Climb - A Generic and Dynamic Approach to Image Processing

Christopher Chedeau

Technical Report n^o1001, Juin 2010
revision 2187

Climb is a generic image processing library. A case study of the erosion algorithm from mathematical morphology highlights the issues of a non-generic implementation. Structures such as Image, Site Set and Accumulator are defined to solve them.

Genericity is even increased by the concept of Morphers: a way to alter the communication between an object and the outside world. All these additions are done using the dynamic aspect of Lisp that allows for rapid prototyping.

Climb est une bibliothèque de traitement d'image générique. Une étude de l'algorithme d'érosion provenant du domaine de la morphologie en mathématiques met en évident les problèmes liés à une implémentation non générique. Des structures telles que les Image, Ensemble de Sites et Accumulateurs sont introduites pour les résoudre.

La généricité est également augmentée grâce concept de Morphers: une façon d'altérer les communications entre un objet et le monde extérieur. Toutes ces additions reposent sur l'aspect dynamique de Lisp qui permet un prototypage rapide.

Keywords
Image Processing, Climb, Morpher, Genericity, Structuring Element

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

1 Introduction **4**
1.1 Acknowledgements 4

2 Case Study **5**
2.1 Problem 5
2.2 Image Structure 6
2.3 Neighborhood 7
2.4 Algorithm 8
2.5 Solution 8

3 Morphers **10**
3.1 Value Morpher 11
3.1.1 Definition 11
3.1.2 Image Morphers 11
3.2 Content Morpher 13
3.2.1 Definition 13
3.2.2 Site Set Morphers 14
3.3 Combination 15
3.3.1 Definition 15
3.3.2 Erosion Algorithm 16
3.3.3 Top Hat Algorithm 17

4 Conclusion **18**

5 Bibliography **19**

Chapter 1

Introduction

In software engineering, genericity is an important notion as it allows to process different types of data while writing each algorithm only once. Image processing takes a great advantage of genericity as it brings different points of view (and thus various types of algorithms) on diverse types of data.

The C++ library Olena[1] developed at the LRDE[2] (EPITA[3] Research and development laboratory) provides a very powerful model to implement generic image processing. It benefits from meta-programming to perform compile-time dispatches and checks. It gives good performance with all the power of genericity. It also inherits from the major C++ drawbacks: its heavy syntax and slow compilation to perform static operations.

The goal of Climb (Denuzière, 2009) is to provide a library with the same generic power as Olena, however focusing on interactivity and user-friendly syntax. The Common Lisp language provides the necessary dynamism and customizability. Its high flexibility lets us consider several implementations of the model provided by Olena.

The article is divided in two sections guided by a case study: the erosion algorithm from mathematical morphology. The example highlights the issues of a non-generic implementation. New structures are added to solve them and those structures are even improved by the concept of Morphers.

1.1 Acknowledgements

Thanks to Didier Verna for being both a great teacher and project manager, Loic Denuziere for the initial Climb project, Thierry Geraud for Olena and his presentations of Image Processing, Lorene Poinsot and Alex Hamelin for the help during the preparation of this article.

[1]http://olena.lrde.epita.fr
[2]http://www.lrde.epita.fr
[3]http://www.epita.fr

Chapter 2

Case Study

2.1 Problem

The erosion is a basic operation in mathematical morphology (Soille, 1999). It can be expressed as a simple sentence: "All the pixels are replaced by the minimum value of the pixels around.". The Figure 2.1 is a common implementation of the erosion algorithm for 2D binary images.

```
/* Loop through all the pixels of the image */
for (x = 0; x < img_width; ++x)
  for (y = 0; y < img_height; ++y)

    /* Initialize the output image */
    new_image[x][y] = TRUE;

    /* Loop through all the valid pixels of the neighborhood */
neighborhood:
    for (i = -neighb_width; i <= neighb_width; ++i)
      for (j = -neighb_height; j <= neighb_height; ++j)
        if ((x + i >= 0 && x + i < img_width) &&
            (y + j >= 0 && y + j < img_height))

          /* Algorithm */
          if (old_image[x + i][y + j] == FALSE)
            new_image[x][y] = FALSE;
            break neighborhood;
```

Figure 2.1: Simple Erosion Algorithm

However it is not generic enough for several reasons:

- Image structure: It only handles images that are based on a regular 2D grid that starts at the coordinates (0, 0).
- Neighborhood: The neighborhood can be of any shape, not only a full rectangle.
- Algorithm: The code structure is the same for many algorithms, only a small part defines the erosion algorithm.
- Image value: The erosion algorithm is not restricted to binary images, this code is.

The goal of this article is to find a way to split this algorithm into different pieces.

2.2 Image Structure

Pixels of an image are usually represented aligned on a regular 2D grid. This is not the only structure that exists, values can be placed on an hexagonal grid, a 3D grid, a custom grid represented by a graph or even a simplicial complex. The underneath structure of the image has nothing to do with the erosion algorithm. Therefore an abstraction has to be established for the image structure.

An image is traditionnaly represented by a set of pixels. In this case, a pixel holds two meanings: the position and the value. Those two concepts do not fit well together when working on generic images. In this article, an image is going to be defined as a function that takes a site and return a value.

$$Image : Site \rightarrow Value$$

The underlying structure of the image is determined by the type of Site. A point is used in a regular n-dimensional grid, a node or edge for a graph structure and so on. All the sites of the image are grouped in a Site Set called its Domain.

Once created, the only way to maninupulate a site set is to iterate on it. The interface has been written to be as light as possible. This will make wrapping easier later in the article.

- Site Set - reset: Restore the internal pointer at the beginning of the Site Set.
- Site Set - next: Get the next site of the set. Nil if it has reached the end of the set.

The most common use of a site set is to iterate over all the elements. A do-sites macro (Figure 2.2) is provided as a helper for this task.

```
(do-sites (site image)
  (format t "~A␣->␣~A~%" site (iref image site)))
```

Figure 2.2: Iterate over all points of an image

2.3 Neighborhood

In mathematical morphology, a structuring element is a shape describing the relation between a pixel and its environment. The choice of the structuring element is important as it changes the result of the algorithm. In a regular grid on the 2D space, there are two main structuring elements representing the 4-connexity and 8-connexity (Figure 2.3). It is common to use other structuring elements when looking for a special pattern such as an edge or a corner.

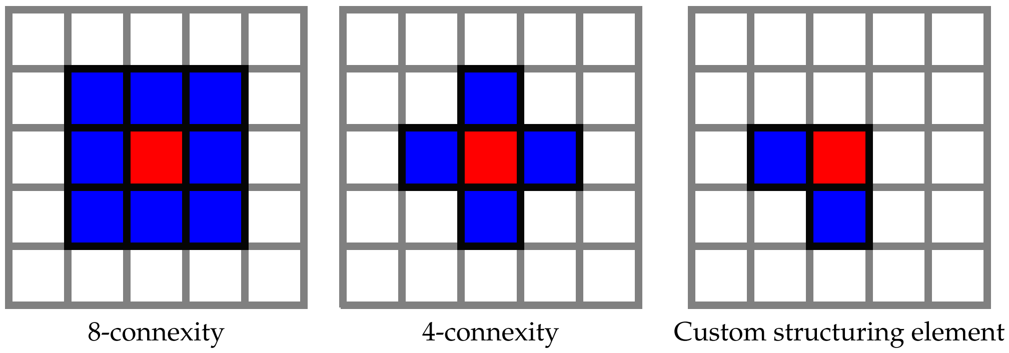

Figure 2.3: Different structuring elements

The structuring element is represented by the site set concept previously defined. The Box is the only implementation of site set for points (sites in a regular grid). It's a filled rectangle (in 2D) between two points and perfectly fits for the 8-connexity. It becomes a 4-connexity structuring element when a binary mask is applied on top.

$$Mask\left(\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, Box\begin{pmatrix} -1 \\ -1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix}\right)$$

All there is left to do is to center the structuring element around each site as represented in the Figure 2.4.

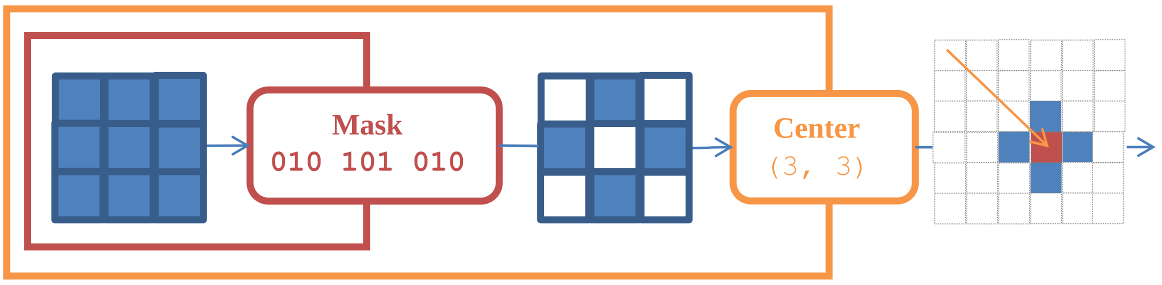

Figure 2.4: Structuring element implementation

2.4 Algorithm

So far it is possible to iterate through all the sites of the image and find the respective neighbors.
The accumulator concept is introduced to make the actual process generic. An accumulator is an object that takes multiple sequential inputs and computes a result from these. Instead of using the common reduce design in functionnal languages, accumulators have been chosen over functions as they are real objects, thus its possible to change their implementation through properties (Denuzière, 2010).

- Accumulator - take: Give a value to the accumulator
- Accumulator - result: Compute a value from all the given elements
- Accumulator - reset: Restore the accumulator to an empty state

Since accumulators are replacing lambda functions in the context of the reduce function, a family of accumulator, the reduce-accus, are there to simulate it. An example of use is the Min accumulator (Figure 2.5).

```
(defun min-accu ()
  (make-instance 'reduce-accu
                 :function (lambda (a b) (min a b))
                 :initial-value 255))
```

Figure 2.5: Min Accumulator

2.5 Solution

Let us define two small helper functions:

- (image-map image function): Create a new image where each value is the result of the function applied on the associated site of the image.
- (accu-reduce site-set accu): Give all the sites of the set to the accumulator and return the computed result.

All the pieces are now available to make a generic erosion algorithm (Figure 2.6).

```
(defun erosion (img neighb) :
  (image-map img            : /* Loop through all the pixels of the image */
                            : for (x = 0; x < img_width; ++x)
                            :   for (y = 0; y < img_height; ++y)
                            :
                            :     /* Initialize the output image */
                            :     new_image[x][y] = TRUE;
                            :
    (accu-reduce neighb     :     /* Loop through all the valid pixels of the neighborhood */
                            : neighborhood:
                            :     for (i = -neighb_width; i <= neighb_width; ++i)
                            :       for (j = -neighb_height; j <= neighb_height; ++j)
                            :         if ((x + i >= 0 && x + i < img_width) &&
                            :             (y + j >= 0 && y + j < img_height))
                            :
      (min-accu))))         :           /* Algorithm */
                            :           if (old_image[x + i][y + j] == FALSE)
                            :             new_image[x][y] = FALSE;
                            :             break neighborhood;
```

Figure 2.6: Erosion Algorithm

The final implementation is very elegant as it is really close to the erosion definition: "All the pixels are replaced by the minimum value of the pixels around". It is not clobbered by implementation details and therefore is easily readable. It solves all the generic issues of the first version:

- Image structure: The algorithm only interact with the image through the image-map function. Therefore it is going to work on any object that implements the image-map function. There is no more implementation detail inside the algorithm.
- Neighborhood: The same way, the neighborhood is hidden behind the neighb site set that must be able to be centered on every site of the image.
- Algorithm: The min-accu is now responsible of the computation. This part can now be re-used in other algorithms and make the structure more apparent.
- Image value: All the values the algorithm accepts are all the values the min-accu handles.

It is also good concerning the software development process. The code is split into small different functions that are easily unit testable. Adding new value types or structures is not going to change the way the algorithm is written. Once an algorithm is written, it is going to work on all the new images that implements the required operations.

Chapter 3

Morphers

The previous chapter presented how to design a generic erosion algorithm, however it did not tell how to implement the various concepts (Site set, Image and Accumulator) in a generic way. The solution adopted in Climb is an extensive use of the Morpher concept introduced in Olena (Ballas, 2008) with SCOOP2 (Géraud and Levillain, 2008).

A morpher is a wrapper around an object that modifies how the object is seen by the outside world. It works is by tampering communications in order to add new behaviors to the object. It can be seen as an extension of the decorator design pattern.

In Olena, the concept of morphers is only defined for images. The goal is to generalize it to any object with a defined communication protocol. In Climb, a distinction is made between Image and Site Set, and Accumulators become good candidates for morphers.

Any message transformation can be considered as a being morpher. Two big categories of message transformations have been identified as providing new features. Those are the Value Morpher and Content Morpher.

3.1 Value Morpher

3.1.1 Definition

A value morpher is altering the values that move through the communication channel. A value in this context is an argument of the object method. It is a simple transformation as

The main use is to have a different view of the object in a non-intrusive way. For example view a color image as a grayscale one by altering the function that read the value of a site of an image.

It can be implemented on the three concepts defined so far: Image, Site Set and Accumulator (Figure 3.1).

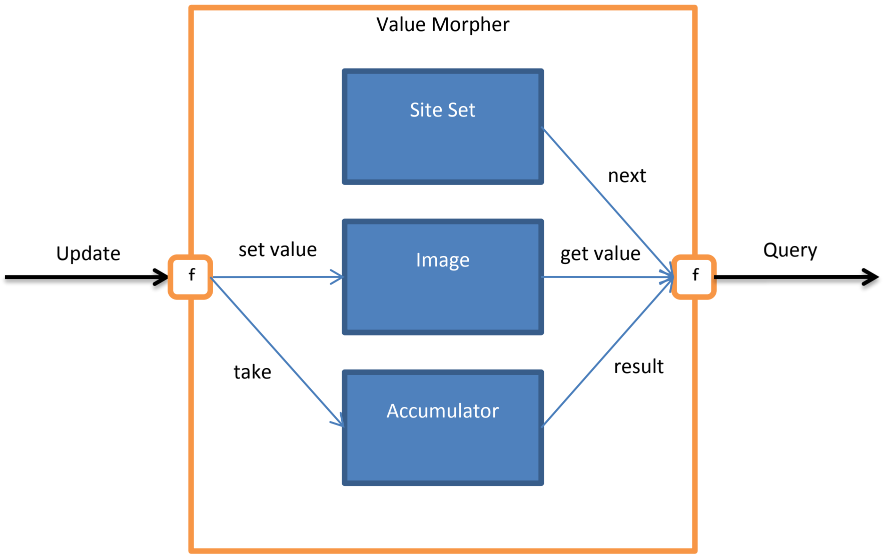

Figure 3.1: Value Morpher

3.1.2 Image Morphers

An image is an object that associates sites with values. To use it there are two primitives: a getter and a setter. An image value morpher is nothing more than a new getter and a new setter (Figure 3.2). It gives the power to change the input and outputs and add new behaviors.

```
(make−value−morpher−image
  image
  (lambda (value image site) ... Setter ...)
  (lambda (image site) ... Getter ...))
```

Figure 3.2: Image Value Morpher Creation

It is now extremly easy to write a basic image morpher that isolates the nth component of an value-vector image (RGB for example) (Figure 3.2). The setter update the nth element of the vector and the getter returns it.

```
(defun image-morph-component (image n)
  (make-value-morpher-image
    image
    (lambda (value image site)        ; Setter
      (setf (aref (iref image site) n) value))
    (lambda (image site)              ; Getter
      (aref (iref image site) n))))

; Red channel of a RGB image
(image-morph-component rgb-image 0)
```

Figure 3.3: Simulate an image composed of the nth component

The getter and setter are being called with the image and site information. This gives flexibility but is clobbering the code when the morpher is working directly on the values. The getter and setter of a direct value morpher (Figure 3.4) take a value and return a modified one. Writing the access to the underlying image is no longer required.

```
(make-direct-value-morpher-image
  image
  (lambda (value) ... Setter ...)
  (lambda (value) ... Getter ...))
```

Figure 3.4: Direct Morpher - It works directly on values

In the Hit or Miss algorithm, an image with all its values negated is needed. There are two ways to write it down (Figure 3.5). The first way that comes in mind is to create a new image with the same structure and negate all the values. This solution may not be acceptable as it doubles the required space. Another one consists in applying a morpher ontop of the image. There is one physical image, but two ways to see it.

```
; Image negation with full copy
(defun image-not (image)
  (image-map image (lambda (x) (not x))))

; Image negation using morpher
(defun image-morph-not (image)
  (make-direct-value-morpher-image
    image
    (lambda (x) (not x))    ; Setter
    (lambda (x) (not x))))  ; Getter
```

Figure 3.5: Negation of an image

3.2 Content Morpher

3.2.1 Definition

A content morpher is altering the structure of the object. Applied on a set it changes how the element are being sent to the outside world. Three main types of transformations are possible.

- Restriction (Figure 3.6): Some elements are removed from the original set. It is used to work on a sub part of an image by restricting its domain, or to obtain a non-full structuring element from a box.
- Addition: New elements are added. It can be used to simulate a mirror border on the edge of an image.
- Order: The order of the elements may be changed, a typical use is to sort the set.

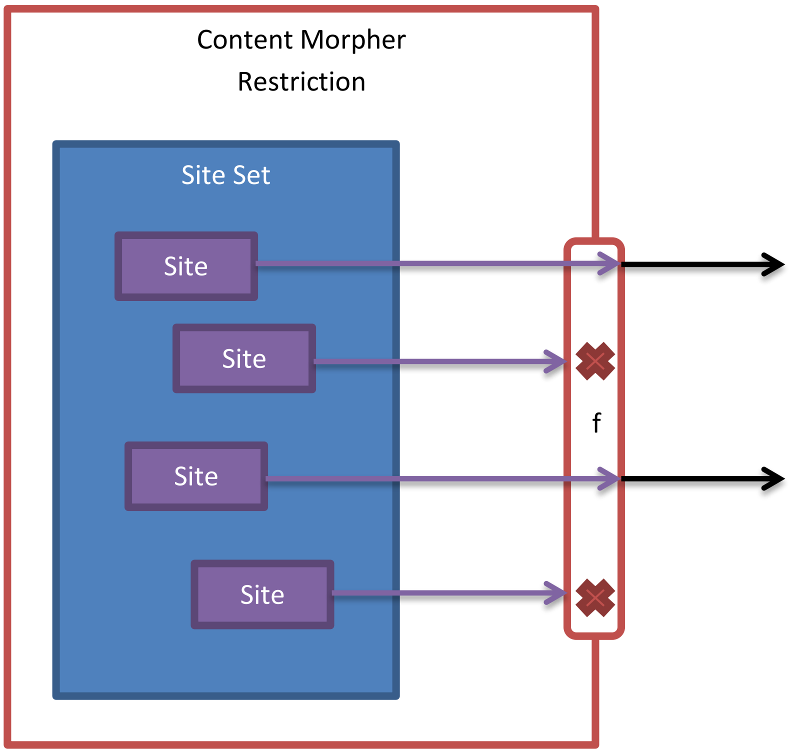

Figure 3.6: Content Morpher using Restriction

3.2.2 Site Set Morphers

The 4-connexity has been previously defined as being a 8-connexity structuring element masked with a binary vector. It is implemented thanks to multiple morphers (Figure 3.8). First a content morpher, the site-set-next operation now skips all the elements that do not match the condition. To create such a morpher, a function that tells wether if the element is valid is required and some work may need to be done when the site set is being resetted (Figure 3.7).

```
(make-restriction-content-morpher-site-set
   site-set
   (lambda (site))            ; Is valid?
  &optional (lambda ())))     ; Reset
```

Figure 3.7: Prototype of a Site Set Content Restriction Morpher

Then, a value morpher is used to translate the site set around the current point.

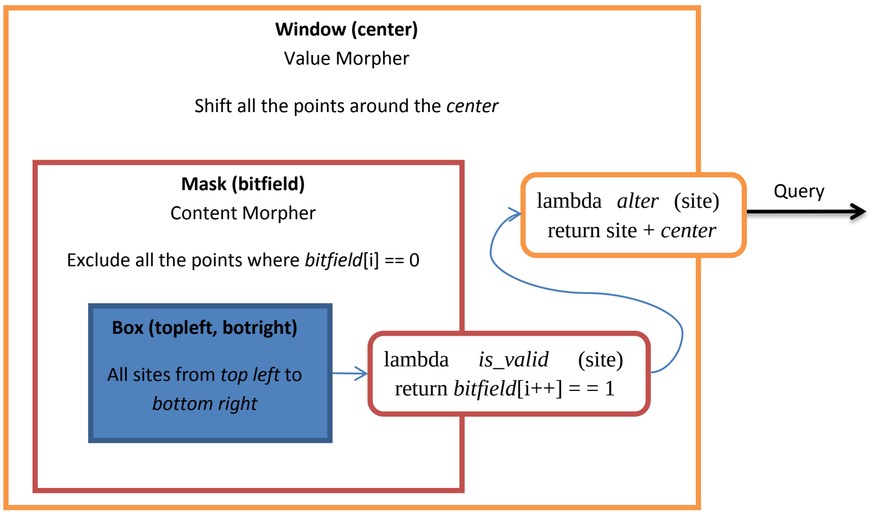
Window (center)
Value Morpher
Shift all the points around the center
Mask (bitfield)
Content Morpher
Exclude all the points where bitfield[i] == 0
Box (topleft, botright)
All sites from top left to bottom right
lambda is_valid (site)
return bitfield[i++] = = 1
lambda alter (site)
return site + center
Query

Figure 3.8: Neighborhood

3.3 Combination

3.3.1 Definition

Morphers are not dependent of the object they are wrapping. Therefore once a communication protocol has been set up, you can combine one concrete object with as many morphers as you want implementing that protocol (Figure 3.9). Such organization allows you to write really small morphers and combine many of them to produce complex behavior. Each morpher is easy to unit test and the combination of morphers is less subject to bugs. The morpher design pattern allows you to write small objects, which are easily testable, with a high degree of combination. It follows the jQuery tagline: « Write less, Do more ».

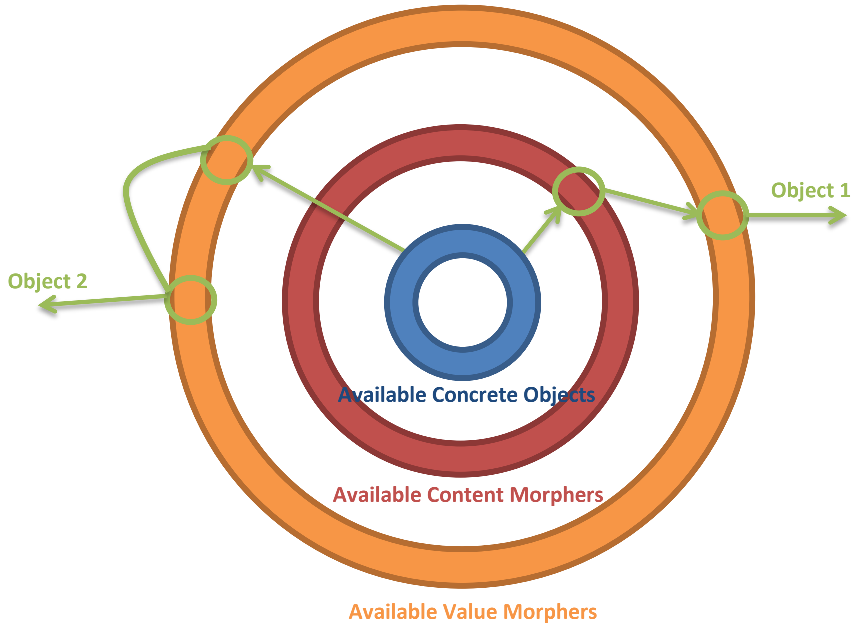

Figure 3.9: Final Object = Combination of a Concrete Object, Content Morphers and Value Morphers

Algorithms are written by combining pieces together.

- Concrete Objects: Specific implementation of an object.
- Algorithm: Set of rules to solve a problem that may be using other objects and algorithms.
- Morpher Objects: Transform the concrete objects to make them fit the algorithm.

The genericity lies in the fact that all the pieces are independent and pluggable in many different ways. The erosion algorithm is a good example. It makes a new image where each pixel is the application of the "and" operation on every nearby pixel of the source image.

3.3.2 Erosion Algorithm

To conclude this study, the full implementation of the erosion algorithm is now available represented by a schema (Figure 3.10). It includes many concrete objects and algorithms along with morphers to make them fit together.

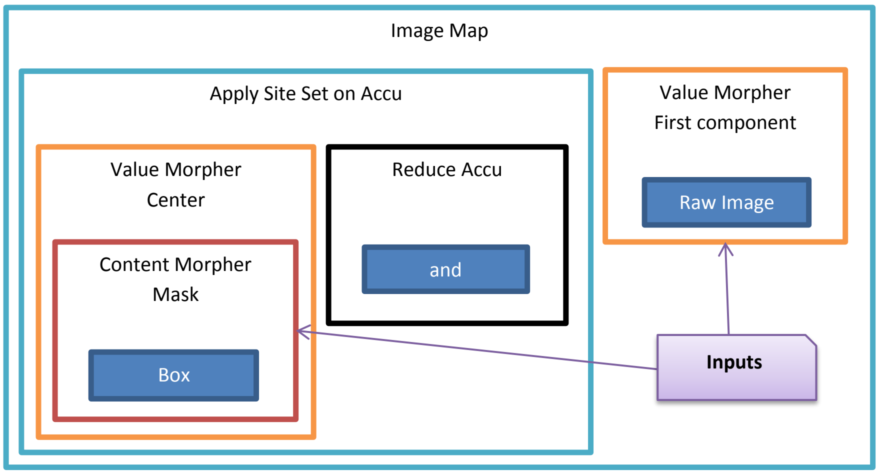

Figure 3.10: Erosion Algorithm

- Concrete: Base objects that are not generic.
  - Box: Site set returning all the sites between the start to the end point.
  - Raw Image: Image stored in a vector.
  - And: Function that does a logic and between the two arguments.
- Morphers: Transform the concrete objects to make them fit the problem.
  - Value Morpher, First component: The raw image use vectors as value, it is used to view only the first component of the vector.
  - Content Morpher, Mask: The box gives all the sites from one point to another, a content morpher is used as a mask to keep only the required values.
  - Value Morpher, Center: A value morpher is then applied on the site set to center it on the current pixel.
- Algorithms: Do action with the objects.
  - Apply Site Set on Accu: Insert all the sites stored in the site set inside the accumulator and returns the result of the accumulator.
  - Image Map: Apply the function on all the pixels of the given image.

3.3.3 Top Hat Algorithm

Algorithms can be used as new bricks to build more complex algorithms. Top Hat algorithm (Figure 3.11) uses image combine with the concrete function minus on the closing and opening algorithms. They are both made with the erosion brick built previously.

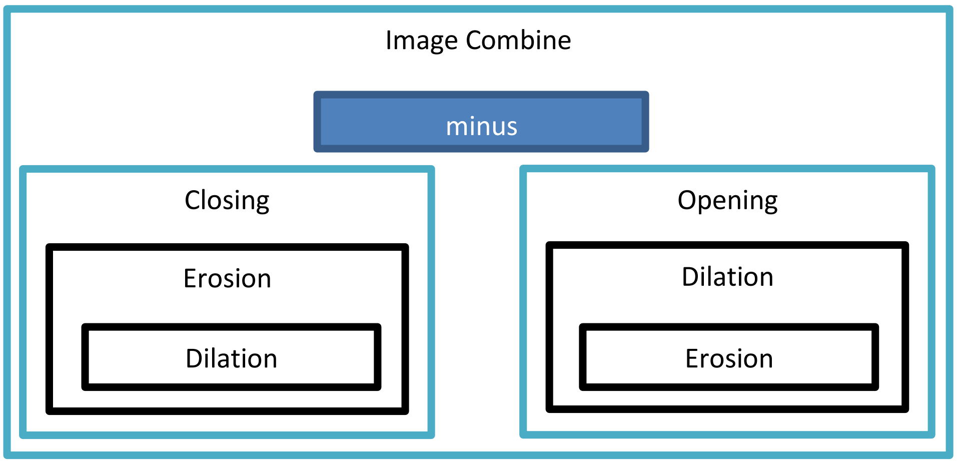

Figure 3.11: Top Hat Algorithm

Many algorithms can be implemented combining together all the bricks defined so far. The following list are present in Climb.

- Erosion
- Dilation
- Hit or Miss
- Opening
- Closing
- Top Hat
- Thinning
- Gradient
- Laplacian

Chapter 4

Conclusion

The erosion algorithm is easy to write in a specific version but requires the addition of many structures such as Image, Site Set and Accumulators to become generic. The algorithm is now written once and the addition of new image types requires localized modifications.

The structures are given genericity through the concept of Morphers. By tampering the communications of an object, it is possible to extend its functionnalities in a non intrusive and transparent way. It gives the ability to transform specific objects to make them fit specific algorithms without memory loss. Olena introduced the Morpher concept for the images, Climb hopes to make it viable for everything else.

The functional aspect of Lisp helps to produce a consise code through the separation in of small functions and built-in lambda expressions. In addition with the Lisp dynamic development environment it makes prototyping fast and easy.

The future work consists mainly in the addition of more image types, algorithms and morpher integration. The performance is a major issue in the current implementation and needs a deep reflexion.

Chapter 5

Bibliography

Ballas, N. (2008). Image taxonomy in milena. Technical report, EPITA Research and Development Laboratory (LRDE).

Denuzière, L. (2009). CLIMB: A dynamic approach to generic image processing. Technical report, EPITA Research and Development Laboratory (LRDE).

Denuzière, L. (2010). Property-based genericity: A dynamic approach. Technical report, EPITA Research and Development Laboratory (LRDE).

Géraud, T. and Levillain, R. (2008). Semantics-driven genericity: A sequel to the static c++ object-oriented programming paradigm (scoop 2). In *6th International Workshop on Multi-paradigm Programming with Object-Oriented Languages*.

Soille, P. (1999). *Morphological Image Analysis - Principles and Applications*. Springer-Verlag.

List of Figures

2.1 Simple Erosion Algorithm 5
2.2 Iterate over all points of an image 6
2.3 Different structuring elements 7
2.4 Structuring element implementation 7
2.5 Min Accumulator 8
2.6 Erosion Algorithm 9

3.1 Value Morpher 11
3.2 Image Value Morpher Creation 11
3.3 Simulate an image composed of the nth component 12
3.4 Direct Morpher - It works directly on values 12
3.5 Negation of an image 12
3.6 Content Morpher using Restriction 13
3.7 Prototype of a Site Set Content Restriction Morpher 14
3.8 Neighborhood 14
3.9 Final Object = Combination of a Concrete Object, Content Morphers and Value Morphers 15
3.10 Erosion Algorithm 16
3.11 Top Hat Algorithm 17