Appendix A

LTI-LIB — a C++ Open Source Computer Vision Library

Peter Dörfler and José Pablo Alvarado Moya

The LTI-LIB is an open source software library that contains a large collection of algorithms from the field of computer vision. Its roots lie in the need for more cooperation between different research groups at the Chair of Technical Computer Science at RWTH Aachen University, Germany. The name of the library stems from the German name of the Chair: **L**ehrstuhl für **T**echnische **I**nformatik.

This chapter gives a short overview of the LTI-LIB that provides the reader with sufficient background knowledge for understanding the examples throughout this book that use this library. It can also serve as a starting point for further exploration and use of the library. The reader is expected to have some experience in programming with C++.

The main idea of the LTI-LIB is to facilitate research in different areas of computer vision. This leads to the following design goals:

- *Cross platform*:
  It should at least be possible to use the library on two main platforms: Linux and Microsoft Windows©.
- *Modularity*:
  To enhance code reusability and to facilitate exchanging small parts in a chain of operations the library should be as modular as possible.
- *Standardized interface*:
  All classes should have the same or very similar interfaces to increase interoperability and flatten the user's learning curve.
- *Fast code*:
  Although not the main concern, the implementations should be efficient enough to allow the use of the library in prototypes working in real time.

Based on these goals the following design decisions were made:

- Use an object oriented programming language to enforce the standardized interface and facilitate modularity.
- For the same reasons and also for better code reuse the data structures should be separated from the functionality unless the data structure itself allows only one way of manipulation.
- To gain fast implementations C++ was chosen as the programming language. With the GNU Compiler Collection (gcc) and Microsoft Visual C++ compilers are available for both targeted platforms.

The LTI-LIB is licensed under the GNU Lesser General Public License (LGPL)[1] to encourage collaboration with other researchers and allow its use in commercial products. This has already lead to many enhancements of the existing code base and also to some contributions from outside the Chair of Technical Computer Science. From the number of downloads the userbase can be estimated to a few thousand researchers.

This library differs from other open source software projects in its design: it follows an object oriented paradigm that allows to hide the configuration of the algorithms, making it easier for unexperienced users to start working with them. At the same time it allows more experienced researchers to take complete control of the software modules as well. The LTI-LIB also exploits the advantages of generic programming concepts for the implementation of container and functional classes, seeking for a compromise between "pure" generic approaches and a limited size for the final code.

This chapter only gives a short introduction to the fundamental concepts of the LTI-LIB. We focus on the functionality needed for working with images and image processing methods. For more detailed information please refer to one of the following resources:

- project webpage: http://ltilib.sourceforge.net
- online manual: http://ltilib.sourceforge.net/doc/html/index.shtml
- wiki: http://ltilib.pdoerfler.com/wiki
- mailing list: ltilib-users@lists.sourceforge.net

The LTI-LIB is the result of the collaboration of many people. As the current administrators of the project we would like to thank Suat Akyol, Ulrich Canzler, Claudia Gönner, Lars Libuda, Jochen Wickel, and Jörg Zieren for the parts they play(ed) in the design and maintenance of the library additionally to their coding. We would also like to thank the numerous people who contributed their time and the resulting code to make the LTI-LIB such a large collection of algorithms:

Daniel Beier, Axel Berner, Florian Bley, Thorsten Dick, Thomas Erger, Helmuth Euler, Holger Fillbrandt, Dorothee Finck, Birgit Gehrke, Peter Gerber, Ingo Grothues, Xin Gu, Michael Haehnel, Arnd Hannemann, Christian Harte, Bastian Ibach, Torsten Kaemper, Thomas Krueger, Frederik Lange, Henning Luepschen, Peter Mathes, Alexandros Matsikis, Ralf Miunske, Bernd Mussmann, Jens Paustenbach, Norman Pfeil, Vlad Popovici, Gustavo Quiros, Markus Radermacher, Jens Rietzschel, Daniel Ruijters, Thomas Rusert, Volker Schmirgel, Stefan Syberichs, Guy Wafo Moudhe, Ruediger Weiler, Benjamin Winkler, Xinghan Yu, Marius Wolf

A.1 Installation and Requirements

The LTI-LIB does not in general require additional software to be usable. However, it becomes more useful if some complementary libraries are utilized. These are in-

[1] http://www.gnu.org/licenses/lgpl.html

cluded in most distributions of Linux and are easily installed. For Windows systems the most important libraries are included in the installer found on the CD.

The script language *perl* should be available on your computer for maintenance and to build some commonly used scripts. It is included on the CD for Windows, and is usually installed on Linux systems. The Gimp Tool Kit (*GTK*)[2] should be installed as it is needed for almost all visualization tools. To read and write images using the Portable Network Graphics (PNG) and the Joint Picture Experts Group (JPEG) image formats you either need the relevant files from the ltilib-extras package, which are adaptations of the Colosseum Builders C++ Image Library, or you have to provide access to the freely available libraries *libjpeg* and *libpng*. Additionally, the data compression library *zlib* is useful for some I/O functions[3].

The LTI-LIB provides interfaces to some functions of the well-known LAPACK (Linear Algebra PACKage)[4] for optimized algorithms. On Linux systems you need to install *lapack*, *blas* , and the fortran to C translation tool *f2c* (including the corresponding development packages, which in distributions like SuSE or Debian are always denoted with and additional "-dev" or "-devel" postfix). The installation of LAPACK on Windows systems is a bit more complicated. Refer to http://ltilib.pdoerfler.com/wiki/Lapack for instructions.

The installation of the LTI-LIB itself is quite straightforward on both systems. On Windows execute the installer program (ltilib/setup-ltilib-1.9.15-activeperl.exe). This starts a guided graphical setup program where you can choose among some of the packages mentioned above. If unsure we recommend installing all available files.

Installation on Linux follows the standard procedure for configuration and installation of source code packages established by GNU as a de facto standard:

```
cd $HOME
cp /media/cdrom/ltilib/*.gz .
tar -xzvf 051124_ltilib_1.9.15.tar.gz
cd ltilib
tar -xzvf ../051124_ltilib-extras_1.9.15.tar.gz
cd linux
make -f Makefile.cvs
./configure
make
```

Here it has been assumed that you want the sources in a subdirectory called `ltilib` within your home directory, and that the CD has been mounted into `/media/cdrom`. The next line unpacks the tarballs found on the CD in the directory `ltilib`. In the file name, the first number of six digits specifies the release date of the library, where the first two digits relate to the year, the next two indicate the month and the last two the day. You can check for newer versions in the library homepage. The sources in the `ltilib-extras` package should be unpacked within the

[2] http://www.gtk.org

[3] http://www.ijg.org, http://www.libpng.org, and http://www.zlib.net, respectively

[4] http://netlib.org/lapack/

directory `ltilib` created by the first tarball. The next line is optional, and creates all configuration scripts. For it to work the auto-configure packages have to be installed. You can however simply use the provided `configure` script as indicated above. Additional information can be obtained from the file `ltilib/linux/README`. You can of course customize your library via additional options of the `configure` script (call `./configure --help` for a list of all available options). To install the libraries you will need write privileges on the `--prefix` directory (which defaults to `/usr/local`). Switch to a user with the required privileges (for instance `su root`) and execute `make install`. If you have *doxygen* and *graphviz* installed on your machine, and you want to have local access to the API HTML documentation, you can generate it with `make doxydoc`. The documentation to the release 1.9.15 which comes with this book is already present on the CD.

A.2 Overview

Following the design goals, data structures and algorithms are mostly separated in the LTI-LIB. The former can be divided into small types such as points and pixels and more complex types ranging from (algebraic) vectors and matrices to trees and graphs. Data structures are discussed in section A.3. Algorithms and other functionality are encapsuled in separate classes that can have parameters and a state and manipulate the data given to them in a standardized way. These are the subject of section A.4. Before dealing with those themes, we cover some general design concepts used in the library.

A.2.1 Duplication and Replication

Most classes used for data structures and functional objects support several mechanisms for duplication and replication. The copy constructor is often used while initializing a new object with the contents of another one. The `copy()` method and the `operator=` are equivalent and permit to acquire all data contained in another instance, whereby, as a matter of style, the former is the way preferred within the library, as it visually permits a faster realization that the object belongs to an LTI-LIB class instead of being an elemental data type (like integers, floating point values, etc.). As LTI-LIB instances might represent or contain images or other relatively large data structures, it is important to always keep in mind that a replication task may take its time, and it is therefore useful to explicitly indicate when this expensive copies are done.

In class hierarchies it is often required to replicate instances of classes to which you just have a pointer to a parent common class. This is the case, for instance, when implementing the *factory* design pattern. For these situations the LTI-LIB provides a virtual `clone()` method.

A.2.2 Serialization

The data of almost all classes of the LTI-LIB can be written to and read from a stream. For this purpose an `ioHandler` is used that formats and parses the stream in a special way. At this time, two such ioHandlers exist. The first one, `lispStreamHandler`, writes the data of the class in a Lisp like fashion, i. e. each attribute is stored as a list enclosed in parenthesis. This format is human readable and can be edited manually. However, it is not useful for large data sets due to its extensive size and expensive parsing. For these applications the second ioHandler, `binaryStreamHandler`, is better suited.

All elementary types of C++ and some more complex types like the STL[5] `std::string`, `std::vector`, `std::map` and `std::list` can be read or written from the LTI-LIB streams using a set of global functions called `read()` and `write()`. Global functions also allow to read and write almost all LTI-Lib types from and to the stream handlers, although as a matter of style, if a class defines its own `read()` and `write()` methods they should be preferred over the global ones to indicate that the given object belongs to the LTI-LIB.

The example in Fig. A.1 shows the principles of copying and serialization on a LTI-LIB container called `vector<double>`, and a STL list container of strings.

A.2.3 Encapsulation

All classes of the LTI-LIB reside in the namespace `lti` to avoid ambiguity when other libraries might use the same class names. This allows for instance to use the class name `vector`, even if the Standard Template Library already uses this type. For the sake of readability the examples in this chapter omit the explicit scoping of the namespace (i. e., the prefix `lti::` before all class names omitted).

Within the library, types regarding just one specific class are always defined within that class. If you want to use those types you have to indicate the class scope too (for example `lti::filter::parameters::borderType`). Even if this is cumbersome to type, it is definitely easier to maintain, as the place of declaration and the context are directly visible.

A.2.4 Examples

If you want to try out the examples in this chapter, the easiest way is to copy them directly into the definition of the `operator()` of the class `tester` found in the file `ltilib/tester/ltiTester.cpp`. The necessary header files to be included are listed at the begining of each example. Standard header files (e.g. `cmath`, `cstdlib`, etc.) are omitted for brevity.

[5] The Standard Template Library (STL) is included with most compilers. It contains common container classes and algorithms.

```
#include "ltiVector.h"
#include "ltiLispStreamHandler.h"
#include <list>

vector<double> v1(10,2.);    // create a double vector
vector<double> v2(v1);       // v2 is equal to v1
vector<double> v3;

v3=v2;                       // v3 is equal to v2
v3.copy(v2);                 // same as above

std::list<std::string> myList; // a standard list
myList.push_back(``Hello'');   // with two elements
myList.push_back(``World!'');

// create a lispStreamHandler and write v1 to a file
std::ofstream os("test.dat"); // the STL output file stream
lispStreamHandler lsh;        // the LTI-Lib stream handler
lsh.use(os);                  // tell the LTI-Lib handler
                              // which stream to use

v1.write(lsh);                // write the lti::vector
write(lsh,myList);            // write the std::list through
                              // a global function
os.close();                   // close the output file

// read the file into v4
vector<double> v4;
std::list<std::string> l2;

std::ifstream is("test.dat"); // STL input file stream
lsh.use(is);                  // tell the handler to read
                              // from the ``is'' stream
v4.read(lsh);                 // load the vector
read(lsh,l2);                 // load the list
is.close();                   // close the input file
```

Fig. A.1. Serialization and copying of LTI-LIB classes.

A.3 Data Structures

The LTI-LIB uses a reduced set of fundamental types. Due to the lack of fixed size types in C/C++, the LTI-LIB defines a set of aliases at configuration time which are mapped to standard types. If you want to ensure 32 bit types you should use therefore `int32` for signed integers or `uint32` for the unsigned version. For 8 bit numbers you can use `byte` and `ubyte` for the signed and unsigned versions, respectively.

When the size of the numeric representations is not of importance, the default C/C++ types are used. In the LTI-LIB a policy of avoidance of unsigned integer types is followed, unless the whole data representation range is required (for example, `ubyte` when values from 0 to 255 are to be used). This has been extended to the values used to represent the sizes of container classes, which are usually of type `int`, as opposite to the `unsigned int` of other class libraries like the STL.

Most math and classification modules employ the double floating point precision (`double`) types to allow a better conditioning of the mathematical computations involved. Single precision numbers (`float`) are used where the memory requirements are critical, for instance in grey valued images that require a more flexible numerical representation than an 8-bit integer.

More involved data structures in the LTI-LIB (for instance, matrix and vector containers) use generic programming concepts and are therefore implemented as C++ template types. However, these are usually explicitly instantiated to reduce the size of the final library code. Thus, you can only choose between the explicitly instantiated types. Note that many modules are restricted to those types that are useful for the algorithm they implement, i. e. linear algebra functors only work on float and double data structures. This is a design decision in the LTI-LIB architecture, which might seem in some cases rigid, but simplifies the maintenance and optimization of code for the most frequently used cases. At this point we want to stress the difference of the LTI-LIB with other "pure" generic programming oriented libraries. For the "generic" types, like `genericVector` or `genericMatrix` you can also create explicit instantiations for your own types, for which you will require to include the header files with the template implementations, usually denoted with a postfix `template` in the file name (for example, `ltiGenericVector_template.h`).

The next sections introduce the most important object types for image processing. Many other data structures exist in the library. Please refer to the online documentation for help.

A.3.1 Points and Pixels

The template classes `tpoint<T>` and `tpoint3D<T>` define two and three dimensional points with attributes `x`, `y`, and if applicable `z` all of the template type `T`. For `T=int` the type aliases `point` and `point3D` are available. Especially `point` is often used as coordinates of a pixel in an image.

Pixels are defined similarly: `trgbPixel<T>` has three attributes `red`, `green`, and `blue`. They should be accessed via the member functions like `getRed()`. However, the pixel class used for color images is `rgbPixel` which is a 32 bit data structure. It has the three color values and an extra alpha value all of which are `ubyte`. There are several reasons for a four byte long pixel representation of a typically 24 bit long value. First of all, in 32 bit or 64 bit processor architectures, this is necessary to ensure the memory alignment of the pixels with the processor's word length, which helps to improve the performance of the memory access. Second, it helps to improve the efficiency when reading images from many 32-bit based frame

grabbers. Third, some image formats store the 24-bit pixel information together with an alpha channel, resulting in 32-bit long pixels.

All of these basic types support the fundamental arithmetic functions. The example in Fig. A.2 shows the idea.

```
#include "ltiPoint.h"
#include "ltiRGBPixel.h"

// create an rgbPixel, set its values and divide them by 2
rgbPixel pix;                   // a 32-bit long pixel
pix.set(127,0,255);             // set the three RGB values
pix.divide(2);                  // divide all components by 2
std::cout << pix << std::endl; // show the new components

// create two points and add the first one to the second
tpoint<float> p1(2.f,3.f);      // create a 2D point
tpoint<float> p2(3.f,2.f);      // and another one
p2.add(p1);                     // add both points and leave
                                // the result in p2
std::cout << p2 << std::endl;  // show the result
```

Fig. A.2. This small code example shows how to create instances of some basic types and how to do arithmetic operations on them.

A.3.2 Vectors and Matrices

Probably the most important types in the LTI-LIB are `vector`and `matrix`. Unlike the well known STL classes these are vectors and matrices in the mathematical sense and provide the appropriate arithmetic functions. Since LTI-LIB vectors and matrices have been designed to support efficient data structures for image processing applications, they also differ from the STL equivalents in the memory management approach.

STL vectors, for instance, increase their memory dynamically by allocating a larger block and copying the old contents by calling the elements' copy constructors. This can be very useful when the final size of a vector is not known but is not very efficient.

On the other hand, since most image processing algorithms require very efficient ways to access the data in their memory, the LTI-LIB `vector` and `matrix` classes optimize the mechanisms to share, transfer, copy and access the memory among different instances, but are less efficient in reserving and deallocating memory. For example, matrices are organized in a row-wise manner and each row can be accessed as a vector. In order to avoid copying the data of the matrix to a vector when the information is required as such, a pool of row vectors is stored within the matrix which

share the data with the matrix. The matrix memory is also a single adjacent and compact memory block, that can therefore be easily traversed. Iteration on the elements of the matrices or vectors has been optimized to allow boundary check in a code debugging stage, but also to be as fast as C pointers when the code is finally released. Additionally, complete memory transfers and exchange of data among matrix and vector instances is allowed, which usually saves valuable time, since unnecessary copying is avoided.

We will not go into a deeper detail of the memory management in the LTI-LIB containers here, but rather point out some of the main container features. Figure A.3 shows a small example for vector matrix multiplication.

```
#include "ltiVector.h"
#include "ltiMatrix.h"

double tmp[]={1., 2., 1., 3.};     // data for a vector
dmatrix mat(2,2,tmp);              // create a 2x2 matrix
dvector vec(2,2.);                 // create a 2D vector
vec.at(1) = 1.;                    // second element to 1
dvector res;
mat.multiply(vec, res);            // multiply mat with vec
mat.at(0,1)*=2.;                   // at 1st row, 2nd column
mat[0][1]/=2.;                     // at 1st row, 2nd column

std::cout << res << std::endl;     // show the results
std::cout << vec.dot(res) << std::endl;  // dot product
```

Fig. A.3. Vector matrix multiplication. The result of the product of the 2×2 matrix `mat` and `vec` is stored in `res`. All arithmetic functions also return the result as can be seen with the dot product between the two vectors.

The example uses the abbreviated form `dvector` instead of `vector<double>` to enhance readability. Two different ways of initializing a `vector`/`matrix` are shown: using a pointer to an array of elements of the proper type or through a single initialization value. Single elements can be accessed via `at()` or the `operator[]`, whereby the `at()` version should be preferred, first, because it is faster (especially with matrices), and second, to indicate that you access an LTI-LIB container. All possibilities of vector and matrix multiplications as well as sum, difference and division by a scalar are provided. Further, mathematical functions such as transpose and sum of elements as well as informational functions such as minimum are available.

For convenience both classes have an `apply()` method which takes a C-style function as argument. The function is then applied to each element of the vector/matrix. The example in Fig. A.4 shows how to use this function along with some more element manipulation features.

```
#include "ltiVector.h"
#include "ltiMatrix.h"

fmatrix mat(2,2,4.f);     // 2x2 matrix filled with 4
mat.getRow(0).fill(9.f); // fill first row with 9
mat[0].fill(9.);          // also fills the first row with 9
mat.apply(sqrt);          // sqrt() to all elements of mat
std::cout << mat << std::endl;
```

Fig. A.4. Using `matrix::apply()`: The matrix `mat` is initialized with the value 4 in all elements. Then all elements in the first row are set to 9. Using the `apply()` function the square root of all elements is calculated and left in the matrix.

The function `getRow()` returns a reference to a `vector` which is a row of the matrix, and so does the `operator[]`. The values of this reference can be manipulated. In the example all elements of the row are set to 9. The example writes a 2×2 matrix on the console whose first row elements are 3 and second row elements are 2.

A.3.3 Image Types

The LTI-LIB provides four container classes for images: `image` represents color images, `channel` and `channel8` are utilized as grey value images, and `channel32` is mostly used to hold so-called label masks. Using the template functionality of `matrix` we define the image types as matrices of `rgbPixel`, `float`, `ubyte`, and `int32`, respectively.

The two grey valued image types have quite different objectives: `channel8` uses 8-bit per pixel which is a common value for monochrome cameras. Algorithms that work directly on these are usually very fast due, on the one hand, to the single-byte fixed point arithmetic and on the other hand to the possibility of using look-up tables to speed up computations as only 256 values are available. However, many image processing methods return results with a wider range of values or need higher precision. For these, the `channel` is more appropriate as its pixels employ single precision floating point numbers.

For a regular monochrome `channel` the value range for a pixel lies between 0.0 and 1.0. Quite obviously a `channel` can have values outside of the normalized interval. This property is often necessary, for example, when the magnitude of an image gradient, or the output of a high-pass filter is computed and stored in a `channel`. The `image` class represents a 24(32)-bit RGB(A) color image, i. e. 24 bits are used for the three colors and an additional byte is used for the alpha channel. Color images are often split into three channels for processing (cf. A.5.3).

The pixel representation as a 32-bit fixed point arithmetic value, as found in `channel32` objects, are mostly employed to store pixel classification results, where each integer number will denote a specific class, and the total number of pixel classes may become greater than the capabilities of a `channel8`. This is the case, for instance, in image segmentation results or in multiple object recognition systems.

The advantage of using `matrix` for image types is twofold: the individual pixels can easily be accessed and the calculation power of `matrix` is available for images as well. The only additional functions in the image types are conversion functions from other image types via `castFrom()`, which maintain certain semantical meaning in the conversion: for example, the range of 0 to 255 of a `channel8`is linearly mapped into the interval 0.0 to 1.0 for a `channel`. The example in Fig. A.5 shows how easy it is to implement a transform of grey values to the correct interval between 0 and 1 using matrix functions.

```
#include "ltiImage.h"

// let ch contain values outside of [0; 1]
channel ch;
// find min and max value
float cmin,cmax;
ch.getExtremes(cmin,cmax);
// calc factor and offset
const float fac=1.f/(cmax-cmin);
const float off=cmin*fac;
ch.multiply(fac);
ch.subtract(off);
```

Fig. A.5. Using `matrix` functions to normalize grey values. Assuming that `ch` contains values outside the range [0,1] or in a much smaller interval we stretch/shrink and shift the grey values so that they cover the complete [0,1] interval.

The `channel ch` could be the result of some previous processing step. First the minimum and maximum values are sought. From these the required stretching factor and corresponding offset are calculated and each element of the image matrix is transformed accordingly. Note that operations like the above can be achieved more efficiently by defining a function and then using the `apply()` member function supplied by `matrix`. For this case in particular `channel` also offers the member function `mapLinear()`.

A.4 Functional Objects

Functional objects in the LTI-LIB consist of three parts: the functionality itself (i. e., the algorithms), parameters that can tune the behavior of the algorithm to the requirements of a user, and an internal state that holds trained data or partial results. Two large groups of functional objects exist: functors and classifiers. While the former mostly behave like a function the latter need to be trained first to be useful. Since the basic usage of functors and classifiers is very similar we'll discuss functors in detail first. Then only the differences of classifiers with respect to functors are elaborated.

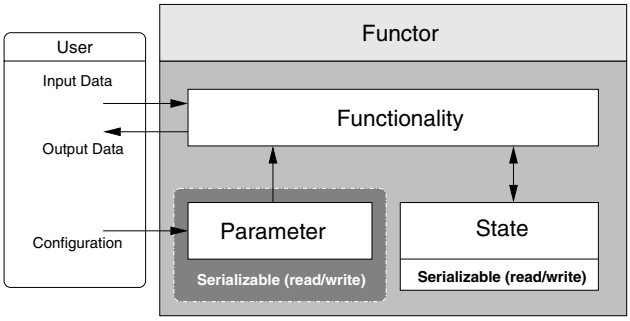


Fig. A.6. A functor has three parts: functionality, parameters, and state. The diagram shows their interaction

A.4.1 Functor

A functor consists of three main parts: functionality, parameters, and state. The functionality of the functor is accessed via the `apply()` methods. In many cases there are on-copy and on-place versions of these methods, so the user can choose whether the output of the algorithm should overwrite the data in the container objects used as input, or if the results should to be stored in container objects passed specially for this purpose as arguments of the `apply()` methods. The functionality can be modified via the parameters of the functor. Finally, some functors require a state, for example, for keeping track of previous input data. Figure A.6 shows the three parts of a functor and their interaction.

The parameters are implemented as an inner class of each functor. This allows to consistently use the type name `parameters` to designate this configuration object in each functor, where the class name is used to clearly indicate their scope. To ensure consistency along lines of inheritance the parameters class of a derived functor is derived from the parameters class of its base class. The current parameters of a functor can be changed via the function `setParameters()`. A read-only reference to the current parameters instance is returned by `getParameters()`. The parameters are never changed by the functor itself. Only the user or another functor may set parameters. This is opposite to the state of a functor which is manipulated exclusively by the functor itself.

The parameters as well as the state of the functor are serializable through the members `read()` and `write()`. Note that when saving the functor the parameters are saved as well.

Setting parameters is one of the most important tasks when using the LTI-LIB. Although most functors have useful default parameters, these need to be adapted for special tasks quite frequently. In Fig. A.7 we take a look at configuring our implementation of the Harris corner detector [1]. As can be seen in the documentation it uses two other functors internally, `localMaxima` and `gradientFunctor`. To keep the code maintainable, the parameters of `harrisCorners` just contain instances of the parameter classes of those two functors. In the example the pa-

rameter kernelType of the internal gradientFunctor is changed as well. Note that most functors offer a constructor that takes the functor's parameters as an argument, which is more convenient than setting the parameters later. This example also introduces another data structure: pointList. It can be used like a std::list<point> but has some additional features.

```
#include "ltiImage.h"
#include "ltiHarrisCorners.h"

// assume this channel contains an image
channel src;

// change some parameters and create functor
harrisCorners::parameters param;
param.gradientFunctorParameters.kernelType
  = gradientFunctor::parameters::OGD;
param.gradientFunctorParameters.gradientKernelSize = 5;
param.kernelSize = 7;
harrisCorners harris(param);

// list of lti::point
pointList corners;
channel cornerness;
float maxCornerness;
harris.apply(src,cornerness,maxCornerness,corners);

//print out how many corners were found
std::cout << corners.size() << std::endl;
```

Fig. A.7. Setting the parameters of a functor. In this example the parameters contain instances of another functor's parameters — gradientFunctor::parameters. These also define an enum which contains the value OGD. The harrisCorners functor is applied on a channel and returns the positions of the corners.

A.4.2 Classifier

Classifiers are very similar to functors. The main difference is that instead of apply() they have train() and classify() member functions. All classifiers have a state which stores the results of the training and which is later used for classification. The input data is strictly limited to dvector (or sequences thereof).

The LTI-LIB discerns three general types of classifiers: supervised and unsupervised instance classifiers, and supervised sequence classifiers. Hidden Markov Models (cf. chapter 2.1.3.6) are currently the only representatives of the last category. Supervised instance classifiers are trained with pairs of dvector and an integer, i. e. feature vector and label. Unsupervised instance classifiers only receive feature

vectors as input and try to find natural groups within the data. This group contains clustering algorithms as well as connectionist algorithms such as Self Organizing Feature Maps. When used for classification the unsupervised version tries to give a measure of how likely the given feature vector would have belonged to each cluster had it been in the original data set.

The classifiers are not much used within the context of this book. The interested reader is refered to the CD and online documentation for more information about classifiers.

A.5 Handling Images

Most algorithms used in computer vision draw their information from images or sequences of images. We have seen that they are mostly manipulated with functors in the LTI-LIB. This section explains some of the fundamental tasks when working with images. The first paragraph explains how to convolve an image with an arbitrary filter kernel. The subject of the second paragraph is input/output for different image formats. The next paragraph shows how to split a color image into different color space representations. Finally, drawing simple geometric primitives and viewing of images is explained.

A.5.1 Convolution

One of the most common opertations in image processing is the convolution of an image with a linear filter kernel such as a Gaussian kernel or its derivatives. The LTI-LIB distinguishes between kernels that are separable and those that are not. A kernel is separable if it can be written as the outer product of two one dimensional kernels:

$$\mathbf{k}(x,y) = \mathbf{k_x}(x)\mathbf{k_y}^T(y)$$

with $\mathbf{k}$ a two dimensional kernel, $\mathbf{k_x}$ and $\mathbf{k_y}$ one dimensional column kernels. The advantage of separable filters is that they can be applied subsequently to the image rows and columns. Thus, far less multiplications are needed for the convolution of the image and kernel. Even if the equation above does not hold, a two dimensional kernel can be approximated by a separable kernel or a set of separable kernels.

The classes `kernel2D` and `sepKernel` represent the two types of kernels that occur when convolving an image with a kernel. The latter consists of several `kernel1D` which can also be used for vector convolution. The class `convolution` applies a kernel which is set in the parameters to an image. The example shown in Fig. A.8 first shows how to manually create a kernel, here a sharpening function. Then is uses the `gaussKernel` to smooth the image.

Note that `convolution` has a convenience function `setKernel()` which sets the kernel in the parameters of the functor. The convolution detects which kind of kernel is given — 2D or separable — and calls the appropriate algorithm. The size of the Gauss kernel used in the example is 5×5 and has a variance of 1.7.

```
#include "ltiImage.h"
#include "ltiLinearKernels.h"
#include "ltiGaussKernels.h"
#include "ltiConvolution.h"

// assume this channel contains an image
channel src;

// a convolution functor
convolution conv;

// create a 3x3 sharpening filter which is not separable
kernel2D<float> sharpKernel(-1,-1,1,1,-0.1f);
sharpKernel.at(0,0) = 1.8f;
// use shortcut function to set kernel
conv.setKernel(sharpKernel);
// sharpen src and leave in sharpImg
channel sharpImg;
conv.apply(src,sharpImg);

// create a Gaussian kernel and smooth the sharp image
gaussKernel2D<float> gaussKern(5,1.7);
conv.setKernel(gaussKern);
conv.apply(sharpImg);
```

Fig. A.8. The example first creates a `kernel2D` to sharpen the channel and then uses a Gauss kernel, which is a `sepKernel` to smooth it again.

Another important parameter of the convolution is the boundary type, which specifies how the data on the border of the image has to be considered. There are five possibilities: a boundary of `Zero` assumes that all pixels outside the image are zero valued. In cases where the derivative of the images has to be approximated, the borders usually cause undesired effects. This can be reduced indicating a `Constant` boundary, which means a constant continuation of the pixel values at the boundary. The image can also be considered as the result of a windowing process of a periodic and infinite two-dimensional signal. If the window has the size of one period, then the boundary `Periodic` has to be used. If the window is half of the period of an even symmetrical signal then `Mirror` is the right choice. If you want to leave the border untouched, because maybe you want to take care of its computation in a very special way, then you have to set the `boundaryType` attribute with a `NoBoundary` value.

A.5.2 Image IO

The LTI-LIB can read and write three common image formats: BMP, JPEG, and PNG. While the former always uses an implementation which is part of the library

the latter two use the standard `libjpg` and `libpng` if they are available. These libraries are mostly present on Unix/Linux systems. For Windows systems source code is available to build the libraries. If the libraries are not available an alternative implementation based on the Colosseum Builders C++ Image Library is available in the extras package of the LTI-LIB.

Unless the LTI-LIB is used to write a specific application it is most practical to be able to load all possible image types. The file `ltiALLFunctor.h` contains two classes `loadImage` and `saveImage` that load/save an image depending on its file extension. The filename can be set in the parameters or given directly in the convenience functions `load()` and `save()`, respectively. The small example in Fig. A.9 shows how a bitmap is loaded and then saved as a portable network graphic.

```
#include "ltiImage.h"
#include "ltiALLFunctor.h"

loadImage loader;
saveImage saver;
image img;

loader.load("/home/homer/bart.bmp",img);
saver.save("/home/homer/bart.png",img);
```

Fig. A.9. In this example a bitmap is loaded from a file and saved as a PNG.

Very often it is not necessary to save images on disk but to process a large set of images. For this purpose the LTI-LIB has the `loadImageList` functor. It can be used to sequentially get all images in a directory, listed in a file or supplied in a `std::vector` of strings. In any case the filenames are sorted alphabetically to ensure consistency across different systems. While the method `hasNext()` returns true there are more images available. The next image is loaded by calling `apply()`.

It is also possible to load grey valued and indexed images with the image I/O functors. If the loaded file actually contains grey value information this can be loaded into a channel. For a 24 bit file the intensity of the image is left in the channel. Indexed images can be loaded into a channel8 and a palette which stores up to 256 colors. If the file contains a 24 bit image the colors are quantized to up to 256 colors.

A.5.3 Color Spaces

Many image processing algorithms that work on color images first split them into the different components in a particular color space (the so called color channels) and then work on these separately. The most common color space is RGB (Red, Green, Blue) which is also the native color space for LTI-LIB images, as it is frequently used when dealing with industrial cameras and image file formats. Particular types of images for additional color spaces are not provided, nor special pixel types, as

the number of available color spaces would have made the maintenance of all LTI-LIB functors very difficult, due mainly to the typical singularities found in different spaces (for example, the case of intensity zero in hue-saturation spaces). Therefore, in this library just one color format is used, while all other cases have to be managed as a collection of separate `channel` objects.

To extract all three color components from an image in a particular color space, the functors `splitImageToXXX` is used, with `XXX` the desired color space abbreviation. The functors `mergeXXXToImage` form an RGB `image` from three separate components in the `XXX` color space. The code in Fig. A.10 shows how color channels can be swapped to create a strangely colored image.

```
#include "ltiImage.h"
#include "ltiSplitImageToRGB.h"
#include "ltiMergeRGBToImage.h"

// this is the image we want to change
image src;
// RGB channel8
channel8 r,g,b;

splitImageToRGB split;
mergeRGBToImage merge;

split.apply(src,r,g,b);
merge.apply(g,b,r,src);
```

Fig. A.10. The image `src` is first split into the three RGB channels. Then an image is reconstructed with swapped colors.

In this example we use `channel8` rather than `channel` because no calculations are performed and we can avoid casting from ubyte to float and back again. An often needed functionality is to get an intensity channel from an image or create an image from a channel (the image will, of course, be grey valued). These are implemented via the `castFrom()` functions of the image types. Figure A.11 shows an example.

These functions are especially helpful for visualization. After a channel is converted to an image we can draw on the channel in color. Note that the above operation can again be implemented via the `apply()` member function of `matrix` for faster processing.

Besides the RGB color space the following conversions are implemented in the LTI-LIB:

- `CIELuv`: CIE $L^*u^*v^*$ is an attempt to linearize color perception. It is related to `XYZ`.
- `HLS`: Hue, Luminance, and Saturation

```
#include "ltiImage.h"

// a color image
image img;
channel ch;

// compute the intensity defined as (R+G+B)/3
ch.castFrom(img);

// now contains same value in all colors
img.castFrom(ch);
```

Fig. A.11. The color image `img` is first converted into an intensity `channel`. Then the same information is copied to all three channels of the image, resulting in a color image that contains only grey tones.

- `HSI`: Hue, Saturation, and Intensity
- `HSV`: Hue, Saturation, and Value
- `OCP`: An opponent color system. The first two are opponent colors, the third is the intensity.
- `rgI`: red and green chromaticity and Intensity.
- `xyY`: normed chromaticity channels
- `XYZ`: a set of three linear-light components that conform to the CIE color-matching functions.
- `YIQ`: Luminance Inphase Quadrature channels used in the NTSC television standard.
- `YUV`: Luminance/chrominance representation according to ITU-RS601. Well known from JPEG.

All of these color representations can be obtained from and reverted to an `image`. The example in Fig. A.12 shows how to generate a false color image from a channel with only a few lines of code.

A.5.4 Drawing and Viewing

In programs related to image processing it is often desired to visualize the results. The LTI-LIB offers classes for drawing in images and for viewing them. While 2D and 3D drawing and viewing is available this introduction only covers the former.

The class `draw<T>` draws on an image of type T, e. g. `draw<rgbPixel>` which is equivalent to `draw<image::value_type>` draws on an `image`. The image that serves as canvas is set with the function `use()`. To change the color/value for drawing call `setColor()`. The graphical representation of many simple types of the LTI-LIB can be drawn via the `set()` method. This is also the fastest way to draw a single pixel in the current color. The following list of functions gives an overview of frequently used functions:

```
#include "ltiImage.h"
#include "ltiMergeHSIToImage.h"

// the source channel
channel ch;
// the false color image
image img;

// functor for merging an RGB image from
// hue, saturation, intensity information
mergeHSIToImage hsiMerger;

channel tmp(ch.size(), 0.8);
hsiMerger.apply(ch,tmp,tmp,img);
```

Fig. A.12. Using the given channel as hue information and a temporary channel of the same size for saturation and intensity we receive a false color image of the original channel.

- *Geometric shapes:* `line()`, `lineTo()`, `arrow()`, `rectangle()`, `arc()`, `circle()`, `ellipse()`
- *Vectors:* A `vector` can be drawn as a function of one discrete integer variable with `set(vector)`.
- *Markers:* With `marker()` a point can be marked by a small shape.
- *Text:* The functions `text()` and `number()` offer rudimentary text drawing functionality.

The markers available are similar to those known from Matlab/Octave. Additionally, they can be scaled via a parameter and selected to be filled. To set the marker type either an enum value or the familiar Matlab or Octave string can be used. Table A.1 gives the available types. When configuring the marker it can be combined with a color and a factor stating the brightness of the color. See the example in Fig. A.13 for a few sample configurations.

The viewer in the LTI-LIB is based on **GTK**[6]. While it also has parameters like a functor, most of these can also be set via the GUI. The only notable exception to this rule is the window title which is most easily set with the constructor. To view an image create a viewer and use `show()`. This function can also be used on other types such as vectors and histograms. The viewer has been designed to take control of the GUI in a separate thread, in a way that the users is allowed to manipulate the visualization parameters as long as the viewer instance exists. This enormously simplifies the task of displaying a window with an image, which otherwise would involve some cumbersome widget programming. The viewer will not block the execution of your program, unless you explicitly call one of the blocking methods like `waitButtonPressed()`, which waits for a left click on the window

[6]The Gimp Tool Kit is a multi-platform GUI toolkit, licensed under the LGPL. http://www.gtk.org

Table A.1. Marker types and color specifiers for `draw`.

Marker types

Marker shape	enum value	marker char
Pixel	Pixel	`.`
Circle	Circle	`o`
Xmark	Xmark	`x`
Plus	Plus	`+`
Star	Star	`*`
Square	Square	`s`
Diamond	Diamond	`d`
Triangle up	TriangleUp	`^`
Triangle down	TriangleDown	`v`
Triangle left	TriangleLeft	`<`
Triangle right	TriangleRight	`>`
Dot	Dot	`#`
LTI-Logo	LtiLogo	

Color specifiers

Color	char
white	`w`
black	`b`
red	`r`
green	`g`
blue	`b`
cyan	`c`
magenta	`m`
yellow	`y`

or `waitKey()` which waits for a keystroke. The function `waitKeyOrButton()` combines both. To stop showing the image the member function `hide()` is supplied, which is anyway automatically called when the viewer object is destroyed. The example code in Fig. A.14 shows how to display the `canvas` from the previous example. The last two lines of the example are needed since there is a technical problem at the very end of the program, where there is no way to coordinate that the GTK has to be stopped before all GUI handlers are destroyed.

The capabilities of `draw` in combination with the viewer are now demonstrated on the data obtained with a Harris corner detector (cf. Fig. A.7). Figure A.15 shows the cornerness after normalization as well as the corners on the original image and the configuration dialog of the viewer. The additional source code can be found in Fig. A.16.

The upper part of the configuration dialog offers sliders for zoom, contrast and brightness of the displayed image. The left part of the middle section shows some statistics information about the image. These are exploited by the different scale functions on the right. In the example Scale Min-Max was used to warp the values of the cornerness to the interval [0,1]. In the bottom section the possibility to save the displayed image is of interest. It uses the functor defined in `ltiALLFunctor.h`.

A.6 Where to Go Next

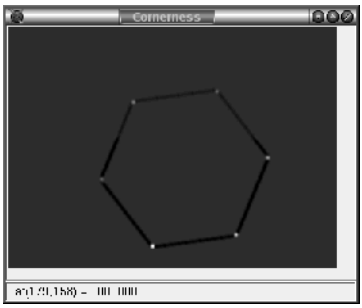
This chapter has introduced a small set of functors usually employed in image processing tasks. The LTI-LIB provides however many additional algorithms and general classes for this and other related areas, which simplify the implementation of complete computer vision systems. You will find functors for segmentation, morphology, edge and corner detection, saliency analysis, color analysis, linear and non-linear filtering, and much more. Images can not only be obtained from files but also

```
#include "ltiImage.h"
#include "ltiDraw.h"

// the canvas is yellow
image canvas(30,30,Yellow);
// draw on that canvas
draw<rgbPixel> painter;
painter.use(canvas);

painter.setColor(Blue);
point p1(25,5),p2(25,25),p3(5,25);
painter.line(p1,p2);
painter.setColor(Green);
painter.lineTo(p3);
point pm;
pm.add(p1,p3).divide(2);
painter.setColor(Magenta);
painter.arc(pm,p1,p3);

//draw a filled dark cyan square, width 3
painter.marker(10,5,3,"c5sf");
//draw a very dark yellow circle, default width 5
painter.marker(5,10,"y2o");
```

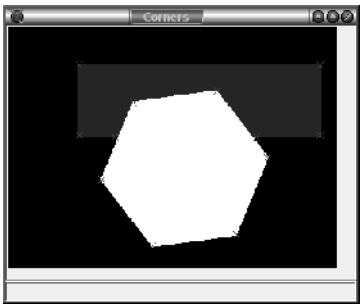
Fig. A.13. Drawing example. A yellow canvas is used for drawing. The first block draws two lines and an arc between three points; `pm` is the center of the arc. The second block shows how to use markers. The third optional argument is the width of the marker, the string has the format color with optional factor and shape with optional 'f' for filled.

```
#include "ltiImage.h"
#include "ltiViewer.h"
#include "ltiTimer.h"

//The canvas from the previous example
image canvas;

viewer v("Draw example");
v.show(canvas);
v.waitButtonPressed();
v.hide();
gtkServer::shutdown(); // workaround
passiveWait(10000);
```

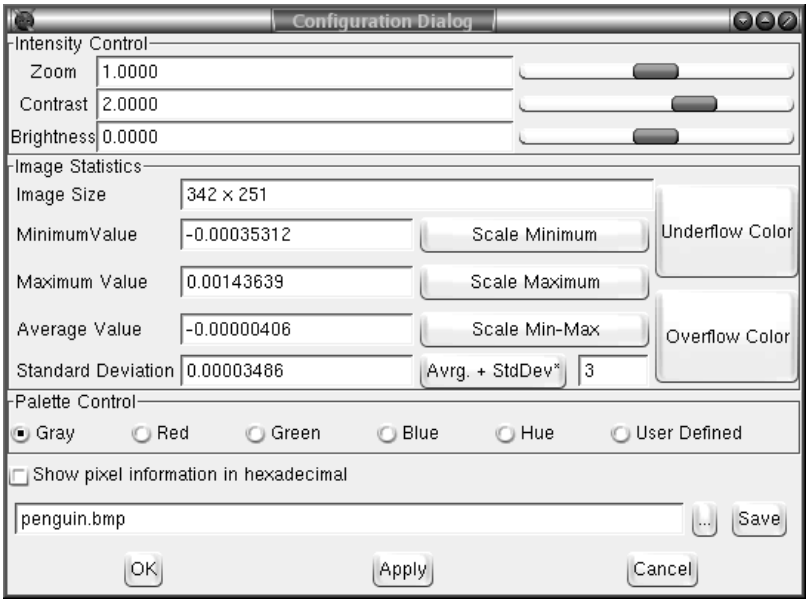
Fig. A.14. Using the viewer to display an image.



(a) Cornerness values



(b) Corners marked on the original image



(c) By using the button Scale Min-Max the values in the cornerness (a) get scaled to completely fill the range from 0 to 1.

Fig. A.15. Using the viewer configuration dialog.

via some functors in the LTI-LIB that provide interfaces to some popular cameras and framegrabbers. Furthermore, you will find functors for programming with multiple threads, for control of serial ports, as well as for system performance evaluation. Since classification and processing tasks require a plethora of mathematical functions, the LTI-LIB contains many classes for statistics and linear algebra. We encourage you to browse the HTML documentation to get an idea of the great possibilities that this library provides.

```
#include "ltiImage.h"
#include "ltiViewer.h"
#include "ltiTimer.h"

// data from the Harris example
channel ch, cornerness;
pointList corners;

image img;
img.castFrom(ch);
draw<rgbPixel> painter;
painter.use(img);
painter.marker(corners,7,"rx");

viewer vCornerness("Cornerness");
viewer vCorners("Corners");
vCornerness.show(cornerness);
vCorners.show(img);
vCorners.waitButtonPressed();
vCorners.hide();
vCornerness.hide();
passiveWait(10000);
gtkServer::shutdown();
```

Fig. A.16. The first viewer just displays the cornerness. For the second viewer the grey scale image is first transformed to an `image`. Then the corners can be marked as red crosses.

References

1. Harris, C. and Stephens, M. A Combined Corner and Edge Detector. In *Proc. 4th Alvey Vision Conference*, pages 147–151. 1988.

Appendix B

IMPRESARIO — A Graphical User Interface for Rapid Prototyping of Image Processing Systems

Lars Libuda

During development of complex software systems like new human machine interfaces, some tasks always reappear: First, the whole system has to be separated into modules which handle subtasks and communicate with each other. Afterwards, algorithms for each module have to be developed whereas each algorithm is described by its input data, output data, and parameters. After implementation the algorithms have to be optimized. Optimization includes the determination of the optimal parameter settings for each algorithm in context of the application. This is normally done by comparing an algorithm's output data for different parameter settings applied to the same input data. Therefore, the output data has to be visualized in an appropriate way. The whole process is very time consuming if everything has to be coded manually, especially visualization of more complex data types like images or 3D data.

But the process may be speeded up if a framework is available, which supports the development process. These kinds of frameworks are called *Rapid Prototyping Systems*. Ideally, such a system has to meet the following requirements:

- Input data from different sources has to be supported, e. g. image data from still images, video streams or cameras in case of image processing.
- It has to define a standardized but flexible and easy to use interface to integrate any algorithm with any input data, output data, and parameters as a reusable component in the framework.
- Processing chains consisting of several algorithms have to be composed and changed easily.
- Parameter settings of each algorithm have to be able to be changed during runtime.
- Output data of each algorithm has to be visualized on demand during execution of the processing chain.
- The system has to provide standardized methods to analyze, compare, and evaluate the results obtained from different parameter settings.
- All components have to be integrated in a graphical user interface (GUI) for ease of use.

Rapid Prototyping Systems exist for different types of applications like Borland's Delphi [5] for data base applications, Statemate [9] for industrial embedded systems, or 3D Studio Max [11] for creating animated virtual worlds.

The remaining part of this chapter introduces a Rapid Prototyping System called IMPRESARIO which was originally developed as a graphical user interface for image processing with the LTI-LIB but is not necessarily limited to it. The software can be

Table B.1. Requirements to run IMPRESARIO on a PC. The DirectX interface is needed if you plan to use a capture device or if image data should be acquired from video streams.

	Minimum	Recommended
Operating system:	Windows 98 SE©	Windows XP©
Memory:	256 MB	512 MB
Hard disc space:	500 MB	-
DirectX version:	8.1	-

found on the CD accompanying this book and is provided for a deeper understanding of the examples in the chapters 2, 5.1, and 5.3.

The next paragraph describes the requirements, installation, and deinstallation of IMPRESARIO. The following paragraphs contain a tutorial on the graphical user interface and detailed references for the different steps needed to setup an image processing system. The last three paragraphs provide details about the internals of IMPRESARIO itself, summarize the IMPRESARIO examples which are part of the chapters related to image processing, and describe how the user can extend IMPRESARIO's functionality with her own algorithms.

B.1 Requirements, Installation, and Deinstallation

To run IMPRESARIO on a personal computer, the requirements listed in Tab. B.1 must be met. To install the software, copy the file `impresario_setup.exe` contained in the directory `impresario` on the CD to the hard disc and execute it. The setup program will prompt for a directory and then extract all necessary files to this directory. Finally, make sure that the graphic accelerator is set to 32 bit color display. Now the software can be started by executing `impresario.exe` in the directory you selected during installation.

For deinstallation delete the directory where the software was copied.

B.2 Basic Components and Operations

After the first start, IMPRESARIO will show up like depicted in Fig. B.1. The user interface is divided in 5 parts.

- The menu bar and the standard toolbar are located at the top of the application window. They contain all application commands. Tab. B.2 lists all toolbar commands which are shortcuts for some menu items.
- Beneath the toolbar and on the left side of the main window, the macro browser is displayed. This is a database from which macros can be added to a document. Each macro represents an image processing algorithm in the user interface.

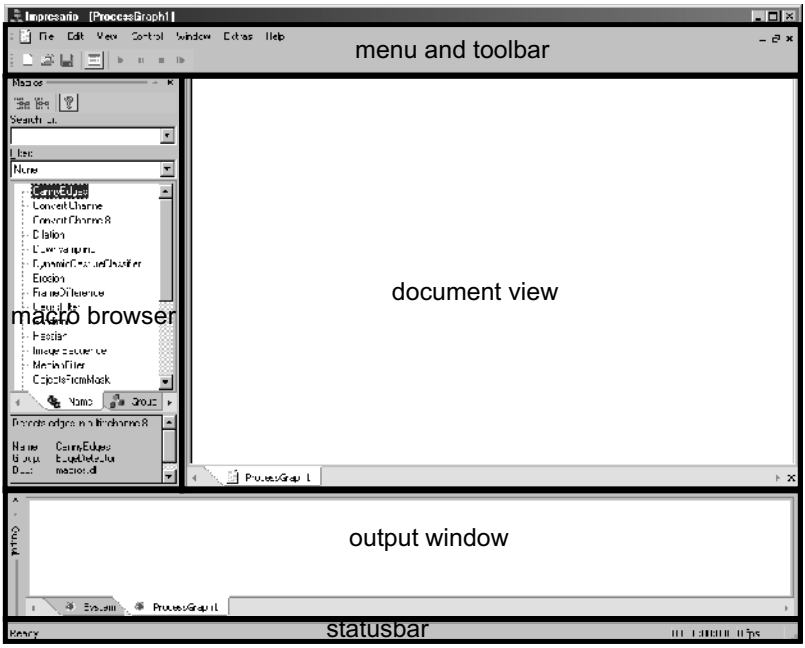


Fig. B.1. IMPRESARIO after first start. The GUI is divided in 5 parts: Menu and toolbar, macro browser, document view, output window, and statusbar.

Table B.2. Application commands located on the standard toolbar

Icon	Command	Triggered action
	Create new document	Creates a new document and displays its view.
	Open document	Displays a file selection dialog to open an existing document.
	Save document	Saves the current document to a file. If the document is unnamed, a dialog will prompt for a filename.
	Enable/Disable output	Enables or disabels the output window attached to the current document.
	Start processing	Starts image processing.
	Pause processing	Pauses image processing.
	Stop processing	Stops image processing.
	Single processing step	Starts image processing for exactly one image and stops automatically after this image has been processed.

- Right to the macro browser, the main workspace displays a view on the current document. At the moment, this document named "*ProcessGraph1*" is empty. It has to be filled with macros from the macro browser.

- Below the main workspace and the macro browser the output window displays messages generated by the application (e. g. see the tab "*System*") and by opened documents. For each document a tab is added to the output window and named after the document.
- At the bottom of the main window, the status bar shows help texts for each menu or toolbar command when it is selected and the overall processing speed of the current document in milliseconds and frames per second when the system executes the active document.

To work with this interface it is necessary to understand the basic components and operations which are described in the following subsections.

B.2.1 Document and Processing Graph

The core component of the system is the document. A document contains a processing graph and the configuration of the user interface used with this graph. A processing graph is a directed graph consisting of vertices and directed edges whereas macros represent the vertices and data links represent the edges. For more theoretical background on graphs, e. g. see [7]. Documents are stored in files with the suffix ".ipg" which is the abbreviation for IMPRESARIO processing graph. The commands to create, open, save, and close documents are located in the "File" menu. Shortcuts for some file commands can also be found on the standard toolbar (see Tab. B.2). For further discussion, we open a simple example:

1. Select the "File"→"open" command in the main menu or click the corresponding toolbar button.
2. In the upcoming dialog select the document called "CannyEdgeDetection.ipg".
3. Confirm the selection by pressing the "Open" button in the dialog.

A new view on the opened document will be created and a new tab will be added to the output window, both named after the document. The view will show the contents depicted in Fig. B.2. This is a very simple processing graph consisting of three vertices and two edges. The vertices are made up by the three macros named "Image Sequence", "SplitImage", and "CannyEdges" and two unnamed data links visualized by black arrows represent the two directed edges.

It is necessary to take a closer look at the macros because they encapsulate the algorithms for image processing and thus the graph's functionality while the links determine the data flow in the graph.

B.2.2 Macros

In IMPRESARIO macros are used to integrate algorithms for image processing in the user interface. A macro has two purposes: First, it contains an algorithm for image processing which, in general, can be described by its input data, its output data, and its parameters. This algorithm is executed by IMPRESARIO with the help of the macro. Second, a macro provides a graphical representation to change the parameters

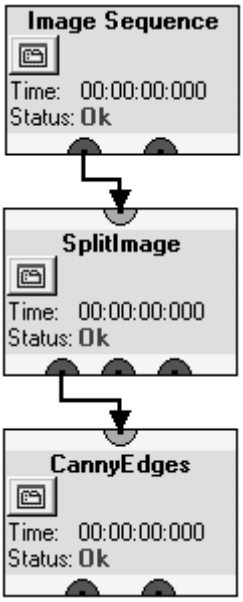


Fig. B.2. View on example document "CannyEdgeDetection.ipg" after opening it.

of the algorithm it contains. In the document view a macro is visualized as a beige block with its name printed in black. The input data is visualized as yellow input ports, the output data as red output ports (see the Canny edge example in Fig. B.2). If the mouse is hovered over the area of an input or output port, a tooltip will pop up showing the port's data type and name separated by a colon. Macro parameters are visualized in a separate macro property window which is opened by clicking the button within the macro (see Fig. B.3). Furthermore, all macros display information about the time needed to process the current input data and their state. Both information are of importance in processing mode (see Sect. B.2.3).

Our example consists of three different macros. The first macro in the graph is named "Image Sequence". This macro acquires image data from files stored on hard disk and therefore represents the source. Every document needs at least one source but there can be more than one in each document. The version of IMPRESARIO included in this book ships three different source macros. Source macros behave like all other macros but do not display the standard interface for parameter changes in their macro property window as shown for the "SplitImage" macro in Fig. B.3. Take a look at Fig. B.4 to see the macro property window of the "Image Sequence" macro. Because every document has to contain a source and source macros feature different graphical user interfaces they are described in detail in Sect. B.5.

All other macros coming with the book's CD represent LTI-LIB functors (see chapter A.4). A macro can represent a single LTI-LIB functor or an assembly of several functors. This also applies to the two remaining macros in our example. The

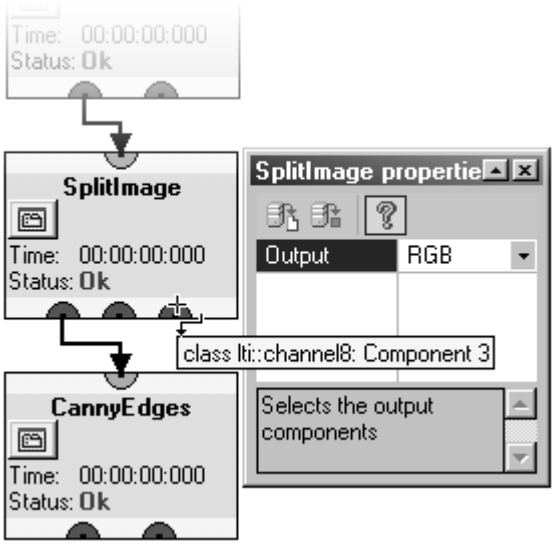


Fig. B.3. Opened macro property window of the "SplitImage" macro from the Canny edge example in Fig. B.2 and a tooltip showing a port's data type and name. The tooltip appears as soon as the mouse cursor is hovered over a port.

"SplitImage" macro in the Canny edge example (Fig. B.3) is a macro assembling all LTI-LIB functors which split color images into the components of a certain color model. The macro takes a color image of type `class lti::image` as input and splits it into three grayvalue images of type `class lti::channel8` representing the three components of a color model which can be selected by the macro's parameter named *Output*. In Fig. B.3 the RGB model is currently selected. The first grey value image of the "SplitImage" macro serves as input for the "CannyEdge" macro which contains a single functor (the Canny algorithm [3, 4]). This macro performs edge detection on the input data and outputs two grey value images of type `class lti::channel8` named "Edge channel" and "Orientation channel". The edge channel is a black and white image where all pixels are white which belong to an edge in the source image. The orientation channel encodes the direction of the gradient used to detect the edges.

A document may contain as many macros as necessary. They can be added using the macro browser, deleted, configured, and linked with each other. A detailed description of handling macros and linking them together is provided in Sect. B.3 and B.4.

But now it is time to see the Canny edge example in action.

B.2.3 Processing Mode

Up to this point we have opened a document and taken a closer look at its contents. But we haven't seen any image processing so far. If it has not already happened, open the macro property window of the "SplitImage" macro and the "Image Sequence" macro. To start image processing, select the "Control"→"Start" command from the menu or press the corresponding toolbar button (see Tab. B.2). IMPRESARIO switches from *edit mode* to *processing mode* and executes the processing graph. There is one major difference between edit and processing mode: In processing mode the structure of the graph cannot be changed. It's not possible to add or delete macros and links.

In the Canny edge example the view on the document in processing mode should look similar to Fig. B.4 now. In the property window of the *Image Sequence* macro, the currently processed image file is highlighted and the three macros display their execution times and their current state. All possible states are color coded and summarized in Tab. B.3. During processing, a macro's state will normally change between *Waiting* and *Processing*. The macros in this example execute their algorithms very fast so that a state change is hardly noticeable. The overall processing time for the whole graph is displayed on the right side of the status bar.

The processing mode can be controlled with the commands located in the "Control" menu. Beside starting, processing can be paused and stopped. In the latter case,

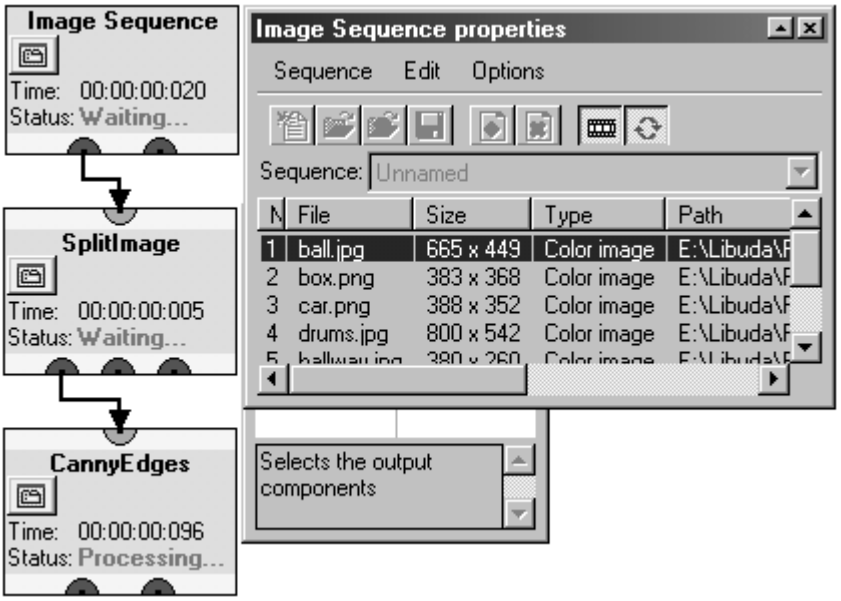


Fig. B.4. View on the Canny edge example document in processing mode. Each macro displays the execution time and its current state. The property window of the *Image Sequence* macro shows the name of the currently processed image file.

Table B.3. Macro states and their meaning

State	Color	Meaning
Ok	green	The macro is ready for processing mode
Waiting	orange	The macro is waiting for predecessing macros to complete processing
Processing	orange	The macro is currently processing its input data
Pause	blue	Processing is paused in the processing graph
Error	red	An error occurred in the macro. Processing of the whole graph is stopped immediately and an error message is displayed in the document's output window.

IMPRESARIO will switch back into edit mode. The "Snap" command demands a special note. Using this command will set IMPRESARIO to processing mode for exactly one image. After processing this image, IMPRESARIO stops processing and reactivates the edit mode.

Now that the Canny edge example is running, the only thing missing is a visualization of the macros' results.

B.2.4 Viewer

Viewer visualize the output data of every macro. A viewer is opened by double-clicking a red output port. Fig. B.5 shows two viewers from the Canny edge example. They visualize the original image "hallway.jpg" which is accessible via the "Color Image" output port of the "Image Sequence" macro and the edge channel of the "CannyEdge" macro (the left output port of the "CannyEdge" macro). When a viewer is opened the first time it appears right to the macro and in a standardized size. Most

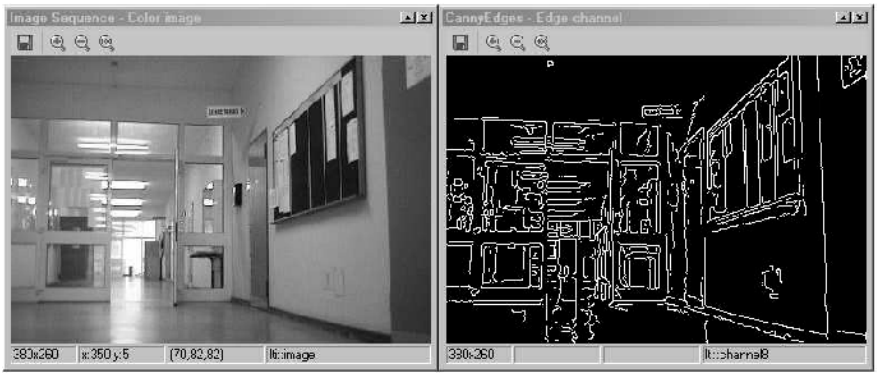


Fig. B.5. Two viewers showing visualizing some results of the Canny edge example. The left viewer shows the original image loaded by the "Image Sequence" macro, the right viewer displays the "Edge channel" of the "CannyEdge" macro.

Table B.4. Commands for image viewer

Icon	Command	Triggered action
	Save image	Saves the current image to a file. Prompts for a filename in a dialog. Supported file formats: Bitmap, Portable Network Graphics, and JPEG.
	Zoom in	Zooms in the current image by factor 2
	Zoom out	Zooms out the current image by factor 2
	Original size	Displays the image in its original size

times, this size is too small to see anything because the image data is adjusted to the window size. The viewer can be resized like every other window by clicking on its frame and moving the mouse until it reaches the desired size. Additionally, every viewer provides a toolbar with commands for zooming the image and adjusting the window size to the image size. Tab. B.4 lists all available viewer commands.

Beside the toolbar a viewer also contains a statusbar which is divided in four panes. The first pane displays the image size in pixels (width x height). When the mouse is hovered over the image the second pane shows the image coordinates the mouse cursor is pointing to. The origin of the coordinate system is the upper left corner of the image. The pixel value belonging to the current image coordinates can be found in the third pane. The value depends on the image type which is shown in the fourth pane. For an image of type `lti::image` the value is a vector consisting of the pixel's red, green, and blue fraction, each in the range of 0 to 255. For a `lti::channel8` the value is a single integer in the range of 0 to 255 and for a `lti::channel` a floating point number. Take a look at chapter A.3.3 for more information about the different image types.

Now that we know how to visualize results of the different macros it is easy to see the influence of parameter changes in a macro on the final result. For a last time, take the Canny edge example, start the execution of the processing graph and open the viewer for the edge channel of the "CannyEdge" macro. After that, open the macro property window of the "SplitImage" macro and change the parameter named *Output* from the value "RGB" to "CIE Luv" by opening the parameter's drop down box and selecting the item. You will notice changes in the edge channel. Different values will even increase the changes up to a point, where the edge channel cannot be interpreted in a meaningful way any more. Try to experiment with the parameters of both processing macros and open some more viewers to get a feeling for the user interface.

B.2.5 Summary and further Reading

In this tutorial the basic components and operations were introduced starting with the *document* as core component containing the *processing graph* which consists of macros and data links. Afterwards *macros* were introduced encapsulating image

acquisition and image processing algorithms in a common interface. The distinction between *edit mode* and *processing mode* allows the system to execute a processing graph with a constant structure. Finally, *viewers* allow the user to observe the output data of every macro in a comfortable way.

The remaining part of this chapter describes the creation of a new document with a working processing graph and provides more detailed information on the topics covered in this tutorial. The creation of a new processing graph can be separated into three steps:

1. Adding the functionality with macros and rearranging macros in the document. See Sect. B.3 for more information about this.
2. Defining the data flow with links (Sect. B.4).
3. Configuring macros. This topic is covered in Sect. B.5 and describes the usage of the standard macro property window as depicted in Fig. B.3 as well as the special user interfaces of the source macros.

For the interested reader, Sect. B.6 gives some insight into the internal structure and concept of IMPRESARIO itself and describes the features which have not been covered until this point. Last but not least, Sect. B.7 describes the examples contained in the image processing chapters of this book and Sect. B.8 addresses all readers who might like to extend IMPRESARIO with own macros.

B.3 Adding Functionality to a Document

IMPRESARIO will create an empty document if the "New" command in the main toolbar is executed. In the first step to build a working processing graph macros have to be added to the document as the functional parts of a process graph. This section provides the necessary information on how macros can be added to a document using the macro browser and how they can be rearranged within and removed from a document.

B.3.1 Using the Macro Browser

Fig. B.6 depicts two views on the macro browser which contains a database with all available macros in the system. Tab. B.5 summarizes the commands available on the macro browser's toolbar.

The macro browsers's main component is the macro tree list. It provides four different categories to view macros. The first category lists the macros alphabetically by their name. The second category named "Group" sorts all macros according to their function. The third category lists the macros by the DLL[1] they are located in, and the fourth category by their creator. A category can be accessed by clicking the corresponding tab beneath the macro tree list with the left mouse button. The right part of Fig. B.6 shows the macro browser displaying the group category and all

[1] DLL – Dynamic Link Library, see Sect. B.6 for more information.

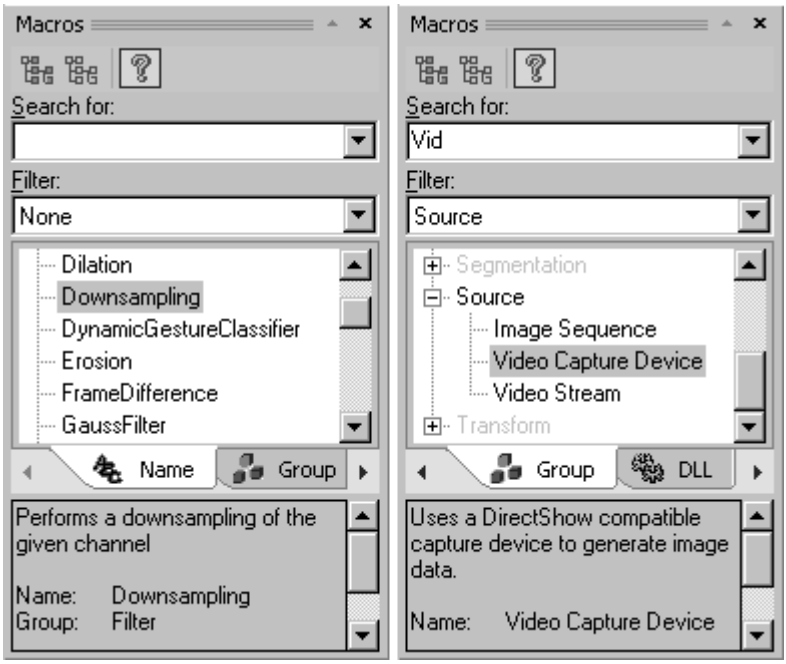


Fig. B.6. Macro browser with all available macros. *Left*: Default view with activated name category. *Right*: View on the group category with activated filter "Source".

Table B.5. Toolbar commands for the macro browser

Icon	Command	Triggered action
	Collapse all tree nodes	Collapses all tree nodes for a compact view in the filter categories
	Expand all tree nodes	Expands all tree nodes for a complete view of all macros in the filter categories
	Toggle help window	Toggles the help window in the macro selection window and macro property window

macros of the group "Source" which will be discussed in detail in Sect. B.5. In each category macros can be selected by a left click on their name. As soon as a macro is selected, a more detailed description is displayed in the help area beneath the tabs.

There are two ways to add macros from the macro selection window to a document. The first one uses the "Search for" input field. Activate the field with a left click. A blinking cursor appears in the field. Now it's possible to enter characters. The first macro name which matches the entered character sequence is selected in the macro tree list. The search can be narrowed by using the "Filter" drop down list box. The filter options vary depending on the selected filter category. The name

category does not provide any filter options but e. g. the group category allows to search only one group (Right part of Fig. B.6). As soon as the Enter key is pressed in the "Search for" input field, the selected macro is added to the active document. It appears in the document view's upper left corner. The second and more comfortable way to add macros works by drag and drop:

1. Click the desired macro with the left mouse button and hold it down.
2. Move the mouse cursor to some free space in the document view while the left mouse button is still held down.
3. Release the left mouse button in the document view and the selected macro will be inserted at the mouse cursor's position.

This operation can be cancelled any time by pressing the Escape key.

B.3.2 Rearranging and Deleting Macros

After macros have been added to a document, they can be rearranged or removed from the document.

Macros can be rearranged by a simple drag and drop operation. Move the mouse cursor onto the desired macro, press and hold down the left mouse button and move the cursor to the desired location in the document view. A rectangle representing the macro's borders will move with the cursor. If the cursor is moved to the view's right or bottom edge, the view will start to scroll to reach invisible parts of the document. As soon as the desired location is reached release the mouse button and the macro is moved there.

To remove a macro from the document, select it by clicking it with the left mouse button. The macro's border will turn light green to indicate that it is selected. Execute the "Delete" command in the "Edit" menu to remove the macro. In the current version of IMPRESARIO only one macro can be selected and removed at one time.

B.4 Defining the Data Flow with Links

After all necessary macros are added to the document, the process graph has to be completed by defining the data flow with links. This section gives a brief overview on how to create and remove links from a document.

B.4.1 Creating a Link

To realize data exchange between different macros they have to be linked. All links underly the following rules:

- A link can only be established between an output port (colored red) and an input port (colored yellow).
- The input port and the output port have to support the same data type. A port's data type is displayed as a tooltip when the mouse pointer is hovered over the port and left there unmoved for a moment.

Table B.6. Cursor appearances during link creation

Cursor	Name	Description
	Link allowed	It is allowed to start link creation from the currently selected port (source) or a link can be established to the selected port (destination).
	Link forbidden	A link cannot be created starting from the currently selected port or a link cannot be established between the source port and the currently selected destination port.

- An output port may be connected to many input ports but one input port excepts exactly one incoming link from an output port.

Links are created by drag and drop operations with the mouse by the following steps:

1. The mouse cursor has to be placed on a red output port or a yellow input port. The cursor appearance changes according to Tab. B.6. The "Link forbidden" cursor will appear only if link creation is started from a yellow input port which is already connected to an output port.
2. If it is allowed to start a link from the selected port (indicated by the "Link allowed" cursor) press the left mouse button, hold it down and start to move the mouse to the desired destination port. The system draws a gray line from the starting point of the link to the current mouse position and the mouse cursor changes to the "Link forbidden" cursor to indicate an incomplete link (see left column in Fig. B.7).
3. To connect two macros which are not concurrently visible in the document view, move the cursor to the view's border. The view will start to scroll without aborting the link operation. Scroll the view until the desired macro becomes visible.
4. As soon as the desired destination port is reached, the system indicates wether it is possible to establish the link or not. If the possible link is valid the gray line will turn green and the cursor will change to "Link allowed" (middle column in Fig. B.7). In case of an invalid link the gray line turns red and the cursor appearances does not change (right column in Fig. B.7). Invalid links are indicated in two cases: First, the source and destination port are both input or output ports. Second, the data types of the source and destination port are unequal.
5. If a valid link is indicated, release the left mouse button to create the link. The green line will be replaced by a black arrow. If the mouse button is released when the system indicates an incomplete or invalid link, the operation will be cancelled.

Link creation can also be cancelled at any time by pressing the Escape key.

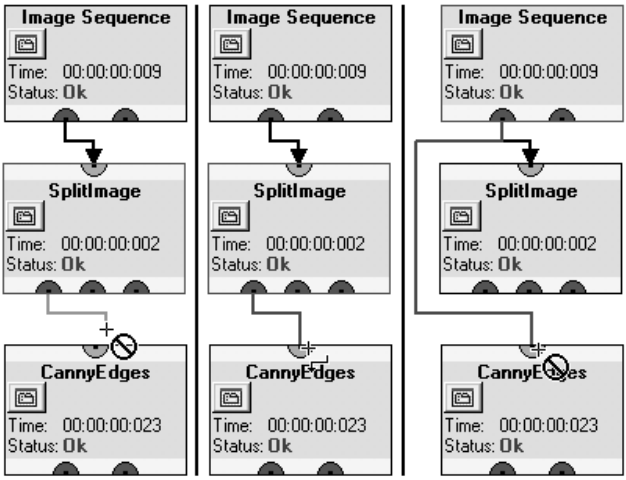


Fig. B.7. Possible appearances of links during creation. *Left column*: Gray line indicating incomplete link. *Middle column*: Green line indicating valid complete link. *Right column*: Red line indicating invalid link.

B.4.2 Removing a Link

To remove an existing link indicated by a black arrow, it has to be selected. Click on the link's black arrow with the left mouse button and the arrow turns light green. Now the link is selected. Execute the "Delete" command in the "Edit" menu to remove the link.

B.5 Configuring Macros

After adding macros to the document and connecting them by links the process graph is nearly ready to be processed. The final step includes the configuration of the macros. This especially applies to the three source macros coming with the version of IMPRESARIO on the books CD (see the group "Source" in the macro browser, Fig. B.6):

- *Image Sequence*: This macro allows the processing of single still images and sequences of still images.
- *Video Capture Device*: Here, image data can be gained from video capture devices which support the DirectX 8.1 interface [6]. These are mainly USB or Firewire cameras [1, 10] like the Phillips ToUCam or the Unibrain Fire-i camera.
- *Video Stream*: The *Video Stream* macro extracts image data from video files of type AVI and MPEG-1. Therefore, the DirectX 8.1 interface is also required.

The source macros have to be fed with information, where they can find the image data. Therefore they provide customized user interfaces which are described in detail in the following subsections. All other macros coming with this book provide a standard user interface which will be discussed at the end of this section.

B.5.1 *Image Sequence*

The *Image Sequence* macro handles still images of the types Bitmap (BMP), Portable Networks Graphics (PNG), and JPEG. These kinds of images may be used as input data to the processing graph either as single images or as a sequence of several images. Therefore, the *Image Sequence* macro maintains so called sequence files which are suffixed with ".seq". Sequence files are simple text files containing the path to one existing image file in each line. They can be generated manually with any text editor or much more comfortably with the macro's user interface depicted in Fig. B.8. On top, the interface contains a menu bar and a toolbar for image sequence manipulation. The commands are summarized in Tab. B.7. The drop down list box beneath the toolbar contains the currently and recently used sequences for fast access. Beneath the list box the image list displays the contents of the current sequence.

Creating and Loading Sequences

By default, an empty and unnamed image sequence is generated when a new document is created. A new sequence can be created by selecting the "Create sequence" command (see Tab. B.7). To load an existing sequence, press the "Load sequence" button in the toolbar. A standard open file dialog appears where the desired sequence can be selected. An alternative way to open one of the recently used sequences is to open the drop down list box and select one of its items. The corresponding sequence will be opened immediately if it exists.

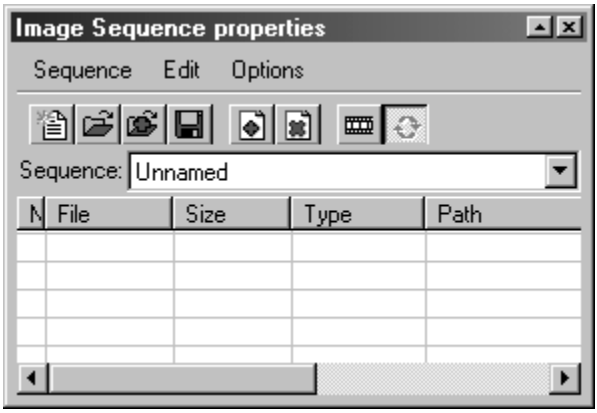


Fig. B.8. User interface of the *Image Sequence* macro. It consists of the command toolbar, the drop down list box with the last opened sequences and the list containing the image files to be processed.

Table B.7. Commands for image sequence manipulation

Icon	Command	Triggered action
	Create new sequence	Asks to save the current sequence if it has been modified and clears the list to begin a new image sequence.
	Load sequence	Asks to save the current sequence if it has been modified and displays a file selection dialog to open an existing sequence file.
	Add sequence	Adds the contents of a different sequence file to the currently used sequence. Displays a dialog to select the sequence file to add.
	Save sequence	Saves the current sequence to a file. If the sequence is unnamed, a file dialog will prompt for a filename.
n. a.	Save sequence as	Saves the current sequence to a file and always prompts for a filename. This command is only available in the menu.
	Add images	Dislays a file dialog to select one or more image files which are added to the image list.
	Remove images	Removes all selected image files from the list.
	Play as sequence	When the icon is pressed, all images in the list will be used as input in the order they appear on the list. When the icon is not pressed, the selected image will be used as input only. The input image can be changed by selecting a different image file from the list.
	Play in loop	When this option is selected, the image sequence is played repeatedly. Otherwise, processing stops as soon as the end of the sequence is reached. Please note, that this option is only available if "Play as sequence" is also selected.

Adding Images to a Sequence

There are two ways to add images to the image list: It is possible to add a whole sequence by choosing the "Add sequence" command or image files with the "Add images" command (see Tab. B.7).

Choosing the "Add sequence" command brings up a dialog which allows to select an existing sequence file. After confirming the selected file by pressing the "Open" button in the dialog, all images referenced in the selected sequence file are appended to the image list.

To add image files, choose the "Add images" command. A dialog appears which allows to select one or more image files. Confirm the selection by pressing the "Add" button in the dialog and the image files are added to the list (see Fig. B.9). The list displays some information about each image. The first column contains the position of the image in the list, the second column the file name, the third column the resolution of the image in the format (width x height) in pixels, the fourth column the type of the image ("color image" or "grey value image"), and the fifth column the absolute path to the image file.

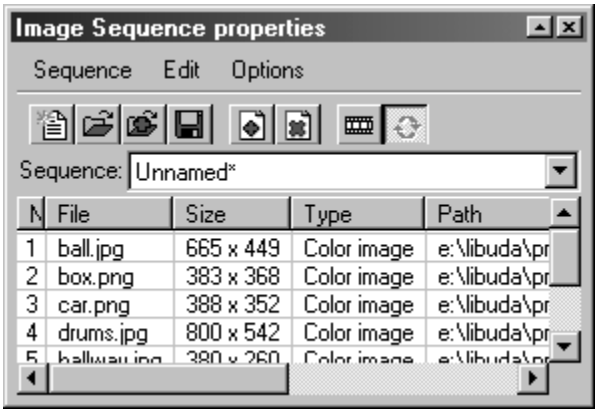


Fig. B.9. Appearance of the *Image Sequence* user interface after adding new images. The asterix appended to the sequence name indicates, that the sequence was changed and needs to be saved to preserve the changes.

If errors occur during analysis of an image to be added, corresponding messages will be displayed in the output window of the document.

Changing the position of an image in the sequence

It is possible to change the position of an image in the list by selecting it with the left mouse button and dragging it to the new position while the left mouse button is held down. After releasing the mouse button at the new position the image will be moved there.

Removing Images

To remove images from the current sequence, they have to be selected in the list. A single image is selected by simply clicking it with the left mouse button. Multiple images may be selected by holding the Ctrl- or Shift-Key during selection with the left mouse button. When all desired images are selected, choose the "Remove images" command from the toolbar.

To clear the whole image list in one step, choose the "Create new image" command from the toolbar.

Storing Sequences

When images are added, moved, or removed the sequence is modified. A modification is indicated by an asterix (*) which is appended to the sequence name in the drop down list box (see Fig. B.9). To preserve the changes the sequence has to be stored on disk.

A sequence can be saved by selecting the "Save sequence" command. If the sequence is unnamed, a dialog will prompt for a filename. Subsequent save commands for the same sequence will be executed without further prompts. To save the current sequence with a different name, choose the "Save sequence as" command. Here, you will always be prompted for a file name.

The system automatically asks to save a sequence if changes to the current sequence may be lost due to a following execution of a "Create new sequence" or "Load sequence" command.

Image control during processing

When the process graph is processed, the "Play as sequence" command allows to switch between two different processing modes of the image sequence. In the first mode, every image is processed in the order it appears in the list. In this mode, the list is grayed and any interaction with the list is impossible. Further, the "Play in loop" command allows to toggle between one and infinite iterations of the sequence. The second mode processes only the selected image. It is possible to change the processed image by selecting a different one in the list. The "Play in loop" command is not available in this mode.

During processing of the image graph it is not possible to modify the image list in any way.

Image Sequence provides two output ports. The left one contains the currently loaded image and the right one the path to the file where the image was loaded from. To visualize any output double click the ports of the macro. A viewer window will open and display the currently processed data.

B.5.2 *Video Capture Device*

With the *Video Capture Device* macro it is possible to acquire life image data from any capture device which supports Microsoft's DirectX interface [6] in the version 8.1 or above. Most USB cameras and Firewire cameras support this interface [1, 10]. The macros's user interface is shown in Fig. B.10 and the meaning of the two buttons is described in Tab. B.8.

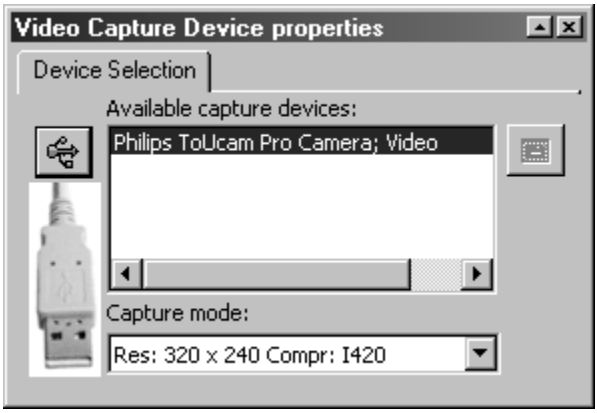


Fig. B.10. Configuration page of the *Video Capture Device* macro. The list box shows all detected capture devices and the combo box all possible capture modes supported by the selected device. In this case a Phillips ToUCam is detected and selected.

Table B.8. Commands for *Video Capture Device* macro

Icon	Command	Triggered action
	Scan system	Scans the system for connected capture devices with DirectX 8.1 support and updates the list box.
	Show device properties	Calls the properties dialog for the selected device if such a dialog is supported by the device driver software.

The main list box in the middle shows all available devices for image capturing and is updated on program start. If a new capture device is connected, press the "Scan system" button to update the list box. By default, the first found device is selected for image acquisition. To change the device, select the corresponding list box entry with the left mouse button.

The drop down list box beneath the main list box contains all capture modes supported by the selected device. A capture mode consists of the video resolution and the used compression method. By default, the first capture mode is selected by the system. It can be changed by opening the drop down list box and selecting a different entry with the left mouse button.

These three controls operate in edit mode and are disabled in processing mode. The button "Show device properties" is available in processing mode only (see Fig. B.11). Pressing this button brings up the properties dialog of the selected device. Pressing the button once again will close the dialog. The properties dialog is normally provided by the device driver but it is possible that some drivers do not offer such a dialog. In this case, there won't be any system reaction if the button is

Fig. B.11. Configuration page of the *Video Capture Device* macro in processing mode. In this mode only the "Show device properties" button is enabled to display and change the device dependent property dialog.

pressed. To see the life image double click the red output port of the *Video Capture Device* macro.

B.5.3 *Video Stream*

The *Video Stream* macro gains image data from video files of type AVI (Audio Video Interleaved [2]) and MPEG-1 (Motion Picture Experts Group). The image data of videos may be compressed with different Codecs (Compressor – Decompressor). To view a video on a computer, the required Codec for decompression has to be installed. If this is not the case the video will not play. Furthermore, dependent on the Codec being used for decompression, some options may not be available during video playback.

To configure the macro, the graphical interface depicted in Fig. B.12 is provided. The interface consists of the video file selection area at the top, the video information area in the middle, and the frame control at the bottom of the window.

Loading a video stream

To load a video file, click the "Open" button in the video file selection area. An open file dialog appears where the desired video file can be selected. After pressing the "Open" button in the file dialog the selected video file is validated. If it can be opened its file name is displayed in the box labeled "*Video file*" and information about the video stream is shown in the video information area (Fig. B.12). This includes the resolution of the video in (width x height) in pixels, the total number of frames contained in the video, the duration, and the frame rate. If the selected file cannot be opened a corresponding message will appear in the output window of the current document. The most probable reason for this failure is a missing Codec for

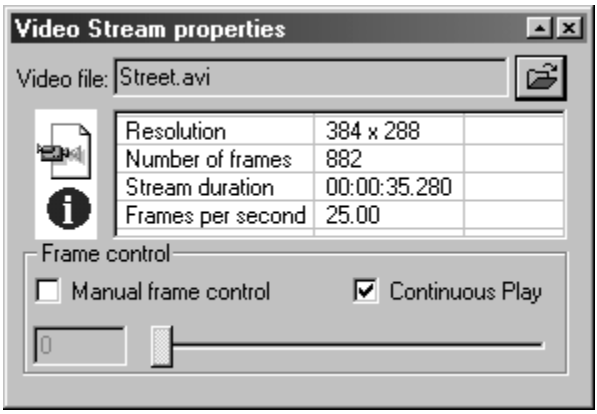


Fig. B.12. User interface of the *Video Stream* macro consisting of the video file selection area, the video information area, and the frame control. In this example a video file named "Street.avi" has already been opened.

decompression in case of AVI video files or an interlaced MPEG video file which is currently not supported.

After a video is loaded, the macro is ready to be processed.

Video Stream Control During Processing

In the default settings, a video is processed frame by frame and when the end is reached, it is started at the beginning again. The frame control allows to change the standard behavior. It consists of four controls:

- Manual frame control: If this checkbox is selected, the edit box and the slider control for selecting a frame will be activated and the "Continuous Play" checkbox will be disabled. Only the selected frame will be processed until a different one is chosen (see Fig. B.13).
  Please note, that the availability of manual frame control depends on the Codec being used for decompressing the video. If it is not supported by the Codec, the check box will be disabled and a corresponding warning message will be displayed in the output window of the document.
- Continuous play: This checkbox will be only available if the "Manual frame control" is not selected and tells the system how to continue processing after the last frame of the video has been reached. If the checkbox is selected, the video will be rewound and processing will continue with the first frame. If the checkbox is not selected, processing will stop after the last frame.
- Frame edit box: If "Manual frame control" is not selected, this control is disabled and just displays the number of the frame currently processed. During manual control it is possible to enter the number of the desired frame. Invalid frame numbers cannot be entered. The entered frame will be processed until a different frame is selected.
- Frame slider control: The slider is coupled with the frame edit box and is also disabled when "Manual frame control" is switched off. In this case, the slider in-

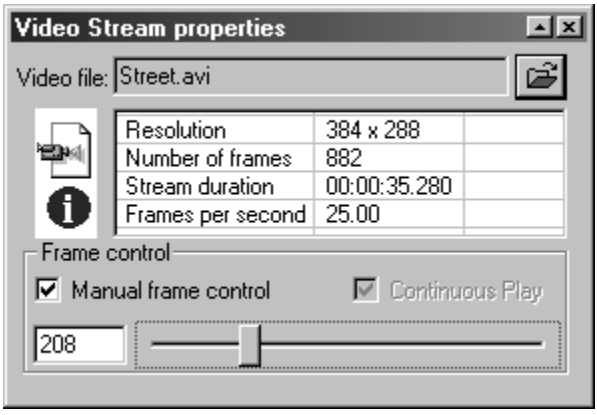


Fig. B.13. *Video Stream* macro with activated manual frame control. The "Continuous Play" checkbox is disabled and the edit and slider control allow the selection of a desired frame.

dicates approximately the position of the currently processed frame in the video. During manual control a frame can be selected by dragging the slider with the mouse. The selected frame will be displayed in the edit control and processed subsequently.

To view the image data of the processed video frame double click the red output port of the *Video Stream* macro. This will bring up a viewer. Finally, it has to be noted, that the video is not played with the frame rate denoted in the video information area but as fast as possible because the software is not intended as a pure video viewer. The main goal is to allow random access to every frame contained in the video. This requires sometimes more time for decompression than playing the frames in sequential order because of interframe compression.

B.5.4 Standard Macro Property Window

Except the source macros introduced in the last three subsections all other macros use a standardized user interface which allows the configuration of their parameters. We already encountered a standard macro property window in the tutorial (Fig. B.3 in Sect. B.2.2). Fig. B.14 shows the standard property window of the *CannyEdges* macro and Tab. B.9 summarizes the available toolbar commands.

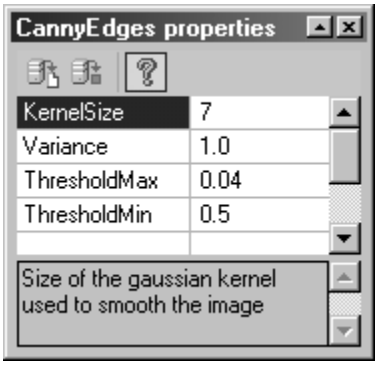


Fig. B.14. Appearance of a standard macro property window.

Table B.9. Toolbar commands for the standard macro property window

Icon	Command	Triggered action
	Restore default values	Restores default values for all parameters in the macro property window
	Restore default value	Restores default value for selected parameter only
	Toggle help window	Toggles the help window in the macro selection window and macro property window

A macro property window is attached to each macro in the document view. It can be opened by clicking the button within the macro with the left mouse button. Parameter changes in these windows apply only to the macro the window is attached to. The macro property windows can be distinguished by their title which display the macro names they are attached to. However, all windows have the same layout and are operated in the same way.

The main component of a macro property window is a list holding a parameter value pair in each line. Therefore it consists of two columns. The first column contains the parameter's name, the second column the current parameter value. A parameter can be selected with a left click in the corresponding line. The help window beneath the parameter list displays more information about the selected parameter. It can be hidden by selecting the "Toggle help window" command from the toolbar contained in the macro property window (see Tab. B.9). The value of the selected parameter can be changed in the value field. Depending of the parameter's data type the following possibilities exist:

- `String`: To edit a parameter value of type `String`, click the parameter's value field with the left mouse button. A blinking cursor will appear in the field and all printable characters can be entered. It is not necessary to confirm the text by pressing the "Enter" key. All changes are noticed by the programm.
- `Integer`: An integer value can be changed like the value of type `String` except that only characters representing decimal digits can be entered and optionally a "+" or "−" sign as the first character.
- `Floating point number`: Floating point numbers can be changed like integer values. The field accepts the same characters like the integer field and additionally the period (".") to separate the integer part and the floating part of the number.
- `Boolean`: A boolean value differs between the states "true" and "false". The state can be changed by choosing one of these options from the drop down list box in the parameter's value field or by a double click in the field with the left mouse button. The latter method will toggle the value from "true" to "false" or vice versa.
- `Option list`: An option list represents a collection of predefined values. This collection is presented in a drop down list box where it is possible to select one of the values. The current value is displayed in the parameter's value field.
- `File`: A parameter of type `File` references a physical file on hard drive. To change this value, press the button in the value field. A file dialog will appear to select a new file. It is not possible to enter a new file name into the value field directly.
- `Folder`: This kind of parameter references a folder or directory on hard drive, e. g. to load sequences of images from this folder or to store some data in this place. It is changed like a file parameter value except that a folder dialog is presented instead of a file dialog.
- `Color`: Colors are used for visualization purposes of processing results. Sometimes it is necessary to change colors for better contrasts in the visualization.

Colors are changed like files and folders but a color selection dialog appears instead of the file or folder dialog.

When experimenting with different parameter values especially in processing mode it may happen that the results produced by the macros look very strange and cannot be interpreted in any sense. This problem can be solved by restoring the default values for all macro parameters. To do this, the macro property window's toolbar (see Tab. B.9) provides two commands. The left most button restores all parameters to their default values whereas the middle button just restores the default value for the currently selected parameter.

B.6 Advanced Features and Internals of IMPRESARIO

For the interested reader this section covers the not yet mentioned features of the user interface which are not necessarily needed to work with IMPRESARIO and also reveals some of the concepts used in this software.

B.6.1 Customizing the Graphical User Interface

Because IMPRESARIO follows the guidelines for standardized user interfaces it is highly customizable in its appearance. Every component of the interface can be resized, hidden or docked at different locations using drag and drop operations. Additionally the menus "Window" and "View" offer some commands for GUI customization. The former may be used for the arrangement and management of the opened document views. The latter allows to recover components which were hidden before and is shown in Fig. B.15. The check mark next to a component indicates its visibility. The three components "Process information", "Console output", and "Blackboard" are hidden and haven't been mentioned so far. The first two components were developed to provide information during development of IMPRESARIO itself. The console output window displays text messages which are normally send to the console (DOS box) and was integrated for debugging purposes. In the software

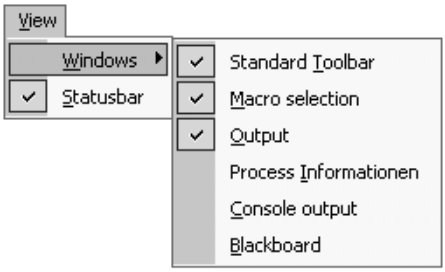


Fig. B.15. View menu. With this menu the parts of the graphical user interface can be hidden or made visible again.

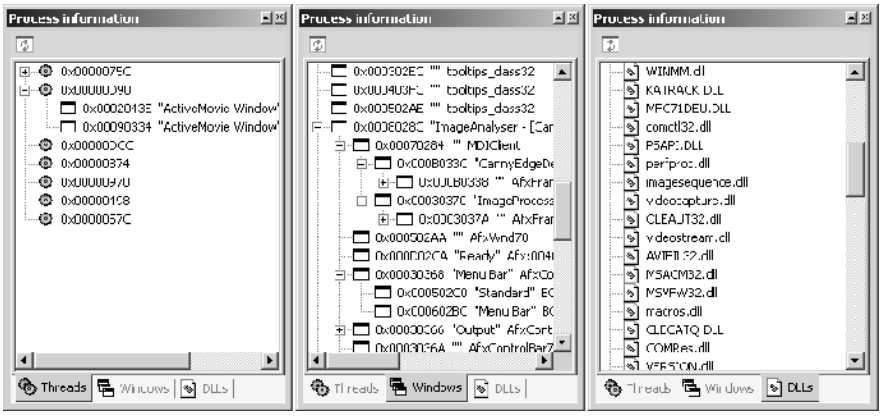


Fig. B.16. The three pages of the process information window displaying the running threads, the window hierarchy of IMPRESARIO and the linked DLLs. The information have to be updated manually by pressing the "Update" command on the toolbar.

distribution at hand, only the macro execution order is printed in this window. The process information window shows the threads with their attached windows running in IMPRESARIO, the window hierarchy as defined by the operating system, and the used Dynamic Link Libraries on separate pages (see Fig. B.16). All information have to be updated manually by pressing the "Update" button on the toolbar.

A Dynamic Link Library (DLL) is a file with the suffix ".dll" which contains executable program code and is loaded into memory by the operating system on application demand [8]. Code within a DLL may be shared among several applications. DLLs play an important role in the concept of IMPRESARIO as we will see in Sect. B.6.2.

The third component is the blackboard. Fig. B.17 shows a view on the blackboard when it is activated using the menu depicted in Fig. B.15. A blackboard can be considered as a place where information can be exchanged globally. This enables macros to define a kind of global variable. For a user of IMPRESARIO it is not very

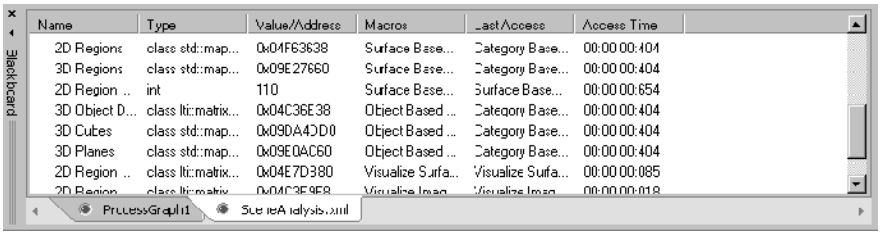


Fig. B.17. View on the activated blackboard. The figure shows a filled blackboard from a document which is not part of the distribution coming with this book.

important but for those readers who might like to develop their own macros it may be of interest. In this case please refer to Sect. B.8 and the online documentation for macro developers. However, the blackboard is not used in the examples included in this book.

B.6.2 Macros and Dynamic Link Libraries

As mentioned in the tutorial, the most important components are the macros. Macros are based on executable code which is stored in Dynamic Link Libraries (DLLs) and loaded at program start. Therefore the libraries have to be located in a directory known to the program. Take a look at the "System" tab in the output window. It reports the loaded *Macro-DLLs*. Each Macro-DLL contains one or more macros which are added to the macro browser. In the version of IMPRESARIO coming with this book several Macro-DLLs are delivered: The three DLLs `ImageSequence.dll`, `VideoCapture.dll`, and `VideoStream.dll` contain the three source macros. The two remaining Macro-DLLs, namely `macros.dll` and `ammi.dll`, contain all other macros. The former file is publicly available, the latter file was exclusively programmed for the examples in this book. The DLL concept allows an easy extension of the system's functionality without changing the application itself. An Application Programming Interface (API) exists for development of own Macro-DLLs. For integration in IMPRESARIO the developed DLLs just have to be copied to the directories where IMPRESARIO expects them. The directories where IMPRESARIO searches for DLLs can be configured in the settings dialog (see Sect. B.6.3).

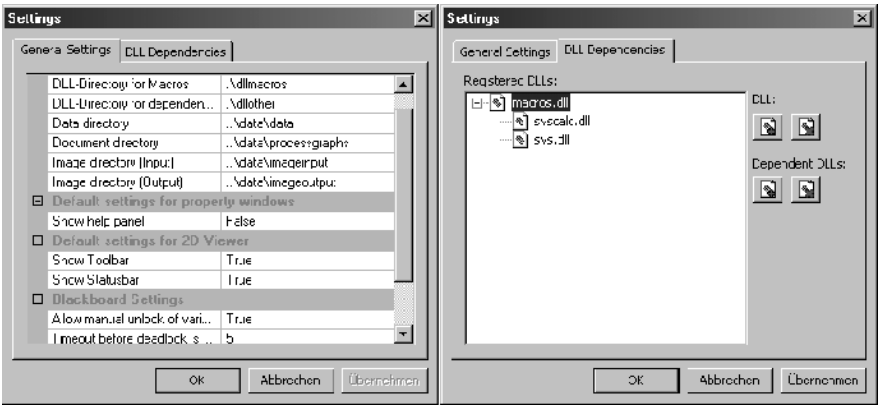


Fig. B.18. The two pages of the settings dialog accessible via the menu "Extras"→"Settings...". *Left*: Default settings for directories, macro property windows, and viewers. *Right*: Fictive declaration of dependent dynamic link libraries. Because no third party products are shipped with the version of IMPRESARIO in this book, this page is empty by default.

B.6.3 Settings Dialog and Directories

The settings dialog allows to customize some aspects of IMPRESARIO. It is opened by selecting the "Settings..." command in the "Extras" menu. Fig. B.18 shows the two pages of the dialog named "General Settings" and "DLL Dependencies".

The "General Settings" page displays a listbox with four categories. The first category contains the directories, where files of different types are searched for. A directory can be changed by selecting the line in the listbox with the left mouse button. A button will appear right to the directory name. Pushing this button brings up a dialog where the new directory can be selected. The directories and their meaning within IMPRESARIO is summarized in Tab. B.10. The remaining three categories allow to set some default options for newly created macro property windows, viewers, and the blackboard.

The "DLL Dependencies" page is not used in the version of IMPRESARIO shipped on the book's CD but for the sake of completeness its purpose has to be mentioned. As described in Sect. B.6.2, the functionality of IMPRESARIO can be extended by developing new Macro-DLLs. Sometimes it is necessary to incorporate third party software into one's own code which may also be delivered in DLLs for further distribution. The DLL developed by oneself then depends on the existence of the third party DLLs. These dependencies can be declared in the "DLL Dependencies" page and allow more detailed checks and error messages at system start up. On the right side in Fig. B.18, a fictive example of a DLL dependency is shown.

Table B.10. Directories used in IMPRESARIO and their meaning

Directory	Meaning
DLL-Directory for Macros	Directory where IMPRESARIO searches for Macro-DLLs at program start. This directory should not be changed. Otherwise it may happen, that no macros are available next time IMPRESARIO is used. Check the "System" tab in the output window for loaded Macro-DLLs.
DLL-Directory for dependent files	In this directory dependent files are stored which may be shipped with third party software. In this version of IMPRESARIO no third party software is contained and this directory remains empty. So, changing it won't have any effect.
Data directory	Macros will store their data in this directory by default.
Document directory	Directory where the document files (*.ipg) are loaded from and stored by default.
Image directory (Input)	Still images in the format Bitmap, Portable Network Graphics and JPEG are stored in this directory for use in the *Image Sequence* source.
Image directory (Output)	Images produced by macros and viewers will be stored here.

Table B.11. Summary of examples using IMPRESARIO

Example	Process graph file	Section
Hand Gesture Commands (Chapter 2):		
Static gesture classification	`StaticGestureClassifier.ipg`	2.1.4
Dynamic gesture classification		
from live video	`DynamicGestureClassifier.ipg`	2.1.5
from images	`DynamicGestureClassifier FromDisk.ipg`	2.1.5
Record gestures for own training	`RecordGestures.ipg`	2.1.5
Facial Expression Commands (Chapter 2):		
Eye detection	`FaceAnalysis_Eye.ipg`	2.2
Lip tracking	`FaceAnalysis_Mouth.ipg`	2.2
Face recognition (Chapter 5):		
Eigenface PCA calculation	`Eigenfaces_calcPCA.ipg`	5.1.7.2
Eigenface classifier training	`Eigenfaces_Train.ipg`	5.1.7.2
Eigenface face recognition	`Eigenfaces_Test.ipg`	5.1.7.2
Component classifier training	`Components_Train.ipg`	5.1.7.3
Component based face recognition	`Components_Test.ipg`	5.1.7.3
People tracking (Chapter 5):		
People tracking indoor	`PeopleTracking_Indoor.ipg`	5.3.5
People tracking outdoor	`PeopleTracking_Parking.ipg`	5.3.5
IMPRESARIO (Appendix B):		
Canny edge detection	`CannyEdgeDetection.ipg`	B.2

B.7 IMPRESARIO Examples in this Book

This section summarizes all examples in this book using IMPRESARIO. For the reader's convenience all examples are stored in process graph files which can be found in the `documents` folder in IMPRESARIO's installation directory. Tab. B.11 names the example, the corresponding file, and the section where the example is discussed in detail. Please refer to this section before you start processing a graph.

B.8 Extending IMPRESARIO with New Macros

This section addresses all readers who might like to extend IMPRESARIO's functionality with their own Macro-DLLs.

B.8.1 Requirements

The API for developing Macro-DLLs is written in C++ because the LTI-LIB has also been developed using this language. To successfully develop new Macro-DLLs the reader has to be familiar with

- concepts and usage of IMPRESARIO,
- concepts and usage of the LTI-LIB,
- object oriented programming in C++ including class inheritance, data encapsulation, polymorphism, and template usage,
- and DLL concepts and programming on Windows platforms.

To use the API an ANSI C++ compatible compiler is needed. The API contains sample projects for Microsoft's Visual Studio .NET 2003. This development environment is therefore recommended. Please note, that Microsoft's Visual Studio 6 or prior versions of this product will not compile the source code because of insufficient template support. Different compilers haven't been tested so far.

Additionally an installed *Perl* environment will be needed if the *Perl* scripts coming with the API are used.

B.8.2 API Installation and further Reading

The necessary files containing the API for Macro-DLLs as well as further documentation and code examples can be found on the book's CD in the file `impresario_api_setup.exe` in the `Impresario` directory. Copy this file to your hard drive and execute it. You will be prompted for a directory. Please provide the directory IMPRESARIO is located in. The setup program will decompress several files into a separate subdirectory named `macrodev`. This subdirectory contains a file called `documentation.html` which contains the complete documentation for the API. Please refer to this file for further reading.

B.8.3 IMPRESARIO Updates

IMPRESARIO is freeware and was not developed for this book only but for the community of LTI-LIB users in first place. Updates and bug fixes for IMPRESARIO as well as new macros can be obtained from the World Wide Web at http://www.techinfo.rwth-aachen.de/Software/Impresario/index.html

References

1. Anderson, D. *FireWire System Architecture*. Addison-Wesley, 1998. ISBN 0201485354.
2. Austerberry, D. *Technology of Video and Audio Streaming*. Focal Press, 2002. ISBN 024051694X.
3. Canny, J. Finding Edges and Lines in Images. Technical report, MIT, Artificial Intelligence Laboratory, Cambridge, MA, 1983.
4. Canny, J. A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, 1986.
5. Cantu, M. *Mastering Delphi 7*. Sybex Inc, 2003. ISBN 078214201X.
6. Gray, K. *The Microsoft DirectX 9 Programmable Graphics Pipline*. Microsoft Press International, 2003. ISBN 0735616531.

7. Gross, J. L. and Yellen, J., editors. *Handbook of Graph Theory*. CRC Press, 2003. ISBN 1584880902.
8. Hart, J. M. *Windows System Programming*. Addison-Wesley, 3rd edition, 2004. ISBN 0321256190.
9. i-Logix Inc., Andover, MA. *Statemate Magnum Reference Manual, Version 1.2*, 1997. Part-No. D-1100-37-C.
10. Jaff, K. *USB Handbook: Based on USB Specification Revision 1.0*. Annabooks, 1997. ISBN 0929392396.
11. Woods, C., Bicalho, A., and Murray, C. *Mastering 3D Studio Max 4*. Sybex Inc, 2001. ISBN 0782129382.

Index

4-neighborhood, 20
8-neighborhood, 20

AAC, 393
accommodation, 266, 271
acoustic alert, 382
acoustic interfaces, 272
Active Appearance Model, 68, 186
active shape model, 87, 186
active steering, 383, 385
active stereo, 268
AdaBoost, 64, 183
adaptive filtering, 384
adjacency
  pixel, 20
agent, 349
ALBERT, 363
alerts and commands, 381
analytic approaches, 186
application knowledge, 386
architecture
  event-driven, 348
ARMAR, 322
articulatory features, 178
assistance in manual control, 372
assisted interaction, 369
auditory display, 273
augmented reality, 266, 392
auto completion, 394
automatic speech recognition, 141, 154
automation, 384
autonomous systems, 358

background
  Gaussian mixture model, 104
  median model, 103
  modeling/subtraction, 102
background subtraction, 234
  shadow reduction, 255
bag-of-words, 182
Bakis topology, 39
Bayes' rule, 148
Bayesian Belief Networks, 388, 389
Bayesian nets, 182
beliefs, desires, and intentions model, 159
binaural synthesis, 274, 275
black box models, 376
border
  computation, 21
  length, 23
  points, 20
braking augmentation, 383

camera model, 251
cascaded classifier, 66
CAVE, 266
cheremes, 119
circle Hough transformation, 83
class-based n-grams, 181
classification, 29, 110
  class, 29
  concepts, 29
  garbage class, 30
  Hidden Markov Models, 37
  Markov chain, 37
  maximum likelihood, 34
  overfitting, 31
  phases, 29
  rule-based, 33
  supervised, 31
  training samples, 29
  unsupervised, 31
color histogram, 226, 241
color space
  RGB, 11
color structure descriptor (CSD), 227
command and control, 153
commanding actions, 325
commenting actions, 325
compactness, 25
conformity with expectations, 391
context of use, 5, 370
control input limiting, 384
control input management, 383
control input modifications, 383
controllability, 391
convergence, 266, 271
conversational maxims, 162

coordinate transformation, 251
cross talk cancellation, 274, 276
crossover model, 373, 376

DAMSL, 163
data entry management, 372, 391, 393
data glove, 336, 363
decoding, 147
Dempster Shafer theory, 388
design goals, 2
dialog context, 391
dialog design, 162
dialog management, 154
dialog system state, 390
dialog systems, 1
dilation, 237
discrete fourier transform (DFT), 210
display codes and formats, 392
distance sensors, 378
do what i mean, 394
driver modeling, 375
driver state, 374
dual hand manipulation, 338
dynamic grasp classification, 346
dynamic systems, 1

early feature fusion, 186
ECAM, 392
eigenfaces, 183, 185, 201
eigenspace, 78
emotion model, 176
emotion recognition, 175
emotional databases, 177
ensemble construction, 180
erosion, 237
error
  active handling, 172
  avoidance, 173
  identification, 393
  isolation, 393
  knowledge-based, 173
  management, 393
  multimodal, 172
  passive handling passive, 172
  processing, 173
  recognition, 173
  rulebased, 173
  skillbased, 172
  systemspecific, 173
  technical, 173
  tolerance, 391
  userspecific, 172
Euclidian distance, 78
event-driven architecture, 348
eye blinks, 375

face localization, 61
face recognition
  affecting factors, 194
  component-based, 206
  geometry based, 200
  global/holistic, 200
  hybrid, 200
  local feature based, 200
face tracking, 67
facial action coding system, 186
facial expression commands, 56
facial expression recognition, 184
facial expressions, 183
feature
  area, 23
  border length, 23
  center of gravity, 23
  classification, 29, 110
  compactness, 25
  derivative, 27
  eccentricity, 24
  extraction, 12, 105
  geometric, 22
  moments, 23
  normalization, 26, 109
  orientation, 25
  selection, 32
  stable, 22
  vector, 8
feature extraction, 178
feature groups, 122, 124
feature selection, 179
filter-based selection, 179
fine manipulation, 338
finite state model, 156
force execution gain, 383
formant, 178
frame-based dialog, 157
freedom of error, 2
function allocation, 385
functional link networks, 376
functionality, 390

functionality shaping, 392
fusion
  early signal, 168
  late semantic, 169
  multimodal integration/fusion, 168
fuzzy logic, 388

Gabor Wavelet Transformation, 83
garbage class, 30
Garbor-wavelet coefficients, 185
Gaussian mixture models, 241
gen locking, 268, 303
geometric feature, *see* feature, geometric
gesture recognition, 7
goal directed processing, 159
goal extraction, 342
grasp, 324
  classification, 344
  dynamic, 324, 344
  segmentation, 343
  static, 324, 343
  taxonomy, 348
  taxonomy of Cutkosky, 345
gray value image, 11
guiding sleeves, 294

Haar-features, 62
hand gesture commands, 7
hand localization, 14
handover strategies, 385
haptic alerts, 381
haptic interfaces, 279
head pose, 375
head pose estimation, 78
head related impulse response, 272
head related transfer function, 272
head-mounted display, 266, 361
head-up display, 380, 382
Hidden Markov Models, 37, 142
  Bakis topology, 39
  multidimensional observations, 46
  parameter estimation, 44
high inertia systems, 380
histogram
  object/background color, 15
  skin/non-skin color, 18
holistic approaches, 184
homogeneous coordinates, 251
homogeneous texture descriptor (HTD), 229
human activity, 322
  decomposition, 323
  direction, 322
  operator selection, 323
human advice, 354
human comments, 354
human transfer function, 374
human-computer interaction, 164
humanoid robot, 316

image
  gray value, 11
  intensity, 11
  mask, 11
  object probability, 16
  preprocessing, 102
  representation, 10
  skin probability, 19
immersion, 264
immersive display, 266
immersive interface, 361, 362
Impresario, 6
information management, 379, 391
input shaping, 384
integral image, 62
intensity panning, 273
intent recognition, 392
intentions, 370
inter-/intra-gesture variance, 22
interaction, 164
  active, 322
  passive, 322
  robot assistants, 321
interaction activity
  classification, 323
  commanding actions, 323
  commenting actions, 323
  performative actions, 323
interaction knowledge, 386
interactive learning, 328
  task learning, 330
interactive programming
  example, 354
interaural time difference, 272
intervention, 372
intrusiveness, 374

Java3D, 6

kernel density estimation, 242
keyword spotting, 180
kNN classifier, 205

language model, 148, 181
late semantic fusion, 186
latency, 263
LAURON, 361
learning
  interactive, 328
learning automaton, 350
learning through demonstration
  classification, 326
  sensors, 336
  skill, 326
  task, 327
level of detail rendering, 300
lexical predictions, 393
Lidstone coefficient, 182
limits of maneuverability, 381
line-of-sight, 375
linear regression tracking, 245
linguistic decoder, 148
local regression, 377

magic wand, 363
magnetic tracker, 336
man machine interaction, 1
man-machine dialogs, 386
man-machine task allocation, 385
manipulation tasks, 338
  classes, 338
  definition, 338
  device handling, 339
  tool handling, 339
  transport operations, 339
manual control assistance, 379
mapping tasks, 350
  movements, 353
  coupling fingers, 351
  grasps, 351
  power grasps, 352
  precision grasps, 352
Markov chain, 37
Markov decision process, 158
maximum a-posteriori, 182
maximum likelihood classification, 34, 85
maximum likelihood estimation, 181
mean shift algorithm, 244
mental model, 386
menu hierarchy, 386
menu options, 386
MFCC, 178
mobile robot, 316
modality, 166
morphological operations, 236
motion parallax, 266
motor function, 165
moving platform vibrations, 384
multimodal interaction, 393
multimodality, 4, 166
multiple hypotheses tracking, 107
mutlimodal interaction, 164

n-grams, 181
natural frequency, 381
natural language generation, 155
natural language understanding, 154
nearest neighbor classifier, 205
noise canceling, 384
normalization
  features, 26, 109
notation system, 118
  movement-hold-model, 120

object probability image, 16
observation, 29
oculomotor factors, 266
ODETE, 363
on/off automation, 385
operator fatigue, 375
oriented gaussian derivatives (OGD), 228
overfitting, 31
overlap resolution, 106

Parallel Hidden Markov Models, 122
  channel, 122
  confluent states, 123
parametric models, 376
particle dynamics, 286
passive stereo, 268
PbD process, 333
  abstraction, 333
  execution, 334
  interpretation, 333
  mapping, 334
  observation, 333
  segmentation, 333

simulation, 334
system structure, 335
perceptibility augmentation, 380
perception, 165
performative actions, 323
person recognition
authentication, 191
identification, 191
verification, 191
phrase spotting, 182
physically based modeling, 286
physiological data, 371
pinhole camera model, 251
pitch, 178
pixel adjacency, 20
plan based classification, 389
plan library, 388, 391
plan recognition, 388
predictive information, 380
predictive scanning, 394
predictive text, 394
predictor display, 381
predictor transfer function, 380
preferences, 370
principal component analysis, 179
principle component analysis (PCA), 204
prioritizing, 392
programming
explicit, 358
implicit, 358
programming by demonstration, 330
programming robots via observation, 329
prompting, 392
prosodic features, 178

question and answer, 153

Radial Basis Function (RBF) network, 345
rational conversational agents, 160
recognition
faces, 193
gestures, 7
person, 224
sign language, 99
recording conditions
laboratory, 9
real world, 9
redundant coding, 382
region
description, 20
geometric features, 22
responsive workbench, 271
RGB color space, 11
rigid body dynamics, 288
robot assistants, 315
interaction, 321
learning and teaching, 325
room-mounted display, 267

safety of use, 2
scene graph, 298
scripting, 163
segmentation, 16, 234
threshold, 16
self-descriptiveness, 391
sense, 165
sensitive polygons, 296
sensors for demonstration, 336
data gloves, 336
force sensors, 337
magnetic tracker, 336
motion capturing systems, 336
training center, 337
sequential floating search method, 179
shape model, 186
shutter glasses, 267
sign language recognition, 5, 99
sign-lexicon, 117
signal
transfer latency, 358
signal processing, 343
simplex, 59
single wheel braking, 385
skill
elementary, 331
high-level, 331
innate, 327
learning, 326
skin probability image, 19
skin/non-skin color histogram, 18
slot filling, 157
snap-in mechanism, 297
soft decision fusion, 169, 187
sparse data handling, 158
spatial vision, 265
speech commands, 382
speech communication, 141
speech dialogs, 153

speech recognition, 4
speech synthesis, 155
spoken language dialog systems, 153
static grasp classification, 344
steering gain, 383
stemming, 181
stereopsis, 265
stick shakers, 381
stochastic dialog modeling, 158
stochastic language modeling, 127
  bigramm-models, 127
  sign-groups, 128
stopping, 180
subgoal extraction, 342
subunits, 116, 117
  classification, 125
  enlargement of vocabulary size, 133
  estimation of model parameters, 131
  training, 128
suitability for individualization, 391
supervised classification, 31
supervised training, 376
supervisory control, 1
support of dialog tasks, 390
Support Vector Machines (SVM), 346

tagging, 163
takeover by automation, 372
task
  abstraction level, 331
  example methodology, 332
  goal, 328
  hierarchical representation, 340
  internal representation, 332
  learning, 327
  mapping, 333
  mapping and execution, 348
  model, 328
  subgoal, 328
task learning
  classification, 331
taxonomy
  multimodal, 167
telepresence, 356
telerobotics, 356
  concept, 356
  controlling robot assistants, 362
  error handling, 360
  exception handling, 360
  expert programming device, 360
  input devices, 359
  local reaction, 359
  output devices, 359
  planning, 359
  prediction, 360
  system components, 359
template matching, 209
temporal texture templates, 240
texture features
  homogeneous texture descriptor (HTD), 229
  oriented gaussian derivatives (OGD), 228
threshold
  automatic computation, 17
  segmentation, 16
tiled display, 271
timing, 392
tracking, 238
  color-based tracking, 246
  combined tracking detection, 243
  hand, 107
  hypotheses, 107
  multiple hypotheses, 107
  shape-based tracking, 244
  state space, 107
  tracking following detection, 238
  tracking on the ground plane, 250
tracking system, 268
traction control, 384
traffic telematics, 378
training, 144
training center, 337
training phase, 29
training samples, 29
trajectory segmentation, 343
transcription, 117, 118
  initial transcription, 129
  linguistic-orientated, 118
  visually-orientated, 121
transparency, 391
Trellis diagram, 146
trellis diagram, 42
tremor, 384
tunnel vision, 373
tutoring, 392

unsupervised classification, 31
usability, 2, 163, 390

user activities, 390
user assistance, 5, 370
user expectations, 371
user identity, 370
user observation, 371
user prompting, 371
user state, 370
user state identification, 388
user whereabouts, 390

variance
  inter-/intra-gesture, 22
vehicle state, 377
viewer centered projection, 268
virtual magnetism, 296
virtual reality, 4, 263
vision systems, 378
visual commands, 382
visual interfaces, 265
visualization robot intention, 362
Viterbi algorithm, 42, 146
voice quality features, 178
VoiceXML, 163
VR toolkits, 301

warning, 393
wave field synthesis, 274
wearable computing, 390, 394
whereabouts, 370
window function, 178
working memory, 386
workload assessment, 371
wrapper-based selection, 179

Karl-Friedrich Kraiss received the Diploma in Communications Engineering and the Doctorate in Mechanical Engineering from Berlin Technical University in 1966, and 1970 respectively. In 1987 he earned the Habilitation for Aerospace Human Factors Engineering from RWTH Aachen University. In 1988 he received the research award "Technical Communication" from the Alcatel/SEL foundation for research in man-machine communication. Since 1992 he has been a full Professor at RWTH Aachen University, where he holds the Chair for Technical Computer Science in the Faculty of Electrical Engineering and Information Technology. His research interests include man-machine interaction, artificial intelligence, computer vision, assistive technology, and mobile robotics.