Canvases for Morphological Algorithms

Ugo Jardonnet

Technical Report n^o0753, MOIS 2007
revision 1446

Abstract: Olena is a generic image processing library developed at LRDE. It provides many morphological algorithms. Mathematical morphology offers several powerful tools in image processing and analysis. Similarities appear when writing morphological algorithms. Thereby, we can classify those tools and then build canvases of algorithms. This report presents what is a canvas and why canvases matter. We will see different manners to implement canvases with their pro and con arguments. Finally, we will explain which canvas implementation we have chosen for Olena and why.

Résumé: Olena est une bibliothèque générique de traitement d'images développée au LRDE. Elle propose un grand nombre d'algorithmes morphologiques. La morphologie mathématique, offre des outils très puissants de traitement et d'analyse d'images.
Des similarités apparaissant dans l'écriture des algorithmes morphologiques, il est possible de les classifier et, ainsi, de proposer un certain nombre de "canevas" d'algorithmes. Ce rapport définie ce que sont les canevas et les avantages qu'ils apportent. Apres une brève introduction à la morphologie mathématique, cet exposé presentera différents canevas d'algorithmes retenus par Olena.

Keywords
Canvas, Mathematical Morphology, Image Processing

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
lrde@lrde.epita.fr – http://www.lrde.epita.fr/

Copying this document

Contents

Chapter 1

Introduction

1.1 This Report

The goal of this report is to introduce a new programming concept called *canvas*. A canvas is a step to factor redundancy in code. It provides interesting functionalities for user, like documentation or great extensibility.

This report deals with canvas implementation in mathematical morphology. However, the implementation we proposed can be used in any area of development. We will see what information we need for the design of canvases, and how to implement them.

This study takes place in the Olena project. Olena is a generic image processing library developed at LRDE. This project is developed in C++, using a the SCOOP paradigm (Static C++ object-oriented programming). All pieces of code presented in this report are parts of the Olena library. This library is free software under the GNU General Public License version 2.

Olena 1.0 (January 2008) should have a larges number of its algorithms and tools based on canvas. Indeed, this report is a part of the work of development made on the Olena project.

This report is composed of three sections. After a small presentation of mathematical morphology, some redundancy in classical morphological algorithms will be underlined in section 1. In the 2nd section, a definition of canvases is given and different canvas implementation are discussed. Finally, we draw some conclusions in section 3.

1.2 Acknowledgment

Thanks to Thierry Géraud for his help. Thanks to for his help. Thanks a lot for the reviewers who helped me to improve this report.

Chapter 2

Redundant code in Mathematical Morphology

Mathematical morphology is a part of non linear Information Processing appeared in the 1960 decade (G. Matheron(2) & J. Serra (1), Ecole des Mines in Paris, France).
In opposition to linear image processing, mathematical morphology is not based on signal processing, but uses concept of set theory, algebra and geometry. Today, this science is widely used in image processing. Morphology can provide boundaries of objects, their skeletons, and their convex hulls. It is also useful for many pre- and post-processing techniques, especially in edge thinning and pruning. The three basic operators of mathematical morphology are erosion, dilation and the discrete convolution product.

2.1 Basic Morphological Operators

This section is an introduction to the three basic morphological set transformations : The erosion, the dilation and the discrete convolution product. We will see it exists some common parts in the writing of their algorithms. We will first introduces those set transformations. Then we will extract the common part of their algorithms.

2.1.1 Definitions

Let A and B be subsets of $\mathbb{Z}^2$. The translation of A by x is denoted A_x and is defined as

$$A_x = \{c : c = a + x, \forall a \in A\}$$

The reflection of B, denoted $\hat{B}$, is defined as

$$\hat{B} = \{x : x = -b, \forall b \in B\}.$$

The structuring element is to mathematical morphology what the convolution kernel is to linear filter theory. It's a usually small set of points also called window. It describes which points around the point p have to be used for the computation of p in the output image. The two most common structuring elements (given a Cartesian grid) are the 4-connected and 8-connected sets, N_4 and N_8. They are illustrated in the following figure.

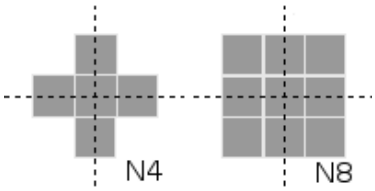

Figure 1: The structuring elements N_4 and N_8.

2.1.2 Erosion

Erosion of the object A by a structuring element B is given by

$$\varepsilon_B(A) = \{x : B_x \subseteq A\}$$

Consider the example (Figure 2) where we can see an eroded image (on the right side) based on the morphological erosion of a binary image (on the left side) by a structuring element w. The original state of the image appears in red on the right side.

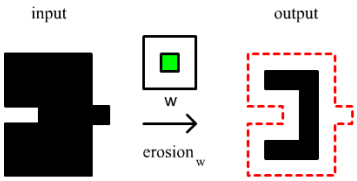

Figure 2: Erosion of a binary image.

2.1.3 Dilation

Dilation of the object A by the structuring element B is given by

$$\delta_B(A) = \{x : \hat{B}_x \cap A \neq \}$$

The result is a new set made up of all points generated by obtaining the reflection of B about its origin and then shifting this reflection by x. Consider the example (Figure 3) where we can see a dilated image (on the right side) based on the morphological dilation of a binary image (on the left side) by a structuring element w. The original state of the image appears in red on the right side.

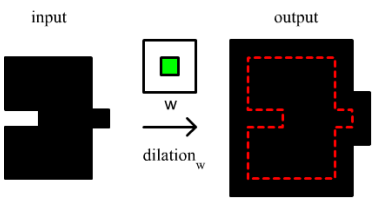

Figure 3: Dilation of a binary image.

2.1.4 Discrete Convolution Product

Let f and g be greyscale images. f is the Signal, g the Response. The discrete convolution product between f and g is given by

$$f * g = \{p : p = \Sigma_q f(p - q) \times g(q)\}$$

The following figure illustrates a convolution product where the Response g is the structural element. It slides along the signal f and is used with the point of f, to compute the convoluted signal.

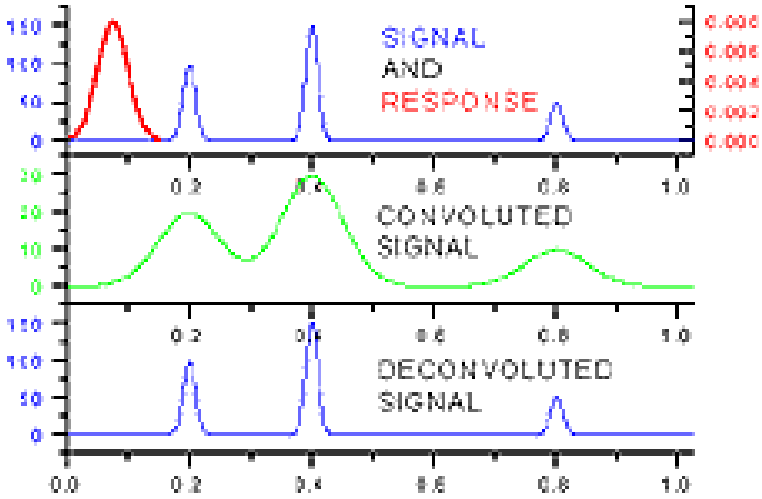

Figure 4: Discrete Convolution product between a Signal f and a Response g.

2.2 Similarities between Algorithms

More than twenty classical morphological algorithms are built upon erosion and dilation (see figure 7). Furthermore, those operators are similar to another basic operator in image processing, the discrete convolution product. Algorithms implementing those operators are similar too. It could be interesting to extract redundancies in those algorithms and make them accessible as a unique programming object. This object would then be used by any other operator or algorithm needing this factored piece of code. For instance, The erosion and the discrete convolution product could both use a single object of this type. The two following pieces of code are the algorithm of erosion and discrete convolution product. The important lines are commented. We want to extract the redundant pieces of code in those algorithms. We will see that this is very simple with the two following algorithms.

5: Algorithm of Erosion

```
function EROSION(f)
    point_iter p
    for_all(p)
        {
            r = ∞ (or supremum)
            window_iter q
            for_all(q)
                r = ⋁ (r, f(q))                    ▷ Minimum
            output(p) = r
        }
end function
```

6: Algorithm of Discrete Convolution Product

```
function CONVOLUTION(f, g)
    point_iter(I) p
    for_all(p)
        {
            r = 0
            window_iter(I) q
            for_all(q)
                r = r + f(p − q) × g(q)                ▷ Sum of products
            output(p) = r
        }
end function
```

Analysis The algorithms of erosion (5) and the discrete convolution product (6) present some similarities. They are both based on nested loops. The first loop creates an iterator walking the entire image and the second one scans points around the iterator. Then a calculus is made with the different values of points in this window. For each iteration, the result of this calculus is assigned to the current point (pointed by the iterator) in the result image.

Hence, the convolution product and the erosion/dilation operators have similar constructs ; we can factor their common parts down to this skeleton :

```
for_all(p)
    {
        Initialisation.
        for_all(q)
            Processing of points p and q.
        output(p) = result
    }
```

Erosion and dilation are at the heart of many algorithms. This is a classical situation where we want to factor code. Because of the prominence of dilation, erosion and discrete convolution product in mathematical morphology, we can observe many redundancy in classical morphological algorithms. Actually, a dozen of algorithm can be written using erosion or dilation (see Figure 7). Thereby, the factoring of this basic operators implicates many pieces of code factored in the whole set of morphological operators.

7: Some Morphological Operator based on Erosion - Dilation

1. Opening :
   $\gamma_B(A) = \delta_B(\varepsilon_B(A))$.
2. Closing :
   $\varphi_B(A) = \varepsilon_B(\delta_B(A))$.
3. Beucher's Gradient :
   $Gradient(A) = \delta_B(A) - \varepsilon_B(A)$
4. External Gradient :
   $Gradient_{ext}(A) = \delta_B(A) - A$

5. Internal Gradient :
   $Gradient_{int}(A) = A - \varepsilon_B(A)$
6. Laplacian :
   $Laplacian(A) = Gradient_{ext}(A) - Gradient_{int}(A)$

2.3 Patterns in Mathematical Morphology

2.3.1 d'Ornellas Pattern

Marcos Cordeiro d'Ornellas wrote many publications on the idea of algorithmic patterns (see his thesis *Algorithmic Patterns for Morphological Image Processing*, 2001 (3)). Algorithmic patterns should be documented, stored, and later reused in software development. This is one reason for the construction of a pattern library, including patterns for morphological image processing that are often used in the design of operators and applications. The idea is that developers should be able to find easily a matching pattern for their algorithms. Patterns must not be absolutely written in a programing language, so the pseudo-code following are still not canvases but patterns.

Here follow the tree main patterns isolated by d'Ornellas :

8: The Parallel Pattern

```
for_all(p)
    {
        for_all(q)
            Process(p, q).                    ▷ Variable part.
    }
```

The parallel pattern is very simple. It is nested loops where the first iterates on the image, while the nested iterates on the points covered by the structural element. Algorithms following this patterns can be parallelized.

9: The Sequential Pattern

```
repeat
    for_all(p_1)                              ▷ Backward pass
        {
            for_all(q_1)
                Process(p_1, q_1).            ▷ Variable part.
            output(p) = result
        }
    for_all(p_2)                              ▷ Forward pass
        {
            for_all(q_2)
                Process(p_2, q_2).            ▷ Variable part.
            output(p) = result
        }
until Stability
```

The sequential pattern is considered as a step towards the design and implementation of efficient morphological operators. In morphological image processing a large number of iterations among elementary operations might be necessary to get a result. This pattern iterates until stability is reached. The two passes allow some specific operators to reduce the number of pixel involved in each operation.

10: The Queue-Based Pattern

```
for_all(p)
    if condition1 then
        enqueue(p)
        ...                                   ▷ Variable part.
    end if
while queue is empty do
    p = dequeue
    for_all(n)                                ▷ around p
        if condition2 then
            Process(p, n)                     ▷ Variable part.
            enqueue(n)
        end if
end while
```

Queue-based algorithms can be treated as a specialization of sequential implementations where the scanning order is derived from a predefined ordering relationship on the pixel intensities. The queue-based data structure improve significantly the efficiency of morphological algorithms.

2.3.2 Suitability to d'Ornellas patterns

Algorithm	Parallel(8)	Sequential(9)	Queue-based(10)
erosion/dilatation	P	S	Q
conditionnal erosion/dilation	P	S	Q
distance transform		S	Q
geodesic distance transform		S	Q
opening/closing	P	S	Q
center filter		S	
alternate sequential filtering (ASF)		S	
area opening/closing			Q
opening/closing/ASF by reconstruction		S	Q
inf/sup -reconstruction		S	Q
regional minima-maxima		S	Q
valley/peak removal		S	Q
minima imposition		S	Q
hole filling		S	Q
edge off		S	Q
opening/closing top-hat		S	
open/close by reconstruction top-hat		S	Q
ultimate erosion		S	Q
morphological skeleton	P	S	Q
sup/inf -generating (hit-miss)	P		
thinning/thickening	P		
conditionnal thinning/thickening	P		
SKIZ (SKeleton by Influence Zone)	P		
Watersheds			Q

Figure 11: Table of algorithms matching d'Ornellas pattern.

This table is a compilation of information given by d'Ornellas in his thesis (**?**), (**?**), (**?**). It shows the suitability of main morphological algorithms to d'Ornellas' patterns. We can see that many algorithms match those pattern. Furthermore a part of those algorithms can be implemented using different patterns. Algorithms matching a given pattern are similar, that's why we would to isolate redundancies in code in order to factor their implementations.

2.4 Discussion

Basic operators of mathematical morphology can be used to forge many other algorithms. Furthermore, main morphological algorithms are made following three patterns : the **Parallel** pattern, the **Sequential** pattern, the **Queue-based** pattern.
In Conclusion, mathematical morphology provides algorithms whose implementation can be factored (due to many redundancies in code). Moreover, a classification of main morphological algorithms is possible. We would like to have pieces of code able to **factor redundancies** and **provide reusable code** for algorithms based on the same pattern.

Chapter 3

Canvas for Morphological Algorithms

3.1 Definition

A canvas is a piece of code that could be use in order to write an algorithm matching a specific pattern. It makes the writing easier, especially if the canvas provides a very long piece of code. Above all, a canvas is step to factor code. This piece of code have to be fully documented in order to help the developer to find the pattern he needs. It is a code with gaps that developers fill with their specific lines of codes. Canvases are also a very good base for framework and toolkit because of the great extensibility they provide.

3.2 Experiments of Canvas implementations

The following section presents different canvas implementations. Each versions was experimented with the CC-Tarjan algorithm (Connect Components Procedure). This algorithm tags the connected components of an image using Tarjan's Union Find Algorithm. It takes a binary image and produces a new image where each connected components are numbered, that is all connected points have the number of their components. The procedure CC-Tarjan that follows uses the disjoint-set operations (UNION & FIND) to compute the connected components of an image.

The CC-Tarjan algorithm is composed of two successive passes, which means that CC-tarjan suits to the following pattern :

11: Two-pass Pattern

```
backward_iterator p1
for_all(p)
    Do Something 1
forward_iterator p2
for_all(p)
    Do Something 2
```

We need some criteria to evaluate each canvas-based implementation of CC-tarjan we are going to present. This a list of criteria we retained for this experiment.

Quality Criteria for implementation of Canvas

- Simplicity.
  Does this implementation is easy to write?
- Readability / Code Fragmentation.
  Is it easy to read?
- Coerciveness.
  Does the writing imply very different uses of presumably similar data?
- Execution time.
  Is the implementation efficient?
- Compile time.
  Is it fast to compile?
- Readable errors messages.
  Are the error messages easy to understand?
- Default Implementation.
  Does the implementation offers the possibility to do not write some missing parts of the canvas like initialisation, if there is nothing to do. Is there a default behavior for some missing part?.

Remember we want developers being able to find easily a matching pattern for their algorithm. Thereby the three main criteria we retained are **Simplicity**, **Readability** and **Coerciveness**. For each implementation we will discuss some of those points.

Compiled Code Each version of canvas-based CC-tarjan were compiled using the following piece of code. Compile and Execution time given result of a compile with the `gcc -DNDEBUG` option. This program builds a 2-dimensional image containing a lot of large connected components.

```
#include <oln/core/2d/image2d.hh>
#include <oln/core/2d/neighb2d.hh>

/* Include CC-tarjan using the $X^th$ version of canvas
implementation */
#include <oln/morpho/cc_tarjan_using_version_ $X$ .hh>

int main()
{
  using namespace oln;
  unsigned N = 1024;
  image2d<bool> img(N, N);
  for (unsigned i = 0; i < N; i++)
    for (unsigned j = 0; j < N; j++)
    {
```

```
      img.at(i, j) = 5 * cos(i) * cos(j) > 0;
    }
  morpho::cc_tarjan(img + c4);
}
```

3.2.1 Version # 0 : Reference version

Here is the evaluation of a procedural version that does not use canvases. We used it as a reference for each canvas-based implementation.

Code

```
template <typename I, typename N>
oln_plain_value(I, unsigned)
cc_tarjan_(const I& f, const N& nhb)
{
  oln_plain_value(I, unsigned) output;
  prepare(output, with, f);
  oln_plain_value(I, oln_point(I)) parent;
  prepare(parent, with, f);

  // Init
  oln_plain_value(I, bool) is_processed;
  prepare(is_processed, with, f);
  level::fill(inplace(is_processed), false);
  // First pass
  first_pass(f, parent, is_processed, nhb);
  // Second pass
  second_pass(f, parent, output);

  return output;
}
```

Pros: Usual way to write code, No surprising error message.

Cons: No Code Factored, Hard and long to write.

3.2.2 Legacy version

This version was developed in 2006 for the Olena project. It was implemented by Damien Thivolle. In this version the functor inherits the canvas (8). Despite a lot of advantage, this version had not been retained due to a poor readability. We will describe those advantages in the following section, but we can notice here that it is very difficult to understand what the canvas provides (two passes). Moreover the static inheritance make this piece of code very abstruse for common developers.

Responsibility The canvas object possess the bare minimum information it needs to be executed, including processing, type and data information. Thought the implementation is different, we will see further that the concept we retained is very similar.

Code

```
template <typename I, typename Exact>
struct two_pass : public virtual Any<Exact>
{
  void first_pass_body(const oln_point(I)& p)
  { exact(this)->impl_first_pass_body(p); }
  void second_pass_body(const oln_point(I)& p)
  { exact(this)->impl_second_pass_body(p); }
  // Concrete method.
  void run()
  {
    for_all(p1)
    first_pass_body(p1);
    for_all(p2)
    second_pass_body(p2);
  }
  protected:
  // Ctor.
  two_pass(const Image<I>& f) :
  p1(f.points()),
  p2(f.points()) {}
  oln_bkd_piter(I) p1;
  oln_fwd_piter(I) p2;
};
```

Pros: Efficient. Natural concept in OOP.

cons: Poor readability (because of static inheritance). Very Fragmented pieces of codes.

3.2.3 Version # 1

In this version, **Data**, **Type** and $process$ are owned by a functor (fun) but not **I** nor **ima**.

Responsibility

- Canvas
  - **owns :**
    p_1, p_2, **I**, **ima**
  - **needs :**
    Process1, **Process2**
- Functor
  - **owns :**
    Process1, **Process2**

Pseudo-Code

```
procedure TWO-PASS( I ima, F fun )
    p1 needs I ima
    for_all(p1)
        fun.Process1(p1, ima)
    ...
end procedure
```

Pros: We have a canvas and it is readable enough.

Cons: Some algorithms use extra data but how to use them here? Should we give those extra data like **ima** (directly to the canvas)?

3.2.4 Version # 2

The first version did not provide the possibility to use extra data. In this version, we add an argument to the canvas : $auxdata$. $auxdata$ hold all extra data needed by the processing. Finally, $auxdata$, **Data** and **Type** information are given to the canvas. Processing are still held by fun, which is a canvas argument too.

Responsibility

- Canvas
  - **owns :**
    p_1, p_2, **auxdata**, **Aux**, **I**, **ima**
  - **needs :**
    Process1, **Process2**
- Functor
  - **owns :**
    Process1, **Process2**

Pseudo-Code

```
procedure TWO-PASS( I ima, F fun, D auxdata )
    p1 I ima
    for_all(p1)
        fun.Process1(p1, ima, auxdata)
    ...
end procedure
```

Pros: **Auxdata** are clearly separated of other data.

Cons: Strange version if no **AuxData** : Users have to give an empty struct as canvas argument. Moreover, this version is not very coercive regarding access to information: **ima** and **auxdata** are both data but are used singly.

3.2.5 Version # 3 : Retained version

In Version 2, we have strange behavior, if there is no extra data. An empty structure **auxdata** is given to the canvas. Furthermore **ima** and **auxdata** are both data, a generalized implementation of canvas must have the same behavior regarding the type of information it use. In this version a single structure, the functor fun provides type, data and processing information.

Responsibility

- Canvas
  - **owns :**
    p_1, p_2
  - **needs :**
    I, **ima**, **Process1**, **Process2**
- Functor
  - **owns :**
    auxdata, **Aux**, **I**, **ima**, **Process1**, **Process2**

Pseudo-Code

```
procedure TWO-PASS(F fun)
    p1 F::I fun.ima
    for_all(p1)
        fun.Process1(p1)                    ▷ No useless argument
    ...
end procedure
```

Pros: Very readable, Coercive : a single entity holds information.

Cons: Classic type, like **I**, must be defined by `typedef} (held by the functor)`.

3.3 Discussion

Compile and Execution Times The time of compile and execution is very important. The advantage supplied by canvas must not imply lack of efficiency or very long compile time. The results that follow was performed on an AMD XP3000+ under the operating system Debian. We used the simple software *Time* to measure the different presented times (See annexe .2 for the meaning of each options). Every times are written in second. Those results are the average of ten experiments[1].

Compile Time	-DNDEBUG	-DNDEBUG -O3	-DNDEBUG -O2	-DNDEBUG -O1	-DNDEBUG -Os
version 0	5,76	10,04	9,15	7,73	8,27
version 1	5,86	11,32	10,2	8,5	8,81
version 2	5,84	10,77	9,66	8,19	8,83
version 3	5,89	10,75	9,61	7,98	8,78
version 4	5,81	10,8	9,64	8,24	8,72

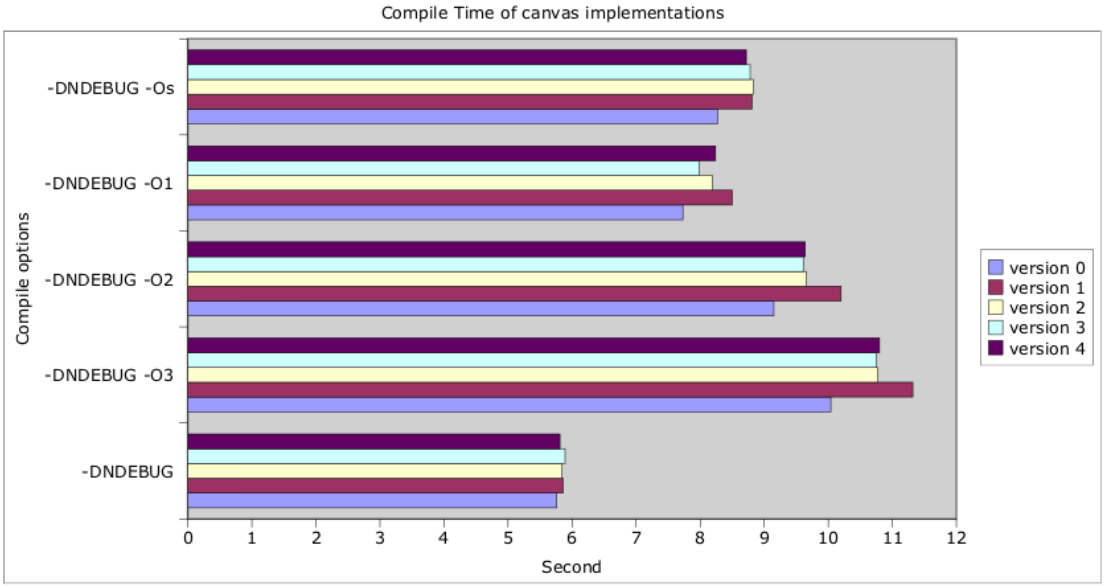

Figure 12: Compile time.

[1]This experiment is automatised as a shell script, which is available on the olena trac repositories (http://trac.olena.epita.fr/).

Execution Time	-DNDEBUG	-DNDEBUG -O3	-DNDEBUG -O2	-DNDEBUG -O1	-DNDEBUG -Os
version 0	5,59	1,02	2,58	3,1	2,31
version 3	5	0,91	2,35	2,52	2,02
version 1	5,75	1,05	2,71	3,1	2,28
version 2	5,33	0,97	2,48	3	2,15
version 4	5,4	0,94	2,36	2,53	2,02

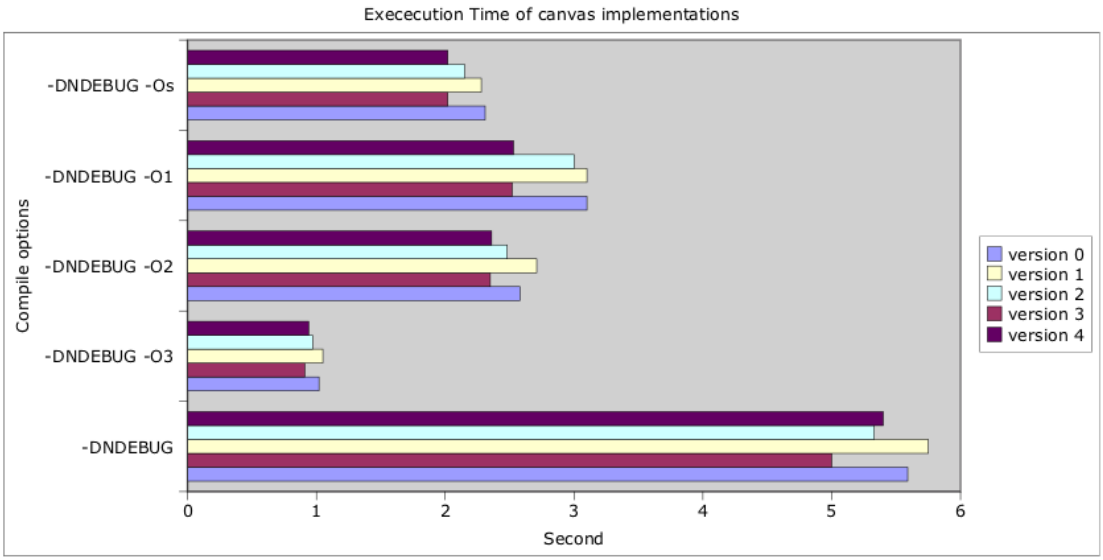

Figure 13: Execution time.

We can see that the five versions do not present really significant compile and execution time differences (about 5% of differences). Meanwhile the retained version (version 3) have always the best execution time.

Version retained The piece of code that follows is the implementation retained for canvas in Olena 1.0. It is based on the **third** version we present. This version is coercive, does not decrease performance, and is very readable.

Canvas implementation in Olena 1.0.

```
template typename F void two_pass(F&fun)
{
  oln_bkd_piter(F::I) p1(fun.f.points());
  for_all(p1)
    fun.first_pass_body(p1);
  oln_fwd_piter(F::I) p2(fun.f.points());
  for_all(p2)
    fun.second_pass_body(p2);
}
```

In this version, the data information must be define as fun's attributes, the type with *typedef* in fun. The processing are methods of fun (example of use in annexe .1).

Chapter 4

Conclusion

Mathematical morphology is part of image processing. The main morphological operators are based on three patterns, the sequential, the parallel and the queue-based pattern. Moreover, some basic algorithms are often used by others. Canvases enable developers to factor this redundancy.

Using canvas has a lot of advantages as long as they are readable and easy to use. First, a canvas is a more or less large piece of code, and using it can improve the development time. Then, a canvas is a reusable and extensible piece of code. Because of its nature (code with gaps), we can extend a canvas as long as we can imagine a new algorithm that matches the canvas.

The first fundamental requirement any canvas-based implementation should have is good theoretical base. The patterns of the area of development should be correctly identified.

The second fundamental requirement is very readable and documented canvases. There is no interest in writing canvases if nobody can find, understand and use those canvases.

This report try to show a good implementation of canvas. We made the choice to give to the canvas the bare minimum of information. The access to information is also coercive and simple. Thereby, any other information must be owned by the functor. This functor must be fully under the responsibility of the user in order to do not obfuscate any important part of the implementation of the canvas. This solution seems to provide readable, coercive and usable canvases.

Chapter 5

Bibliography

[1] J. Serra. *Image Analysis and Mathematical Morphology*. Academic Press, London, 1982.

[2] G. Matheron. Elements pour une théorie des milieux poreux. *Masson*, Paris, 1967.

[3] M. Ornellas. Algorithmic Patterns for Morphological Image Processing. *UVA*, ASCI, 2001.

[4] M. Ornellas. A Parallel Algorithmic Pattern. *sugarloafplop*, PIGS/CCC/UFSM, LACESM/UFSM - INPE/UFSM, 2002.

[5] M. Ornellas. A Sequential Algorithmic Pattern. *sugarloafplop*, PIGS/CCC/UFSM, LACESM/UFSM - INPE/UFSM, 2004.

[6] M. Ornellas. A Queue-based Algorithmic Pattern. *sugarloafplop*, PIGS/CCC/UFSM, LACESM/UFSM - INPE/UFSM, 2002.

[7] G. Darbon, T. Geraud, A. Duret-lutz. Generic implementation of morphological image operators. *ISMM*, Sciro Publishing, 2002.

[8] D. Thivolle. Canvas in Olena. https://trac.lrde.org/olena/

Index

Chapter 6

Annexe

.1 How to use our Canvas implementation proposal?

```
// Load the two-pass canvas.
#include <oln/canvas/two_pass.hh>



// Define the class of the functor.
struct cc_tarjan
{

  // Type Information.
  typedef oln::Image2d<int> I;


  // Data Information.
  const I& ima;


  // Process Information.
  void first_pass_body(const oln_point(I)& p) { ... }
  void second_pass_body(const oln_point(I)& p) { ... }


  cc_tarjan_(const I& f)
  : ima(f)
  {
  }

}
```

```
// Use cc-tarjan on a 2d image.
int main()
{

  // Declare the image.
  oln::Image2d<int> ima(50, 50);

  // Init ima.
  ...


  // Declare the functor.
  cc_tarjan fun(ima);


  // Execute cc-tarjan using the two-pass canvas.
  oln::canvas::two_pass(fun);

}
```

.2 Optimization under *gcc*

	Included in Level			
Optimization	-O1	-O2	-Os	-O3
defer-pop	●	●	●	●
thread-jumps	●	●	●	●
branch-probabilities	●	●	●	●
cprop-registers	●	●	●	●
guess-branch-probability	●	●	●	●
omit-frame-pointer	●	●	●	●
align-loops	○	●	○	●
align-jumps	○	●	○	●
align-labels	○	●	○	●
align-functions	○	●	○	●
optimize-sibling-calls	○	●	●	●
cse-follow-jumps	○	●	●	●
cse-skip-blocks	○	●	●	●
gcse	○	●	●	●
expensive-optimizations	○	●	●	●
strength-reduce	○	●	●	●
rerun-cse-after-loop	○	●	●	●
rerun-loop-opt	○	●	●	●
caller-saves	○	●	●	●
force-mem	○	●	●	●
peephole2	○	●	●	●
regmove	○	●	●	●
strict-aliasing	○	●	●	●
delete-null-pointer-checks	○	●	●	●
reorder-blocks	○	●	●	●
schedule-insns	○	●	●	●
schedule-insns2	○	●	●	●
inline-functions	○	○	○	●
rename-registers	○	○	○	●