HALCON Version 5.2

HDevelop
User's Manual

April 21, 1999

This manual explains the interactive tool HDevelop for HALCON version 5.2

1st Edition. July 1997
2nd Edition. November 1997
3rd Edition. March 1998
4th Edition. April 1999

Information concerning HALCON:
`http://www.mvtec.com`

Address: MVTec Software GmbH
Orleansstr. 34
D-81667 Munich
Germany
E-mail: `mvtec@mvtec.com`

Contents

Chapter 1

About this Manual

This manual is a guide to HDevelop — the graphical user interface for HALCON. HDevelop facilitates rapid prototyping using the concept of *Computer Aided Vision Engineering (CAVE)*, which offers a highly interactive programming environment for designing image analysis programs. Together with the HALCON library, it is a sophisticated image analysis package suitable for product development, research, and education. HALCON provides operators covering a wide range of applications: Factory automation, quality control, remote sensing, aerial image interpretation, medical image analysis, and surveillance tasks.
This manual provides all necessary information to understand HDevelop's basic philosophy and to use HDevelop.

1.1 Readers

This manual is intended for all new users of HALCON. It does not assume that you are an expert in image processing. Regardless of your skills, it is quite easy to work with HDevelop. Anybody should be able to understand the basic HALCON principles to solve his or her image analysis problems quickly. Nevertheless, it is helpful to have an idea about the functionality of *graphical user interfaces (GUI)*[1], and about some basic image processing aspects (see, e.g., [Bal82][Jai89][Rus92]).

1.2 Organization of this Manual

Each part of the manual concerns different areas of application and shows different levels of difficulty.
The HDevelop user's manual is structured as follows:

Chapter 2 Introduction Section 2.1 explains the basic concepts of HDevelop (data structures, configuration, etc.).

Chapter 3 Example Session Chapter 3 contains a first example that illustrates working with HDevelop.

Chapter 4 Graphical User Interface Chapter 4 explains the graphical user interface of HDevelop. Sections 4.1 to 4.7 describe the interactive elements of the graphical user interface.

[1] Consult your platform's documentation for general information.

Chapter 5 Language Chapter 5 explains syntax and semantics of the language used in HDevelop programs.

Chapter 6.1 Code Generation for C++ This chapter explains the export of a HDevelop program to C++. Further information about using HALCON operators within conventional languages (C and C++) can be obtained in the user manuals HALCON/C and HALCON/C++.

Chapter 6.2 Code Generation for Visual Basic This chapter explains the export of a HDevelop program to Visual Basic. Further information about using HALCON operators with the COM interface can be obtained in the user manuals HALCON/COM.

Chapter 7 Program Examples Programming in HDevelop is explained by examples in Chapter 7.

Chapter 8 Miscellaneous Chapter 8 explains how to start HDevelop and describes keycodes, warning- and error windows, and restrictions.

Further information and hints can be found in:

Settings: The following sections explain how to set up the system:

- 2.2: General settings
- 2.3: Help files for HTML-browser
- 8.1: Starting HDevelop
- 8.2: Keyboard settings of HDevelop

Debugging: Hints for debugging can be found in:

- Sections 2.2 and 8.1: Starting the program
- Section 5: Programming HDevelop

1.3 Additional Sources of Information

For further information, you may consult the following manuals:

- Getting Started User's Manual.
- HALCON/HDevelop: The reference manual for all HALCON operators (HDevelop version).
- HALCON/C++ User's Manual: How to use the HALCON library in your C++ programs.
- HALCON/C++: The reference manual for all HALCON operators (C++ version).
- HALCON/C User's Manual: How to use the HALCON library in your C programs.
- HALCON/C: The reference manual for all HALCON operators (C version).
- HALCON/COM User's Manual: How to use the HALCON library in your COM programs, e.g., in Visual Basic.

- HALCON C-Interface Programmer's Guide: How to extend the HALCON system with your own operators.
- Framegrabber Integration Programmer's Manual: How to integrate a new framegrabber into the HALCON system.

The reference manuals are available as HTML documents as well.

1.4 Release Notes

Please note the latest updates of this manual:

- 4^{th} **Edition, HALCON 5.2 (January 1999)**
  Keyboard shortcuts have been introduced for the most frequently used menu operations, e.g., `<Ctrl> S` for `File ▷ Save`. The menu `File` contains two new entries: `Insert...` and `Modules...`. The menu item `File ▷ Read Image` now has a submenu with several useful image directories. A new menu `Execute` with entries to run, stop, and reset a program has been added. The menu items `Edit ▷ Activate` and `Edit ▷ Deactivate` have been moved to this menu. The menu item `Visualization ▷ Image Info...` has been renamed to `Visualization ▷ Pixel Info...`. A tool for realtime zooming of graphics window contents has been added under `Visualization ▷ Zooming...`. A tool to visualize the gray value histogram of an image and to select thresholds has been added under `Visualization ▷ Gray Histogram Info...`. An inspection tool for shape and gray value features of regions has been added under `Visualization ▷ Region Info...`. The menu item `Visualization ▷ Size Window ▷ Reset` has been replaced with three, more convenient, menu items `Original`, `Original half`, and `Original quarter`. Two new HDevelop operators have been added: `dev_clear_obj` and `dev_set_window_extents`. The operator window has been extended with an `Apply` button that helps you to find the optimum parameters of an operator even more quickly.
- 3^{rd} **Edition, HALCON 5.1 (March 1998)**
  The new operators `dev_set_check` and `dev_error_var`, together with the constants for return values (`H_MSG_TRUE`, etc.) have been introduced. The menu `Visualization` has been extended by `Size Window`, `Reset Parameters`, `Zooming`, and `Paint`. Three new parameter functions (`mean`, `deviation` and `gen_tuple_const`) have been introduced. The chapters 'About this Manual', 'Graphical User Interface', and 'Code Generation for C++' have been revised. The selection of program lines has been changed. Now an extended mode (`<Shift>` and `<Ctrl>` keys together with the mouse) is used. A description can be found in section 'Program Window'. The menu `Options` has been extended. A visualization of a framegrabber handle with automatic online grabbing (double click inside `Variable Window` has been introduced. Further examples for the use of different HALCON operators can be found in `%HALCONROOT%\examples\hdevelop\<Chapter>\*\*`. Online inspection of gray values and basic image features has been introduced in the menu `Visualization ▷ Image Info...`. The edit buttons `Cut` and `Copy` have been changed according to the standard order in other Windows NT tools.
- 2^{nd} **Edition (November 1997)**
  The menu `Visualization ▷ Set Parameters...` has been extended and the description has been revised.

Chapter 2

Introduction

2.1 Concepts

The following sections explain the basic ideas and concepts of HALCON.

2.1.1 Working environment

Basically, HALCON supports two methods of programming:

- Interactive development of programs via HDevelop (as described in this manual) and
- using the programming languages C, and C++. HALCON provides an interface to these languages.

While developing programs according to the second method is done using standard programming tools, HDevelop actively supports the user in different ways:

- With the graphical user interface of HDevelop operators and iconic objects can be directly analyzed, selected, and changed within one environment.
- HDevelop suggests operators for specific tasks. In addition, a thematically structured operator list helps you finding an appropriate operator.
- An integrated online help contains information about each HALCON operator, such as a detailed description of the functionality, typical successor- or predecessor-operators, complexity of the operator, costs of computation time, error handling, examples of application. The online help is based on an internet browser such as Netscape Navigator or Microsoft Internet Explorer.
- HDevelop comprises a program interpreter with edit and debug functions. It supports programming features, such as loops and conditions.
- HDevelop immediately displays the results of operations. You can try different operators and/or parameters, and directly see the effect on the screen.
- Several graphical tools allow the examination of iconic and control data.
- Variables with an automatic garbage collection are used to manage iconic objects or control values.

There are three basic ways to process and develop image analysis programs with HDevelop:

- Rapid prototyping in the interactive environment HDevelop.
- Development of an application that runs within HDevelop.
- Use HDevelop to develop an application that can be exported as C++ source code. This program can then be compiled and linked with the HALCON library so it runs as a stand alone (console) application.

2.1.2 Data Structures

HALCON distinguishes between two types of data: *iconic* and *control* data.

- *Iconic* data are all kinds of image *objects* (images, regions, or XLD objects describing areas and contours), whereas
- *control* data are all kinds of "numerical" values, such as integer, floating point numbers and also strings.

Control data defines input values for operator control parameters and is used to build complex structures like bar charts or arrays of control values. Both — *iconic* and *control* data — are processed according to the "tuple scheme" within HALCON, i.e., they may be a tuple (a number of items) consisting of several elements. HALCON always processes all elements of a tuple simultaneously. You can handle tuples just in the same way as single objects/values. For example, to filter several images you may call the filter operator several times on the different images or you can put them into one tuple iconic object and pass it as input to the filter operator. HALCON processes the filter operation simultaneously on all tuple elements and returns another tuple containing the resulting images.
We will now describe iconic data in detail to give you a better understanding how to handle image data. Basic types are *images, regions* and *XLD objects* (e**X**tended **L**ine **D**escription).
Images are defined as a (sub-)set of pixels of a rectangular index range. An image consists of one domain deftermining the image's area of definition and one or more "channels" containing the gray values of the pixels. Pixel data can be stored by the following types: `byte`, `integer` (1,2, and 4 byte), `real`, and `complex`. While processing an image, an operator exclusively works on the pixels inside the domain of the image. All pixels outside are ignored and may even have invalid (undefined) values. The domain (i.e., region of interest) can be of *any* shape and is not restricted to rectangles like in many other systems. The region of interest concept allows you to *focus* the image processing. The amount of data to work on becomes smaller and the processing is much faster.
Regions are arbitrary subsets of $\mathbb{Z} \times \mathbb{Z}$. They are used to define image areas of any size and shape at pixel precision. To reduce memory costs and to speed up the region processing they are stored by a runlength encoding. Regions may define the domain of an image but can also be used — a basic feature of HALCON — as a flexible data structure of its own. The size of regions is *not* limited to the size of images. This strongly influences the effect of operations, e.g., in morphology, since it prevents image border artefacts. Moreover, regions may overlap without an implicit merging that would happen when storing them as images. This results in a flexible usage, e.g., for describing segmentation results or regions of interest.

XLD is the abbreviation for e**X**tended **L**ine **D**escription. This is a data structure used for describing contours of areas (e.g., arbitrarily sized regions or polygons) and lines. An XLD object is built up by setting base points along the contour/lines and connecting them with lines. They can represent a contour (or set of lines) at any precision by varying the number and position of the base points at subpixel precision. Additional information, such as gradient, filter response, or angle is stored with the points and lines. Typically, XLD objects describe the result of operators for edge/line detection (e.g., `edges_sub_pix`, `lines_gauss`). In particular, they are used for image processing at subpixel precision and offer a wide range of features and transformations.

2.1.3 Module Structure

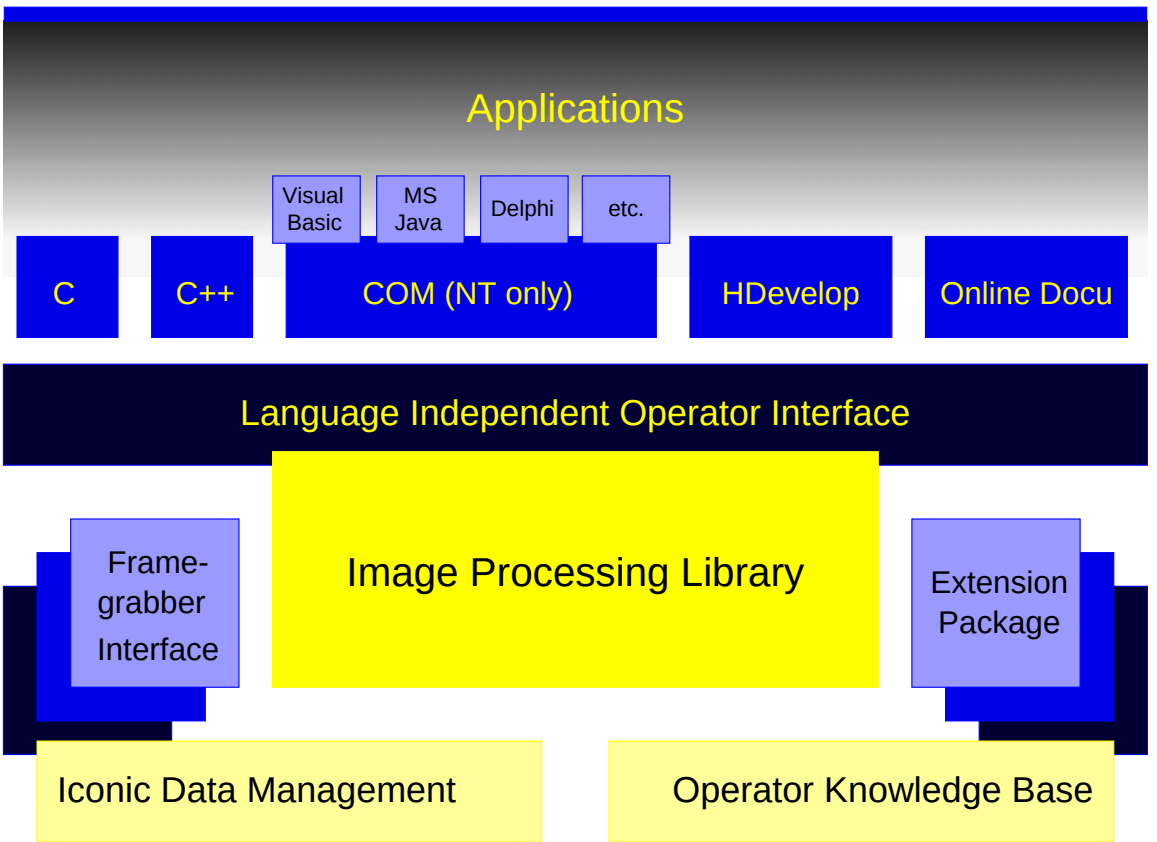


Figure 2.1: *Layered structure of HALCON.*

The basis of HALCON is the *data management* together with the *operator knowledge base*. The data management is reponsible for basic memory management (optimized for image processing) and on top of this for the creation, handling, and deletion of iconic objects and tuples. The knowledge base stores information about operators, which can be accessed while the system is running. All other modules make use of it to process image operations and to obtain information about the configuration and current state of the system. It is also used to *automatically* generate the HALCON language interface (for C and C++), all operator information used by HDevelop, the online help, and the reference manuals. The operator knowledge base contains information about names and number of operators, as well as parameter types, assertions and suggested values for the parameters. This data set is stored in a structured way. There exists a detailed

description for each operator which handles the effects of the operator and which indicates the complexity, the operator class, as well as cross references and alternatives.
On top of these two modules the operators are implemented. Most of them are contained in the HALCON image processing library, which is decribed in the Operator Reference Manual. This library can dynamically be extended using so-called *packages*. The operators of these packages behave like normal HALCON operators, but they contain extensions to the standard library either generated by MVTec (for easy update) or by the user. This concept permits the user to extend the system in a very flexible way. For more information see the C-Interface Manual. Using a similar mechanism, framegrabbers are integrated using dynamic libraries. This allows the user to integrate his/her own framegrabber or to download new releases of framegrabber interfaces via the internet for fast and easy update without further changes of the rest of the system.
The next module is the *language-independent operator interface*. It contains modules for calling operators, handling the input/output and passing data objects to the host language. An operator call is performed by the *language interface* which is generated *automatically* by a compiler using the information in the operator data base. It allows to access HALCON operators within the programming languages C, C++, and COM (Windows NT only). The interface to C++ allows the development of object oriented tools.
HDevelop is implemented on top of the language interface using HALCONC and the GUI toolkit wxWindows developed by the Artificial Intelligence Applications Institute at The University of Edinburgh to get a portable user interface for Windows and UNIX.

2.2 Configuration

The following section describes the environment variables that are used by the HALCON library and HDevelop. A detailed description can be found the the Getting Started Manual. The description here is given for the UNIX environment.

- `HALCONROOT`
  This is the most important environment variable. It designates the directory in which HALCON is installed. Typical paths might be, e.g., `/usr/local/halcon` or `/usr/halcon` on Unix systems, or `C:\Program Files\MVTec\Halcon` on Windows NT systems. According to this variable the system can access subdirectories that are important for running HALCON. Some of them are:
  - `$HALCONROOT/help`
    Here the operator data base is situated. It is accessible by all HALCON programs to obtain information about HALCON operators.
  - `$HALCONROOT/doc/html/reference/hdevelop`
    HDevelop uses this directory for online help, which can be displayed by a suitable HTML browser like Netscape Navigator.
  - `$HALCONROOT/lut`
    User defined look-up tables are situated in this directory.
  - `$HALCONROOT/ocr`
    This directory includes trained fonts.

  - $HALCONROOT/license
    This directory contains the license keys necessary for using HALCON.
  - $HALCONROOT/images
    If the variable HALCONIMAGES (see below) is not set the system looks for image files in this directory.

- HALCONIMAGES
  To search for image files specified by a relative path, the system uses this environment variable. Usually it contains several directory names, which are separated by colons on Unix systems and semicolons on Windows NT systems.

- ARCHITECTURE
  Executable HALCON programs reside in $HALCONROOT/bin/$ARCHITECTURE. To indicate shared libraries the directory $HALCONROOT/lib/$ARCHITECTURE is used. ARCHITECTURE designates the used platform by an abbreviation (e.g., hppa1.1-hp-hpux9 or sparc-sun-solaris2.5; syntax: *processor-hardware_vendor-operating_system*) or does not have any content (i.e., it is empty or undefined), if the installation for HALCON was made for one architecture only. It is useful to include the path $HALCONROOT/bin/$ARCHITECTURE or $HALCONROOT/bin in the path (environment variable PATH) of a shell script.

- LD_LIBRARY_PATH and SHLIB_PATH
  Using an HP-UX architecture, the HALCON library path should be included in the environment variable SHLIB_PATH to access shared libraries. For all other Unix architectures the environment variable LD_LIBRARY_PATH is used.

- HALCONSPY
  If this environment variable is defined (regardless of the value) before you start a HALCON program, the HALCON spy tool is activated. This corresponds to executing the HALCON operator set_spy("mode","on") in a HALCON program. With this, it is possible to monitor an already linked HALCON program during runtime without modifications.

  Additionally, you may specify any monitoring mode for the HALCON debugging tool. For this you choose a parameter value (e.g. 'operator', 'input_control', 'output_control', etc.) and activate it. This is done by adding these parameters to the envitonment variable HALCONSPY:

  ```
  setenv HALCONSPY operator=on:input_control=on
  ```

  Please note that the default output of spy is "standard out", which might lead to problems in a Windows NT environment. For further information concerning set_spy, please refer to the HALCON reference manual.

- HALCONEXTENSIONS
  This is a list of directories in which user-defined extension operators (so-called packages) are kept. Each package consists of a number of operators linked into a shared library, plus the additional operator documentation in help-files and HTML files. See the C-Interface Programmer's Manual for details on user packages.

- DISPLAY
  On Unix systems, HALCON uses this environment variable to determine the display on which to open windows. It is used in the same way as for other X11 applications.

- HOME
  If you start HDevelop the system searches for a startup file in the home directory (see page 125). The corresponding directory is designated by the variable HOME.

Typically, the definition of these environment variables belongs to a start shell script like .cshrc.

Examples for entries in .cshrc	
setenv HALCONROOT	/usr/local/halcon
setenv ARCHITECTURE	hppa2.0-hp-hpux10
setenv PATH	...:$HALCONROOT/bin/$ARCHITECTURE:...
setenv LD_LIBRARY_PATH	...:$HALCONROOT/lib/$ARCHITECTURE:...
setenv HALCONIMAGES	$HALCONROOT/images:.

On Windows NT systems they are set by the installation program. They may be changed by the System Settings tool available from the Start menu.
You may obtain additional information concerning configuration in the Getting Started Manual.

2.3 Online Help

Online documentation is available in PDF and partly in HTML format. In a UNIX environment, the full documentation is available in postscript format as well.
To display the HTML files containing information on HALCON operators, you need a browser. It is not provided in the HALCON distribution, but nevertheless used by HDevelop. Such a tool may already be installed on your computer. Otherwise you may obtain it for free, e.g., via the Internet. One browser that is suitable for displaying HTML files is Netscape Navigator. It is a WWW browser that is able to display HTML documents. Since the reference manual for HALCON operators is also stored in HTML format, it is convenient to use a standard WWW browser. In the tool HDevelop you may call Netscape via the menu Help ▷ html-help. It will start Netscape with the corresponding help files (see page 57). An alternative to Netscape is to use the Microsoft Internet Explorer.
Besides HTML, the documentation is available in PDF format as well. To display the manuals, the Adobe file viewer Acrobat Reader is included in the distribution for Windows NT systems. This viewer is not activated from HDevelop, but has to be started from the start menu of Windows NT.

Chapter 3

Example Session

To get a first impression how to use HDevelop, you may have a look at the following example session. Every important step during the image processing session is explained in detail. Thus, having read this chapter thoroughly, you will understand the main HALCON ideas and concepts. Furthermore, you will learn the main aspects of HDevelop's graphical user interface (for more details see chapter 4). A simple introduction can be found the in Getting Started Manual as well. In this example, the task is to detect circular marks attached to a person's body in a gray value image. The program can be found in the file

`%HALCONROOT%\examples\hdevelop\Manuals\HDevelop\marks.dev`

After starting HDevelop (see pages 12 and 125), your first step is the loading of the image `marks.tif` from the directory `%HALCONROOT%\images`. You may perform this step in three different ways:

- First, you may specify the operator name `read_image` in the operator window's input text field.
- Second, you may select this operator in `Operators` ▷ `File` ▷ `Images` ▷ `read_image`.
- The most often used and most convenient way is the third one. Here, you open the image selection box pressing menu item `File` ▷ `Read Image` ▷ `...` The menu `File` ▷ `Read Image` contains several predefined directories, one of which is `%HALCONROOT%\images`. Usually, this directory will be `C:\Program Files\MVTec\Halcon\images`. Select this directory by pressing the appropriate menu button. Now you can browse to your target directory and choose a file name. By clicking the button `Open`, a dialog window appears, in which you may specify a (new) name for the iconic variable which contains the image you are about to load. The variable will be used later in the program to access this image.

  To facilitate the specification process, HDevelop offers you a default variable name, which is derived from the image's file name. Pressing the button `Ok` transfers the operator into the program window and inserts a first program line, similar to the following line, into your program:

  ```
  read_image (Marks, 'C:\\Program Files\\MVTec\\Halcon\\images\\marks.tif')
  ```

  This new program line is executed immediately and the loaded image is displayed in the active graphics window. Please note the double backslashes, which are necessary since

a single backslash is used to quote special characters (see page 73). In our example we change the default for the name from `Marks` to `Christof`.

Using this selection box, you are able to search images rapidly without knowing their exact file names. In contrast to the two other possibilities, the parameters of operator `read_image` are specified automatically. Thus, an explicit input of path and file name is not necessary in this case. An icon with an appropriate variable name is created in the iconic variable area of the variable window. Double-clicking on such an icon displays its contents in the currently active graphics window. Figure 3.1 shows a complete configuration of HDevelop for the explained scenario. In addition, a new window is opened — after closing the default window — to display the image in its original size.

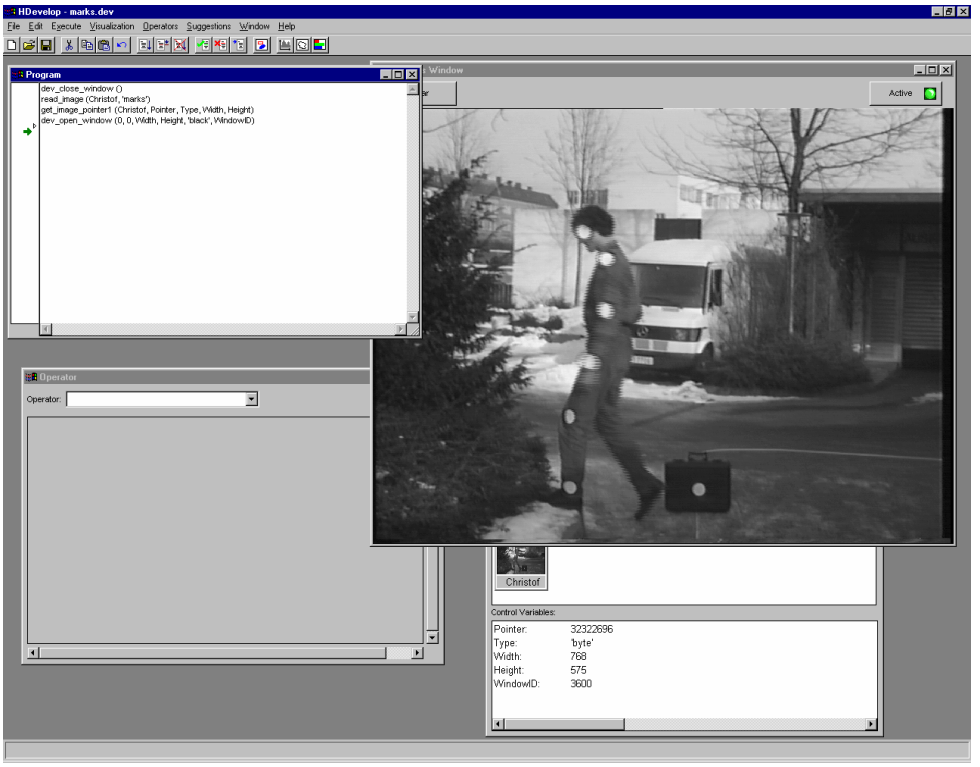


Figure 3.1: *Screen configuration after image loading.*

If you look closer at the image in figure 3.1 you will see the typical temporal offset between two half images that occurs when taking images with a video camera. This temporal offset is 20 ms for PAL systems and 16.6 ms for NTSC systems. HALCON offers an algorithm that computes an interpolated full image from such a video image. The name of the appropriate operator is `fill_interlace` (see the HALCON Reference Manual). The next step is to specify this name in the operator window's operator name field. If it is indicated completely, HDevelop shows the operator immediately in the operator window. Now you have to specify the variable name of your image. For this you put in the name `Christof` in the parameter field `ImageCamera`. To do so you have two possibilities:

- Direct input via the keyboard.
- Using the combo box that is associated with the parameter text field, you may choose an appropriate name.

The system's suggestion for the interpolated image is `ImageFilled`. By clicking button `OK` you insert this operator into the program and execute it immediately. The computed image is displayed automatically in the active graphics window.
In the next step you try to separate bright from dark pixels in the image using a thresholding operation. In this case, a segmentation using the simple thresholding operator `threshold` does not result in a satisfying output. Hence you have to use the dynamic thresholding operator `dyn_threshold`. For execution you need the original image (i.e., the interpolated full image) and an image to compare (containing the thresholds). You obtain this image by using the smoothing filter, e.g., `mean_image`. As input image you choose your original image `ImageFilled`. After estimating the marks' size in pixels, you specify a filter size which is approximately twice the marks' size (compare the HALCON Reference Manual entry for `dyn_threshold`).

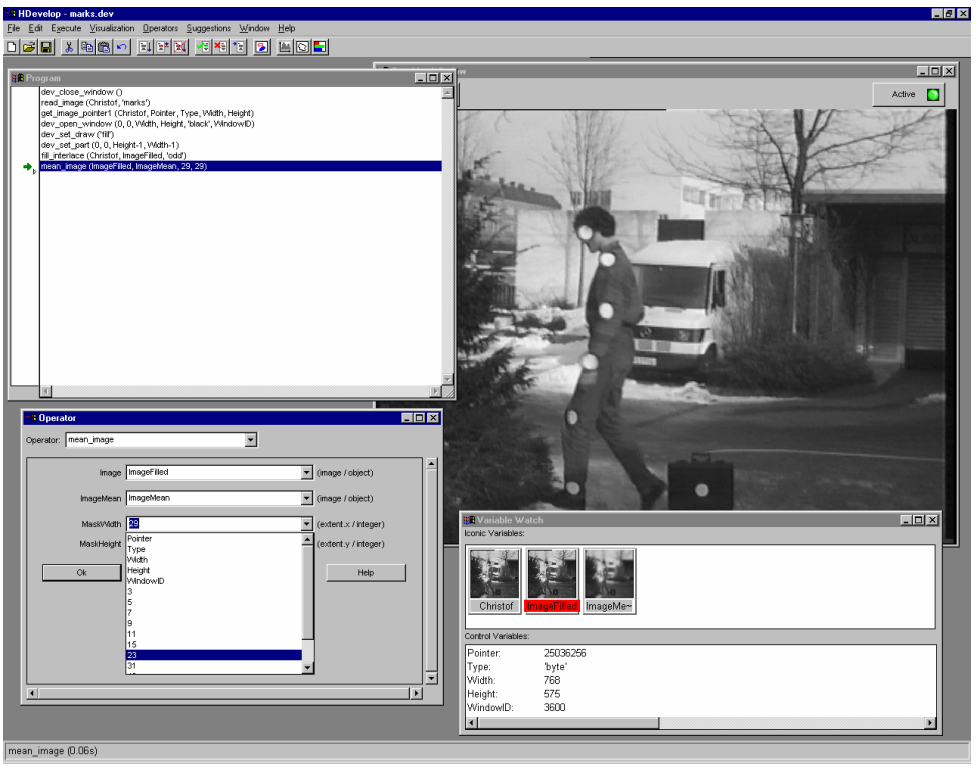


Figure 3.2: *Selecting* `Operators` ▷ `Filter` ▷ `Smoothing` ▷ `mean_image` *opens an operator dialog for the operator* `mean_image`. *With the help of the opened combo box you may specify a reasonable value for the filter size.*

To choose the operator `mean_image`, you traverse the menu hierarchy `Operators` ▷ `Filter` ▷ `Smoothing` ▷ `mean_image`. It will be displayed in the operator window immediately. Now you specify the image variable names `ImageFilled` in the text field called `Image` and `ImageMean` in the output text field. The filter matrix size is chosen by opening the combo boxes of the corresponding text fields (`MaskWidth`, `MaskHeight`). These combo boxes contain a selection of reasonable input values which is offered by HDevelop. In our example the size is set to 29 (see figure 3.2).
By clicking the button `OK` you insert the operator `mean_image` in the program and execute it. Now you have to search for the name of the dynamic thresholding. For this you specify a

substring that is included in the operator name in the operator window's operator name text field. Three letters are already sufficient to produce a result. You will notice the open combo box that presents all HALCON and/or HDevelop operators containing the first three specified letters. Now you are able to select the operator `dyn_threshold` and to specify its input parameters. The value `ImageFilled` is used for `OriginalImage`. `ImageMean` is used as the component to compare (here `ThresholdImage`). For the output parameter `RegionDynThresh` the variable name remains unchanged (see figure 3.3).

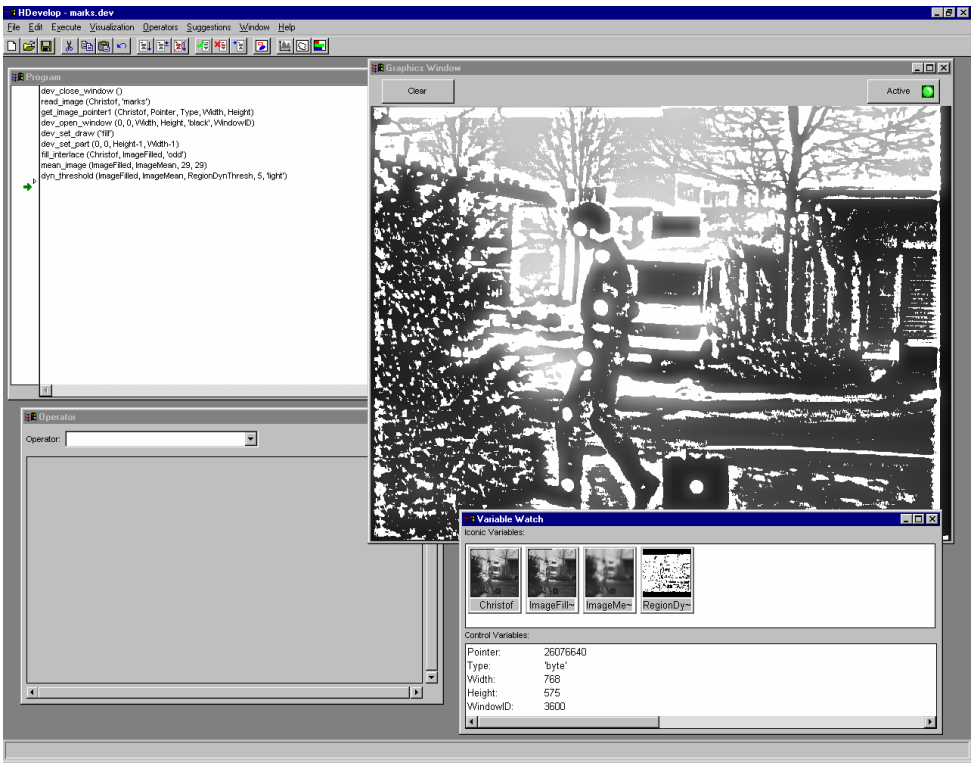


Figure 3.3: *The displayed image is the threshold operation result.*

Image pixels touching each other and remaining above the given threshold have to be merged to single regions. The computation of these connected components is realized by operator `connection` (menu item `Operators` ▷ `Regions` ▷ `Transformations`). The input region `RegionDynThresh` is specified in the text field `Region`. The output variable's default name `ConnectedRegions` is changed to `ConnectedRegionsDynThresh`. After the operator's execution all resulting regions are stored in this output variable. This shows a great advantage of HALCON's tuple philosophy: although you have several different results you do not have to worry how to handle them. This is done transparently by HALCON. HALCON operators recognize a tuple type variable and process it accordingly. This results in more compact programs because you may combine several similar operator calls in one operator.
To obtain a better visualization of the results after calling `connection` you select the menu `Visualization` ▷ `Colored`. Here you specify the 12 predefined color presentation. Now every computed region receives one of the 12 colors. This presentation mode is very useful to indicate each region with a different color. If there are more than 12 regions the system uses the same color for several different regions. With `dev_display` of the image `ImageFilled` you refresh the graphics window to see the results of next step much better. Select the menu

`Operators` ▷ `Develop` ▷ `dev_display`.
In the next step you have to specify the regions which correspond to the circular marks of the indicated person in shape and size. For this you have to call the operator `select_shape` in menu `Operators` ▷ `Regions` ▷ `Features`. The first call is used to obtain a preselection with the estimated object size given in pixels. With an estimated size of 15×15 pixels you will get approximately 225 pixels. After choosing `select_shape` you specify the parameters as follows:

(i) The input region will be `ConnectedRegionsDynThresh`,

(ii) the output variable name remains unchanged,

(iii) values `'area'` and `'and'` remain unchanged.

(iv) The region's minimum size should be 150 (`Min`) and

(v) the region's maximum size should not exceed 500 (`Max`).

(vi) The mean intensity should be between 120 and 255.

In figure 3.4 the extended program can be seen.

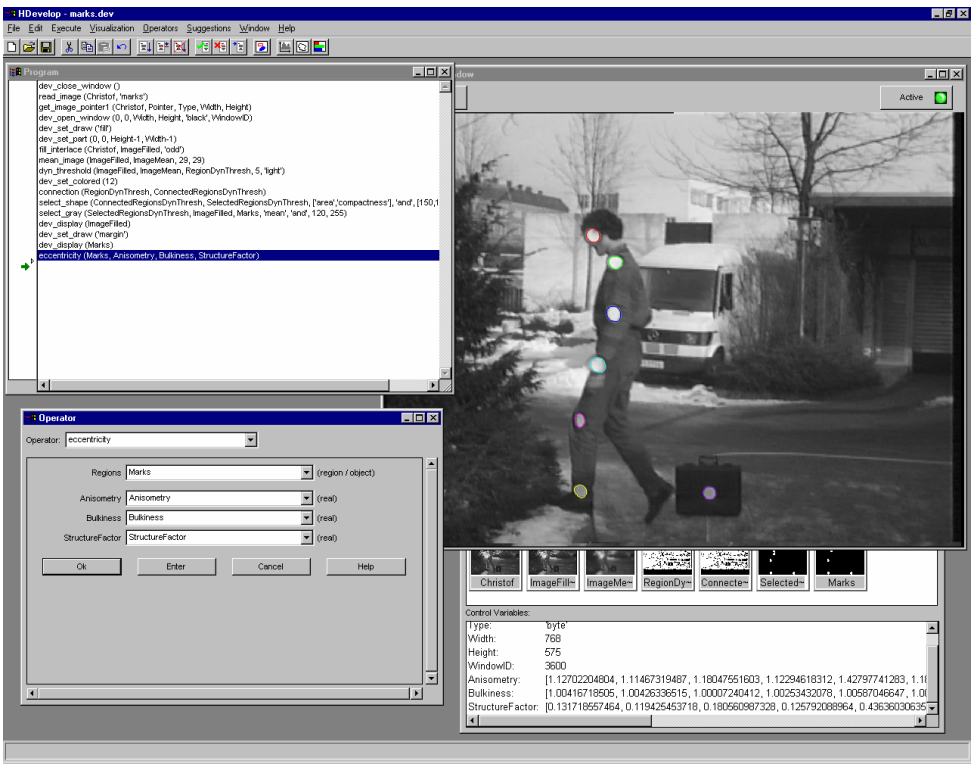


Figure 3.4: *Region selection based on shape features.*

Now you have to extract from the remaining regions the regions that match the objects to look for. As you can see, the regions representing the marks have a circular shape — contrary to all others. This is expressed by a compactness value close to 1. For this you have to choose the operator `select_shape` once again. Use the combo box of the parameter text field `Features` to specify the value `compactness`. As a range of values you may specify

the parameters `Min` and `Max` with the values 1.0 and 1.4, respectively. The output regions of the first call (`SelectedRegions`) are the input regions for the second call of operator `select_shape`. The output parameter's name `SelectedRegions` is replaced by the name `SelectedRegionsDynThresh`.
The last step to make the application stable is to add a selection of regions based on *gray value* features. This is done by using the operator `select_gray`. In this case, the mean gray value is used to discriminate the objects.
Finally, we want to obtain some numerical information about the matched marks. For example, we might want to compute three shape features of the marks. They are derived from the regions' geometric moments. The calculation is done by the operator `eccentricity`. The input parameters are all regions of the variable `Marks`. The computed values `Anisometry`, `Bulkiness`, and `StructureFactor` are displayed as a list (a tuple in HALCON terminology) in the variable window. Figure 3.5 shows the example session's result.

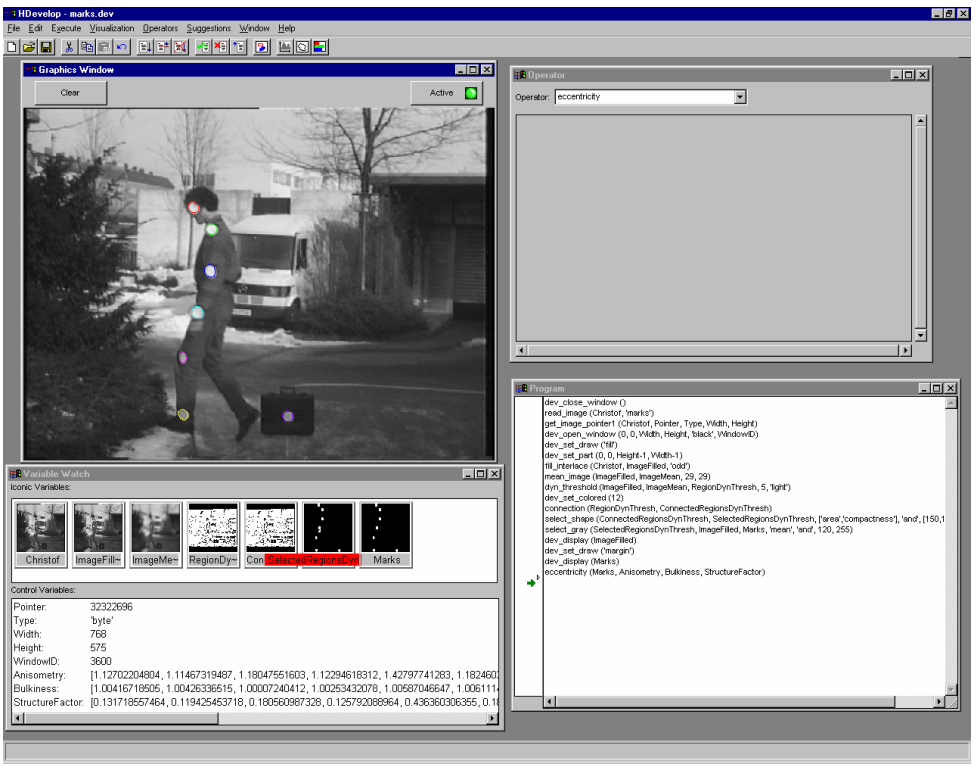


Figure 3.5: *After calling the operator* `eccentricity` *the output parameters are displayed in the variable window in tuple notation.*

As you can see in figure 3.5 and 3.4 it is possible to combine the two successive calls of `select_shape` into one call. This reduces the length of the program and saves runtime.

Chapter 4

Graphical User Interface

HDevelop is an *interactive tool* to create image analysis programs. It comprises an *editor*, an *interpreter* with debug functions, a *management unit* for variables (iconic and control data) and extensive possibilities to visualize iconic data. You may use HDevelop for *rapid prototyping* as well as for developing complete programs. You create a program by choosing operators and specifying their parameters. To do so, you may use default values or system-proposed values. After having selected the appropriate parameters, you execute the operator and insert it into the program text (i.e., the program window). You can modify and verify your generated program interactively. All intermediate results (variables) are displayed graphically (images, regions and polygons (XLD)) or textually (numbers and strings). If you want to get a detailed step by step introduction, have a look at the Getting Started Manual.
You start HDevelop (on Windows NT) by calling:[1]

`Start ▷ Programs ▷ Halcon ▷ HDevelop`

After starting the tool, the main window, which includes the following windows, will appear on your screen (see also figure 4.1):

- a program window,
- an operator window,
- a variable window, and
- a graphics window.

In the following you will learn the functionality of these five windows and their effective use while creating HDevelop programs.
Please note that in the UNIX environment the main window, the program window, and the operator window are combined into one window. Thus, there are only *three* windows.

4.1 Interacting with HDevelop

You interact with HDevelop through its graphical user interface. With the mouse you can manipulate visual controls such as menus or buttons in the HDevelop windows.

[1] In chapter 8.1 you can obtain further information on how to start the tool (see page 125). The necessary settings for the operation system are described in chapter 2.2 on page 12

4.2 Mouse Handling

You can use the mouse as follows:

- Clicking the left mouse button once,
  you are able to *select* window-specific components, such as menu items, iconic variables, control variables, action buttons, checkboxes, and you give the insertion focus to a specific text field. Some of these text fields comprise a combo box which you may open in the same way.

  Furthermore, you select (invert) text in certain windows, e.g., in the program window. With this you are able to perform the general editor functions like `cut`, `copy` and `paste` (see sections 4.3.2 and 4.3.3).

  In the program window there is an extended mode to select lines by pressing the `<Shift>` or the `<Ctrl>` key during the mouse click. More than one line can be activated using the `<Shift>` key: All lines between the last activation and the new one will become activated. The `<Ctrl>` key is used to active or deactivate more than one line using single mouse clicks.

  Clicking at an item for the second time (after a short pause) will deactivate it (e.g., lines in the program window or variables in the variable window). Similarly the activation passes to another item by clicking at it.

  Very important is the possibility to set the program counter (`PC`) at the left side of the program window (see 4.4). By combining a mouse click with special keys you can activate further functions:

  - Clicking the left mouse button once while pressing the `<Shift>` key:
    This places the insert cursor in the program window (see 4.4).
  - Clicking the left mouse button once while pressing the `<Ctrl>` key:
    The breakpoint will bet set in the program window. By performing this action once more, the breakpoint will disappear (see 4.4).

- Clicking the left mouse button twice
  results in an action that will be performed with the activated item. In the program window the operator corresponding to the program line together with its parameters is displayed directly in the operator window and can then be modified.

  Iconic and control variables are displayed in the graphics window or in specific dialogs.

4.3 Main Window

The *main window* contains the other four HDevelop windows and possibly additional graphics windows.[2] The main window can handle HDevelop programs, manipulate the graphics output, offer all HALCON and HDevelop operators, give suggestions and help on choosing operators and manage the HDevelop windows. After starting HDevelop you will see a window configuration similar to figure 4.1.
The main window comprises five areas:

[2] In a UNIX environment the main window comprises the program window and the operator window. It has no special Window manager functionality like in Windows NT.

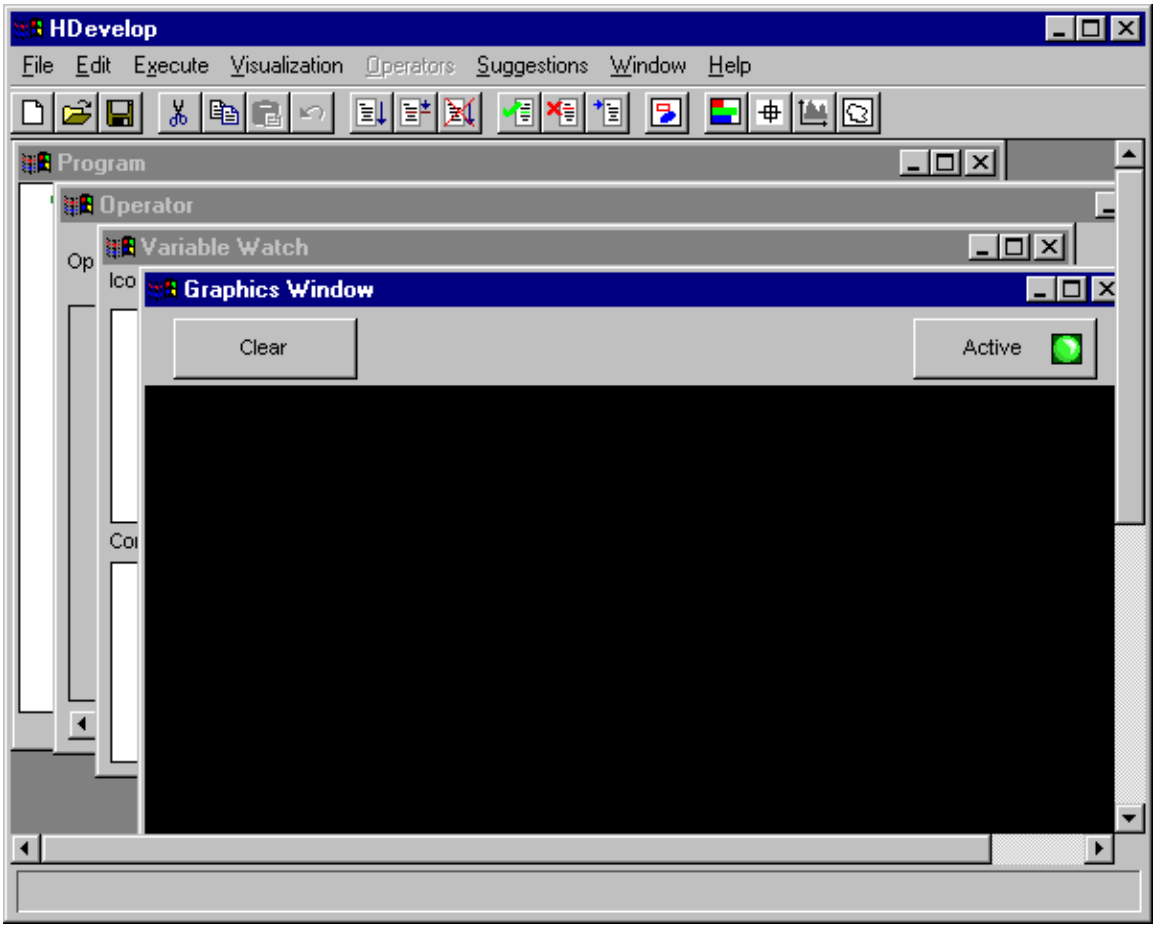


Figure 4.1: *The main window.*

- a title bar,
- a menu bar,
- a tool bar,
- a window area, and
- a status bar.

In the following chapters you will find all necessary information to interact with this window.

4.3.1 Title Bar

Your HDevelop main window is identified by the title `HDevelop` in the window's title bar. After loading or saving a file, the file name will be displayed in the title bar. Additionally it offers three buttons on the right hand side to iconify and to maximize the window, and to exit the HDevelop session.

4.3.2 Menu Bar

In the menu bar of the main window HDevelop functionality is offered. Here you may perform the important actions to solve your image processing tasks, i.e., to choose any HALCON or HDevelop operators or to manipulate the graphical output. Every menu item opens a *pull-down* menu (henceforth abbreviated as menu) with optional submenus. You open a menu by clicking a menu item (inside the appropriate text) or via the keyboard (by pressing the key `<Alt>` in combination with the underlined letter of the menu item). All menu items are going to be explained in the following.

Menu item: `File`

In the menu item `File` you will find all functions to load an image and existing programs and to save recently created or modified programs, respectively. Furthermore, you may export HDevelop programs to C++ and Visual Basic. Figure 4.2 shows all the functions in this menu item.

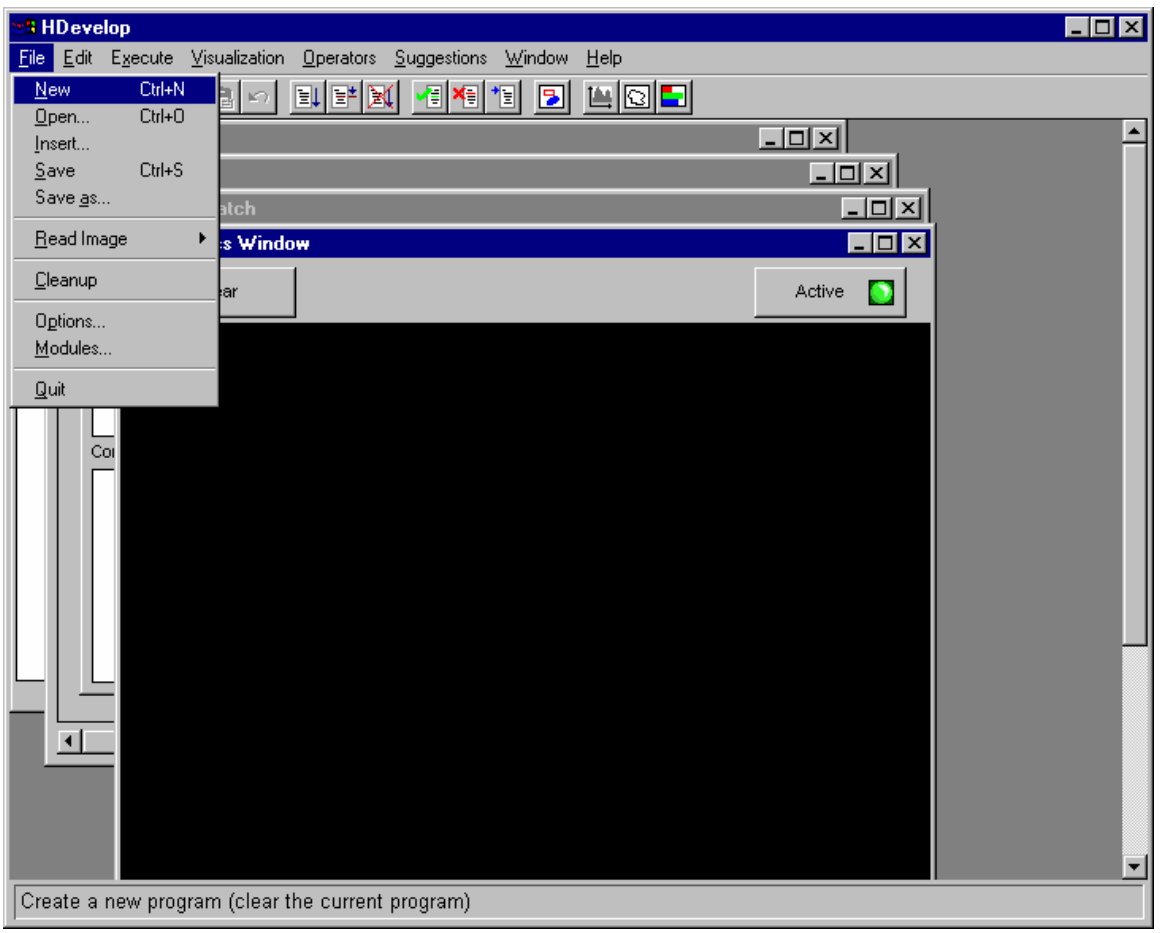


Figure 4.2: *The menu item* `File`.

- The menu item `File` ▷ `New` (keyboard shortcut `<Ctrl>` `N`) deletes the current program and all associated variables. The contents of the variables are deleted before removing them. In addition, all graphics windows except one are closed. The last window will be

cleared. The display parameters for the remaining graphics window are identical to those when starting HDevelop. The first four parameters of the menu `File ▷ Options` are reset to their initial state: The update of windows, variables, `PC`, and time is on.

A security check prevents you from deleting the current program accidentally if the program has not been saved. A dialog box appears and waits for your response, whether you really want to delete the HDevelop program. Your confirmation only deletes the HDevelop program you are working on and not the file associated with it. Now you are ready to input a new HDevelop program. If you give a negative response, nothing will happen. You have to press one of the two buttons before you are able to continue interacting with HDevelop.

- By clicking on the menu item `File ▷ Open...` (keyboard shortcut `<Ctrl> O`) you can load an existing HDevelop program. Alternatively, you can select `File ▷ Insert...` to insert a file into the current program at the line in which the insert cursor is located. In both cases, a dialog window pops up and waits for your input (see figure 4.3). It is called `Load HDevelop Program File`. Please note that text, Visual Basic and C++ versions of a file cannot be loaded.

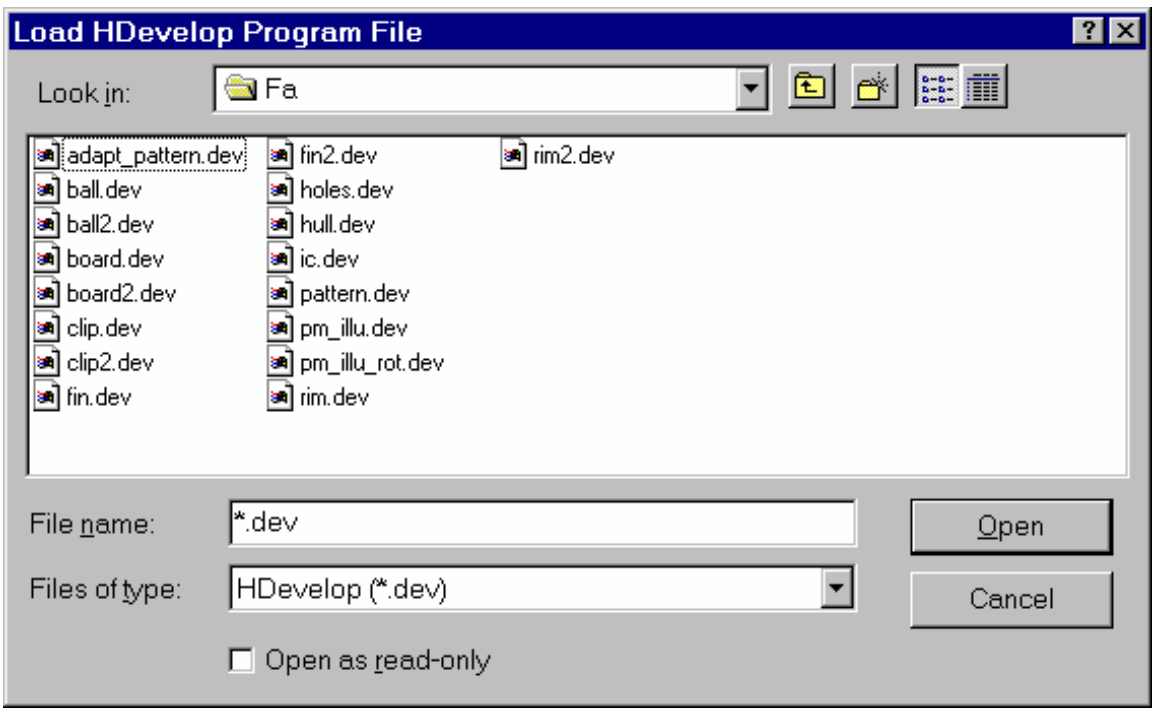


Figure 4.3: *The dialog window to open an HDevelop file.*

In the topmost text field you may specify a directory which contains your HDevelop programs. A combo box at the right hand side helps you browsing your directories. To move one directory level up, you press the button on the right hand side of this text field. The next button creates a new folder to store HDevelop programs. By pressing the last button you can activate or deactivate the option to see more details about your HDevelop programs, i.e., the program size, the program type, the date when the most recent user update occurred, and file attributes.

The middle text area displays all available HDevelop files to choose from. By clicking the left mouse button on a file name you select it. Double-clicking a file name opens the file immediately and displays it in the program window (see chapter 4.4).

Furthermore, you may specify the file name in the text field below. The combo box for file type has no effect because only HDevelop programs with the extension `.dev` can be loaded. If you want to open your file with a write protection choose the checkbox at the bottom of this dialog window. To open your specified file, you press the `Open` button. This action deletes an already loaded HDevelop program and all created variables. The same actions as with `File ▷ New` are performed. Now you can see your new program in the program window. The file name is displayed in the title bar of the main window. All its (uninstantiated) variables are shown in the variable window. To indicate that they do not have any computed values, the system provides the iconic and control variables with a question mark. The program counter is placed on top of your program and you are ready to execute it. The visualization and options will be reset after loading (same as `File ▷ New`).

If you want to cancel this task you have to press the corresponding button. By using one of these two buttons (`Open` or `Cancel`) the dialog window disappears.

After you have loaded a program the corresponding file name will be appended at the end of the menu `File` (after the menu item `Quit`). This allows you to switch between recently loaded files quickly. The most recently loaded file is always listed first.

- The menu item `File ▷ Save` (keyboard shortcut `<Ctrl> S`) saves the current program to a file. If no file name has been specified so far, the dialog corresponding to `File ▷ Save As...` will be activated.

- Menu item `File ▷ Save As...` saves the current program to a file. The type of file (HDevelop, text, Visual Basic or C++) can be selected (see Figure 4.4).

  A dialog box (similar to the window described in menu item `File ▷ Open...`) is opened in which you can specify the appropriate settings. You may specify a new file name and a directory where to locate this file. You may indicate whether the HDevelop program remains a HDevelop program or is transformed to a C++, Visual Basic or an ASCII file. This is done by clicking the combo box of the text field called `Files of type`. In UNIX the selection of the file type has to be done by entering the corresponding file extension manually. For C++ code you have to add `.cpp` to the file name, for ASCII you have to add `.txt`. Default type is the HDevelop type (extension `.dev`). The details of the C++ code generation are described in chapter 6.1. The extension for Visual Basic is `.bas`. The export to Visual Basic is described in Chapeter 6.2.

  Similar to loading, the file name of the program you save is appended at the end of the menu `File`.

- The menu `File ▷ Read Image` contains several directories from which images are usually loaded. The first entry of this menu always is the directory from which the most recent image was loaded. This is useful, when several images from a non-standard directory must be read. The remaining entries except the last one are the directories contained in the `%HALCONIMAGES%` environment variable. The final directory, denoted by `.`, is the current working directory of the HDevelop program, which usually will be `%HALCONROOT%`

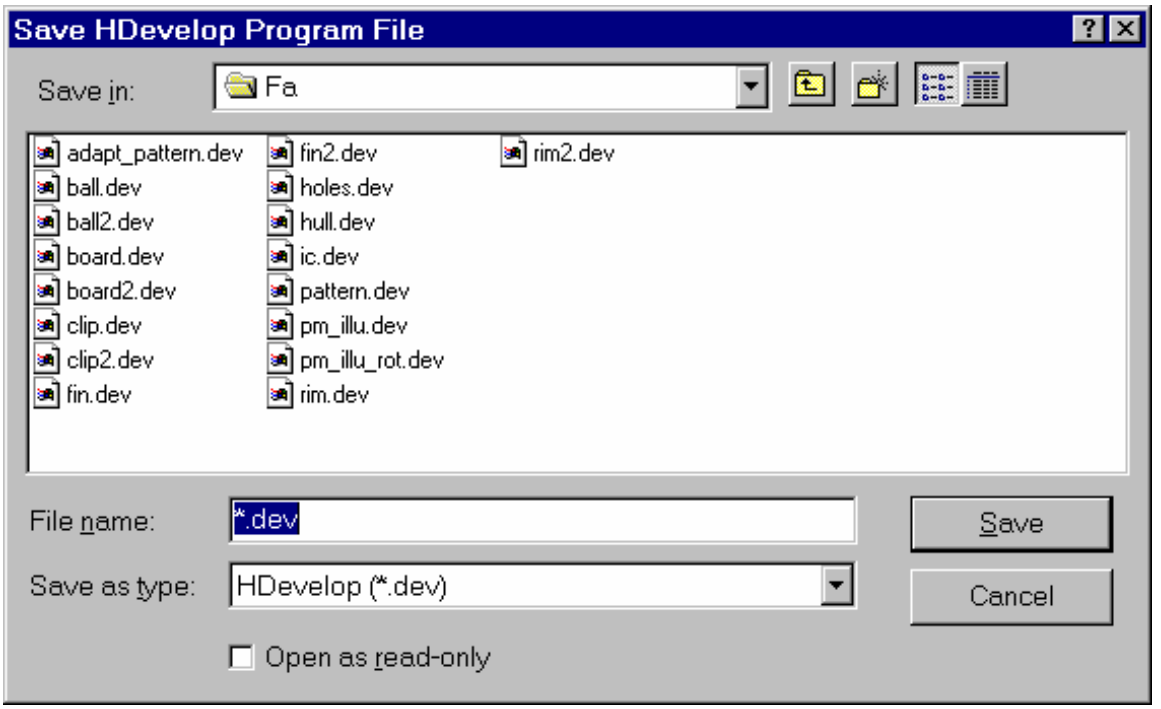


Figure 4.4: *The dialog window to save a program to a file*

on Windows NT systems, and the directory in which HDevelop was started on Unix systems.

When any of the directories is selected, an image file selection box appears. Again, its functionality is similar to the dialog discribed in menu item `File ▷ Open...`. Figure 4.6 shows an example of the `Load Image File` dialog.

After selecting a file name, the name of the variable for the image in the HDevelop program has to be selected. To do this, a small dialog appears after pressing `Open` or double clicking a file. For easy handling, HDevelop suggests a name derived from the selected file name. You may adopt or edit this name. If you want to use a name of an already created iconic variable, a combo box offers you all iconic variable names. To do so, you click the button on the right side of the text field. Note that the reuse of a variable name deletes the old content and replaces it with the new image.

- The menu item `File ▷ Cleanup` deletes all unused variables (iconic and control data). These are variables in the variable window that are no longer used in any operator of the program. This can happen after the deletion of program lines or after editing variable names, because the corresponding variables are not deleted automatically. You may use this menu item during a longer editing process to reorganize your variable window (see also page 65).

- Menu item `File ▷ Options...` opens a control window, which you can use to modify output behavior during runtime (see figure 4.7).

  – `Update PC`
  The first item (see page 60) concerns the display of the current position while run-

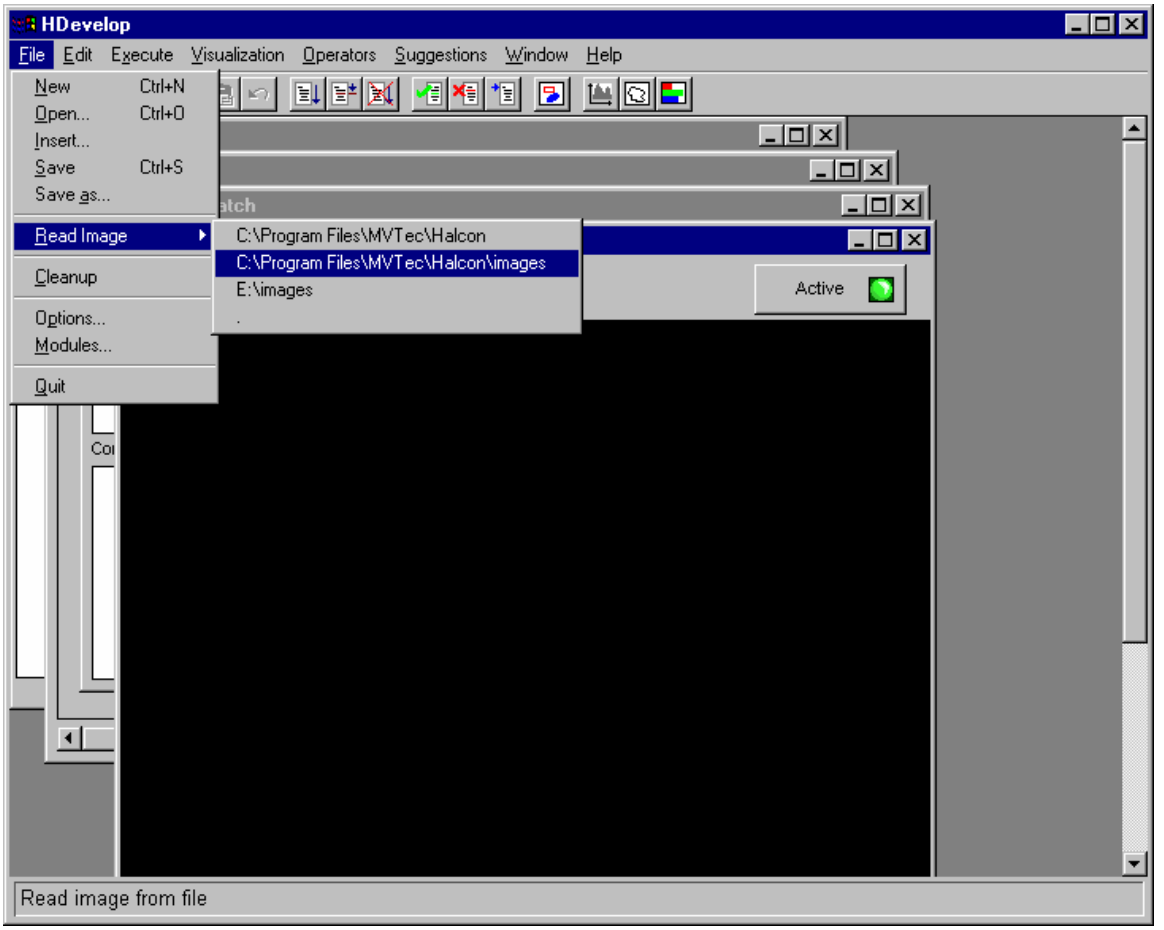


Figure 4.5: *The menu item* `File ▷ Read Image`.

ning the program. The so called `PC` (*Program Counter*) always indicates the line of the currently executing operator or the line before the next operator to execute. Using the `PC` in this way is time consuming. Therefore, you may suppress this option after your test phase or while running a program with a lot of "small" operators inside a loop.

– `Update Variables`
This checkbox concerns the execution of a program: Every variable (iconic and control) is updated by default in the variable window (see page 65). This is very useful in the test phase, primarily to examine the values of control data, since iconic data is also displayed in the graphics window. If you want to save time while excecuting a program with many operator calls you may suppress this output. Independent of the selected mode, the display of all variables will be updated after the program has stopped.

– `Update Window`
This item concerns the output of iconic data in the graphics window after the execution of a HALCON operator. With the default setting, all iconic data computed in the `Run` mode (see page 32) is displayed in the current graphics window. You may want to suppress this automatic output, e.g., because it slows down the performance

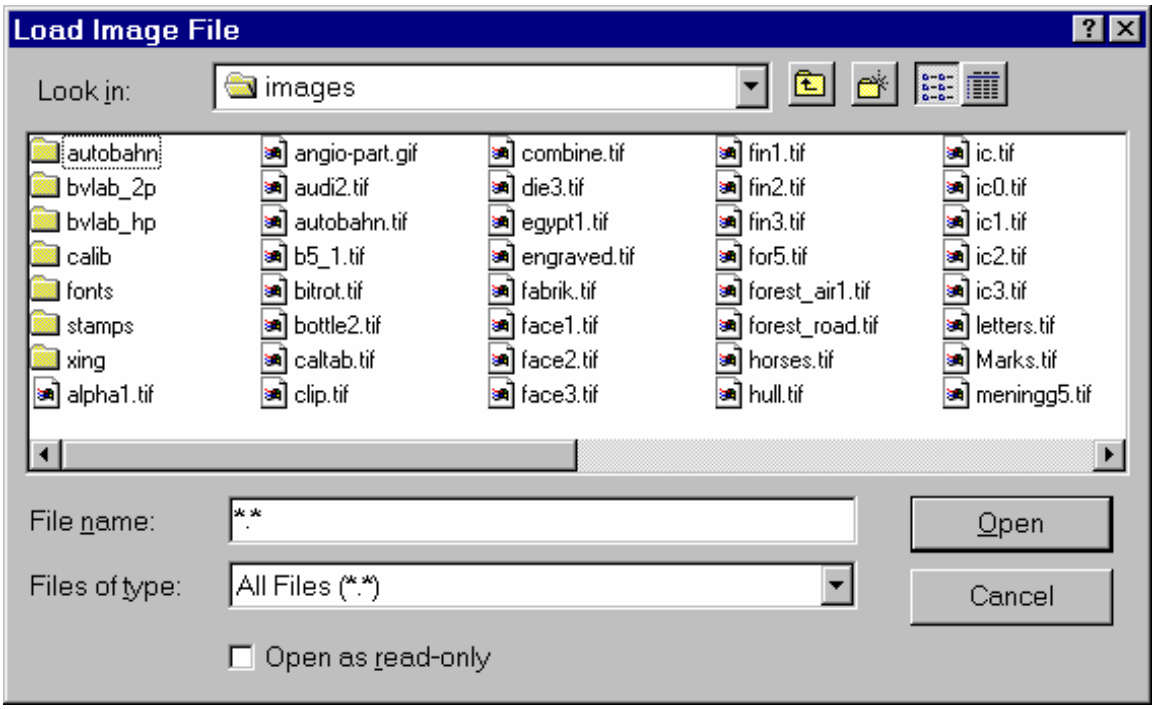


Figure 4.6: *The dialog window to load an image.*

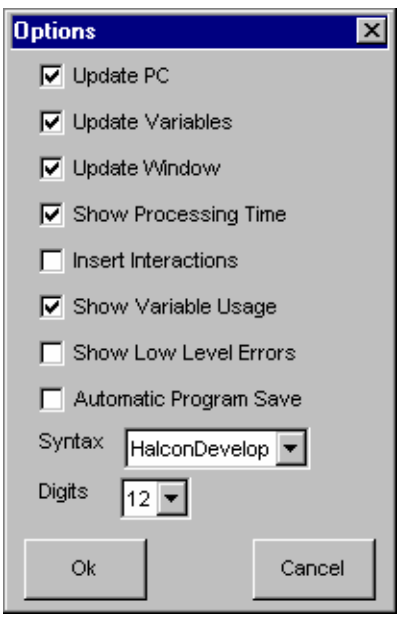


Figure 4.7: *The options window.*

time. If the output is suppressed you have the same behavior as exported C++ and Visual Basic code, where automatic output of data is not supported.

– `Show Processing Time`
This checkbox indicates whether the required runtime of the last operator should be displayed after the execution has stopped. It is a measurement of the needed time

for the current operator (without output and other management tasks of HDevelop). Along with the required runtime, the name of the operator is displayed in the status bar at the bottom of the main window. Please note that the displayed runtime can vary considerably. This is caused by the inaccuracy of the operating system's time measurement procedure.

- `Insert Interactions`
  Sometimes it is very helpful to record user interactions as a sequence of operators in the program. To do so, you select this option. From now on interactions are inserted as a program line in the program window. For example, select the graphic color red by choosing the appropriate menu inserts the program line

  ```
  dev_set_color('red')
  ```

  into the program window.
- `Show Variable Usage`
  If you activate a variable (by single-clicking on it) all lines in the program that contain the variable are marked on the left with a black frame. This works with iconic and control variables. You can activate one iconic and one control variable simultaneously. Each activated variable is marked by a black background for the name in the variable window.
- `Show Low Level Errors`
  Low level errors of HALCON are normally invisible for the user because they are transferred into more comprehensive error messages or simply ignored. Activating this item generates a message box each time a low level error occurs.
- `Automatic Program Save`
  If you activate this option, the program is automatically saved before each execution of the program, i.e., before a Run or Step operation. The file name the program is saved to is the file name of the current program. Therefore, if you create a new program you have to select `File` ▷ `Save as...` manually first to give the program a file name.
- `Syntax`
  Using a combo box, you may specify the output mode inside the program window. Depending on the mode, each HALCON or HDevelop operator is shown in a specific syntax like `HalconDevelop` (default syntax), `TRIAS`, `C` or `Pascal`.
- `Digits`
  With this checkbox, you can control how many digits of floating point numbers are displayed in the Variable window. The selected number is the total number of digits displayed. Therefore, if you have selected four digits, the result of the following assignment

  ```
  assign (4*atan(1), PI)
  ```

  is displayed as 3.142. Note that the changes do not take effect until the values of the variables are actually updated by running the program, i.e., the the variables are not redisplayed automatically.

Before continuing your HDevelop session, you have to close the option window by pressing the button `Ok` or by cancelling the action. If `Insert Interactions` is activated, the

changes applied inside the dialog will result in automatic operator insertion *after* pressing `OK`.

- Menu item `File` ▷ `Modules...` opens a window, in which the HALCON modules used by the current program are displayed (see figure 4.8). This window allows you to get an estimate of how many modules your application would need in a runtime license. Only calls to the HALCON library are taken into account for the computation of the modules, and not HDevelop control structures like `assign` or `ifelse`, or HDevelop operators like `dev_open_window` or `dev_set_color`. Therefore, when you export your program to C++ or Visual Basic, the actual number of modules required may be higher than the modules displayed in the Module window, depending on how many operators you add to the program, e.g., for visualization purposes.

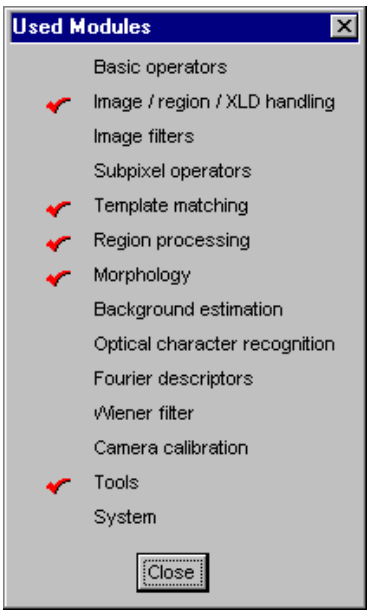


Figure 4.8: *The modules window.*

- Menu item `File` ▷ `Quit` terminates HDevelop without saving the current program.

Menu item: `Edit`

In this menu item you find all necessary functions to modify a HDevelop program in the *program window* (see chapter 4.4). You have the following items to choose from:

- You may undo your previous activities by clicking `Edit` ▷ `Undo`. For example, by clicking three times this item you cancel the last three user actions.

- You may use the items `Edit` ▷ `Cut`, `Edit` ▷ `Copy`, and `Edit` ▷ `Paste` for changing the program contents. First you have to select the part of the program (at least one program line) that has to be changed (use the left mouse button). Then you may delete this part by clicking the item `Cut` (keyboard shortcut `<Ctrl> Z`). The deleted program part is stored in an internal buffer. Thus, by using the item `Paste` (keyboard shortcut `<Ctrl> V`) the buffer remains unchanged.

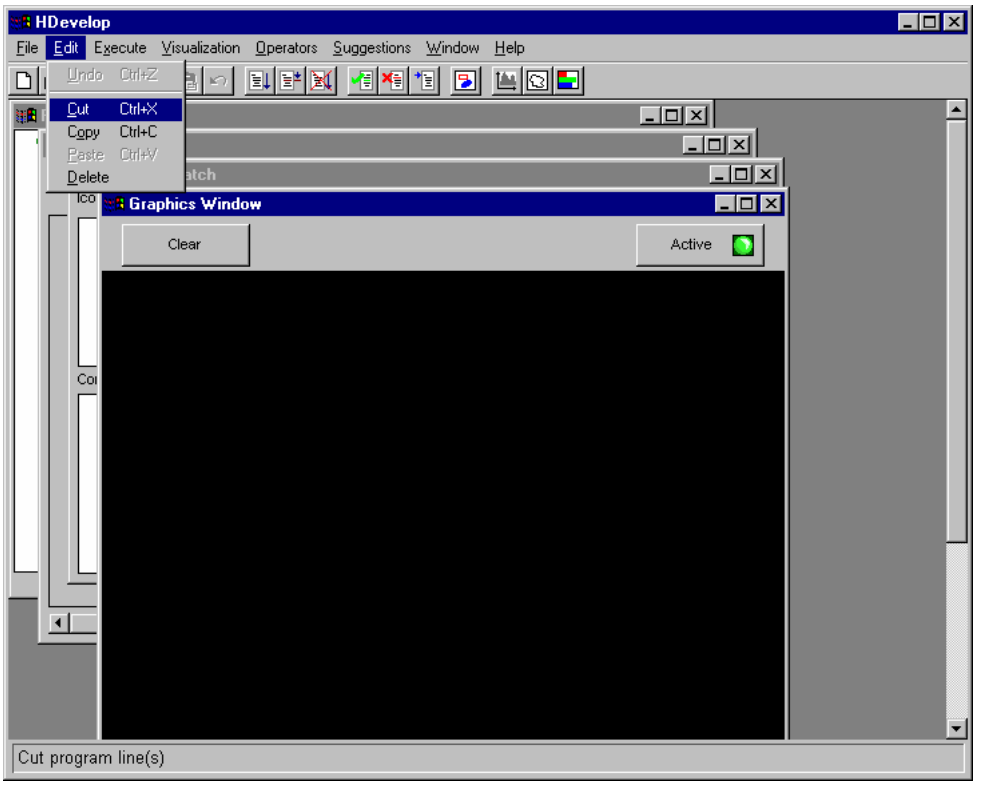


Figure 4.9: *Main window's menu item* `Edit`.

- By clicking `Copy` (keyboard shortcut `<Ctrl> C`) you store the selected program lines directly in an internal buffer. To insert this buffer in the HDevelop program you place your mouse pointer at the desired position and then click item `Paste`. Menu item `Edit` ▷ `Delete` deletes all selected program lines without storing them in an internal buffer. The only way to get the deleted lines back in your program is to use the item `Undo`.

Menu item: `Execute`

In this menu item you find all necessary functions to execute a HDevelop program in the *program window* (see chapter 4.4). You have the following items to choose from:

- By selecting `Execute` ▷ `Run` (keyboard shortcut `F5`), HDevelop executes your program depending on the `PC`'s position in the program window. The `PC` indicates the next program line to execute. All following program lines are going to be performed until the program end. Notice, that a breakpoint may interrupt the run of your program.

  During the execution of operators the following special behaviour occurs:

  – Although the mouse pointer indicates that HDevelop is not ready to handle user input (clock shape of the mouse pointer) you may initiate limited transactions. For example if you click variables, they will be visualized; you may modify output parameters for graphics windows; you may even modify the program. Note that HDevelop may be slow to react to your actions while the program is running. This is caused by the fact that HALCON reacts to user input only *between* calls to operators.

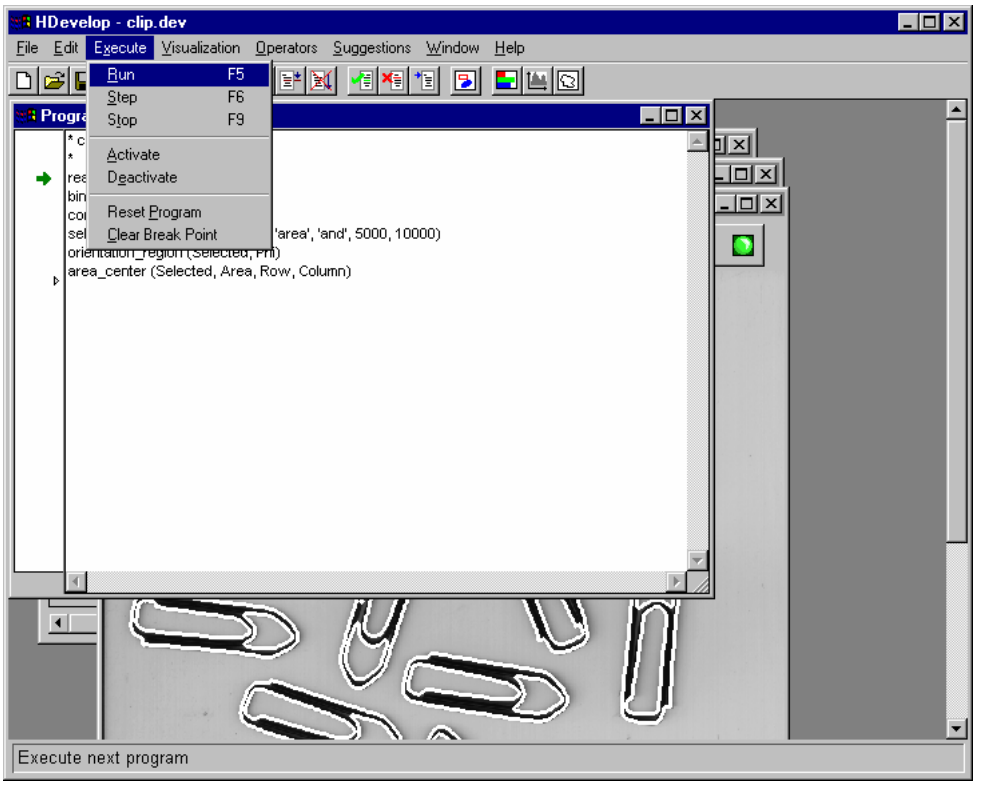


Figure 4.10: *Main window's menu item* `Execute`.

– A variable window update during runtime will only be performed if it has not been suppressed (see page 36). In any case, the values of all variables are shown in the variable window after the execution's termination.

While the program is running, the menus `Execute` ▷ `Run` and `Execute` ▷ `Step` (and the toolbar buttons `Run` and `Step`) are grayed out, i.e., you cannot execute them. You have the following possibilities to stop your HDevelop program:

(i) The program runs until the last operator in this program has been called. The `PC` is positioned behind this operator. This is the usual way to terminate a program.

(ii) The menu `Execute` ▷ `Stop` (or the toolbar button `Stop`) has been pressed.

(iii) A breakpoint has been set (see chapter 4.4). In this case the last operator that will be executed is situated *before* the breakpoint.

(iv) Menu item `File` ▷ `Quit` has been executed (see page 31).

(v) A runtime error occurred. An input variable without a value or values outside a valid range might be typical reasons. In this case the `PC` remains in the line of the erroneous operator.

- Selecting `Execute` ▷ `Step` (keyboard shortcut `F6`) enables you to execute a program (even if it is not complete) step by step. HDevelop executes the operator directly to the right of the green arrow, which is used as the program counter (`PC`) (see chapter 4.4). The mouse pointer changes its shape to a clock. This indicates that HDevelop is active and

not available for any user input. After the operator has terminated, all computed values are assigned to their respective variables that are named in the ouput parameter positions. Their graphical or textual representation in the variable window is also replaced. If iconic data has been computed, you will see its presentation in the current graphics window. In the status bar of the program window the operator runtime is indicated (if the time measurement has not been deactivated).

The PC is set before the next operator to execute. If the operators are specified in a sequential order, this is the textual successor. In case of control statements (e.g., `if ... endif` or `for ... endfor`), the PC is set *on* the end marker (e.g., `endif` or `endfor`) after the execution of the last operator inside the statement's body. After `endfor` and `endwhile` the PC is always set on the beginning of the loop. If a condition (as `if` or `while`) evaluates to `FALSE`, the PC is set *behind* the end marker.

Suggestions in the menu `Suggestions` are determined for the recently executed operator. Finally the mouse pointer's shape switches to the arrow shape and HDevelop is available for further transactions. Any user input which has been made during execution is handled now.

- You may terminate the execution of a program (mode `Run`) by selecting `Execute ▷ Stop` (keyboard shortcut `F9`). If you do so, HDevelop continues processing until the current operator has completed its computations. This may take a long time if the operator is taking a lot of time to execute. There is no way of interrupting a HALCON operator. After interrupting a program you may continue it by selecting `Execute ▷ Run` and `Execute ▷ Step`. You may even edit the program before restarting it (e.g., by parameter modification, by exchanging operators with alternatives, or by inserting additional operators).

- It is often useful for testing purposes to prevent some lines of the program from being executed. This can be done by selecting the appropriate lines in the program window and calling `Execute ▷ Deactivate` from the menu. With this, an asterisk is placed on the beginning of the selected lines, and hence appear as comments in the program window. They have no influence on the program during runtime. The deactivated lines are still part of the program, i.e., they are stored like all other lines in a file and their variables are still needed like all other variables. To reverse this action you may press item `Edit ▷ Activate`.

  Note that you can insert a comment into your program by using the operator `comment`.

- With the menu item `Execute ▷ Reset Program` you can reset the variables of the current program to their initial states, i.e., all variables have undefined values. Furthermore, the break point is cleared and the program counter is set to the first executable line of the program. This menu item is useful for testing and debbuging of programs.

- The menu item `Execute ▷ Clear Break Point` is used to clear the break point. This is often useful when the current program is long and you want to avoid having to scroll the program window to locate the break point.

Menu item: `Visualization`

All items which can be selected are shown in figure 4.11:

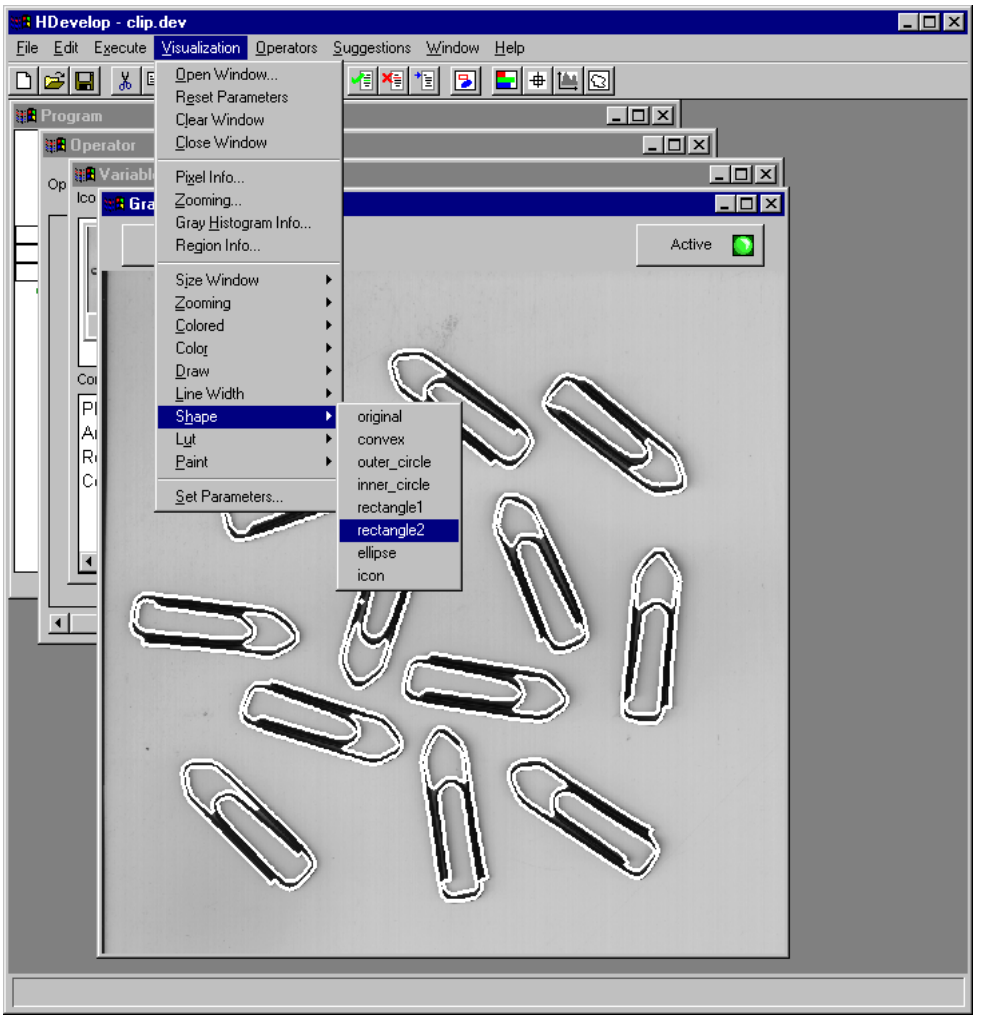


Figure 4.11: *Submenu* `Shape` *of menu* `Visualization`.

With this you are able to open or to close graphics windows and to clear their displays. Furthermore, you may specify their output behavior during runtime.

- `Visualization` ▷ `Open Window...`
  By using this menu item, you open additional[3] graphics windows.

  For this, a dialog window pops up (see figure 4.12). Here you may specify some graphics windows attributes. The position, size and background color of the new graphics window can be specified. For example, it is more convenient to have a white background while building graphics for slides or reports (see the HALCON operator `dump_window`). If the window height and width are set to -1, the window obtains the same size as the largest image in the current session. A position value of -1 specifies that the window position is determined by the window manager (UNIX). If you have not already created an image, the size 512×512 is used. The handling of graphics windows is described in chapter 4.7 at page 68.

[3]Normally upon starting, HDevelop automatically opens one graphics window.

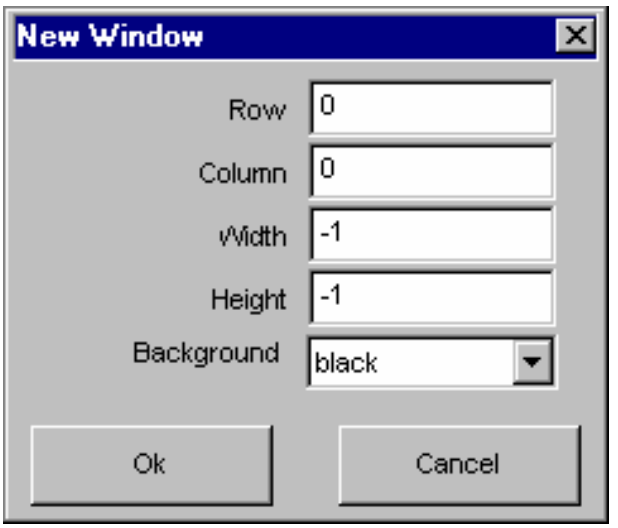


Figure 4.12: *Dialog window of menu item* `Visualization ▷ Open Window...`

- `Visualization ▷ Reset Parameters`
  Here, the display parameters of all graphics windows are set to their initial state (the state after starting the program). The only exception is the history of previously displayed objects and the size of each window. To clear the history you can use `Visualization ▷ Clear Window`, to set the size you can use `Visualization ▷ Size Window`.

- `Visualization ▷ Close Window`
  Selecting this item closes the active graphics window.

- `Visualization ▷ Clear Window`
  The active graphics window is cleared. The history (previously displayed objects) of the window is also removed.

- `Visualization ▷ Pixel Info...`
  Here you can open an inspection display. This is used for interactive examination of gray values of images. Apart from this, the size, pixel type, and the number of channels are displayed.

  The upper part of the dialog contains a gauge to display the gray value graphically. The range goes from 0 (left) to 255 (right). Normally the gray value of the first channel is displayed with a black bar. For color images in RGB-space (three channels with red, green, and blue values) three colored bars are used. If the gray value is below 1 the gauge is white (background). If the value is above 255 the gauge is black or colored for rgb images.

  Below the gauge, the gray values are displayed as numbers. If more than three channels are present only the gray value of the first channel is displayed.

  Below the gray values the coordinates of the mouse position is displayed. Below these, the size, pixel type, and the number of channels of the selected image are shown.

- `Visualization ▷ Zooming...`
  With this menu item, a tool for realtime viewing of zoomed parts of an image object is

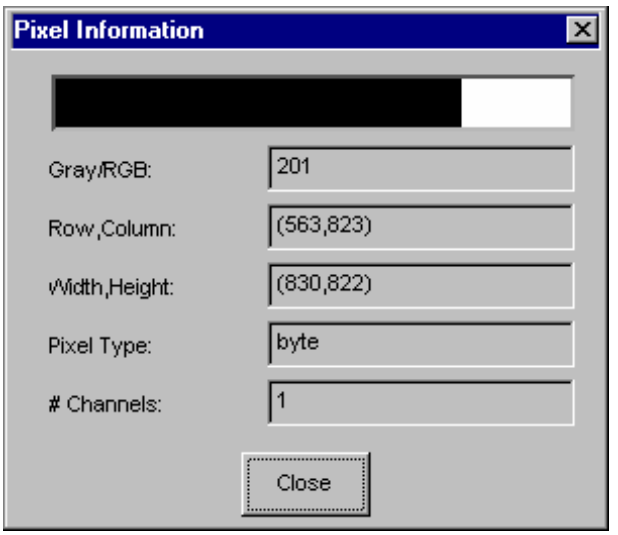


Figure 4.13: *Online gray value inspection and basic image features.*

opened. Figure 4.14 shows the layout of the realtime zooming window.

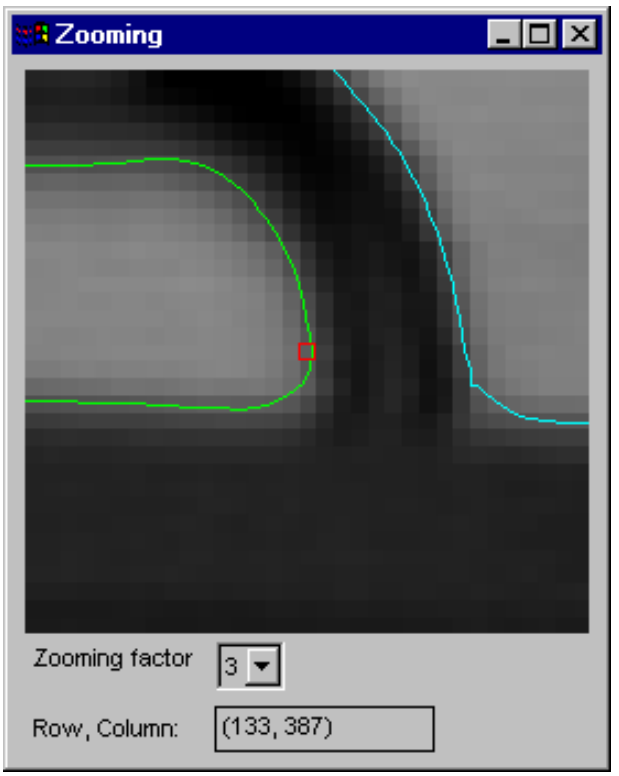


Figure 4.14: *Realtime zooming.*

The upper part of the tool contains a window of fixed size 256×256, in which the part of the graphics window, over which the mouse pointer is located, is displayed enlarged. In the zooming window, this pixel is marked by a red square; its coordinates are displayed at the bottom of the zooming window. The factor, by which the enlargement is done can

be adjusted with the combo box `Zooming factor`. A zooming factor of 0 corresponds to displaying the contents of the graphics window in to normal resolution, i.e., one pixel in the image object corresponds to one pixel in the zooming window. Increasing the zooming factor by 1 roughly increases the enlargement by a factor of 2.[4] You can select a particular pixel by single-clicking on it with the left mouse button. The zooming tool stores this position internally, and will redisplay the thus selected part of the image object when you leave the graphics window. This enables you to have a meaningful display in the zooming tool whenever you want to do actions outside of the graphics window.

- `Visualization` ▷ `Gray Histogram Info...`
  This menu item opens a sophisticated tool for the inspection of gray value histograms, that can also be used to select thresholds interactively and to set the range of displayed gray values dynamically. Figure 4.15 shows the layout of the gray histogram inspection window.

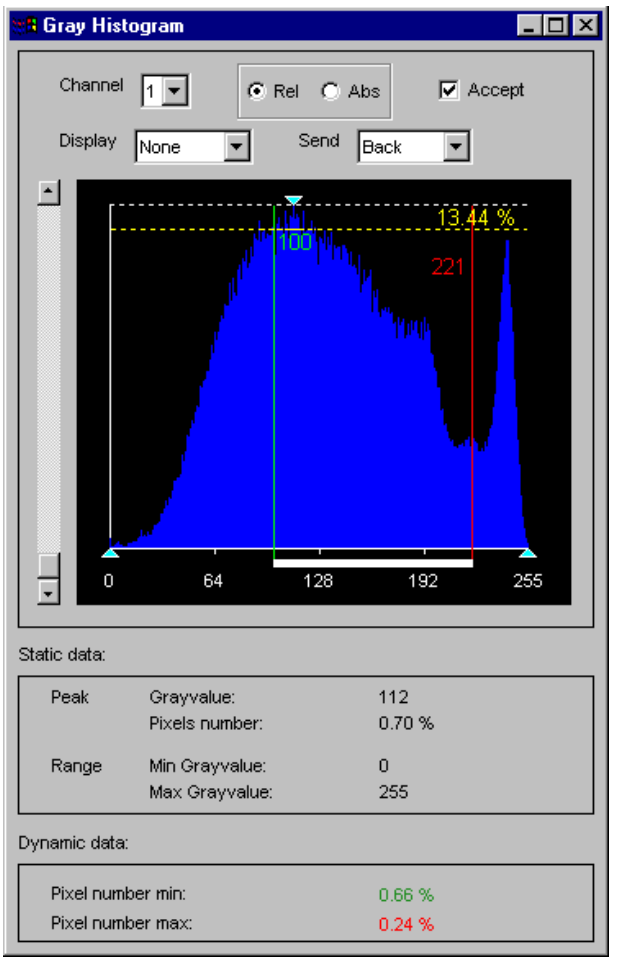


Figure 4.15: *Online gray histogram inspection.*

[4] Yes, only roughly by a factor of 2, since the image is scaled such that the red square that indicates the mouse pointer position is located in the middle of the zooming window. Therefore, the zoom factor is adjusted to display one pixel more than the power of 2 indicated by the zooming factor. The width and height of the zoomed part of the image hence are $2^{8-f}+1$, where f is the zooming factor.

When opening the tool, the histogram of the image shown in the currently active graphics window is displayed. When the tool is already open, four modes of sending new image data to the tool are available. The simplest mode is to display an image in the active graphics window. Whenever you do so, the histogram of this image is computed and drawn, and the tool records the graphics window from which the image was sent. Another simple method to send new data to the tool is to single-click into an image that is displayed in a graphics window. The third mode can be used whenever image data is displayed overlaid with region data in a graphics window (the graphics window does not need to be active for this). In this mode, you can click into any of the segmented regions, and the histogram of the image within that region will be computed and shown. If you click into a part of the image that is not contained in any of the overlaid regions, the histogram of the entire image will be displayed. In the fourth mode, the same mechanism is used for regions that have gray value information, e.g., image objects created by `reduce_domain` or `add_channels`. Here, the histogram of the image object you click into will be displayed. Finally, when the graphics window the image was sent from is cleared, the histogram is not reset.

When a multi-channel image, e.g., a RGB color image, is sent to the tool, by default the histogram of the first channel is displayed. The combo box `Channel` lets you select the channel from which to compute the histogram.

The radio box in the upper center of the tool lets you select whether to display the histogram with relative or absolute frequencies. When you select `Rel`, the frequencies of individual gray values are displayed as percentages; when you select `Abs`, they are displayed as the actual number of pixels having a particular gray value. See the operator `gray_histo` in the HALCON Reference Manuals for details.

Sometimes it is desirable to suppress the updating of the histogram when new image data is available, e.g., if you want to select thresholds for a gradient image, but want to visualize the original image along with the segmentation (see below). To do so, you can deselect the checkbox `Accept`.

The main part of the tool is the area, in which the histogram of the image is displayed in blue. This area contains static parts and parts that can be interactively manipulated. The first static part is the horizontal coordinate axis, which displays the gray values in the image. For byte images, this range is always $0 \ldots 255$. For all other image types, e.g., real images, the horizontal axis runs from the minimum to the maximum gray value of the image, and the labeling of the axis is changed accordingly. To the left of the display, the vertical axis representing the frequency of the gray values is drawn in white. The final static parts of the display are three cyan arrows. The two upward pointing arrows denote the maximum and minimum grayvalue of the image. The downward pointing arrow denotes the gray value that occurs most frequently, i.e., the peak of the histogram. These data are displayed in textual form within the `Static data` area of the display.

The dynamic parts of the histogram area are the three colored lines, which can be manipulated. The dashed horizontal yellow line can be dragged vertically. The label on this line indicates the frequency of gray values above this line. The vertical green and red lines denote the minimum and maximum selected gray value of the histogram, respectively. The selected range is drawn as a white bar below the horizontal gray value axis. The gray values on which the two vertical lines lie are displayed next to the lines in the same color.

The frequency of the respective gray values is displayed within the `Dynamic data` area of the display.

The selected range of gray values can be used for two major purposes. First, when the combo box `Display` is set to `Threshold`, the image, from which the histogram was computed, is segmented with a `threshold` operation with the selected minimum and maximum gray value. Depending in the setting of the combo box `Send`, the segmentation result is either displayed in the graphics window, from which the image was originally sent (`Send = Back`), or to the active graphics window (`Send = To Active`). Second, if `Display` is set to `Scale`, the gray values of the image are scaled such that the gray value 0 of the scaled image corresponds to the minimum selected gray value and the gray value 255 to the maximum selected gray value. Again, the combo box `Send` determines the graphics window, in which the result is displayed. This mode is useful to interactively set a "window" of gray values that should be displayed with a large dynamic range.

If you want to select threshold parameters for a single image, display the image in the active graphics window and open the histogram tool. For optimum visualization of the segmentation results, it is best to set the visualization color to a color different from black or white (see `Visualization` ▷ `Color` below). Now set `Display` to `Threshold` and interactively drag the two vertical bars until you achieve the desired segmentation result. The parameters of the threshold operation can now be read off the two vertical lines.

If you want to select threshold parameters for an image, which is derived from another image, but want to display the segmentation on the original image, e.g., if you want to select thresholds for a gradient image, two different possibilities exist. First, you can display the derived image, open the histogram tool, deselect `Accept`, display the original image, and then select the appropriate thresholds. This way, only one window is needed for the visualization. For the second possibility you can display the derived image in one window, make another window active or open a new window, display the original image there, make the first window active again, open the histogram tool, make the second window active again, set `Send` to `To Active`, and select your thresholds. Although in this case it is not necessary to deselect `Accept`, it is advantageous to do so, because this prevents the histogram from being updated if you click into a graphics window accidentaly.

- `Visualization` ▷ `Region Info...`
  This menu item opens a tool for the convenient inspection of shape and gray value features of individual regions. It can, for instance, be used to determine thresholds for operators that select regions based on these features, e.g., `select_shape` or `select_gray`. Figure 4.16 shows the layout of the region feature inspection window.

  The strategy to determine the data from which to compute the features is very similar to that of the gray histogram inspection window. You can display an image or region by double-clicking on it in the variable window or you can select a region or an image which is already displayed, by single-clicking it. If you display or click into an image, only the gray value features of the entire image will be calculated. If you click into a region that is not underlaid with an image, only the shape features of this region will be displayed. If you click into a region that is underlaid with an image or into a region that has gray value information (e.g., from `reduce_domain` or `add_channels`), both the shape and gray value features of that region will be displayed. Finally, if you have overlaid an image with a region, but click into a part of the image that is outside the region, only the gray value

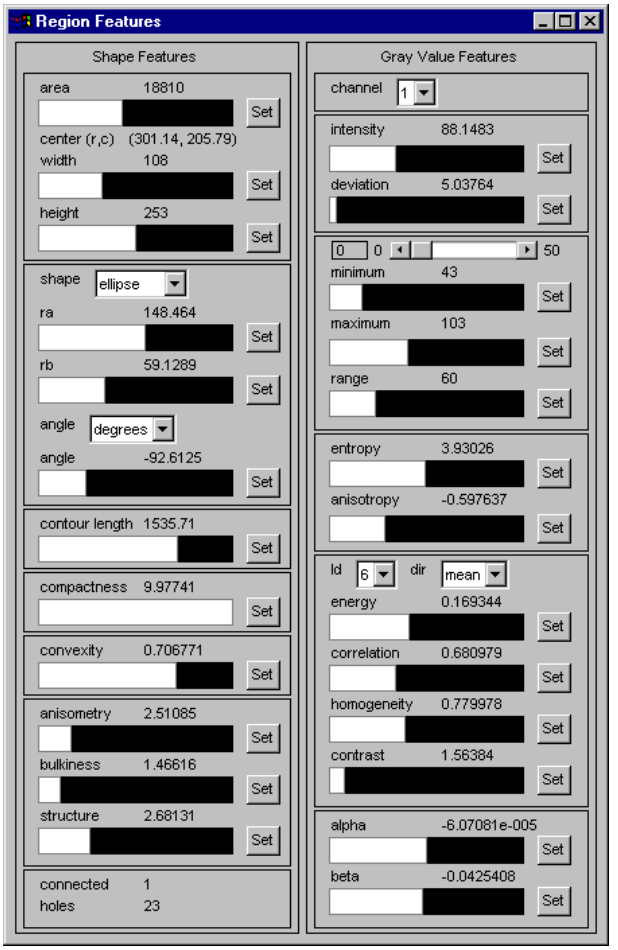


Figure 4.16: *Online region feature inspection.*

features of the entire image will be calculated.

Analogously to the gray histogram inspection window, the gray value features of a multi-channel image are calculated from the first channel by default. You can use the combo box `Channel` to select the desired channel.

The shape features on the left side of the region inspection window are grouped into seven categories, which correspond roughly to individual HALCON shape feature operators. The topmost of the displays shows the most basic region features, namely the area and center of gravity of the region (see `area_center` in the Reference Manual) and the width and height of the smallest axis-parallel rectangle of the region. The latter is computed from the output of the operator `smallest_rectangle1`.

The second display contains information about the orientation (`angle`) and size of the region along the two principal directions (ra and rb) of the region. With the combo box `shape`, you can select by what means the size is computed. If you select `ellipse`, the size is computed with the operator `elliptic_axis`. This means that the parameters ra and rb are the major and minor axis of an ellipse that has the same moments as the selected region. Note that this ellipse need not enclose the region. If you set `shape` to `rectangle`, the size is computed with the operator `smallest_rectangle2`. This means, that ra and

rb are half the width and height of the smallest rectangle with arbitrary orientation that completely contains the selected region. The orientation of the region is computed in both cases with the operator `orientation_region` to get the full range of 360° for the angle. You can select whether to display the angle in `degrees` or `radians` with the corresponding combo box.

The next three displays show simpler shape features of the selected region. The first of these displays shows the contour lenght of the region, i.e., the euclidean length of its boundary (see `contlength`). The second one shows the compactness of the region, i.e., the ratio of the contour length of the region and the circumference of a circle with the same area as the region (see `compactness`). The compactness of a region is always larger than 1. The compacter the region, the closer the value of the compactness is to 1. The third display shows the convexity of the selected region, i.e., the ratio of the area of the region and the area of the convex hull of the region (see `convexity`). The convexity of a region is always smaller than 1. Only convex regions will reach the optimum convexity of 1.

The last but one display shows shape features derived from the ellipse parameters of the selected region, which are calculated with `eccentricity`. The anisometry of the region is the ratio of the major and minor axis of the ellipse (i.e., the ratio of ra and rb in the second display if you set `shape` to `ellipse`). This feature measures how elongated the region is. Its value is always larger than 1, with isometric regions having a value of 1. The definition of the more complex features bulkiness and structure factor (abbreviated as structure in the display) can be obtained from the HALCON Reference Manual.

The final shape feature display shows the connected components and number of holes of the selected region, as computed by `connect_and_holes`.

The gray value features are grouped into five displays on the right side of the region inspection window. Again, they correspond roughly to individual HALCONoperators. The first display shows the mean gray value intensity and the corresponding standard deviation of the selected region. These are computed with the operator `intensity`.

The second display shows the output of the operator `min_max_gray`. This operator computes the distribution (hestogram) of gray values in the image and returns the gray values corresponding to an upper and lower percentile of the distribution. This percentile can be selected with the slider at the to of the display. For a percentile of 0 (the default), the minimum and maximum gray values of the region are returned. The display also shows the range of gray values in the region, i.e., the difference between the maximum and minimum gray values.

In the third display, the gray value entropy of the selected region is displayed (see `entropy_gray`. Again, this is a feature derived from the histogram of gray values in the region. The feature entropy measures whether the gray values are distributed equally within the region. This measure is always smaller than 8 (for byte images — the only suppurted image type for this operator). Only images with equally distributed gray values reach this maximum value. The feature anisotropy measures the symmetry of the distribution. Perfectly symmetric histograms will have an anisometry of -0.5.

The fourth display contains gray value features derived from the coocurrence matrix of the selected region are displayed (see `cooc_feature_image`. The combo box `ld` can be used to select the number of gray values to be distinguished (2^{ld}). The combo box `dir` selects

the direction in which the coocurrence matrix is computed. The resulting features — energy, correlation, heomgeneity, and contrast — have self-explanatory names. A detailed description can be found in the reference of the operator `cooc_feature_matrix`.

The final display contains the output of the operator `moments_gray_plane`. This are the angles of the normal vector of a plane fit through the gray values of the selected region.

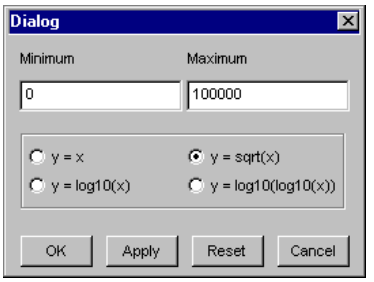


Figure 4.17: *Configuration dialog for single region features.*

Each of the gauges corresponding to a display can be configured to set the minimum and maximum values for each gauge. Furthermore, the scaling function of the gauge can be determined. This can be used to make the relation of the features of different regions more intuitive. For example, features that depend on the area of the region are more intuitively grasped when the scaling is set to `sqrt(x)`. The configuration dialog is the same for all gauges, and is shown in figure 4.17. It can be brought up by pressing the arrow button next to each gauge.

- `Visualization` ▷ `Size Window`
  There are convenient methods to change the size of the active graphics window dependent on the size of the previously displayed image. Using the submenu `Original`, the window is set to the same size as the most recently displayed image, that means, for each pixel of the image one pixel on the screen is used for displaying. Similar to this, you can select `Original half` or `Original quarter` to make the window half or a quarter as big as the displayed image. The submenus `Half` and `Double` change the size of the graphics window to half and double its current size, respectively, independent of the size of the previously displayed image. You can combine `Double` with `Original`. The submenu `Aspect` changes the aspect ratio of the graphics window, so that pixels are displayed as squares on the screen. For this operation, again the size of the previously displayed image is used.

- `Visualization` ▷ `Zooming`
  This is a convenient menu for manipulation of the zooming mode. The submenu `Reset` switches zooming off, i.e., an image will be displayed so that it fills the graphics window completely. The submenus `Zoom In` and `Zoom Out` apply a zooming “in” and “out” to the image or region by a factor of two. Finally, there are two interactive modes to control zooming: `Draw Rectangle` allows the specification of a rectangular part of the window to be zoomed while `Draw Center` allows the definition of a pixel coordinate that should be at the center of the window (e.g., for a successive `Zoom In`).

  For more information see the menu `Visualization` ▷ `Set Parameters...` ▷ `Zoom`.

- `Visualization ▷ Colored`
  This is an easy way to display multiple regions or XLDs. Each region is displayed in a different color, where the number of different colors is specified in the submenu. You can choose between 3, 6 and 12 colors. If all regions are displayed with one color, you have to use the operator `connection` beforehand. You can check this also with the operator `count_obj`.

- `Visualization ▷ Color`
  This item enables color specification to display segmentation results (regions and XLD), text (`write_string`) and general line drawings (e.g., 3D plots, contour lines, and bar charts). The number of colors which are available in the submenu depends on the graphics display (i.e., the number of bits used for displaying). After selecting a color, the previously displayed region or XLD object will be redisplayed with this color. The default color is white.

- `Visualization ▷ Draw`
  Here you can select a visualization mode to display region data. It can either be *filled* (item `fill`) or the *borders* are displayed only (item `margin`). The border line thickness of the displayed regions is specified using the menu item `Line Width` (see figure 4.19).

- `Visualization ▷ Line Width`
  Here you determine the line width for painting XLDs, borders of regions or other types of lines. You can select between a wide range of widths using the submenu. This parameter is effective if `dev_set_draw` is set to mode `margin` only.

- `Visualization ▷ Shape`
  Here you specify the representation shape for *regions*. Thus you are able to display not only the region's original shape but also its enclosing rectangle or its enclosing circle.

- `Visualization ▷ Lut`
  This menu activates different look up tables, which can used to display gray images and color images in different intensities and colors. In the case of a true color display the image has to be redisplayed due to the missing support of a look-up-table in the graphics hardware. For color images only the gray look-up-tables can be used, which change each channel (separately) with the same table.

- `Visualization ▷ Paint`
  This menu defines the mode to display gray images. For more information see the menu item `Visualization ▷ Set Parameters...`.

- `Visualization ▷ Set Parameters...`
  By using this menu item, a dialog called `Visualization Parameters` is opened, which handles more complex parameter settings. Select one setting with your left mouse button and the window brings up the according parameter box. Each box contains different buttons, text fields, or check boxes to modify parameters.

  Each box has an `Update` button. If this button is pressed, every change of a parameter will immediately lead to a redisplay of the image, regions, or XLD in the graphics window. If the button is "off" the parameters become active for the next display of an object (double click on an icon or excecution of an operator). By default the update is

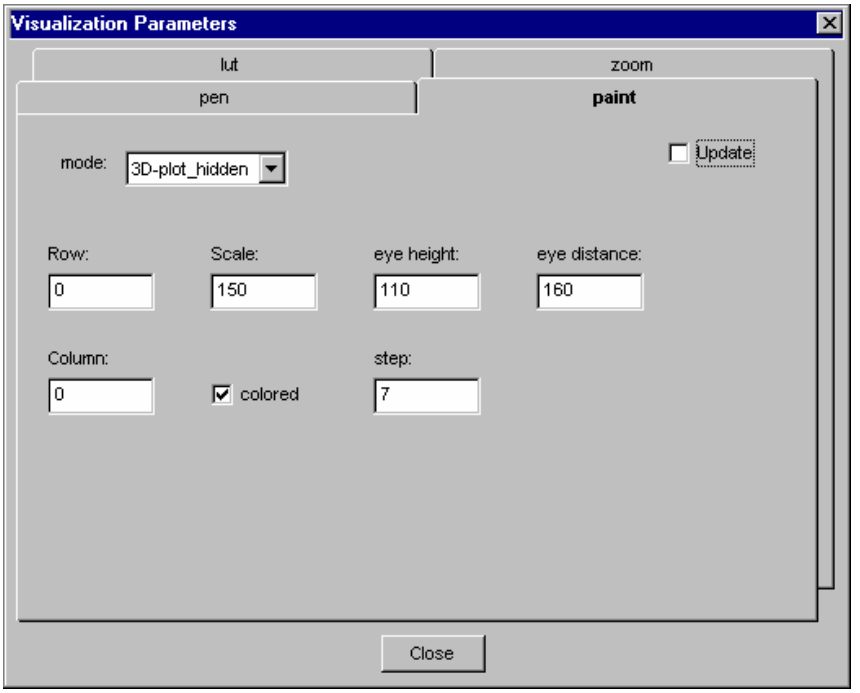


Figure 4.18: *Settings of parameter* `paint`.

deactivated for the boxes `Lut` and `Paint`.

You may specify the following parameter settings.

– `Visualization` ▷ `Set Parameters...` ▷ `Paint`
  Here you can select between several graphical presentations for images. Examples are `contourline` and `3D-plot`. In the default mode the image will be displayed as a picture (see figure 4.18).
  If you have chosen a presentation mode, the window displays all possible parameters you may modify. For example, after selecting the item `3D-plot` you have to specify the following parameters:
  - Step (the distance of plot lines in pixels),
  - Colored (use the gray value of a pixel to draw a line segment instead of one graphic color),
  - Eye height,
  - Eye distance (view point),
  - Scale (height of 3D plot),
  - Row and
  - Column (position of the center).

– `Visualization` ▷ `Set Parameters...` ▷ `Pen`
  Here the display modes for regions and XLDs are specified. You can select the color (single or multiple), the drawing mode (filled or border), the line width for border mode and the shape of the regions.

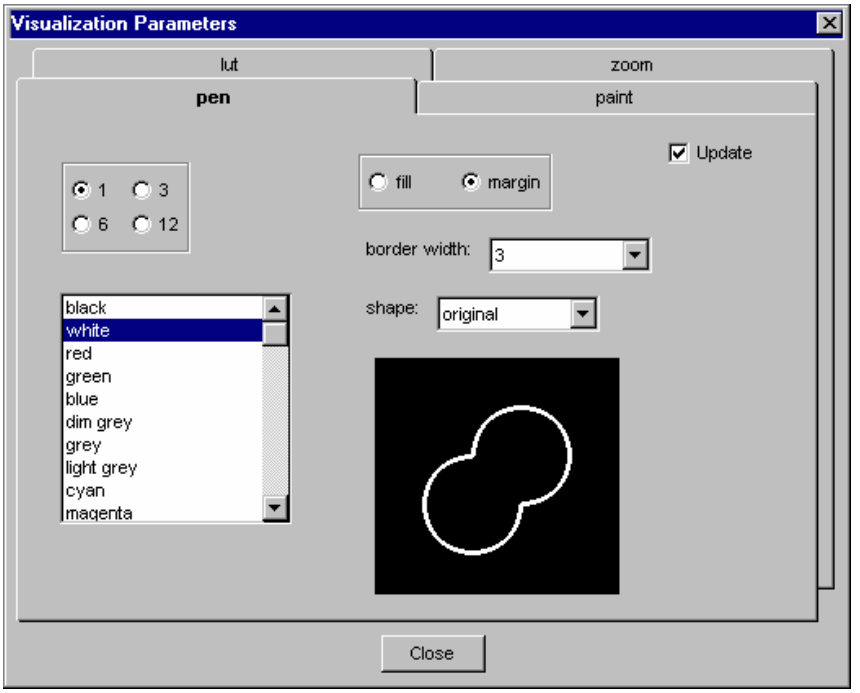


Figure 4.19: *Settings of parameter* `pen`.

You can select up to 12 colors by clicking the appropriate checkbox. They are used to emphasize the connectivity of different regions in the graphics window. If you choose a single color presentation you may specify this color by selecting it in the list box (see figure 4.19).

With the parameter shape (default is `original`) you may specify the presentation shape for regions. Thus you are able to display not only the region's original shape but also its enclosing rectangle or its enclosing circle, etc.

For regions the *draw mode* can be specified: Either it might be *filled* (item `fill`) or the *borders* are displayed (item `margin`) only. The border line thickness of the presented regions is specified with help of the menu item `border width`. This menu is *visible only* if the draw mode is set to `margin`, because in the draw mode `fill` this parameter has no effect.

– `Visualization` ▷ `Set Parameters...` ▷ `Zoom`
The menu item specifies which part of an image, region, XLD, or other graphic item is going to be displayed (see figure 4.20). The upper left four text fields specify the coordinate system. `left/upper` defines the pixel which will be displayed at the upper left corner of the window. `lower/right` defines the pixel which will be displayed at the lower right side of the window. By selecting the upper button `Interactive...` you specify a rectangular part in the graphics window interactively. For this, you press the left mouse button to indicate the rectangle's upper left corner. Hold the button and drag the mouse to the lower right corner's position. Release the button and correct the size by grabbing the borders or corners of the rectangle. By pressing the right mouse button inside your specified rectangle you display the objects inside the rectangle in the graphics window.

You also have the possibility to enter the coordinates of the desired clipping man-

ually. In order to do so you have to specify the coordinates of the upper left corner and the lower right corner in the respective text fields.

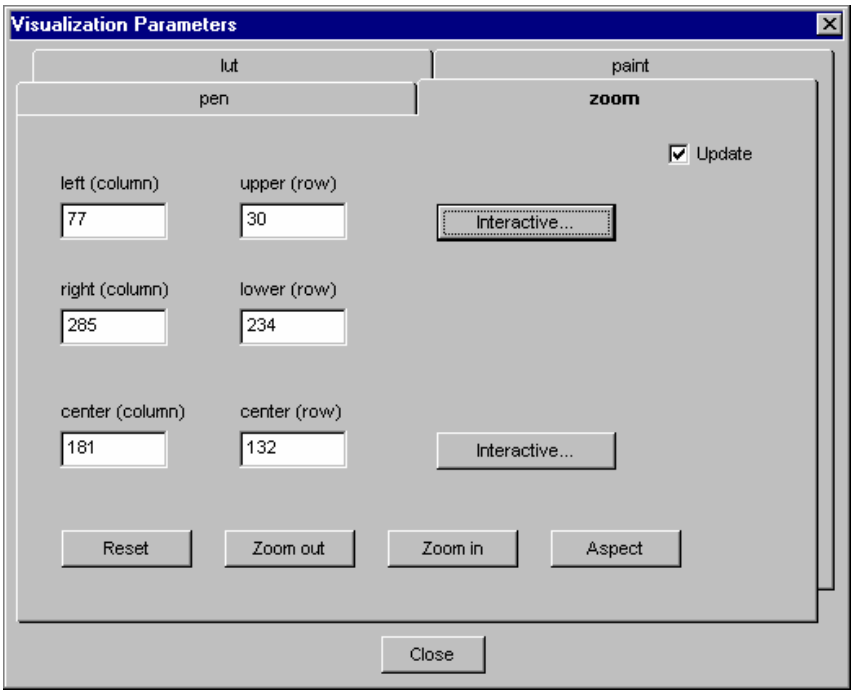


Figure 4.20: *Settings of parameter* `zoom`.

Below the coordinates of the rectangle you see its center. This center can also be specified interactively using the button `Interactive...`. Activating this mode, you first have to click with the left button into the active graphics window. Now you can correct this position by again pressing the left mouse button. To quit, press the right mouse button.

The buttons `Zoom out` and `Zoom in` activate a zooming with factor 2 or 0.5, respectively.

To get the image's full view back on your graphics window you simply click the checkbox `Reset`.

– `Visualization` ▷ `Set Parameters...` ▷ `Lut`
Using `lut` you are able to load different *look-up-tables* for visualization (see figure 4.21). With the help of a false color presentation you often get a better impression of the gray values of an image. In the case of a true color display the image has to be redisplayed due to the missing support of a look-up-table in the graphics hardware. For color images only the gray look-up-tables can be used, which change each channel (separately) with the same table.

Menu item: `Operators`

This menu item comprises all HALCON and HDevelop operators including the HDevelop control constructs. In the following you will see a description of all items to select.

□ *The item* `Control`

Here you may select control structures for the program to create. This involves execution of

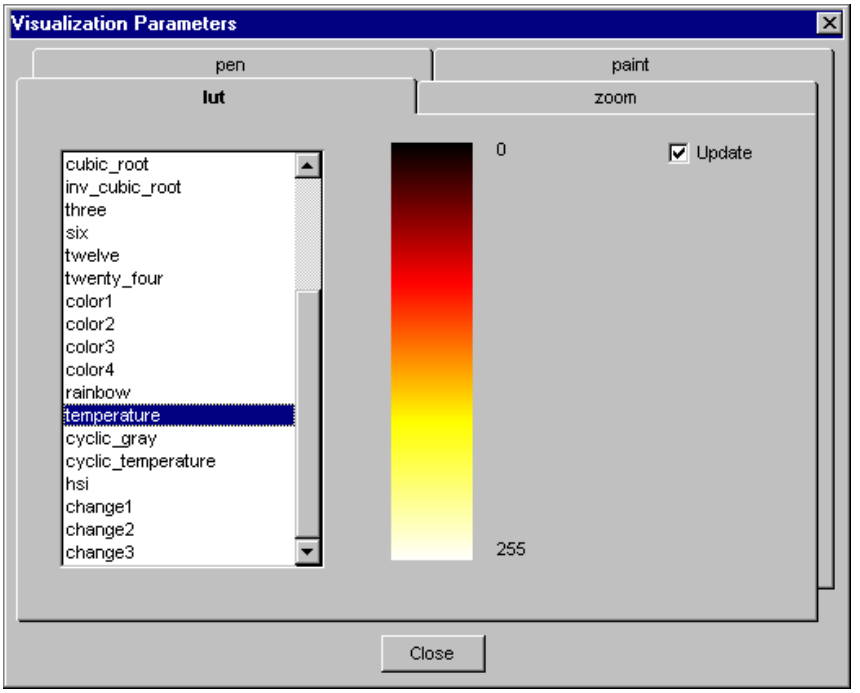


Figure 4.21: *Settings of parameter* `lut`.

a program segment (henceforth named body) dependent on a test (`if` and `ifelse`) and the repetition of a program segment (`for` and `while`). Furthermore, you may stop the program's execution at any position (`stop`) or terminate HDevelop (`exit`). The operators `assign` and `insert` do not influence the execution. They rather serve to specify values for control data (assignment). The operator `comment` is used to add a comment, that means any sequence of characters, to the program. The corresponding menu is shown in figure 4.22.
Selecting a menu item displays the corresponding control construct in the operator window, where you can set the necessary parameters. After specifying all parameters you may transfer the construct into your program. A direct execution for loops and conditions is not possible, in contrast to other HDevelop and HALCON operators, because you have to specify the loop's and condition's body first to obtain useful semantics. If necessary, you may execute the program after the input with `Step` or `Run`. The insertion cursor is positioned after the construct head to ensure the input of the construct's body occurs in the correct place. This body is indented to make the nesting level of the control constructs visible, and thus to help you in understanding the program structure (see figure 4.23). To get an idea how to use loops, you may look at the example session in section 3, and at the programs in chapters 7.3, 7.8 and 7.9. The semantics for loops and conditions are shown in chapter 5.7.
The operator `assign` serves as an assignment operator for control variables (numbers and strings). Analogously to "normal" operators the input is made in the operator window by specifying both "parameters" `Input` and `Result` (i.e., right and left side of the assignment). An instruction in C, e.g.,

```
x = y + z;
```

is declared inside the operator window as

```
assign(y + z,x)
```

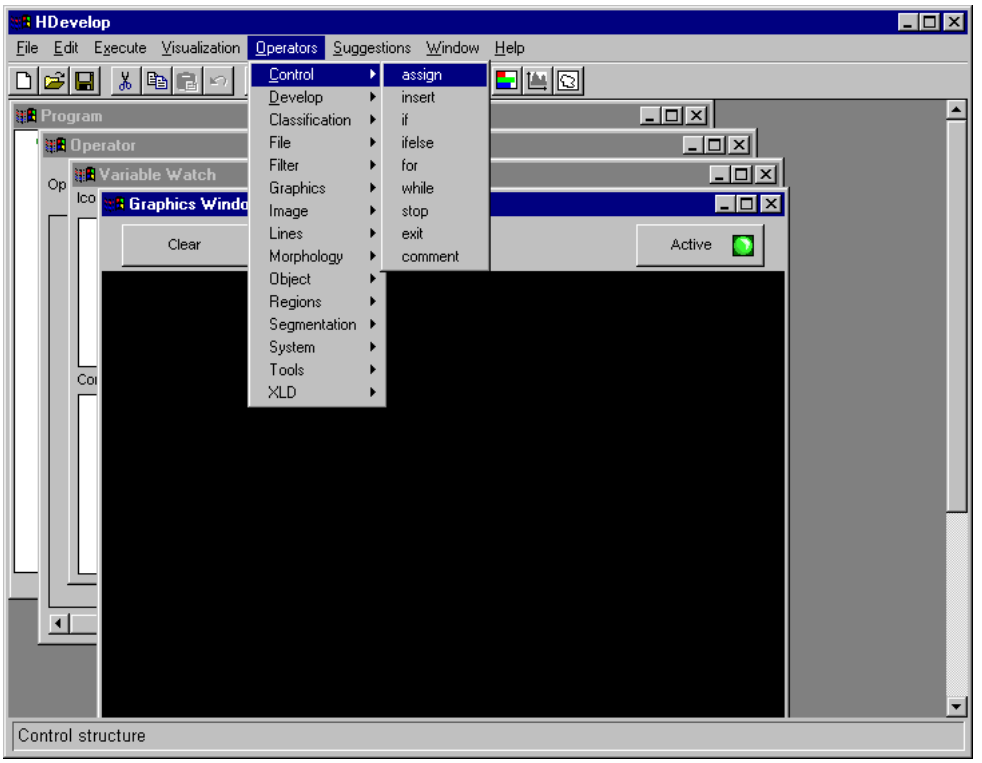


Figure 4.22: *Menu item* `Control`.

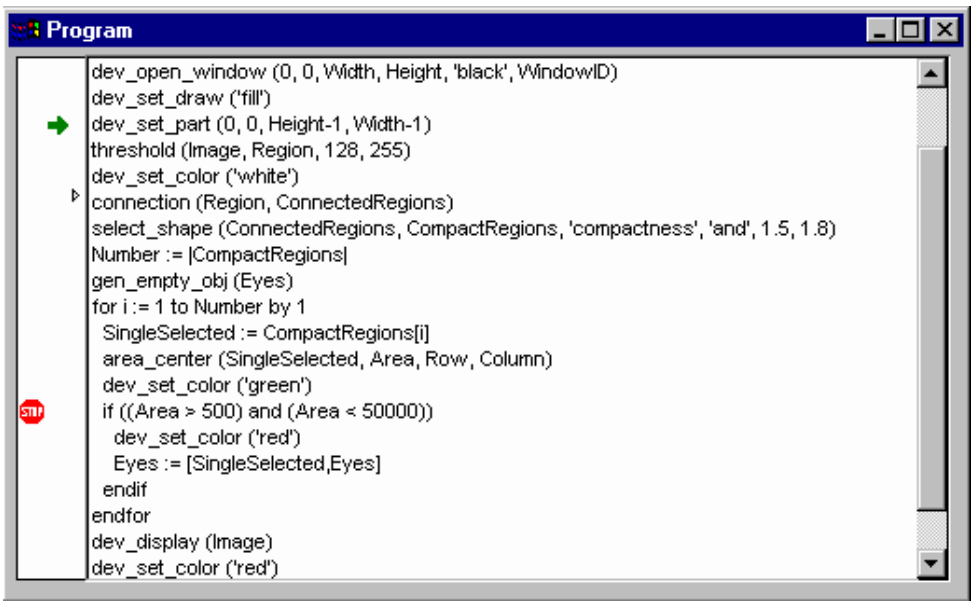


Figure 4.23: *Example for using a* `for` *loop.*

and displayed in the program window by

```
x := y + z
```

The operator `insert` implements the assignment of a single value (tuple of length 1) at a spec-

ified index position of a tuple. Thus an array assignment (here in C syntax)

```
a[i] = v;
```

is entered as

```
insert(a,v,i,a)
```

in the operator window, and is displayed as

```
a[i] := v
```

in the HDevelop program window.
The operators `stop` and `exit` are used to terminate the program. More precisely, `stop` *interrupts* an execution and `exit` *terminates* HDevelop. Having interrupted the execution you may continue the program by pressing `Step` or `Run`. This is useful, e.g., in demo programs to install defined positions for program interruption. Using `exit` is in particular recommended in combination with the startup file (see page 125). Thus you may terminate HDevelop if your program, which is included in the startup file, has been processed (see page 125).
The operator `comment` allows to add a line of text to the program. This text has no effect on the execution of the program. A comment may contain any sequence of characters.
□ *The item* `Develop`
This menu contains several operators that help to adapt the user interface. These operators offer the same functionality that you have using mouse interaction otherwise. They are used to configure the environment without (or with little) user interaction. Using these operators, the program performs actions similar to the setting of a color in the parameter window (see section 4.7), opening a window in the menu bar (see section 4.3.2) or iconifying the program window with the help of the window manager. All operators in this menu start with the prefix `dev_`. It has been introduced to have a distinction to the underlying basic HALCON operators (e.g., `dev_set_color` and `set_color`). You can find the complete listing in figure 4.24.
The effects of each operator are described as follows:

- `dev_open_window, dev_close_window, dev_clear_window`
  The operators `dev_open_window` and `dev_close_window` are used to open and to close a graphics window, respectively.During opening, the parameterization allows you to specify the window's size and position. The operator `dev_clear_window` clears the active window's content and its history. This corresponds to the usage of the button `Clear` in the graphics window. Please note that `dev_open_window` and `dev_close_window` are not supported for Visual Basic export because here one `HWindowXCtrl` is used.

- `dev_set_window_extents`
  With this operator, you can set the size and position of the active HDevelop graphics window.

- `dev_set_window`
  This operator activates the graphics window containing the given ID. This ID is an output parameter of `dev_open_window`. After the execution, the output is redirected to this window. This operator is not needed for exported code in C++, because here every window operation uses the ID as a parameter. The operator has no effect for exported code in Visual Basic.

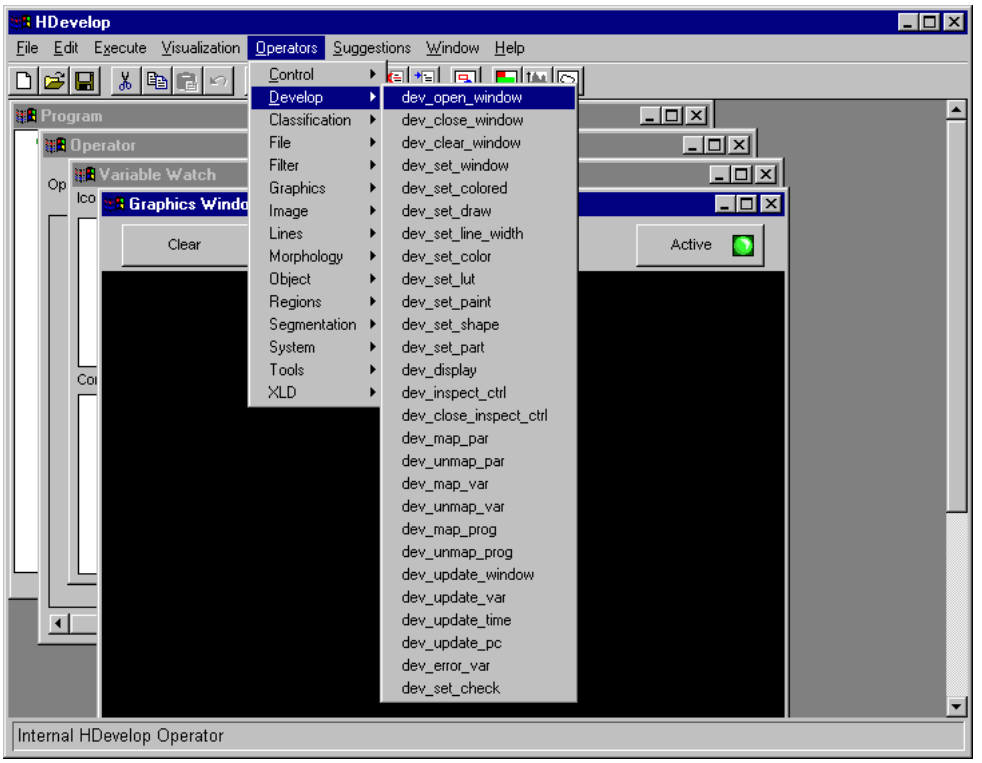


Figure 4.24: *Operators in menu item* `Develop`.

- `dev_set_color, dev_set_colored`
  `dev_set_color` has the same effects as the menu item `Visualization` ▷ `Color`. `dev_set_colored` is equal to the menu item `Visualization` ▷ `Colored`.

- `dev_set_draw`
  This operator has the same effects as `Visualization` ▷ `Draw`.

- `dev_set_line_width`
  For an explanation see item `Visualization` ▷ `Line Width`.

- `dev_set_lut`
  For an explanation see item `Visualization` ▷ `Lut`.

- `dev_set_paint`
  For an explanation see item `Visualization` ▷ `Paint`. If you want to specify all possible parameters of a given paint mode, you have to specify them as a tuple, analogously to the HALCON operator `set_paint`.

- `dev_set_shape`
  For an explanation see item `Visualization` ▷ `Shape`.

- `dev_set_part`
  This operator adjusts the coordinate system for image, region, XLD and other graphic output. This is done by specifying the upper left and the lower right corner coordinates. This

specified part is shown in the entire graphics window. If the width or height of the specified rectangle has a negative value (e.g., `Row1` > `Row2`) the result is equivalent to the menu `Visualization` ▷ `Zooming` ▷ `Reset`: the zoom mode is switched off, i.e., the *most recently* displayed image fills the whole graphics window. This feature of `dev_set_part` is *not* supported for exported C++ and Visual Basic code.

- `dev_display`
  Iconic variables are displayed in the active graphics window by this operator. It is reasonable to do this when the automatic output is suppressed (see `dev_update_window` and `File` ▷ `Options...` on page 29).

- `dev_clear_obj`
  This operator deletes the iconic object stored in the HDevelop variable that is passed as the input parameter. In the variable window, the object is displayed as undefined (with a `?` as its icon).

- `dev_inspect_ctrl`
  This operator opens an inspection window displaying the values of the variable passed to the operator. In most cases a list dialog is opened, which shows *all* values of the variable. In the case of a framegrabber handle, a description of this framegrabber is opened. In addition, this dialog allows online grabbing of images. This operator is not supported for exported C++ and Visual Basic code.

- `dev_close_inspect_ctrl`
  This is the opposite operator to `dev_inspect_ctrl`, and closes the inspect window. This operator is not supported for exported C++ and Visual Basic code.

- `dev_map_par, dev_unmap_par`
  These operators open and close the parameter dialog, which can also be opened using the menu `Visualization` ▷ `Set Parameters...`. This operator is not supported for exported C++ and Visual Basic code.

- `dev_map_var, dev_unmap_var`
  These operators iconify the variable window (`dev_unmap_var`), and retransform the iconified window to the normal visualization size, respectively (`dev_map_var`). This means that the variable window always remains visible on the display in one of the two ways of visualization. These operators can be executed with the help of the window manager. These operators are not supported for exported C++ and Visual Basic code.

- `dev_map_prog, dev_unmap_prog`
  Analogously to `dev_map_var` and `dev_unmap_var`, these operators iconify or deiconify the program window. These operators are not supported for exported C++ and Visual Basic code.

- `dev_update_window, dev_update_var, dev_update_time, dev_update_pc`
  Using these operators, you may configure the output at runtime. It corresponds to the settings in menu `Visualization` ▷ `Options...` (see page 36). These operators are not supported for exported C++ and Visual Basic code.

- dev_set_check
  This operator is equivalent to set_check of the HALCON library. It is used to handle runtime errors caused by HALCON operators that are executed inside HDevelop. The parameter value 'give_error', which is the default, leads to a stop of the program together with an error dialog if a value not equal to H_MSG_TRUE is returned. Using the value '~give_error', errors or other messages are ignored and the program can continue. This mode is useful in connection with operators like get_mposition, file_exists, read_image, or test_region_point, which can return H_MSG_FAIL. An example can be found on page 121.

- dev_error_var
  This operator specifies a variable that contains the return value (error code) of an operator after execution. This value can be used to continue, depending on the given value. dev_error_var is normally used in connection with dev_set_check. An example how to use dev_error_var in connection with dev_set_check can be found in

  %HALCONROOT%\examples\hdevelop\Graphics\Mouse\get_mposition.dev .

Please note that operations concerning graphics windows and their corresponding operators have additional functionality as HALCON operators with corresponding names (without dev_): graphics windows in HDevelop are based on HALCON windows (see open_window in the HALCON reference manual), but in fact, they have an enhanced functionality (e.g., history of displayed objects, interactive modification of size, and control buttons). This is also true for operators that modify visualization parameters (dev_set_color, dev_set_draw, etc.). For example, the new visualization parameter is registered in the parameter window when the operator has been executed. You can easily check this by opening the dialog Visualization ▷ Set Parameters... ▷ Pen and apply the operator dev_set_color. Here you will see the change of the visualization parameters in the dialog box. You have to be aware of this difference if you export dev_* to C++ and Visual Basic code.
In contrast to the parameter dialog for changing display parameters like color, the corresponding operators (like dev_set_color) do not change to contents of the graphics window (i.e., they don't cause a redisplay). They are used to prepare the parameters for the *next* display action.
□ *All HALCON operators*
Here you can find all HALCON operators, arranged in chapters and subchapters. This set of image analysis operators forms the most important part of HALCON: the HALCON library. HALCON operators implement the different image analysis tasks such as preprocessing, filtering, or measurement (see figure 4.25).
You may look for a detailed description of each operator in the HALCON reference manual.[5]
The menu has a cascade structure, according to the chapter structure of the HALCON reference manual. As this menu has to be built up after opening the program window, it might take some time until it is available. During the build-up time the menu is "grayed out". Selecting a chapter of the menu opens a pulldown menu with the corresponding subchapters or operators, respectively.
This operator hierarchy is especially useful for novices because it offers all operators sorted by thematic aspects. This might be interesting for an experienced user, too, if he wants to compare,

[5]Operators of the menus Control and Develop are special operators of HDevelop. Thus you will not find them in the reference manuals.

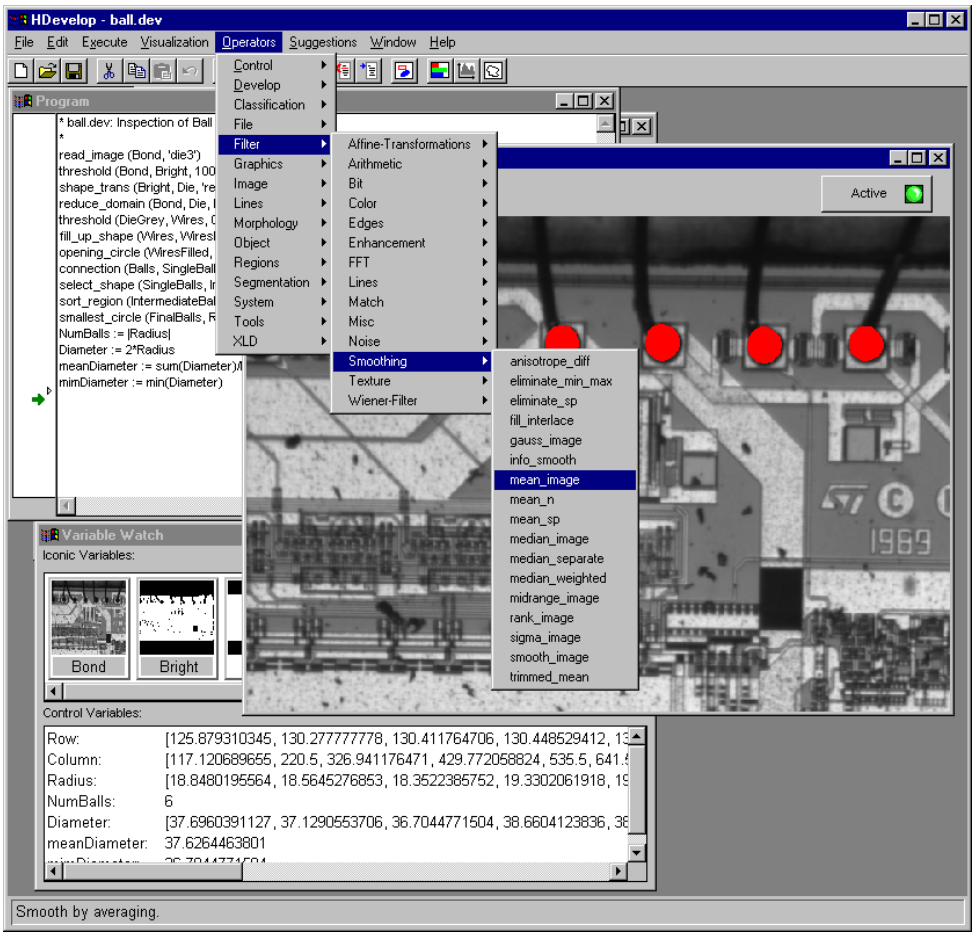


Figure 4.25: *Menu hierarchy of all HALCON operators.*

e.g., different smoothing filters, because they reside in the same subchapter. To get additional information, a short description of an operator (while activating its name in the menu) is displayed in the status bar (see figure 4.25).

Menu item: `Suggestions`

This menu shows you another possibility how to select HALCON operators. But here they are proposed to you in a different manner. It is assumed that you have already selected or executed an operator in a previous step. Depending on this operator, five different suggestions are offered. Figure 4.26 shows possible successor suggestions for operator `read_image`.
Suggestions are separated into groups as follows:

`Predecessor:` Many operators require a reasonable or necessary predecessor operator. For example before computing junction points in a skeleton (`junctions_skeleton`), you have to compute this skeleton itself (`skeleton`). To obtain a threshold image you have to use a lowpass filter before executing a dynamic threshold (`dyn_threshold`). Using the watershed algorithms (`watersheds`), it is reasonable to apply a smoothing filter on an image first, because this reduces runtime considerably.

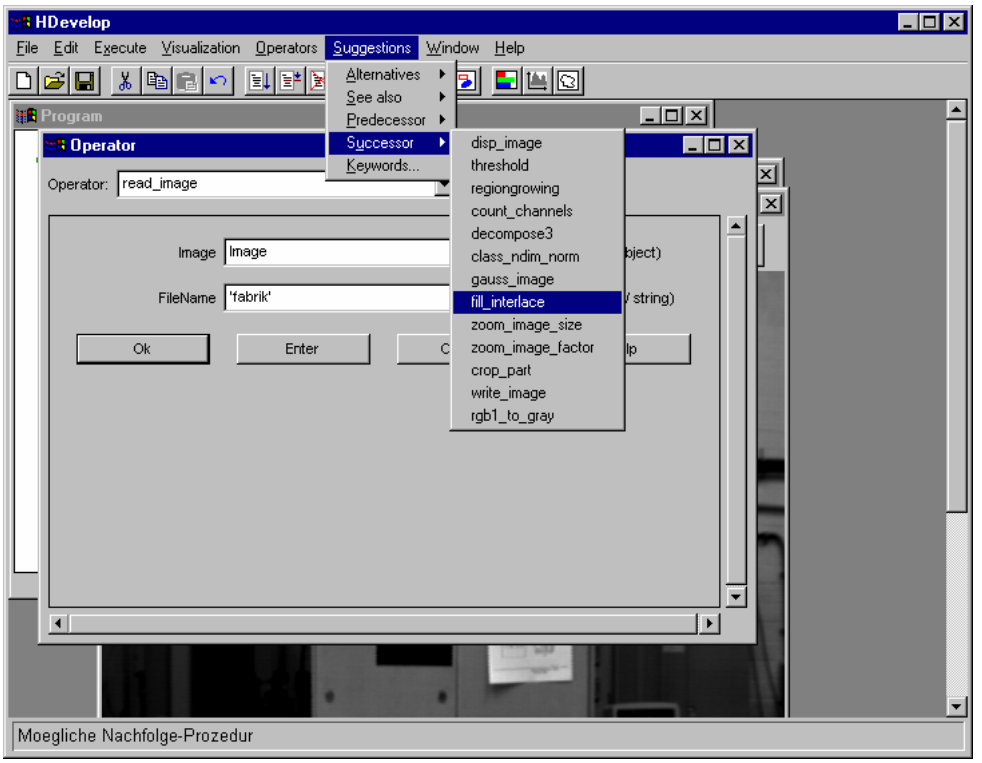


Figure 4.26: *Suggestions to select a successor of HALCON operator* `read_image`.

`Successor:` In many cases the task results in a "natural" sequence of operators. Thus as a rule you use a thresholding after executing an edge filter or you execute a region processing (e.g., morphological operators) after a segmentation. To facilitate a reasonable processing all the possible operators are offered in this menu item.

`Alternatives:` Since HALCON includes a large library, this menu item suggests alternative operators. Thus, you may, for example, replace `mean_image` with operators such as `gauss_image`, `sigma_image`, or `smooth_image`.

`See also:` Contrary to `Alternatives`, operators are offered here which have some *connection* to the current operator. Thus, the median filter (`median_image`) is not a direct alternative to the mean filter (`mean_image`). Similarly, the regiongrowing operator (`regiongrowing`) is no alternative for a thresholding. In any case, they offer another approach to solve a task. References might consist of pure informative nature, too: the operator `gen_lowpass`, which is used to create a lowpass filter in the frequency domain, is a reasonable reference to a Gaussian filter.

`Keywords:` This menu item gives access to HALCON operators by using keywords which are associated with each operator. You get a window, divided into two parts, which contains all keywords on the left hand side and the selected operators on the right (see figure 4.27).

After the suggestions for an operator have been generated, all keywords belonging to this operator are marked (reversed) on the left hand side of the window[6]. On the right side you

[6]Because there are many entries in the left keyword list, you may see all marked keywords only by scrolling it.

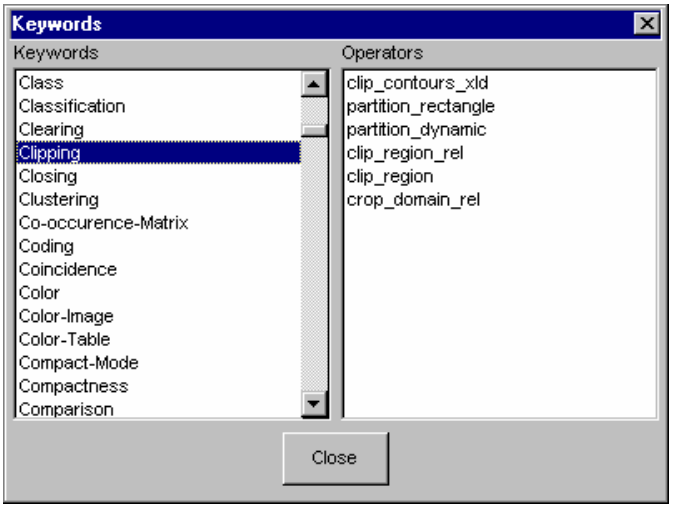


Figure 4.27: *Operator suggestions according to keyword "Clipping".*

will find all operators associated with at least one of these keywords. Clicking a keyword on the left list causes the addition of operators belonging to this keyword. If you want to transfer one of these operators to the operator dialog area, you click one of them with the left mouse button. Afterwards the selection window is closed.

Menu item: `Window`

This menu item offers support to manage your four windows, i.e., the program, operator, variable and graphics window. They are very useful while working with HDevelop. You see the items in figure 4.28. This menu item is not supported in a UNIX environment, because according to the X-Windows style we don't have a main window with a functionality similar to Windows NT.

- The item `Cascade`
  By pressing this item, HDevelop arranges the four windows in a cascade as you can see in figure 4.28.

- The item `Tile`
  You see all four windows inside the main window. They have the same size and fit exactly in the main window. Thus, you get a global view of the windows' contents at once. Notice that the four windows may shrink depending on their size to fit in the main window. Figure 4.29 shows you the effect using this item.

- The item `Arrange Icons`
  As in every system using windows, you are able to iconify and deiconify your windows. You may even move your icons on the display. This might create a confusing working environment if you are handling several graphics tools simultaneously. To get the HDevelop icons back on top of the main window's status bar you just have to press this button.

- The item `Next`
  By using this item you bring the top window in the background. Hence it looses its window focus. The window to get the window focus and to become the top window is the window which was only hidden by the former top window.

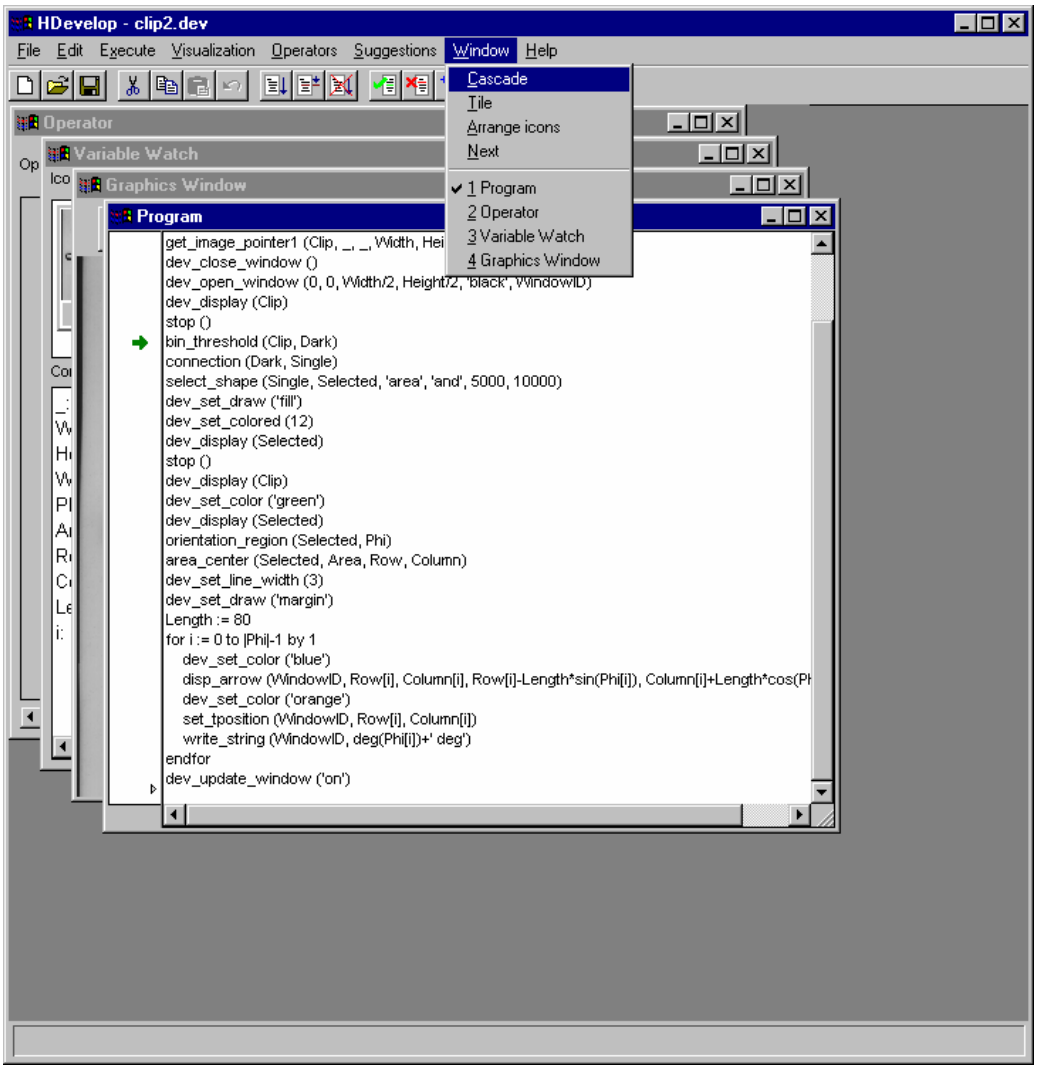


Figure 4.28: *Window management functions.*

If you use one of the next four items (`Program Window`, `Operator Window`, `Variable Window` and `Graphics Window`) the specified window will become the top window and gets the window focus.

Menu item: `Help`
Here you may query information about the system itself and all HALCON and HDevelop operators.
The menu item `Help` ▷ `About...` delivers information about the current HALCON version (see figure 4.30).
The next menu item is `Help` ▷ `Halcon Operators`. This help is based on an HTML browser (see chapter 2.3). The browser will display the main page of all HALCON and HDevelop operators. It is quite easy for you to browse through this operator menu and to find the desired operator.
The menu item `Help` ▷ `HDevelop Language` starts the HTML browser with a description of the language, similar to chapter 5 of this manual.

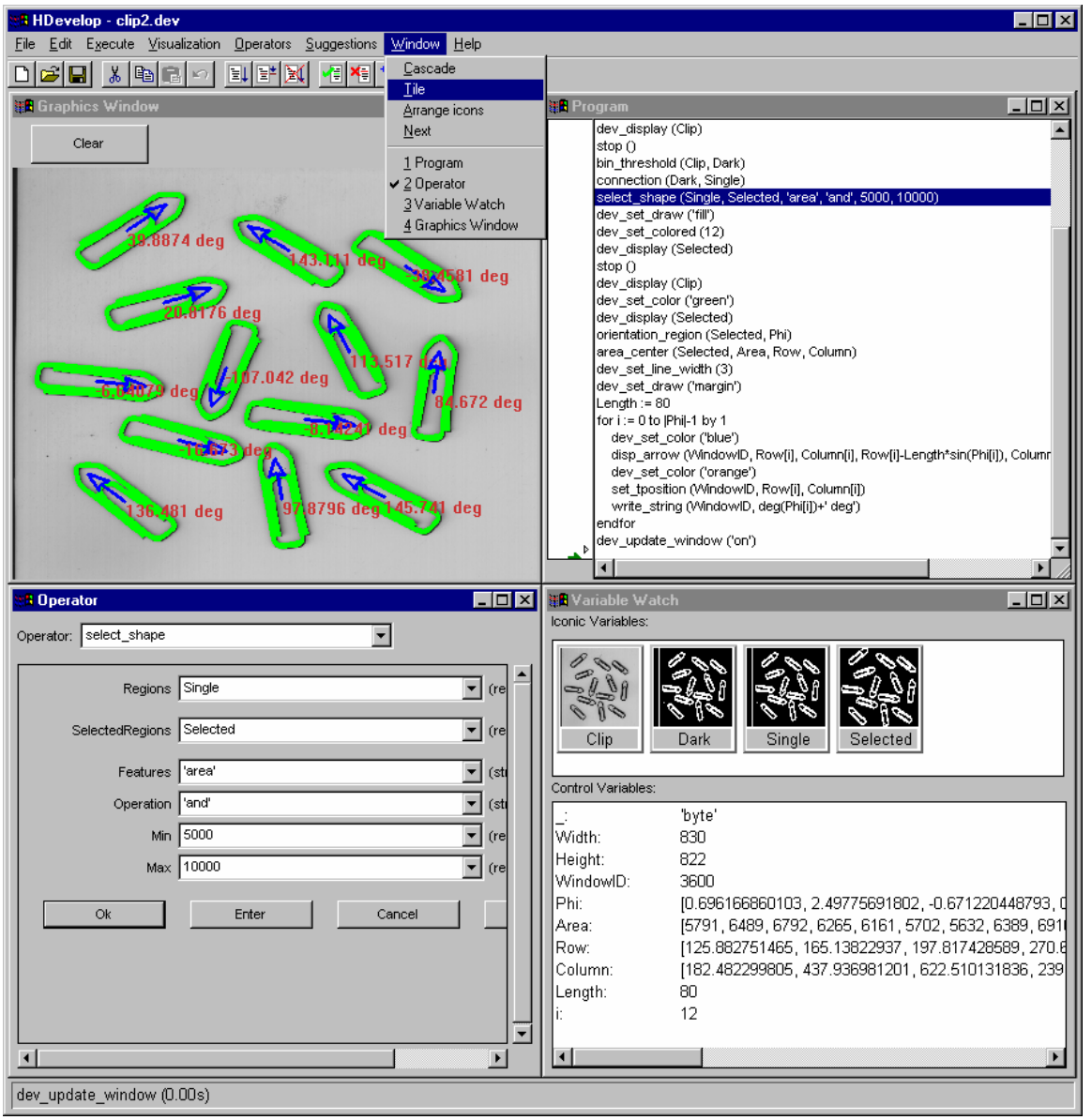


Figure 4.29: *The window management function* `Tile`.

Figure 4.30: *Information about the current HALCON version.*

Another possibility of requesting information about the current operator is pressing button `Help` inside the operator window (see page 61).

4.3.3 Tool Bar

You use most icons in this tool bar to accelerate accessing important HDevelop features. These are features which you are performing many times while working with HDevelop. Hence there are buttons to handle your HDevelop programs and to edit them. The most important buttons are used to start and to stop a program (or parts of a program). These icons are explained in figure 4.32:

Figure 4.31: *The HDevelop tool bar.*

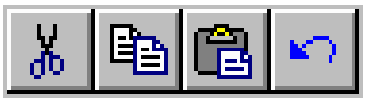
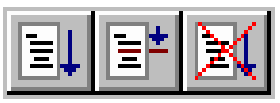
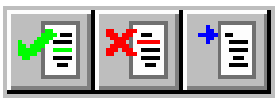
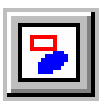
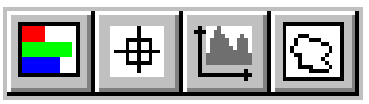
	These icons are shortcuts for the menu items `File` ▷ `New`, `File` ▷ `Load`, and `File` ▷ `Save` in the menu bar. For a detailed description, see page 24ff.
	These icons are shortcuts for the menu items `Edit` ▷ `Cut`, `Edit` ▷ `Copy`, `Edit` ▷ `Paste`, and `Edit` ▷ `Undo` in the menu bar. For a detailed description, see page 31ff.
	These icons are shortcuts for the menu items `Execute` ▷ `Run`, `Execute` ▷ `Step` and `Execute` ▷ `Stop` in the menu bar. For a detailed description, see page 32ff.
	These icons are shortcuts for the menu items `Execute` ▷ `Activate`, `Execute` ▷ `Deactivate`, and `Execute` ▷ `Reset Program` in the menu bar. For a detailed description, see page 32ff.
	This icon is a shortcut for the menu item `Visualization` ▷ `Set Parameters...` in the menu bar. For a detailed description, see page 34ff.
	These icons are shortcuts for the menu items `Visualization` ▷ `Pixel Info...`, `Visualization` ▷ `Zooming...`, `Visualization` ▷ `Gray Histogram Info...`, and `Visualization` ▷ `Region Info...` in the menu bar. For a detailed description, see page 34ff.

Figure 4.32: The different parts to the HDevelop toolbar.

4.3.4 Window Area

The window area contains all necessary windows to show your HDevelop programs, to visualize your iconic and control results, and to specify any operator's parameters. Additionally, you may

open as many graphics windows as you want to get a detailed view of your iconic results.
You are free to move the windows according to your needs and preferences inside this area. You may iconify and/or deiconify them. To handle these windows in a comfortable way, HDevelop supports you with some window management functions (see 4.3.2).

4.3.5 Status Bar

The status bar at the bottom of the program window shows you information which is important while working with HALCON, e.g., context sensitive information about a specific user action or the operator runtime (if time measurement has not been deactivated).

4.4 Program Window

The program window is divided into two areas. The left one (a small column) contains the program counter (`PC`, represented as a green arrow pointing to a program line), the insertion cursor (a triangle between two program lines) and optionally a *breakpoint* (`BP` — a red STOP sign). You may position or activate these three labels as follows:

- The `PC` is set by pressing the left mouse button only.
- The insertion cursor is set by pressing the left mouse button and the `<Shift>` key.
- The `BP` is set by pressing the left mouse button and the `<Ctrl>` key.

The program counter resides in the line of the next operator to execute. The insertion cursor indicates the position to insert a new program line. The breakpoint shows the program line before which the program is stopped. Clicking on the breakpoint again while pressing the `<Ctrl>` key deletes it. A breakpoint is active only if it is visible. You may activate only one breakpoint. In figure 4.33 you see a program and the column with the `PC` (indicated as an arrow), the `BP` and the insertion cursor.
The big text area at the right side of the program window contains the program code of the HDevelop program. Here the user has the possibility to obtain information about the inserted operators. A program is built up such that every line contains exactly *one* operator with its parameters or an assignment. An exception are the condition constructs `if` and `ifelse` respectively, and the loop constructs `while` and `for`. They contain two, in case of `ifelse` even three, program lines, which enclose the body. Every line starts with an operator name, which is indented if necessary, to highlight the structure created by the above mentioned control structures. After the operator name the parameters are displayed in parentheses. Parameters are separated by commas.
The program window is used to *visualize* program lines, but not to modify them. You cannot change a program by modifying the text directly. Editing the program text in HDevelop is done in the *operator window* (this will be described below). The main reason for this principle is the advantage of providing sophisticated help. Thus you are able to avoid many input errors.
To edit a line of a program you chose an operator in the program window by clicking the left mouse button twice. In case of conditions and loops it is unimportant which lines (e.g., `for` or `endfor`) are selected. In any case, the head with its parameters is selected. You may edit only *one* operator at a time.

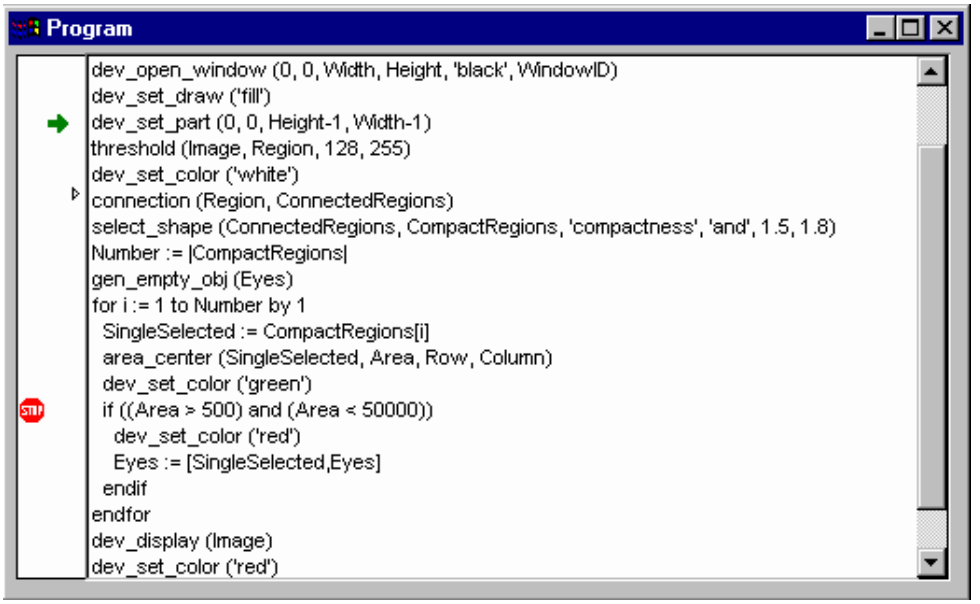


Figure 4.33: *Program example with the* `PC` *(the arrow pointing to the right), insertion cursor and the the breakpoint (*`BP`*).*

Besides editing the parameters of a single operator, single and multiple lines can be deleted, cut, or pasted in one step using simple mouse functions. To use this feature, one has to select one or more lines using the mouse:

- The selection of *one* line is done by clicking on it. Previously activated lines will then become deactivated.
- To activate more than one line you have to press the `<Ctrl>` key while clicking on the line. If the line is already activated it will become deactivated, while the state of all other lines remains unchanged.
- The `<Shift>` key is used to activate a sequence of lines using one mouse click: All lines between the most recent activation and the new one will become activated.

After the selection of lines, the edit function can be activated by either using the menu `Edit` (see sections 4.3.2) or the tool bar (see sections 4.3.3). Further information on the use of the mouse can be found in section 4.2.

4.5 Operator Window

This window is mainly used to edit and display an operator and all its parameters. Here you will obtain information about the number of the operator's parameters, the parameter types, and parameter values. You are able to modify the parameter values according to your image processing tasks. For this you may adopt the proposed system values or specify your own values. The operator window consists of the following three parts:

- The first one is the operator text field.

- The second one is the largest part. It is called parameter display and is used to edit the parameters of an operator.
- The last one is a row of buttons to control the parameter display.

Parameter Display

The parameter display is the main part of the operator window. It is empty in its initial state. If you have selected an operator, HDevelop displays the operator's parameter data, i.e., name, number, type, and default values, in the display.

- In the first column of the operator window you find the parameter names.
- The second column consists of the text fields, which contain variable names in case of iconic and control output parameters and expressions in case of control input parameters. If you want to change the suggestions offered by the system (variable names or default values) you may do so either manually or by pressing the arrow button connected with the respective text field. This opens a list containing a selection of already defined variables and other reasonable values from the operator knowledge base. By clicking the appropriate item you set the text field and the list disappears.
- The third column indicates the parameter's default type in parentheses. This is a raw hint for the user, what types of data have to be specified for each operator.

Please refer to the following rules on how parameters obtain their values and how you may specify them:

Iconic input parameters:
These are the only parameters that do not have any defaults. You have to make sure that there is an input. Possible inputs are iconic variables of the corresponding list. If there is no need to execute the operator immediately, you may even specify new variable names, i.e., names, that do not already exist in the variable window, but will be instantiated later by adding further operators to the program. In any case you have to specify iconic parameters *exclusively with variable names*. It is not possible to use expressions.

Iconic output parameters:
These parameters contain default variables, which have the same names as the parameters themselves. If a variable with the same name as the output parameter is already being used, a number is added to the name to make it unique. Because the parameter names characterize the computed result very well, you may adopt these default names in many cases. Besides this, you are free to choose arbitrary names either by yourself or by opening the list (see above). If you use a variable that already has a value, this value is deleted during execution before overwriting it with new results. It is possible to specify a variable both in an input and output position.

Control input parameters:
These parameters normally possess a default value. In particular, this is to support novices selecting an appropriate value. As an alternative, you may use the text field's button to open a combo box and to select a value suggestion. In addition, this combo box contains

a list of variables that contain values of the required type. A restriction of proposed variables is especially used for parameters that contain data like file, framegrabber, or ocr handles.

Input control parameters may contain constants, variables, and expressions. Common types are integer numbers (`integer`), floating point numbers (`real`), boolean values (`true` and `false`) and character strings (`string`). You are able to use multiple values of these types at once. This is done by using the *tuple type*. This is an enumeration of values, separated by commas and enclosed in brackets. Furthermore, you may build up expressions with these values. The possibilities of using tuples are very extensive. You may use expressions in HDevelop similar to the use of expressions in C or in Pascal. You will find a more detailed description in section 5.5.

Control output parameters:
These parameters are handled in the same way as output object parameters. Their defaults are variables, named with their parameter names. Other possibilities to obtain a control output variable name are either using the combo box or specifying variable names manually. You cannot use any expressions for these parameters, either.

After discussing what can be input for different parameters, it is explained *how* this is done. Nevertheless, you have to keep in mind that you need to modify a parameter only, if it contains no values or if you are not satisfied with the HALCON default values.

Text input:
To specify a parameter using your keyboard is the simplest but not the most often used method. Here you have to click into a text field with the left mouse button. This activates the field and prepares it for user input. Simultaneously, the writing position is marked by a vertical bar. Now you may input numbers, strings, expressions, or variables. There are some editing functions to help you doing input: `<Backspace>` deletes the left and `<Delete>` deletes the right character. You may also select (invert) a sequence of characters in the text field using the mouse. If there is a succeeding input, the marked region is going to be deleted first and afterwards the characters are going to be written in the text field. You can find additional editing functions on page 125.

Combo box selection:
Using this input method, you can obtain rapid settings of variables and constants. To do so, you have to click the button on the text field's right side. A combo box is opened, in which you may select an item. Thus you are able to choose a certain variable or value without risking erroneous typing. This item is transferred to the operator name field. Previous entries are deleted. Afterwards the combo box is closed. If there are no variables or appropriate values, the combo box remains closed.

Below the parameter edit fields you find four buttons that comprise the following functions (see figure 4.34):

- By clicking `OK` you execute the operator with the specified parameters. In doing so, the execution mode is dependent on the position of the `PC`: If the `PC` is placed above the insertion position, the system computes the program from the `PC` until the insertion position first. *Then* the operator that has been edited in the operator window is executed.

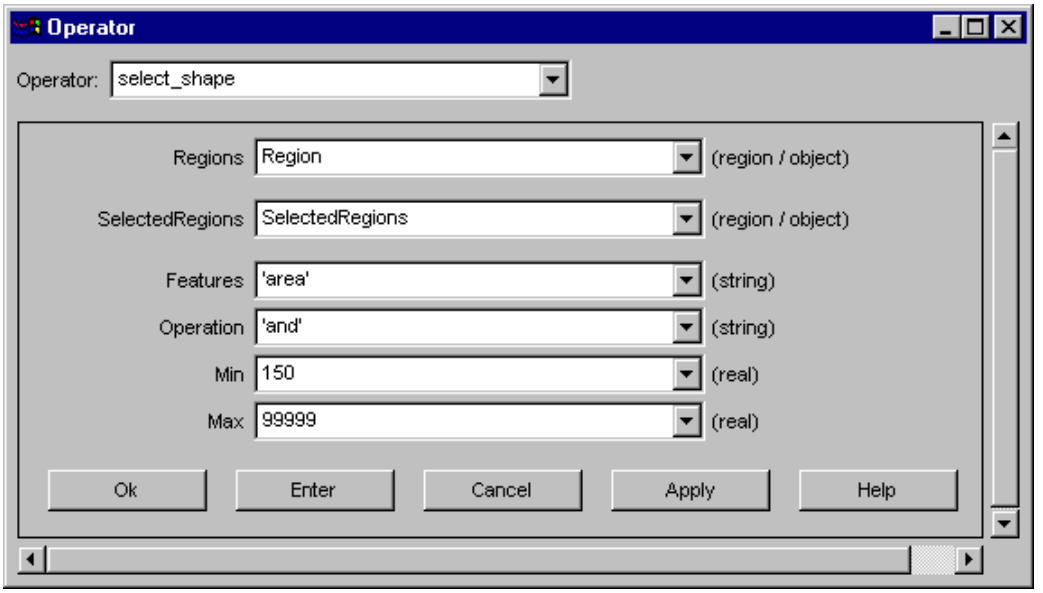


Figure 4.34: *Operator window with operator* `select_shape`.

The reason for this is that the parameter values that are used as input values for the new operator have to be calculated. In case the `PC` is placed at or after the insertion position, this operator is executed only. Before execution the operator is entered into the program window. The `PC` and the insertion cursor are positioned after the new operator. The computed output parameter values are displayed in the variable window. Iconic variables are shown in the current graphics window, if you haven't suppressed this option (see page 36). Afterwards the operator window is cleared. If you did not specify all parameters or if you used wrong values, an error dialog is raised and execution is canceled. In this case, the operator window remains open to support appropriate changes.

- By clicking the button `Enter` the program line is only transferred in the program window. There is no operator execution, i.e., the position of the `PC` is not changed. The insertion cursor is positioned after the new operator.

- If you click `Apply` the operator is executed with the specified parameters, but not entered into or changed in the program. This enables you to determine the optimum parameters of an operator rapidly since the operator dialog remains open, and hence you can change parameters quickly. Unlike the button `OK`, only the single line you edit or enter is executed, no matter where the `PC` is located. Thus, you have to ensure that all the input variables contain maningful values. By pressing `Apply`, the corresponding output variables are changed or created, if necessary, to allow you to inspect their values. If you decide to not enter the line into the program, some unused variables may thus be created. You can easily remove them by selecting `File` ▷ `Cleanup`.

- `Cancel` clears the contents of the operator window. Thus, there are neither changes in the program nor in any variables.

- `Help` invokes an appropriate help text for the selected operator. For this the system activates an HTML-browser (see chapter 2.3).

Operator Name Field

The operator name field is another possibility to select operators. You simply have to enter a substring of an operator name. By pressing `<Return>` or pressing the button of the combo box the system is looking for all operators (in the menu `Operators`) that contain the user-specified substring (see figure 4.35). If there is an unambiguous search result, the operator is displayed immediately in the operator window. If there are several matching results, a combo box opens and displays all operators containing the specified substring. By clicking the left mouse button you select one operator and the combo box disappears. Now the operator's parameters are shown in the operator window.

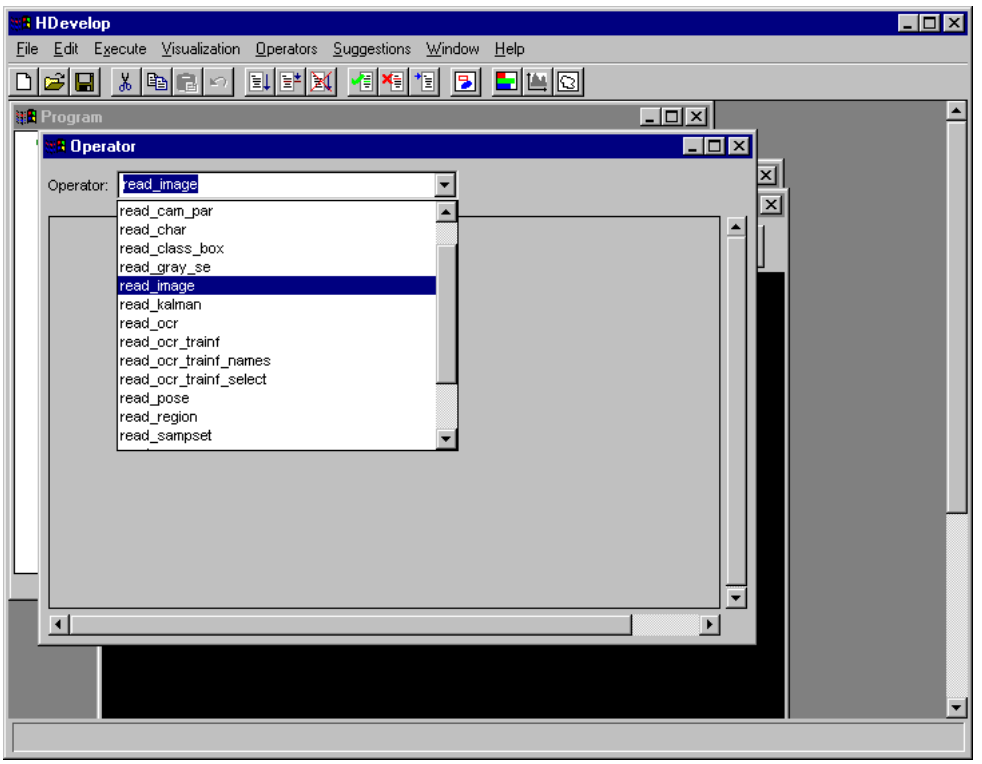


Figure 4.35: *Operator selection in the operator name field.*

If you are already more familiar with HDevelop, it is reasonable to select an operator in the operator name field. However, in order to do so, you obviously have to be familiar with the operator names.

4.6 Variable Window

There are two kinds of variables in HALCON. This corresponds to the two parameter types of HALCON: iconic objects (images, regions, and XLDs) and control data (numbers, strings). In HALCON the corresponding variables are called iconic and control variables. These variables may possess a value or be undefined. An undefined variable is created, for example, when loading a program or after inserting an operator with a new variable that is not executed immediately into a program. You may access these undefined variables by writing them only. If you try to read such a variable, a runtime error occurs. If a variable obtains a value, the variable type is

specified more precisely. A control variable that contains, for example, an integer is of type `integer`. This type might change after specification of new values for this variable to `real` or a tuple of `integer`. But it always remains a control variable. The more refined type is bound to the value and not to the variable. Similarly, this is the case for iconic variables, which may contain regions, images, or XLDs. You may assign new values as often as you want to. But you cannot change them to the state before the first assignment (see above).
Creation of a new variable happens in the operator dialog area during specification of operator parameters. Here every sequence of characters without single quotation marks is interpreted as a variable name. If this name did not exist before, the variable is created in the operator dialog area by pressing `OK` or `Enter`. The variable type is specified through the type of the parameter where it was used for the first time: Variables that correspond to an iconic object parameter create an iconic variable; variables for a control parameter create a control variable. Every time an operator is excecuted the results are stored in variables connected to its output parameters. This is done by first deleting the contents of the variable and then assigning the new value to it.
The variable window is a kind of watch window used in window-oriented debuggers. Inside this window you are able to keep track of variable values. Corresponding to the two variable types, there are two areas in the variable window. One for iconic data (above) and the other for control data (below) (see figure 4.36).

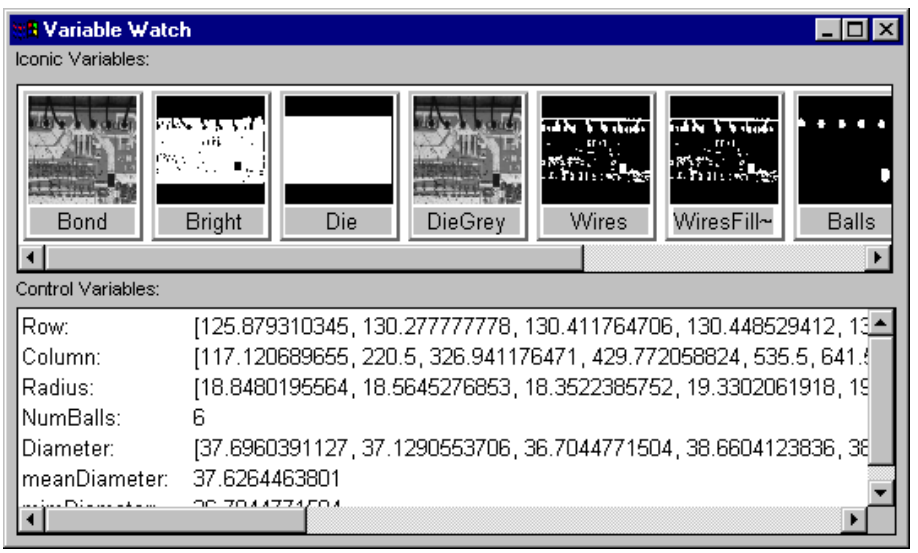


Figure 4.36: *Variable window for both iconic and control data.*

All computed variables are displayed showing their iconic or control values (see page 27). In case of a tuple result which is too long, the tuple presentation is shortened, indicated by three dots.

4.6.1 Area for Iconic Data

Here you can see iconic variables. They are represented by icons, which contain a gray image, a region or an XLD, depending on the current value. The icons are created depending on the type of data according the following rules:

- In the case of images the icon contains a zoomed version of it filling the icon completely. Due to the zooming onto the square shape of the icon the aspect ratio of the small image might be wrong. If there is more than one image in the variable, only the *first* image is used for the icon. Similarily for multi channel images only the *first* channel is used.
- Regions are displayed by first calculating the smallest surrounding rectangle and then zooming it so that it fills the icon using a border of one pixel. In contrast to images, the aspect ratio is always correct. This can lead to black bars at the borders. The color used to draw the region is always white without further modifications (except zooming).
- XLD data is displayed using the coordinate system of the largest image used so far. The color used for XLD objects is white on black background.

Due to the different ways of displaying objects you have to be aware that the coordinates cannot be compared. The variable name is positioned below each icon. They are displayed in the variable window in the order of creation from left to right. If there is not enough space, a horizontal scrollbar is created, which you can use to scroll the icons.
Clicking on an icon with the mouse will select this icon. This is indicated by the black background for the icon name. For an activated icon all operators that use the corresponding variable are marked in the program area with a black rectangle on the left.
Double-clicking with the left mouse button on an icon displays the data in the active graphics window. If you use images of different sizes in a program, the system uses the following output strategy for an automatic adaption of the zooming: Every window keeps track of the size of the most recently displayed image. If you display an image with a different size, the system modifies the graphics window coordinate system in a way that the image is visible completely in the graphics window. If a partial zooming has been activated before (see chapter 4.7), it is going to be suppressed.
Normally, regions, images, and XLDs are represented in variable icons. Besides this there are three exceptions which are shown by special icons:

- Empty variables are displayed as a question mark (?) icon. You may *to write* but not read them, because they do not have any values.
- Brackets ([]) are used if a variable is instantiated but does not contain an iconic object (empty tuple). This may be the case using operators like `select_shape` with "wrong" specified thresholds or using operator `empty_obj`. Such a value might be reasonable if you want to collect iconic objects in a variable gradually in a loop (`concat_obj`). Here, an empty tuple is used as starting value for the loop.
- A last exception is an *empty region*. This is *one* region that does not contain any pixels (points), i.e., the area (number of points) is 0. You must not confuse this case with the empty tuple, because there the area is not defined. The empty region is symbolized by an empty set icon (∅).

4.6.2 Area for Control Data

To the right of the variable name you find their values in the default representation[7]. If you specify more than one value for one variable (tuple), they are separated by commas and enclosed by

[7] You have to keep in mind that a floating point number without significant fractional part is represented as an integer (e.g., 1.0 is represented as 1).

brackets. If the number of values exceeds an upper limit, the output is clipped. This is indicated by three dots at the end of the tuple. For empty variables, their name and a `?` are shown in the variable field. An empty tuple is represented by `[]`. Both exceptions use the same symbols as the corresponding cases for the iconic variables.
Clicking on a variable will select it. Similar to iconic variables, all program lines that use this variable are then marked with a black rectangle on the left.
Double-clicking a control variable opens a window that displays all its values. In most cases this will be a dialog containing a scrolled list. This is helpful if you have variables with a large number of values that you want to inspect. In the case of a framegrabber handle, a dialog

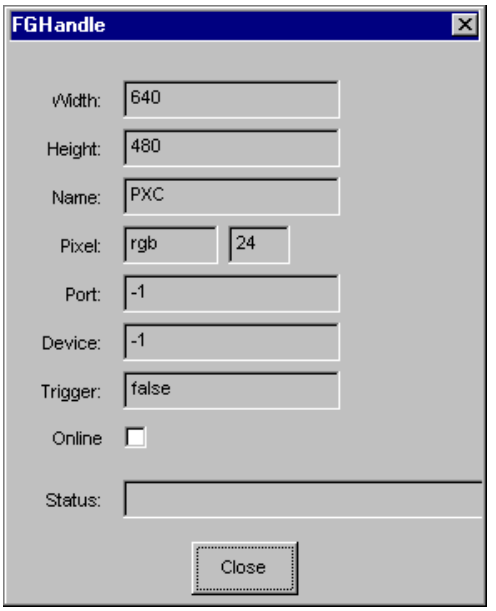


Figure 4.37: *Variable inspection for framegrabber handles.*

representing basic framegrabber parameters is opened (see figure 4.37). Here you find the size, name, device, port, and other features of the framegrabber. The toggle button `Online` allows to grab images continuously and to display them in the active graphics window. If an error occurs during grabbing, it is displayed in the status bar of the dialog. At most one of these framegrabber dialogs can be opened at the same time.

4.7 Graphics Window

This window displays iconic data. It has the following properties:

- The user may open several graphics windows.
- The active graphics window is shown by the green dot in the `Active` button.
- Pressing the `Clear` button clears the graphics window content and the history of the window.
- You close a graphics window using the close button of the window frame.

Figure 4.38: *HDevelop's graphics window.*

Figure 4.38 shows an example for a graphics window.
The the origin of the graphics window is the upper left corner with the coordinates (0,0). The x values (column) increase from left to right, the y values increase from top to bottom. Normally, the coordinate system of the graphics window corresponds to the the most recently displayed image, which is automatically zoomed so that every pixel of the image is visible. The coordinate system can be changed interactively using the menu `Visualization` ▷ `Set Parameters...` ▷ `Zoom` (see section 4.3.2) or with the operator `dev_set_part` (see page 50). Every time an image with another size is displayed, the coordinate system will be adapted automatically.
Each window has a history that contains all

- objects and
- display parameters

that have been displayed or changed since the most recent `Clear` or display of an image. This history is used for redrawing the contents of the window. The history is limited to a maximum number of 30 "redraw actions", where one redraw action contains all objects of one displayed variable.

Other output like text or general graphics like `disp_line` or `disp_circle` or iconic data that displayed using HALCON operators like `disp_image` or `disp_region` are *not* part of the history, and are *not* redrawn. Only the object classes image, region, and XLD that are displayed with the HDevelop operator `dev_display` or by double clicking on an icon are part of the history.

You may change the size of the graphics window interactively by "gripping" the window border with the mouse. Then you can resize the window by dragging the mouse pointer. After this size modification the window content is redisplayed. Now you see the same part of the window with changed zoom.

The menu area of the graphics window has an additional function: If the mouse cursor is in this area the look up table of the window is reactivated. This is necessary if other programs use their own look up table. Thus if there is a "strange" graphics window presentation, you may load the proper look up table by placing the mouse near the buttons.

If you want to specify display parameters for a window you may select the menu item `Visualization` in the menu bar. Here you can set the appropriate parameters by clicking the desired item (see section 4.3.2). The parameters you have set in this way are used for *all* windows. The effects of the new parameters will be applied direcly to the *last* object of the window history and alter its parameters only.

For further information on parameter effects please refer to the appropriate HALCON operators in the reference manual.

Chapter 5

Language

The following chapter introduces the syntax and the semantics of the HDevelop language. In other words, it illustrates what you can enter into a parameter slot of an operator. In the simplest case this is the name of a variable, but it might also be an expression like `sqrt(A)`. Besides, control structures (like loops) and the semantics of parameter passing are described.
Chapter 7 explains the application of this language in image analysis. However, the HALCON operators are not described in this chapter. For this purpose refer to the HALCON reference manual. All program examples used in this chapter can also be found in the directory `$HALCONROOT/examples/hdevelop/Manuals/HDevelop`.

5.1 Basic Types of Parameters

HALCON distinguishes two kinds of data: control data (numerical/string) and iconic data (images, regions, etc.).
By further distinguishing *input* from *output parameters*, we get four different kinds of parameters. These four kinds always appear in the same order in the HDevelop parameter list. Table 5.1 shows their order of appearance.

iconic	input
iconic	output
control	input
control	output

Table 5.1: *Order of appearance of the four basic parameter types*

As you see, iconic input objects are always passed as the first parameter(s), followed by the iconic output objects. The iconic data is followed by the control data, and again, the input parameters succeed the output parameters. Each parameter is separated from its neighbours by a comma:

```
read_image (Image, 'Name')
area_center (Region, Area, Row, Column)
mean_image (Image, Mean, 11, 11)
```

In the above example the operator `read_image` has one output parameter for iconic objects (`Image`) and one input control parameter (filename). `area_center` accepts regions as input (iconic) and three control parameters as output (`Area`, `Row`, `Column`). The filter operator `mean_image` has one iconic parameter as input and one as output. Its two input control parameters specify the size of the filter mask.
Input control parameters can either be variables, constants or even complex expressions. An expression is evaluated *before* it is passed to a parameter that receives the result of the evaluation. Since iconic objects always are represented by variables all iconic parameters only accept variables. Control output parameters must always contain variables, too, as they store the results of an operator evaluation.

5.2 Control Types and Constants

All non-iconic data is represented by so called *control data* (numerical/string) in HDevelop. The name is derived from their respective functions within HALCON operators where they *control* the behaviour (the effect) of image processing operators (e.g., thresholds for a segmentation operator). Control parameters in HDevelop may contain arithmetic or logical operations. A control data item can be of one of the following types: `integer`, `real`, `boolean`, and `string`.

- The types `integer` and `real` are used under the same syntactical rules as in C. Integer numbers can be input in the standard decimal notation, in hexadecimal by prefixing the number with `0x`, and in octal by prefixing the number with `0`. For example:

```
4711
-123
0xfeb12
073421
73.815
0.32214
.56
-17.32e-122
32E19
```

  Data items of type `integer` or `real` are converted to their machine-internal representations: `real` becomes the C-type `double` (8 bytes) and `integer` becomes the C-type `long` (4 or 8 bytes).

- A String (`string`) is a sequence of characters that is enclosed in single quotes (`'`). The maximum string length is limited to 1024 characters. Special characters, like the line feed, are represented in the C-like notation, as you can see in table 5.2 (see the reference of the C language for comparison). Examples of strings are shown in table 5.3.

- The constants `true` and `false` belong to the type `boolean`. The value `true` is internally represented by the number 1 and the value `false` by 0. This means, that in the expression `Val := true` the effective value of `Val` is set to 1. In general, every integer value $\neq 0$ means `true`. Please note that some HALCON operators take logical values for input (e.g., `set_system`). In this case the HALCON operators expect string constants like `'true'` or `'false'` rather than the represented values `true` or `false`.

Meaning	Abbreviation	Notation
line feed	NL (LF)	\n
horizontal tabulator	HT	\t
vertical tabulator	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
bell	BEL	\a
backslash	\	\\
single quote	'	\'

Table 5.2: *Surrogates for special characters*

String	Meaning
'Hugo'	letters
'10.9'	numbers (not real)
'Text...\n'	NL at the end of the string
'\t Text1 \t Text2'	two tabs in a text
'Sobel\'s edge-filter'	single quote within the text
'c:\\Programs\\MVTec\\Halcon\\images'	Directory

Table 5.3: *String examples*

- There are constants for the return value (result state) of an operator. The constants can be used together with the operator dev_error_var and dev_set_check. These constants represent the normal return value of an operator, so called *messages*. For errors no constants are available[1]. In table 5.4 all return messages can be found.

Constant	Meaning	Value
H_MSG_TRUE	No error; for tests: (true)	2
H_MSG_FALSE	For tests: false	3
H_MSG_VOID	No result could be computed	4
H_MSG_FAIL	Operator did not succeed	5

Table 5.4: *Return values for operators*

The control types are only used within the generic HDevelop type *tuple*. A tuple of length 1 is interpreted as an atomic value. A tuple may consist of several numerical data items with *different* types. The standard representation of a tuple is a listing of its elements included into brackets (see figure 5.1).
[] specifies the empty tuple. A Tuple with just one element is to be considered as a special case, because it can either be specified in the tuple notation or as an atomic value: [55] defines the same constant as 55. Examples for tuples are:

[1]There exist more than 400 error numbers internally (see C-Interface Manual).

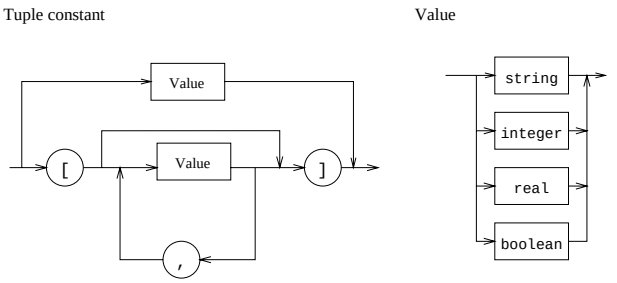


Figure 5.1: *The syntax of tuple constants.*

```
[]
4711
0.815
'Text'
[16]
[100.0,100.0,200.0,200.0]
['FileName','Extension']
[4711,0.815,'Hugo']
```

The maximum length of a tuple is limited to 1000000.

5.3 Variables

Names of variables are built up as usual by composing letters, digits and the underscore `'_'`. The maximum length of a variable name is limited to 256 characters. The kind of a variable (iconic or control variable) depends on its position in the parameter list in which the variable identifier is used for the first time (see also chapter 5.1). The kind of the variable is determined during the input of the operator parameters: whenever a new identifier appears, a new variable with the same identifier is created. Control and iconic variables must have different names. The value of a variable (iconic or control) is undefined until the first assignment defines it (the variable hasn't been instantiated yet). A read access to an undefined variable leads to a runtime error (`Variable <x> not instantiated`).
Instantiated variables contain tuples of values. Depending on the kind of the variable the data items are either iconic objects or control data. The length of the tuple is determined dynamically by the performed operation. A variable can get new values any number of times, but once a value has been assigned the variable will always keep beeing instantiated, unless you select the menu item `Execute ▷ Reset Program`. The content of the variable is deleted before the variable is assigned with new values.
The concept of different kinds of variables allows a first ("coarse") typification of variables (control or iconic data), whereas the actual type of the data (e.g., `real`, `integer`, `string`, etc.) is undefined until the variable gets assigned with a concrete value. Therefore, it is possible that the type of a new data item differs from that of the old.

5.4 Operations on Iconic Objects

Iconic objects are exclusively processed by HALCON operators. HALCON operators work on tuples of iconic objects, which are represented by their surrogates in the HALCON data management. The results of those operators are again tuples of iconic objects or control data elements. For a detailed description of the HALCON operators refer to the HALCON reference manual and the remarks in chapter 5.5.3.

5.5 Expressions for Input Control Parameters

In HDevelop, the use of expressions is limited to control input parameters; all other kinds of parameters must be assigned by variables.

5.5.1 General Features of Tuple Operations

This chapter is intended to give you a short overview over the features of tuples and their operations. A more detailed description of each operator mentioned here is given in the following sections.
Please note that in all following tables variables and constants have been substituted by letters. These letters give information about possible limitations of the areas of definition. A single letter (inside these tables) represents a data type. Operations on these symbols can only be applied to parameters of the indicated type or to expressions that return a result of the indicated type. To begin with, table 5.5 specifies the names and types of the symbolic names.

Symbol	Types
`i`	`integer`
`a`	arithmetic, that is: `integer` or `real`
`l`	`boolean`
`s`	`string`
`v`	all types (atomic)
`t`	all types (tuple)

Table 5.5: *Symbolic variables for the operation-description*

Operators are normally described for atomic tuples (tuples of length 1). If the tuple contains more than one element, most operators work as follows:

- If one of the tuples is of length one, all elements of the other tuples are combined with that single value for the chosen operation.
- If both tuples have a length greater than one, both tuples must have the same length (otherwise a runtime error occurs). In this case, the selected operation is applied to all elements with the same index. The length of the resulting tuples is identical to the length of the input tuples.
- If one of the tuples is of length 0 (`[]`), a runtime error occurs.

In table 5.6 you can find some examples for arithmetic operations with tuples. In this example you should pay special attention to the order in which the string concatenations are performed.

Input	Result
`5 * 5`	25
`[5] * [5]`	25
`[1,2,3] * 2`	`[2,4,6]`
`[1,2,3] * 2.1 + 10`	`[12.1,14.2,16.3]`
`[1,2,3] * [1,2,3]`	`[1,4,9]`
`[1,2,3] * [1,2]`	runtime error
`'Text1' + 'Text2'`	`'Text1Text2'`
`17.23 + ' Text'`	`'17.23 Text'`
`'Text1 ' + 99 + ' Text2'`	`'Text1 99 Text2'`
`'Text ' + 3.1 * 2`	`'Text 6.2'`
`3.1 * (2 + ' Text')`	runtime error
`3.1 + 2 + ' Text'`	`'5.1 Text'`
`3.1 + (2 + ' Text') )`	`'3.12 Text'`
`'Text ' + 2.1 + 3`	`'Text 2.13'`

Table 5.6: *Examples for arithmetic operations with tuples and strings*

5.5.2 Assignment

In HDevelop, an assignment is treated like an operator. To use an assignment you have to select the operator `assign (Input, Result)`. This operator has the following semantics: It evaluates `Input` (right side of assignment) and stores it in `Result` (left side of assignment). However, in the program text the assignment is represented by the usual syntax of the assignment operator: `':='`. The following example outlines the difference between an assignment in C syntax and its transformed version in HDevelop:
The assignment in C syntax

```
u = sin(x) + cos(y);
```

is defined in HDevelop using the assignment operator as

```
assign(sin(x) + cos(y), u)
```

which is displayed in the program window as:

```
u := sin(x) + cos(y)
```

If the result of the expression doesn't need to be stored into a variable, the expression can directly be used as input value for any operator. Therefore, an assignment is necessary only if the value has to be used several times or if the variable has to be initialized (e.g., for a loop).
A second assignment operator is available: (`insert(Input,Value,Index,Result)`). It is used to assign tuple elements. If the first input parameter and the first output parameter are identical, the call:

```
insert (Areas, Area, Radius-1, Areas)
```

is not presented in the program text as an operator call, but in the more intuitive form as:

```
Areas[Radius-1] := Area.
```

To construct a tuple with `insert`, normally an empty tuple is used as initial value and the elements are inserted in a loop:

```
Tuple = []
for i := 0 to 5 by 1
  Tuple[i] = sqrt(real(i))
endfor
```

As you can see from this example, the indices of a tuple start at 0.
An insertion into a tuple can generally be performed in one of the following ways:

(i) In case of appending the value at the 'back' or at the 'front', the concatenation can be used. Here the `assign` operator is used with the following parameters:

```
assign([Tuple,NewVal],Tuple)
```

which is displayed as

```
Tuple := [Tuple,NewVal]
```

(ii) If the index position is somewhere in between, the operator `insert` has to be used. It takes the following arguments as input: first the tuple in which the new value should be inserted; then the new value and after that the index position as the third input parameter. The result (the fourth parameter) is almost identical with the input tuple, except of the new value at the defined index position.

In the following example regions are dilated with a circle mask and afterwards the areas are stored into the tuple `Areas`. In this case the operator `insert` is used.

```
read_image (Mreut, 'mreut')
threshold (Mreut, Region, 190, 255)
Areas  := []
for  Radius := 1 to 50 by 1
  dilation_circle (Region, RegionDilation, Radius)
  area_center (RegionDilation, Area, Row, Column)
  Areas[Radius-1] := Area
endfor
```

Please note that first the variable `Areas` has to be initialized in order to avoid a runtime error. In the example `Areas` is initialized with the empty tuple (`[]`). Instead of `insert` the operator `assign` with tuple concatenation

```
Areas := [Areas,Area]
```

could be used, because the element is appended at the back of the tuple.
More examples can be found in the program `assign.dev`.

[t,t]	concatenation of tuples
|t|	number of elements
t[i]	selection of an element
t[i:i]	selection of (a part of) a tuple

Table 5.7: *Basic operations on tuples*

5.5.3 Basic Tuple Operations

A basic tuple operation may be selecting one or more values, combining tuples (concatenation) or reading the number of elements.
The concatenation accepts one or more variables or constants as input. They are all listed between the brackets, separated by commas. The result again is a tuple.
[t_1,t_2] is the concatenation of tuple t_1 and t_2. Example:

```
[[5,'Text'],[5.9]] ↦ [5,'Text',5.9]
```

So even the following holds: `[[t]] = [t] = t`.
`|t|` returns the number of elements of a tuple. The indices of elements range from zero to the number of elements minus one (i.e., `|t|-1`). Therefore, the selection index has to be within this range.[2]

```
Tuple := [V1,V2,V3,V4]
for i := 0 to |Tuple|-1 by 1
  fwrite_string (FileHandle,Tuple[i]+'\n')
endfor
```

There are no direct operations on parameter position for the handling of iconic variables. This means that single iconic objects cannot be directly selected using [] and their number cannot be directly determined using []. For this purpose, however, HALCON operators are offered which carry out the equivalent tasks. In table 5.8 you can see tuple operations that work on control data and their counterparts that work on iconic data. In the table the symbol `t` represents a control tuple, and the symbols `p` and `q` represent iconic tuples. Further examples can be found in the program `tuple.dev`.

control	iconic
[]	empty_obj ()
|t|	count_obj (p, num)
[t1,t2]	concat_obj (p1, p2, q)
t[i]	select_obj(p, q, i+1, 1)
t[i,j]	copy_obj(p, q, i+1, j-i+1)

Table 5.8: *Equivalent tuple operations for control and iconic data*

[2] Please note that the index of objects (e.g., `select_object`) ranges from 1 to the number of elements.

5.5.4 Tuple Creation

The simplest way to create a tuple, as mentioned in section 5.2, is the use of constants together with the `assign` operator:

```
assign([],empty_tuple)
assign(4711,one_integer)
assign([4711,0.815],two_numbers)
```

This code is displayed as

```
empty_tuple := []
one_integer := 4711
two_numbers := [4711,0.815]
```

This is useful for constant tuples with a fixed (small) length. More general tuples can be created by successive application of the concatenation or the `insert` function together with variables, expressions or constants. If we want to generate a tuple of length 100, where each element has the value 4711, it might be done like this:

```
assign([],tuple)
for i := 1 to 100 by 1
  assign([tuple,4711],tuple)
```

which is transformed to

```
tuple := []
for i := 1 to 100 by 1
  tuple := [tuple,4711]
```

Because this is not very convenient a special function called `gen_tuple_const` is available to construct a tuple of a given length, where each element has the same value. Using this function, the program from above is reduced to:

```
assign(gen_tuple_const(100,4711),tuple)
```

which is displaed as

```
tuple := gen_tuple_const(100,4711)
```

If we want to construct a tuple with the same length as a given tuple there are two ways to get an easy solution, The first one is based on `gen_tuple_const`:

```
assign(gen_tuple_const(|tuple_old|,4711),tuple_new)
```

which is displayed as

```
tuple_new := gen_tuple_const(|tuple_old|,4711)
```

The second one is a bit tricky and uses arithmetic functions:

```
assign((tuple_old * 0) + 4711,tuple_new)
```

which is displaed as

```
tuple_new := (tuple_old * 0) + 4711
```

Here we get first a tuple of the same length with every element set to zero. Then we add the constant to each element.
In the case of tuples with different values we have to use the loop version to assign the values to each position:

```
assign([],tuple)
for i := 1 to 100 by 1
  assign([tuple,i*i],tuple)
```

which is displaed as

```
tuple := []
for i := 1 to 100 by 1
  tuple := [tuple,i*i]
```

In this example we construct a tuple with the square values from 1^2 to 100^2.

5.5.5 Simple Arithmetic Operations

Table 5.9 shows an overview of the available simple arithmetic operations.
All operations are left-associative, except the right-associative unary minus operator. The evaluation usually is done from left to right. However, parentheses can change the order of evaluation and some operators have a higher precedence than others (see chapter 5.5.14). The arithmetic

`a / a`	division
`a * a`	multiplication
`v + v`	addition and *concatenation* of strings
`a - a`	subtraction
`-a`	negation

Table 5.9: *Arithmetic operations*

operations in HDevelop match the usual definitions. Expressions can have any number of parentheses.
The division operator (`a / a`) can be applied to `integer` as well as to `real`. The result is of type `real`, if at least one of the operands is of type `real`. If both operands are of type `integer` the division is an integer division. The remaining arithmetic operators (multiplication, addition, subtraction, and negation) can be applied to either `integer` or `real` numbers. If at least one operand is of type `real`, the result will be a `real` number as well. In the following example.

```
V1 := 4/3
V2 := 4/3.0
V3 := (4/3) * 2.0
```

`V1` is set to 1, `V2` to 1.3333333, and `V3` to 2.0. Simple examples can be found in the program `arithmetic.dev`.

5.5.6 Bit Operations

This section describes the operators for bit processing of numbers. The operands have to be integers.

`lsh(i,i)`	left shift
`rsh(i,i)`	right shift
`i band i`	bitwise and
`i bor i`	bitwise or
`i bxor i`	bitwise xor
`bnot i`	bitwise complement

Table 5.10: *Bit operations*

The result of `lsh(i1,i2)` is a bitwise left shift of `i1` that is applied `i2` times. If there is no overflow this is equivalent to a multiplication by 2^{i2}. The result of `rsh(i1,i2)` is a bitwise right shift of `i1` that is applied `i2` times. For non-negative `i1` this is equivalent to a division by 2^{i2}. For negative `i1` the result depends on the used hardware. For `lsh` and `rsh` the result is undefined if the second operand has a negative value or the value is larger than 32. More examples can be found in the program `bit.dev`.

5.5.7 String Operations

There are several string operations available to modify, select and combine strings. Furthermore, some operations allow to convert numbers (`real` and `integer`) to strings.

`v$s`	string conversion
`v + v`	*concatenation* of strings and addition
`strchr(s,s)`	search character in string
`strstr(s,s)`	search substring
`strrchr(s,s)`	search character in string (reverse)
`strrstr(s,s)`	search substring (reverse)
`strlen(s)`	length of string
`s{i}`	selection of one character
`s{i:i}`	selection of substring
`split(s,s)`	splitting in substrings

Table 5.11: *String operations*

`$` converts numbers to strings or modifies strings. The operator has two parameters: The first one (left of the `$`) is the number that has to be converted. The second one (right of the `$`) specifies the conversion. This format string consists of the following four parts

```
<flags><field width><precision><conversion characters>
```

So a conversion might look like

```
1332.4554 $ '.6e'
```

`flags` Zero or more flags, in any order, which modify the meaning of the conversion specification. Flags may consist of the following characters:

- `-` The result of the conversion is left justified within the field.
- `+` The result of a signed conversion always begins with a sign, `+` or `-`.
- `<space>` If the first character of a signed conversion is not a sign, a space character is prefixed to the result. This means that if the space flag and `+` flag both appear, the space flag is ignored.
- `#` The value is to be converted to an "alternate form". For `d` and `s` conversions, this flag has no effect. For `o` conversion (see below), it increases the precision to force the first digit of the result to be a zero. For `x` or `X` conversion (see below), a non-zero result has `0x` or `0X` prefixed to it. For `e`, `E`, `f`, `g`, and `G` conversions, the result always contains a radix character, even if no digits follow the radix character. For `g` and `G` conversions, trailing zeros are not removed from the result, contrary to usual behavior.

`field width` An optional string of decimal digits to specify a minimum field width. For an output field, if the converted value has fewer characters than the field width, it is padded on the left (or right, if the left-adjustment flag, - has been given) to the field width.

`precision` The precision specifies the minimum number of digits to appear for the `d`, `o`, `x`, or `X` conversions (the field is padded with leading zeros), the number of digits to appear after the radix character for the e and f conversions, the maximum number of significant digits for the `g` conversion, or the maximum number of characters to be printed from a string in s conversion. The precision takes the form of a period `.` followed by a decimal digit string. A null digit string is treated as a zero.

`conversion characters` A conversion character indicates the type of conversion to be applied:

- `d,o,x,X` The integer argument is printed in signed decimal (`d`), unsigned octal (`o`), or unsigned hexadecimal notation (`x` and `X`). The `x` conversion uses the numbers and letters `0123456789abcdef`, and the `X` conversion uses the numbers and letters `0123456789ABCDEF`. The precision component of the argument specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits than the specified minimum, it is expanded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of 0 is no characters.
- `f` The floating-point number argument is printed in decimal notation in the style `[-]dddrddd`, where the number of digits after the radix character, `r`, is equal to the precision specification. If the precision is omitted from the argument, six digits are output; if the precision is explicitly 0, no radix appears.
- `e,E` The floating-point-number argument is printed in the style `[-]drddde+dd`, where there is one digit before the radix character, and the number of digits after it is equal to the precision. When the precision is missing, six digits are produced; if the

precision is 0, no radix character appears. The `E` conversion character produces a number with `E` introducing the exponent instead of `e`. The exponent always contains at least two digits. However, if the value to be printed requires an exponent greater than two digits, additional exponent digits are printed as necessary.

`g,G` The floating-point-number argument is printed in style `f` or `e` (or in style `E` in the case of a `G` conversion character), with the precision specifying the number of significant digits. The style used depends on the value converted; style `e` is used only if the exponent resulting from the conversion is less than -h or greater than or equal to the precision. Trailing zeros are removed from the result. A radix character appears only if it is followed by a digit.

`s` The argument is taken to be a string, and characters from the string are printed until the end of the string or the number of characters indicated by the precision specification of the argument is reached. If the precision is omitted from the argument, it is interpreted as infinite and all characters up to the end of the string are printed.

`b` Similar to the s conversion specifier, except that the string can contain backslash-escape sequences which are then converted to the characters they represent.

In no case does a nonexistent or insufficient field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result.

Examples for the string conversion can be found in the program `string.dev`.
The string concatenation (+) can be applied in combination with strings or all numerical types; if necessary, the operands are first transformed into strings (according to their standard representation). At least one of the operands has to be already a string so that the operator can act as a string concatenator. In the following example a filename (e.g., `'Name5.tiff'`) is generated. For this purpose two string constants (`'Name'` and `'.tiff'`) and an integer value (the loop-index `i`) are concatenated:

```
for i := 1 to 5 by 1
  read_image (Bild, 'Name'+i+'.tiff')
endfor
```

`str(r)chr(s1,s2)` returns the index of the first (last) as a tuple occurrence of one character in `s2` in string `s1`, or `-1` if none of the characters occurs in the string.
`str(r)str(s1,s2)` returns the index of the first (last) occurrence of string s2 in string s1, or `-1` if s2 does not occur in the string.
`strlen(s)` returns the number of characters in `s`.
`s{i}` returns the character at index position `i` in `s`. The index ranges from zero to the length of the string minus 1. The result of the operator is a string of length one.
`s{i1:i2}` returns all characters from index position `i1` up to position `i2` in `s` as a string. The index ranges from zero to the length of the string minus 1.
`split(s1,s2)` devides the string `s1` into single substrings. The string is split at those positions where it contains a character from `s2`. As an example the result of

```
split('/usr/image:/usr/proj/image',':')
```

consists of the two strings

```
['/usr/image','/usr/proj/image']
```

5.5.8 Comparison Operators

In HDevelop, the comparison operators are defined not only on atomic values, but also on tuples with an arbitrary number of elements. They always return values of type `boolean`. Table 5.12 shows all comparison operators.

`t < t`	less than
`t > t`	greater than
`t <= t`	less or equal
`t >= t`	greater or equal
`t = t`	equal
`t # t`	not equal

Table 5.12: *Comparison operators*

`t = t` and `t # t` are defined on all types. Two tuples are equal (`true`), if they have the same length and all the data items on each index position are equal. If the operands have different types (`integer` and `real`), the integer values are first transformed into `real` numbers. Values of type `string` cannot be mixed up with numbers, i.e., `string` values are considered to be not equal to values of other types.

1st Operand	2nd Operand	Operation	Result
`1`	`1.0`	`=`	`true`
`[]`	`[]`	`=`	`true`
`''`	`[]`	`=`	`false`
`[1,'2']`	`[1,2]`	`=`	`false`
`[1,2,3]`	`[1,2]`	`=`	`false`
`[4711,'Hugo']`	`[4711,'Hugo']`	`=`	`true`
`'Hugo'`	`'hugo'`	`=`	`false`
`2`	`1`	`>`	`true`
`2`	`1.0`	`>`	`true`
`[5,4,1]`	`[5,4]`	`>`	`true`
`[2,1]`	`[2,0]`	`>`	`true`
`true`	`false`	`>`	`true`
`'Hugo'`	`'hugo'`	`<`	`true`

Table 5.13: *Examples for the comparison of tuples*

The four comparison operators compute the lexicographic order of tuples. On equal index positions the types must be identical, however, values of type `integer`, `real` and `boolean` are adapted automatically. The lexicographic order applies to strings, and the `boolean` `false` is considered to be smaller than the boolean `true` (`false < true`). In the program `compare.dev` you can find examples for the comparison operators.

5.5.9 Boolean Operators

The boolean operators `and`, `or`, `xor` and `not` are defined only for tuples of length 1. `1 and 1` is set to `true` (1) if both operands are `true` (1), whereas `1 or 1` returns `true` (1) if at least one of

`not l`	negation
`l and l`	logical 'and'
`l or l`	logical 'or'
`l xor l`	logical 'xor'

Table 5.14: *Boolean operators*

the operands is `true` (1). `l xor l` return `true` (1) if exactly one of both operands is true. `not l` returns `true` (1) if the input is `false` (0), and `false` (0), if the input is `true` (1).

5.5.10 Trigonometric Functions

All these functions work on tuples of numbers as arguments. The input can either be of type `integer` or `real`. However, the resulting type will be of type `real`. The functions are applied to all tuple values and the resulting tuple has the same length as the input tuple. For `atan2` the two input tuples have to be of equal length. Table 5.15 shows the provided trigonometric functions.

`sin(a)`	sine of `a`
`cos(a)`	cosine of `a`
`tan(a)`	tangent of `a`
`asin(a)`	arc sine of `a` in the intervall $[-\pi/2, \pi/2]$, `a` $\in [-1, 1]$
`acos(a)`	arc cosine `a` in the intervall $[-\pi/2, \pi/2]$, `a` $\in [-1, 1]$
`atan(a)`	arc tangent `a` in the intervall $[-\pi/2, \pi/2]$, `a` $\in [-1, 1]$
`atan2(a,b)`	arc tangent `a`/`b` in the intervall $[-\pi, \pi]$
`sinh(a)`	hyperbolic sine of `a`
`cosh(a)`	hyperbolic cosine of `a`
`tanh(a)`	hyperbolic tangent of `a`

Table 5.15: *Trigonometric functions*

For the trigonometric functions the angle is specified in radians.

5.5.11 Exponential Functions

All these functions work on tuples of numbers as arguments. The input can either be of type `integer` or `real`. However, the resulting type will be of type `real`. The functions are applied to all tuple values and the resulting tuple has the same length as the input tuple. For `pow` and `ldexp` the two input tuples have to be of equal length. Table 5.16 shows the provided exponential functions.

5.5.12 Numerical Functions

The functions `min` and `max` select the minimum and the maximum values of the tuple values. All values either have to be of type `string`, or `integer`/`real`. It is not allowed to mix strings with numerical values. The resulting value will be of type `real`, if at least one of the elements is of type `real`. If all elements are of type `integer` the resulting value will also be of type

exp(a)	exponential function e^{a}
log(a)	natural logarithm $\ln(a)$, a> 0
log10(a)	decadic logarithm, $\log_{10}(a)$, a> 0
pow(a1,a2)	$a1^{a2}$
ldexp(a1,a2)	$a1 \cdot 2^{a2}$

Table 5.16: *Exponential functions*

integer. The same applies to the function sum that determines the sum of all values. If the input arguments are strings, string concatenation will be used instead of addition.

min(t)	minimum value of the tuple
max(t)	maximum value of the tuple
sum(t)	sum of all elements of the tuple or string concatenation
mean(a)	mean value
deviation(a)	standard deviation
sqrt(a)	square root $\sqrt{a}$
deg(a)	convert radians to degrees
rad(a)	convert degrees to radians
real(a)	convert integer to real
round(a)	convert real to integer
abs(a)	absolute value of a (integer or real)
fabs(a)	absolute value of a (always real)
ceil(a)	smallest integer value not smaller than a
floor(a)	largest integer value not greater than a
fmod(a1,a2)	fractional part of a1/a2, with the same sign as a1

Table 5.17: *Numerical functions*

The functions sqrt, mean, deviation, deg, rad, fabs, ceil, floor and fmod can work with integer and real; the result is always of type real.
The function mean calculates the mean value and deviation the standard deviation of numbers.
sqrt calculates the square root of a number.
deg and rad convert numbers from radians to degrees and from degrees to radians, respectively.
The function round always returns an integer value and the function abs always returns the absolut value that is of the same type as the input value.
real converts an integer to a real. For real as input it returns the input.
round converts a real to an integer and rounds the value. For integer it returns the input.
The following example (filename: euclid_distance.dev) shows the use of some numerical functions:

```
V1 := [18.8,132.4,33,19.3]
V2 := [233.23,32.786,234.4224,63.33]
Diff := V1 - V2
Distance := sqrt(sum(Diff * Diff))
```

```
Dotvalue := sum(V1 * V2)
```

First, the Euclidian distance of the two vectors `V1` and `V2` is computed, by using the formula:

$$d = \sqrt{\sum_i (V1_i - V2_i)^2}$$

The difference and the multiplication (square) are successively applied to each element of both vectors. Afterwards `sum` computes the sum of the squares. Then the square root of the sum is calculated. After that the dot product of `V1` and `V2` is determined by the formula:

$$\langle V1, V2 \rangle = \sum_i (V1_i * V2_i)$$

5.5.13 Miscellaneous Functions

`sort(t)`	sorting in increasing order
`sort_index(t)`	return index instead of values
`inverse(t)`	reverse the order of the values
`number(v)`	convert `string` to a number
`is_number(v)`	test if value is a number
`environment(s)`	value of an environment variable
`ord(a)`	ASCII number of a character
`chr(a)`	convert an ASCII number to a character
`ords(s)`	ASCII number of a tuple of strings
`chrt(i)`	convert a tuple of integers into a string

Table 5.18: *Miscellaneous functions*

`sort` sorts the tuple values in ascending order, that means, that the first value of the resulting tuple is the smallest one. But again: strings must not be mixed up with numbers. `sort_index` sorts the tuple values in ascending order, but in contrast to `sort` it returns the index positions (0..) of the sorted values.
The function `inverse` reverses the order of the tuple values. Both `sort` and `inverse` are the identity operation, if the input is empty, if the tuple is of length 1, or if the tuple contains only one value in all positions, e.g., [1,1,...,1].
`number` converts a `string` to an `integer` or a `real` depending on the representation. For `integer` and `real` it returns the input.
`is_number` returns `true` for numbers and `string` if it represents a number.
`environment` returns the value of an environment variable. Input is the name of the environment variable as a string.
`ord` gives the ASCII number of a character as an `integer`. `chr` converts an ASCII number to a character.
`ords` converts a tuple of strings into a tuple of (ASCII) integers. `chrt` converts a tuple of integers into a string.

5.5.14 Operator Precedence

Table 5.19 shows the precedence of the operators for control data. Some operations (like functions, `| |`, `t[]`, etc.) are left out, because they mark their arguments clearly.

or, xor, bor, bxor
and , band
#, =
<=, >=, <, >
+, -
/, *
- (unary minus), ~, !
$

Table 5.19: *Operator precedence (increasing from top to bottom)*

5.6 Reserved Words

The strings shown in table 5.20 are reserved words and their usage is strictly limited to their predefined meaning. They cannot be used as variables.

true	false	and	or
xor	bor	bxor	chr
ord	chrt	ords	band
bnot	not	sum	sin
cos	tan	asin	acos
atan	sinh	cosh	tanh
exp	log	log10	ceil
floor	atan2	pow	fabs
abs	fmod	ldexp	round
deg	rad	min	max
sort	inverse	strlen	strchr
strrchr	strstr	strrstr	split
environment	is_number	number	real
lsh	rsh	deviation	mean
sqrt	gen_tuple_const	H_MSG_TRUE	H_MSG_FALSE
H_MSG_FAIL	H_MSG_VOID		

Table 5.20: *Reserved words*

5.7 Control Structures

HDevelop provides the following constructs to structure programs:

- The simplest control structure is `if`. The condition contains a boolean expression. If the condition is true, the body is executed. Otherwise the execution is continued at the first expression or operator call that follows the word `endif`.

```
if (<Condition>)
  ...
endif
```

- Another simple control structure is the condition with alternative. If the condition is true, all expressions and calls between the head and the word `endif` are performed. If the condition is false the part between `else` and `endif` is executed. Note that the operator is called `ifelse` and it is displayed as `if` in the program text area.

```
if (<Condition>)
  ...
else
  ...
endif
```

- The `while` loop has a boolean expression as the conditional part. As long as it is `true`, the body of the loop is performed. In order to enter the loop, the condition has to be true in the first place.

```
while (<Condition>)
 ...
endwhile
```

  In chapter 7.3 you can find an example for using the `while` loop.

- The `for` loop is controlled by a start and termination value and an incrementation value that determines the number of loop steps. These values may also be expressions which are evaluated immediately before the loop is entered. The expressions may be of type `integer` or of type `real`. If all input values are of type `integer` the loop variable will also be of type `integer`. In all other cases the loop variable will be of type `real`. If the start value is less or equal to the termination value, the loop index is assigned with the starting value and the body of the loop is entered. If the increment is less than zero the loop is entered if the start value is larger or equal to the end value. Each time the body is executed, the loop index is incremented by the incrementation value. If the loop index is equal to the termination value, the body of the loop is performed for the last time. If the loop index is larger than the termination value the body will not be excecuted any longer.[3] Please note that it is not necessary, that the loop index has to be equal to the termination value before terminating the loop. The loop index is set to the termination value when the loop is being left. Please note, that the expressions for start and termination value are evaluated only once when *entering the loop*. A modification of a variable that appears within these expressions has no influence on the termination of the loop. The same applies to the modifications of the loop index. It also has no influence on the termination. The loop

[3] For negative increment values the loop is terminated if the loop index is less than the termination value.

value is assigned to the correct value each time the `for` operator is executed. For more details, see section 6.1.4 on the code generation of `for` loops.

If the `for` loop is left too early (e.g., if you press `Stop` and set the `PC`) and the loop is entered again, the expressions will be evaluated, as if the loop were entered for the first time.

```
for <loop value> := <Start> to <End> by <Increment>
  ...
endfor
```

In the following example the sine from 0 up to 6π is computed and printed in to the graphical window (filename: `sine.dev`):

```
old_x := 0
old_y := 0
dev_set_color ('red')
dev_set_part(0, 0, 511, 511)
for x := 1 to 511 by 1
    y := sin(x / 511.0 * 2 * 3.1416 * 3) * 255
    disp_line (WindowID, -old_y+256, old_x, -y+256, x)
    old_x := x
    old_y := y
endfor
```

In this example the assumption is made that the window is of size 512×512. The drawing is always done from the most recently evaluated point to the current point.

Further examples on how to use the `for` loop can be found in chapter 7.8 and 7.9.

- `stop` stops the program after the operator is executed. The program can be continued by pressing the `Step` or `Run` button.
- `exit` *terminates* the session of HDevelop.

5.8 Limitations

This section summarizes the restritions of the HDevelop language:

- Maximum number of objects per parameter : 100000
- Maximum length of strings : 1024 characters
- Maximum length of a variable name : 256 characters
- Maximum length of a tuple : 1000000

The general restrictions of the HALCON operators can be found in the “Getting Started” Manual.

Chapter 6

Code Generation

The idea of code generation is as follows: After developing a program according to the given requirements it has to be transferred to its final environment. Here, you often don't want to use HDevelop to save memory or disk space. In addition the program should execute as fast as possible, especially without the overhead of an interpreter. Therefore the program is transferred into another language that can be compiled and allows a faster execution. In addition to this features of the new environment like special libraries or graphical user interface builder can be used.
The aim of this chapter is to describe the general steps of program development using this feature. Later on some details of the code generation and optimization aspects are discussed.

6.1 Code Generation for C++

In this section the use of HALCON in C++ is describes.

6.1.1 Basic Steps

Interactive Development

The first step is the well known process of program development in HDevelop as it is described in the rest of this manual.

Program Export

The next step is to export the program using the menu `File` ▷ `Save As`. Here you have to select the language (C++) and save it to file. In UNIX you specify the file by giving it the corresponding extension, which is "`.cpp`". A file will be created that contains the HDevelop program as C++ source code in a procedure `action()`. This procedure is called in function `main()`. Besides the program code, the file contains all necessary `#include` instructions. All variables (iconic as well as control) are declared locally in the procedure `action()`. Iconic variables belong to the class `Hobject` and all other variables belong to `HTuple`.

Compiling and Linking

The last step is now to compile and link this new program. In the case of UNIX this can be done using the predefined `makefile` which can be found in the directory

`$HALCONROOT/examples/C++`. To compile and link a program called `test.cpp`, you call the `makefile` like

```
make PROG_NAME=test
```

or set the variable `PROG_NAME` in `makefile` to `test` and then just type `make`. In the Windows NT environment, Visual C++ is used for the compiling and linking. For details see the User's Manual of HALCON/C++.

6.1.2 Optimization

Optimization might be necessary for variables of class `HTuple`. This kind of optimization can either be done in HDevelop or in the generated C++ code. In most cases optimization is not necessary if you program according to the following rules.

(i) Using the tuple concatenation, it is more efficient to extend a tuple at the "right" side, like:

```
T := [T,New]
```

because this can the transformed to

```
T.Append(New);
```

in C++ and requires no creation of a new tuple, whereas

```
T := [New,T]
```

which is transferred to

```
T = New.Append(T);
```

would need the creation of a new tuple.

(ii) Another good way to modify a tuple is the operator `insert` (see section 5.5.2). In this case HDevelop code like

```
T[i] := New
```

can directly be transferred to the efficient and similar looking code

```
T[i] = New;
```

6.1.3 Used Classes

There are only two classes that are used: `HTuple` for control parameters and `Hobject` for iconic data. There is no need for other classes as long as the program has the same functionality as in HDevelop. When editing a generated program you are free to use any of the classes of HALCON/C++ to extend the functionality.

6.1.4 Limitations and Troubleshooting

Besides the restrictions mentioned in the next section, please check also the description of the HDevelop operators on page 50.

Assignment

In HDevelop each time a new value is assigned to a variable its old contents are removed. This is also the case for iconic objects (`Hobject`). The class `HTuple` also has a destructor which removes the data stored inside a tuple. But problems arise if a tuple contains a handle (for example file, windows, ocr, etc.). In this case, the memory of the handle, but not the data which it points at, is automatically removed. Using C++, this data has to be removed explicitly by calling the corresponding operators called `close_*` like `close_ocr` or `close_ocv`. That means that the `close_*` operators for all handles in use have to be called

- before a new value is assigned to a handle and
- at the end of the program.

`for` - Loops

HDevelop and C++ have different semantics for loops, which can cause confusion. Because the problems are so seldom and the generated code would become very difficult to understand otherwise, the code generation ignores the different semantics. Now, what are these differences:

(i) In C++ you can modify the loop variable (e.g., by setting it to the end value of the condition) to terminate the loop. This can't be done in HDevelop, because here the current value is stored "inside" the `for`-operator and is automatically updated when it is executed again.

(ii) In C++ you can modify the step range if you use a variable for the increment. This is also not possible with HDevelop because the increment is stored "inside" the `for`-operator when the loop is entered.

(iii) The last difference concerns the value of the loop variable after exiting the loop. In C++ it has the value with which the condition becomes false for the first time. In HDevelop it contains the end value, which was calculated when the loop was entered.

Looking at the mentioned points we recommend to do the programming according to the following rules:

(i) Don't modify the loop variable or the step value inside the loop. If you need this behaviour use the `while`-loop.

(ii) Don't use the loop variable after the loop.

Automatic Display

Normally, the result of every operator is displayed in the graphics window of HDevelop. This is not the case when using an exported C++ program. It behaves like the HDevelop program running with the options: "update window = off". That means, whenever you want to have data to be displayed you must explicitly call the operator `dev_display`. This will be converted to the appropriate C++ call in the exported code. Of course, you have to open a window (e.g. via `dev_open_window`) before displaying any data.

Exception Handling

In HDevelop, every exception normally causes the program to stop and report an error message in a dialog window. This might not be useful in C++. In addition there are different default behaviours concerning the result state of operators.

Messages In the case of C++ only severe errors cause an exception handling which terminates the program and prints an error message. This might cause problems with minor errors, so called *messages* in HALCON. These messages are handled as return values of the operators and can have the following values, which are also available in HDevelop as constants:

```
H_MSG_TRUE
H_MSG_FALSE
H_MSG_FAIL
H_MSG_VOID
```

One of these messages is always returned indicating the status of the operator. Normally, the result is `H_MSG_TRUE`. Some operators return `H_MSG_FAIL` like `read_image` or `read_region` to indicate that they could not open a file or there was no permission to read it. In this case the programmer has to check the return value and apply some adequate action. If the message `H_MSG_FALSE` is ignored, errors like

```
Halcon Error #4056: Image data management: object-ID is NULL
```

will happen in successive operators, because the predecessor operator did not calculate an appropriate value.

Errors In the case of hard errors (i.e., no message as described above) the program stops with an error message. To prevent this behaviour the HDevelop operators `dev_error_var` and `dev_set_check` can be used to control the exception handling in the application. This works similarly in HDevelop and C++. One difference is caused by the dynamic evaluation of `dev_error_var` in HDevelop. That means that each time the operator is executed (e.g., in a loop) the use of the error variable might change. In contrast to this, in C++ special code is added to store the return values of operators. This code will therefore be static and cannot change during program excecution. To understand how the code generation works let us have a look at a short example. Here at first the HDevelop program:

```
dev_set_check('~give_error')
dev_error_var(error,true)
threshold(image,region,100,255)
```

```
dev_error_var(error,false)
if (error # H_MSG_TRUE)
  write_string(WindowId,'error number = ' + error)
  exit()
endif
dev_set_check('give_error')
```

This program will be translated to

```
HTuple error;
::set_check("~give_error");
error = ::threshold(image,&region,100,255);
if (error != 2)
{
  ::write_string(WindowId,HTuple("error number = ") + HTuple(error));
  exit(1);
}
::set_check("give_error");
```

As can be seen, the operator `dev_error_var` is eliminated and replaced by the use of the error variable later on.
The points mentioned above might cause these two problems:

- If the second parameter of `dev_error_var` cannot be derived from the program (because no constant `false` or `true` are used but expressions, the value will be interpreted as `true`, that means: “start to use the variable”. To avoid confusion use only the constants `false` or `true` as values for the second parameter.
- The usage of a variable starts after the first call of `dev_error_var(ErrVariable,true)`. In C++ this means that all successive lines (i.e., lines “below”), until the first `dev_error_var(ErrVariable,false)` will have the assignment to `ErrVariable`. This might lead to a different behaviour compared with HDevelop, if `dev_error_var` is called inside a loop, because here the operators inside the loop before `dev_error_var` might also use `ErrVariable` after the second excecution of the loop body. Therefore: Try not to use `dev_error_var` inside a loop. Use it right at the beginning of the program.

Wrong Window-ID

In HDevelop, no window identifiers for most graphic operators are used, because windows are simply inspection tools during program development and the concept of “active” windows is used. But the window operators of the HALCON/C++ library require a window identifier. Therefore it is added automatically to each operator call. But in the case of multiple HDevelop windows the mechanism might fail, especially if the `Activate`-button was used during program execution.

Compiler errors

Sometimes it happens that messages like

```
CC: "./example.cpp", line 17: bad operands for *: int * HTuple
```

or

```
CC: "./example.cpp", line 17: error ambiguous call
CC: "./example.cpp", line 17: choices of HTuple::operator *():
CC: "./example.cpp", line 17:       HTuple::operator *(const HTuple&) const;
CC: "./example.cpp", line 17:       HTuple::operator *(double) const;
CC: "./example.cpp", line 17:       HTuple::operator *(int) const;
```

are reported by the compiler. Both errors are caused by conflicting operators. In this case one either has to change the HDevelop or the C++ program. To understand how, let us look at the code which caused the errors above.[1] For the first error the C++ program would look like this:

```
HTuple  T1,T2;
T1 = 2 * T2;
```

Because there is no operator `int * HTuple` a compiler error is given. This error can be handled in two ways:

(i) Do appropriate type casting in C++: `T1 = HTuple(2) * T2;`

(ii) Change the order of the Operands in HDevelop and export the program again: `T1 = T2 * 2;`

Both changes will do. The first one would be used by the code generation anyway.
The second error mentioned above is caused by a similar reason. The program might look like this:

```
HTuple  T1,T2;
long    val;
T1 = T2 * val;
```

In this case `val` is a `long` variable and there is no multiplication available for the type `long` in the class `HTuple`. So again we have to change the program slightly by adding the cast Operator:

```
HTuple  T1,T2;
long    val;
T1 = T2 * HTuple(val);
```

System Parameters

You should know that HDevelop performs some changes of system parameters of HALCON by calling the operator `set_system` (see reference manual). This might cause the C++ program not to produce identical output. If such a problem arises, you may query the system parameters by means of `get_system` in HDevelop after or while running the original HDevelop version of the program. Depending to the problem, you can now modify relevant parameters by explicitly calling the operator `set_system` in your own C++ program.

[1] Both concrete errors shown above are hypothetical, as they would be avoided by the automatic code generation in this special case. But they are good examples for similar errors that might be caused by conflicting operators.

Graphics Windows

The graphics windows of HDevelop and the basic window of the HALCON/C++ library have different functionality. Because the HDevelop windows are not available in C++, one is restricted to the basic windows. To get more information about both types of windows look at section 4.7 in this manual and read the description of `open_window` in the HALCON/C++ reference manual.

6.2 Code Generation for Visual Basic

This section describes the export of an HDevelop program to Visual Basic. HALCON can be used together with Visual Basic based on the COM interface of HALCON. The description of the interface can be found in the HALCON/COM interface User's Manual.

6.2.1 Basic Steps

Interactive Development

The first step is the well known process of program development in HDevelop as it is described in the rest of this manual.

Export

The next step is to export the program using the menu `File` ▷ `Save As`. Here you have to select the language (`Visual Basic`) and save it to file. In UNIX you specify the file by giving it the corresponding extension, which is "`.bas`". The result is a new file with the given name and the extension "`.bas`".

Visual Basic Template

The exported file is intended to be used together with the predefined Visual Basic project that can be found in

```
%HALCONROOT%\examples\vb\HDevelopTemplate
```

This template has to be loaded by Visual Basic. The project contains a form with a display window (`HWindowXCtrl`) and a `Run`-button. The file generated by HDevelop has to be added to this project. This is done by using the menu `Project` ▷ `Add Module` ▷ `Existing` and selecting the file. Now the project is ready for execution: Run the project and then press the `Run`-button of the form which will call the exported code.

6.2.2 Program Structure

The file created by HDevelop consists of the subroutine `RunHalcon()` which corresponds to the original program. In addition to this another subroutine is created with the name `InitHalcon()`. This subroutine applies the same initializations which HDevelop does.
Most of the variables (iconic as well as control) are declared locally in the subroutine `RunHalcon()`. Iconic variables inside `RunHalcon()` belong to the class `HUntypedObjectX`

and control variables belong to `Variant`. The subroutine `RunHalcon()` has a parameter `Window` which is of type `HWindowX`. This is the link to the window in the panel where all output operations are passed to.
In addition to this, depending on the program, additional subroutines and variables are declared.

Array Assignment

If a single value is assigned to a variant array, a special subroutine is called to ensure that the index is valid. If the array is to small the variable is resized.

Expressions

All parameter expression inside HDevelop are transferred to expressions based on the HALCON tuple operators. Therefore an expression might look somewhat complex. In many cases these expressions can be exchanged by simple Visual Basic expressions, like `tuple_sub` becomes a simple subtraction. To ensure that the exported program has the same effect in Visual Basic this exchange is not applied automatically, because the semantic is not always identical.

Stop

The HDevelop operator `stop` is transferred into a subroutine in Visual Basic which creates a message box. This message box causes the program to halt until the button is pressed.

Exit

The HDevelop operator `exit` is transformed into the Visual Basic routine `End`. Because this routine has no parameter the parameters of `exit` are suppressed.

Used Classes

There are only six classes/types that are used: `Variant` for control parameters and `HUntypedObjectX` for iconic data. In addition to this there is the container class `HTupleX` which comprises all operators of HALCON processing tuples, in this case the data type `Variant`. Then there are the classes `HWindowXCtrl` and its low level content `HWindowX`. `HWindowXCtrl` is used inside the project for the output window and a variable of class `HWindowX` directs the output to this window. Finally the class `HOperatorSetX` is used as a container for all HALCON operators. There is no need for other classes as long as the program has the same functionality as in HDevelop. When editing a generated program you are free to use any of the classes of HALCON/COM to extend the functionality.

6.2.3 Limitations and Troubleshooting

Besides the restrictions mentioned in the next section, please check also the description of the HDevelop operators on page 50.

Duplicate Parameters

Due to the parameter handling of Visual Basic / COM it is not possible to use the same variable more than once in one call. Thus for input and output parameters different variables have to be used. Also it is not possible to use the same variable twice for input or output. Examples for code which is *not* allowed are:

```
mean_image(Image,Image,3,3)
add_image(Image,Image,Add,1,0)
```

You have to introduce additional variables.

Reserved Words

In contrast to C++ and HDevelop Visual Basic has many reserved words. Thus the export adds the prefix `HXP` to all variables to avoid collisions with these reserved words.

Graphics Windows

The graphics windows of HDevelop and the basic window of the HALCON/COM library have different functionality.

- Instead of the HDevelop windows *one* window of the class `HWindowXCtrl` is used. Thus if you want to use more than one window you have to modify the code accordingly.
- The size of the window in the panel is predefined (512×512) thus it will normally not fit to your image size. Therefore you have to adapt this interactively or by using the properties of the window.
- The graphics windows in HDevelop automatically adapts to new image sizes. This is not the case in Visual Basic. Thus you have to specify the zooming using the operator `(dev_)set_part`.

To get more information about both types of windows look at section 4.7 in this manual and read the description of `open_window` in the HALCON Reference Manual.

Assignment

In HDevelop each time a new value is assigned to a variable its old contents is removed. This is also the case for iconic objects (`HUntypedObjectX`). The type `Variant` also has a destructor which removes the data stored inside a tuple. But problems arise if a tuple contains a handle (for example file, ocr, etc.). In this case, the memory of the handle, but not the data which it points at, is automatically removed. Using the exported code, this data has to be removed explicitly by calling the corresponding operators called `close_*` like `close_ocr` or `close_file`. That means that the `close_*` operators for all handles in use have to be called

- before a new value is assigned to a handle and
- at the end of the subroutine.

The ideal way would be to use the specific COM classes for this kind of data in combination with the member function. This exchange has to be done “by hand” because the export is not able to generate appropriate code.

`for` - Loops

HDevelop and Basic have different semantics for loops, which can cause confusion. Because the problems are so seldom and the generated code would become very difficult to understand otherwise, the code generation ignores the different semantics. Now, what are these differences:

(i) In Visual Basic you can modify the loop variable (e.g., by setting it to the end value of the condition) to terminate the loop. This can't be done in HDevelop, because here the current value is stored "inside" the `for`-operator and is automatically updated when it is executed again.

(ii) In Visual Basic you can modify the step range if you use a variable for the increment. This is also not possible with HDevelop because the increment is stored "inside" the `for`-operator when the loop is entered.

(iii) The last difference concerns the value of the loop variable after exiting the loop. In Visual Basic it has the value with which the condition becomes false for the first time. In HDevelop it contains the end value, which was calculated when the loop was entered.

Looking at the mentioned points we recommend to do the programming according to the following rules:

(i) Don't modify the loop variable or the step value inside the loop. If you need this behaviour use the `while`-loop.

(ii) Don't use the loop variable after the loop.

Automatic Display

Normally, the result of every operator is displayed in the graphics window of HDevelop. This is not the case when using an exported Basic program. It behaves like the HDevelop program running with the options: "update window = off". That means, whenever you want to have data to be displayed you must explicitly call the operator `dev_display`. This will be converted to the appropriate Basic call in the exported code. Because the template project already has a window, there is no need to explicitly open a window inside HDevelop.

Exception Handling

In HDevelop, every exception normally causes the program to stop and report an error message in a dialog window. This might not be useful in Visual Basic. The standard way to handle this in Visual Basic is by using the `On Error Goto` command. This allows to access the reason for the exception and to continue accordingly. Thus for HDevelop programs containing error handling (`dev_set_error_var`) the corresponding code is automatically included.
Please note that a call of `(dev_)set_check("give_error")` has no influence on the operator call. The exception will *always* be raised. This is also true for messages like `H_MESS_FAIL` which are not handled as exceptions in C++ e.g..

Special Comments

HDevelop comments containing the # symbol as the first character are exported as Visual Basic statement. Thus the line

```
* #Call MsgBox("Press button to continue",vbYes,"Program stop","",1000)
```

in HDevelop will result in

```
Call MsgBox("Press button to continue",vbYes,"Program stop","",1000)
```

in Visual Basic. This feature can be used to integrate Visual Basic code into an HDevelop program.

System Parameters

You should know that HDevelop performs some changes of system parameters of HALCON by calling the operator `set_system` (see reference manual). This might cause the Visual Basic program not to produce identical output. If such a problem arises, you may query the system parameters by means of `get_system` in HDevelop after or while running the original HDevelop version of the program. Depending to the problem, you can now modify relevant parameters by explicitly calling the operator `set_system` in your own Visual Basic program.

Multiple Windows

The exported code is intended to work together with the Visual Basic template. Thus only *one* window is available. All window operations like `open_window`, `close_window` and `set_window_extents` are therefore suppressed. If you want to use more than one window you have to modify the Basic code and project accordingly.

Chapter 7

Program Examples

This chapter contains examples that illustrate how to program with HDevelop. To understand the examples you should have a basic knowledge of image analysis.
The user interface is described in chapters 3 and 4. Language details are explained in chapter 5.
The examples of this chapter are also available as program code in the directory

```
$HALCONROOT/examples/hdevelop/Manuals/HDevelop
```

for UNIX or

```
%HALCONROOT%\examples\hdevelop\Manuals\HDevelop
```

on WindowsNT. More detailed information on HALCON operators is available in the reference manuals.

7.1 Stamp Segmentation

File name: `stamps.dev`
The first example performs a document analysis task.Figure 7.1 shows a part of a stamp catalog page. It contains two types of information about stamps: a graphical presentation and a textual description of the stamp.
In this example you have to transform the textual information into a representation that can be processed by a computer with little effort. You might use an OCR program for this task, but you will soon recognize that most of the available products create many errors due to the graphical presentation of the stamps. Thus another task has to be preprocessed: the elimination of all stamps (i.e., changing stamps to the gray value of the paper). After this preprocessing it is possible to process the remaining text using an OCR program. Figure 7.2 shows the segmentation result.
When creating an application to solve this kind of problem, it is helpful to describe characteristic attributes of the objects to be searched (here: stamps). This task can be solved by a novice with some experience, too. In this case, a characterization might look as follows:

- Stamps are darker than paper.
- Stamps are connected image areas that do not overlap.
- Stamps have a minimum and maximum size.

Figure 7.1: *Part of the page of a* Michel *catalog*

- Stamps are rectangular.

The task would be very simple if the attribute list would directly represent the program. Unfortunately, this is not possible due to the ambiguity of spoken language. Thus you need language constructs with a precise syntax and a semantics that are as close as possible to the informal description. Using the HDevelop syntax, an appropriate program would look like this:

```
dev_close_window ()
read_image (Catalog, 'swiss1.tiff')
get_image_pointer1 (Catalog, Pointer, Type, Width, Height)
dev_open_window (0, 0, Width/2, Height/2, 'black', WindowID)
```

Figure 7.2: *Segmentation result for stamps.*

```
dev_set_part (0, 0, Height-1, Width-1)
dev_set_draw ('fill')
threshold (Catalog, Dark, 0, 110)
dev_set_colored (6)
connection (Dark, ConnectedRegions)
fill_up (ConnectedRegions, RegionFillUp)
select_shape (RegionFillUp, StampCandidates, 'area',
              'and', 10000, 200000)
select_shape (StampCandidates, Stamps,
              'compactness', 'and', 1, 1.5)
```

```
smallest_rectangle1 (Stamps, Row1, Column1, Row2, Column2)
dev_display (Catalog)
dev_set_draw ('margin')
dev_set_line_width (3)
disp_rectangle1 (WindowID, Row1, Column1, Row2, Column2)
```

Due to the unknown operators and unfamiliar syntax this program appears unclear to the user at first glance.
But if you look closer at the operators you will notice the direct relation to the description above.

`threshold` selects all image pixels darker than the paper.

`connection` merges all selected pixels touching each other to connected regions.

`select_shape` selects the regions with areas (attribute: `'area'`) inside a specified interval.

`smallest_rectangle1` computes each region's coordinates (row/column) of the enclosing rectangle.

Once the user is familiar with the single operators and their syntax, the transformation becomes easy. In particular, it is not important to the program whether *an image* or *a set of regions* is processed. You can handle them both in the same way. In addition memory management of internal data structuresis transparent to the user. Thus, you do not need to bother about memory management and you can concentrate on the image analysis tasks to solve.

7.2 Capillary Vessel

File name: `vessel.dev`
The task of this example is the segmentation of a capillary vessel. In particular, you have to separate the cell area in the upper and lower part of figure 7.3 (left image) from the area in the middle of the image.

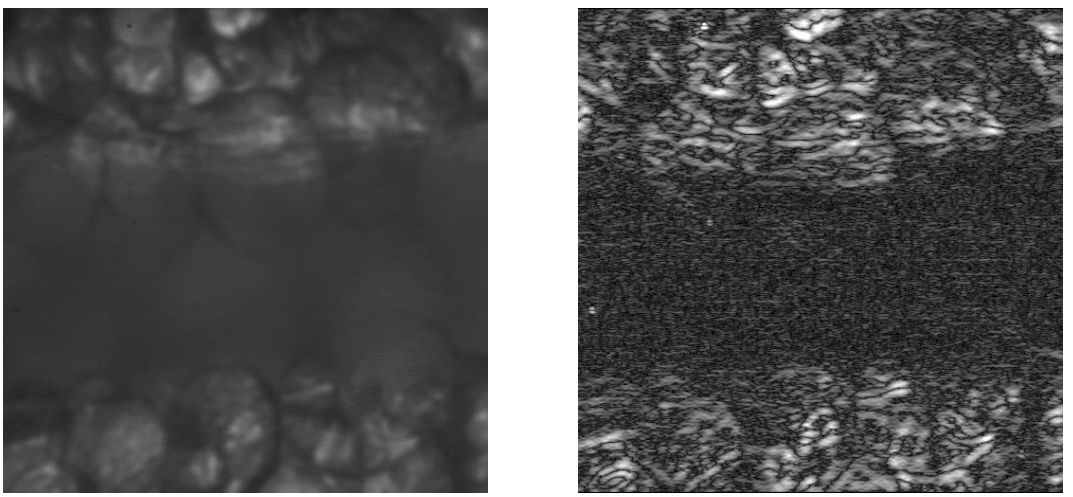


Figure 7.3: *Capillary vessel (left) and texture transformation (right).*

The area boundaries are very blurred and even a human viewer has difficulties recognizing them. At first glance it seems very difficult to find a segmentation criterion: There is neither a clear edge nor a significant difference between the gray values of both areas. Thus it is not very promising to use an edge operator or a threshold operation.
One solution of this problem makes use of the different textures within the areas: Cells are more textured than the part which is supplied with blood. To emphasize this difference you can use a *texture transformation* by Laws [Laws80]. Texture transformations are linear filters that intensify certain frequencies which are typical for the requested texture. The corresponding HALCON operator is `texture_laws`. You have to specify the filter size and type. Both attributes determine the frequency properties. In this program the filter `'el'` with mask size 5×5 is used. It performs a derivation in vertical direction and a smoothing in horizontal direction. Thus structures in vertical direction are intensified. You cannot directly use the computed result of `texture_laws` (see figure 7.3 right), because it is too speckled. Therefore you must generalize the texture image by a mean filter (`mean_image`). From this you obtain the so called *texture energy* (figure 7.4 left).

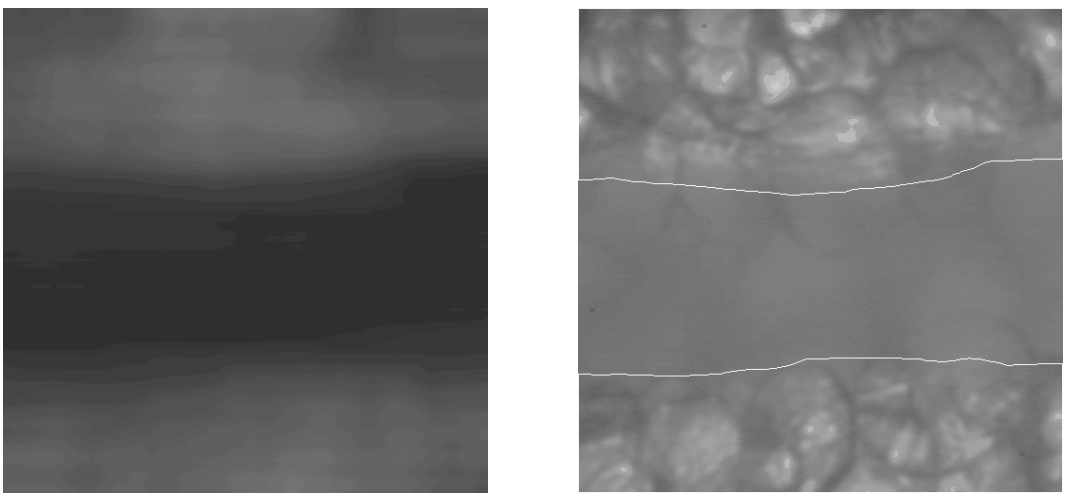


Figure 7.4: *Capillary vessel texture energy (left) and segmentation (right).*

The filter mask is chosen very large within this program. The mask size for the horizontal direction is 211 and 61 for the vertical direction. The asymmetry is used, because the vessel is nested in horizontal direction. From this you obtain an image with an upper and lower part that is brighter than that in the middle.

```
read_image (Image, 'vessel')
texture_laws (Image, Texture, 'el', 5, 5)
mean_image (Texture, Energy, 211, 61)
bin_threshold (Energy, Vessel)
```

To separate these areas you just have to find the appropriate threshold. In this case — we have only two types of textures — the threshold can be found automatically. This is done by the operator `bin_threshold`, which also applies the resulting threshold and thus extracts the vessel. The right side of figure 7.4 shows the result of the segmentation.

7.3 Particles

File name: `particle.dev`
This program example processes an image that was taken from a medical application. It shows tissue particles on a carrier (figure 7.5 left).

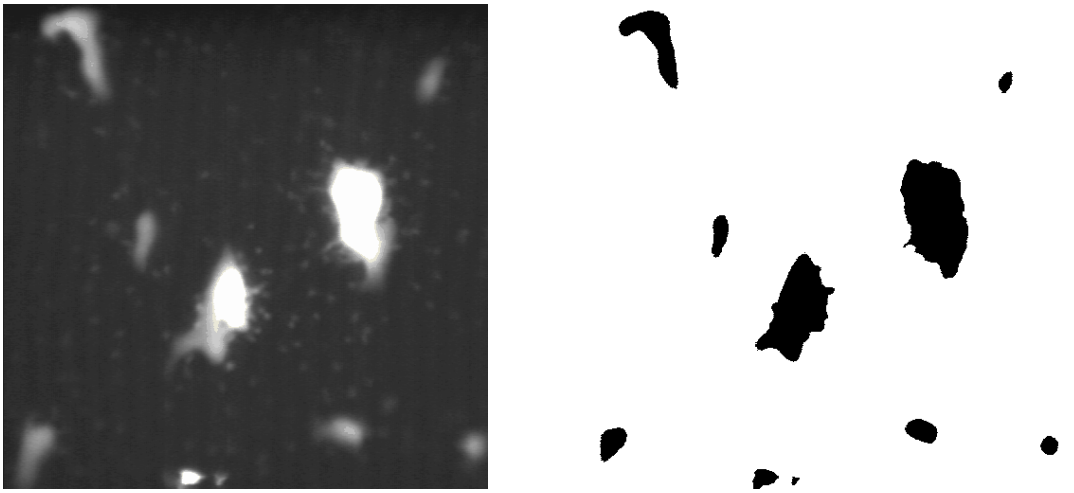


Figure 7.5: *Tissue particles (left) and large objects (right).*

As in many other medical applications, the existing objects have to be evaluated statistically. This means that different objects have to be extracted and classified according to their size or other attributes for example. After this, you can analyze them. An important step to solve this problem is the image segmentation that locates the relevant objects. For the statistical evaluation you may have a look at appropriate literature about statistics.
In our case there are two object classes:

- large, bright particles
- small, dark particles

The large, bright particles differ clearly from the background because of their gray values. The informal description 'brighter than the background' leads directly to the algorithmic solution using a thresholding. The only thing to decide is whether you specify the threshold automatically or empirically. In our case, a fixed threshold is completely sufficient due to the good contrast. Hence you get the following simple segmentation operator:

```
read_image (Particle, 'particle')
threshold (Particle, Large, 110, 255)
```

The variable `Large` contains all pixels whose gray values are brighter than 110. You can see the result on the right side of figure 7.5.
It is more difficult to find the small, dark particles. A first effort to specify a threshold interactively shows that there is no fixed threshold suitable to extract all particles. But if you look closer at the image you will notice that the smaller particles are much brighter than their local environment, i.e., you may specify suitable threshold values that are valid for a small image part

each. Now it is easy to transform this observation into an algorithm. One way is to determine the threshold values locally (e.g., from a bar chart). Another solution might be the definition of a local environment by an $n \times n$ window. This method is used in the example. The window's mean value is used as an approximation of the background intensity. This can be done by applying a low pass filter, such as a mean filter or a Gaussian filter. The window size n defines the size of the local environment and should approximately be twice as large as the objects to search for. Since they show an average diameter of 15 pixels, a mask size of 31 is used.
The resulting pixels are specified by the comparison of the original gray values with the mean image. To reduce problems caused by noise you add a constant to the mean image (3). The appropriate program segment looks as follows:

```
mean_image (Particle, Mean, 31, 31)
dyn_threshold (Particle, Mean, Small, 3, 'light')
```

The operator `dyn_threshold` compares two images pixel by pixel. You can see the segmentation result in figure 7.6 left.

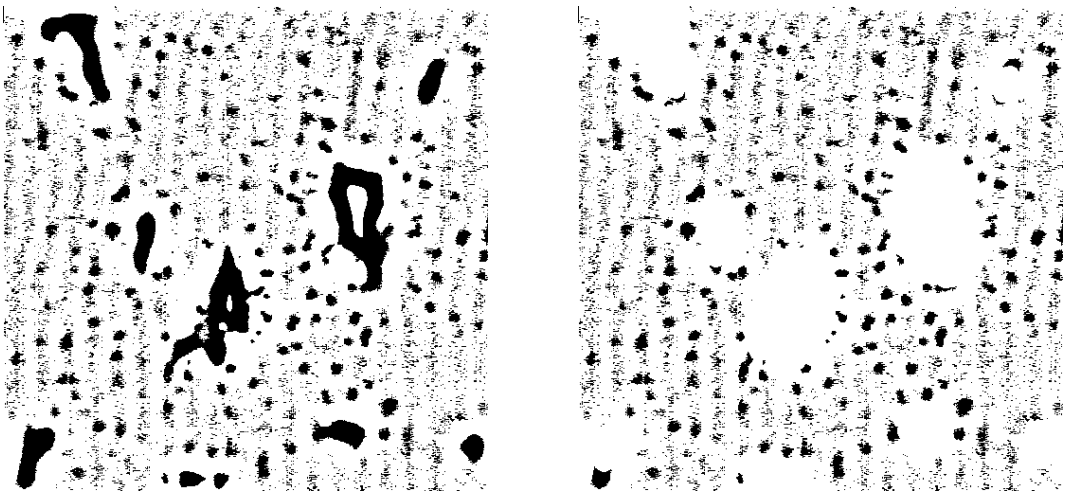


Figure 7.6: *Small objects: simple (left) and advanced segmentation (right).*

As we see, all objects have been found. Unfortunately, the edges of the large particles and several very small regions that emerged due to the noisy image material were found, too.
We first try to suppress the edges. One way is to eliminate all objects that exceed a certain maximum size. You can do this by calling:

```
conncetion (Small, SmallSingle)
select_shape (SmallSingle, ReallySmall, 'area', 'and', 1, 300)
```

By the same method you might also eliminate all objects which are too small (blurring). For this, you would just have to increase the minimum size with the call of `select_shape`. But if you examine the segmentation results again, you will notice that some of the resulting pixels were already extracted by the first segmentation. Thus you should search the small particles within the complement of the large ones only. To avoid the segmentation of small particles in the direct neighbourhood of the large ones, those are enlarged before building their complement. Thus we get the following modified program:

```
dilation_circle (Large, LargeDilation, 8.5)
complement (LargeDilation, NotLarge)
reduce_domain (Particle, NotLarge, ParticleRed)
mean_image (ParticleRed, Mean, 31, 31)
dyn_threshold (ParticleRed, Mean, Small, 3, 'light')
```

This method shows two advantages: First, the (reliable) model of the large particles can be used to extract the small ones. This increases the quality of the segmentation. Second, the processing speed is increased, as the second segmentation works only on a part of the image data. The right side of figure 7.6 shows the segmentation result.
Unfortunately, the image still contains noise. To remove it, you may either sort out noisy objects by their area as described above, or by an *opening* operation. We prefer the second method as it additionally smooths the object edges.

```
opening_circle (Small, SmallClean, 2.5)
```

Here, a circle is used as the structuring element of the opening operation. The operator preserves regions only that may at least cover a circle of radius 2.5. Smaller regions are eliminated.

Figure 7.7: *Noise-removed segmentation (left) and final result (right).*

Figure 7.7 shows the result of the segmentation with noise removal on the left side. The right side contains the final result.
Finally, we would like to show within this example how to select regions with the mouse interactively. At this, a loop is executed until you press the middle or right mouse button. When pressing a mouse button, the operator `get_mbutton` returns the button that was pressed and the position (coordinates) where it was pressed. This information is used to select the chosen object. In the following you see the corresponding program part:

```
dev_clear_window (WindowID)
connection (SmallClean, SmallSingle)
Button := 1
dev_set_color ('red')
```

```
while (Button = 1)
  get_mbutton (WindowID, Row, Column, Button)
  select_region_point (SmallSingle, OneObject, Row, Column)
  intensity (OneObject, Particle, MeanGray, Deviation)
endwhile
```

First, the window is cleared via `dev_clear_window`. After that, `connection` calculates all connected components to allow the selection of single regions. This also displays the region components in the HDevelop window. Then you may set the drawing color (here: red) to visualize the selected regions. The loop is initialized by assigning 1 to the variable `Button` (1 is the code for the left mouse button). Within the loop the mouse state is queried and the chosen region is selected. As an example the mean gray value and the standard deviation are computed for each selected region. As long as you press only the left mouse button within the window the loop continues. You can terminate it by pressing any other mouse button.

7.4 Annual Rings

File name: `wood.dev`
Everyone knows the task to determine the age of a tree by counting its annual rings. This will now be done automatically using the example program. The first step is the segmentation of annual rings. This is quite simple as you can see them clearly as bright or dark lines. Again, the dynamic thresholding (`dyn_threshold`) can be used (as before during the particle segmentation in section 7.3). To achieve a suitable threshold image you apply the mean filter (`mean_image`) with size 15×15 first.
The segmentation result contains many tiny regions that are no annual rings. To eliminate them you have to create the connected components (`connection`) and suppress all regions that are too small (`select_shape`). Counting the rings becomes difficult, as there might be fissures in the wood (see figure 7.8).

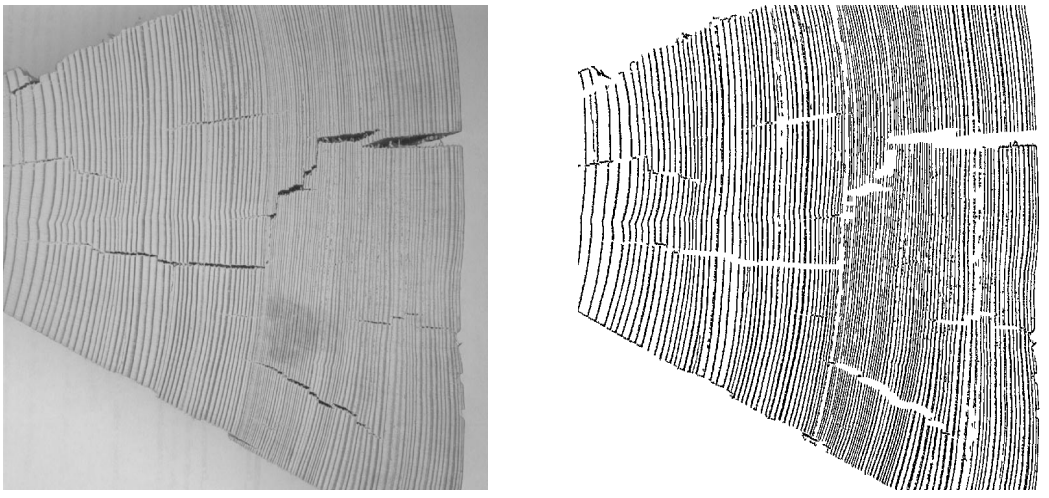


Figure 7.8: *Annual rings of a tree.*

Thus, we suggest the following method: You define the start and end point of a line across the annual rings using your mouse. Then the number of intersections with annual rings is counted

along this line. This can be done by the following HALCON operators: The start and end points, represented by their x- and y-coordinates, are transformed into a line (`gen_region_line`). This line is intersected (`intersection`) with the annual rings (`SelectedRegions`). The number of the connected regions (`count_obj`) in this intersection is the number of annual rings. The complete program looks as follows:

```
dev_close_window ()
read_image (WoodPiece1, 'woodring')
get_image_pointer1 (WoodPiece1, Pointer, Type, Width, Height)
dev_open_window (0, 0, Width/2, Height/2, 'black', WindowID)
mean_image (WoodPiece1, ImageMean, 9, 9)
dyn_threshold (WoodPiece1, ImageMean, Regions, 5.0, 'dark')
threshold (WoodPiece1, Dark, 0, 90)
dilation_rectangle1 (Dark, DarkDilation, 30, 7)
difference (Regions, DarkDilation, RegionBright)
connection (RegionBright, ConnectedRegions)
select_shape (ConnectedRegions, SelectedRegions,
              'area', 'and', 30, 10000000)
get_mbutton (WindowID, Row1, Column1, Button1)
get_mbutton (WindowID, Row2, Column2, Button2)
gen_region_line (Line, Row1, Column1, Row2, Column2)
intersection (Line, SelectedRegions, Inters)
connection (Inters, ConnectedInters)
Number := |ConnectedInters|
```

7.5 Bonding

File name: `ball.dev`
This is the first example in the field of quality inspection. The task is to detect bonding balls. Figure 7.9 shows two typical microscope images of a *die*.

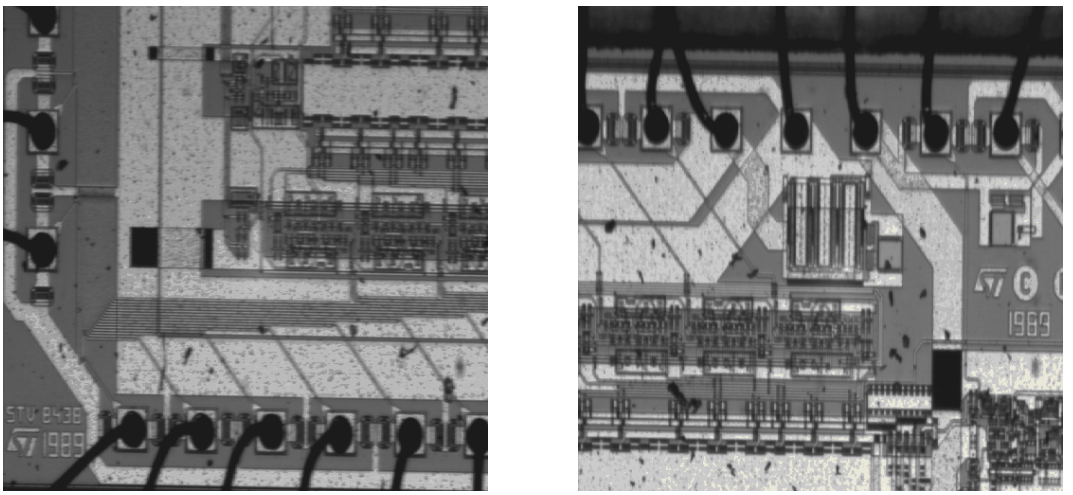


Figure 7.9: *Examplary images with bonding balls on a* die.

The *die* border and the bonding wires appear dark. Thus you may apply a thresholding. Since the background is also dark we have to extract the *die* before doing the segmentation. The *die* is rather bright. Thus we can select the pixels by their gray values.

```
read_image (Bond, 'die3')
threshold (Bond, Bright, 120, 255)
shape_trans (Bright, Die, 'rectangle2')
```

All pixels of the *die* that got lost by the thresholding can be recoverd by using a hull computation. Since the *die* is rectangular and may be slightly turned during the assembly we use the smallest enclosing rectangle as a hull.
Now you can start the segmentation of wires and bonding balls. Since only those parts of wires and balls are of interest that lie within the *die* area, you may restrict the segmentation to this region. All dark pixels within the *die* area belong to wires. Unfortunately, there are some bright reflections on the wires that are not found by the segmentation. You may fill these gaps by using `fill_up_shape`. In our case, the gaps with a certain size (1 up to 100 pixels) are filled.

```
reduce_domain (Bond, Die, DieGray)
threshold (DieGray, Wires, 0, 100)
fill_up_shape (Wires, WiresFilled, 'area', 1, 100)
opening_circle (WiresFilled, Balls, 15.5)
connection (Balls, SingleBalls)
select_shape (SingleBalls, IntermediateBalls, 'circularity',and,0.85, 1.0)
sort_region (IntermediateBalls, FinalBalls, 'FirstPoint', 'True', 'column')
smallest_circle (FinalBalls, Row, Column, Radius)
```

Since the balls are wider than the wires, you may clean this region using a simple opening. The radius (here 15.5) should correspond to the minimum size of one ball. In both images you see an erroneous segmentation that was created by a rectangular dark region. This can be suppressed by a shape segmentation. Since in practice a bonding detection would be performed only close to the anticipated positions of bonding balls. Figure 7.10 shows the results of the whole segmentation.
Balls are shown in white color. Every radius of a ball you can find in the tuple variable Radius. The number of balls within the example you can get with the absolute value of Radius.

```
NumBalls := |Radius|
Diameter := 2*Radius
MeanDiameter := sum(Diameter)/NumBalls
MinDiameter := min(Diameter)
```

Diameter, MeanDiameter and MinDiameter are some examples for calculations possible with HDevelop.

7.6 Calibration Board

File name: `calib.dev`
This example works with the image of a calibration board. It is used to specify the internal

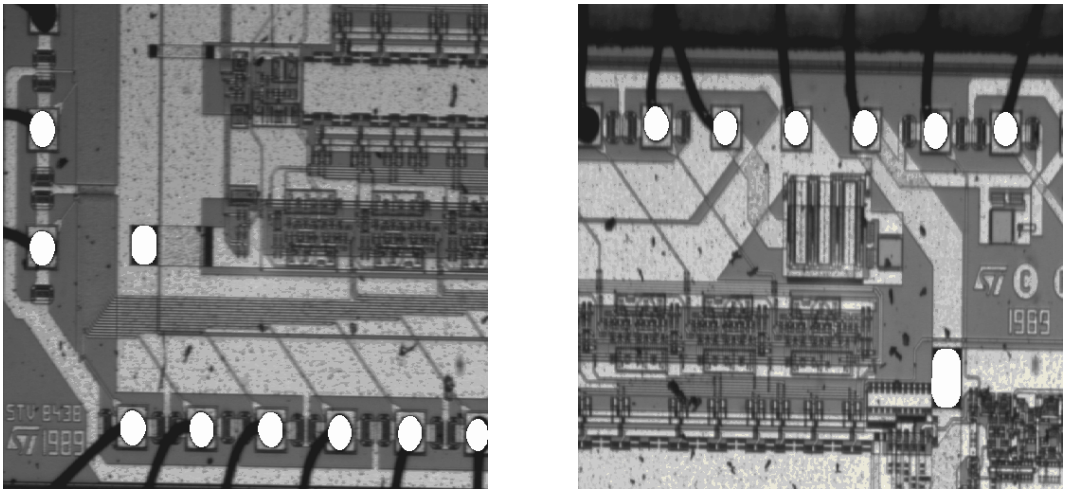


Figure 7.10: *Detected bonding positions.*

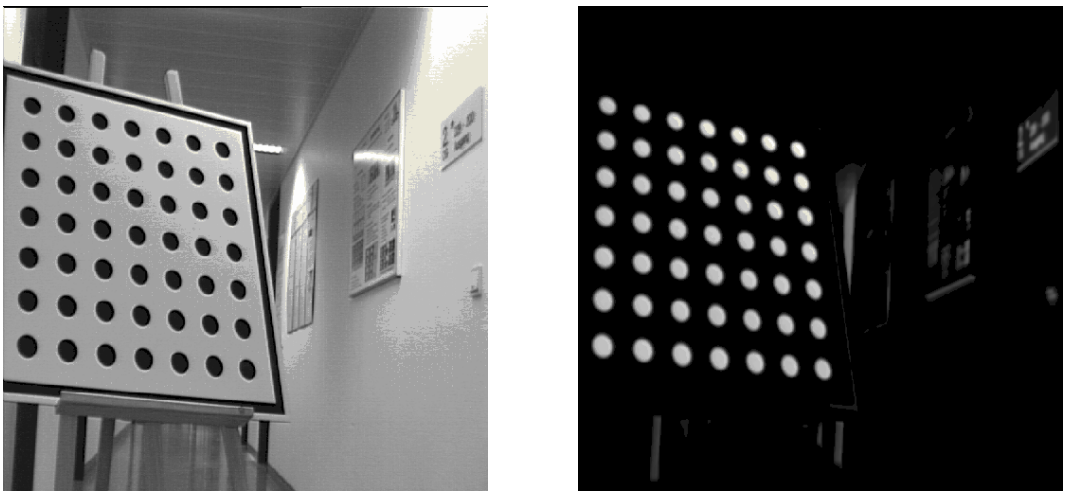


Figure 7.11: *Calibration board and* `gray_inside` *result.*

parameters of a CCD camera. Therefore, you have to extract the circles on the board (see left side of figure 7.11).
This example describes an interesting operator. It is called `gray_inside` and is a so-called *fuzzy operator*. In this case, fuzzy means that the value of each pixel is not interpreted as gray value but as the *affiliation* to a certain class. The bigger the number (max. 1), the stronger the affiliation.[1]
By applying `gray_inside` to an image every pixel value is interpreted as the "potential energy" you have to afford to get from the pixel position to the image border. The dark pixels present valleys and the bright pixels mountains. Thus a dark region in the middle of an image is equivalent to a hole in a mountain that needs a lot of energy to be left. This is also true for the dark circles on the bright background in the image of the calibration board.
Before calling `gray_inside` you should use a smoothing filter to suppress small valleys. This

[1] In HALCON the range of 0 to 1 is mapped to values of a byte image (0 to 255).

reduces runtime considerably.
If you look at the operator result on the right side of figure 7.11 you will notice the circles as significant bright points. Now a simple thresholding is sufficient to extract them.

```
read_image (Caltab, 'caltab')
gauss_image (Caltab, ImageGauss, 9)
gray_inside (ImageGauss, ImageDist)
threshold (ImageDist, Bright, 110, 255)
connection (Bright, Circles)
elliptic_axis (Circles, Ra, Rb, Phi)
```

After calculating the ellipse parameters of each circle (`elliptic_axis`), you may compute the camera parameters.

7.7 Devices

File name: `ic.dev`
This example discusses the combination of different segmentation methods. It works with an image of multiple electronic components. These differ in shape, size and arrangement. The left side of figure 7.12 shows the input image.

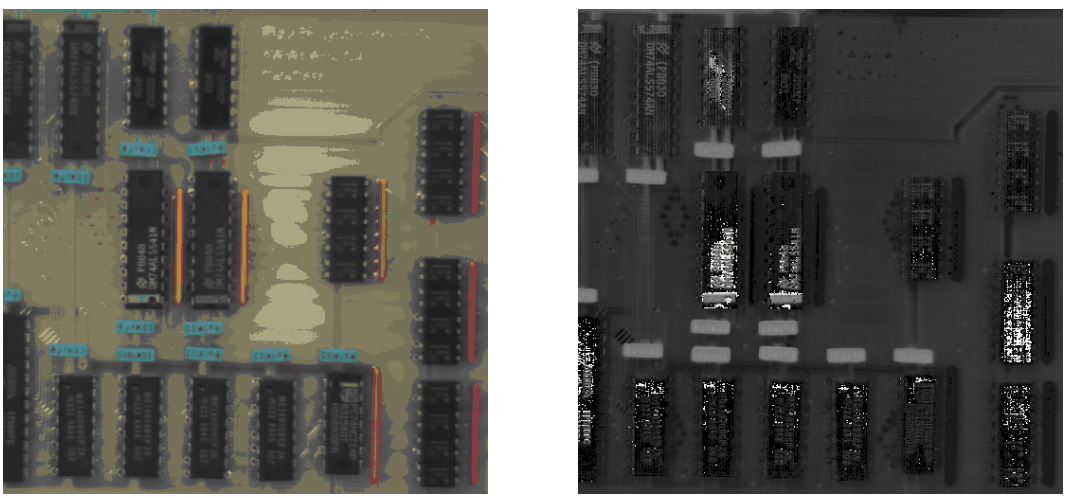


Figure 7.12: *Board with electonic devices (left) and the corresponding color value image in the HSV space (right).*

First you extract resistors and capacitors. This is quite simple because you have a color image and both component types have different colors. The input image consists of three channels containing the red, green, and blue channels. Since segmentation in the RGB space is difficult, you have to transform the image into the HSV space. Here the color information is stored in one single channel. The right side of figure 7.12 shows the image representation in this channel (*Hue*). Elements that are too small can be eliminated via `select_shape`. The program sequence to extract resistors and capacitors is shown below:

```
read_image (ICs, 'ic')
decompose3 (ICs, Red, Green, Blue)
trans_from_rgb (Red, Green, Blue, Hue, Saturation, Intensity)
threshold (Saturation, Colored, 100, 255)
reduce_domain (Hue, Colored, HueColored)
threshold (HueColored, Blue, 114, 137)
connection (Blue, BlueConnect)
select_shape (BlueConnect, BlueLarge, 'area', 'and', 150, 100000)
shape_trans (BlueLarge, Condensators, 'rectangle2')
threshold (HueColored, Red, 10, 19)
connection (Red, RedConnect)
select_shape (RedConnect, RedLarge, 'area', 'and', 150, 100000)
shape_trans (RedLarge, Resistors, 'rectangle2')
```

If you look closer at this program segment you will notice some obvious enhancements that can be made. One is necessary due to the color model: The thresholding of the color image chooses all pixels with a certain color. This selection is independent of the color saturation. Thus it might happen that very bright pixels (nearly white pixels) or very dark pixels (nearly black pixels) have the same color value as the components. But you are only looking for stronger colors. For this you select all pixels first whose color is strong, i.e., all pixels with a high saturation.
The second enhancement concerns the objects' shape. As the devices are rectangular you can specify the smallest enclosing rectangle of all connected components to enhance the segments.On the left side of figure 7.13 the resulting components are marked.

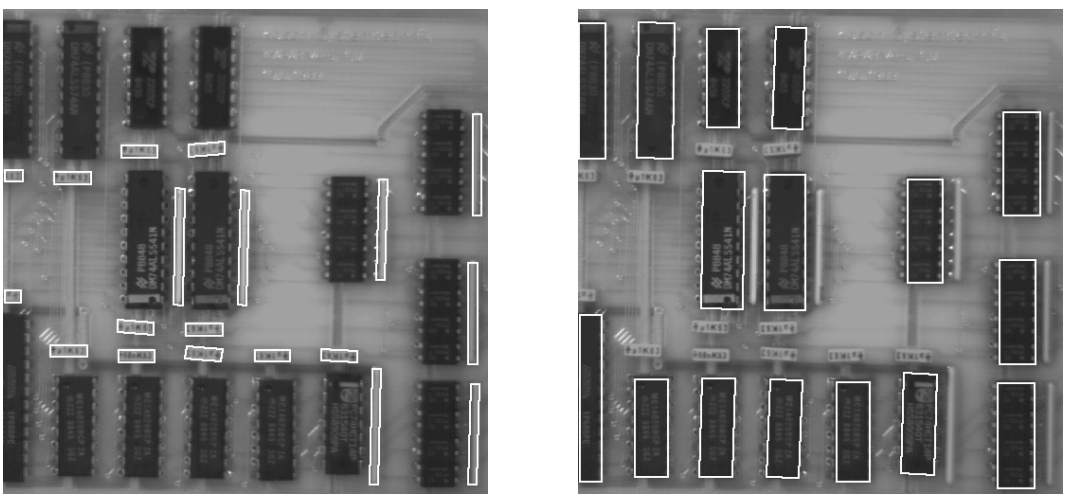


Figure 7.13: *Resistors and capacitors (left) and ICs (right).*

In a second step, we will search for all ICs. This seems to be easy, as they are rather large and dark. However, some problems emerge due to the bright labels that are printed across some ICs. Thus a simple thresholding alone is not sufficient. In addition you have to combine the segments belonging to one IC. This is done by examining the spatial adjacencies of the segments. A dilation is used to enlarge the regions until they overlap each other. This dilation must not be so large that different ICs are merged. Thus gaps caused by labels have to be smaller than gaps between ICs. Now you can separate the enlarged ICs in their connected components.

Unfortunately, they have become too large by the dilation. Another thresholding for each connected component will detect the dark pixels of each IC. Finally, you can specify the enclosing rectangles analogously to the resistors and the capacitors (see above).

```
threshold (Intensity, Dark, 0, 50)
dilation_rectangle1 (Dark, DarkDilate, 15, 15)
connection (DarkDilate, ICLarge)
add_channels (ICLarge, Intensity, ICLargeGray)
threshold (ICLargeGray, ICsDark, 0, 50)
shape_trans (ICsDark, IC, 'rectangle2')
```

The right side of figure 7.13 shows the resulting ICs. We have to mention two aspects about the program segment above. Here the operator `add_channels` has been used instead of `reduce_domain`. This is necessary as *several* regions have to be "supplied" with gray values. The situation of previous programs was quite different: there the number of valid pixels of *one* image has been restricted. From this follows the second point: here the operator `threshold` gets several images as input.[2] The thresholding is performed in every image. Thus you receive as many regions as input images.
Finally, the segmentation of IC contacts has to be done. They are bright and small. Thus it is easy to extract them using a dynamic thresholding (compare chapter 7.3). However, several other tin elements on the board remain a problem, because they have to be distinguished from the IC contacts. This can be done by restricting the search on a *region of interest*. IC contacts may only appear either on the right or the left side of IC's. The coarse region of interest is defined by enlarging the IC regions with a following set subtraction. Then the result is resized appropriately by using another dilation. Figure 7.14 shows the operator result on the left side.

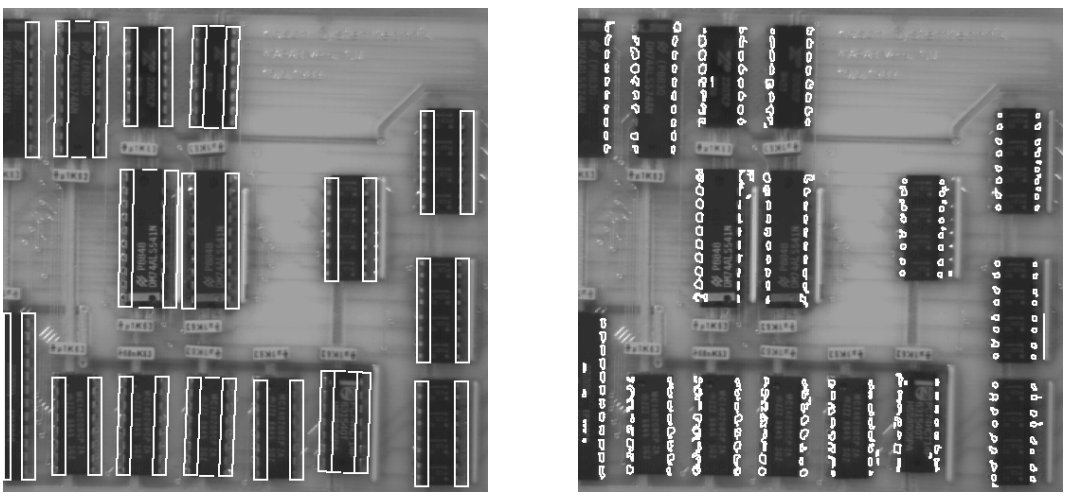


Figure 7.14: *Searching regions for contacts (left) and IC contacts (right).*

Now you only have to intersect the result of the thresholding with the region of interest.

```
dilation_rectangle1 (IC, ICWidth, 5, 1)
difference (ICWidth, IC, SearchingArea)
```

[2]One matrix is shared by several iconic objects to reduce costs of memory and computation time.

```
dilation_rectangle1 (SearchingArea, SearchingAreaWidth, 14, 1)
union1 (SearchingAreaWidth, SearchingAreaUnion)
reduce_domain (Intensity, SearchingAreaUnion, SearchGray)
mean_image (SearchGray, Mean, 15, 15)
dyn_threshold (SearchGray, Mean, Contacts, 5, 'light')
connection (Contacts, ContactsConnect)
fill_up (ContactsConnect, ContactsFilled)
select_shape (ContactsFilled, ContactsRes, 'area', 'and', 10, 100)
```

The result of the intersection is still not satisfying. Too many small and too many wrong regions have been found. So we have to eliminate them by using `select_shape`. Figure 7.14 shows the final result of the segmentation on the right side.

7.8 Cell Walls

File name: `wood_cells.dev`
In this example we will examine the alteration of the cell wall's proportion during a tree's growth. The input image is a microscope view of wooden cells (see figure 7.15).

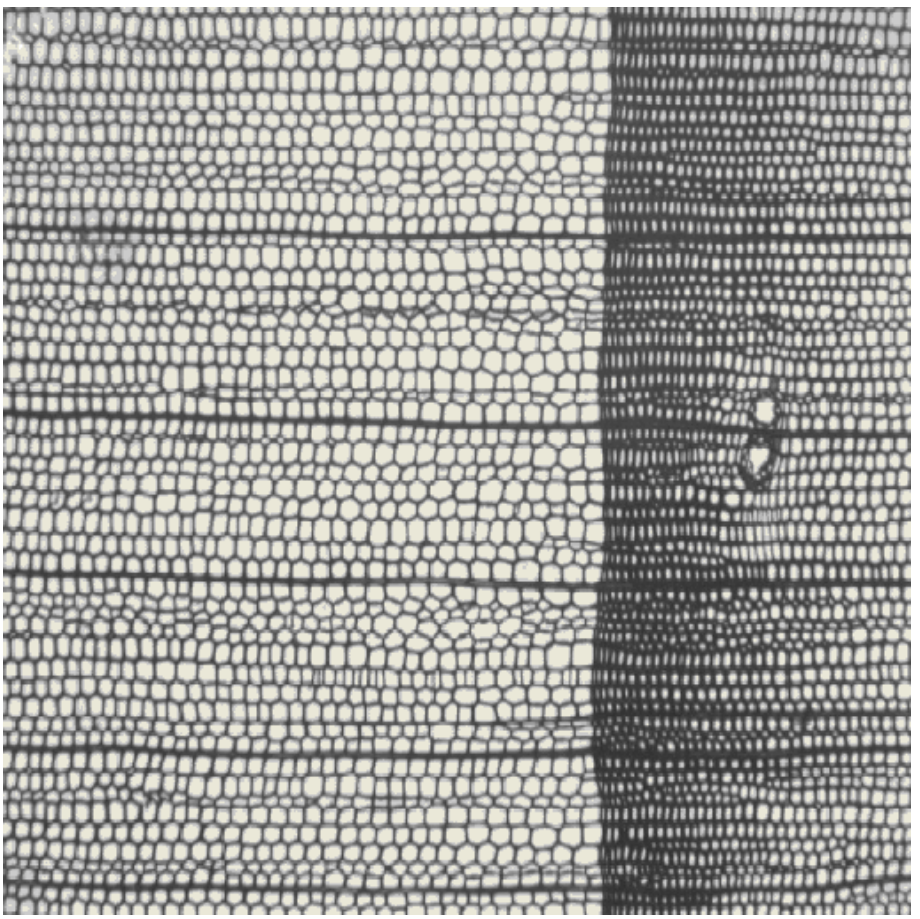


Figure 7.15: *Microscope image of wooden cells.*

You can clearly see the single cells and the discontinuity that is caused by the stopped growing in winter.
Extracting cell walls is simple because they are significantly darker. The remaining “difficulty” lies in the computation of the distribution in growth direction, i.e., along the image x-axis. First, we define the width of the window over which the cell distribution is computed by assigning it to the variable `X` in the program. Then we fetch the image size using `get_image_pointer1` to get the corresponding loop parameters. The broader the search range, the stronger the smoothing during the measurement.
Now the loop starts from the “left” side to compute the whole image. The ratio of the area of the cell walls and a rectangle of width `X` is computed for every value of the loop variable `i`. The number of pixels belonging to a cell wall (`Area`) is determined by `area_center`. This value is transformed to percent for the output.

```
X := 20
read_image (WoodCells1, 'woodcell')
threshold (WoodCells1, CellBorder, 0, 120)
get_image_pointer1 (WoodCells1, Pointer, Type, Width, Height)
open_file ('wood_cells.dat', 'output', FileHandle)
for i := 0 to Width-X-1 by 1
  clip_region (CellBorder, Part, 0, i, Height-1, i+X)
  area_center (Part, Area, Row, Col)
  fwrite_string (FileHandle, i + ' ' + (Area * 100.0 / (X * Height)))
  fnew_line (FileHandle)
endfor
close_file (FileHandle)
```

Figure 7.16 shows the measurement result.

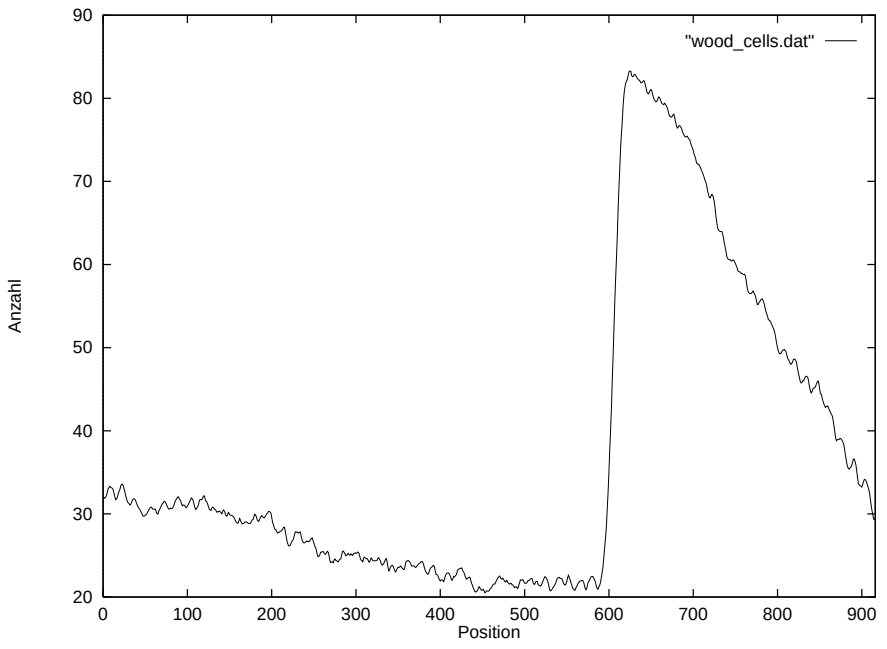


Figure 7.16: *Cell wall proportion in growth direction in percent*

To allow further processing of the data (such as for plotting using `gnuplot` as in figure 7.16) it has to be written to a file. Therefore, a text file is opened first (`open_file`). Now you can write to this file by using `fwrite_string` and `fnew_line`. Note the formatting of output when using `fwrite_string`. The output text starts with the loop variable that is followed by a space character. Thus the number is transformed into a string. Finally, the proportion of the cell wall (in percent) is concatenated to the string. At this it is important that the first or second value of the expression is a string, so that the following numbers are converted into strings. Here + denotes the concatenation of characters instead of the addition of numbers.

7.9 Region Selection

File name: `eyes.dev`
This example explains how to handle single iconic objects. In contrast to numerical data, where many different functions may be executed on parameter positions (see chapter 5.5), iconic objects may only be handled by using HALCON operators. The most important operators to select and combine iconic objects are shown in this example.
The task is to search the eyes of the mandrill in figure 7.17.

Figure 7.17: *Mandrill and the detected result.*

This is a simple task. First, we extract the bright parts by a thresholding. Then we have to examine the connected components according to their shape and size to select the eyes. At this, you could use the operator `select_shape` and get a fast program of five lines that processes the task. For demonstration purpose we use a kind of “low level” version instead: every region is extracted separately and examined afterwards. If it conforms to a given shape, it is added to a result variable.

```
dev_close_window ()
read_image (Image, 'monkey')
threshold (Image, Region, 128, 255)
connection (Region, ConnectedRegions)
select_shape (ConnectedRegions, CompactRegions,
```

```
                'compactness', 'and', 1.5, 1.8)
Number := |CompactRegions|
Eyes := []
for i := 1 to Number by 1
  SingleSelected := CompactRegions[i]
  area_center (SingleSelected, Area, Row, Column)
  dev_set_color ('green')
  if ((Area > 500) and (Area < 50000))
    dev_set_color ('red')
    Eyes := [SingleSelected,Eyes]
  endif
endfor
```

Note that you have to specify the number of regions (`count_obj`) in order to run a `for` loop from 1 to `Number`. Within this loop a region is selected (`select_obj`) according to the loop variable `i` in order to evaluate its attributes. If its area is within certain bounds the region is added to variable `Eyes` (`concat_obj`). You have to specify the variable `Eyes` properly, as it is also used as input for `concat_obj`. This can be done by using `empty_object` that assigns no iconic object in a defined way to the variable, i.e., `count_obj` returns zero for it.
During the run time of the program you can see how the individual regions are selected and examined. To speed up the processing you can use the menu `File` ▷ `Options` to suppress the automatic output.

7.10 Exception Handling

File name: `exception.dev`
In some applications it is necessary to have explicit control over the result state of an operator. By default HDevelop stops if an operator returns a different state than `H_MSG_TRUE` and gives an error message. To have explicit control over the result state, two HDevelop operators are available: `dev_error_var` and `dev_set_check`. The following example shows how to use these operators.
The task is to get online information about the position of the mouse inside a graphics window and to display the gray value at this position. This can be achieved using the two operators `get_mposition` and `get_grayval`. The problem with `get_mposition` in HDevelop is, that it returns `H_MSG_FAIL` if the mouse is outside of the window to indicate that the mouse coordinates are invalid. This would lead to an interruption of the program. Therefore an explicit error handling is needed. The complete program is given below:

```
read_image (Image, 'mreut')
dev_close_window ()
dev_open_window (0, 0, -1, -1, 'black', WindowID)
dev_display (Image)
Button := 1
while (Button # 4)
  dev_error_var (Error, 1)
  dev_set_check ('~give_error')
  get_mposition (WindowID, Row, Column, Button)
```

```
    dev_error_var (Error, 0)
    dev_set_check ('give_error')
    if (Error = H_MSG_TRUE)
      get_grayval (Image, Row, Column, Grayval)
      dev_set_color ('black')
      disp_rectangle1 (WindowID, 0, 0, 22, 85)
      dev_set_color ('white')
      set_tposition (WindowID, 15, 2)
      write_string (WindowID, '('+Row+','+Column+')='+Grayval)
    endif
  endwhile
```

After loading an image and opening a window we enter the loop to query the mouse position. Because the operator `get_mposition` might cause an exception we call `dev_set_check` to declare that HDevelop should not stop if an exception occurs. `dev_set_check` has to be called before and after the critical call(s). If we want to know which error occurred we have to specify the variable in which the return value will be stored. This is done by using `dev_error_var`. Now `get_mposition` can be called independent of the context. To check if the coordinates are valid, the error variable is compared to one of the constants for standard return values (like `H_MSG_TRUE` or `H_MSG_FAIL`). If the call succeeded, this coordinate is used to query the gray value of the corresponding pixel in the image, which is then displayed in the window.

7.11 Road Scene

File name: `road_signs.dev`

The computing time is a critical factor in many image analysis tasks. Thus the system has to offer features to speed up the processing. But direct hardware access must be avoided in any case. All operators should work on encapsulated data structures. To allow optimization for performance, data structures have to be used that support transparent and efficient programming. The example segmentation of a road scene demonstrates how HALCON helps to achieve this goal.

Here the task is to find the middle and border road markings of a motorway. The program is performed by a normal workstation with a processing time of maximum 20 ms per half image (video frequency) at a resolution of 512×512 pixels. In figure 7.18 you see an image of such a road sequence on the left side.

Assume that there is no specialized operator for this task. Thus, you have to make use of standard methods. The data structure used consists of a gray value image with a covering mask, i.e., the definition range.[3] All operators work only on those parts of the image data that lie within the definition range. This can be exploited to reduce computation time.

The following assumptions on the image data help to specify a region as a search mask:

(i) Road markings remain in a certain image part only.

(ii) Road markings have a certain minimum length in y-direction.

(iii) Road markings are separated by an edge from their environment.

[3] See section 2.1.2 for a short introduction to the data structures used by HDevelop.

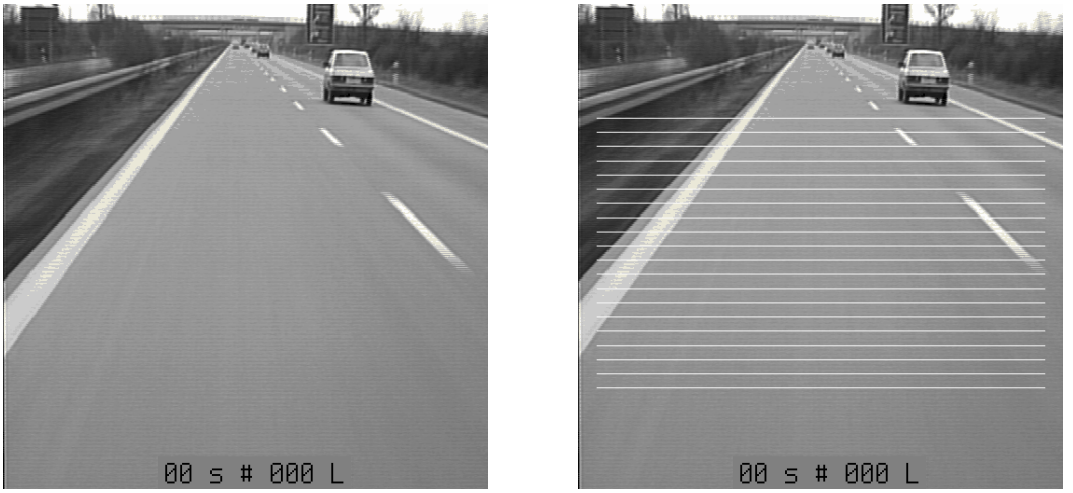


Figure 7.18: *Part of an image sequence (left) and search grid for marking band (right).*

The first two assumptions can restrict the search area enormously. To make use of this, we create a region as a grid whose line distance is determined by the minimum size of the road marking. Figure 7.18 shows the corresponding region (= line grid) on the right side.
While performing an edge filter within the grid all pixels with a high gradient are candidates on the contour of a road marking. By enlarging these pixels by the minimum diameter of the markings (dilation) with rectangle, you will get the search window shown in figure 7.19 on the left side.

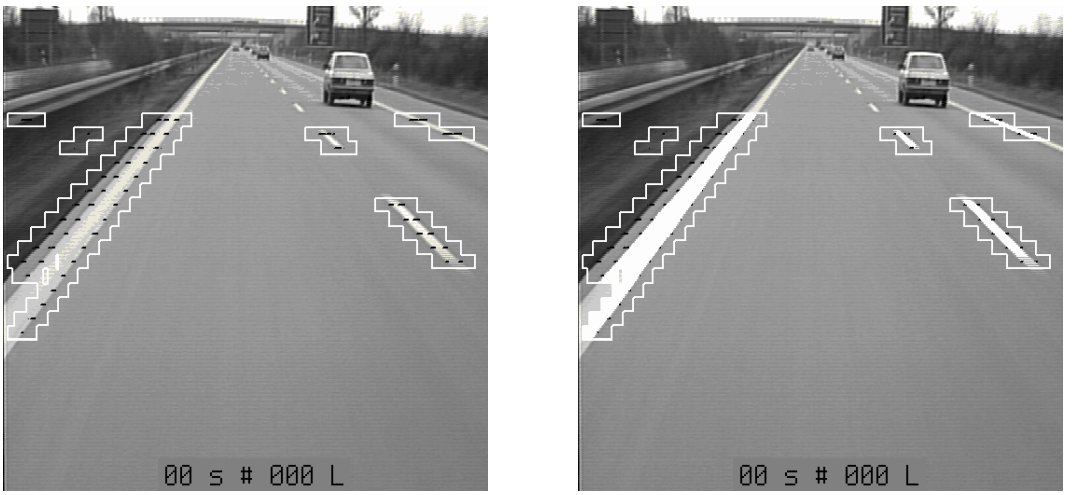


Figure 7.19: *Search areas for markings (left) and segmentation (right).*

Now the road markings can be easily extracted by a thresholding within the search windows. The segmentation result is shown on the right side of figure 7.19. The corresponding HDevelop program looks as follows:

```
MinSize := 30
set_system ('init_new_image', 'false')
```

```
read_image (Motorway, 'motorway')
count_seconds (Seconds1)
gen_grid_region (Grid, MinSize, MinSize, 'lines', 512, 512)
clip_region (Grid, GridRoad, 130, 10, 450, 502)
reduce_domain (Motorway, GridRoad, Mask)
sobel_amp (Mask, Gradient, 'sum_abs', 3)
threshold (Gradient, Points, 40, 255)
dilation_rectangle1 (Points, RegionDilation, MinSize, MinSize)
reduce_domain (Motorway, RegionDilation, SignsGray)
threshold (SignsGray, Signs, 190, 255)
count_seconds (Seconds2)
Time := Seconds2-Seconds1
dev_display (Signs)
```

First you create a grid by using `gen_grid_region`. It is reduced to the lower image half with `clip_region`. The operator `reduce_domain` creates an image containing this pattern as definition range. This image is passed to the operator `sobel_amp`. You obtain pixels with high gradient values using `threshold`. These pixels are enlarged to the region of interest (ROI) by a dilation with a rectangular mask. Within this region another thresholding is performed. Correct road markings are equivalent to bright areas in the search window (ROI).
The HALCON program needs an average of 20 ms on a standard Pentium. Notice that this is even possible under the following restrictions:

(i) Only standard operators have been used.

(ii) Only encapsulated data structures have been used.

(iii) Despite optimization the program is quite comprehensible.

(iv) The program is very short.

This example shows that you can write efficient programs even while using complex data structures. Hence a significant reduction of development time is achieved. Furthermore, data encapsulation is a basic condition for the portability of the whole system and the user software.

Chapter 8

Miscellaneous

In this chapter different aspects of working with HDevelop are discussed in detail.

8.1 Starting of HDevelop

In UNIX HDevelop is started from the shell like any other program. Optionally, an application name can be given as a parameter to HDevelop:

```
hdevelop <File>.dev
```

This application is then loaded. This is identical to a normal invocation of HDevelop (without any parameter) and a subsequent loading of the application. If you want to run the application immediately after it has been loaded, you can invoke HDevelop as follows:

```
hdevelop -run <File>.dev
```

This is identical to a normal invocation of HDevelop (without any parameters) with a loading of the application and pressing the `Run`-button in HDevelop. After the application has been executed, HDevelop waits for input from the user. The program can be interrupted or stopped by pressing the `Stop`-button. To stop or halt the complete program without user interaction the `stop` or `exit` operator can be used.

8.2 Keycodes

In order to speed up the entering of values in the input fields of HDevelop (e.g., operator parameters), several keycodes are defined, which have special functions. They conform to the standards of the `emacs` editor. This feature is only available for UNIX systems. Some of them are shown in table 8.1.

8.3 Interactions during Program Execution

The interpreter of HDevelop allows some user interactions during the execution of a program. First, the stop button has to be mentioned, which is responsible for interrupting the execution of a program. When the stop button is pressed, the execution is stopped at the active HALCON operator.

Delete	Delete single character at current cursor position.
<Ctrl> a	Move the cursor to the beginning of the line.
<Ctrl> b	Move cursor left one character.
<Ctrl> d	Analogous to Delete
<Ctrl> e	Move cursor to last character in line.
<Ctrl> f	Move cursor right one character.
<Ctrl> h	Delete single character immediately preceding current cursor position
<Ctrl> k	Delete all characters from current position to end of line.
<Meta> b	Backward to previous word.
<Alt> b	Backward to previous word.
<Meta> d	Delete from current cursor position to end of current word.
<Alt> d	Delete from current cursor position to end of current word.
<Meta> f	Forward to next word.
<Alt> f	Forward to next word.

Table 8.1: Keycodes for special editing functions

Other features of the HDevelop interpreter are the possibility to display iconic variables by simply double clicking on them, and the facility to set the parameters which control the display to the appropriate values. In addition to this, it is possible to insert commands into the program text, no matter whether this makes any sense or not. Please note that interactions during the execution of HALCON application can only be used in a sensible way, if the single operators have short runtimes, because HDevelop can only react within the “gaps”, that is, between the calls to the HALCON library.
Please note that neither the PC nor the BP can be set during the execution of the HALCON application.

8.4 Warning- and Error-Windows

Warning- and error-windows are popups, that make the user aware of user errors. Usually, they interrupt the faulty actions with a description of the error. For this purpose information about the kind of the error is determined during the execution. Figure 8.1 shows an example of an error window.

8.5 Restrictions

Not every HALCON operator that is available in HALCON/C or HALCON/C++ can or should be used in HDevelop. There are two reasons for this. On the one hand the HALCON system is influenced by HDevelop so deeply, that some operators don’t behave like in a normal user program. Usually this concernes the graphical operators like set_color. For this class of operators, specific versions for HDevelop are available, e.g., dev_set_color.
On the other hand some low-level operators exist (like reset_obj_db or clear_obj), that will bring HDevelop “out of balance.”

Figure 8.1: *Example for an error-window*

Not the whole functionality of HDevelop can be transferred to a C++-program, because the graphics windows of HDevelop are more comfortable than the simple HALCON windows.
However, the points described above are very special and will not bother the normal user, because the appropriate functions can be found in both working environments. If you use `dev_set_color` in HDevelop, for example, you would use `set_color` as its counterpart in HALCON/C++. Further restrictions can be found in section 6.1.4.

Appendix A

Control

`assign ( : : Input : Result )`

Assign a new value to a control variable.

`assign` assigns a new value to a variable. In HDevelop an assignment is treated like an operator. To use an assignment you have to select the operator `assign(Input,Result)`. This operator has the following semantics: It evaluates `Input` (right side of assignment) and stores it in `Result` (left side of assignment). However, in the program text the assignment is represented by the usual syntax of the assignment operator: '`:=`'. The following example outlines the difference between an assignment in C syntax and its transformed version in HDevelop:

The assignment in C syntax

```
u = sin(x) + cos(y);
```

is defined in HDevelop using the assignment operator as

```
assign(sin(x) + cos(y), u)
```

which is displayed in the program window as:

```
u := sin(x) + cos(y)
```

Parameter

- ▷ `Input` (input_control) . . . . . real(-array) ⇝ *real* / integer / string
  New value.
  Defaultvalue: 1
- ▷ `Result` (output_control) . . . . . real(-array) ⇝ *real* / integer / string
  Variable that has to be changed.

Example

```
Tuple1 := [1,0,3,4,5,6,7,8,9]
Val := sin(1.2) + cos(1.2)
Tuple1[1] := 2
Tuple2 := []
for i := 0 to 10 by 1
  Tuple2[i] := i
endfor
```

Result

`assign` returns 2 (H_MSG_TRUE) if the evaluation of the expression yields no error.

Alternatives

`insert`

Module

Basic operators

comment (: : Comment :)

Add a comment of one line to the program.

`comment` allows to add a comment of one line to the program. As parameter value, i.e. as comment, all characters are allowed. This operator has no effect on the program execution.

Parameter

▷ `Comment` (input_control) . . . . . string ⇝ *string*
Arbitrary sequence of characters.

Example

```
* This is a program width comments
* 'this is a string as comment'
* here are numbers: 4711, 0.815
stop ()
```

Result

`comment` always returns 2 (H_MSG_TRUE).

Module

Basic operators

exit (: : :)

Terminate HDevelop.

`exit` terminates HDevelop. The operator is aquivalent to the menu `File` ▷ `Quit`. Internally and for exported C++ code the C-function call `exit(0)` is used.

Example

```
read_image (Image, 'fabrik')
intensity (Image, Image, Mean, Deviation)
open_file ('intensity.txt', 'output', FileHandle)
fwrite_string (FileHandle, Mean + ' ' + Deviation)
close_file (FileHandle)
exit ()
```

Result

`exit` returns 0 (o.k.) to the calling environment of HDevelop = operating system.

See Also

`stop`

Module

Basic operators

for (: : Start, End, Step : Variable)

Execute the body for a fixed number.

The `for` loop is controlled by a start and termination value and an incrementation value that determines the number of loop steps. These values may also be expressions which are evaluated immediately before the loop is entered. The expressions may be of type `integer` or of type `real`. If all input values are of type `integer` the loop variable will also be of type `integer`. In all other cases the loop variable will be of type `real`. If the start value is less or equal to the termination value, the loop index is assigned with the starting value and the body of the loop is entered. If the increment is less than zero the loop is entered if the start value is larger or equal to the end value. Each time the body is executed, the loop index is incremented by the incrementation value. If the loop index is equal to the termination value, the body of the loop is performed for the last time. If the loop index is larger than the termination value the body will not be excecuted any longer. For negative increment values the loop is terminated if the loop index is less than the termination value.

Please note that it is not necessary, that the loop index has to be equal to the termination value before terminating the loop. The loop index is set to the termination value when the loop is being left. Please note, that the expressions for start and termination value are evaluated only once when entering the loop. A modification of a variable that appears within these expressions has no influence on the termination of the loop. The same applies to the modifications of the loop index. It also has no influence on the termination. The loop value is assigned to the correct value each time the `for` operator is executed.

If the `for` loop is left too early (e.g. if you press `Stop` and set the `PC`) and the loop is entered again, the expressions will be evaluated, as if the loop were entered for the first time.

Attention

For exported C++ please note the different semantics of the for loop.

Parameter

- ▷ `Start` (input_control) . . . . . number ⇝ *integer* / real
  Start value for the loop variable.
  Defaultvalue: 1
- ▷ `End` (input_control) . . . . . number ⇝ *integer* / real
  End value for the loop variable.
  Defaultvalue: 5
- ▷ `Step` (input_control) . . . . . number ⇝ *integer* / real
  Increment value for the loop variable.
  Defaultvalue: 1
- ▷ `Variable` (output_control) . . . . . number ⇝ *integer* / real
  Loop variable.

Example

```
dev_update_window ('off')
dev_close_window ()
dev_open_window (0, 0, 728, 512, 'black', WindowID)
read_image (Bond, 'die3')
dev_display (Bond)
stop ()
threshold (Bond, Bright, 100, 255)
shape_trans (Bright, Die, 'rectangle2')
dev_set_color ('green')
dev_set_line_width (3)
dev_set_draw ('margin')
dev_display (Die)
stop ()
reduce_domain (Bond, Die, DieGrey)
threshold (DieGrey, Wires, 0, 50)
fill_up_shape (Wires, WiresFilled, 'area', 1, 100)
dev_display (Bond)
dev_set_draw ('fill')
dev_set_color ('red')
dev_display (WiresFilled)
stop ()
opening_circle (WiresFilled, Balls, 15.5)
dev_set_color ('green')
dev_display (Balls)
stop ()
connection (Balls, SingleBalls)
select_shape (SingleBalls, IntermediateBalls, 'circularity', 'and', 0.85, 1.0)
sort_region (IntermediateBalls, FinalBalls, 'first_point', 'true', 'column')
dev_display (Bond)
dev_set_colored (12)
dev_display (FinalBalls)
stop ()
```

```
smallest_circle (FinalBalls, Row, Column, Radius)
NumBalls := |Radius|
Diameter := 2*Radius
meanDiameter := sum(Diameter)/NumBalls
mimDiameter := min(Diameter)
dev_display (Bond)
disp_circle (WindowID, Row, Column, Radius)
dev_set_color ('white')
set_font (WindowID, 'system26')
for i := 1 to NumBalls by 1
  if (fmod(i,2)=1)
    set_tposition (WindowID, Row[i-1]-1.5*Radius[i-1], Column[i-1]-60)
  else
    set_tposition (WindowID, Row[i-1]+2.5*Radius[i-1], Column[i-1]-60)
  endif
  write_string (WindowID, 'Diam: '+Diameter[i-1])
endfor
dev_set_color ('green')
dev_update_window ('on')
```

Result

`for` returns 2 (H_MSG_TRUE) if the evaluation of the expression yields no error. `endfor` (as operator) always returns 2 (H_MSG_TRUE)

Alternatives

`while`

See Also

`if`, `ifelse`

Module

Basic operators

`if ( : : Condition : )`

Conditional statement.

`if` is a conditional statement. The condition contains a boolean expression. If the condition is true, the body is executed. Otherwise the execution is continued at the first expression or operator call that follows the corresponding `endif`.

Parameter

▷ `Condition` (input_control) . . . . . integer ⇝ *integer*
Condition for the if statement.
Defaultvalue: 1

Result

`if` returns 2 (H_MSG_TRUE) if the evaluation of the expression yields no error. `endif` (as operators) always returns 2 (H_MSG_TRUE)

Alternatives

`ifelse`

See Also

`for`, `while`

Module

Basic operators

ifelse (: : Condition :)

Condition with alternative.

`ifelse` is a condition with an alternative. If the condition is true (i.e. not 0), all expressions and calls between the head and operator `endif` are performed. If the condition is false (i.e. 0) the part between `else` and `endif` is executed. Note that the operator is called `ilfelse` and it is displayed as `if` in the program text area.

Parameter

▷ `Condition` (input_control) . . . . . integer ⇝ *integer*
Condition for the if statement.
Defaultvalue: 1

Result

`ifelse` returns 2 (H_MSG_TRUE) if the evaluation of the expression yields no error. `else` and `endif` (as operators) always return 2 (H_MSG_TRUE)

Alternatives

`if`

See Also

`for`, `while`

Module

Basic operators

insert (: : Input, Value, Index : Result)

Assignment of a value into a tuple.

`insert` assigns a single value into an tuple. If the first input parameter and the first output parameter are identical, the call:

```
insert (Areas, Area, Radius-1, Areas)
```

is not presented in the program text as an operator call, but in the more intuitive form as:

```
Areas[Radius-1] := Area
```

.

Parameter

- ▷ `Input` (input_control) . . . . . real(-array) ⇝ *real* / integer / string
  Tuple, where the new value has to be inserted.
  Defaultvalue: '[]'
- ▷ `Value` (input_control) . . . . . real ⇝ *real* / integer / string
  Value that has to be inserted.
  Defaultvalue: 1
  Value Range: $0 \leq$ `Value` ≤ 1000000
- ▷ `Index` (input_control) . . . . . integer ⇝ *integer*
  Index position for new value.
  Defaultvalue: 0
  Value Suggestions: `Index` $\in \{0, 1, 2, 3, 4, 5, 6\}$
  Minimal Value Step: 1
- ▷ `Result` (output_control) . . . . . real(-array) ⇝ *real* / integer / string
  Result tuple with inserted values.

Result

`insert` returns 2 (H_MSG_TRUE) if the evaluation of the expression yields no error.

Alternatives

`assign`

Module

Basic operators

`stop ( : : : )`

Stop program execution.

`stop` stops the program execution of HDevelopṪhe `PC` is then placed at the program line behind `stop`. The operator is equivalent the presing the `stop` button in the menu bar.

Attention

`stop` is not available in C++.

Example

```
read_image (Image, 'fabrik')
regiongrowing (Image, Regions, 3, 3, 6, 100)
Number := |Regions|
dev_update_window ('off')
for i := 1 to Number by 1
  RegionSelected := Regions[i]
  dev_clear_window ()
  dev_display (RegionSelected)
  stop ()
endfor
```

Result

`stop` always returns 2 (H_MSG_TRUE)

See Also

`exit`

Module

Basic operators

`while ( : : Condition : )`

Continue to execute the body as long as the condition is true.

`while` executes the body as long as the condition is true. The `while` loop has a boolean expression as the conditional part. As long as it is `true` (i.e. not equal 0), the body of the loop is performed. In order to enter the loop, the condition has to be true in the first place.

Parameter

▷ `Condition` (input_control) ..... integer ⇝ *integer*
Condition for loop.

Example

```
dev_update_window ('off')
dev_close_window ()
dev_open_window (0, 0, 512, 512, 'black', WindowID)
read_image (Image, 'particle')
dev_display (Image)
stop ()
threshold (Image, Large, 110, 255)
dilation_circle (Large, LargeDilation, 7.5)
dev_display (Image)
dev_set_draw ('margin')
dev_set_line_width (3)
dev_set_color ('green')
dev_display (LargeDilation)
dev_set_draw ('fill')
stop ()
complement (LargeDilation, NotLarge)
reduce_domain (Image, NotLarge, ParticlesRed)
mean_image (ParticlesRed, Mean, 31, 31)
dyn_threshold (ParticlesRed, Mean, SmallRaw, 3, 'light')
opening_circle (SmallRaw, Small, 2.5)
connection (Small, SmallConnection)
dev_display (Image)
dev_set_colored (12)
dev_display (SmallConnection)
stop ()
```

```
dev_set_color ('green')
dev_display (Image)
dev_display (SmallConnection)
Button := 1
while (Button = 1)
  dev_set_color ('green')
  get_mbutton (WindowID, Row, Column, Button)
  dev_display (Image)
  dev_display (SmallConnection)
  dev_set_color ('red')
  select_region_point (SmallConnection, SmallSingle, Row, Column)
  dev_display (SmallSingle)
  NumSingle := |SmallSingle|
  if (NumSingle=1)
    intensity (SmallSingle, Image, MeanGray, DeviationGray)
    area_center (SmallSingle, Area, Row, Column)
    dev_set_color ('yellow')
    set_tposition (WindowID, Row, Column)
    write_string (WindowID, 'Area='+Area+', Int='+MeanGray)
  endif
endwhile
dev_set_line_width (1)
dev_update_window ('on')
```

Result

`while` returns 2 (H_MSG_TRUE) if the evaluation of the expression yields no error. `endwhile` (as operator) always returns 2 (H_MSG_TRUE)

Alternatives

`for`

See Also

`if`, `ifelse`

Module

Basic operators

Appendix B

Develop

`dev_clear_obj ( Objects : : : )`

Delete an iconic object from the HALCON database.

`dev_clear_obj` deletes iconic objects, which are no longer needed, from the HALCON database. It should be noted that `dev_clear_obj` cannot be exported to C++ due to the automatic memory management in C++.

Attention

Never use `clear_obj` to clear objects in HDevelop. The operator `dev_clear_obj` has to be used instead.

Parameter

▷ `Objects` (input_object) . . . . . object(-array) ⇝ *Hobject*
Objects to be deleted.

Result

`dev_clear_obj` returns 2 (H_MSG_TRUE) if the variable is instantiated. If necessary, an exception is raised.

See Also

`clear_obj`, `test_obj_def`, `dev_set_check`, `reset_obj_db`

Module

Basic operators

`dev_clear_window ( : : : )`

Clear the active graphics window.

`dev_clear_window` clears the graphics window content and the history of the active window. Parameters assigned to this window (e.g. with `dev_set_color`, `dev_set_draw`, etc.) remain unmodified. The operator is equivalent to pressing the `Clear` button of the active graphics window. A graphics window can be activated by calling `dev_set_window`.

Attention

If dev_clear_window should be used for exported Code (C++), please note the description of clear_window due to the different semantics in C++.

Example

```
read_image (Image, 'fabrik')
regiongrowing (Image, Regions, 3, 3, 6, 100)
Number := |Regions|
dev_update_window ('off')
for i := 1 to Number by 1
  RegionSelected := Regions[i]
  dev_clear_window ()
  dev_display (RegionSelected)
  * stop ()
endfor
```

Result

dev_clear_window always returns 2 (H_MSG_TRUE).

Possible Predecessor Functions

dev_set_window, dev_open_window, dev_display

Possible Successor Functions

dev_display

See Also

clear_window

Module

System operators

dev_close_inspect_ctrl (: : Variable :)

Close an inspect window of a control variable.

dev_close_inspect_ctrl is the opposite operator to dev_inspect_ctrl, and closes the inspect window corresponding to Variable. The window can also be closed by pressing the Close-button of the dialog.

Attention

This operator is not supported for exported C++ code.

Parameter

▷ Variable (input_control) . . . . . real(-array) ⇝ *real* / integer / string
Name of the variable which inspect window has to be closed.

Example

```
Var := 1
dev_inspect_ctrl (Var)
```

```
Var := [1,2,3,9,5,6,7,8]
Var[3] := 4
stop
dev_close_inspect_ctrl (Var)
```

Result

If an inspect window associated with `Variable` is open `dev_close_inspect_ctrl` returns 2 (H_MSG_TRUE).

Possible Predecessor Functions

`dev_inspect_ctrl`

Module

Basic operators

dev_close_window (: : :)

Close the active graphics window.

`dev_close_window` closes the active graphics window which have been opened by `dev_open_window` or by HDevelop (default window). The operator is equivalent to pressing the `Close` button of the active window. A graphics window can be activated by calling `dev_set_window`.

Attention

If `dev_close_window` should be used for exported Code (C++), please note the description of `close_window` due to the different semantics in C++.

Example

```
* close all windows
for i := 1 to 10 by 1
  dev_close_window ()
endfor
read_image (For5, 'for5')
get_image_pointer1 (For5, Pointer, Type, Width, Height)
dev_open_window (0, 0, Width, Height, 'black', WindowHandle)
dev_display (For5)
```

Result

`dev_close_window` always returns 2 (H_MSG_TRUE).

Possible Predecessor Functions

`dev_set_window`, `dev_open_window`

Possible Successor Functions

`dev_open_window`

See Also

`close_window`

Module

System operators

dev_display (Object : : :)

Displays image objects in the current graphics window.

dev_display displays an image object (image, region, or XLD) in the active graphics window. This is equivalent to a double click on an icon variable inside the variable window.

Attention

If dev_display should be used for exported Code (C++), please note the description of disp_obj due to the different semantics in C++.

Parameter

▷ Object (input_object) . . . . . object(-array) ⇝ *Hobject*
Image objects to be displayed.

Example

```
read_image (Image, 'fabrik')
regiongrowing (Image, Regions, 3, 3, 6, 100)
dev_clear_window ()
dev_display (Image)
dev_set_colored (12)
dev_set_draw ('margin')
dev_display (Regions)
```

Result

dev_display always returns 2 (H_MSG_TRUE)

Alternatives

disp_object, disp_image, disp_region, disp_xld

See Also

dev_set_color, dev_set_colored, dev_set_draw, dev_set_line_width

Module

System operators

dev_error_var (: : ErrorVar, Mode :)

Define or undefine an error variable.

dev_error_var defines an error variable, i.e. a variable which contains the status of the last call of an operator. ErrorVar will be H_MSG_TRUE (2) if no error had occured. The parameter Mode specifies if the error variable should be used (1) or not (0). If an error variable is active it will be updated each an operator excetion is finished. Thus a value is only valid until the next call of an operator. The value can be saved by assigning it to another variable (see example) or by calling dev_error_var(ErrorVar,0).

Attention

If dev_error_var should be used for exported Code (C++), please note the different handling of return values in C++.

Parameter

- ▷ ErrorVar (input_control) ..... integer(-array) ⇝ *integer*
  Name of the variable which shall contain the error status.
- ▷ Mode (input_control) ..... integer ⇝ *integer*
  Switch the error variable on or off.
  Defaultvalue: 1
  Value List: Mode ∈ {0, 1}

Example

```
dev_close_window ()
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
dev_error_var (Error, 1)
dev_set_check ('~give_error')
FileName := 'wrong_name'
read_image (Image, FileName)
ReadError := Error
if (ReadError # H_MSG_TRUE)
  write_string (WindowHandle, 'wrong file name: '+FileName)
endif
```

Result

dev_error_var always returns 2 (H_MSG_TRUE)

Possible Predecessor Functions

dev_set_check

Possible Successor Functions

dev_set_check, if, ifelse, assign

See Also

set_check

Module

Basic operators

dev_inspect_ctrl (: : Variable :)

Open a window to inspect a control variable.

dev_inspect_ctrl opens a dialog to check the contents of a control variable. This dialog has a scrolled list with all the values of the variable. In the case of an framegrabber handle a specific dialog is opened which displays the most important framegrabber parameters and can be used to switch the framegrabber only interactively. The contents of the dilaog will be updated whenever the value(s) of variable changes. The update mode can influenced by the operator dev_update_ctrl. The dialog can be closed by pressing the Close-button or by calling dev_close_inspect_ctrl.

Attention

This operator is not supported for exported C++ code.

Parameter

▷ `Variable` (input_control) . . . . . integer(-array) ⇝ *integer* / real / string
Name of the variable to be checked.

Example

```
read_image (Image, 'fabrik')
regiongrowing (Image, Regions, 3, 3, 6, 100)
area_center (Regions, Area, Row, Column)
dev_inspect_ctrl (Area)
```

Result

`dev_inspect_ctrl` always returns 2 (H_MSG_TRUE)

See Also

`dev_update_var`

Module

Basic operators

dev_map_par (: : :)

Open the dialog to specify the display parameters.

`dev_map_par` opens the dialog which can also be accessed from the menu `Visualization` ▷ `Set Parameters...`. The dialog is used to configure the modes to display data like images, regions, or polygons.

Attention

This operator is not supported for exported C++ code.

Example

```
read_image (Image, 'fabrik')
threshold (Image, Region, 128, 255)
dev_map_par ()
```

Result

`dev_map_par` always returns 2 (H_MSG_TRUE)

Possible Successor Functions

`dev_unmap_par`

Module

Basic operators

dev_map_prog (: : :)

Make the main window of HDevelop visible.

dev_map_prog is used to map the main window of HDevelop after it has been unmapped by dev_unmap_prog.

Attention

This operator is not supported for exported C++ code.

Depending on the operating system or the window manager the execution of dev_map_prog will result only in a visible icon of the window. In this case it has to be opened by the user with mouse interaction.

Result

dev_map_prog always returns 2 (H_MSG_TRUE)

Possible Predecessor Functions

dev_unmap_prog

Possible Successor Functions

dev_unmap_prog

See Also

dev_map_par, dev_map_var

Module

Basic operators

dev_map_var (: : :)

Map the variable window on the screen.

dev_map_var maps the variable window on the screen (i.e. makes it visible) that has been unmapped using dev_unmap_var.

Attention

This operator is not supported for exported C++ code.

Result

dev_map_var always returns 2 (H_MSG_TRUE)

Possible Predecessor Functions

dev_unmap_var

Possible Successor Functions

dev_unmap_var

See Also

dev_map_par, dev_map_prog

Module

Basic operators

dev_open_window (: : Row, Column, Width, Height, Background : WindowHandle)

Open a graphics window.

dev_open_window opens a new graphics window, which can be used to perform output of gray value data, regions, and graphics as well as to perform textual output. This new window automatically becomes active, which means that all output (dev_display and automatical display of operator results) is redirected to this window. This is shown by the green dot in the Active button.

In the case of the standard display operators (like disp_image, disp_region, disp_line, etc.) instead of dev_display the logical window number WindowHandle has to be used.

The background of the created window is set to the color specified in Background.

Pressing the Clear button clears the graphics window contents and the history of the window. This can also be achived by using the operator dev_clear_window. You close a graphics window using the Close button of the window frame or by calling dev_close_window.

The the origin of the graphics window is the upper left corner with the coordinates (0,0). The x values (column) increase from left to right, the y values increase from top to bottom. Normally, the coordinate system of the graphics window corresponds to the the most recently displayed image, which is automatically zoomed so that every pixel of the image is visible. The coordinate system can be changed interactively using the menu Visualization ▷ Set Parameters ▷ Zoom or with the operator dev_set_part. Every time an image with a different size is displayed, the coordinate system will be adapted automatically.

Each window has a history which contains all

- objects and
- display parameters

which have been displayed or changed since the most recent clear action or display of a full image. This history is used for redrawing the contents of the window. Other output like text or general graphics like disp_line or disp_circle or iconic data that is displayed using HALCON operators like disp_image or disp_region are **not** part of the history, and are **not** redrawn. Only the object classes image, region, and XLD that are displayed with the HDevelop operator dev_display or by double clicking on an icon are part of the history.

You may change the size of the graphics window interactively by "gripping" the window border with the mouse. Then you can resize the window by dragging the mouse pointer. After this size modification the window content is redisplayed. Now you see the same part of the window with changed zoom.

If the mouse cursor is inside the window its look-up-table is reactivated. This is necessary if other programs use their own look-up table. Thus if there is a "strange" graphics window presentation, you may load the proper look-up table by placing the mouse inside the window.

Opening a window causes the assignment of a default font. It is used in connection with procedures like write_string and you may overwrite it by performing set_font after calling

dev_open_window. On the other hand you have the possibility to specify a default font by calling set_system('default_font',<Fontname>) before opening a window (and all following windows; see also query_font).

If you want to specify display parameters for a window you may select the menu item Visualization in the menu bar. Here you can set the appropriate parameters by clicking the desired item. Parameters which you have set in this way are used for **all** windows (in contrast to standard windows opened with open_window). The effects of the new parameters will be applied direcly to the **last** object of the window history and alter its parameters only.

Attention

Never use close_window to close an HDevelop graphics window. The operator dev_close_window has to be used instead.

If dev_open_window should be used for exported Code (C++), please note the description of open_window due to the different semantics in C++.

Parameter

▷ Row (input_control) . . . . . rectangle.origin.y ⇝ *integer*
Row index of upper left corner.
Defaultvalue: 0
Value Range: 0 ≤ Row
Minimal Value Step: 1
Recommended Value Step: 1
Restriction: Row ≥ 0

▷ Column (input_control) . . . . . rectangle.origin.x ⇝ *integer*
Column index of upper left corner.
Defaultvalue: 0
Value Range: 0 ≤ Column
Minimal Value Step: 1
Recommended Value Step: 1
Restriction: Column ≥ 0

▷ Width (input_control) . . . . . rectangle.extent.x ⇝ *integer*
Width of the window.
Defaultvalue: 256
Value Range: 0 ≤ Width
Minimal Value Step: 1
Recommended Value Step: 1
Restriction: (Width > 0) ∨ (Width = -1)

▷ Height (input_control) . . . . . rectangle.extent.y ⇝ *integer*
Height of the window.
Defaultvalue: 256
Value Range: 0 ≤ Height
Minimal Value Step: 1
Recommended Value Step: 1
Restriction: (Height > 0) ∨ (Height = -1)

▷ Background (input_control) . . . . . integer ⇝ *integer* / string
Color of the background of the new window.
Defaultvalue: ”black”

▷ `WindowHandle` (output_control) . . . . . window ⇝ *integer*
Window identifier.

Example

```
dev_close_window ()
read_image (For5, 'for5')
get_image_pointer1 (For5, Pointer, Type, Width, Height)
dev_open_window (0, 0, Width, Height, 'black', WindowHandle)
dev_display (For5)
dev_set_lut ('rainbow')
dev_display (For5)
stop ()
dev_set_lut ('default')
dev_display (For5)
stop ()
dev_set_part (100, 100, 300, 300)
dev_display (For5)
```

Result

If the values of the specified parameters are correct `dev_open_window` returns 2 (H_MSG_TRUE). If necessary an exception handling is raised.

Possible Successor Functions

`dev_display`, `dev_set_lut`, `dev_set_color`, `dev_set_draw`, `dev_set_part`

Alternatives

`open_window`

See Also

`query_color`

Module

System operators

`dev_set_check` (: : `Mode` :)

Specify the error handling.

`dev_set_check` sepcifies how HDevelop should react if an error occures. If `Mode` has the value **'give_error'** – which is the system default – HDevelop stops the program execution if an exception occures and displays an error message. If you use **'˜give_error'** the exception will be ignored and the program continues. `dev_set_check` is intended to used in connection with `dev_error_var`, which allows to check for the result state of an operator.

Attention

If `dev_set_check` should be used for exported Code (C++), please note the description of `set_check` due to the different semantics in C++.

Parameter

▷ `Mode` (input_control) . . . . . string ⇝ *string*
Mode of error handling.
Defaultvalue: 'give_error'

Example

```
dev_close_window ()
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
dev_error_var (Error, 1)
dev_set_check ('~give_error')
FileName := 'wrong_name'
read_image (Image, FileName)
dev_set_check ('give_error')
ReadError := Error
if (ReadError # H_MSG_TRUE)
  write_string (WindowHandle, 'wrong file name: '+FileName)
endif
* Now the program will stop with an exception
read_image (Image, FileName)
```

Result

`dev_set_check` always returns 2 (H_MSG_TRUE)

Possible Successor Functions

`dev_error_var`

See Also

`set_ckeck`

Module

System operators

dev_set_color (: : ColorName :)

Set output color.

`dev_set_color` defines the color for region and line oriented output in the graphics windows. The available colors can be queried with the operator `query_color`. The "colors" 'black' and 'white' are available for all screens. If colors are used that are not displayable on the screen, HALCON can choose a similar, displayable color of the output. For this, `set_check('~color')` must be called.

The defined color is used until `dev_set_color` or `dev_set_colored` is called.

Colors are defined for all graphics windows in contrast to the operator `set_color`.

Attention

If `dev_set_color` should be used for exported Code (C++), please note the description of `set_color` due to the different semantics in C++.

Parameter

▷ `ColorName` (input_control) . . . . . string(-array) ⇝ *string*
Output color names.
Defaultvalue: 'white'
Value Suggestions: `ColorName` ∈ {'white', 'black', 'grey', 'red', 'green', 'blue'}

Example

```
read_image(Image,'mreut')
dev_set_draw('fill')
dev_set_color('red')
threshold(Image,Region,180,255)
dev_set_color('green')
threshold(Image,Region,0,179)
```

Result

`dev_set_color` always returns 2 (H_MSG_TRUE)

Possible Predecessor Functions

`dev_open_window`, `query_color`, `query_all_colors`

Possible Successor Functions

`dev_display`

Alternatives

`dev_set_colored`

See Also

`dev_set_draw`, `dev_set_line_width`, `set_color`

Module

System operators

`dev_set_colored ( : : NumColors : )`

Set multiple output colors.

`dev_set_colored` allows the user to display a tuple of regions in different colors. `NumColors` defines the number of colors that are used. Valid values for `NumColors` can be queried with `query_colored`.

Attention

If `dev_set_colored` should be used for exported Code (C++), please note the description of `set_colored` due to the different semantics in C++.

Parameter

▷ `NumColors` (input_control) . . . . . integer ⇝ *integer*
Number of output colors.
Defaultvalue: 6
Value List: `NumColors` ∈ {3, 6, 12}

Example

```
read_image(Image,'monkey')
threshold(Image,Region,128,255)
dev_set_colored(6)
connection(Region,Regions)
```

Result

`dev_set_colored` always returns 2 (H_MSG_TRUE)

Possible Predecessor Functions

`dev_open_window`

Possible Successor Functions

`dev_display`

Alternatives

`dev_set_color`

See Also

`dev_set_draw`, `dev_set_line_width`, `set_colored`

Module

System operators

dev_set_draw (: : DrawMode :)

Define the region fill mode.

`dev_set_draw` defines the region fill mode. If `DrawMode` is set to 'fill', output regions are filled, if set to 'margin', only contours are displayed. It is used by region output like `dev_display`, `disp_region`, `disp_circle`, `disp_rectangle1`, `disp_rectangle2`, `disp_arrow`, etc. If the mode is 'margin', the contour can be affected by `dev_set_line_width`, `set_line_approx` and `set_line_style`.

Attention

If `dev_set_draw` should be used for exported Code (C++), please note the description of `set_draw` due to the different semantics in C++.

Parameter

▷ `DrawMode` (input_control) . . . . . string ⇝ *string*
Fill mode for region output.
Defaultvalue: 'fill'
Value List: `DrawMode` ∈ {'fill', 'margin'}

Example

```
read_image(Image,'monkey')
threshold(Image,Region,128,255)
dev_clear_window
dev_set_color('red')
```

```
dev_set_draw('fill')
dev_display(Region)
dev_set_color('white')
dev_set_draw('margin')
dev_display(Region)
```

Result

`dev_set_draw` always returns 2 (H_MSG_TRUE)

Possible Successor Functions

`dev_set_line_width`, `dev_display`

See Also

`set_draw`

Module

System operators

dev_set_line_width (: : LineWidth :)

Define the line width for region contour output.

`dev_set_line_width` defines the line width (in pixel) in which a region contour or lines are displayed (e.g. with `dev_display`, `disp_region`, `disp_line`, `disp_polygon`, etc.).

Attention

If `dev_set_line_width` should be used for exported Code (C++), please note the description of `set_line_width` due to the different semantics in C++.

Parameter

▷ `LineWidth` (input_control) . . . . . integer ⇝ *integer*
Line width for region output in contour mode.
Defaultvalue: 1
Restriction: `LineWidth` ≥ 1

Example

```
read_image(Image,'monkey')
threshold(Image,Region,128,255)
dev_set_draw('margin')
dev_set_line_width(5)
dev_clear_window
dev_display(Region)
```

Result

`dev_set_line_width` always returns 2 (H_MSG_TRUE)

Possible Successor Functions

`dev_display`

See Also

`set_line_with`, `query_line_width`

Module

System operators

`dev_set_lut ( : : LutName : )`

Set "look-up-table" (lut).

`dev_set_lut` sets look-up-table of the the output window. A look-up-table defines the transformation of a "gray value" within an image into a gray value or color on the screen. It describes the screen gray value/color as a combination of red, green and blue for any image gray value (0..255) (so it is a 'table' to 'look up' the screen gray value/color for each image gray value: look-up-table). Transformation into screen-colors is performed in real-time at every time the screen is displayed new (typically this happens about 60 - 70 times per second). So it is possible to change the look-up-table to get a new look of images or regions. Please remind that not all machines support changing the look-up-table (e.g. monochrome resp. truecolor).

For common monitors only one look-up-table can be loaded per screen. Whereas `dev_set_lut` can be activated separately for each window. There is the following solution for this problem: It will always be activated the look-up-table that is assigned to the "active window" (a window is set into the state "active" by placing the mouse inside the window).

look-up-tables can also be used with truecolor displays. In this case the look-up-table will be simulated in software. This means, that the look-up-table will be used each time an image is displayed.

`query_lut` lists the names of all look-up-tables.

Attention

If `dev_set_lut` should be used for exported Code (C++), please note the description of `set_lut` due to the different semantics in C++.

Parameter

▷ `LutName` (input_control) . . . . . string ⇝ *string*
Name of look-up-table, values of look-up-table (RGB) or file name.
Defaultvalue: 'default'
Value Suggestions: `LutName` ∈ {'default', 'linear', 'inverse', 'sqr', 'inv_sqr', 'cube', 'inv_cube', 'sqrt', 'inv_sqrt', 'cubic_root', 'inv_cubic_root', 'color1', 'color2', 'color3', 'color4', 'three', 'six', 'twelfe', 'twenty_four', 'rainbow', 'temperature', 'cyclic_gray', 'cyclic_temperature', 'hsi', 'change1', 'change2', 'change3'}

Example

```
read_image(Image,'mreut')
dev_set_lut('inverse')
* For true color only:
dev_display(Image)
```

Result

`dev_set_lut` always returns 2 (H_MSG_TRUE)

Possible Successor Functions

`dev_display`

See Also

`set_lut`

Module

System operators

dev_set_paint (: : Mode :)

Define the grayvalue output mode.

`dev_set_paint` defines the output mode for grayvalue display in the graphics window. The mode is used by `dev_display`.

This page describes the different modes, that can be used for grayvalue output. It should be noted, that the mode 'default' is the most suitable.

A different way to display grayvalues is the histogram (mode: 'histogram'). This mode has three additional parameter values: Row (second value) and column (third value). They denote row and column of the histogram center for positioning on the screen. The scale factor (fourth value) determines the histogram size: a scale factor of 1 distinguishes 256 grayvalues, 2 distinguishes 128 grevalues, and so on. The four values are passed as a tuple, e.g. ['histogram', 256,256,1]. If only the first value is passed ('histogram'), the other values are set to defaults or the last values, respectively. For histogram computation see `gray_histo`.

The modes 'line' and 'column' allow to display gray values along lines or columns, respecively. The position (line- and columnindex) is passed with the second paramter value. The third parameter value is the scale factor in percent (100 means 1 pixel per grayvalue, 50 means one pixel per two grayvalues).

Gray images can also be interpreted as 3d data, depending on the grayvalue. To view these 3d plots, select the modes 'contourline', '3D-plot' or '3D-plot_hidden'.

Paramters for modes that need more than one parameter can be passed the following ways:

- Only the name of the mode is passed: the defaults or the last values are used, respectively. Example: `dev_set_paint('contourline')`
- All values are passed: all output characteristics can be set. Example: `dev_set_paint(['contourline',10,1])`
- Only the first n values are passed: only the passed values are changed. Example: `dev_set_paint(['contourline',10])`

Attention

If `dev_set_paint` should be used for exported Code (C++), please note the description of `set_paint` due to the different semantics in C++.

Parameter

▷ `Mode` (input_control) . . . . . string-array ⇝ *string* / integer
Grevalue output name. Additional parameters possible.
Defaultvalue: 'default'
Value List: `Mode` ∈ {'default', 'histogram', 'line', 'column', 'contourline', '3D-plot', '3D-plot_hidden', '3D-plot_point'}

Example

```
read_image(Image,'fabrik')
dev_set_paint('3D-plot')
dev_display(Image)
```

Possible Predecessor Functions

`dev_open_window`

Possible Successor Functions

`dev_set_color`, `dev_display`

See Also

`set_paint`

Module

System operators

dev_set_part (: : Row1, Column1, Row2, Column2 :)

Modify the displayed image part.

`dev_set_part` modifies the image part that is displayed in the graphics window. (`Row1`,`Column1`) denotes the upper left corner and (`Row2`,`Column2`) the lower right corner of the image part to display.

If `Row1` is larger than `Row2` the zooming will be reset. That means that the last displayed image will be completetly visible. Please note that this is not possible with the operator `set_part` outside HDevelop.

Attention

If `dev_set_part` should be used for exported Code (C++), please note the description of `set_part` due to the different semantics in C++.

Parameter

▷ `Row1` (input_control) . . . . . rectangle.origin.y ⇝ *integer*
Row of the upper left corner of the chosen image part.
Defaultvalue: 0

▷ `Column1` (input_control) . . . . . rectangle.origin.x ⇝ *integer*
Column of the upper left corner of the chosen image part.
Defaultvalue: 0

▷ `Row2` (input_control) . . . . . rectangle.corner.y ⇝ *integer*
Row of the lower right corner of the chosen image part.
Defaultvalue: 128

▷ `Column2` (input_control) . . . . . rectangle.corner.x ⇝ *integer*
Column of the lower right corner of the chosen image part.
Defaultvalue: 128

Example

```
read_image (Image, 'fabrik')
for i := 1 to 240 by 10
  dev_set_part (i, i, 511-i, 511-i)
  dev_display (Image)
endfor
dev_set_part (1, 1, -1, -1)
dev_display (Image)
```

Result

`dev_set_part` always returns 2 (H_MSG_TRUE)

Possible Successor Functions

`dev_display`

See Also

`set_part`

Module

System operators

dev_set_shape (: : Shape :)

Define the region output shape.

`dev_set_shape` defines the shape for region output. The output shape is used by `dev_display` for regions. The available shapes can be queried with `query_shape`.

Available modes:

'original': The shape is displayed unchanged. Nevertheless modifications via parameters like `dev_set_line_width` can take place. This is also true for all other modes.

'outer_circle': Each region is displayed by the smallest surrounding circle. (See `smallest_circle`.)

'inner_circle': Each region is displayed by the largest included circle. (See `inner_circle`.)

'ellipse': Each region is displayed by an ellipse with the same moments and orientation (See `elliptic_axis`.)

'rectangle1': Each region is displayed by the smallest surrounding rectangle parallel to the coordinate axes. (See `smallest_rectangle1`.)

'rectangle2': Each region is displayed by the smallest surrounding rectangle. (See `smallest_rectangle2`.)

'convex': Each region is displayed by its convex hull (See `shape_trans`.)

'icon' Each region is displayed by the icon set with `set_icon` in the center of gravity.

Attention

If `dev_set_shape` should be used for exported Code (C++), please note the description of `set_shape` due to the different semantics in C++.

Parameter

▷ `Shape` (input_control) ..... string ⇝ *string*
Region output mode.
Defaultvalue: 'original'
Value List: `Shape` ∈ {'original', 'convex', 'outer_circle', 'inner_circle', 'rectangle1', 'rectangle2', 'ellipse', 'icon'}

Example

```
read_image(Image,'monkey')
threshold(Image,Region,128,255)
connection(Region,Regions)
dev_set_shape('rectangle1')
dev_set_draw('margin')
dev_display(Regions)
```

Possible Successor Functions

`dev_display`, `dev_set_color`

See Also

`set_shape`, `dev_set_line_with`

Module

System operators

dev_set_window (: : WindowID :)

Activate a graphics window.

`dev_set_window` activates a graphics window. This is equivalent to pressing the `Active` button of the graphics window.

Attention

If `dev_set_window` should be used for exported Code (C++), please note the different handling of windows in C++.

`dev_set_window` is not supported for C++.

Parameter

▷ `WindowID` (input_control) ..... window ⇝ *integer*
Window_id.

Example

```
dev_open_window (1, 1, 200, 200, 'black', WindowID1)
dev_open_window (1, 220, 200, 200, 'black', WindowID2)
read_image(Image,'monkey')
dev_set_window(WindowID1)
dev_display(Image)
dev_set_window(WindowID2)
dev_display(Image)
```

Possible Predecessor Functions

`dev_open_window`

Possible Successor Functions

`dev_display`

Module

Basic operators

`dev_set_window_extents ( : : Row, Column, Width, Height : )`

Change position and size of a graphics window.

`dev_set_window_extents` changes the position and/or the size of the currently active graphics window.

The parameters `Row` and `Column` specify the new position (upper left corner) of the window. If one of both values is negative, the position will remain unchanged. The parameters `Width` and `Height` specify the new size of the window. This is the size of the inner part that actually displayes the data. If one of the two values is negative, the size will remain unchanged.

Attention

Never use `set_window_extents` to change the size and position of an HDevelop graphics window. The operator `dev_set_window_extents` has to be used instead.

Parameter

▷ `Row` (input_control) . . . . . rectangle.origin.y ⇝ *integer*
Row index of upper left corner.
Defaultvalue: 0
Value Range: $0 \leq$ `Row`
Minimal Value Step: 1
Recommended Value Step: 1
Restriction: $(\texttt{Row} \geq 0) \vee (\texttt{Row} = -1)$

▷ `Column` (input_control) . . . . . rectangle.origin.x ⇝ *integer*
Column index of upper left corner.
Defaultvalue: 0
Value Range: $0 \leq$ `Column`
Minimal Value Step: 1
Recommended Value Step: 1
Restriction: $(\texttt{Column} \geq 0) \vee (\texttt{Column} = -1)$

▷ `Width` (input_control) ..... rectangle.extent.x ⇝ *integer*
Width of the window.
Defaultvalue: 256
Value Range: $0 \leq$ `Width`
Minimal Value Step: 1
Recommended Value Step: 1
Restriction: $(\texttt{Width} > 0) \vee (\texttt{Width} = -1)$

▷ `Height` (input_control) ..... rectangle.extent.y ⇝ *integer*
Height of the window.
Defaultvalue: 256
Value Range: $0 \leq$ `Height`
Minimal Value Step: 1
Recommended Value Step: 1
Restriction: $(\texttt{Height} > 0) \vee (\texttt{Height} = -1)$

Example

```
dev_close_window ()
read_image (For5, 'for5')
get_image_pointer1 (For5, Pointer, Type, Width, Height)
dev_open_window (0, 0, Width, Height, 'black', WindowHandle)
dev_display (For5)
stop ()
dev_set_window_extents (-1,-1,Width/2,Height/2)
dev_display (For5)
stop ()
dev_set_window_extents (200,200,-1,-1)
```

Result

If the values of the specified parameters are correct `dev_set_window_extents` returns 2 (H_MSG_TRUE). If necessary an exception handling is raised.

Possible Successor Functions

`dev_display`, `dev_set_lut`, `dev_set_color`, `dev_set_draw`, `dev_set_part`

See Also

`set_window_extents`

Module

System operators

`dev_unmap_par ( : : : )`

Hide the window for the graphic parameters.

`dev_unmap_par` hides the window for the graphic parameters so that it is no longer visible. It can be mapped again using the operator `dev_map_par`.

Attention

This operator is not supported for exported C++ code.

Result

dev_unmap_par always returns 2 (H_MSG_TRUE)

Possible Successor Functions

dev_map_prog

See Also

dev_map_par, dev_map_prog, dev_map_var

Module

Basic operators

dev_unmap_prog (: : :)

Hide the main window.

dev_unmap_prog hides the main window so that it is no longer visible. It can be mapped again using the operator dev_map_prog.

Attention

This operator is not supported for exported C++ code.

Result

dev_unmap_prog always returns 2 (H_MSG_TRUE)

Possible Successor Functions

dev_map_prog, stop

See Also

dev_map_par, dev_map_prog, dev_map_var

Module

Basic operators

dev_unmap_var (: : :)

Hide the variable window.

dev_unmap_var hides the variable window so that it is no longer visible. It can be mapped again using the operator dev_map_var.

Attention

This operator is not supported for exported C++ code.

Result

dev_unmap_var always returns 2 (H_MSG_TRUE)

Possible Successor Functions

dev_map_var

See Also

dev_map_par, dev_map_prog

Module

Basic operators

dev_update_pc (: : DisplayMode :)

Specify the behaviour of the PC during program execution.

dev_update_pc specifies the behaviour of the PC during program execution. In the mode 'on' (default) the PC is always displayed in front of the current operator. In addition the program text is scrolled – if necessary – so that the current operator is visible. In the mode 'off' the PC is not visible during program execution and the program text will not be scrolled automatically.

This option can also be controled by the dialog

File ▷ Options ▷ Update PC.

Attention

This operator is not supported for exported C++ code.

Parameter

▷ DisplayMode (input_control) ..... string ⇝ *string*
Mode for runtime behaviour.
Defaultvalue: 'off'
Value List: DisplayMode ∈ {'on', 'off'}

Result

dev_update_pc always returns 2 (H_MSG_TRUE)

See Also

dev_update_time, dev_update_window, dev_update_var

Module

Basic operators

dev_update_time (: : DisplayMode :)

Switch time measurement for operators on or off.

dev_update_time controls if the execution time of an operator has to be measured.

This option can also be controled by the dialog

File ▷ Options ▷ Show Processing Time.

Attention

This operator is not supported for exported C++ code.

Parameter

▷ DisplayMode (input_control) ..... string ⇝ *string*
Mode for graphic output.
Defaultvalue: 'off'
Value List: DisplayMode ∈ {'on', 'off'}

Result

dev_update_time always returns 2 (H_MSG_TRUE)

See Also

`dev_update_pc`, `dev_update_window`, `dev_update_var`

Module

Basic operators

dev_update_var (: : DisplayMode :)

Specify the behaviour of the variable window during program execution.

`dev_update_var` specifies the behaviour of the variable window during program execution. Using the mode 'on' (default) the contents of the variable window (iconic and control variables) is updated each time a variable is modified by the program. In the mode 'off' the variables are updated only when the execution is finished. Please not that update in this contents only means the graphical representation of the internal values in the variable window.

This option can also be controled by the dialog

`File` ▷ `Options` ▷ `Update Variables`.

Attention

This operator is not supported for exported C++ code.

Parameter

▷ `DisplayMode` (input_control) ..... string ⇝ *string*
Mode for graphic output.
Defaultvalue: 'off'
Value List: `DisplayMode` ∈ {'on', 'off'}

Result

`dev_update_var` always returns 2 (H_MSG_TRUE)

See Also

`dev_update_pc`, `dev_update_window`, `dev_update_time`

Module

Basic operators

dev_update_window (: : DisplayMode :)

Specify the output behaviour during program execution.

`dev_update_window` specifies the output behaviour during program execution. By default every object (image, region, or XLD) is displayed in the active graphics window. This can be changed by using the value 'off' for `DisplayMode`. In this case objects are only displayed in single step mode. Here one would use the operator `dev_display` to output objects.

This option can also be controled by the dialog

`File` ▷ `Options` ▷ `Update Window`.

Attention

This operator is not supported for exported C++ code.

Parameter

▷ `DisplayMode` (input_control) . . . . . string ⇝ *string*
Mode for graphic output.
Defaultvalue: 'off'
Value List: `DisplayMode` ∈ {'on', 'off'}

Result

`dev_update_window` always returns 2 (H_MSG_TRUE)

Possible Successor Functions

`dev_display`

See Also

`dev_update_pc`, `dev_update_var`, `dev_update_time`

Module

Basic operators

Appendix C

Glossary

Boolean is the type name for the truth values `true` and `false` as well as for the related boolean expressions.

Body A body is part of a conditional instruction (`if`) or a loop (`while` or `for`) and consists of a sequence of operator calls. If you consider the `for`-loop, for instance, all operator calls, that are located between `for` and `endfor` form the body.

Button A button is part of a graphical user interface. With the mouse the user can press a button to cause an action to be performed.

Control data Control data can be either numbers (↑`integer` and ↑`real`), character strings (↑`string`) and truth values (`boolean`). This data can be used as atomic values (i.e., single values) or as ↑tuples (i.e., arrays of values).

Empty region An empty ↑region contains no points at all, i.e., its area is zero.

Graphics window A graphics window is used in ↑HDevelop for displaying ↑images, ↑regions, or ↑XLD.

HDevelop is an interactive program for the creation of HALCON applications.

Iconic data are image data, i.e., image arrays and data, which are described by coordinates and are derived from image arrays, e.g., ↑regions, ↑image and ↑XLD.

Image An image consists of one or more (multichannel image) image arrays and a ↑region as the definition domain. All image arrays have the same dimension, but they can be of different pixel types. The size of the ↑region is smaller or equal than the size of the image arrays. The ↑region determines all image points that should be processed.

Iconic object Generic implementation of ↑iconic data in HALCON.

integer is the type name for integer numbers. Integers are implemented using the C-type `long` (4 or 8 byte).

Operator data base The operator data base contains information about the HALCON operators. They are loaded at runtime from the binary files in `%HALCONROOT%\help`.

Program window In HDevelop the program window contains the program. It is used to edit (copy, delete, and paste lines) and to run or debug the program.

Operator window In the operator window of HDevelop the parameters of the selected operators can be entered or modified.

Real is the type name for floating point numbers. They are implemented using the C-type `double` (8 bytes).

Region A region is a set of image points without gray values. A region can be imagined as a binary image (mask). Regions are implemented using runlength encoding. The region size is not limited to the image size (see also `set_system('clip_region','true'/'false')` in the HALCON reference manual.

String is the type name for character strings. A string starts and ends with a single quote; in between any character can be used except single quote. The empty string consists of two consecutive single quotes. The maximum length of a character string is limited to 1024 characters.

Tuple A tuple is an ordered multivalue set. In case of ↑control data a tuple can consist of a large number of items with different data types. The term tuple is also used in conjunction with ↑iconic objects, if it is to be emphasized that several ↑iconic objects will be used.

Type ↑iconic variables can be assigned with data items of type ↑image, ↑region, and ↑XLD. The types of ↑control data items can be one of ↑`integer`, ↑`real`, ↑`boolean`, or ↑`string`.

Variable window In HDevelop the variable window manages the ↑control and ↑iconic data.

XLD is the short term for e*X*tended *L*ine *D*escription. It is used as a superclass for contours, polygons, and lines (see also the HALCON Reference Manual).

Index

Bibliography

[Jai89] A.K. Jain *Fundamentals of Digital Image Processing*, Prentice–Hall International Editions, Englewood Cliffs, New Jersey, 1989

[Bal82] D.H. Ballard, C.M. Brown *Computer Vision*, Prentice–Hall, Englewood Cliffs, New Jersey, 1982

[Rus92] J.C. Russ *The Image Processing Handbook*, CRC Press, Boca Raton, Florida, 1992

[Radig93] B. Radig (Hrsg.) *Verarbeiten und Verstehen von Bildern*, Oldenbourg, München, 1993

[Abmayr94] W. Abmayr *Einführung in die digitale Bildverarbeitung*, B.G.Teubner, Stuttgart, 1994

[Haralick92] R.M. Haralick, L.G. Shapiro *Computer and Robot Vision*, Addison-Wesley, Massachusetts, 1992

[Eckstein91] W. Eckstein, W. Glock. *Development and Implementation of Methods for Segmentation of bond-wedges*, In R. Klette, editor, Proc. 4th Computer Analysis of Images and Patterns, volume 5 of Research in Informatics, pages 153–161, Dresden, 1991.

[Eckstein93] W. Eckstein, G. Lohmann, U. Meyer-Gruhl, R. Riemer, L. Altamirano Robles, J. Wunderwald *Benutzerfreundliche Bildanalyse mit HORUS: Architektur und Konzepte*, Proceedings DAGM 1993, Springer Verlag, Berlin, 1993

[Jaehne89] B. Jähne *Digitale Bildverarbeitung*, Springer-Verlag, Berlin, 1989

[Laws80] K.I. Laws *Texture image segmentation*, Ph.D. dissertation, Dept. of Engineering, University of Southern California, 1980

[Eck86] W. Eckstein, S.J. Pöppl. *PSIWAG - A Language for Logic Programming in Image Analysis*, Proc. 8th ICPR, Paris, 1986

[Eck88] W. Eckstein. *Das ganzheitliche Bildverarbeitungssystem HORUS*, Proc. 10. DAGM Symposium, Zürich, 1988

[Eck93] W. Eckstein. *Die Bildanalysesprache TRIAS*, Dissertation. Infix-Verlag, St. Augustin, 1993.

[Hae86] S. Haenel, W. Eckstein. *Ein Arbeitplatz zur halbautomatischen Luftbildauswertung*, Proc. 8. DAGM Symposium, Paderborn, 1986

[Klo93] K. Klotz. *Eine mehrschichtige Architektur zur Fehlerdiagnose und Fehlerbehebung bei der Entwicklung von logischen Programmen*, Dissertation, Infix-Verlag, St. Augustin, 1993.

[Kri92] H. Kristen, O. Munkelt. *Markov-Feld-basierte Bildinterpretation mit automatisch generierter Datenbasis*, Proc. 14. DAGM Symposium, Dresden, 1992.

[Lan91] S. Lanser, W. Eckstein. *Eine Modifikation des Deriche-Verfahrens zur Kantendetektion*, Proc. 13. DAGM Symposium, München, 1991

[Mes92] T. Messer. *Wissensbasierte Synthese von Bildanalyseprogrammen*, Dissertation, Infix-Verlag, St. Augustin, 1992.

[Rad92] B. Radig, W. Eckstein, K. Klotz, T. Messer, J. Pauli. *Automatization in the Design of Image Understanding Systems*, Proc. 5th International Conference, IEA/AIE-92, Paderborn, Springer-Verlag 1992, LNAI 604, S. 35-45

List of Figures

List of Tables