Introduction and Tutorial to the Image Library

Companion to the book
AN INTRODUCTION TO 3D COMPUTER VISION TECHNIQUES AND ALGORITHMS
by Bogusław Cyganek & J. Paul Siebert, Wiley 2009

Copyright 2009 – Bogusław Cyganek
This document is intended only as a companion to the book *An Introduction to 3D Computer Vision Techniques and Algorithms* by B. Cyganek and J.P. Siebert (Wiley, 2009) [2]. Its main objective is to provide a tutorial and in-depth insight into the software library attached to the book (available from the book web page [7]) which was published in an electronic version to keep track of possible changes and updates to the software platform. For better understanding the reader is recommended to read Chapters 3, 4, 6, and 13 of the book [2] before reading this document.
# Table of Contents

1 Introduction ........................................................................................................... 6
   1.1 What this Technical Report Contains ................................................................. 6
   1.2 Genesis of the Library ......................................................................................... 6
   1.3 Software Copyright Note ..................................................................................... 6

2 Getting Started ....................................................................................................... 7
   2.1 User Requirements ............................................................................................. 7
   2.2 System Requirements ......................................................................................... 7
   2.3 Development Platform ....................................................................................... 8
   2.4 Installation and First Build ................................................................................ 9
      2.4.1 Dealing with the Most Common Problems .................................................... 10
      2.4.2 Setting Parameters of the Build .................................................................. 11
   2.5 System Debugging ............................................................................................. 12

3 Exemplary Projects ................................................................................................. 14
   3.1 Preliminary Readings ......................................................................................... 14
   3.2 Getting Started with the Library – Running in a Console ................................. 14
      3.2.1 Creating and Manipulating Images ............................................................... 15
      3.2.2 Input-Output of Image Objects ................................................................... 17
      3.2.3 Video Streams ............................................................................................ 18
      3.2.4 Basic Operations on Images ...................................................................... 19
   3.3 Test Application for Windows ........................................................................... 23
   3.4 The SimpleFromTheBook Project .................................................................. 24

4 Connection to Existing Platforms ......................................................................... 26
   4.1 Using HIL in Other Projects ............................................................................. 26
   4.2 Cooperation with the Open CV Library ............................................................ 27

5 Fixed-Point Representation of Pixels ...................................................................... 28
   5.1 Basic information .............................................................................................. 28
   5.2 Software Model for Fixed-Point Data Representation ....................................... 29
5.3 Mathematical Routines for Fixed-Point Representation .............................................................. 31

6 More on Image Models ................................................................................................................... 34
   6.1 Images with Fixed-Point Pixels ............................................................................................... 34
   6.2 Image Specializations for Boolean Pixels – Masked Images .................................................. 35
   6.3 Memory Allocators for Objects .............................................................................................. 36
   6.4 Representation of Video Objects ........................................................................................... 39

7 More on Software Architecture of the Library ............................................................................. 44
   7.1 Exception Handling Hierarchy ............................................................................................... 44
   7.2 Advanced Image Operations ................................................................................................. 45
      7.2.1 Call Mechanisms for Image Operations ............................................................................. 45
      7.2.2 The Class for Unary Image Operations ............................................................................ 50
      7.2.3 The Classes for Multi Image Operations ......................................................................... 51
      7.2.4 The Composition of Image Operations ............................................................................. 53
      7.2.5 Arithmetic Operations on Images ................................................................................... 54
      7.2.6 Logical Operations on Images ....................................................................................... 55
      7.2.7 Image Format Converters ............................................................................................... 56
      7.2.8 Colour Space Conversions ............................................................................................ 56
      7.2.9 Global Operations in Images ........................................................................................ 57

Bibliography ......................................................................................................................................... 59
## List of Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>HIL</td>
<td>Hardware Image Library</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Platform</td>
</tr>
<tr>
<td>OOD</td>
<td>Object-Oriented Design</td>
</tr>
<tr>
<td>OOP</td>
<td>Object-Oriented Programming</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>STL</td>
<td>Standard Template Library</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 What this Technical Report Contains

This document is a supplement to the book An Introduction to 3D Computer Vision Techniques and Algorithms by Bogusław Cyganek & J. Paul Siebert (Wiley, 2008) [2]. It contains additional information on inner mechanisms of the library, as well as on practical use of the software platform provided for readers of the book. Therefore it should be read after reading at least Chapter 3 of the aforementioned book which contains information on basic data structures used in the presented library. Because of this, this technical report can be treated as a ‘dynamic’ contents of Chapter 14 of the book [2].

1.2 Genesis of the Library

The library was designed to facilitate image processing and computer vision tasks from external applications. One of the main assumptions was to create a highly object-oriented platform that would separate interfacing from implementation. The main goal of this approach is to allow evolution of the implementation layers, for instance to take advantage of the new computing capabilities introduced by reprogrammable hardware. Thus, a platform could ‘decide’ in a run-time which implementation module is selected for execution: it is hardware if such a functionality is currently available. Otherwise a software implementation is chosen. An example of advanced image acceleration hardware is a family of image processing boards developed by the Pandora Inc. These were used to steer development of the hardware dependent layers of the library. Because of the aforementioned assumptions the library was named Hardware Image Library (HIL). At the same time the book [2] had been growing which, apart from the theoretical sections, contains dozens of image processing algorithms. Therefore to test both, i.e. the methods presented in the book and performance of the library, this library was also chosen as a development platform for computer vision methods presented in the book.

1.3 Software Copyright Note

The software [7] accompanying the book An Introduction to 3D Computer Vision Techniques and Algorithms by Bogusław Cyganek & J. Paul Siebert (Wiley, 2009) [2] is copyright. It is supplied only for educational and/or academic purposes and to accompany the book. All other commercial and/or non-private applications require written permission. The software is supplied as it is without any guarantees or responsibility for its use in any application.
2 Getting Started

The goal of this chapter is to provide basic information necessary to install, build, run, and debug the projects included in the software available from the web site of the book [7].

2.1 User Requirements

A basic knowledge of C++ is required to compile and run examples included in this document, as well as to use the library in the user’s applications. There are three projects attached to the book, each at a different level of expertise in C++ and Win32. For tutorial purposes the best start-up is probably the SimpleFromTheBook project which is discussed in (3.4). It is the shortest one and presents procedures in the same format as they appear in the book [2]. As already pointed out, the acquaintance with the contents of the Chapters 3, 4, 6, and 13th of the book [2] is a prerequisite to using the software discussed in this document.

There are many sources of information on C++, templates, and Standard Template Library (STL). The book The C++ Programming Language by Stroustrup can be recommended for all C++ programmers [19]. It can be read first as a tutorial, and then serve as a reference to the language (although in this role The Annotated C++ Reference Manual is much better [3]). For a brief introduction to C++ the Essential C++ by Lippman can be recommended [15]. One of the best books dealing with C++ templates is the C++ Templates by Vandervoorde & Josuttis [20], while in-depth knowledge on STL is provided in The C++ Standard Library by Josuttis [12]. Design patterns are explained in the classical book by Gamma et al. [4]. The book by Alexandrescu [1] provides in-depth information of generic programming and application of advanced patterns. Finally, Chapter 13 of the book [2] provides an introduction to programming of the visual systems.

2.2 System Requirements

The software was developed and tested using the Microsoft Visual C++ 6.0, run on the Windows XP Professional. Then it was ported to Microsoft .NET 2005. Therefore to run the software (with no need for porting) the following platforms are required:

1. Operating System – Microsoft Windows XP Home, Professional, Windows Vista
2. Microsoft Visual C++ 6.0 (with Service Pack 2.0) or
3. Microsoft .NET 2005 or higher.

Although developed for Windows one of the main assumptions was to use the standard C++ so porting to other platforms should not cause any serious problems (section 4).
2.3 Development Platform

The Hardware Image Library (HIL) contains about seventy files written in standard C++. Due to template classes the majority of the files are the header files (.h) with class definitions. Some parts of implementation fall into source (.cpp) files, however. To use the library in an existing project both groups should be added to this project (preferably in a separate folder). More details on project integration are provided in Chapter 4 of this document.

Figure 1 depicts the development platform with the library project open. This can guide software organization in other projects. Figure 2 shows the project in the newer .NET 2005 platform.

For development the Windows XP Professional operating system was used. The library is written in object-oriented fashion in C++ with Standard Template Libraries (STL). It was noticed, however, that
Visual C++ 6.0 does not support some of the C++ template techniques, such as template template pattern, etc. Therefore the Visual C++ 6.0 platform is treated as the basic development platform to facilitate software porting process to other programming platforms – see also section 2.4.2.

Both platforms are complete development environments with editor, compiler, linker and many other tools. More details are easily accessible through MSDN [8]. Although in-depth knowledge of myriads of features can help in development of advanced computer system to start building new projects it is sufficient to go through the project wizard and follow some basic platform specifics, such as the precompiled headers.

### 2.4 Installation and First Build

Installation consists in downloading the .zip'ped repository and unpacking it into your disk. Make sure that after this all files have the read/write attribute set to ‘on’.

The projects require installed Microsoft Visual C++ 6.0 or the Microsoft .NET 2005 development platforms mentioned in the previous section. Other IDEs can also be used. These, however, can require some changes to the organization of the project and/or minor modifications to the source files.

After installing, load the simplest *SimpleFromTheBook* project into your IDE (described in section 3.4), then launch “Build” and then “Run” commands. The project should get built and executed. If there are some problems with this, please check the hints from the next section.
2.4.1 Dealing with the Most Common Problems

The software was designed to make it easy to install, run and extend. Code is also as much as possible self documented. However, unfortunately and despite many efforts it is quite common that if a software platform is installed in a new environment some discrepancies can arise. The following table contains some common problems that can arise when dealing with software in new programming environments and how to try to solve them.

<table>
<thead>
<tr>
<th>Problem description</th>
<th>Possible solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cannot find a file</td>
<td>If this is one of the project files (such as .h or .cpp) then the first thing to do is try to find it in the system with the help of the file search application (WinKey + F). If present in some folders then the problem lies probably in the project settings of additional directories. These can be viewed by launching the Project Settings dialog (Alt+F7) in the IDE, then going to the C/C++ tab and choosing the “Preprocessor” entry in the “Category” combo box. The “Additional Include Directories” should contain relative paths to the folders which contain the project files. However, if a file cannot be found in the system, then the installation process should be repeated. In case of persistent problems the project .zip file should be searched for the missing file and/or another computer can be tried for verification.</td>
</tr>
<tr>
<td>Cannot link</td>
<td>This type of error happens if some object files (i.e. the compiled source .cpp files) or external libraries are missing. The former requires compilation of a reported file (re-build of the whole project can be recommended in such a case). The latter means that either project settings are incorrect and should be amended or a file is missing (see previous comment).</td>
</tr>
<tr>
<td>Precompiled headers</td>
<td>Precompiled headers are meant to speed up compilation by placing the majority of the project headers in one file which is then compiled once. However, this is also a feature of the VC and .NET (and some other) platforms which ties the project to this specific setting. Therefore the provided vision library does not use the precompiled headers. However, when a new file is added, by default it is set to use a precompiled header. The option of using or not using the precompiled headers can be controlled in the Project Settings dialog – invoked in VC and .NET by pressing Alt+F7.</td>
</tr>
</tbody>
</table>
If this happens then at first be sure that the latest Service Packs for system and compiler have been installed. Certainly this category is very broad and mostly is due to break(s) in the syntax or visibility of some variables or other language entities. If an error of this type happens in the library then the code should be checked to see whether it has been inadvertently changed.

Apart from the syntax errors this type of message happens if some `#include` are missing. In such a case find a reported function, class or variable and then `#include a .h` file in which it has been found. If errors of this type are persistent in some parts of a project then such problematic code lines can be temporarily excluded from the build by commenting them out or by placing the conditional preprocessor statement:
```cpp
#if 0
    // the-troublesome-lines-here
#endif
```

Sometimes it happens that the excluded code fragment is not necessary at all for some tests we wish to run at a moment. Such situations can be verified by placing in debug mode the `REQUIRE( false )` just before the excluded fragment. This will alert if the excluded lines are to be called. Then after other parts of the project are successively compiled we can return and work on the previously excluded fragment to make it compile.

---

Using the library in other programming environments should not pose problems if the other platform is endowed with a compiler that complies with the C++ standard, this concerns especially its ability to compile some template constructions. For this reasons the library has the `MODERN_COMPILER` flag defined which should be set to 0 if a compiler has some problems with more advanced template structures (such as partially specialized classes, etc.). See also the next section.

We are well aware that despite our best efforts the software provided surely is not free of bugs or code fragments which could be made faster or more readable. Apologizing for this we ask for reports and constructive help to make the platform better.

### 2.4.2 Setting Parameters of the Build

When using the library the following statement should be placed before the other code fragments

```cpp
using namespace PHIL;
```

Table 2 presents the most important preprocessor flags of the project.
### Table 2. The most important preprocessor flags and their meaning

<table>
<thead>
<tr>
<th>Preprocessor flag</th>
<th>In file</th>
<th>Meaning</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONNECT__OPEN_CV</td>
<td>IntelLib_ImageConverters.h</td>
<td>Can be set only if the Intel’s Open CV library is installed in your system. Then, if set, the copy functions to and from the OpenCV image representation are included in the project. This allows exchange of images between the two platforms (section 4.2).</td>
<td>0</td>
</tr>
<tr>
<td>MODERN_COMPILER</td>
<td>ImageDeclarations.h</td>
<td>If set then partial template specialization is included in the build. Some compilers cannot cope with such constructions.</td>
<td>0</td>
</tr>
</tbody>
</table>

To facilitate connection of the library to the projects which utilize the very popular OpenCV library, some software bridges were created. These are under the suite of MixedCopy() functions which allow conversion of the formats of images from the two libraries. More information may be found in section 4.2.

### 2.5 System Debugging

Writing reliable software belongs to hard tasks. Section 13.2.3 in the book provides some information on asserting code correctness which should be a primary goal from the very beginning of software development. Once a version of a code is written is should be thoroughly debugged. For this purpose the debugging tools should be used which are offered by all software development platforms. For instance, in the assumed Microsoft platforms (VC, .NET) the breakpoints can be set by placing the cursor in a code line intended to be stopped, then choosing from a menu or by pressing F9. Apart from a line of a breakpoint the stop conditions for this breakpoint can also be specified. These are accessed from the Breakpoints… dialog which is called by Alt+F9 (or from the Edit menu). The following figure present a view of the development platform during software debugging.
A good hint is never to run your newly built code for the first time. Instead step through its line codes checking variables and execution flow. After asserting that the code runs as expected we have to check its performance. For this purpose some test data is necessary. However, preparing meaningful tests is again not easy. More information on development of computer vision platforms is given in Chapter 13 of the book [2].
3 Exemplary Projects

Software accompanying the book can be downloaded from the Internet [7]. The goal of this chapter is to give a short and practical introduction to practical use of the library. The library can be used as a main platform (so called console mode) to which we add specific functionality, or it can be added to other bigger projects that make use of image operations or computer vision functionality.

In this chapter we discuss three types of projects available on the web page of the book [7]:

1. Library in the console mode. In this mode the library is augmented with some functions that allow testing of its basic operations.
2. The simple multi document Win32 application that has the library attached to it. The application allows image operation in a classical multi document environment of the Win32 operating system.
3. The minimal project that contains the software procedures in exactly the same format as presented in the book. Such a simple structure of code in the book was chosen to facilitate explanation of basic algorithms. In this project we augment them with some additional interfacing functions so they can run in a real application.

3.1 Preliminary Readings

To understand the methods implemented in the provided library the preliminary reading is Chapter 3 of the book An Introduction to 3D Computer Vision Techniques and Algorithms [2]. It provides basic information on class hierarchies for representation of such fundamental objects as pixels, images, and operations on these.

Then, Chapter 4 of the book should be read or at least skimmed for better orientation. It contains information on basic image operations, such as linear and nonlinear filtering, structural tensor and feature detectors (edges, corners). Chapter 5 reports on image pyramids for scale-space processing. Chapter 6 contains information of software modules for nonparametric (Census and Rank) and log-polar transformations of images, as well as on stereo matching. Software modules for image warping are dealt with in Chapter 12. Finally, Chapter 13 of the book [2] discusses programming techniques for image processing and computer vision.

3.2 Getting Started with the Library – Running in a Console

In the console mode we focus on the functionality of the library minimizing interaction with the operating system at the same time. Thus, the project is minimal in the sense of a number of included project files. At the same time it allows testing of all simple and advanced functions of the library.

In the following sections we provide a brief tutorial on the library. It should be read and tested in the real project. After finishing this tutorial an interested reader should be able to develop his or her own computer vision projects with the help of this library.
3.2.1 Creating and Manipulating Images

Pixel and image objects create a foundation of the library. Thus, knowledge of their structure and behaviour is important for understanding and usage of the whole library. In the following few steps we will show how to create some images and then how to perform actions with them.

At first let us create an image, called “theImage”, with resolution of 100x200 of monochrome pixels (i.e. with depth of 8 bits). This can be done as follows:

```c++
TImageFor< unsigned char > theImage( 100, 200, 0 ); // an image 100x200 filled with 0
```

Additionally, we require to initialize each pixel with a value 0 just after the image had been created. The above can be written in an equivalent but shorter form thanks to the predefined types as

```c++
MonochromeImage theImage( 100, 200, 0 );  // ... the same as above
```

Having defined an image we can try the most common operations on it, i.e. setting and getting a value of a pixel at a certain location within its discrete space. See the following exemplary lines

```c++
theImage.SetPixel( 1, 2, 22 );
unsigned char thePixel = theImage.GetPixel( 1, 2 );   // should read 22
```

Once again the last line can be written more ‘generically’ using internal definition of a pixel put in an image class, that is

```c++
MonochromeImage::PixelType thePixel2 = theImage.GetPixel( 1, 2 );
```

That is, instead of explicitly stating type of a pixel, which in our case is “unsigned char” or a byte, we simply write `whateverImageType::PixelType` and a type of pixel for that image is used automatically (in our case `whateverImageType` is certainly the `MonochromeImage`). However, let us remember to check the range of a pixel position as, if wrong, an exception can be generated. That is, in our example the maximum pixel position (for the bottom right one) is `GetPixel( 99, 199 )` since the range of valid pixel positions is 0–99 and 0–199 for columns and rows, respectively.

The other two most common image operations are accessing its dimensions if not known a priori. This can be done with the following two functions:

```c++
int num_of_image_columns = theImage.GetCol(); // to get number of columns in the image
int num_of_image_rows    = theImage.GetRow(); // to get number of rows in the image
```

The last group of common actions which we discuss here are the arithmetic operations on images. These are directly defined in the image class, so we can easily call code like the following one

```c++
// Two 100x200 images with double pixels, all set to pi and -pi at creation
TRealImage theRealImage_1( 100, 200, kPi ), theRealImage_2( 100, 200, -kPi );
// we can do some "classic" operations with images of the same size
theRealImage_1 += theRealImage_2;
```

The result of the last line is that all pixels in the `theRealImage_1` get value 0. The other ‘common’ operations are defined in the file `GraphicLibrary.h`.

Another type of operation is conversion from one pixel representation to the other as in the following

```c++
// Conversions between different types of images can be done
```
Here with help comes the group of overloaded MixedCopy() functions that, if images are of the same size and pixels can be converted from one format to the other, perform a copy and pixel-change operation.

However, images are not limited to store only simple data types like bytes or doubles. They can store almost any data, also defined by yourself. See the following code:

```cpp
struct MyContainer
{
  int  fFirstNumber;
  char fStringTable[ 10 ];
  double fSecondNumber;
  MyContainer( void );  // a default constructor – write its body
};
```

and then the image with pixels being of type `MyContainer`

```cpp
TImageFor< MyContainer > theSpecImage( 128, 256 );
```

In the above each pixel, i.e. an object of the `MyContainer` type, will be initialized with whatever value will be set in its default constructor. The somewhat clumsy template definition can be shortened by suitable typedef, like the following one

```cpp
typedef TImageFor< MyContainer > SpecialPixelImage;
```

which is useful especially if that type tends to be applied frequently in other parts of the code.

Finally, let us notice that we can go even further and create an image-of-images, i.e. an image with pixels being other images. This way we create a tensor

```cpp
TImageFor< SpecialPixelImage > theImageOfImage( 100, 200 );
```

Such flexibility is achieved thanks to generic definition of an image. In this example let us see that pixels are images, so we can set

```cpp
theImageOfImage.SetPixel( 2, 5, theSpecImage ); // now, a single pixel is and image...
```

Apart from monochrome, colour images are very common. These can be in the so called interlaced or non-interlaced formats. In the former each pixel contains three values of the same type for three basic RGB colours. In the latter there are three independent layers or images, each containing values of a single colour, i.e. red, green, or blue.

For the interlaced images the library contains a special template class for definition of n-valued pixels. This is the `MMultiPixelFor` class which can be accessed from the `HIL_MultiPixelFor.h` file. `MMultiPixelFor` with three 8-bit wide components is used as a template to define the class `ColorImage` for interlaced colour images.

In our example, first we create a pixel which will be used to initialize the subsequent colour images

```cpp
Color_3x8_Pixel theColor( 128, 200, 141 ); // R-G-B
```
Then the interleaved colour image can be created as follows

```cpp
// a 100x200 interlaced color image
ColorImage theInterlacedColorImage( 100, 200, theColor );
```

The non-interlaced images can be created with help of the `TMultiChannelImageFor` class, which is a derived version of the basic `TImageFor` class (in the file `MultiChannelImageFor.h`). The helper `typedefs` were then created to allow easy construction of the non-interlaced images, for example like the following

```cpp
// a 100x200 non-interlaced color image
NI_Color_3x8_Image theNonInterColorImage( 100, 200, theColor );
```

The resolution is again 100x200 pixels in both cases. The two images can be copied to/from with help of the overloaded `MixedCopy()` functions, for example

```cpp
MixedCopy( theInterlacedColorImage, theNonInterColorImage );
MixedCopy( theNonInterColorImage, theInterlacedColorImage );
```

The above are only very simple examples of manipulating pixels and images. In the project files there are many other examples of simple and advanced manipulations of images. Therefore the next steps are to analyse (read, apply, then debug) these and, above all, to write your own examples.

### 3.2.2 Input-Output of Image Objects

The image objects can be saved and loaded to/from disc files (or other streams). For this purpose the two simple functions `SaveImage` and `LoadImage` are provided (in the file `HIL_ImageFor.h`). These save/load images are in the binary format which is also known as the RAW format. Images in this representation can be manipulated in many graphic programs, such as Adobe Photoshop®, or in the attached test application for Windows, which is described in section 3.3. The main drawbacks of this format is large size of a stored image since no data compression is applied. The other problem is that size of an image is not saved with pixel data so it has to be known before an image is loaded from a stream.

Below we present an example of loading image contents from a disc file “myFile.raw”

```cpp
MonochromeImage theMonoImage( kImageCols, kImageRows ); // create an empty image
if( LoadImage( theMonoImage, "myFile.raw" ) == false ) // load contents from a RAW file
    return;    // Cannot open a file - wrong path or missing file...
```

In the above, size of the image, expressed by the number of its columns and rows in `kImageCols` and `kImageRows`, respectively, has to be known a priori since it is not saved in the file. Pixels will be loaded from the “myFile.raw” file to the `theMonoImage` image.

Saving an image is even simpler
SaveImage( theMonoImage, "myFile.raw" ); // save an image in a RAW format

If other image formats are necessary then an external conversion library has to be employed, as exemplified in the test application for Windows (section 3.3), or the one described in reference [14]. To save/load images in the ASCII format the other set of functions can be used: \texttt{Load\_ASCII\_Image} and \texttt{Save\_ASCII\_Image}. These allow visual inspection of the saved data, for instance.

### 3.2.3 Video Streams

To facilitate operations on video streams the library has been endowed with the \texttt{TVideoFor} template class. Its definition can be found in the \texttt{VideoFor.h} file. Once again, the main template parameter gives the type of pixel in each frame. Basically a video stream is composed of a series of frames which are \texttt{TImageFor} objects (section 6.4). Thanks to templates we are free to create a video with different types of pixels. However, since the most common are video streams with the scalar and colour pixels then for the two \texttt{MonochromeVideo} and the \texttt{ColorVideo} typedefs were created.

There are also two helper functions that facilitate creation of the \texttt{MonochromeVideo} or the \texttt{ColorVideo} of certain size. These are \texttt{OrphanMonochromeVideo} and \texttt{OrphanColorVideo}, respectively (both in the \texttt{VideoFor.h} file). Both functions orphan the returned video objects, so a caller is responsible for their deletion. In this respect application of the auto pointers (\texttt{MVAP} and \texttt{CVAP}, respectively) can avoid potential memory leaks. For instance, the following code creates a monochrome video object which consists of 100 frames, each of size 320x240 pixels

```cpp
int imCols = 320;
int imRows = 240;
int imImages = 100;
MVAP theVideo( OrphanMonochromeVideo( imCols, imRows, imImages ) );
```

Initially each pixel in each frame is set to 0. The returned object is kept in the auto-pointer (MVAP). Thanks to this we do not need to bother with deletion of this object. Now, some examples of setting and accessing pixels of the video object follow

```cpp
theVideo->SetPixel( 1, 1, 0, 100 );
MonochromeVideo::PixelType pixel = theVideo->GetPixel( 1, 1, 0 );
theVideo->SetPixel( 1, 1, 1, 101 );
pixel = theVideo->GetPixel( 1, 1, 1 );
theVideo->SetPixel( 1, 1, 2, 102 );
pixel = theVideo->GetPixel( 1, 1, 2 );
theVideo->SetPixel( 1, 1, 3, 103 );
pixel = theVideo->GetPixel( 1, 1, 3 );
```

More details on \texttt{TVideoFor} can be found in section 6.4.
3.2.4 Basic Operations on Images

Regarding the image operations the library allows two general types of access to them from an external software module:

1. An access trough the TImageOperation hierarchy (see the HIL_BaseOperations.h file). This has the advantage of separating an interface from an implementation at a cost of additional indirections and more complex code.

2. A direct access to image processing methods. This is a simpler form of calling image processing operations, justified in applications which do not foresee changes of implementations of the basic methods (a common case).

The basic suite of the arithmetical and logical pixel operations is defined in the TImageFor class, that is in the base class representing an image with pixels of a certain type. All of them can be compiled and run with no problems if these operations are allowed for the type of pixel of a given image. In other words, if we wish, for example, to add images, then we have to be able to add the pixels first, and so on. This feature holds for all built-in types such as unsigned char, int, long or double. However, we know that an image can store more complex pixels such as complex numbers, data containers or other images (or MyContainer in the example above). For some of them definition of some operations, like addition, multiplication, etc., can be problematic.

3.2.4.1 Linear Operations

Linear filtering belongs to the group of the most important operations on digital images. Therefore our first discussed operations are smoothing and edge detection obtained with the commonly known filters. For instance let us test behaviour of a filter given by the formula 4.12 (page 98) in the book [2]:

```cpp
const int kImageCols = 512, kImageRows = 512;
MonochromeImage theInImage( kImageCols, kImageRows ); // create an empty image
// load its contents from a RAW file
if( LoadImage( theInImage, "Kamil_mono_512x512.raw" ) == false )
    return;   // Cannot open the file - wrong path or missing file...

// Create two vectors with filter coefficients (see formula (4-12) in the book)
vector< double > horz_filter_mask, vert_filter_mask;
vert_filter_mask.push_back( -0.09375 );
vert_filter_mask.push_back( -0.3125 );
vert_filter_mask.push_back( -0.09375 );
horz_filter_mask.push_back( -1.0 );
horz_filter_mask.push_back( 0.0 );
horz_filter_mask.push_back( +1.0 );

// Since the convolution requires operations with fractionals
// in the simplest approach we need to convert images to the
// domain of real numbers
TRealImage theIn_Image_Real( theInImage ); // this will call a copy constructor

// and create an image for temporary results
TRealImage theTmp_Image_Real( kImageCols, kImageRows );
```

```cpp
```
Continuing the above snippet of a code let us now test behaviour of the not separated version of the
mask of the same filter

```c
// ... we continue with image objects already created by
// the code above

// Now do the same with the 2D 3x3 filter mask
TRealImage _2D_filter_mask( 3, 3, 0.0 );
_2D_filter_mask.SetPixel( 0, 0, 3.0 );
_2D_filter_mask.SetPixel( 0, 1, 10.0 );
_2D_filter_mask.SetPixel( 0, 2, 3.0 );
_2D_filter_mask.SetPixel( 2, 0, -3.0 );
_2D_filter_mask.SetPixel( 2, 1, -10.0 );
_2D_filter_mask.SetPixel( 2, 2, -3.0 );
_2D_filter_mask /= 32.0;

MixedCopy( theIn_Image_Real, theInImage );
Convolve( theIn_Image_Real, _2D_filter_mask, theTmp_Image_Real ); // Convolve 2D
ChangeImageRange( theTmp_Image_Real, 0.0, 255.0 );
MixedCopy( theOutImage, theTmp_Image_Real );
SaveImage( theOutImage, "Kamil_mono_512x512_edge2.raw" ); // save it in a RAW format
```

What is left is to compare results of the two convolutions from the two code fragments. In accordance
with Equation 4.12 these should be the same. The library is endowed with some generic functions to
perform comparison of two images. For instance the `Get_MSE` function will return a mean-square-
error after comparing pixel by pixel in its two input images (in `ImageErrorMeasurementLibrary.h`).
The results of these tests are shown in Figure 4.
As alluded to previously, the convolution can also be done by calling methods from the image operation layer (i.e. the handles), thus separating interface from its implementation (which can be done e.g. in hardware, invisibly to the caller). For example the above 2D convolution can be written as

```cpp
// At first create
ImageOperation_AutoPtr convolve_test {
  // out image, in image, the mask
  _2D_Convolve_AP( theTmp_Image_Real, theIn_Image_Real, _2D_filter_mask );
  // then execute a function object
  ( * convolve_test )();
}
```

The first type of accessing the methods is simpler since it does not require control of template parameters. Also, due to lack of one layer of indirection, it allows a simpler debugging. However, the second approach is recommended in larger systems in which changes in implementation are expected or mixed implementation platforms are envisaged (such as the already mentioned cooperation of the software and hardware modules).

### 3.2.4.2 Non-linear Operations

In the first example of non-linear operations we median filter an input monochrome image. Before starting please be sure to include the following header files

```cpp
#include "GraphicLibrary.h"
#include "HIL_MaskedImage.h"
#include "Morphology.h"
```
The following code loads a monochrome image and then performs median filtering in the 5x5 neighbourhoods of pixels. The result is saved back to a file.

```cpp
const int kImageCols = 512, kImageRows = 512;
MonochromeImage theInImage( kImageCols, kImageRows ); // create an empty image
// load its contents from a RAW file
if( LoadImage( theInImage, IMAGE_RELATIVE_PATH"Kamil_mono_512x512.raw" ) == false )
    return;  // Cannot open the file - wrong path or missing file...

MonochromeImage theOutImage( kImageCols, kImageRows ); // create an empty output image
Dimension mask_horz_size = 5, mask_vert_size = 5;  // filter in the 5x5 windows
MedianFilter( theInImage, mask_horz_size, mask_vert_size, theOutImage );

// save in a RAW format
SaveImage( theOutImage, "Kamil_mono_512x512_median.raw" );
```

It is also possible to median filter colour images.

In the second example the morphological gradient is computed. The new construction here is the `MaskedMonochromeImage` object which defines a structural element for the morphological operations. This is another type of image which, apart from a matrix of pixel values, contains a dual matrix of bit values which indicate whether a given pixel is in the state ‘on’ or ‘off’. With this feature we can define any region in an image by setting bits belonging to this object to ‘on’.

```cpp
const int kStructElem_Cols = 3;
const int kStructElem_Rows = 3;
const int kStructElem_InitVal = 0;
// Create the structural element
MaskedMonochromeImage theStructuralElement( kStructElem_Cols, kStructElem_Rows, kStructElem_InitVal, true );
MorphologyFor< MonochromeImage::PixelType > theMorphoObject;
MIAP outGradientImage( theMorphoObject.Gradient( theInImage, theStructuralElement ) );
SaveImage( * outGradientImage, "Kamil_mono_512x512_morpho_grad.raw" );  // save in a RAW
```

Results of the above two code fragments run for an exemplary test image “Kamil_mono_512x512.raw” are depicted in Figure 5.

![Image](image1.png)  ![Image](image2.png)

**Figure 5.** Results of the non-linear operations from the exemplary code fragments. Image median filtered (a), morphological gradient (b).
As in the case of linear filters it is possible to call indirectly the operators from the `TImageOperation` hierarchy. In our example these objects can be obtained by calling the helper functions `Median_Filter_AP` or `MorphoGradient_AP`, respectively.

### 3.3 Test Application for Windows

With the help of the HIL a simple Win32 application was created [7]. This is a standard multi document-view architecture of the MFC platform, depicted in Figure 6. Visible are the original image of a road scene with its affinely warped version (behind) and below the 5x5 median filtered (below).

![Figure 6. Win32 test application that joins the HIL for image processing and the MFC library with the multiple document-view architecture.](image)

The application can be compiled and debugged either in the Microsoft Visual C++ 6.0 or .NET 2005. Both projects are available from the web site of the book [7]. The application provides a very simple interface which shows how window actions are translated into image operations. Other operations can be easily added to the framework.
3.4 The SimpleFromTheBook Project

The SimpleFromTheBook project [7] contains the basic functions in exactly the same formatting of code as they were presented in the book [2]. These are selected functions for nonparametric transformations and area-based stereo matching (local and global – see Chapter 6 in reference [2]). These are augmented with a minimum set of classes for image representation and some basic operations. However, apart from the functions from the book there are other procedures that help them to run in a real console application and for real test images. The following table contains a short description of the main files in this project.

<table>
<thead>
<tr>
<th>Project file</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>main.cpp</td>
<td>Contains the test functions that call the functions presented in the book with concrete parameters.</td>
</tr>
<tr>
<td>source_from_the_book.cpp</td>
<td>Contains the functions in the same format as printed in the book.</td>
</tr>
<tr>
<td>ImageDeclarations.h</td>
<td>This file contains the basic definitions for the library.</td>
</tr>
<tr>
<td>GraphicLibrary.h</td>
<td>The file contains functions for different simple actions for image processing.</td>
</tr>
<tr>
<td>HIL_ImageFor.h</td>
<td>Contains classes for representation of images.</td>
</tr>
<tr>
<td>PixelAccessTraits.h</td>
<td>Auxiliary file for the HIL_ImageFor.h</td>
</tr>
</tbody>
</table>

Other classes, class hierarchies, as well as functions are contained in the more ample HIL project, which was described in the previous section. The posted version of the project is only for Visual C++ 6.0. However, if necessary, porting to .NET or other platforms should not be a problem. To facilitate porting the precompiled headers are not used.

This project is also a good example of a minimal configuration of the HIL which creates a foundation for basic operations on images.
Figure 7. Results of the run of the SimpleFromTheBook project. The left image of a stereo-pair “Artkor” (from Bonn University) (a), its Census version (b).

Figure 8. Further results of the SimpleFromTheBook project. Disparity map from the point-oriented area-based matching (a), disparity-oriented version (b).
4 Connection to Existing Platforms

4.1 Using HIL in Other Projects

The library follows the handle/body pattern (see section 13.3.2 in the book). Thus usually there is a separation of the interface of an operation (a handle) from its software implementation (body). As a consequence in the software projects it is possible to call either the interface and follow the semantics of HIL or directly the bodies to simplify the indirection, however at a cost of closer coupling with specific solutions.

Figure 9 presents a diagram describing ways of integration of provided library with the new and existing projects. There are basically three paths of development:

1. User's extensions to the library.
2. Integration with existing projects.
   - direct integration;
   - integration with the help of the wrapper interface [4].
3. Direct integration with the new projects.

![Figure 9. Integration strategies with external projects.](image-url)
Due to the object-oriented architecture of the software interface, the provided library can easily be extended by users. In practice this is achieved either by derivation from the existing classes or with the help of the object wrapper technique \cite{4,19}.

Binding the library to the existing projects is more complicated due to existing dependencies in those projects. Quite often different image formats are used and additional techniques have to be provided to connect such projects. In this case the wrapper and adapter patterns can also be recommended \cite{4}.

### 4.2 Cooperation with the Open CV Library

There are other image processing and computer vision software platforms. Possibly, they have been used for implementation of user specific applications. One of very the most common is Intel's Open Source Computer Vision Library \cite{10}. Since this library is very frequently used, an interface between the two platforms is created.

The suite of six \textit{MixedCopy()} functions was implemented for the conversions between image formats of HIL and the OpenCV Library. An additional suite of \textit{MixedCopy()} functions was added to copy between interlaced and non-interlaced colour images.

Adding new conversions by a HIL user requires only an overloaded version of the \textit{MixedCopy()}. It doesn't impose any additions to the \textit{HIL FormatConvert_OperationFor} interface however.

Basic version of the library has its parameters set to disconnect from the OpenCV. However, to activate this liaison, one needs to change some parameters of the project, as follows:

1. Set the flag CONNECT\_2\_OPEN\_CV to 1 in \textit{IntelLib\_ImageConverters.h}

2. Add the libraries to the build. In the “Library\/Open CV” folder of the project right click on the library files and from the pop-up menu choose “Settings…”. In the opened “Project Settings” dialog select the “General” tab and unmark the “Exclude file from build” (or in the Project Settings (Alt+F7), in the tab “Link”, enter new libraries \textit{highgui.lib cv.lib}).
5 Fixed-Point Representation of Pixels

Selection of the right format of data representation is important especially when processing massive amounts of data. This is the case of image processing in which size of data structure chosen for pixels directly determines size of images, and indirectly affects the processing time. On the other hand, there are operations which require representation of both integer and fractional parts of results. In such situations the float or double types in C/C++ are instant choice. However, these usually consume 4 or 8 bytes per value. Therefore a tighter representation should be sought. This can be achieved with the fixed data format in which a computer word is divided into predefined two parts of bits, one for the integer and the other for the fractional parts, respectively. Unfortunately, C/C++ do not provide methods for manipulation of such data. Therefore the FixedFor<> template class was added to the library which allows utilization in C++ of the fixed data format with defined precision.

5.1 Basic information

Scientific computations usually require usage of real numbers. However, computer processing is characteristic of fixed length for number representation. Therefore there is no one-to-one mapping between the set of real numbers and the set of computer words, which is always of finite cardinality. The question arises of what is the best (i.e. with the possibly the lowest loss of precision) representation of real numbers. Thus for the fractional arithmetic in computers the two main formats have been developed [13]:

- The floating-point representation (e.g. defined in the IEEE 754-1985 standard [9])
- The fixed-point representation.

The former can comprise much broader dynamics of real numbers, however at a cost of loss of precision. The latter is an alternative for pixel representation since dynamics of operations on them is usually constrained. At the same time size of these representations can be even half of the single floating data. The $N$-bits long fixed-point representation of data can be visualized as follows:

![Fixed-point representation of numbers.](image)

**Figure 10. Fixed-point representation of numbers.**
The fixed-point format implicitly assumes a fixed position of the decimal point between the integer and fractional parts (thus the name of this representation): it is located between the bits $b_0$ and $b_1$ in Figure 10 and its position cannot change. Thus

$$ N = p + q + 1 $$

(1)

The value of a number represented by a given permutation of bits in Figure 10 depends entirely on their interpretation. Apart from the integer and fractional parts information on sign should be also conveyed. For these purposes there are many possible schemes of which the most common are:

- One’s compliment (U1)
- Two’s compliment (U2)
- Sign-magnitude (SM)

### 5.2 Software Model for Fixed-Point Data Representation

The definition of the `FixedFor<>` template class is presented in Algorithm 1. Its template parameters define type of data used for representation of the whole number and precision, i.e. number of bits for the fractional part.

```cpp
template< typename DATA_TYPE, int PRECISION >
class FixedFor
{
  private:
    // Class inherent variables
    // The best number representation for the FPGA implementation is the sign-magnitude format, albeit C++ integer arithmetic assumes U2 representation.
    DATA_TYPE fValue;
  public:
    enum { kPrecision = PRECISION };
    enum { kSignMask = 1 << 8 * sizeof( DATA_TYPE ) - 1 };
    typedef FixedFor< DATA_TYPE, PRECISION > ThisFixedFor_Type;
  public:
    // ===================================================
    // class constructors
    FixedFor( void );
    FixedFor( int x );
    FixedFor( long x );
    FixedFor( char x );
    FixedFor( double x );
    template< class D, int P >
```
FixedFor( const FixedFor< D, P > & f ); // mixed copy constructor

// ==============================================================
template< class D, int P >
ThisFixedFor_Type & operator = ( const FixedFor< D, P > & f );
// mixed assignement

// class destructor (not virtual since we don't wish
// to derive from this class)
- FixedFor() {}

// ==============================================================
operator double() const;
operator int() const;
operator long() const;
operator char() const;

// ==============================================================
// Helpers:
bool IsNegative( void ) const;
bool IsPositive( void ) const;

void ChangeSign( void ) { fValue ^= kSignMask; }
// Simple XOR will do this nicely
void MakeNegative( void ) { fValue |= kSignMask; }
// Turn the sign bit ON
void MakePositive( void ) { fValue &= ~kSignMask; }
// Turn the sign bit OFF

DATA_TYPE GetAbs( void ) const { return fValue & ~kSignMask; }
// Return the absolute value
void MakeAbs( void ) { MakePositive(); }
// Get rid of the sign, i.e. make it positive

// This always returns positive value
DATA_TYPE GetMagnitudeOfIntegerPart( void ) const;
DATA_TYPE GetFractionalPart( void ) const;
// This always returns positive value
static ThisFixedFor_Type GetMinimumMagnitude( void );
static ThisFixedFor_Type GetMaximumMagnitude( void );

// ==============================================================
// Basic library of operations:
ThisFixedFor_Type operator + ( ThisFixedFor_Type f ) const;
ThisFixedFor_Type operator - ( ThisFixedFor_Type f ) const;
ThisFixedFor_Type operator * ( ThisFixedFor_Type f ) const;
ThisFixedFor_Type operator / ( ThisFixedFor_Type f ) const;
ThisFixedFor_Type operator += ( ThisFixedFor_Type f );
ThisFixedFor_Type operator -= ( ThisFixedFor_Type f );
ThisFixedFor_Type operator *= ( ThisFixedFor_Type f );
ThisFixedFor_Type operator /= ( ThisFixedFor_Type f );

// ==============================================================
ThisFixedFor_Type operator ++ ( int ); // postfix
ThisFixedFor_Type & operator ++ (); // prefix
ThisFixedFor_Type operator -- ( int ); // postfix
ThisFixedFor_Type & operator -- (); // prefix

// ==============================================================
ThisFixedFor_Type operator <<= ( int shift );
ThisFixedFor_Type operator >>= ( int shift );
ThisFixedFor_Type operator <<= ( int shift );
ThisFixedFor_Type operator >>= ( int shift );
Algorithm 1. The template FixedFor<> base class for fixed-point representation of numbers.

Some features of the FixedFor<> class are controlled during compilation time (i.e. by preprocessor directives in the file FixedNumber.h). These are:

- The saturation arithmetic
- The exception handling
- The static flags monitoring some arithmetic conditions (such as divide by zero, overflow, etc.).

5.3 Mathematical Routines for Fixed-Point Representation

Table 4 presents mathematical functions that were implemented for the fixed-point data (available in the MathLibraryForFixed.h).

Table 4. Mathematical functions defined and implemented for fixed-point data

<table>
<thead>
<tr>
<th>Function</th>
<th>Approximating Taylor series</th>
<th>Implementation details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sine</td>
<td>( \sin(x) = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{(2k+1)!} )</td>
<td>Implemented as a template function: Sin_ForFixed&lt;&gt; based on the Taylor series expansion (only four initial terms are used) [6]. Function can accept all input arguments (in radians).</td>
</tr>
<tr>
<td>Cosine</td>
<td>( \cos(x) = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k}}{(2k)!} )</td>
<td>Implemented as a template function: Cos_ForFixed&lt;&gt; based on the Taylor series expansion (only four initial terms are used). Function can accept all input arguments (in radians).</td>
</tr>
</tbody>
</table>
### Tangent

\[ \tan(x) = \sum_{k=1}^{\infty} \frac{2^{2k}(2^{2k} - 1)x^{2k-1}}{(2k)!}B_{2k}, \quad |x| < \frac{\pi}{2} \]

where \( B_n \) denotes the Bernoulli numbers that are defined in terms of the Bernoulli polynomials [6], given as follows:

\[ B_n = \frac{(2n)!}{n^2 n!} \left( 1 - \frac{1}{2^n} + \frac{1}{3^n} + \frac{1}{4^n} + \ldots \right). \]

Implemented as a template function: \( \text{Tan\_ForFixed<>()} \)

If the module of its argument (in radians) is smaller than \( \pi/2 \) a value is computed based on the Taylor series expansion of the tangent function (only four initial terms are used). Otherwise the formula \( \tan(x) = \sin(x)/\cos(x) \) is used for \( \cos(x) \neq 0 \). If \( \cos(x) = 0 \) then an exception is generated.

### Arcus Tangent

\[ \text{arctan}(x) = \begin{cases} \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{2k+1} & \text{for } |x| \leq 1 \\ \frac{\pi}{2} - \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)x^{2k+1}} & \text{for } |x| > 1 \end{cases} \]

Implemented as a template function: \( \text{Atan\_ForFixed<>()} \)

A value of this function is computed from one of the Taylor series expansions for the arcus tangent function, which is chosen based on a value of its argument (only four initial terms of each series are used).

### Exponent

\[ \exp(x) = \sum_{k=0}^{\infty} \frac{x^k}{k!} \]

Implemented as a template function: \( \text{Exp\_ForFixed<>()} \)

based on the Taylor series expansion (only five initial terms are used). Function can accept all input arguments.

### Natural Logarithm

\[ \ln(x) = 2 \sum_{k=1}^{\infty} \frac{1}{2k-1} \left( \frac{x-1}{x+1} \right)^{2k-1}, \quad x > 0 \]

Implemented as a template function: \( \text{Ln\_ForFixed<>()} \)

based on the Taylor series expansion (only four initial terms are used). Function can accept all input arguments that are greater than 0 (otherwise an exception is generated).
All functions in Table 4 are implemented as template functions that accept different types of the numbers obtained from the `FixedFor<>` class. Their implementation is based on Taylor series expansions of pertinent functions [6]. With this implementation the following factors should be taken into consideration:

1. Some Taylor series assume only a limited range on its input parameter. It is checked in the current implementation and if the input argument lies outside a scope of operation then a software exception is generated. This should be considered when calling these functions and the mathematical function should be embraced by the `try-catch` block, otherwise a whole application can generate an unhandled exception and the process will be released by an operating system.

2. The Taylor series are indefinite in their definitions. However, in the implementations only four or five expansion terms are taken for computations. This unavoidably causes some errors in the output value. The other type of errors is caused by finite precision of the fixed-point representation of the input arguments.

3. Computation of power series involves many multiplications and additions which can exhibit restrictive run-time performance for some applications. In such cases other implementations could be considered, e.g. the look-up tables with precomputed values or CORDIC, etc. [16].

<table>
<thead>
<tr>
<th></th>
<th>NA</th>
<th>Implemented as a template function:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power</td>
<td>NA</td>
<td><code>Power_ForFixed&lt;&gt;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Its value is computed from the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>following formula:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ x^a = \exp \left( a \ln(x) \right) \text{, for } x &gt; 0 ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>while functions ( \exp() ) and ( \ln() ) are</td>
</tr>
<tr>
<td></td>
<td></td>
<td>computed from the Taylor series</td>
</tr>
<tr>
<td></td>
<td></td>
<td>expansions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Function can accept all input</td>
</tr>
<tr>
<td></td>
<td></td>
<td>arguments that are greater than 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(otherwise and exception is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>generated).</td>
</tr>
</tbody>
</table>

| Square Root | NA | Implemented as a template function: |
|            |    | `Sqrt_ForFixed<>`                  |
|            |    | Its action is delegated to the     |
|            |    | `Power_ForFixed<>`                 |
|            |    | function with exponent fixed to 0.5. |
6 More on Image Models

In this chapter we discuss more complex structures and specializations of image objects. These are possible mostly due to generic definition of the base `TImageFor<>` (discussed in Chapter 3 of the book [2]).

The fixed-point class allows definition of images with this type of pixel which can save occupied memory, compared with images with floating formats of pixels (i.e. pixels being `float` or `double`). At the same time fixed point format allows computations with required precision of the majority of image processing routines. This also has an advantage in hardware implementation, greatly reducing hardware resources devoted to the arithmetic operations.

Control of memory allocation for image object can be crucial in applications dealing with large amount of data, as well as in systems cooperating with hardware accelerators. In the latter the fast DMA transfers can avoid bus stalls.

Definition of specialized objects for video streams, with separate control on their allocation, helps in all operations on video. These are also discussed in this chapter.

6.1 Images with Fixed-Point Pixels

After defining new types of pixels, as expected, new types of images with fixed-point representation for pixels have been defined. These can successively replace images with pixels of the `TRealImage` specialization. The file `HIL_ImageForFixed.h` contains some useful typedefs of images with pixels of different size and precision fixed-point formats. For instance we have

```c++
typedef TImageFor< FIXED_8_8 >  ImageForFixed_8_8;
typedef TImageFor< FIXED_16_16 > ImageForFixed_16_16;
```

or for colour images

```c++
typedef MMultiPixelFor< FIXED_8_8 >  Color_3xFi8_8_Pixel;
typedef TMultiChannelImageFor< FIXED_8_8 > NI_Color_3xFi8_8_Image; //non-interlaced image
```

Let us observe that with two bytes, one for the integer and one for the fractional part, we are able to process values from 0.0 up to 255.99609375 with a precision of 0.00390625. This is sufficient for many simple filters, etc. At the same time memory allocation is reduced four times. This means about 1.8 MBytes reduction in memory consumption for a single frame of 640x480 pixels.

The following code fragment presents application of the images with pixels in fixed-point format to some arithmetic operations:

```c++
const int kCols = 25;
const int kRows = 12;
ImageForFixed_16_16 i1( kCols, kRows ); // create three images
ImageForFixed_16_16 i2( kCols, kRows ); // with FIXED pixels
ImageForFixed_16_16 i3( kCols, kRows );
i1.SetPixel_Modulo( 1000, 1000, 0 ); // setting modulo frees from checking
```
Let us observe that the final operation can generate an exception (discussed in section 7.1) which should be caught as soon as possible so as not to break an application.

6.2 Image Specializations for Boolean Pixels – Masked Images

For some tasks of image processing it is useful to have different specializations of the basic objects for image representation. For instance, the set of morphological operators relies on structural elements which can be efficiently implemented by the specialized classes of the TImageFor<> base representation. Therefore the new classes have been added to the platform. These are: the template specialization TImageFor<bool> and the derived TMaskedImageFor<> template class.

The purpose of the TImageFor<bool> is efficient implementation of the binary image objects, i.e. images which pixels take on only two values, 0 or 1 (true or false). Taking into consideration that each computer word on modern PCs has 32 or even 64 bits we can pack up to 64 pixel values in each computer word. This significantly reduces memory required to store image representation. The class TImageFor<bool> is also an important member of the second class, namely the TMaskedImageFor<> which was already mentioned in section 3.2.4.2 in the context of definition of the structural element for morphological operators.

The TMaskedImageFor<> class was designed to represent some irregular regions of an image. There are many ways to achieve these tasks – e.g. we have already developed the TProxyImageFor<> class to define a rectangular region inside an image. However, in general for each pixel of a ‘mother’ image we have to assign a membership function. This can be done by a binary image or a set of binary images for even more general cases. The TMaskedImageFor<> class is derived from the base TImageFor<> class. However, it also has one protected object member – the instance of the
The `TImageFor<bool>` class. This member defines a membership function for the pixels (which can be of any type).

The `TImageFor<bool>` is defined in the `HIL_BooleanImage.h` file.

The `TMaskedImageFor<>` is defined in the `HIL_MaskedImage.h`.

### 6.3 Memory Allocators for Objects

In some applications it is important to control the process of memory allocation for objects. The precise control of an object’s placement and its lifetime belong to the key problems in compound processing systems.

Therefore it is not surprising that in the Standard Template Library (STL) special techniques of memory allocation have been developed. These consist of the addition of a special template parameter of the `allocator` which is added to the definitions of all data containers of the STL [19].

In the HIL system it is also very important to have control of exact memory location and dynamics of allocation of a data buffer (matrix of pixels). This could be achieved with the previous version of the library by means of class inheritance. However, this requires user’s definitions for each specific memory allocation requirement. Therefore the STL-like techniques have been incorporated into HIL. The method relies on the policy technique.

```cpp
template <typename U>
class InPlaceAllocatorFor
{

    // This class implements a special allocator which
    // allocates a container into a given buffer of memory.
    // This buffer has to be created beforehand and must
    // have a capacity suitable for a given container.

    // Allocation of "num" elements U without initialization
    pointer allocate( size_type num, const void * = 0 )
    {
        return fStartDataPtr;  // just return the pointer to the buffer
    }

    // Initialization of the elements in the already allocated storage,
    // given by a pointer "p", with a value "value"
    void construct( pointer p, const U & value )
    {
        new ( (void*)p ) U( value );  // ... use the placement new
    }

    // Destroy elements of initialized storage given by "p"
    void destroy( pointer p )
    {
        p->~U();  // call destructor at address "p"
    }
};
```
void deallocate(pointer p, size_type num) {
    // Do nothing here - it is up to an external owner of the buffer to take care of it
    // -----------------------------------------------
    // ...
};

Algorithm 2. Interface of the InPlaceAllocatorFor class for the in-place memory allocations.

The InPlaceAllocatorFor in Algorithm 2 follows the interface of the STL memory allocators [19]. However, contrary to the default allocators the main role of this allocator is to provide a constant pointer to the already reserved block of memory. Such a situation is frequent when a special memory is dedicated to image processing, e.g. special memory block for DMA operations with a hardware board. This technique can also be used to create a kind of proxy object to the already created image object. Instead of creating a new buffer and copying data, operations can be performed on external data buffer in terms of a TimageFor<> object.

Another frequent situation in cooperation of software and hardware platforms is creation of a series of images, such as a video stream, in a solid continuous block of memory. For this purpose the SaveBufferAllocator template class has been created. Its interface is outlined in Algorithm 3.
Algorithm 3. Interface of the SaveBufferAllocator class for the allocations of continuous blocks of memory for compact video objects.

The SaveBufferAllocator class has three main tasks:

1. Creation of the continuous block of memory for requested number of images of the same size. For creation of this buffer an external memory allocator (such as InPlaceAllocatorFor) can be provided. Otherwise, a default allocator is used which relies on standard new and delete operations.

2. The buffer is then maintained by this class and is destroyed in the destructor of SaveBufferAllocator.

3. The GetNext_InPlaceAllocator and Get_InPlaceAllocatorAt methods provide an instance of the allocator when allocating a single image (a memory slice) in the memory block maintained by the SaveBufferAllocator class.

The SaveBufferAllocator class can be useful when creating continuous blocks of memory devoted to storage of the video streams which are transferred to and from external memory, such as a memory on a hardware acceleration block. This strategy allows the fastest DMA access to the whole video stream at a time, compared for instance to a case of a linked list of memory locations for consecutive video frames.

The main constraint of the above approach is that once allocated memory cannot be extended. Thus, this technique is applicable for video frames for which it is possible to predict size a priori.
6.4 Representation of Video Objects

Based on the basic definition of the image objects it is possible to define an image-of-image which can serve to represent series of images, i.e. a video stream. However, to facilitate operations on video streams of dynamic length or different types of images a separate TVideoFor template class was added to the library. It is presented in Algorithm 4.

```cpp
// This class implements an interface for video streams.
// A strict assumption is that the frames contain the
// same type of pixels. Dimensions of the frames can
// be different, though.

// For a default memory allocator the images are stored
// rather as a list, i.e. they
// do not constitute a compact memory region.
// If other schemes are required then a proper
// memory allocator should be passed.

// This function attaches a new and orphaned frame to the end
// of the video object. The frame is owned, and in consequence
// deleted, by this video object.

// INPUT:
//   newImage - pointer to the orphaned frame

// OUTPUT:
//   true if operation successful
//   false otherwise

// REMARKS:
//
// virtual bool AttachOrphanedFrame( FrameImagePtr newImage );
```

```cpp
template < typename PixType, typename PAT = PixelAccess_Trait< PixType >, typename AL = allocator< PixType > >
class TVideoFor
{
// ...

public:

TVideoFor( void ); // class default constructor
```

This class implements an interface for video streams. A strict assumption is that the frames contain the same type of pixels. Dimensions of the frames can be different, though.

For a default memory allocator the images are stored rather as a list, i.e. they do not constitute a compact memory region. If other schemes are required then a proper memory allocator should be passed.

This function attaches a new and orphaned frame to the end of the video object. The frame is owned, and in consequence deleted, by this video object.

INPUT:
   newImage - pointer to the orphaned frame

OUTPUT:
   true if operation successful
   false otherwise

REMARKS:
   
   virtual bool AttachOrphanedFrame( FrameImagePtr newImage );
```
virtual bool InsertOrphanedFrameAt(int index, FrameImagePtr newImage);

rem This function removes and deletes frames from the video stream in positions from "from" up to "to".
rem
// INPUT:
//   from - index to the first frame to be removed
//   to - index to the last frame to be removed
// 
// OUTPUT:
//   true if operation successful
//   false otherwise
// 
// REMARKS:
//   To remove a single frame "from" and "to"
//   should be equal and point at a given frame.
// 
virtual bool RemoveFrames(int from, int to);

rem This function returns a pointer to a frame at position or 0 if the index is out of range.
rem
// INPUT:
//   index - 0 based index of the frame to be returned
// 
// OUTPUT:
//   ptr to the frame at index or 0 if index is out of range
// 
// REMARKS:
// 
virtual FrameImagePtr GetFrameAt(int index);

rem This function returns a pixel given its coordinates.
rem
// INPUT:
//   cols - column position of a pixel
//   rows - row position of a pixel
//   frame - 0 based index of a frame in the video stream from which a pixel is accessed
// 
// OUTPUT:
//   pixel through defined access type (such as a value, or reference; defined by PAT)
// 
// REMARKS:
//   This function is not save against wrong indices, so they should be checked before calling this function!
// 
FramePixelAccessType GetPixel(int cols, int rows, int frame);

rem This function sets a value to a pixel.
rem
// INPUT:
//   cols - column position of a pixel
//   rows - row position of a pixel
Video streams are composed of strictly timed image frames. Figure 11 depicts class hierarchy for this video data structure. Data organization of a video object is presented in Figure 12.

**Algorithm 4. Interface of the TVideoFor class.**

The template class `TVideoFor<>` contains a vector of pointers to the frame images. Each frame is an object of the `TimageFor<>` class.
Figure 12. Illustration of the video data structure. Pointers to frames are stored in the vector structure.

However, an object of the `TVideoFor<>` class stores only data location with pointers to the video frames which are located elsewhere in the memory space. A way of allocating these areas of memory can be controlled by an external allocator object. Thanks to this technique we are able to locate a continuous block of memory for the whole video object or allocate memory in some areas other than those available through standard memory allocation.

Algorithm 5 presents a simple `VideoTest()` function which shows an example of using the new memory allocation and video classes.

```c
void VideoTest ( void )
{
    typedef unsigned char MonoPixel;
    typedef InPlaceAllocatorFor< MonoPixel > InPlaceAllocatorForMonochrome;
    typedef TVideoFor< MonoPixel, PixelAccess_Trait< MonoPixel >, InPlaceAllocatorForMonochrome > MonochromeVideoInPlace;

    int imCols = 128;
    int imRows = 256;
    int imImages = 17;

    unsigned long single_image_size = imCols * imRows * sizeof( unsigned char );
    typedef SaveBufferAllocator< MonoPixel > MonochromeVideo_SafeBufferAllocator;

    // Create a solid buffer of memory for the whole video stream
    // - it also works like auto_ptr
    MonochromeVideo_SafeBufferAllocator theMonochromeVideo_SafeBufferAllocator( single_image_size, imImages );

    // Create a video object
    MonochromeVideoInPlace monoVid;

    // Attach some frames to it - all these frames will be placed
    // in consecutive memory locations for future DMA transfer
    monoVid.AttachOrphanedFrame( new MonochromeVideoInPlace::FrameImage( imCols, imRows, 0, theMonochromeVideo_SafeBufferAllocator.GetNext_InPlaceAllocator() ) );
}
Algorithm 5. Exemplary function with a monochrome video stream created in a solid block of memory (more examples in the HIL project).

In VideoTest() the SaveBufferAllocator is used to create a continuous block of memory for the whole video object. Then subsequent frames are added by means of the AttachOrphanedFrame member of the video object. In each call the GetNext_InPlaceAllocator returns an in-place memory allocator for the next frame of the video.
7 More on Software Architecture of the Library

This chapter deals with advanced concepts of the library. We start with exception handling. Then advanced mechanisms of image processing are discussed.

7.1 Exception Handling Hierarchy

A unified way of handling unexpected or erroneous situations is required in any compound system to avoid software crashes and to ensure reliability of that system. In C++ based systems there are means of catching such situations to allow the system to decide in a controlled way the next action. In the HIL such functionality was achieved with the hierarchy of exception handling classes, derived from the std::exception:

```
enum HIL_Standard_ErrorHandlerError { 
    Unknown = 0, 
    DivideByZero, 
    WrongInputParam, 
    WrongOutputParam, 
    ImageDimensionsMismatch, 
    WrongPixelType, 
    ResultInfinity 
};
```

The class hierarchy derives from the Standard Template Library. By this token it is possible to catch all "standard" exceptions in a single "catch" statement. The T_HIL_Exception is an actual base for the HIL specific exceptions. All future specialized HIL exception handlers (e.g. for driver exceptions) should be
derived just from this base class. The specialized \textit{T\_Standard\_HIL\_Exception} defines the most common, or standard, exceptions that are envisaged for this library at a moment.

### 7.2 Advanced Image Operations

#### 7.2.1 Call Mechanisms for Image Operations

Algorithm 3-4 in reference [2] presents the template class hierarchy of the HIL software interface. \textit{TImageOperation} is the main base class for all operations. It defines the common function operator which is extended in derived classes. There are four major derived classes that define each type of image operation and operation composition. The general template solution mechanism, chosen in the derived classes, allows for any type of arguments of an operation. \textit{TImageOperation} is a root (base) pure virtual class for all image operations. All other operations are derived from this class. The general ideas for the image operations can be summarized as follows:

1. The base class \textit{TImageOperation} is a pure virtual class, so all operators should be objects of the derived classes.

2. The base \textit{TImageOperation} accepts and stores references to the two external objects:
   - Thread security object that in run-time controls access to the processing resources
   - Operation callback object which is called upon completion of an operation.

3. Each operation (i.e. the functional operator) should be decorated with operation begin and operation end sequences. To help this action an automatic variable of the inner \textit{MImageOperationRetinue} class should be defined at the beginning of each \texttt{operator()} in a derived class. Then when this variable gets out of scope of the \texttt{operator()} its destructor is automatically invoked which, in turn, calls \texttt{operator\_end()} member.

4. Wherever it is possible all parameters are treated as images. So, an image is a more ample notion than a classical "visible" image. For example an image can store in its pixels a value of a just found maximum pixel in another image, as well as \texttt{x} and \texttt{y} coordinates of that pixel, as its next pixels. This is analogous to a matrix-processing context where each value is treated as a matrix.

It is interesting to notice that the base \textit{TImageOperation} is a pure virtual class (i.e. it can serve only to be derived from, no objects of this class are allowed) but it is not a template class, whereas its derived classes are – see Algorithm 3-4 in reference [2]. The hierarchy is described in section 3.7.1.3 of reference [2].

Figure 14 presents a flow chart with steps of execution of each image operation. There are three stages of execution:
1. An operation preamble which consists of the acquisition of processing resources
2. The main image operation
3. The operation finishing sequence which consists of resource release and callback (notification) mechanism.

**Figure 14.** Flow chart of each image operation. There are three stages of execution: operation preamble which consists of the acquisition of processing resources, the main image operation, and the operation finishing sequence which consists of resource release and callback (notification) mechanism.

Figure 15 depicts an activity diagram of image operations. Operations are provided with many input and output images which can be of different size and have different types of pixels. An image can constitute an input for some operations and an output for others. Each operation has an associated number of input images and a single output image, however.

The operations can be further grouped in certain compositions. Order of execution is determined by position of that operation in enclosing composition object. The composition objects can be recursively composed in clusters, and so on.

Operations can launch callback notification upon completion. Each operation can be supplied with resource access object. For this reason the automatic variable of class `TImageOperation::MImageOperationRetinue` should be defined in each `operator()`.

An important feature of the presented mechanism is that the complex image operations are constructed recursively by building simple operations and adding them to composite objects. The latter can also be grouped in bigger composites since composites are image operations by themselves – in terms of the C++ inheritance because they are derived from the `TImageOperation` base class.
Figure 15. Activity diagram of image operations. Operations accept many images and are grouped in certain compositions. Each operation has an associated number of input images and a single output image. An image can constitute an input for some operations and output for others. Order of execution is determined by position of that operation in enclosing composition object. Some operations can launch callback notification upon completion. Each operation can be supplied with resource access object.

7.2.1.1 Resource Access and Release Sequence for an Operation

The resource acquisition and access can pose a serious problem in a multithreaded environment. Each thread should check if a resource it tries to access is available and if not it should be blocked waiting to be notified.

The HIL is prepared for such situations: each operation can be supplied at its construction time with an external object, of a class `TThreadSecurity` (Algorithm 6), or a class derived, so that it controls an access to the computer resources (such as the hardware board, etc.). Each operation has been endowed with a special calling sequence – see Figure 14. Before the main computation part commences, the resource acquisition preamble is called. Actually this is done by action delegation to the resource control object (supplied in a constructor). If such a resource is not available at a time, the whole thread can be blocked. However, the actual action is entirely up to the external object. After the computations the resource release procedure is invoked. The C++ implementation of this calling sequence can be simplified by utilizing semantics of automatic variables.
Algorithm 6. The TThreadSecurity base class. The responsibility of derived classes is to control access to different computer resources.
The main two members of the `TThreadSecurity` class, presented in Algorithm 6, are `Enter_CriticalSection()` and `Exit_CriticalSection()`. In derived classes they should implement an access to particular resources.

### 7.2.1.2 Operation Completion Callbacks

Algorithm 7 presents the base class for image operation callbacks. Objects of the derived classes can be registered to the image operations. They are called on operation completion. This mechanism allows efficient notification of finished actions.

```cpp
istream(49) ostream(58)

///////////////////////////////////////////////////////////
// This is a root (base) class for the callback objects
// that can be (optionally) supplied to the image operator
// objects when it is created. The callback is called when
// the related image operation is finished.
// Derive your own class for more advanced callbacks.
///////////////////////////////////////////////////////////
class TOperationCompletionCallback
{
  public:

    virtual void * operator()( void )
    {
      return 0;
    }
};
```

**Algorithm 7.** Definition of the base class for image callbacks. Objects of the derived classes can be registered to each of the image operations and then invoked when such an operation is completed. This mechanism allows efficient notification of finished actions.

The callbacks are optional. It is also possible to share the same callback among many image operations. However, it must be ensured that a callback object is alive at least as long as all the image operation objects it is registered to.
7.2.2 The Class for Unary Image Operations

Algorithm 8 presents definition of the \texttt{TUnaryImageOperationFor<>} template class that constitutes a root for all image operations that accept only one input image.

```
// This is a base pure virtual class for all
// UNARY IMAGE OPERATIONS, i.e. those that accept
// a single image and return a single image
// of any specific type, however.
// This class is parameterized by two template
// types: for the return image and for the
// input image, respectively.
// It is derived from the base \texttt{TImageOperation}.

template< typename RetIm_Type, typename InIm_Type1 >
class TUnaryImageOperationFor : public TImageOperation {
public:
  enum { eNumOfParams = 1 };
  typedef RetIm_Type RetType;
  typedef InIm_Type1 InType_1;
protected:
  RetType & fRetImageRef; // the reference to the output image
  const InType_1 const & fInImage1_Ref; // the reference to the input image
public:
  // Base class constructor
  TUnaryImageOperationFor( RetType & retImage, const InType_1 const & inImage1,
                          TThreadSecurity & resourceAccessPolicy = kgThreadSecurity,
                          TOperationCompletionCallback & opCompCallback= kgOperationCompletionCallback );
};
```

```
The function operator which - in a derived class - 
defines an image operation. 

/// INPUT:
/// none 
/// OUTPUT:
/// user defined (in a derived class) void pointer 
/// REMARKS:
/// The input and output images should be already 
/// supplied to the class constructor. 
virtual void * operator() ( void ) = 0;

Algorithm 8. The pure virtual base class TUnaryImageOperationFor<> for hierarchy of all unary image operations.

The most distinctive feature of this class compared to its base is that it is a template with two parameters:

- RetIm_Type
- InIm_Type1

These template parameters determine types of a return image and a single input image, respectively. The virtue of this approach is that this technique can be used for all types.

7.2.3 The Classes for Multi Image Operations

Algorithm 9 presents definition of the TUnaryImageOperationFor<> template class that constitutes a root for all image operations that accept binary input. The same concept is used also for the TImageTemplateOperationFor<> for template-image operations.
const InType_2 & fInImage2; // the reference to the second input image
public:

///////////////////////////////////////////////////////////
// Base class constructor
///////////////////////////////////////////////////////////

// INPUT:
// retImage - reference to the output image
// of type RetIm_Type (specified by the
// first template parameter)
// inImage1 - constant reference to the first constant
// input image of type InIm_Type1 (specified
// by the second template parameter)
// inImage2 - constant reference to the second constant
// input image of type InIm_Type2 (specified
// by the third template parameter)
// resourceAccessPolicy - optional reference to
// the thread security object (derivative
// of the TThreadSecurity class); by default
// the static kgThreadSecurity object is supplied
// which does nothing
// opCompCallback - optional reference to the callback
// object which is called upon completion of operation;
// by default the static kgOperationCompletionCallback
// object is supplied which does nothing

///////////////////////////////////////////////////////////
// The function operator which - in a derived class -
// defines an image operation.
///////////////////////////////////////////////////////////

virtual void *
operator() ( void ) = 0;
};

Algorithm 9. The pure virtual base class TBinaryImageOperationFor<> for hierarchy of all binary image operations. The same construction is used for the TImageTemplateOperationFor<> for template-image operations.

The most distinctive feature of the class compared to its base is that it is a template with three parameters:

- RetIm_Type
- InIm_Type1
- InIm_Type2
These template parameters determine types of a return image and two input images, respectively. The virtue of this approach is that all types of images are possible here.

### 7.2.4 The Composition of Image Operations

The other types of image operations are compositions of other image operations. Such compositions are also image operations and can be composed of other image operations and so on. Figure 15 explains this mechanism. The class implements the composite design pattern which can be used for representation of recursive structures [4]. Our realization is presented in Algorithm 10.

```cpp
template <class Container = DefaultContainerFor_Operation>
class TComposedImageOperationFor : public TImageOperation {

protected:

    typedef Container ImageOperationContainer;

    // here we store the registered operations
    ImageOperationContainer fImageOperationContainer;

public:

    /// This function calls function operator for each
    /// object added (with AddAdoptNewOperation) to this collection.
    /// Order of execution is dependent on the semantics of
    /// the container supplied as a template parameter (Container).
    /// It is linear for vectors and lists.
    virtual void * operator(){ void }; 

    /// This function adds a new operation to the collection.
    /// The added object is abandoned, i.e. it is left
    /// for exclusive use of the TComposedImageOperationFor
    /// object.
    void AddAdoptNewOperation(TImageOperation* op);

    /// This function adds a new operation to the collection.
    /// The added object is abandoned, i.e. it is left
    /// for exclusive use of the TComposedImageOperationFor
    /// object.
    void AddNewOperation(TImageOperation* op);

    /// INPUT:
    /// none
    ///
    /// OUTPUT:
    /// return value of the last operation
    ///
    /// REMARKS:
    /// Execute all registered
    ///
    /// operator(void); 

    /// INPUT:
    /// op - ptr to the TImageOperation object
    /// that is added to this collection
    /// The pointed object is ADOPTED by
    /// this collection
```
Algorithm 10. The template base class TComposedImageOperationFor<> which is an image operation by itself. This class implements the composite design pattern.

This template class is parameterized by the type of a container rather than the type of an image. Semantics of this container determines order of execution of image operations registered to this image operation composite.

The functional operator invokes all respective functional operators from the registered elements.

The other overloaded versions of the AddAdoptNewOperation methods are used to add image operations to this composite. It should be remembered that once submitted, an object is exclusively owned by the composite object. Finally it is destroyed during composite destruction.

7.2.5 Arithmetic Operations on Images

Figure 16 depicts the hierarchy of arithmetic operators available in HIL. There are four basic operators in this group: image addition, subtraction, multiplication, and division. Versatile template approach allows different kinds of input arguments, for example it is possible to add images or a value to an image. The only one restriction in this mechanism is existence of the implementation of a requested operation. In HIL most of this implementation is enclosed in dedicated members of the image classes.

![Class hierarchy of the basic arithmetic operators.](image)
Since all the mentioned arithmetic operations take two operands, their representative classes are derived from the \textit{TBinaryImageOperationFor<>} base template class.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{binary_operations_class_hierarchy.png}
\caption{Class hierarchy of operators implementing simple arithmetic functions of single images.}
\end{figure}

Figure 17 depicts class hierarchy of operators for simple arithmetic functions operating on single images. In this case it is the \textit{Abs\_OperationFor<>} which computes the absolute value of image elements. However, the scheme of Figure 16 and Figure 17 gives a cue for implementation of all other functions taking two or one images as their arguments, respectively.

\subsection{Logical Operations on Images}

Figure 18 depicts the hierarchy of logical operators available in HIL. There are four basic operators in this group: image and, or, exclusive-or (xor), and negation.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{logical_operations_class_hierarchy.png}
\caption{Class hierarchy of the basic logical operators.}
\end{figure}
All logical operators, except NOT, are binary and thus their classes are derived from the TBinaryImageOperationFor<> base template class. The NOT operation is unary and therefore its class derives from TUnaryImageOperationFor<>.

7.2.7 Image Format Converters

TImageFor<> is the base data structure for all image operations in HIL. However, it is possible to use other image formats, e.g. the ones from other image libraries, such as Intel's Open Source Computer Vision Library [10]. To facilitate image conversions the FormatConvert_OperationFor<> template class has been created which is a derivative of the TUnaryImageOperationFor<> template class. Thus, image conversions are unary operations.

The FormatConvert_OperationFor<> constitutes only a façade to the actual implementation which, in turn, is based on an ample suite of MixedCopy() functions. The scheme of this function is as follows:

```c++
bool MixedCopy( <destination image>, <const source image> )
```

The function takes on a destination image, then source, and returns binary status of an operation (true if successful).

New conversions can be easily implemented by providing an overloaded version of the MixedCopy() that operates on images to be converted. It is worthy noticing that no other change to source is necessary, specifically one does not need to modify the FormatConvert_OperationFor<>.

7.2.8 Colour Space Conversions

Colour space conversions are other types of image change. This time, however, the format of an image does not change. Instead, an interpretation of a pixel is changed. There are many colour spaces, i.e. mathematical models for representation of the phenomenon of colour or multi-spectral perception. Due to some hardware (e.g. monitor or printer rendering) or application reasons (e.g. graphics or animation), the most common colour spaces are: RGB, HSI, and YCrCb. The latter is a linear combination of the former [17][5]. The following conversion operators are included in the library:

1. RGB_2_HSI_Convert_OperationFor,
2. HSI_2_RGB_Convert_OperationFor,
3. RGB_2_YCrCb_Convert_OperationFor,
4. YCrCb_2_RGB_Convert_OperationFor,

The actual conversion routines were put in the RGB_2_HSI, HSI_2_RGB, YCrCb_2_RGB, and RGB_2_YCrCb classes for conversions among three colour spaces: RGB, HSI, and YCrCb.
Figure 19. Colour image and its colour channels in different colour spaces: R, G, B, and H, S, I.

Figure 19 depicts R, G, B, and H, S, I colour channels in different colour spaces of the image “Krakow” 640×480. It is evident that the channels R, G, B, and I are strongly correlated.

7.2.9 Global Operations in Images

To find specific values in images the following operations were added to HIL: FindMinVal_OperationFor, FindMaxVal_OperationFor, and Thresholding_OperationFor. Their class hierarchy is depicted in Figure 20.

These operations belong to the so called global image processing group, since they have to process the whole image contents to produce a result. All of these operators belong to the group of unary operators and therefore are derived from the TUnaryImageOperationFor<> base class.

Figure 20. Class hierarchy of the new HIL operators.
7.2.9.1 Image Thresholding

Image thresholding is a nonlinear mapping of one image on to an output image. Based on the set of threshold values, a relation is determined of each pixel value in respect to the given threshold. Based on this relation, computed for each pixel of the input image, pixels in the output image are set. In HIL it was decided to implement two-value-thresholding method, described as follows:

\[
\forall_{i,j \in Z} I_{out}(i,j) = \begin{cases} 
    v_1 & \text{if } I_{in}(i,j) < t_1 \\
    v_2 & \text{if } t_1 \leq I_{in}(i,j) < t_2 \\
    v_3 & \text{if } t_2 \leq I_{in}(i,j) 
\end{cases}
\]  

where \( t_{1,2} \) are threshold values, \( v_{1,2,3} \) output pixel value set according to relation, \( Z \) a set of allowable indexes for \( I_{in} \) and \( I_{out} \) images.

The \textit{Threshold\_OperationFor} class in HIL contains implementation of the described image thresholding technique. It is a unary global image operation and therefore \textit{Threshold\_OperationFor} derives from \textit{TUnaryImageOperationFor} base – see Figure 20.

\[\text{Figure 21. Results of image thresholding: Original image (a), thresholded around median of the intensity signal (b).}\]

Figure 21 is an example of monochrome image thresholding: exemplary image (a), and its thresholded version around a median value (b). More on thresholding methods can be found in references [5] and [11].
Bibliography


[7] [www.wiley.com/go/cyganek3dcomputer](http://www.wiley.com/go/cyganek3dcomputer)


Kraków, 2008