Generic Image Processing With Climb

Laurent Senta
senta@lrde.epita.fr

Christopher Chedeau
christopher.chedeau@lrde.epita.fr

Didier Verna
didier.verna@lrde.epita.fr

Epita Research and Development Laboratory
14-16 rue Voltaire, 94276 Le Kremlin-Bicêtre
Paris, France

ABSTRACT

We present Climb, an experimental generic image processing library written in Common Lisp. Most image processing libraries are developed in static languages such as C or C++ (often for performance reasons). The motivation behind Climb is to provide an alternative view of the same domain, from the perspective of dynamic languages. More precisely, the main goal of Climb is to explore the dynamic way(s) of addressing the question of genericity, while applying the research to a concrete domain. Although still a prototype, Climb already features several levels of genericity and ships with a set of built-in algorithms as well as means to combine them.

Categories and Subject Descriptors

I.4.0 [**Image Processing And Computer Vision**]: General—*image processing software*; D.2.11 [**Software Engineering**]: Software Architectures—*Data abstraction, Domain-specific architectures*

General Terms

Design, Languages

Keywords

Generic Image Processing, Common Lisp

1. INTRODUCTION

Climb is a generic image processing library written in Common Lisp. It comes with a set of built-in algorithms such as erosion, dilation and other mathematical morphology operators, thresholding and image segmentation.

The idea behind genericity is to be able to write algorithms only once, independently from the data types to which they will be applied. In the Image Processing domain, being fully generic means being independent from the image formats, pixels types, storage schemes (arrays, matrices, graphs) *etc.* Such level of genericity, however, may induce an important performance cost.

In order to reconcile genericity and performance, the LRDE[1] has developed an Image Processing platform called Olena[2]. Olena is written in C++ and uses its templating system abundantly. Olena is a ten years old project, well-proven both in terms of usability, performance and genericity [2].

In this context, the goal of Climb is to use this legacy for proposing another vision of the same domain, only based on a dynamic language. Common Lisp provides the necessary flexibility and extensibility to let us consider several alternate implementations of the model provided by Olena from the dynamic perspective.

First, we provide a survey of the algorithms readily available in Climb. Next, we present the tools available for composing basic algorithms into more complex Image Processing chains. Finally we describe the generic building blocks used to create new algorithms, hereby extending the library.

2. FEATURES

Climb provides a basic image type with the necessary loading and saving operations, based on the lisp-magick library. This allows for direct manipulation of many images formats, such as PNG and JPEG. The basic operations provided by Climb are loading an image, applying algorithms to it and saving it back. In this section, we provide a survey of the algorithms already built in the library.

2.1 Histograms

A histogram is a 2D representation of image information. The horizontal axis represents tonal values, from 0 to 255 in common grayscale images. For every tonal value, the vertical axis counts the number of such pixels in the image. Some Image Processing algorithms may be improved when the information provided by a histogram is known.

Histogram Equalization. An image (*e.g.* with a low contrast) does not necessarily contain the full spectrum of intensity. In that case, its histogram will only occupy a narrow band on the horizontal axis. Concretely, this means that some level of detail may be hidden to the human eye.

[1]EPITA Research and development laboratory
[2]`http://olena.lrde.epita.fr`

Histogram equalization identifies how the pixel values are spread and transforms the image in order to use the full range of gray, making details more apparent.

Plateau Equalization. Plateau equalization is used to reveal the darkest details of an image, that would not otherwise be visible. It is a histogram equalization algorithm applied on a distribution of a clipped histogram. The pixel values greater than a given threshold are set to this threshold while the other ones are left untouched.

2.2 Threshold

Thresholding is a basic binarization algorithm that converts a grayscale image to a binary one. This class of algorithms works by comparing each value from the original image to a threshold. Values beneath the threshold are set to `False` in the resulting image, while the others are set to `True`.

Climb provides 3 threshold algorithms:

Basic. Apply the algorithm using a user-defined threshold on the whole image.

Otsu. Automatically find the best threshold value using the image histogram. The algorithm picks the value that minimizes the spreads between the `True` and `False` domains as described by Nobuyuki Otsu [4].

Sauvola. In images with a lot of variation, global thresholding is not accurate anymore, so adaptive thresholding must be used. The Sauvola thresholding algorithm [5] picks the best threshold value for each pixel from the given image using its neighbors.

2.3 Watershed

Watershed is another class of image segmentation algorithms. Used on a grayscale image seen as a topographic relief, a watershed extracts the different basins within the image. Watershed algorithms produce a segmented image where each component is labeled with a different tag.

Climb provides a implementation of Meyer's flooding algorithm [3]. Components are initialized at the "lowest" region of the image (the darkest ones). Then, each component is extended until it collides with another one, following the image's topography.

2.4 Mathematical Morphology

Mathematical Morphology is heavily used in Image Processing. It defines a set of operators that can be cumulated in order to extract specific details from images.

Mathematical Morphology algorithms are based on the application of a structuring element to every points of an image, gathering information using basic operators (*e.g.* logical operators `and`, `or`, *etc.*) [6].

Erosion and Dilation. Applied on a binary image, these algorithms respectively erodes and dilate the `True` areas in the picture. Combining these operators leads to new algorithms:

Opening Apply an erosion then a dilation. This removes the small details from the image and opens the `True` shapes.

Closing Apply a dilation then an erosion. This closes the `True` shapes.

Mathematical Morphology can also be applied on grayscale images, leading to news algorithms:

TopHat Sharpen details

Laplacian operator Extract edges

Hit-Or-Miss Detect specific patterns

3. COMPOSITION TOOLS

Composing algorithms like the ones described in the previous section is interesting for producing more complex image processing chains. Therefore, in addition to the built-in algorithms that Climb already provides, a composition infrastructure is available.

3.1 Chaining Operators

3.1.1 The $ operator

The most essential form of composition is the sequential one: algorithms are applied to an image, one after the other.

Programming a chain of algorithms by hand requires a lot of variables for intermediate storage of temporary images. This renders the code cumbersome to read an maintain.

To facilitate the writing of such a chain, we therefore provide a chaining operator which automates the plugging of an algorithm's output to the input of the next one. Such an operator would be written as follows in traditional Lisp syntax:

```
(image-save
  (algo2
    (algo1
      (image-load "image.png")
      param1)
    param2)
  "output.png")
```

Unfortunately, this syntax is not optimal. The algorithms must be read in reverse order, and arguments end up far away from the function to which they are passed.

Inspired by Clojure's Trush operator [3] and the JQuery[4] (->) chaining process, Climb provides the `$` macro which allows to chain operations in a much clearer form.

[3] `http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/->`
[4] `http://api.jquery.com/jQuery/`

```
$((image-load "image.png")
  (algo1 param1)
  (algo2 param2)
  (image-save "output.png"))
```

3.1.2 Flow Control

In order to ease the writing of a processing chain, the `$` macro is equipped with several other flow control operators.

- The `//` operator splits the chain into several independent subchains that may be executed in parallel.
- The quote modifier (`'`) allows to execute a function outside the actual processing chain. This is useful for doing side effects (see the example below).
- The `$1`, `$2`, *etc.* variables store the intermediate results from the previous executions, allowing to use them explicitly as arguments in the subsequent calls in the chain.
- The `#` modifier may be used to disrupt the implicit chaining and let the programmer use the `$1`, *etc.* variables explicitly.

These operators allow for powerful chaining schemes, as illustrated below:

```
($ (load "lenagray.png")

   (//
    ('(timer-start)
      (otsu)
     '(timer-print "Otsu")
      (save "otsu.png")) ; $1

    ('(timer-start)
      (sauvola (box2d 1))
     '(timer-print "Sauvola 1")
      (save "sauvola1.png")) ; $2

    ('(timer-start)
      (sauvola (box2d 5))
     '(timer-print "Sauvola 5")
      (save "sauvola5.png"))) ; $3

   (//
    (#(diff $1 $2)
      (save "diff-otsu-sauvola1.png"))
    (#(diff $1 $3)
      (save "diff-otsu-sauvola5.png"))
    (#(diff $2 $3)
      (save "diff-sauvola1-sauvola5.png"))))
```

3.2 Morphers

As we saw previously, many Image Processing algorithms result in images of a different type than the original image (for instance the threshold of a grayscale image is a Boolean one). What's more, several algorithms can only be applied on images of a specific type (for instance, watershed may only be applied to grayscale images).

Programming a chain of algorithms by hand requires a lot of explicit conversion from one type to another. This renders the code cumbersome to read an maintain.

To facilitate the writing of such a chain, we therefore provide an extension to the morpher concept introduced in Olena with SCOOP2 [1] generalized to any object with a defined communication protocol. Morphers are wrappers created around an object to modify how it is viewed from the outside world.

Two main categories of morphers are implemented: *value morphers* and *content morphers*.

Value morpher. A value morpher operates a dynamic translation from one value type to another. For instance, an RGB image can be used as a grayscale one when seen through a morpher that returns the pixels intensity instead of their original color. It is therefore possible to apply a watershed algorithm on a colored image in a transparent fashion, without actually transforming the original image into a grayscale one.

Content morpher. A content morpher operates a dynamic translation from one image structure to another. For instance, a small image can be used as a bigger one when seen through a morpher that returns a default value (*e.g.* black), when accessing coordinates outside the original image domain. It is therefore possible to apply an algorithm built for images with a specific size (*e.g.* power of two), without having to extend the original image.

Possible uses for content morphers include *restriction* (parts of the structures being ignored), *addition* (multiple structures being combined) and *reordering* (structures order being modified).

4. EXTENSIBILITY

The pixels of an image are usually represented aligned on a regular 2D grid. In this case, the term "pixel" can be interpreted in two ways, the position and the value, which we need to clearly separate. Besides, digital images are not necessarily represented on a 2D grid. Values can be placed on hexagonal grids, 3D grids, graphs *etc.* In order to be sufficiently generic, we hereby define an image as a function from a position (a *site*) to a value: $img(site) = value$.

A truly generic algorithm should not depend on the underlying structure of the image. Instead it should rely on higher level concepts such as neighborhoods, iterators, *etc.* More precisely, we have identified 3 different levels of genericity, as explained below.

4.1 Genericity on values

Genericity on values is based on the CLOS dispatch. Using multimethods, generic operators like comparison and addition are built for each value type, such as RGB and Grayscale.

4.2 Genericity on structures

The notion of "site" provides an abstract description for any kind of position within an image. For instance, a site might be an n-uplet for an N-d image, a node within a graph, *etc.*

Climb also provides a way to represent the regions of an image through the site-set object. This object is used in order to browse a set of coordinates and becomes particularly handy when dealing with neighborhoods. For instance, the set of nearest neighbors, for a given site, would be a list of coordinates around a point for an N-d image, or the connected nodes in a graph.

4.3 Genericity on implementations

Both of the kinds of genericity described previously allow algorithm implementers to produce a single program that works with any data type. Some algorithms, however, may be implemented in different ways when additional information on the underlying image structure is available. For instance, iterators may be implemented more efficiently when we know image contents are stored in an array. We can also distinguish the cases where a 2D image is stored as a matrix or a 1D array of pixels *etc.*

Climb provides a property description and filtering system that gives a finer control over the algorithm selection. This is done by assigning properties such as `:dimension` = `:1D`, `:2D`, *etc.* to images. When defining functions with the `defalgo` macro, these properties are handled by the dispatch system. The appropriate version of a function is selected at runtime, depending on the properties implemented by the processed image.

5. CONCLUSION

Climb is a generic Image Processing library written in Common Lisp. It is currently architected as two layers targeting two different kinds of users.

- For an image processing practitioner, Climb provides a set of built-in algorithms, composition tools such as the chaining operator and morphers to simplify the matching of processed data and algorithms IO.
- For an algorithm implementer, Climb provides a very high-level domain model articulated around three levels of abstraction. Genericity on values (RGB, Grayscale *etc.*), genericity on structures (sites, site sets) and genericity on implementations thanks to properties.

The level of abstraction provided by the library makes it possible to implement algorithms without any prior knowledge on the images to which they will be applied. Conversely supporting a new image types should not have any impact on the already implemented algorithms.

From our experience, it appears that Common Lisp is very well suited to highly abstract designs. The level of genericity attained in Climb is made considerably easier by Lisp's reflexivity and CLOS' extensibility. Thanks to the macro system, this level of abstraction can still be accessed in a relatively simple way by providing domain-specific extensions (*e.g.* the chaining operators).

Although the current implementation is considered to be reasonably stable, Climb is still a very young project and should be regarded as a prototype. In particular, nothing has been done in terms of performance yet, as our main focus was to design a very generic and expressive API. Future work will focus on:

Genericity By providing new data types like graphs, and enhancing the current property system.

Usability By extending the `$` macro with support for automatic insertion of required morphers and offering a GUI for visual programming of complex Image Processing chains.

Performance By improving the state of the current implementation, exploring property-based algorithm optimization and using the compile-time facilities offered by the language.

Climb is primarily meant to be an experimental platform for exploring various generic paradigms from the dynamic languages perspective. In the future however, we also hope to reach a level of usability that will trigger interest amongst Image Processing practitioners.

6. REFERENCES

[1] Th. Géraud and R. Levillain. Semantics-driven genericity: A sequel to the static C++ object-oriented programming paradigm (SCOOP 2). In *Proceedings of the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL)*, Paphos, Cyprus, July 2008.

[2] R. Levillain, Th. Géraud, and L. Najman. Why and how to design a generic and efficient image processing framework: The case of the Milena library. In *Proceedings of the IEEE International Conference on Image Processing (ICIP)*, pages 1941–1944, Hong Kong, Sept. 2010.

[3] F. Meyer. *Un algorithme optimal pour la ligne de partage des eaux*, pages 847–857. AFCET, 1991.

[4] N. Otsu. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man and Cybernetics*, 9(1):62–66, Jan. 1979.

[5] J. Sauvola and M. Pietikäinen. Adaptive document image binarization. *Pattern Recognition*, 33(2):225–236, 2000.

[6] P. Soille. *Morphological Image Analysis: Principles and Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2 edition, 2003.