Olena: a Developer's Handbook

Last updated 7 August 2003, for Olena 0.9

Raphaël Poss

1 Abstract

This is the ninth public release of Olena, a generic image processing library in C++.

Olena is a project developed by the EPITA Research and Development Laboratory (`http://www.lrde.epita.fr`) since 1997. We did numerous prototypes and throw-away experiments before settling into the kind of programming paradigm which is finally here.

This version of Olena is quite young. In fact most of the operators have been translated from the sources of Milena, an image processing library which works only on 2D images (Milena stands for "mini-Olena") and which is not publicly available.

We haven't written a real documentation yet. In the '`doc/`' directory you will find the start of a reference manual which presently documents only the morphological operators, together with a developer's guide to the use of Olena from user projects. In the '`doc/demo/`' directory lie a few sample programs. The file '`doc/demo/tour.cc`' attempts to introduce you to the basics of Olena. For the rest, we're afraid you will have to dig the code or e-mail us.

Please direct any question or comments to `olena@lrde.epita.fr`, or `olena-bugs@lrde.epita.fr`.

Olena also has a web page, located at `http://www.lrde.epita.fr/cgi-bin/twiki/view/Projects/Olena`.

2 Olena fast installation guide

2.1 Required software

Here is a non-exhaustive list of required software required to build Olena successfully.

- to compile the user tools:
  - a POSIX shell, like Bash
  - a decent C++ compiler, like GNU C++
  - a `make` utility, like GNU or BSD `make`
- to compile the documentation:
  - a LaTeX distribution
  - the ‘`listings`’ TEX package
  - the utility `convert` from ImageMagick
  - GNU Autogen
  - `hevea`, a TEX to HTML conversion tool
  - the `texinfo` utilities from GNU
- to develop *in* Olena:
  - GNU Autotools (Autoconf 2.54, Automake 1.7)

2.2 Configuration

In order to prepare the build process, you need to configure the source tree.

Assuming your Olena distribution is uncompressed in directory ‘`oln-0.9`’, follow these steps:

```
$ mkdir ../build
$ cd ../build && ../oln-0.9/configure CXXFLAGS=''
```

Note: take care to set `CXXFLAGS` always when running `configure`, for the default value computed by `configure` may yield to compilation issues (conflicts between optimization and debugging).

The build process can be altered by a number of options you can pass to the `configure` script. The following sections describe them.

Additionally, if you are an Olena maintainer (a person who runs `make distcheck`), *prefer setting* `CXXFLAGS` *as an environment variable*: the flags given on the commandline to `configure` are not propagated to recursive runs by `make distcheck`.

2.2.1 Installation path

By default, Olena is installed in the standard “local” directory of your system. This is usually ‘`/usr/local`’ under Unix.

You can change this path with the following flag:

```
--prefix=<installation prefix>
```

2.2.2 Compiler selection and compilation flags

By default, `configure` will try to use the first C++ compiler it encounters on your system. If `CXX` is not set, it will look, in order, for:

- the value of the `CCC` environment variable,
- the GNU C++ compiler (`g++`),
- the `c++` or `gpp` commands on your system,
- `aCC`, the HP-UX standard C++ compiler,
- the `CC`, `cxx`, `cc++` or `cl` commands on your system,
- KAI's C++ compiler (`KCC`),
- `RCC`, `xlC_r` or `xlC`.

You can override the detection system by passing your favourite compiler name to `configure`, as follows:

```
$ .../configure CXX=<your-favorite-C++-compiler>
```

As an alternative, you can also set the environment variable '`CXX`'.

For most compilers, `configure` will select debugging and minimal optimization ('`-g -O2`' with g++), which is wrong. You should override the default C++ flags by giving `configure` your selection of flags:

```
$ .../configure CXXFLAGS="<your-favorite-flags>"
```

This is especially useful to solve a number of problems, described in the following section.

2.3 Using `CXXFLAGS` to solve compilation problems

2.3.1 Olena needs C99

While Olena is written in C++, it uses a number of features (math functions) from ISO C99. However most C++ compilers do not enable these features by default. If your compilation fails with (e.g.) undefined references to `roundf`, but you know what flags can activate these functions, add them to `CXXFLAGS`.

In case your system does not provide some math functions necessary for Olena, you can force the use of a local, overloaded, implementation, by using macros of the form '`-DOLN_NEED_xxx`', where '`xxx`' stands for the name of the missing function, in uppercase. For example, on Darwin (MacOS X), the flag '`-DOLN_NEED_SQRTF`' is needed (but `configure` should add it anyway).

2.3.2 Olena needs deep template recursion

The C++ design patterns used in Olena use deep template nesting and recursion. However, the C++ standard specifies that C++ compiler need only handle template recursion upto 19 levels, which is insufficient for Olena. This is a problem for GCC 2.95 and probably other compilers.

Hopefully, `configure` tries to fix this automatically by adding '`-ftemplate-depth-NN`' when necessary, but other compilers than GCC may need other flags. If you know these flags, add them to `CXXFLAGS`.

2.3.3 Debugging flags make Olena slow

Because Olena depends on C++ optimizations to provide the best performance, and enabling debugging flags often disable optimizations, you are advised to override the `CXXFLAGS_OPTIMIZE` with any options that gives the best optimization/conformance trade-off. However, note that passing '`-DNDEBUG`' disable many sanity checks, while providing only a poor performance improvement.

2.3.4 Speeding up the compilation

When using GCC, by default separate phases of the compilation of each file are run sequentially (compilation then assembly). Using '`-pipe`' in `CXXFLAGS` allows GCC to fork processes and run compilation phases in parallel, which brings a compilation speedup on multiprocessor machines or machines with slow storage access (when using '`-pipe`', no intermediary data is saved).

2.4 Speeding up the configuration process

`configure` can manage a cache of autodetected features and values. This cache speeds up `configure` runs and can be activated with the '`-C`' option.

NOTE: the effects of many of the flags passed to `configure` are stored in the cache. If you decide to re-run `configure` with other flags, delete the '`config.cache`' file first.

2.5 Optional Features

2.5.1 Using external libraries

Several parts of Olena can make use of the Zlib compression library (in Olena I/O) and the FFTW fast Fourier transforms library (in Olena fft transforms).

By default, `configure` will try to autodetect their presence. However, if your version of any of these libraries is located in a non-standard path, you should specify it as follows:

```
--with-fftw=<path-to-libfftw>
--with-zlib=<path-to-zlib>
```

Additionally, if for a reason or another you need to prevent Olena from using any of these libraries, you can disable their use with the following flags:

```
--without-fftw
--without-zlib
```

2.5.2 Elidable components

Several build targets can be disabled, in case you are only interested in "parts" of the full Olena distribution.

The elidable parts are so-called *components*, and you can obtain a list of them by running:

```
$ .../configure --help
```

2.6 Building

Once your build directory is `configure`d, you can run

```
$ make
```

to recursively build all selected components.

Additionnally, you can build and run the testsuite and demonstration programs with:

```
$ make check
```

However, this process is very time- and memory- consuming. It takes up to 25mn and 250-300Mb of virtual memory on a Debian GNU/Linux 2.54GHz bi-Xeon machine.

2.7 Compiler notes

Olena has been tested on the following configurations :

System	**Compiler**
Linux	g++ 3.0 and 3.2
Linux	icc (Intel's C++ Compiler) v7
MacOS X	g++ 3.1
NetBSD 1.6	g++ 3.2
Cygwin	g++ 3.2

Olena used to be compatible with g++ 2.95 for performance reasons. With g++ 3.2, this constraint is becoming obsolete. Moreover, it has many annoying issues, here are the two more important ones:

- g++ 2.95 rejects valid expressions, often implying ugly workarounds;
- under various circumstances, optimizations sometimes generates invalid code, especially with intensive inlining.

Actually Olena yet compiles with g++ 2.95, but some wrong code might be generated with data types.

Compilation time may have important differences between compilers, the following benchmark gives an idea of the time needed to complete a `make check`. The tests have been run on a Bi-Xeon 2.4Ghz machine.

Compiler	**Time**
g++-2.95	16m42s
g++-3.0	23m20s
g++-3.2	20m03s
icc-7	12m52s

These tests include compilation and running time, the following ones just show the runtime benchmarks for the '`extrkiller`' test:

Compiler	**Options**	**Time**
g++-2.95	'`-O3 -finline-limit-1500`'	3m14s
g++-3.0	'`-O3 -finline-limit-1500`'	2m08s
g++-3.2	'`-O3 -finline-limit-1500`'	1m50s
icc-7	'`-O3`'	5m41s

2.8 Installing

To install the Olena headers, command-lines utilities and additional files on your system, run:

```
$ make install
```

from the build directory.

If not overriden with '`--prefix`' (see Section 2.6 [Building], page 5) , this will install:

- the headers in '`/usr/local/include/oln`',
- the utilities in '`/usr/local/bin`',
- sample images in '`/usr/local/share/oln`',
- the Autoconf helper '`oln.m4`' in '`/usr/local/share/aclocal`'.

You can later remove Olena from your system by running

```
$ make uninstall
```

from the build directory.

3 Upgrading from older versions

3.1 Upgrading from 0.8 to 0.9

Abstract interfaces are available and dispatch the methods to their implementations. Thus, it is not necessary to downcast abstract-typed variables through "Exact_ref", "Exact_cref", "Exact_ptr" or "Exact_cptr" macros anymore.

Abstract classes have moved from "oln" to "oln::abstract" namespace.

- oln::image => oln::abstract::image
- oln::iter => oln::abstract::iter
- oln::neighborhood => oln::abstract::neighborhood
- oln::point => oln::abstract::point
- oln::struct_elt => oln::abstract::struct_elt
- oln::w_window => oln::abstract::w_window
- oln::window => oln::abstract::window

The "data" concept changed into the "implementation" one. The "implementation" can be simple data storage, but it will also provide special proxies and function-generated images. The access to this "implementation" has changed from "data()" to "impl()" member function.

Functions modifying borders were made image member functions.

- border::set_width() => abstract::image::border_set_width()
- border::adapt_width() => abstract::image::border_adapt_width()
- border::adapt_copy() => abstract::image::border_adapt_copy()
- border::adapt_mirror() => abstract::image::border_adapt_mirror()
- border::adapt_assign() => abstract::image::border_adapt_assign()

Olena now has its own coding style (http://www.lrde.epita.fr/cgi-bin/twiki/view/Projects/OlenaCodingStyle).

In particular, macros have been renamed:

- Exact(I) => mlc_exact_type(I)
- Point(I) => oln_point_type(I)
- Iter(I) => oln_iter_type(I)

Typedefs have been renamed too:

- image2d<bin>::iter => image2d<bin>::iter_type
- image2d<bin>::point => image2d<bin>::point_type

More details can be found in the coding style web page.

3.2 Upgrading from 0.7 to 0.8

Olena has been massively reorganized between versions 0.7 and 0.8. The idea was to split the library into three distinct components:

- Image processing
- Basic data types
- Meta programming tools

Thus, two additional directories and namespaces have been created:

- ‘`mlc/`’ directory and `mlc` namespace for meta-programming tools
- ‘`ntg/`’ directory and `ntg` namespace for data types

Here is the list of general renaming rules for header files:

- ‘`<oln/types/*.hh>`’ => ‘`<ntg/*.hh>`’
- ‘`<oln/meta/*.hh>`’ => ‘`<mlc/*.hh>`’
- ‘`<oln/core/type.hh>`’ => ‘`<mlc/type.hh>`’
- ‘`<oln/core/contract.hh>`’ => ‘`<mlc/contract.hh>`’

Namespaces changes can be deduced from files renaming. Indeed, each file moved into ‘`mlc/`’ has seen its member moved into the `mlc` namespace. The same rule stands for `ntg`.

4 The Olena configuration system

Olena was developed in pure C++ (it does not depend on nonstandard libraries), and a large amount of work was done so that it can be compiled with any ISO C++ compliant compiler. Therefore, if all things were perfect, Olena headers could be used as-is, with no configuration required.

However, two facts darken the picture:

1. Olena uses few but some C99 functions, while ISO C++ was standardized in 1998.
2. C++ compilers are not born equal, and few of them are really ISO C++ compliant.

As a consequence, to ensure that Olena works properly, several known compiler and language “bugs” or “misfeatures” must be checked, in order to enable workarounds. These checks and the available workarounds are described in the following sections.

The reason why the workarounds are not all enabled by default, so that checks would be unnecessary, is that they are inelegant and might break some compiler optimizations on systems where they are unneeded.

4.1 Known and handled issues

4.1.1 Template recursion support

The ISO C++ standard specifies that compliant compiler must support a recursion depth of *at least* 17 levels. Some code pieces in Olena need at least 50. Most compilers happen to support recursion upto many more levels, however it is not guaranteed.

In particular, GCC 2.95 is known to need the flag ‘`-ftemplate-depth-xxx`’ to support extra levels of recursion.

Check and workaround in ‘`oln.m4`’

The provided Autoconf macro `AC_CXX_TEMPLATE_DEPTH` takes an optional numeric argument N (default value 50) and works as follows:

1. attempt to compile a program using recursion depth N;
2. if it works, stop the check, no workaround required.
3. add ‘`-ftemplate-depth-N`’ to `CXXFLAGS`, and try again;
4. if it works, add ‘`-ftemplate-depth-N`’ to `CXXFLAGS`, then stop.
5. if it does not work, warn the user.

Rationale for the check:

‘`-ftemplate-depth`’ is not supported by newer versions of GCC and probably other compilers, and thus cannot be added to `CXXFLAGS` always.

On the other hand, several compilers, if not GNU, support GCC options : exotic yet unknown compilers might have the same problem as GCC 2.95 and require the same option.

Tested configurations

Among ICC, Comeau C++, GCC 3.x and GCC 2.95, none but the latter need extra flags to support deep template recursion. For GCC 2.95, ‘`-ftemplate-depth-N`’, with N sufficiently large, fixes the problem.

4.1.2 Numeric limits

ISO C++ specifies that the standard library must provide the class template `std::numeric_limits` and its specializations in header '`limits`'. Olena uses this class to retrieve infinity values for the C++ types `float` and `double`. However, it is not available in all implementations of the C++ standard library.

A substitute is known: the C89 constant `HUGE_VAL` and C99 `HUGE_VALF`, defined in '`cmath`'. However, they are not satisfying because they are do not really represent infinity.

Therefore, the Olena header '`oln/config/math.hh`' works as follows:

1. if the macro `USE_C_LIMITS` is not defined, use `std::numeric_limits`.
2. if the macro `USE_C_LIMITS` is defined, then:
   a. include '`cmath`';
   b. if `HUGE_VAL` is not defined, abort with an error ("Cannot define infinity in this configuration").
   c. if it is, use it as the infinity value for type `double`;
   d. if `HUGE_VALF` is defined, use it as the infinity value for type `float`;
   e. if it is not, use `HUGE_VAL` casted to `float` instead.

Check and workaround in '`oln.m4`'

The provided Autoconf macro `AC_CXX_NUMERIC_LIMITS` works as follows:

1. attemt to compile a program using `std::numeric_limits`;
2. if it works, do nothing.
3. if it does not, add '`-DUSE_C_LIMITS`' to `CPPFLAGS`.

4.1.3 C math functions

Olena uses functions from the C89 and C99 math libraries. However, most C++ environments only know about C89 math functions, since the C++ standard predates C99. It noticeably happens, on several known architectures, that some C99 functions are not available directly, or indirectly, from C++ code.

A kludge is known, and several workarounds are available:

- When using the GNU C library and headers on a GNU system, it is sufficient to define the `_ISOC99_SOURCE` macro to make C99 math available from C++.
- Replacements for (as of 0.7) `sqrtf`, `floorf`, `round` or `roundf` can be enabled by defining macros of the form `NEED_xxx`, where xxx is the function name.

Check and workaround in '`oln.m4`'

The provided Autoconf macro `OLN_FLOAT_MATH` invokes `AC_CXX_CHECK_MATH` successively for `sqrtf`, `floorf`, `round` and `roundf`.

`AC_CXX_CHECK_MATH` takes the name of the function to test and works as follows:

1. try to compile and link a program using the function;
2. if it works, do nothing.
3. else, try again to compile the program with '`-D_ISOC99_SOURCE=1`';
4. if it works, add '`-D_ISOC99_SOURCE=1`' to `CPPFLAGS`.
5. if it does not work, add '`-DNEED_function`' to `CPPFLAGS`.

4.1.4 Using the FFTW library

The implementation of the FFT transform in Olena requires the FFTW library (`http://www.fftw.org/`). Because this library might be unavailable, it is only used if the macro `HAVE_FFTW` is defined to nonzero, and the correct include path is given to the compiler.

Check in '`oln.m4`'

The provided Autoconf macro `AC_WITH_CXX_FFTW` works as follows:

1. if the user didn't provide the flag '`--with-fftw`', do nothing.
2. if the user provided a prefix directory with '`--with-fftw=dir`', add '`-Idir`' and '`-Ldir`' to `FFTW_CXXFLAGS` and `FFTW_LDFLAGS`, resp.
3. attempt to compile a program that uses a function from the FFTW library, using the C++ compiler with `FFTW_CXXFLAGS` and `FFTW_LDFLAGS`;
4. if it works, `AC_DEFINE` `HAVE_FFTW` to 1.

Rationale for using the C++ compiler (instead of the C compiler): the FFTW library is a C library and there are systems where C++ programs cannot link with any C library without options. This ckeck ensures that faulty link configurations fail early.

4.1.5 Using the Zlib library

The implementation of the I/O operators in Olena can make use of the Zlib library fo save or load images from gzipped files. Because this library might be unavailable, it is only used if the macro `HAVE_ZLIB` is defined to nonzero, and the correct include path is given to the compiler.

Check in '`oln.m4`'

The provided Autoconf macro `AC_WITH_CXX_ZLIB` works as follows:

1. if the user didn't provide the flag '`--with-zlib`', do nothing.
2. if the user provided a prefix directory with '`--with-zlib=dir`', add '`-Idir`' and '`-Ldir`' to `ZLIB_CXXFLAGS` and `ZLIB_LDFLAGS`, resp.
3. attempt to compile a program that uses a function from the Zlib library, using the C++ compiler with `ZLIB_CXXFLAGS` and `ZLIB_LDFLAGS`;
4. if it works, `AC_DEFINE` `HAVE_ZLIB` to 1.

Rationale for using the C++ compiler: See Section 4.1.4 [Using the FFTW library], page 11.

4.1.6 Using exceptions

Olena code self-checks using preconditions and postconditions, in addition to static checks pertaining to the type system. By default, the C/C++ function `assert` is used for these checks.

However, failure in a condition checked by `assert` causes the program to abort, with no possible error recovery. When using Olena from a dynamic, interpreted language where the user is likely to call Olena functions with incorrect arguments, this "feature" becomes a nuisance.

For this purpose, when the `OLN_EXCEPTIONS` macro is defined, exceptions are thrown instead. However, this option cannot be used if the compiler does not support proper exception handling.

Checks in ‘`oln.m4`’

The provided Autoconf macro `OLN_ENABLE_EXCEPTIONS` takes an optional boolean argument (default value yes) and works as follows:

1. if the user does not give the ‘`--enable-oln-exceptions`’ flag to `configure`, *and* the argument to `OLN_ENABLE_EXCEPTIONS` is set to “no”, do nothing.
2. check for the availability of exceptions with `AC_CXX_EXCEPTIONS` (described below);
3. if exceptions are available, add ‘`-DOLN_EXCEPTIONS`’ to `CPPFLAGS`.

The provided Autoconf macro `AC_CXX_EXCEPTIONS` works as follows:

1. try to compile a program that throws and catches an exception;
2. if it does not compile, fail the test.

4.2 Important variables

Programs using Olena with the provided ‘`oln.m4`’ have to take the following ‘`Makefile`’ variables into consideration:

`CPPFLAGS`	C++ preprocessor flags specific to Olena. See Section 4.2.1 [Values for CPPFLAGS], page 12.
`CXXFLAGS`	C++ compiler flags specific to Olena. See Section 4.2.2 [Values for CXXFLAGS], page 13.
`FFTW_CXXFLAGS`	C++ compiler flags to use the FFTW library. See Section 4.1.4 [Using the FFTW library], page 11.
`FFTW_LDFLAGS`	C++ linker flags to use the FFTW library. See Section 4.1.4 [Using the FFTW library], page 11.
`ZLIB_CXXFLAGS`	C++ compiler flags to use the Zlib library. See Section 4.1.5 [Using the Zlib library], page 11.
`ZLIB_LDFLAGS`	C++ linker flags to user the Zlib library. See Section 4.1.5 [Using the Zlib library], page 11.

4.2.1 Values for `CPPFLAGS`

‘`-DUSE_C_LIMITS`’	See Section 4.1.2 [Numeric limits], page 10.
‘`-DHAVE_FFTW=1`’	See Section 4.1.4 [Using the FFTW library], page 11.
‘`-DHAVE_ZLIB=1`’	See Section 4.1.5 [Using the Zlib library], page 11.
‘`-DOLN_EXCEPTIONS`’	See Section 4.1.6 [Using exceptions], page 11.

4.2.2 Values for `CXXFLAGS`

‘`-ftemplate-depth`’
See Section 4.1.1 [Template recursion support], page 9.

5 Using Olena from another project

6 The Olena source tree

The Olena source tree is divided into several distrinct components:

‘`top source directory`’

The base directory for Olena sources. It contains Autoconf/Automake definitions that allow to run, recursively, the following toplevel operations:

- creating initial configuration files and command-line utilities (`make all`);
- running the testsuite and building the demonstration programs (`make check`);
- installing Olena to the system (`make install`);

‘`olena/`’ Image processing sources and testsuite.

‘`integre/`’

Data types sources and testsuite.

‘`metalic/`’

Meta programming tools and testsuite.

‘`doc/`’ The documentation and demonstration programs.

‘`tools/`’ The user programs. This directory and its sub-directories contain auto-generated sources that yield a set of user programs and commands exhibing several Olena features.

You can find in the following sections a more detailed description of the contents of each directory.

The generation of `configure` from ‘`configure.ac`’ is led by the toplevel script `bootstrap.sh`.

6.1 Image processing library files: ‘`olena/`’

This directory contains the main Olena sources, the testsuite and some additional programs.

Here are the subdirectories:

6.1.1 Olena headers: ‘`olena/oln/`’

This directory contains the Olena library strictly speaking, that is, the C++ header files.

‘`oln/config/`’

Olena global configuration definitions, reachable by including ‘`oln/config/system.hh`’. This directory also provides replacements for missing math functions in ‘`math.hh`’.

‘`oln/core/`’

Definitions for image types and various other Olena data types. This directory contains definitions for:

- image types;
- structural element types (windows, neighborhoods);
- iterators;
- points;
- borders.

‘`oln/transforms/`’	Transformation operators over images. Includes Fast Fourier Transforms (FFT) and Discreet Wavelets Transforms (DWT).
‘`oln/morpho/`’	Morphological operators.
‘`oln/level/`’	Level processing operators.
‘`oln/convol/`’	Convolution operators.
‘`oln/arith/`’	Arithmetical operators (over images). Covers both arithmetical, conversion and logical operators.
‘`oln/convert/`’	Value types conversion functions.
‘`oln/io/`’	Input/Output operators for several Olena data types.
‘`oln/utils/`’	Utility operators.
‘`oln/math/`’	Utility mathematical functions.

In addition to these categories, four multi-purpose headers are provided in ‘`oln/`’:

‘`basics.hh`’	Recursively includes all *base types* definitions from ‘`oln/core/`’.
‘`basics1d.hh`’	Recursively includes all definitions from ‘`oln/core/`’ that allow handling of 1D images.
‘`basics2d.hh`’	Likewise, for 2D images.
‘`basics3d.hh`’	Likewise, for 3D images.

6.1.2 Testsuite files: ‘`olena/tests/`’

This directory contains most of the Olena testsuite. It contains one directory per test category, in addition to a library directory.

The directories are:

‘`arith/`’	Tests pertaining to types arithmetics.
‘`convert/`’	Tests pertaining to image value conversions (color-color, color-b/w, etc...).
‘`convol/`’	Tests pertaining to convolution operators.
‘`io/`’	Tests pertaining to image I/O.
‘`morpho/`’	Tests pertaining to morphological operators.
‘`sanity/`’	Tests that check that each Olena header can be separately included in C++ programs.

‘`transforms/`’	Tests pertaining to image transformations (FFT, DWT, ...).
‘`check/`’	Library containing several utilities used multiple times in other test directories.

6.2 Data types library files: ‘`integre/`’

In ‘`integre/`’ can be found everything related to basic data types.

6.3 Meta programming library files: ‘`metalic/`’

In ‘`metalic/`’ can be found all the meta programming tools used by both olena and intgre.

6.3.1 Autoconf helpers: ‘`config/`’

In ‘`config/`’ can be found several files automatically generated by the Autoconf command `autoreconf` (with the exception of ‘`oln.m4`’ and ‘`oln-local.m4`’ presented separately).

‘`depcomp`’	Compute dependencies from files.
‘`install-sh`’	Installs a file to its final location.
‘`missing`’	Presents the user with an intelligible error message if a tool is missing to the build process.
‘`mkinstalldirs`’	Creates the installation directories.
‘`mdate-sh`’	Computes the last modification date from a file (used in ‘`doc/dev/`’ to create ‘`version.texi`’).
‘`texinfo.tex`’	Texinfo definitions for the documentation.
‘`oln.m4`’	M4 file containing general-use macro definitions for use by the Olena distribution and user projects.
‘`oln-local.m4`’	M4 file containing macro definitions for the ‘`configure.ac`’ included in the distribution of Olena.

6.3.2 User configuration tools: ‘`olena/conf/`’

This directory contains the files used to create the utility scripts of the form `oln-config.sh`, which retain compiler-specific flags for later invocation by Olena users.

The files are:

‘`oln-config.shin`’	Template script used by the accompanying `configure` to generate the final utilities.
‘`gen-scripts.sh`’	A script that calls `configure` repeatedly to generate various versions of `oln-config.sh`.

‘`compilers.def`’
: Compiler list for use by `gen-scripts.sh`.

‘`configure.ac`’
: Lightweight Autoconf source file, leading to the utility `configure` used by `gen-scripts.sh`.

The creation of `configure` from ‘`configure.ac`’ in this directory is led by the toplevel `bootstrap.sh`.

6.4 User tools source tree: ‘`tools/`’

In this directory are stored the sources for run-time, user-level utilities.

The subdirectories are:

‘`utilities/`’
: Automatically-generated sources for commandline utilities. Generated programs allow the use of Olena functions from shell scripts.

‘`swilena/`’
: SWIG (`http://www.swig.org/`) wrappers for Olena, to allow the use of Olena functions from scripting languages like Python and Perl. This is *EXPERIMENTAL* work.

6.5 Documentation source tree: ‘`doc/`’

This directory contains all files needed to build the documentation, except headers files from ‘`oln/`’, which contain comments used in the documentation build process.

Here is a list of the most important files:

‘`doc/dev/`’
: A directory containing Texinfo sources for the Olena Developer’s Handbook.

‘`doc/ref/`’
: A directory containing TeX sources and definitions to build the Reference Manual. It noticeably contains:

  ‘`oln-ref.tex`’
  : The master TeX file for the Reference Manual.

  ‘`ref-types.tex`’
  : Handwritten documentation about Olena value types, included in the Reference Manual.

  ‘`ref-morpho.tex`’
  ‘`ref-level.tex`’
  : TeX sources describing Olena components. They are autogenerated by AutoGen from Olena C++ header files[1], using definitions in ‘`processing.tpl`’.

  ‘`Makefile.am`’
  : Automake definitions that control the build process, which (as of 0.7) depends on GNU Make.

---

[1] more precisely, from C++ comments

'`bin/`'	Auto-generated programs that create the pictures included in the Reference Manual.
'`html/`'	The HTML version of the Reference Manual.
'`processing.tpl`'	AutoGen parameters for generating parts of the Reference Manual.

Running `make all` in the '`doc/`' toplevel subdirectory generates the Reference Manual and the Developer's Handbook. To achieve this goal, it uses the Olena headers it can find in '`../olena`' and the Texinfo source '`../olena/config/texinfo.tex`'.

7 Frequently Asked Questions

7.1 Cleaning up the source tree

Question: my source tree behaves strangely.

Answer: make sure you have many development tools installed, and then run:

```
$ make maintainer-clean
$ ./bootstrap.sh
```

(from the toplevel source directory)

This will clear anything that can be regenerated back, and re-generate the project control files (`autoreconf`)

7.2 Missing tools

Question: I do not want to involve the documentation in my build process (it takes too long and/or I do not have the tools to build it). How can I disable it ?

Answer 1: run the toplevel configure with the ‘`--without-doc`’ option.

Answer 2: if you do not want to build the reference manuals but still compile the demonstraction programs and developer’s info files, use ‘`--without-doc-ref`’.

7.3 Using Olena

Question: How can I use Olena in my projects ?

Answer 1: add ‘`-I<path_to_installed_headers>`’ to your compile flags and it *should* work. In practice, of course, it does not. Proceed with the following answers.

Answer 2: use Autoconf and the provided ‘`oln.m4`’. Several macros can be used:

`AC_WITH_OLN`	Checks compiler features and AC_SUBST the variables `OLN_CPPFLAGS` and `CXXFLAGS`.
`OLN_ENABLE_EXCEPTIONS`	Enable the raise of C++ exceptions instead of aborting on errors. This breaks some optimizations, so do not use unless required. Updates `CPPFLAGS`.

See the file ‘`configure.ac`’ for an example invocation of these macros.

Answer 3: use the generated `oln-config-xxx.sh`, substituting ‘`xxx`’ with your favourite compiler. This script dumps to its standard output the flags necessary to build programs that use Olena successfully with the corresponding compiler. Use the ‘`--help`’ flag to see what data is available.

7.4 Troubleshooting

Missing functions at link-time

Problem My program compiles successfully, but refuses to link: the linker complains about missing `_roundf`.

Explanation
Your standard library headers declare `roundf` but it is not actually defined.

Solution Add ‘`-DNEED_ROUNDF`’ to your `CXXFLAGS`.

Incorrect behavior of generated code

Problem My programs compiles and runs, but either the compiler (GCC) issues warnings at compile-time in the Olena headers, or the results are weird and/or inaccurate.

Explanation 1
You are using GCC 2.95 and heavy optimisation (‘`-O3`’) flags. This is known to produce invalid code with Olena.

Solution 1 Use ‘`-O2`’ instead.

Explanation 2
You used the default, invalid, value for `CXXFLAGS` when ‘configure’ has run, and the sources were compiled using heavy optimization and debug settings, which is inconsistent.

Solution 2 Run `configure CXXFLAGS=''`. See the file ‘`BUILD`’ at the toplevel source directory.

Wrong include path

Problem My source file includes ‘`basics2d.hh`’ but compilation fails: the compiler complains about missing ‘`oln/config/system.hh`’.

Solution Include ‘`oln/basics2d.hh`’ instead, and use ‘`-I/usr/local/include`’ instead of ‘`-I/usr/local/include/oln`’ in your compilation flags.

Errors defining Infinity

Problem Compilation fails at points where `OLN_FLOAT_INFINITY` or `std::numeric_limits` is used.

Explanation
Your C++ standard library is broken.

Solution As a workaround, add ‘`-DUSE_C_LIMITS`’ to your `CXXFLAGS`.

Warnings in standard headers

Problem `make check` fails because warnings are treated as errors and the standard headers on my system generate warnings (as on e.g. HP-UX and FreeBSD).

Solution Run `configure` with `CXXFLAGS_STRICT_ERRORS` set to more tolerant warning flags (for example, set ‘`-Wall -W`’ for GCC but not ‘`-Werror`’).

Invalid data saved on I/O

Problem High resolution images are saved with invalid data on the Macintosh.

Explanation

You are using a big-endian host and there are known bugs in the image I/O operators.

Solution Save your images in the "plain pnm" (‘`.ppnm`’) file format instead of raw. Beware, while this is a correct workaround, the generated images are bigger.

8 Credits

8.1 Aknowledgements

The following people contributed to Olena, maybe indirectly through one of the numerous prototypes Olena has uprisen from. Olena would not be what it is today without their work.

THIERRY GÉRAUD
- for managing the project in the first place,
- for his work on the type system,
- for his numerous hours spent thinking about Olena.

ALEXANDRE DURET-LUTZ
- for having maintained the source tree for several years,
- for his work on the type system,
- for his work on the test system,
- for his work on the documentation system,
- for his numerous hours spent on Olena to make it better.

REDA DEHAK
- for managing the project,
- for his work on the color conversions,
- for his contributions to cleanup the sources.

AKIM DEMAILLE
- for his help with the configuration system,
- for his help to keep things clean.

ANTHONY PINAGOT
- for his work on Olena I/O,
- for his work on statistical operators,
- for his study on the FFT.

ASTRID WANG

for her work on static arrays.

DAVID LESAGE
- for his work on the type system,
- for his work on the new paradigm,
- for his contributions to cleanup the sources.

DIMITRI PAPADOPOULOS-ORFANOS

for his work on the type system

EMMANUEL TURQUIN
- for implementing transforms,
- for his work on integre.

GIOVANNI PALMA

FOR HIS CONTRIBUTIONS TO CLEANUP THE SOURCES.

HERU XUE

for his work on the color system.

Ignacy Gawedzki

for his work on the color system.

Jean Chalard

- for his work on colors,
- for implementing vectors and matrices,
- for implementing Olena iterators,
- for his study of wavelets.

Jean-Sébastien Mouret

- for his work on image I/O,
- for his work on the source tree and configuration system,
- for his work on fast morphological operators.

Jérôme Darbon

for his work on image morphology and Olena morpho.

Ludovic Perrine

for his study of fuzzy types.

Michaël Strauss

- for his work on image morphology,
- for his work on the watershed algorithms,
- for his work on Olena I/O.

Nicolas Burrus

- for his work on integre,
- for his work on Olena I/O,
- for his work on the source tree.

Pierre-Yves Strub

- for his work on Olena morpho,
- for his work on the source tree and configuration system,
- for his work on the type system.

Quôc Peyrot

for his work on the watershed algorithm.

Raphaël Poss

- for his work on the source tree and configuration system,
- for his work on the documentation.

Rémi Coupet

- for his work on Olena morpho,
- for his work on data types (pre-0.6),
- for his work on the Olena core,
- for his bibliographic research.

Renaud François

for his bibliographic research.

Sylvain Berlemont

- for his work on combinatorial maps,
- for his contributions to cleanup the sources.

YANN RÉGIS-GIANAS

- for his work on the type system,
- for his work on graphs,
- for his numerous contributions to various parts of Olena.

YOANN FABRE

for his work on the type system.

VINCENT BERRUCHON

In addition, we would like to thank EPITA and its user groups EpX and Prologin for giving us access to Solaris, FreeBSD, NetBSD, OpenBSD and CygWin machines.

8.2 Bibliography

Further information about Olena can be found into the following related papers:

- Thierry Graud, Yoann Fabre, Dimitri Papadopoulos-Orfanos, and Jean-Franois Mangin. *Vers une rutilisabilit totale des algorithmes de traitement d'images.* In the Proceedings of the 17th Symposium GRETSI on Signal and Image Processing, vol. 2, pages 331-334, Vannes, France, September 1999. In French (available in English as Technical Report 9902: *Towards a Total Reusability of Image Processing Algorithms*).
- Thierry Graud, Yoann Fabre, Alexandre Duret-Lutz, Dimitri Papadopoulos-Orfanos, and Jean-Franois Mangin. *Obtaining Genericity for Image Processing and Pattern Recognition Algorithms.* In the Proceedings of the 15th International Conference on Pattern Recognition (ICPR'2000), IEEE Computer Society, vol. 4, pages 816-819, Barcelona, Spain, September 2000.
- Alexandre Duret-Lutz. *Olena: a Component-Based Platform for Image Processing, mixing Generic, Generative and OO Programming.* In the Proceedings of the 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE 2000), Young Researchers Workshop (published in "Net.ObjectDays2000"; ISBN 3-89683-932-2), pages 653-659, Erfurt, Germany, October 2000.
- Alexandre Duret-Lutz, Thierry Graud, and Akim Demaille. *Generic Design Patterns in C++.* In the Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001), pages 189-202, San Antonio, Texas, USA, January-February 2001.
- Thierry Graud, Yoann Fabre, and Alexandre Duret-Lutz. *Applying Generic Programming to Image Processing.* In the Proceedings of the IASTED International Conference on Applied Informatics (AI'2001) – Symposium Advances in Computer Applications, ACTA Press, pages 577-581, Innsbruck, Austria, February 2001.
- *Generic Implementation of Morphological Image Operators*, Jrme Darbon, Thierry Graud, and Alexandre Duret-Lutz, submitted to International Symposium On Mathematical Morphology VI (ISMM 2002), April 3-5, 2002, Sydney, Australia.

You can download these papers and related materials from `http://www.lrde.epita.fr/cgi-bin/twiki/view/Publications`

Index and Table of contents

Table of Contents