Olena: like others?

Thierry Géraud

EPITA Research and Development Laboratory (LRDE)

June 2007

Outline

Outline

Outline

Outline

Outline

Objectives

abbrev: IP = Image Processing

A twofold objective

- Give thought-provoking ideas about IP libraries.
- Provide some clues to
  - catch what users expect from a new library
  - better understand the Olena project.

This talk is **not** about:

- the other parts of IP platforms/environments
- the interaction between the library and the other parts.

Why Focusing on the Library Part?

Library v. Platform

The library is the **heart** of an IP platform / environment.

The library:

- is the part that actually does the work
- transmits its features and limitations to the platform
- can be seen as a stand-alone tool.

I heard a disturbing statement 10 years ago...

The Disturbing Statement!

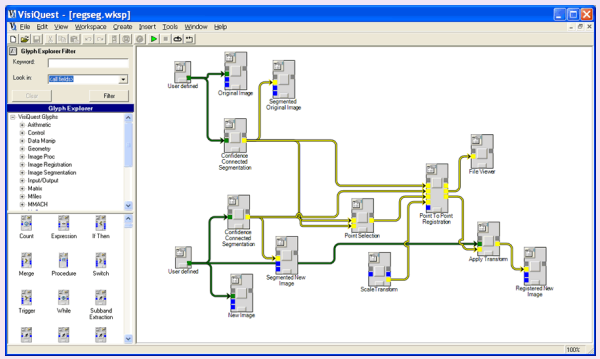
"Why do you want to write an IP library?
There are too many algorithms...
you cannot implement them all!"

(Josiane Zérubia, researcher with INRIA, private conversation, 1997.)

Outline

Image Processing with Blocks

from VisiQuest `http://www.accusoft.com/products/visiquest/`



a block =

- **a box**:
  - either a single IPO (finest grain, e.g., a filter)
  - or a program combining existing boxes (coarser grain)
- **or data**: usually images.

IP People

IP practitioners are not really skilled in programming.

Just consider that:

- they are dummies in computer science
- a library is only a tool / a means for them
- they do not think like us!

A Taxonomy of Library Users

- an **assembler** composes blocks to solve an IP problem

  [artifact] writes a lot of function calls

- a **designer** adds some new boxes

  [artifact] needs low-level loops

- a **provider** adds some new data types

  [artifact] knows about the `struct` and/or `class` keywords

- an **architect** works on the internals of the library

  [artifact] deals with hardcore code

Required programming skills go up from assembler to architect.

Library Usage

Ideal / **naive** distribution of a library usage (people $\times$ time):

task	*usage*	*comments*
assembling	90%	**using** is great! (v. **re-coding** is a pity)
designing	9%	adding some new box can occur...
providing	1%	needing a new data type hardly happens!

Is the access to a large set of ready-to-use blocks sufficient to justify getting a tool such as a library?

We do **not** think so...

Some Libraries In a Nutshell

That is actually the way some libraries are presented:

CImg	"contains useful image processing algorithms for image loading/saving, displaying, resizing/rotating, filtering, object drawing, etc."
ImLib3D	"an open source C++ library for 3D (volumetric) image processing."
ITK	"an open-source software toolkit for performing registration and segmentation."
OpenCV	"example areas are object identification; segmentation and recognition; [...]"
Pandore	"regroupe des opérateurs traitant d'images 1D, 2D et 3D, en niveaux de gris, en couleurs et multi-spectrales."
QgarLib	"a set of C++ components implementing basic graphics analysis and recognition methods."

- IP libraries offer **blocks**
- they target
  - either some given data
  - or some application domain
- yet we can find something different...

Outline

GIL & VIGRA

VIGRA "is a library that puts its main emphasize on **customizable** algorithms and data structures."

`http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra/`

GIL "is concept-based and allows virtually every component to be **replaced**."

`http://opensource.adobe.com/gil/`

:-(

But reality is disappointing:

VIGRA "After all, the point of VIGRA is **algorithms**, not the core; the core is just there to give the algorithms something to work with."

(Ullrich Köthe, VIGRA's author and maintainer, Private mail, December 2006.)

GIL "Computer vision is a niche domain. There is a much broader domain of **basic** image manipulation [...] loading, converting, and displaying. People who need a Canny edge detector are a minority."

(Lubomir Bourdev, GIL's author and maintainer, in Boost thread, October 2006.)

Temporary Conclusion

What can we expect from an IP library?

say...

reusability, generality, flexibility, customization, etc.

Let us think differently about some well known issues in designing an IP library...

Truth is Elsewhere

Our position:

- featuring algorithms is useless
- morphers are only luxury
- efficiency is useless
- genericity $\Rightarrow$ performance loss.

Outline

Classical Algorithms

Many **classical** IPOs are usually:

- featured by several available libraries
- or locally available in every IP lab
- and almost everybody has already its favorite solution!

So:

- nobody expects a replacement
- there is no point in targeting simple users.

Non-Classical Algorithms

Non-classical algorithms are really very specific; thus

- they usually are uninteresting
  speaking from a statistical point of view,

  i.e., very few people might be interested in very specific solutions;
- they are plethoric

  and there is nothing to do against that.

Featuring algorithms is **not** an attractive library feature!

Outline

Reminder

Morpher

A morpher is a data type built over one or several other data types.

For instance:

```
value_cast<rgb8>( stack(ima_r, ima_g, ima_b) )
```

is a color image created on the fly with no own data.

Three Kinds of Morphers

The set of morphers provide one of those 3 features:

- a lightweight type to replace an existing one
  so they ease to save both memory and execution time
- a non-intrusive equipment
  so they just save the client a copy-paste-modify operation
- a brand new type,
  for instance, a region of an image is an image.

Useless or What?

Morphers do not really bring new features!

Instead:

- they are like some complicated stuff that one does not want to hear about (new users start with simple features)
- somehow they sound like some computer scientist brain fuck (IP practitioners do not care about that).

Outline

C Age is Over

Some easy questions to ask to a practitioner:

- when your IP solution runs X, do you need to save Y? with:
  - X = all night long / Y = a couple of hours
  - X = 9 min / Y = 3 min
  - X = 1 sec / Y = 1 sec
- do you prefer an IP library to be
  - simple
  - or more efficient but harder to program with?

Efficiency?

- Only few applications / people are sensitive to efficiency.
- Efficiency is **never** a goal in itself.

Last

- providing some dedicated “fast” algorithms is better than optimizing definitely “slow” ones.

Outline

Genericity is Slow

Just try Olena, you will know that!

Outline

No restriction

Feature 1:

Olena insures that the client is not restricted / limited.

- Artifact:
  you know that whatever you want is possible!
- Key idea:
  capability is much more important than **availability**.

Reusable Algorithms

Feature 2:

Olena insures that every algorithm is reusable.

- Artifact:
  whatever you do is transposable to some other contexts.
- Key idea:
  capitalize your work, instead of reproduce **one-shot** works.

Many Tools

Feature 3:

Olena offers a bunch of simple tools to ease the task of designers.

- Artifact:
  you get some help whatever you want to do.
- Key ideas:
  concentrate on **what** to do, not on **how** to do it.

Outline

Main Target

Our potential users are DESIGNERS.

Our Guide (1/2)

Rules

1. Prefer simplicity to efficiency.

   Corollary: getting efficiency will be possible.

2. Get genericity without sacrificing simplicity.

   Corollary: we have already done a lot of work to that aim.

Illustration

Algorithm:

$\forall p \in \mathcal{D}(f),\quad oper(f(p),\ c)$

Olena in 2007:

```
template <typename O, typename I, typename T>
void op(Image<I>& f, T c)
{
  O oper;
  oIn_piter(I) p(f.domain());
  for_all(p)
    oper(f(p), c);
}
```

Olena in 2000:

```
template< typename O,
          template< class U > class get_A = get_value,
          typename P = Pred_true >
struct op
{
  template< typename I > static
  void on( I& f,
           const get_A< I::value_type >::output_type& c,
           P pred = P() )
    {
      O oper;
      get_A< I::value_type > access;
      I::iterator_type iter( f );
      for ( iter.first(); ! iter.isDone(); iter.next() )
        if ( pred( access( iter() ) ) )
          oper( access( iter() ), c );
    }
};
```

Our Guide (2/2)

Rules

3 Multiply the number of helper tools.

Corollary: writing algorithms will be quick and easy.

4 Provide flexibility through interfaces / image taxonomy.

Corollary: algorithms and morphers are nothing but a proof of concept.

Conclusion

So what?

What's Worth Remembering

- We should be different.
- We should feature tools, not algorithms.
- We shall prove that genericity can go with simplicity.

The To-Do List

- Identify useful tools.
- Clearly define abstractions and interfaces.
- Be also a PROVIDER.

Questions?

Think different!