Swilena

Raphaël Poss and Nicolas Burrus

This document is intended to describe Swilena, a simplified set of wrappers around the image processing library Olena.

1 Introduction

Swilena aims at providing interpreted languages access to the Olena image software processing library. In order to reach this goal, it relies on SWIG to create interfaces to Olena in different languages.

Swilena is made of three software components:

- SWIG definition files describing Olena,
- SWIG definition files describing Swilena components,
- a source tree able to generate extensions for Python, and hopefully Perl and other languages.

When compiled for a target interpreted language, the following *modules* are created:

swilena   Contains definitions for pixel types.

swilena_image1d, swilena_image2d, swilena_image3d
        Contain definitions for image and point types.

swilena_structelt1d, swilena_structelt2d, swilena_structelt3d
        Contain definitions for structural elements.

swilena_conversion1d, swilena_conversion2d, swilena_conversion3d
        Contain conversion functions between image types.

swilena_morpho1d, swilena_morpho2d, swilena_morpho3d
        Contain morphological operators over images.

swilena_arith1d, swilena_arith2d, swilena_arith3d
        Contain arithmetical operators over images.

The primary target language for Swilena is Python, because Python is the best supported back-end for SWIG. However, the SWIG definition files of Swilena are not bound to a particular interpreted language: any SWIG target language *providing enough expressiveness* can be used. Ruby modules are defined too, they are not documented here but they work almost the same way as Python modules. Here are the required features from the interpreted language:

- It must support overloading. O'Caml is therefore excluded.
- It should support objects. Else all method calls must be transformed into function calls, and object destruction must be made explicit.
- It must support dynamically loaded modules with dependencies between them.

Typically "ideal" target languages are Python, Ruby, Perl5, Tcl, Scheme.

Currently, the source tree only knows about Python and Ruby, but this may evolve in the future.

1.1 Using Swilena for Olena development

Obviously, Swilena provides the developer with a programming framework around Olena that has much shorter development cycles: new algorithms can be tested in Python without waiting for the compilation of C++ test sources.

Moreover, because compiling Swilena actually means instantiating Olena templates for a nearly complete Cartesian product of types, the success of the Swilena build process proves Olena's completeness.

2 Of SWIG and Swilena principles

As already suggested, Swilena and SWIG are closely related. In fact, SWIG is a wrapper generator, and Swilena is a set of input files for SWIG bundled in a package providing appropriate 'Makefile's to ease their handling.

This section provides some information about SWIG itself and presents the general guidelines that directed Swilena's development.

2.1 Introduction to SWIG

The following information is partly taken from the SWIG manual.

The best way to illustrate SWIG is with a simple example. Consider the following C code:

```c
/* File : example.c */

double  My_variable  = 3.0;

/* Compute n factorial */
int  fact(int n) {
        if (n <= 1) return 1;
        else return n * fact(n-1);
}

/* Compute n mod m */
int my_mod(int n, int m) {
        return n % m;
}
```

Suppose that you wanted to access these functions and the global variable My_variable from Python. You start by making a SWIG interface file as shown below (by convention, these files carry a .i suffix) :

2.1.1 SWIG interface file

```c
/* File : example.i */
%module example
%{
/* Put headers and other declarations here */
%}

extern double My_variable;
extern int    fact(int);
extern int    my_mod(int n, int m);
```

The interface file contains ANSI C function prototypes and variable declarations. The %module directive defines the name of the module that will be created by SWIG. The %{,%} block provides a location for inserting additional code such as C header files or additional C declarations.

2.1.2 The `swig` command

SWIG is invoked using the `swig` command. We can use this to build a Python module (under Linux) as follows :

```
unix > swig -python example.i
unix > gcc -c -fPIC example.c example_wrap.c -I/usr/include/python2.2
unix > gcc -shared example.o example_wrap.o -o _example.so
unix > python
Python 2.2.2 (#4, Oct 15 2002, 04:21:28)
[GCC 2.95.4 20011002 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from example import *
>>> fact(4)
24
>>> my_mod(23,7)
2
>>> My_variable + 4.5
7.5
>>>
```

The `swig` command produced two new files called 'example.py' and 'example_wrap.c'. The file 'example_wrap.c' should be compiled along with the 'example.c' file. Most operating systems and scripting languages now support dynamic loading of modules. In our example, our Python module has been compiled into a shared library that can be loaded into Python. When loaded, Python can now access the functions and variables declared in the SWIG interface. A look at the file 'example_wrap.c' reveals a hideous mess. However, you almost never need to worry about it.

2.2 SWIG and C++

Hopefully for our purpose, SWIG knows about many C++ language features. The following sections present SWIG features and their application with Olena.

2.2.1 A first example

For instance, it knows about classes: a SWIG description of a class yields the availability of this class in the target interpreted language. Here is an example:

```
/* oln_window.i */
%module oln_window
%{
#include "oln/basics2d.hh"
%}

namespace oln
{
  class window2d
    {
        window2d();

        unsigned card() const;
        int delta() const;

        window2d& add(int, int) ;
    };

  const window2d& win_c4p();
}
```

This SWIG definition file can be used with Python[1] as follows:

```
unix > swig -c++ -python oln_window.i
unix > g++ -c -fPIC oln_window_wrap.cxx -I/usr/include/python2.2 -Ipath_to_olena
unix > g++ -shared oln_window_wrap.o -o _oln_window.so
unix > python
Python 2.2.2 (#4, Oct 15 2002, 04:21:28)
[GCC 2.95.4 20011002 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from oln_window import *
>>> w = window2d()
>>> w.card()
0
>>> w.add(1,1).add(-1,-1)
<C window2d instance at _60bd2a08_p_oln__window2d>
>>> w.delta()
1
>>> w.card()
2
>>> w2 = win_c4p()
>>> w2.card()
5
```

This example exhibits several key points:

− SWIG knows about class constructors and references and treats them trivially.

− The SWIG description need not follow exactly the strict C++ definition. In the previous example, the Olena class `window2d` is far more complex than what is expressed in the SWIG declaration; however, for a SWIG description to be valid, it only needs to describe a class more *general* than the real one.

---

[1] for a Python primer, See Chapter 3 [Python Usage], page 10

– Although it is possible to do so, the SWIG description need not express class inheritance.

2.2.2 Operators and class extensions

When the target interpreted language allows overloading arithmetical operators for classes, SWIG can propagate this C++ feature. However, if it does not, it is needed to provide an artificial method-like interface to the class operators.

Here is a demonstration:

```
/* oln_window.i */
%module oln_window
%{
#include "oln/basics2d.hh"
#include <sstream>
%}

%include std_string.i // for SWIG to know about std::string

namespace oln
{
  class window2d
    {
        window2d();
        window2d operator-() const;

        // the negation cannot be overload in all interpreted
        // languages. Therefore, we create on-the-fly a new
        // method in class window2d to call operator- :
        %extend {
           window2d neg() const
           { return -(*self); }
        };

        // Similarly, interpreted languages cannot cope
        // with C++ iostreams. Therefore, here is a workaround:
        %extend {
           std::string describe() const
           {
               std::ostringstream s;
               s << *self;
               return s.str();
           }
        };
    };
}
```

The module generated by SWIG can then be used as follows:

```
unix > python
>>> from oln_window import *
>>> w = window2d()
>>> w.add(1,1).add(0,1).describe()
```

```
'[(1,1)(0,1)]'
>>> w2 = -w1
>>> w2.describe()
'[(-1,-1)(0,-1)]'
>>> w2.neg().describe()
'[(1,1)(0,1)]'
>>>
```

Here are the key points exhibited by this example:

— When the interpreted language allows so, SWIG understands C++ operator overloading and treats it trivially.

— The `%extend` SWIG sections allows adding pseudo-methods to interfaced classes. It can be used to provide function names to C++ operators for interpreted languages that do no not cope with operator overloading (e.g. Perl).

— When the description file includes '`std_string.i`', SWIG knows about the C++ standard type `std::string`, and knows how to convert it to and from the interpreted language's native string type.

2.2.3 SWIG and C++ templates

In addition to function, variables, structures and classes, SWIG knows about templates. However, because scripting languages do not support templates and template instanciation, information must be provided to SWIG to explain what template instances must be available to the scripting language.

Here is a demonstration:

```
/* oln_window.i */
%module oln_window
%{
#include "oln/basics2d.hh"
#include <sstream>
%}

%include std_string.i // for SWIG to know about std::string

namespace oln
{
  template <typename T>
  class w_window2d
    {
        w_window2d();

        window2d& add(int, int, T) ;

        unsigned card() const;
        T w(unsigned) const;

        %extend {
           std::string describe() const
           {
               std::ostringstream s;
               s << *self;
               return s.str();
           }
        };
    };
}

%template(w_win2d_int) oln::w_window2d<int>;
%template(w_win2d_float) oln::w_window2d<float>;
```

This module allows e.g. the following Python session:

```
unix > python
>>> from oln_window import *
>>> w = w_win2d_int()
>>> w.add(1,1,10).add(0,1,3).describe()
'[((1,1),10)((0,1),3)]'
>>> w2 = w_win2d_float()
>>> w2.add(1,1,10.4).add(0,1,3.14).describe()
'[((1,1),10.5)((0,1),3.14)]'
>>> w2.w(1)
3.1400001049041748
>>>
```

Here are the key points exhibited by this example:

− SWIG can only wrap template *instances*. The instantiation must be made explicit.

- However, when instantiating a template class, all its methods are instantiated at the same time.
- Template instances must be given a unique identifier (e.g. `w_win2d_int`), because C++ template instance names (e.g. `oln::w_window2d<int>`) are not valid scripting type identifiers.

2.2.4 SWIG & C++ gotchas

When using SWIG and C++ sources, the following notes need be taken into consideration.

- SWIG collates all C++ namespaces in the global module namespace. Therefore, beware of wrapped function or class names that appear simultaneously in several namespaces with different definitions: they are not handled properly by SWIG.
- The C++ parser in SWIG cannot deal with C++ template partial specialization. Therefore, C++ tricks such as static hierarchies and virtual types cannot be exposed to SWIG. Consider hiding the static inheritance tree and exposing the most derived classes instead.
- Families of similar template functions cannot be instantiated with a single SWIG directive. Use SWIG macros and appropriate naming conventions for this purpose:

```
template<typename T>
void foo(T x);

template<typename T>
void bar(T x);

%define Instantiate_Templates_For(Type)
%template (foo_ ## Type) foo<Type >;
%template (bar_ ## Type) bar<Type >;
%enddef

Instantiate_Templates_For(int);
        // yields foo_int and bar_int

Instantiate_Templates_For(float);
        // yields foo_float and bar_float
```

2.3 Olena and SWIG

3 Python Usage

3.1 Starting Python

Start your python interpreter in the usual way:

```
~/src/swilena/python % python
Python 2.2.2 (#4, Oct 15 2002, 04:21:28)
[GCC 2.95.4 20011002 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The `>>>` text is the Python standard prompt, where you can enter Python statements.

You can also write Python programs as scripts, using the following script template:

```
#! /usr/bin/env python

... your program here ...
```

3.2 Python Basics

Python does not have mandatory statement terminators. Statements end at the end of the line. However, you can use the semicolon (';') as a command separator.

```
>>> print "hello"
hello
>>> print "hello"; print "world"
hello
world
```

Python data types are the integer (signed), the float (C 'double'), and the character string. Constants can be expressed intuitively:

```
>>> print "hello"; print 123
hello
123
>>> 1./3
0.33333333333333331
>>>
```

Assigning variables is also simple:

```
>>> i=123
>>> print i
123
>>> i+=42
>>> i
165
>>>
```

There are several forms of loops. The most intuitive are:

```
>>> k=0
>>> for i in range(0, 10, 1):
...     k=k+i
...
>>> print k
45
>>> k=0;i=0
```

```
>>> while i < 100:
...    k=k+i
...    i+=1
...
>>> print k
45
>>>
```

3.2.1 Python Modules

Python is module- and object- oriented. It has a unique scope operator (the period, '.')
to access module components and object methods and attributes.

Modules must be loaded before their functions can be used:

```
>>> os.getcwd() # FAILS
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'os' is not defined

>>> import os # load the "os" module
>>> os.getcwd() # OK, access getcwd() in module "os"
'/home/lrde/stud/raph/src/swilena/python'
>>>
```

You can import all fields of a module in the global scope, to avoid prefixing function
calls:

```
>>> from os import *
>>> getcwd() # "os." is not needed anymore
'/home/lrde/stud/raph/src/swilena/python'
>>>
```

3.2.2 Python Objects

By convention, standard Python Classes have names that start with a capital. This helps
disambiguate class names and module names:

```
>>> import random
>>> r=random.Random() # call constructor for class Random in module random
```

As shown in the previous example, constructors have the name of the class, as in C++.
Method calls are also very intuitive:

```
>>> r.randint(10, 20) # call method "randint" over object r just constructed
17
>>> r.randint(10, 20)
12
```

3.3 Using Swilena

3.3.1 Fooling around

First, start Python and load the Swilena modules for 2D:

```
>>> from swilena_image2d import *
```

If everything succeeds, you can start creating images and saving them. Here is how to
create a random images using the standard Python class 'Random' and Olena:

```
>>> import random;r=random.Random()
>>> i=image2d_u8(10, 10)
```

```
>>> for x in range(0, i.ncols(), 1):
...  for y in range(0, i.nrows(), 1):
...   i.at(x, y) = r.randint(0,255)
...
>>>
```

The previous code creates an empty 2D image using 8-bit unsigned integers to store pixel values, with a size of 10x10 pixels. It then initalizes all pixels in the image using loops.

It is then possible to store this image in a file:

```
>>> i.save("foo.pgm")
1
>>>
```

The 'save' method takes a file name and attemps saving the image with the format specified as extension. Intuitively, there is also a 'load' method:

```
>>> i.load("lena.pgm")
1
>>> i.at(3,3)
162
>>>
```

Both 'load' and 'save' return a boolean set to 1 if the operation succeeded, 0 else.

As demonstrated in the previous example, Swilena images have methods. However, there are not many:

'load'   load an image from a file.

'save'   save the image to a file.

'at'   return a reference to the indicated pixel.

'set'   set the value of the indicated pixel.

'ncols'   return the number of columns in the image. This method is valid for 1D, 2D and 3D images.

'nrows'   return the number of rows in the image. This method is only valid for 2D and 3D images.

'nslices'   return the number of slices of a 3D image.

Also it hasn't been demonstrated yet, the 'at' method is polymorphic, and accepts a 'point' instead of coordinates as a pixel location:

```
>>> i.at(point2d(3, 3))
162
>>>
```

Algorithms are grouped in corresponding modules. For example, morphological operators are grouped in the '*morpho' modules:

```
>>> import swilena_structelt2d
>>> import swilena_morpho2d
>>> i2=swilena_morpho2d.erosion(i, win_c4p())
>>> i3=swilena_morpho2d.dilation(i, win_c4p())
```

These function are polymorphic and should work for nearly all image types. Refer to the following chapter (see Chapter 4 [API Reference], page 13) for a description of what are valid calls.

4 API Reference

4.1 Pixel Types

Swilena Type	C++ Type	Description
int_u8	ntg::int_u8	8-bit unsigned integer
int_u32	ntg::int_u32	32-bit unsigned integer
int_s8	ntg::int_s8	8-bit signed integer
int_s32	ntg::int_s32	32-bit signed integer
bin	ntg::bin	1-bit value
float_d	ntg::float_d	double precision float
cplx_rect	ntg::cplx<rect, float_d>	complex number in rectangular representation.
cplx_polar	ntg::cplx<polar, float_d>	complex number in polar representation.

All pixel types are represented by classes in Swilena.

All pixel types share the following interface:

`(default constructor)`
Create a default pixel value, typically 0 (or 0,0,0 for rgb).

`(constructor from value)`
Create a pixel with the specified value.

`operator==(pixel), equals(pixel)`
Compare the pixel with another.

In addition, scalar (integer, floating) pixel types share the following interface:

`val()`   Get the scalar value of the pixel.

`value(integer or float)`
Set the value of the pixel.

The module name for these types is **swilena**.

4.2 Point Types

Here are the 'point' and 'dpoint' types:

Swilena Type	C++ Type
point1d	oln::point1d
point2d	oln::point2d
point3d	oln::point3d
dpoint1d	oln::dpoint1d
dpoint2d	oln::dpoint2d
dpoint3d	oln::dpoint3d

These classes share the following interface:

`(default constructor)`
Create a point designating the origin of an image.

`(constructor with coordinates)`
Create a point designating the specified location.

`col(), row(), slice()`
Access the coordinates of the point.

```
col(unsigned), row(unsigned), slice(unsigned)
```
   Set the coordinates of the point.

 "dpoints" represent distances between points, hence can be added to "points".

 The module names for these types are **swilena1d**, **swilena2d** and **swilena3d**.

4.3 Image Types

Here are the image types corresponding to pixel data types:

Swilena Type	C++ Type	Status
image1d_bin	oln::image1d<oln::bin>	ready
image1d_u8	oln::image1d<oln::int_u8>	ready
image1d_u32	oln::image1d<oln::int_u32>	ready
image1d_s8	oln::image1d<oln::int_s8>	ready
image1d_s32	oln::image1d<oln::int_s32>	ready
image1d_float_d	oln::image1d<oln::float_d>	ready
image2d_bin	oln::image2d<oln::bin>	ready
image2d_u8	oln::image2d<oln::int_u8>	ready
image2d_u32	oln::image2d<oln::int_u32>	ready
image2d_s8	oln::image2d<oln::int_s8>	ready
image2d_s32	oln::image2d<oln::int_s32>	ready
image2d_float_d	oln::image2d<oln::float_d>	ready
image3d_bin	oln::image3d<oln::bin>	ready
image3d_u8	oln::image3d<oln::int_u8>	ready
image3d_u32	oln::image3d<oln::int_u32>	ready
image3d_s8	oln::image3d<oln::int_s8>	ready
image3d_s32	oln::image3d<oln::int_s32>	ready
image3d_float_d	oln::image3d<oln::float_d>	ready

 All image types are classes in Swilena.

 All images types share the following interface:

```
(default constructor)
```
   Create an empty image. After calling this constructor, the image does not yet
   "exist" and must be (for example) 'load'ed or 'convert'ed to.

```
(constructor with dimensions and border)
```
   Create a blank image with the specified dimensions, and a hidden zone aiming
   to serve as a "border" for algorithms.

```
(constructor with dimensions)
```
   Create a blank image, using a default border width of 2.

```
at(point)
```
   Access the pixel at 'point', which can be of type 'point1d', 'point2d' or
   'point3d'.

```
at(dimensions)
```
   Access the pixel at point specified by one, two, or three coordinates.

```
set(point, value)
```
   Set the value of the pixel at 'point'.

```
set(dimensions, value)
```
   Set the value of the pixel at the given coordinates.

`ncols(), nrows(), nslices()`
> Retrieve the dimensions of the image.

`load(filename), save(filename)`
> Input/output to files.

The module names for these types are **swilena_image1d**, **swilena_image2d** and **swilena_image3d**.

4.4 Structural Element types

Here are the structural elements:

Swilena Type	C++ Type
window1d	oln::window1d
window2d	oln::window2d
window3d	oln::window3d

All these types family share the following interface:

`(default constructor)`
> Create an empty window.

`(constructor from size)`
> Create an empty window with the specified size.

`delta()`   Return the magnitude of the window.

`unary operator-(), neg()`
> Return the symmetric window.

`card()`   Return the number of points defined in the window.

`dp(i)`   Return the i'nth "dpoint" in the window.

`has(dpoint)`
> Return true if the window contains the specified dpoint.

`describe()`
> Return a string describing the structure of the window.

In addition, members of the "window" family share the following interface:

`add(dpoint), add(coordinates)`
> Add the specified relative point to the window.

`inter(other window)`
> Return the intersection of this window and another.

`uni(other window)`
> Return the union of this window and another.

Here are the corresponding instantiation functions, which have the same name as their C++ counterpart:

Swilena Name	Return Type
win_c2_only()	window1d
win_c2p()	window1d
mk_win_segment(width)	window1d
win_c4_only()	window2d
win_c4p()	window2d

win_c8_only()	window2d
win_c8p()	window2d
mk_win_rectangle(nrows,ncols)	window2d
mk_win_ellipse(yradius,xradius)	window2d
mk_win_square(width)	window2d
mk_win_disc(radius)	window2d
win_c6_only()	window3d
win_c6p()	window3d
win_c18_only()	window3d
win_c18p()	window3d
win_c26_only()	window3d
win_c26p()	window3d
mk_win_block(nslices,nrows,ncols)	window3d
mk_win_ellipsoid(zradius,yradius,xradius)	window3d
mk_win_cube(width)	window3d
mk_win_ball(radius)	window3d

See the documentation of Olena for a description of these functions.

The module names for these types and functions are **swilena1d**, **swilena2d** and **swilena3d**.

4.5 Neighborhood Types

Here are the neighborhoods:

Swilena Type	C++ Type
neighborhood1d	oln::neighborhood1d
neighborhood2d	oln::neighborhood2d
neighborhood3d	oln::neighborhood3d

Neighborhoods behave like windows in regards to their interface.

Here are the corresponding instantiation functions, which have the same name as their C++ counterpart:

Swilena Name	Return Type
neighb_c2()	neighborhood1d
mk_neighb_segment(width)	neighborhood1d
mk_win_from_neighb(neigh1d)	window1d
neighb_c4()	neighborhood2d
mk_neighb_square(width)	neighborhood2d
mk_neighb_rectangle(nrows,ncols)	neighborhood2d
mk_win_grom_neighb(neigh2d)	window2d
neighb_c6()	neighborhood3d
neighb_c18()	neighborhood3d
neighb_c26()	neighborhood3d
mk_neighb_block(nslices,nrows,ncols)	neighborhood3d
mk_neighb_cube(size)	neighborhood3d
mk_win_from_neighb(neigh3d)	window3d

The module names for these types and functions are **swilena1d**, **swilena2d** and **swilena3d**.

4.6 Conversion Functions

An image can be converted into another kind of image if it their types are compatible. Several conversions are possible: with truncation, with stretching or direct.

The naming scheme is simple, append the name of the destination type to the kind of conversion. For example, to perform a conversion from an image of `int_u32` toward an image of `int_u8` with stretching, use `stretch_to_int_u8(your_image_in_int_u32)`.

Reducing conversions

Stretching (`stretch_to`) and truncation (`bound_to`) are available.

Here are the possible reduction conversions:

Source type	Destination type
int_u32	int_u8
int_s32	int_s8

Direct conversions

Direct conversions (`cast_to`) are possible between all scalar types. Some additional conversions are possible:

Source type	Destination type
bin	any scalar type
any integer type	bin

The module names for these functions are **swilena_conversions1d**, **swilena_conversions2d** and **swilena_conversions3d**.

4.7 Morpho Functions

The following morpho functions are available, from their counterpart in the C++ namespace `oln::morpho`:

Swilena

fast_opening(img, win)
fast_closing(img, win)
fast_dilation(img, win)
fast_erosion(img, win)
fast_beucher_gradient(img, win)
fast_internal_gradient(img, win)
fast_external_gradient(img, win)
fast_white_top_hat(img, win)
fast_black_top_hat(img, win)
fast_self_complementary_top_hat(img, win)
fast_thinning(img, win)
fast_thickening(img, win)

opening(img, win)
closing(img, win)
dilation(img, win)
erosion(img, win)
beucher_gradient(img, win)
internal_gradient(img, win)

external_gradient(img, win)
white_top_hat(img, win)
black_top_hat(img, win)
self_complementary_top_hat(img, win)
thinning(img, win, win)
thickening(img, win, win)

simple_geodesic_dilation(img, img, neighb)
simple_geodesic_erosion(img, img, neighb)
geodesic_dilation(img, img, neighb)
geodesic_erosion(img, img, neighb)

sure_geodesic_reconstruction_dilation(img, img, neighb)
sequential_geodesic_reconstruction_dilation(img, img, neighb)
vincent_sequential_geodesic_reconstruction_dilation(img, img, neighb)
hybrid_geodesic_reconstruction_dilation(img, img, neighb)
exist_init_dilation(point img, img, win)

sure_geodesic_reconstruction_erosion(img, img, neighb)
sequential_geodesic_reconstruction_erosion(img, img, neighb)
hybrid_geodesic_reconstruction_erosion(img, img, neighb)
exist_init_erosion(point img, img, win)

watershed_seg(img_int, img, neighb)
watershed_con(img_int, img, neighb)
watershed_seg_or(img, img_int, neighb)

FIXME: laplacian and hit_or_miss are missing.

Some of them do not actually work with every image type. The exact list of supported algorithms for each kind of image should be added to the documentation. This list may evolve soon between Olena releases though.

The module names for these functions are **swilena_morpho1d**, **swilena_morpho2d** and **swilena_morpho3d**.

Index and Table of contents

(Index is nonexistent)

Table of Contents