Tutorial: Computer Vision with Allegro Common Lisp and the VIGRA Library using VIGRACL

Benjamin Seppke, Leonie Dreschler-Fischer
Department Informatik, University of Hamburg, Germany
{seppke, dreschler}@informatik.uni-hamburg.de

1 Introduction

In this tutorial we present the interoperability between the VIGRA C++ computer vision library and Allegro Common Lisp. The interoperability is achieved by an extension called VIGRACL, which uses a multi-layer architecture. We describe this architecture and present some example usages of the extension. Contrary to other computer vision extensions for Common Lisp we focus on a generic design, which allows for easy interoperability using other programming languages like PLT Scheme.

VIGRA is not just another computer vision library. The name stands for "Vision with Generic Algorithms". Thus, the library that puts its main emphasis on customizable and therefore generic algorithms and data structures (see [Köthe 1999]). It uses template techniques similar to those in the C++ Standard Template Library (STL) (see [Köthe 2000]), which allows for an easy adaption of any VIGRA component to the special needs of computer vision developers without losing speed efficiency (see [Köthe 2010]). The VIGRA library was originally designed and implemented by Ullrich Köthe as a part of his Ph.D. thesis. Meanwhile, many people are involved to improve the library and the user group is growing further. The library is currently in use for various educational and research tasks in German Universities (e.g. Hamburg and Heidelberg) and has proven to be a reliable (unit-tested) testbed for low-level computer vision tasks.

Although C++ can lead to very efficient algorithmic implementations, it is still an imperative low-level programming language, that is not capable of interactive modeling. Thus, the VIGRA library also offers specialized numpy-bindings for the Python programming language as a part of the current development snapshot.

Functional programming languages like Lisp on the other hand provide an interesting view on image processing and computer vision because they support symbolic processing and thus symbolic reasoning at a higher abstraction level. Common Lisp has already proven to be adequate for solving AI problems, because of its advantages over other programming languages, like. the extendability of the language, the steep learning curve, the symbolic processing, and

the clarity of the syntax. Moreover, there are many extensions for Lisp like e.g. description logics, which support the processes of computer vision and image understanding.

2 Related work

Before introducing the VIGRACL interface, we will compare some competitive Common Lisp extensions, which also add image processing capabilities to Common Lisp:

The first system is the well-known OBVIUS (Object-Based Vision and Understanding System) for Lisp (see [Heeger and Simoncelli 2010]). It is an image-processing system based on Common Lisp and CLOS (Common Lisp Object System). The system provides a flexible interactive user interface for working with images, image-sequences, and other pictorially displayable objects. It was last updated on 1994, so it does not supply state-of-the-art algorithms used for image processing and computer vision.

ViLi (Vision Lisp) has been developed by Francisco Javier Snchez Pujadas at the Universitat Autnoma de Barcelona until 2003 (see [Snchez Pujadas 2010]). Although it offers algorithms for basic image processing tasks, it seems to be restricted to run only at Windows and to be no longer maintained.

IMAGO is another image manipulation library for Common Lisp developed by Matthieu Villeneuve until 2007 (see [Villeneuve 2010]). It supports file loading/saving in various formats and image manipulation functionalities . Although it supports some basic filtering and composition tools, there are important image processing parts missing like segmentation tools or the Fourier transform.

The last system is called ch-image (cyrus harmon image) and was last updated in 2008 (see [Harmon 2010]). Like OBVIUS, it provides a CLOS Lisp layer for image processing, but puts its main emphasize on computer graphics (e.g. creating images containing graphical elements). This system introduces many different classes for different image pixel types and therefore requires more studying of the API than systems on a more abstract level.

Contrary to these systems, our proposed VIGRACL binding is generic, lightweight, and offers advanced functions like image segmentation. There is no need for introducing new data types or classes (using CLOS) in Allegro Common Lisp. It currently provides the basic functionalities of the VIGRA library and will be extended in future to 3D processing methods etc. Further, we do not use named parameters, but named the function arguments according to their meaning, which will be shown by auto-completion e.g. when using EMACS in conjunction with Allegro Common Lisp.

Note that we will present a generic interface to the VIGRA library, that allows for the use of many other languages and programming styles, although we favor

the use of the VIGRA in together with functional languages, like Common Lisp or PLT Scheme (see [Seppke 2010]). The generic approach is also reflected in the platform availability, as the VIGRACL has already proven to work with Allegro Common Lisp at Windows, Linux or Mac.

3 Hierarchical Design of the VIGRACL

We will now present the embedding of the VIGRA's algorithms into Allegro Common Lisp. We will start with the lowest layer, and show how an image is represented on the C++ side. We then iteratively move up layer to layer to end by the Common Lisp high-order functions, which assist at the use of the library. As an example, we show the representation of a classical image smoothing filter at each level.

3.1 Lowest Layer: C++

At the lowest layer an image is represented by the VIGRA-Class `BasicImage<T>`. The template parameter `T` determines the pixel-type of the image and can either be a scalar type or a vector type (like [R,G,B]). At this layer we abstract a to allow only two types of images: Images with a floating-point pixel-type and images with a (R,G,B)-Vector of floating-point values. Multi-band images may be supported in future releases.

```
try{
    //Create result image of same size
    vigra::BasicImage<float> img2(img.width(), img.height());

    //Perform filtering with sigma=3.0
    vigra::gaussianSmoothing(srcImageRange(img),
                             destImage(img2), 3.0);
}
catch (vigra::StdException & e){  }
```

3.2 Intermediate Layer: C (shared object)

At the intermediate layer, we create C-wrapper functions around the C++ functions and types of the lowest layer. We favor the use of an own implementation instead of using automatic wrapper generation libraries like SWIG (see [Beazley 1996]) to keep full control of the interface and its definitions. To allow for an easy re-use of all the functions contained inside this shared object, we make some minimalistic assumptions according to the programming language, which connects to this wrapper library:

Image-bands are represented as one-dimensional C-arrays of type float, which is sufficient for most tasks. For matrix-computation one-dimensional C-arrays of

type double are used. Additional function-parameters can either be boolean, integer, float or double. The resulting type of each function is an integer, which is also used to indicate errors. The pointer to these C-arrays will not be created, maintained or deleted by the wrapper library. The client (Allegro Common Lisp) has to take care of all array creation ans deletion tasks. All image functions are yet defined to work band-wise.

It should be mentioned, that this layer results in a shared object (or DLL under Windows) with a very simple interface: All interface signatures consist of elementary C-data types or pointers to float and double. This simplifies the use for other programming languages. Besides Allegro Common Lisp, we have implemented interfaces for PLT Scheme and ittvis IDL (see [Seppke 2010]).

```
LIBEXPORT int vigra_gaussiansmoothing_c( const float *arr,
   const float *arr2, const int width, const int height,
   const float sigma){
   try {
      // create a gray scale image of appropriate size
      vigra::BasicImageView<float> img (arr,  width, height);
      vigra::BasicImageView<float> img2(arr2, width, height);

      vigra::gaussianSmoothing(srcImageRange(img),
                               destImage(img2), sigma);
   }
   catch (vigra::StdException & e) {
      return 1;
   }
   return 0;
}
```

3.3 High layer: Allegro Common Lisp FFI

We now connect the wrapper library to Allegro Common Lisp using the built-in Foreign Function Interface (FFI). For the representation of images we have chosen lists of the (2D) array ype of the FFI. Each array depicts a certain band of an image. The bands are ordered '(Red Green Blue) for color images and '(Gray) for grayscale images. We have chosen Allegro's FFI instead of CFFI because of the native array memory sharing between Common Lisp and C. The corresponding FFI-adapter on the lisp side for the Gaussian smoothing of a single band is given by:

```
(ff:def-foreign-call vigra_gaussiansmoothing_c
   ((arr      (:array :float))
    (arr2     (:array :float))
    (width    :int fixnum)
    (height   :int fixnum)
    (sigma    :float))
 :returning   (:int fixnum))
```

Note that the creation of images is also perfomed on this layer. The Lisp function for the gaussian smoothing of an image band on this layer is given below:

```
(defun gsmooth-band (arr sigma)
  (let* ((width  (array-dimension arr 0))
         (height (array-dimension arr 1))
         (arr2 (make-array (array-dimensions arr)
                :element-type (array-element-type arr)
                :initial-element 0.0)))
    (case (vigra_gaussiansmoothing_c arr arr2
                                     width height sigma)
      ((0) arr2)
      ((1) "Error in vigracl.filters.gsmooth: ..."))))
```

3.4 Highest layer: Allegro Common Lisp High-Order Functions

At the topmost layer, we provide a set of generic and flexible tools, which assist in mapping and to traversing of images. These functions refer to both images and image-bands. They can be seen as extensions to the well-known high-order functions in Common Lisp, but fitted to the data types of images and image-bands.

The first set of functions corresponds to the `mapcar` for lists. We define `array-map`, `array-map!`, `image-map` and `image-map!` for this purpose. We will present their use in the examples section of this tutorial. We also define folding operations for bands and images: `array-reduce` and `image-reduce`. Further, we introduce simple image- and band-iterator functions to write and apply own algorithms to images: `array-for-each-index` and `image-for-each-index`.

Most image processing functions use the mapping facilities to apply a band-defined image operation on each band of an image. For instance the gaussian smoothing of an images is defined of the gaussian smoothing of all image's bands:

```
(defun gsmooth (image sigma)
  (mapcar #'(lambda (arr) (gsmooth-band arr sigma)) image))
```

Note that the VIGRACL introduces its own package namespace (`VIGRACL`) and is defined as a Common Lisp system using Allegro's `excl:defsystem`.

4 Interactive Examples

We will now present some examples that show the practical use of the VIGRACL. The necessary first step is to let Allegro Common Lisp know where the library is located. Afterwards, we can load the library using a single command:

```
CL-USER> :ld vigracl
```

If the library has been loaded successfully, you can switch into the package by typing:

```
CL-USER> (in-package :vigracl)
```

You should get a response message from the Lisp system, which indicates that you are inside the correct package. We now start with loading an image (taken from [Ssawka 2010]):

```
VIGRACL> (setq img (loadimage "images/tools-bits.jpg"))
```

You can easily proof that this image is a RGB image by typing:

```
VIGRACL> (length img)
```

which counts the number of bands of the loaded image (three). Let us now try some basic filters of the VIGRA that are based on convolution with Gaussian kernels: Gaussian smoothing and the calculation of the Gaussian gradient magnitude (both at scale $\sigma = 2.0$):

```
VIGRACL> (setq smooth_img (gsmooth img 2.0))
VIGRACL> (setq gradient_img (ggradient img 2.0))
```

(a) original image (b) smoothed image (c) gradient magnitude

Figure 1: The loaded image and filter results

We now demonstrate how to save these images using the `saveimage` function and review the result using your favourite image browser, or view the result directly using the built-in function `showimage`, i.e. for the smoothed image:

```
VIGRACL> (saveimage smooth_img "images/smooth_bits.png")
VIGRACL> (showimage smooth_img)
```

Both functions will return true, if the saving is successful and print out error messages otherwise. Note that various image formats are supported for the import and export of images.

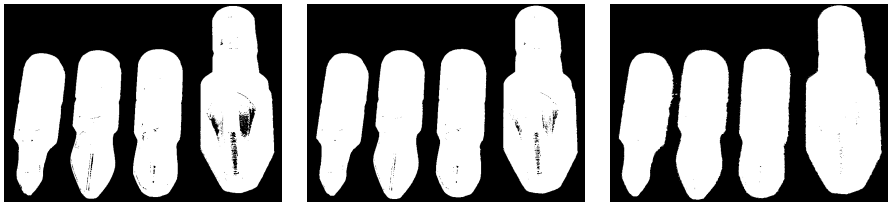
The next task will be counting the bits that are visible on our loaded image. They appear darker that the background on all bands, so we will work with a single-band image from now on:

```
VIGRACL> (setq gray_img (list (second img)))
```

We have arbitrary selected the green channel and defined a new image. To detect whether a pixel corresponds to an object, we will start with writing our first own image processing algorithm, a thresholding filter:

```
VIGRACL> (defun threshold (image val)
           (image-map #'(lambda (x)
               (if (> x val) 0.0 255.0)) image))
VIGRACL> (setq thresh_img (threshold gray_img 220.0))
```

As you can see in Figure 2, there is no threshold that perfectly separates the objects from the background. On a low threshold, many parts inside the bits remain classified as background whereas on the higher threshold some image content around the objects is misclassified.



(a) $t = 190$ (b) $t = 220$ (c) $t = 250$

Figure 2: Results of thresholding the green channel of Figure 1(a)

For a correct segmentation of the bits from the background, we need to close the holes inside of the thresholded image (with $t = 220$). This operation is performed by a morphological operator called closing. We select a radius of 5 pixel to close the unwanted holes:

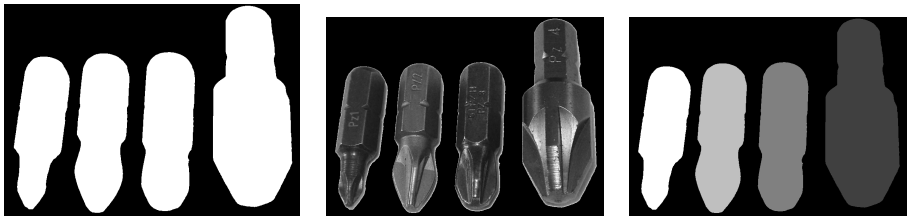
```
(setq closed_img (closingimage thresh_img 5))
```

We can also visualize the segmentation to mask out the original image using the high-order functions of the VIGRACL:

```
(showimage (image-map #'(lambda (i m) (* i (/ m 255.0)))
                      gray_img closed_img))
```

At last, we need to count the to connected components of the image to get the number of objects. This is performed by a labeling algorithm, which assigns each component an unique label. Note that the background will also be counted as one component. Therefore, the result has to be decremented by one.

```
(setq label_img (labelimage closed_img))
(setq bit_count (- (first (image-reduce #'max label_img 0.0))
                   1))
```



(a) segmented image (b) masked green channel (c) labeled image

Figure 3: Results after the closing image operation of Figure 2(b)

This finally results in a `bit_count` of 4, which means that four objects have been recognized (see Figure 3(c)). Next possible steps could be the measurement of size, mean intensity or other features of each labeled component.

5 Conclusions

We have presented some of the functionalities for the VIGRACL extension to Allegro Common Lisp. The extension uses a multi-layer architecture to grant access to the computer vision algorithms that are provided by the VIGRA library. Note that this tutorial cannot be more than an introduction into the interesting field of computer vision besides the presentation of the inter-operational design of the VIGRACL. However, it shows how easy the various functions of the VIGRA can be used within Allegro Common Lisp, given a light-weight generic interface.

We have shown that the integrated high-order functions for images and image-bands help when working with the library as they extend the well-known high-order functions by means of image-bands and images. Thus, we currently use the VIGRACL-bindings for research to assists with the low-level image processing tasks that have to taken out before the symbolic processing. Another advantage is the use for fast interactive image analyzing in scientific computing.

Due to the steep learning curve and interactive experience, we currently use the very similar VIGRAPLT (a VIGRA interface to PLT Scheme using the same intermediate layer) in undergraduate student computer vision projects.

References

[Beazley 1996] D. Beazley, SWIG: an easy to use tool for integrating scripting languages with C and C++, In: Proceedings of the Fourth USENIN Tcl/Tk Workshop, 1996, pp. 129-139.

[Harmon 2010] C. Harmon, The ch-image Homepage, `http://cyrusharmon.org/static/projects/ch-image/doch/ch-image.xhtml` (Jan. 27, 2010)

[Heeger and Simoncelli 2010] D. Heeger, E.Simoncelli, The OBVIUS Homepage, http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/vision/obvius/0.html (Jan. 27, 2010)

[Köthe 1999] U. Köthe, Reusable Software in Computer Vision, in: B. Jhne, H. Hauecker, P. Geiler: "Handbook on Computer Vision and Applications", vol. 3, A. Press, 1999.

[Köthe 2000] U. Köthe, STL-Style Generic Programming with Images, in : C++ Report Magazine 12(1), January 2000

[Köthe 2010] U. Köthe, The VIGRA Homepage, http://hci.iwr.uni-heidelberg.de/vigra/ (Jan. 26, 2010).

[Roerdink and Meijster 2000] J. B. T. M. Roerdink, and A. Meijster (2000). The watershed transform: Definitions, algorithms, and parallellization strategies. In Goutsias, J. and Heijmans, H., editors, Mathematical Morphology, vol. 41, pages 187228. IOS Press.

[Snchez Pujadas 2010] F. J. Snchez Pujadas, The ViLi Homepage, http://www.cvc.usb.es/~javier/vili.htm (Jan. 27, 2010)

[Seppke 2010] B. Seppke, Digital image processing using Allegro Common Lisp and VIGRA = VIGRACL, http://kogs-www.informatik.uni-hamburg.de/~seppke/index.php?page=vigracl&lang=en (Jan. 26, 2010).

[Ssawka 2010] Ssawka, Image of four bits, http://commons.wikimedia.org/w/index.php?title=File:PZ_1-4_0.jpg&oldid=26986252

[Villeneuve 2010] M. Villeneuve, IMAGO Common Lisp image manipulation library, http://matthieu.villeneuve.free.fr/dev/imago/, (Feb. 1, 2010)