if pDest and pSource is the same, in that case the internal image
pointer pTarget are set to point to an internal created image. In the case of two different
images pTarget is initialized to point to the pDest and are not changed then. All the code
operates on what pTarget points to. At the end a copy is performed if an internal image were
created, otherwise nothing is done.

A comprehensive test of the algorithm must be done in order to make it stable in all
situations. It is important to check and remove for memory leaks as even small leaks can be
serious if a program is running for a long time (in a production it may be running for years!).

Adding history

Adding history to a new function is not as easy as it seems although in some situations it is
straight forward. It is recommended to wait with the history untill you are certain that the
algorithm works in all cases. The general rules to follow when adding to the history is:

- All code regarding the history must be between two defines, the first one is #ifdef
  IPL_ADD_HISTORY and the last one #endif. This makes it possible for the user of the
  library to disable the history.

- Code must be added when a function or method changes something in an image.

- Copy the important information given in the parameters into the history, in case of
  CopyConvert() it is the source bit depth, destination bit depth, and the source file
  information.

- Only add to the history if the function succeeds.

- Always add the global char array ipl_HistoryIndent to the beginning of a new history
  line (read more about this below).

There are several problems when adding history:
11. Adding code to IPL98

- Only add history if the function succeeds
- The kernel function may copy between different internal images to calculate its output. That means you must be careful about how the history are copied.
- Making sure the history items are added in a chronological order.

Let's look at an example to show how to avoid the problems. The following shows the lines appended to a history when calling the method CopyConvert(...) on a CImage object:

```cpp
CStdImage::CopyConvert() From 24 b/p to 1 b/p, source image file ...
   k_CopyConvert() From 24 b/p to 1 b/p, source image file ...
   k_CopyImage() Source image file info: D:/temp/circle_2.bmp
```

The first thing to notice is the order of the functions in use and the indentation of lines in subroutines. Two global macros adding and removing a tabulator character from a global char array is available, both macros takes care of allocation and deallocation. The macros are:

- `k_IplStartHistoryMacro()` Adds a tabulator character to the global `ipl_HistoryIndent` char array.
- `k_IplStopHistoryMacro()` Removes a tabulator character from the global `ipl_HistoryIndent` char array.

Every time a function or method begins (after checking that no errors are present) the `k_IplStartHistoryMacro()` macro is called just after adding to the history and right before leaving a function the `k_IplStopHistoryMacro()` is called. Each line added to the history is created with the `ipl_HistoryIndent` array prefix. Here is how it works in the `CStdImage::CopyConvert()` method:

```cpp
bool CImage::CopyConvert(UINT16 Bits, CStdImage& SourceImage) {
#ifdef IPL_ADD_HISTORY
   /* ... adding to source and removing later (hack) */
   TString History;
   k_InitString(&History);
   k_SprintfString(&History, "%sCStdImage::CopyConvert() From ...",
                   ipl_HistoryIndent,...);
   SourceImage.m_History.AppendIPL(History.pChars);
   k_EmptyString(&History);
   k_IplStartHistoryMacro();
#endif
   /* call kernel function */
   bool ReturnValue=
   k_CopyConvert(Bits,&m_Image, SourceImage.GetConstTImagePtr());
   if (ReturnValue==false) {
      TString str;
      k_InitString(&str);
```
11. Adding code to IPL98

```c
k_AddFileAndLine(str);
k_ShowMessage(IPL_ERROR,&str,"CImage::CopyConvert: Failed ... ");
k_EmptyString(&str);
}
#endif
if (&SourceImage!=this)
{
    /* remove the history just added to the source image */
    SourceImage.m_History.RemoveLastLine();
}
/* end history appending (remove a leading '\t' character) */
k_IplStopHistoryMacro();
return ReturnValue;
```

Also notice from the example that when creating the history string in the call to `k_SprintfString(...)` the `ipl_HistoryIndent` is added as a prefix to the string.

Now to the difficult part! In the C++ part it is not possible to add the history to the target ('this') object because it is emptied in the kernel function. This is solved by adding to the source instead, and the kernel function takes care of copying the source history to destination. But then we have to remember to remove the history from the source again before leaving `CopyConver()`. But there is one more thing to take into consideration. If the source and destination is the same we actually adds to the target image in the C++ part. In that case we should not remove the line when leaving the C++ method. This is a bit complex and is not an elegant way of solving the problem so if anyone comes up with a better solution please let the developer of IPL98 know. To get more used to adding history it is highly recommended to look at some of the code in the library.

**Code recommendations**

Here is a list of recommendations:

- **CStdImage** If a method can work on both a CImage or CByteImage object let the method parameter be a CStdImage object, the method `Erode(...)` shown above is an example of this.

- **Speed optimization** Optimize the most time consuming parts of the code.

- **Extra comments** Write comments in the code to explain the not so obvious parts of the code.

Here are two examples on how to optimize code. It is always recommended to start out with the safe functions which perform a range check. When you are sure the algorithm works optimization like the ones given here can be done.

1. Fast conversion from 24 b/p to 8 b/p:
11. Adding code to IPL98

```c
for (y=0;y<h;y++)
{
    for (x=0;x<w;x++)
    {
        PalValue=k_GetPixel24bp(x,y,*pSource);
        Histogram[(UINT8)((int)((k_PalGetRVal(PalValue)+
                         k_PalGetGVal(PalValue)+
                         k_PalGetBVal(PalValue))/3+0.5))++;
    }
}
```

2. Using a border to avoid edge problems when working with a mask (about 3 times faster):

```c
k_InitImage(&Temp);
k_AllocImage(W,H,8,&Temp);
k_SetPeriodicBorder(BorderSize,pImg); /*** setting border ***/
for (y1=0;y1<H;y1++)
{
    for (x1=0;x1<W;x1++)
    {
        sum=0;
        for (y=0;y<h;y++)
        {
            for (x=0;x<w;x++)
            {
                sum+=pImg->ppMatrix[y1-y][x1-x]*pMask->ppMatrix[y][x];
            }
        }
        Temp.ppMatrix[y1][x1]=(UINT8)sum;
    }
}
```

This can be optimized to:
new optimized code

```
k_InitImage(&Temp);
k_AllocImage(W,H,8,&Temp);
for (y1=0;y1<H;y1++)
{
    for (x1=0;x1<W;x1++)
    {
        sum=0;
        for (y=0;y<h;y++)
        {
            for (x=0;x<w;x++)
            {
                q=pMask->ppMatrix[y][x];
                if (q!=0)
                {
                    if (x<=x1)
                        x2=x1-x;
                    else
                        x2=W+x1-x;
                    if (y<=y1)
                        y2=y1-y;
                    else
                        y2=H+y1-y;
                    sum+=pImg->ppMatrix[y2][x2]*q;
                }
            }
        }
        Temp.ppMatrix[y1][x1]=(UINT8)sum;
    }
}/
```
A. Appendix: C code from C++ examples

This appendix gives the C code corresponding to many of the given C++ examples.

A.1 Create an image in RAM

C code corresponding to the C++ code given at page 29 in section 7.1:

```c
#include <ipl98/ipl98.h>

void main()
{
    TImage Image1; /* image structure, no memory allocated for image data */
    TImage Image2, Image3;

    TImage ByteImage1; /* In C use the same basic structure with */
    /* a special set of functions */
    TImage ByteImage2, ByteImage3;

    TFloatImage FloatImage1; /* image structure used for float images */
    TFloatImage FloatImage2;

    TIntImage IntImage1; /* image structure used for float images */
    TIntImage IntImage2;

    k_InitImage(&Image1); /* always initialize structure before using it */
    k_InitImage(&Image2);
    k_AllocImage(768,512,8,&Image2); /* allocate image with dimensions */
    /* (768,512) and 8 b/p */

    k_InitImage(&Image3);
    k_CopyImage(&Image3,&Image2); /* copy contents of Image2 to Image3 */

    k_InitImage(&ByteImage1); /* always init structure before using it */
    k_InitImage(&ByteImage2);
    kB_AllocImage(768,512,&ByteImage2); /* allocate image with dimensions */
    /* (768,512), always 8 b/p when using a kB_ function */

    kB_InitImage(&ByteImage3);
    kB_CopyImage(&ByteImage2,&ByteImage1); /* copy contents of ByteImage1 */
    /* to ByteImage2 */

    kF_InitImage(&FloatImage1); /* always initialize structure before */
    /* using it */
    kF_InitImage(&FloatImage2);
    kF_AllocImage(10,10,&FloatImage1); /* allocate float image with */
    /* dimensions (10,10) */
}```
A.2 Manipulate an image in RAM

void main()
{
    TImage Img1;
    TImage Img2;
    TImage ByteImage; /* only using this image with 8 b/p */
    k_InitImage(&Img1);
    k_InitImage(&Img2);
    kB_InitImage(&ByteImage);

    k_AllocImage(768,512,24,&Img1); /* allocating 768*512 24 b/p image */
    k_AllocImage(768,512,8,&Img2); /* allocating 768*512 8 b/p image */

    k_CopyConvert(1,&Img2,&Img1); /* copy Img1 to Img2 and convert it */
    /* from 24 b/p to 1 b/p */
    k_PrintImageInfo(&Img1);
    printf("\n\n");
    k_CopyConvert(1,&Img1,&Img1); /* convert Img1 from 24 b/p to 1 b/p */
    /* this is a bit slower */
    printf("After converting Img1: \n");
    k_PrintImageInfo(&Img1);
    printf("\n");

    /* using the normal k_ kernel functions when working with a ByteImage */
    /* the user must be careful only to use 8 as bit depth */
    k_CopyConvert(8,&ByteImage,&Img1); /* copy Img1 to ByteImage and */
    /* convert to 8 b/p. */
    k_CopyConvert(1,&ByteImage,&Img1); /* error, CByteImage needs 8 b/p. */
}
A.3 Segmentation

C code corresponding to the C++ code given at page 47 in section 8.1.1:

```c
void main()
{
    TImage Img, ImgDest; /* Images */
    TPixelGroups PixelGroups; /* Holds information about blobs */
    unsigned int BlobIndex;
    unsigned int LastPos;
    clock_t t = k_StartTimer(); /* stop watch for time elapsed */
    T2DInt pnt;
    unsigned int PosIndex;

    k_InitGroups(&PixelGroups); /* always initialise the TPixelGroups structure! */
    k_InitImage(&Img); /* always initialise the TImage structure! */
    k_InitImage(&ImgDest);

    if (k_Load("c:/temp/three_segments.bmp", &Img)) /* your image with 8 bits/pixel */
    {
        k_PrintImageInfo(&Img);

        /* Find the blobs in image */
        k_DeriveBlobs(&Img, &PixelGroups, HIGHCOLOR, 128, EIGHTCONNECTED);

        /* Ask which blob pixel position (30,30) belongs to */
        pnt.x = 30;
        pnt.y = 30;
        if (k_GetGroupWithPos(&PixelGroups, &BlobIndex, pnt, &PosIndex))
        {
            LastPos = PixelGroups.pGroups[BlobIndex].NumberOfPixels;
            printf("Pixel at pos (%d,%d) in blob %d\n", pnt.x, pnt.y, BlobIndex);
            printf("First pixel (%d,%d)\n", PixelGroups.pGroups[BlobIndex].pPos[0].x,
                       PixelGroups.pGroups[BlobIndex].pPos[0].y);
            printf("Last pixel (%d,%d)\n", PixelGroups.pGroups[BlobIndex].pPos[LastPos-1].x,
                       PixelGroups.pGroups[BlobIndex].pPos[LastPos-1].y);
            printf("Total pixels in blob: %d\n", PixelGroups.pGroups[BlobIndex].NumberOfPixels);
        }

        /* Remove a blob from the list of groups */
        printf("Total groups now: %u\nRemoving one group\n", PixelGroups.NumberOfGroups);
        k_RemoveGroup(&PixelGroups, 0);
        printf("Total groups now: %d\n", PixelGroups.NumberOfGroups);

        if (PixelGroups.NumberOfGroups > 1)
        {
            /* Copy blob number 1 to gray tone image and write to disk (c:/temp/test.bmp) */
            k_AddColorsToGroups(&Img, &PixelGroups); /* first add colors */
        }
    }
}
```
A.4 Morphological operations

C code corresponding to the C++ code given at page 55 in section 8.4.1:

```c
void main()
{
    char* dty=NULL; // Holding the date, time and year
    clock_t time;
    TImage Img, ImgDest, ImgMask;
    unsigned int count;
    k_InitImage(&Img); /* always initialise the TImage structure! */
    k_InitImage(&ImgDest);
    k_InitImage(&ImgMask);

    /* set a 3*3 mask */
    k_AllocImage (3,3,1,&ImgMask);
    k_FlushImage(1,&ImgMask);

    /* load image */
    if (k_Load("c:/temp/doggie.bmp", &Img))
    {
        k_PrintImageInfo(&Img);
        printf("\n");

        /* an example of using the timer and date functions */
        printf("On a 400 MHz Pentium II this program takes"
            "0.83 seconds to complete\n");
        time=k_StartTimer();

        /* convert to 1 b/p */
```
A.5 Corresponding 3D-2D points

C code corresponding to the C++ code given in section 69 page 69:

```c
#include <stdio.h> #include <stdlib.h>

int main()
{
    // Convert image from 8 b/p to 1 b/p
    printf("Converting image from %d b/p to 1 b/p\n", Img.Bits);
    k_CopyConvert (1,&ImgDest,&Img);

    /* invert image */
    printf("Inverting image\n");
    k_InvertImage(&ImgDest);

    /* dilate */
    printf("Dilate image 5 times\n");
    for(count=0; count<5; count++)
    {
        k_Dilate(&ImgMask,&ImgDest);
        /* replace the next line with the one above to make it faster */
        /*k_DilateFast(&ImgDest,7,7,7);*/
    }

    printf("Erode image 5 times\n");
    for(count=0; count<5; count++)
    {
        k_Erode(&ImgMask,&ImgDest);
        /* replace the next line with the one above to make it faster */
        /*k_ErodeFast(&ImgDest,7,7,7);*/
    }

    printf("Saving image as "c:/temp/test.bmp"
");
    k_Save("c:/temp/test.bmp",&ImgDest);

    if (!k_GetDateTimeYear(&dty))
        printf("Failed reading date and time....\n");
    else
        printf("This program was run at: %s\n",dty);

    printf("Seconds elapsed: %f\n",k_GetTimer(time));

    if(getchar()){ }
}
```

A.5 Corresponding 3D-2D points

C code corresponding to the C++ code given in section 69 page 69:
A.5 Corresponding 3D-2D points

```c
T3D2DPoints PointSet,PointSet2;
T2DFloat P2d;
T3DFloat P3d;
int i;
clock_t t = k_StartTimer(); /* stop watch for time elapsed */
k_Init3D2DPoints(&PointSet);
k_Init3D2DPoints(&PointSet2);
for(i=0;i<10;i++)
{
    P2d.x=(float)(i+(float)0.9);
P2d.y=(float)i;
P3d.x=(float)i;
P3d.y=(float)i;
P3d.z=(float)45.6346;
    k_AddPointSetAndIdTo3D2DPoints(&P3d,&P2d,i,&PointSet);
}
k_Copy3D2DPoints(&PointSet2,&PointSet);
k_Empty3D2DPoints(&PointSet2);
k_ReAlloc3D2DPoints(&PointSet2,5); /* example on using realloc */
    /* for the T3D2DPoints structure */
k_Print3D2DPoints(&PointSet2);
printf(" NumberOfPointSets=%d AllocatedPointSets=%d\n",
    PointSet2.NumberOfPointSets,PointSet2.AllocatedPointSets);
printf("Total time: %f\n",k_GetTimer(t));
if (getchar()){}
return 0;
}
```
A.5 Corresponding 3D-2D points
References


Index

<< with CPerspective class, 58
<< with image, 31
<< with pixelgroup, 43
<<, 18
>>, 91
= operator, 34
2D to 3D, 58
3D to 2D, 58

Acquire(...), 72
AddColors(...), 41
AddDate(...), 25
AddGroup(...), 45
adding algorithms to IPL98, 74
AddPosition(...), 41
AddTime(...), 25
AddToImage(...), 41
AddTrailingSlashToPath, 67
AdjustSizeOfPositions(), 41
ANTICLOCKWISE, 49
Append(...), 25
AppendIPL(...), 25
assignment operator, 34

blob, 47
BMP, 32
border, 31
Borland C++ Builder, 6, 14
ByteWidth, 31

C IPL98 and the C kernel, 6
C using the C part only, 12
C++ and C in IPL98, 6
C++ using the C++ part, 12
Calc3Dto2D(...), 58
Calibrate(...), 58
camera calibration, 58
camera hardware, 71
camera matrix, 58
CArray2D<T>, 27
CArrayOS2D<T>, 28
CByteImage, 28
CCameraFG, 72
CCameraFireWire, 71
CCameraGeneric, 71
CCameraUSB, 71
CCoordinateTransform, 51
CCorresponding3D2DPoints, 69
CFeatures, 52
CFloatImage file format, 32
CGraphics, 68
CHoughProbRHTLine, 47
CHoughRHTCircle, 47
CHoughRHTLine, 47
CHoughSHTLine, 47
CImage, 28
ClntImage file format, 32
circles, 68
CircleW, 68
CLOCKWISE, 49
CMorphology, 55
ColorsAvailable(), 41
COLORTYPE, 47
CompareNoCase, 68
compiling IPL98, 12
CONNECTIVITY, 47
container classes, 27
ContinueTimer(), 22
countour tracing, 49
convert bit depth, 34
ConvertBackslashToSlash, 67
ConvertLinesToComments, 68
coordinate transformation, 51
CopyConvert(...), 34
CopySubImage(...), 35
CopyToImage(...), 41
CPalette, 35
CPerspective, 58
CPixelGroup, 40
CPoint2D, 20
CPoint3D, 20
CreateRGB(...), 37
CSegmentate, 47
CSpectral, 47
CText, 25
CTimeDate, 22
CTopography, 47
date, 22
DegreeToRad, 67
DeriveBlobs(...), 47
DeriveCentralMoments(), 53
DeriveCentralNormMoments(), 53
DeriveHuFromCentralMom(), 53
DeriveHuFromCentralNormMom(), 54
DeriveMomentsBinary(...), 53
DeriveMomentsGrayTone(...), 53
dilate, 55
DilateFast(...), 55
DIRECTION, 49
Direction(...), 58
DISABLE_IPL_ERROR, 23
DISABLE_IPL_NORMAL, 23
DISABLE_IPL_WARNING, 23
DOC++, 74
EAST, 49
EIGHTCONNECTED, 47
Empty(), 30
environments, 13
erode, 55
ErodeFast(...), 55
error messages, 23
error handling, 23
error redirecting, 24
exception handling, 23
feature extraction, 52
file formats, 32
Find(...), 25
FindAndFollowHighContour(...), 49
FindAndFollowLowContour(...), 49
FlipHorizontal(), 40
FlipVertical(), 40
FLOAT32, 7
fonts available, 68
FOURCONNECTED, 47
g++, 15
gcc, 15
GetBlueVal(...), 35
GetCamMatrix(...), 58
GetColor(...), 35, 41
GetGreenVal(...), 35
GetGroupWithPos(...), 45
GetMaxX(), 36
GetMaxY(), 36
GetMinX(), 36
GetMinY(), 36
GetPixel(...), 36
GetPixelFast(...), 36
GetRedVal(...), 36
GetTimer(), 22
GNU g++, 15
Gonzales & Woods, 2
graphics file formats, 32
graphics in images, 68
Graphics User Interface, 6
GUI, 6
hardware dependency, 11
HIGHCOLOR, 47
history, 24
HoldTimer(), 22
home page, 2
image << operator, 31
image Classes available, 27
image Creating an image, 27
image load, 32
image representation in RAM, 7
image save, 32
image show, 32
including correct header file, 29
InsertGroupInGroups(...), 45
InsertPosition(...), 41
Installing IPL98, 11
INT16, 7
INT32, 7
INT8, 7
Invert(), 40
IPL98 file structure, 8
IPL98 home page, 2
IPL98 version, 18
iterating a whole image, 40
iterator, 37
kernel functions, 6
Line, 68
lines, 68
Linux, 15
Load(...), 32
LOWCOLOR, 47
INDEX

m_History, 25
Max(...), 67
maximum value, 67
Min(...), 67
minimum value, 67
moments, 53
morphology, 55

namespace tricks, 37
naming conventions, 76
Next(...), 25
nomenclature, 76
NORTH, 49

Origin, 31
origo, 36

palette, 35
perspective camera transformation, 58
PGM, 32
PI, 67
PI2, 67
pinhole, 58
Pinhole(...), 58
pixel, 35
pixelgroup, 40
PlotString, 68
points, 20
PosInGroup(...), 45
PrintInfo(...), 18

RadToDegree, 67
Region Of Interest, 36
releases, 4
RemoveColors(), 41
RemoveGroup(...), 45
RemoveGroupSlow(...), 45
RemovePosition(...), 41
RemovePositionSlow(...), 41
return values, 23
RGB, 35
ROI, 36
rotate, 51
Rotate90(), 40
ROTATION, 49
Round(...), 67
rounding off value, 67
rounding off UINT8, 67
RoundUINT8(...), 67

Save(...), 32
scale, 51
ScaleAndRotateAuto(...), 51
ScaleAndRotateFixed(...), 51
ScaleAuto(...), 51
ScaleFixed(...), 51
SecondsToHMS(...), 22
segmentation, 47
SetColor(...), 41
SetOrigo(), 36
SetPixel(...), 36
SetPixelFast(...), 36
show an image on screen, 32
SizeImage, 31
skeleton, 57
Skeletonizing(...), 57
SkeletonZhou(...), 57
SkipSpaceAndComments, 67
SOUTH, 49
SplitFileName, 67
Sqr(...), 67
square, 67
StartTimer(), 22
StopTimer(), 22
stream, 18
streaming a UINT8 type, 36
streaming operator, 31
structure, 5
swap two values, 67
Swap(...), 67

thinning, 57
Thinning(...), 57
TImage C structure, 7
timer, 22
ToUpperCase, 68
types in IPL98, 7

UINT16, 7
UINT32, 7
UINT8, 7
UNIX, 15
utilities, 67

version, 2, 4, 18
Visual C++, 6, 13
warnings, 23
WEST, 49
writing new algorithms to IPL98, 74