# Efficient Applicative Programming Environments for Computer Vision Applications

## Integration and Use of the VIGRA Library in Racket

Benjamin Seppke
University of Hamburg
Dept. Informatics
Vogt-Kölln-Str. 30
22527 Hamburg, Germany
seppke@informatik.uni-hamburg.de

Leonie Dreschler-Fischer
University of Hamburg
Dept. Informatics
Vogt-Kölln-Str. 30
22527 Hamburg, Germany
dreschler@informatik.uni-hamburg.de

## ABSTRACT

Modern software development approaches, like agile software engineering, require adequate tools and languages to support the development in a clearly structured way. At best, they shall provide a steep learning curve as well as interactive development environments. In the field of computer vision, there is a major interest for both, general research and education e.g. of undergraduate students. Here, one often has to choose between understandable but comparably slow applicative programming languages, like Racket and fast but unintuitive imperative languages, like C/C++. In this paper we present a system, which combines the best of each approaches with respect to common tasks in computer vision, the applicative language Racket and the VIGRA C++ library. This approach is based on a similar Common Lisp module and has already proven to be adequate for research andr education purposes [12]. Moreover, it provides the basis for many further interesting applications. For this paper we demonstrate the use in one research and one educational case study. We also make suggestions with respect to the design and the needs of such a module, which may be helpful for the generic extension of applicative programming languages into other research areas as well.

## Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming—*Allegro Common Lisp, SBCL, Racket*;
D.2.2 [**Software Engineering**]: Design Tools and Techniques—*modules and interfaces, software libraries*;
I.4.8 [**Image Processing and computer vision**]: Scene Analysis

## General Terms

Applicative Programming, Racket, Language Interoperability, Computer Vision, Image Processing

## 1. INTRODUCTION

Although applicative programming languages have a long tradition, they still do not belong to the scrap heap. Instead, they have proven to support state-of-the-art development approaches by means of an interactive development cycle, genericity and simplicity. The influence of applicative programming paradigms is even observable in modern languages, like Python, Dart and Go. However, there are some research areas, which are computationally of high costs and are thus currently less supported by applicative programming languages.

In this paper, we select the research field of computer vision and show how to connect applicative languages to a generic C++ computer vision library called VIGRA [5]. The interoperability is achieved by two similar modules, VIGRACL for Allegro and SBCL Common Lisp and VIGRACKET for Racket. Both are using a multi-layer architecture with a common C wrapper library. In contrast to [12], where we describe the architecture on the C/C++ and the Common Lisp side in more detail, we focus on the Racket extension in this paper. We also present useful applicative programming language additions for a seamless Racket integration.

Although C++ can lead to very efficient implementations, it is not capable of interactive modeling. Applicative programming languages like Lisp and derivatives on the other hand support symbolic processing and thus symbolic reasoning at a higher abstraction levels than typical imperative languages. Common Lisp has e.g. proven to be adequate for solving AI problems since decades. There are extensions for Lisp like e.g. description logics, which support the processes of computer vision and image understanding. Thus, the integration of computer vision algorithms has the potential to result in a homogenous applicative programming environment. Besides research tasks, the steep learning curve makes applicative programming languages interesting for educational purpose. To demonstrate this, we present two case studies for the application of the VIGRACKET module. The research case study shows the implementation of a state-of-the-art image processing algorithm. The other case study shows the use in an educational context by means of interpreting a board game from its image. The second case study has been performed with undergraduate students of computer science at the University of Hamburg.

## 2. RELATED WORK

The name VIGRA stands for "Vision with Generic Algorithms". Its main emphasis is on customizable generic algorithms and data structures (see [6], [7]). This allows an easy adaption of any VIGRA component to the special needs of computer vision developers without losing speed efficiency (see [5]). The VIGRA library was invented and firstly implemented by Dr. Ullrich Köthe as a part of his PhD thesis. Meanwhile, many people are involved to improve and extend the library. Moreover, the library is currently being used for various educational and research tasks in German Universities (e.g. Hamburg and Heidelberg) and has proven to be a reliable testbed for low-level computer vision tasks. The VIGRA library follows a current trend in computer vision. Instead of providing large (overloaded) libraries, smaller, generic and theoretically founded building blocks are defined and provided. These building blocks may then be connected, combined, extended and applied for each unique low-level computer vision problem. To demonstrate the need of an interactive and dynamic way of using this building block metaphor, VIGRA comes with (highly specialized) interactive Python-bindings. Other computer vision software assist the user by visualizing the building blocks metaphor and really let the user stack components together visually, e.g. MEVISLab [8].

Despite this trending topic, there are currently no competitive computer vision libraries for interactive development with Racket. For Common Lisp, few systems are still existing, but most of them are no longer maintained. Referring to the survey of Common Lisp computer vision systems in [12], beside VIGRACL [10] currently only one system "opticl" (the successor of ch-image) is still maintained [4]. In contrast to other systems, our proposed module is generic, light-weight, and offers advanced functionality like image segmentation or sub-pixel based image analysis.

Our aim is a generic interface to the VIGRA library, that allows the use of many other languages and programming styles. This generic approach is reflected in the programming languages (Racket, SBCL and Allegro Common Lisp) as well as in the platform availability (Windows, Linux or Mac). The only requirement on the "high-level" programming language is, that a foreign function interface needs to be existent. The interface has to support shared memory access of C-pointers and value passing. These interfaces, Common Lisp UFFI or Racket FFI, in conjunction with a C wrapper library called VIGRA_C [11], yield computationally demanding tasks to the compiled wrapper library. Figure 1 illustrates the scheme of each applicative library. Besides the Common Lisp and Racket bindings, we provide bindings for other interactive development environments, too.

## 3. DESIGN OF VIGRACKET

We will now present the embedding of the VIGRA algorithms into Racket. In contrast to [12], we will not discuss the multi-layer design of the extension, but focus on the specific extensions, which are needed for Racket and the use of the module in this language. Since Racket already has extended capabilities in visualizing and displaying graphics, we will also give an overview of the seamless integration of VIGRACKET [9] into Racket's GUI.

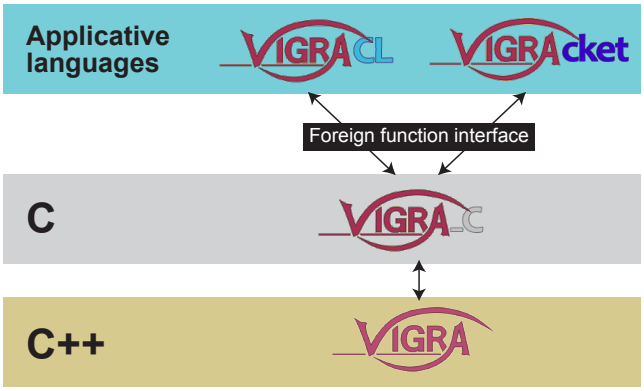


**Figure 1: Schematic view of the connection between applicative languages and C++ by using FFI and a common C-library as applicative abstraction layer.**

### 3.1 Data Structures for Images

At the lower layer of the VIGRACKET library, we need to select an appropriate data type for images. Since we assume digital images to be two-dimensional grid-aligned data, a two-dimensional array data structure is the first choice. Additionally, the array type needs to have access to a shared C-ordered memory space, where the computations of the image processing algorithms are carried out by means of the VIGRA_C wrapper library. Unlike Common Lisp, where the built-in 2d arrays are mostly capable of using a shared address space (cf. [12]), Racket provides only 1d arrays with shared memory: `cvector`. Thus, in a first step, we extend these `cvectors` to $n$ dimensions, and rename this newly defined type `carray`. We also provide fast copying operations for `carrays`, constructors, accessors and a list conversion.

Since the VIGRACKET should be able to analyze multi-band color images, we construct an image based on the introduced data type as `carrays` of type `float`:

- For single-band (gray value) images:
  `(list gray_carray)`
- For multi-band images images:
  `(list ch1_carray ch2_carray ... chN_carray)`

This implementation favors genericity by means of a fast single-band access, but may yield in slower simultaneous access of one pixel's band values. For RGB-images, we get:

```
'(RED_carray GREEN_carray BLUE_carray)
```

The interface functions use the FFI included in Racket to pass the arguments as well as the `carray`'s pointers to the VIGRA_C library. The C wrapper mostly implements band-wise operations, which can be identified by the suffix `-band`. To work on images of any number of bands, most image processing functions use the `map` function to apply a band-defined operation on each band of an image. For instance, the gaussian smoothing of an image is defined of the gaussian smoothing of all the image's bands:

```
(define (gsmooth image sigma)
  (map (curryr gsmooth-band sigma) image))
```

## 3.2 Applicative Extensions

Besides the new data types and the interaction of the library with the VIGRA_C library through the Racket FFI, we provide a set of generic and flexible high order functions, which assist in the practical development of computer vision algorithms. These functions refer to both images and image bands. They can be seen as extensions to the well-known high-order functions, but tailored to the data types of images and image bands.

The first set of functions corresponds to the `map` function for lists. We define `array-map`, `array-map!`, `image-map` and `image-map!` for this purpose. These functions may be used to apply a function to one or more images (bands) to generate a result image. The functions with a bang at the end of the function name override the first given image instead of allocating new memory for the resulting image. Although this saves memory, it introduces side-effects and should only be used carefully. An an example for the applicative variants without side-effects, the absolute difference of two images may be computed by:

```
(image-map  (compose abs -) img1 img2)
```

We also define folding operations for bands and images: `array-reduce` and `image-reduce`. These functions can be used to derive single values from bands or a list of values for images, like the maximum intensity of an image:

```
(apply max (image-reduce max 0 img))
```

Here, the inner term derives the band-wise maximum by applying the maximum function to each band array. Since the result is a list of maximal values, we get the overall maximum by applying the `max` function to the result.

Further, we introduce image traversal functions, which further support the development of own algorithms:

- `array-for-each-index`,
- `image-for-each-index` and
- `image-for-each-pixel`.

These functions call any given function of correct signature at at position of the array or image. The signature is specific for each function. Examples of the use of these functions are given in the second case study.

## 3.3 Racket-specific Extensions

Unlike other applicative languages, Racket was designed to be an easy to learn beginner's language. One way to support the learning of a language is to learn programming by design (see [3]). To support the drawing and other GUI functionality, Racket offers a variety of different packages. However, the main package for the applicative programming of shapes, image creation and drawing is still `2hdtp/image`.

Since the VIGRACKET should benefit from the existing drawing capabilities, we provide conversion functions for the different formats of the `2hdtp/image` module (for 1-band gray- and 3-band RGB-images).

These conversion functions are defined as:

- `image->racket-image`
  for the conversion from shared memory to `2hdtp/image`,
- `racket-image->image`
  for the conversion from `2hdtp/image` to shared memory.

This conversion is often necessary, since it allows to present the (processed) image without saving it to disk. However, Racket's native interface only allows to read from or to write to a device context. This interface results in very slow conversions for moderate and large image sizes. In order to enhance the execution speed, we re-implemented this conversion at machine level on the shared memory VIGRA_C side. Instead of drawing onto a device context, we rearrange the list of `carrays` for display purpose to the byte pattern, which is needed for construction of an object of class `%bitmap` (ARGB order). Compiled in machine-code, this is performed within milliseconds. This allows us to switch from one side to the other whenever needed without notable delays.

## 3.4 Preliminaries and Automatic Installation

To make the VIGRACKET accessible to a wide range of researchers as well as teachers and (undergraduate) students, only few preliminaries exist. The Racket module has already proven to run stable on Windows, Linux and Mac OS under 32- and 64-bits, depending on the Racket version installed.

Since Windows is missing a powerful package manager, the necessary binaries are bundled inside the installation package, and thus no further preliminaries exist. Although this is a very efficient and simple approach, it is not favored for Mac OS and Linux, since they provide powerful package managers, like dpkg, rpm or macports [13]. Thus, under Mac OS and Linux, you need to have a C++ compiler installed as well as a current version of the VIGRA library. Note, that the Racket installation must be of the same architecture as the VIGRA installation (32bit or 64bit). After checking this, the only preliminary is, that the "vigra-config" script must be accessible from within your shell.

If all preliminaries are met, the VIGRACKET installation package needs to be unzipped and the "install.rkt" needs to be executed. This calls the automatic installation routine of the VIGRACKET module. This routine looks up the OS and the environment and builds the VIGRA_C wrapper library for Mac OS and Linux or copies the correct binaries for Windows. A typical output is:

```
Searching for vigra using 'vigra-config ...
-------------- BUILDING VIGRA-C-WRAPPER ...
cd bin && gcc -I/src `vigra-config --cpp ...
gcc `vigra-config --libs` `vigra-config ...
```

Afterwards, all needed files and the created or copied shared object or dynamic linked library of the VIGRA_C wrapper are copied into the user's collects directory. It may then be used like any other Racket module by calling:

```
(require vigracket)
```

The provided demos in "examples.rkt" may help in getting a first impression of this module.

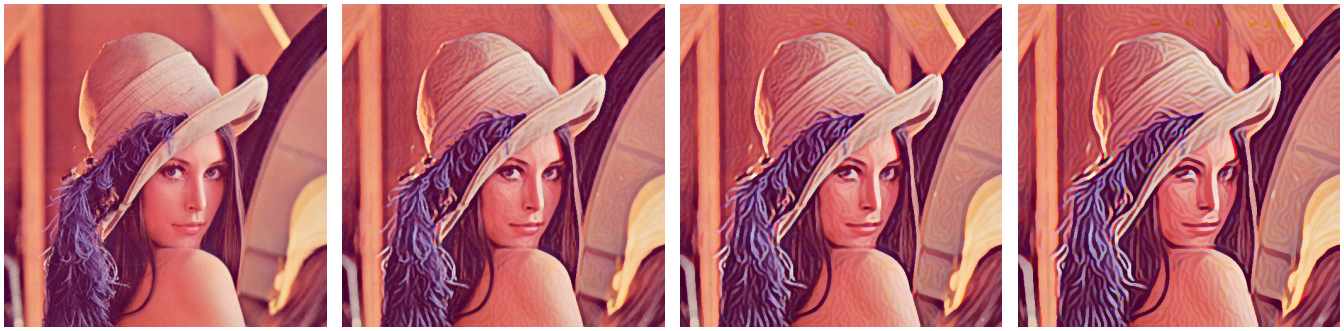

**Figure 2: Resulting images of the coherence enhancing shock filter using the parameters: $\sigma = 6$, $\rho = 2$, $h = 0.3$. From left to lower right: original image, result after 5, 10, and 20 iterations.**

# 4. CASE STUDIES

To demonstrate the use of the VIGRACKET module, we choose two different scenarios: the first case study describes the use by means of implementing a state-of-the-art algorithm for anisotropic image diffusion. The second case study is an example of an undergraduate task during a Bachelor practice at the University of Hamburg.

## 4.1 Coherence Enhancing Shock Filter

In [14], Weickert et al. describe a coherence enhancing shock filters. Shock filters are a special class of diffusion filters, which correspond to morphological operations on images. They apply either a dilation or an erosion, depending on the local gray value configurations. Weickert et al. refer to these configurations as "influence zones", which either correspond to a minimum or a maximum of the image function. The aim of the filter is to create a sharp boundary (shock) between these influence zones. The main idea of [14] is to use the Structure Tensor approach [1] for determining the orientation of this flow field instead of the explicitly modeling the partial differential equations of the diffusion equation. The Structure Tensor at scale $\sigma$ of an image is defined by:

$$ST_\sigma(I) = G_\sigma * \begin{pmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{pmatrix} \quad (1)$$

where $I$ is the image function and $I_x$ and $I_y$ are the partial derivatives in $x$- and $y$-direction. $G_\sigma$ is a Gaussian at scale $\sigma$ and $*$ is the convolution operation. To derive the main direction, in which the filter should act, the eigenvalues and eigenvectors of the Structure Tensor need to be computed. Additionally the Hessian matrix of the second partial derivatives of the image needs to be computed:

$$H(I) = \begin{pmatrix} I_{xx} & I_{xy} \\ I_{xy} & I_{yy} \end{pmatrix} \quad (2)$$

According to [14], the following equation indicates whether a pixel will be eroded or dilated:

$$D = c^2 I_{xx} + 2cs I_{xy} + s^2 I_{yy} \quad (3)$$

where $w = (c, s)^T$ denotes the normalized dominant eigenvector of the Structure Tensor of $I$. The sign of $D(I)$ is then been used in a morphological upwinding scheme to determine if an erosion or a dilation has to be performed. This approach is designed in an iterative way. Fortunately, many of the mathematical operations are already included in the VIGRA and in the VIGRACKET module.

Besides the parameter $\sigma$, we need an additional parameter $\rho$ to define the inner derivative of the Structure Tensor as well as a parameter $h$, which controls the intensity of the morphological operations. Thus, the implementation of the shock filter begins with:

```
(define (shock-image image sigma rho h iter)
  (if (= iter 0)
      image
```

This ensures that the image is given back after the last iteration. Otherwise the filtering needs to be performed. Thus, the following lines mainly show a straight-forward application of the former equations:

```
(let*
  ((st   (structuretensor image sigma rho))
   (te   (tensoreigenrepresentation st))
   (hm   (hessianmatrixofgaussian image sigma))
   (ev_x (image-map cos (third img_st_te)))
   (ev_y (image-map sin (third img_st_te)))
```

Note that the third element of the `tensoreigenrepresentation` function is the angle of the largest eigenvector. To derive the image $D$ of Eq. 3, we use the `image-map` function:

```
   (d (image-map (lambda (c s I_xx I_xy I_yy)
                    (+ (* c c I_xx)
                       (* 2 c s I_xy)
                       (* s s I_yy)))
                 ev_x                    ; <= c
                 ev_y                    ; <= s
                 (first hm)              ; <= I_xx
                 (second hm)             ; <= I_xy
                 (third hm))))           ; <= I_yy
```

The resulting image $D$ can now be used as the sign image in the unwinding morphological operation. Since this operation was not originally included in the VIGRA, it has been implemented by means of the VIGRA_C wrapper library and a corresponding Racket function was implemented, too. Thus, we end up with the recursive scheme for the last function call:

```
  (shock-image (upwindimage image d h)
               sigma rho h (- iter 1)))))
```

The results of the application of this filter to the famous Lenna image are shown in Fig. 2.

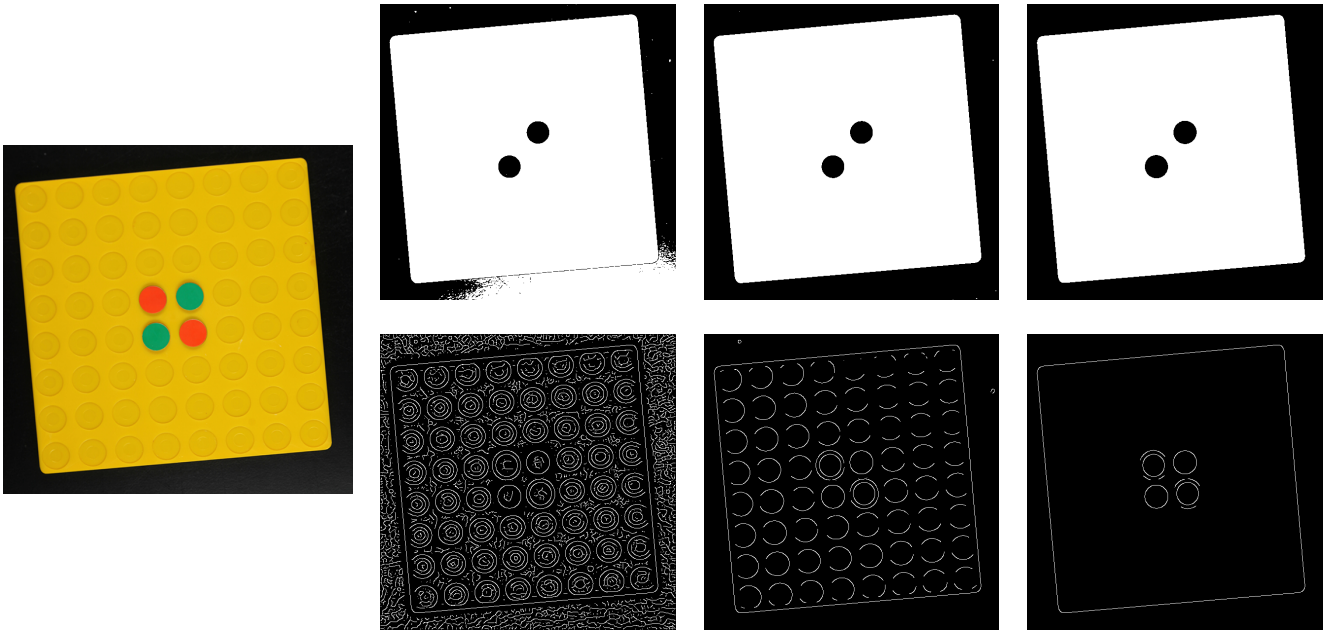

**Figure 3: Detection of the game board from the image (left). Upper row, from left to right: thresholding results for $t = 25$, $t = 50$ and $t = 100$. Lower row, from left to right: results for the Canny algorithm at scale $\sigma = 3$ with (edge) thresholds $t = 0$, $t = 1$ and $t = 2$.**

## 4.2 Board Game: Reversi

For the second case study, we selected the task of an undergraduate student practice. The results were achieved by a team of students during one term (13×4 hrs) in a Bachelor practice. The participants had few experience in applicative programming an no knowledge in image processing or computer vision. However, the aim of this practice is to teach both, applicative programming as well as computer vision at once. After the first two weeks, where the students participate in guided exercises, they select a board game. The selected game will be photographed at some states for each team. To pass the practice, the students have to perform the following tasks using DrRacket and VIGRACKET:

1. Retrieve the game state from the image,
2. Write a GUI to visualize the game state,
3. Implement the game logic and
4. Extend the GUI to continue the game.

Since we focus on the VIGRACKET integration in this paper, we present the retrieval of a game state from the image. The game is Reversi (also known as Othello), which can be considered as a prototypical example for this task. The retrieval may further be divided into the separation of the board from the background and the derivation of the game state from the sub-image.

Figure 3 (left) shows a typical image of the initial state. The board is slightly rotated and comparably brighter than the dark background. To determine the bounding box (Fig. 4, left) of the board, the boundaries between board and background need to be estimated.

One naive approach is to classify each pixel by its gray value (intensity), e.g. at the red band. This thresholding may be expressed by the following function:

```
(define (threshold v t)
  (if (< v t) 0.0 255.0))
```

To apply the threshold for $t = 50$ to the red band of `img`, we can use the `image-map` function:

```
(define img_gray (image->red img))

(define thresh_img50
  (image-map (curryr threshold 50)
             img_gray))
```

The application of different thresholds yields different results, which are shown in Fig. 3 (upper row). Note that the missing of the green game token does not influence the detection of the outer boundaries of the game board.

Another possibility is to use edge detection, since the strongest edges in the image may correspond to the boundaries of the game board. Here, we decided to use the Canny edge detector [2] at a scale of $\sigma = 3$ using various thresholds. To apply the detector for $t = 2$ to the red band of `img` and mark each edge pixel by 255.0, we can call the Canny implementation of the VIGRA directly:

```
(define canny_img2
  (cannyedgeimage img_gray 3.0 2.0 255.0)
```

The application of the Canny algorithm at different thresholds yields different results, which are shown in Fig. 3 (lower row). At low thresholds many edges are detected due to the image noise but vanish at a threshold of $t = 2$.

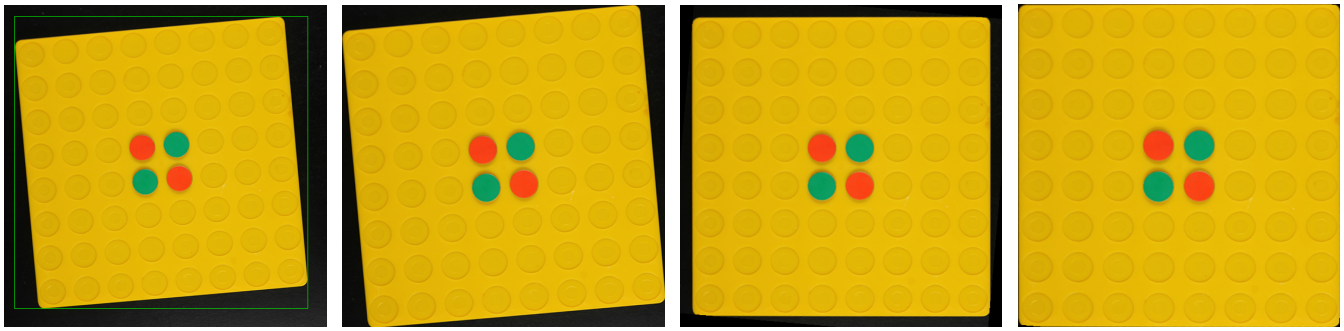

**Figure 4: Extraction of the game board from the image. From left to right: estimated bounding box (green), cropped image, rotation corrected image, cropped rotation corrected image.**

The next step is to derive the bounding box of the game board from either the threshold or the canny resulting image. We first define a bounding box as a four element vector (left, upper, right, and lower position). Then the derivation of the box at an image position ca be expressed as:

```
(define (findBBox x y col bbox)
  (when (>  (car col) 0.0)
    (begin
      (when (> x (vector-ref bbox 2))
            (vector-set! bbox 2 x))
      (when (< x (vector-ref bbox 0))
            (vector-set! bbox 0 x))
      (when (> y (vector-ref bbox 3))
            (vector-set! bbox 3 y))
      (when (< y (vector-ref bbox 1))
            (vector-set! bbox 1 y)))))
```

The initial state of the box depends on the update scheme of the above function and is given by:

```
(define bbox
  (vector (image-width img) (image-height img)
          0 0))
```

We can now use the iteration/traversal framework of the VIGRACKET to iterate over each pixel of the canny image and call the `findBBox` function at each coordinate `x`, `y` with the color list `col` at that position:

```
(image-for-each-pixel
              (curryr findBBox bbox)
              canny_img2)
```

This updates the bounding box to contain the minimal and maximal coordinates. For this example, we use the results of the canny approach to proceed with the next steps. Fig. 4 (left) shows the procedure after the determination of the first bounding box.

The next step it to crop, the image according to the bounding box `bbox`. Since VIGRACKET does not provide such a functionality, we use the Racket interface and import the `2htp/image` module. However, to resolve name conflicts, we have to restrict the `require` command by:

```
(require
  (rename-in 2htdp/image
             (save-image save-plt-image)
             (image-width plt-image-width)
             (image-height plt-image-height)))
```

After importing this module, we use the Racket's `crop` function to perform the cropping:

```
(define (cropimage img ul_x ul_y lr_x lr_y)
  (let ((w (- lr_x ul_x))
        (h (- lr_y ul_y)))
    (racket-image->image
      (crop ul_x ul_y w h
        (image->racket-image img)))))

(define cropped_img
  (cropimage img
             (vector-ref bbox 0)
             (vector-ref bbox 1)
             (vector-ref bbox 2)
             (vector-ref bbox 3)))
```

The cropped image is shown in Fig. 4 (second image). One can observe that the image is still not adequately cropped, since it is rotated around the new image center. To estimate this rotation, we search for the position of leftmost and rightmost marked pixel at the first line of the box of the edge image using a helper function:

```
(define (findFirstPixelInRow img x1 x2 row)
  (let ((intensity
         (apply max (image-ref img x1 row))))
    (if (= intensity 0)
        (if (= x1 x2)
            #f
            (findFirstPixelInRow
              img
              (+ x1 (sgn (- x2 x1))) x2 row))
        x1)))
```

To get the leftmost position relative to the beginning of the bounding box, we call the function and pass the extracted positions as parameters:

```
(define canny_left
  (- (findFirstPixelInRow canny_img2
                          (vector-ref bbox 0)
                          (vector-ref bbox 2)
                          (vector-ref bbox 1))
     (vector-ref bbox 0)))
```

To find the rightmost marked pixel, the second and third parameter of the `findFirstPixelInRow` need to be swapped. Both positions are then used to compute the arithmetic mean position on the first line.

Finally, due to the rounded borders of the game board, a constant correction factor is added. Let `pos` be the corrected position, `bbox_width` be the width of the bounding box. Then the rotation angle can be derived as:

```
(define angle
    (/ (* (atan  (- bbox-width pos) pos) 180)
       pi))
```

To correct the rotation of the cropped image, we use the rotation function provided by the VIGRACKET module. Its arguments are the angle (in degrees) and the degree of the interpolation function used (here: bilinear):

```
(define croped-rotated
   (rotateimage cropped_img (- angle) 1))
```

The result of this rotation correction is shown in Fig. 4 (third image). To crop this image, to get the final result, we repeat the former steps. Thus the edge image has to be cropped and rotated in the same manner as the image of the game board:

```
(define cropped_canny2
  (cropimage canny_img2
             (vector-ref bbox 0)
             (vector-ref bbox 1)
             (vector-ref bbox 2)
             (vector-ref bbox 3)))

(define cropped-rotated_canny2
   (rotateimage cropped_canny2 angle 1))
```

After these operations, the bounding box has to estimated again to crop the rotated image. This may be written as:

```
(define bbox2
  (vector (image-width  cropped-rotated_canny2)
          (image-height cropped-rotated_canny2)
          0 0))

(image-for-each-pixel  (curryr findBBox bbox2)
                       cropped-rotated_canny2)

(define cropped_img2
  (cropimage cropped-rotated
             (vector-ref bbox2 0)
             (vector-ref bbox2 1)
             (vector-ref bbox2 2)
             (vector-ref bbox2 3)))
```

The result of this application, the image `cropped_img2`, is shown in the rightmost image of Fig. 4. It does only contain the game board. The next step is to extract the game state from this cropped image. This is done by sampling the image at the center positions of every possible token location. Since there are 8×8 fields, the extraction of a color value for a place $x, y \in \{0, 1...7\}$ is performed by:

```
(define dx (/ (image-width  cropped_img2) 8))
(define dy (/ (image-height cropped_img2) 8))

(define (board_pos->color image x y)
  (image-ref image
    (inexact->exact
        (round (+ (/ dx 2) (* x dx))))
    (inexact->exact
        (round (+ (/ dy 2) (* y dy))))))
```

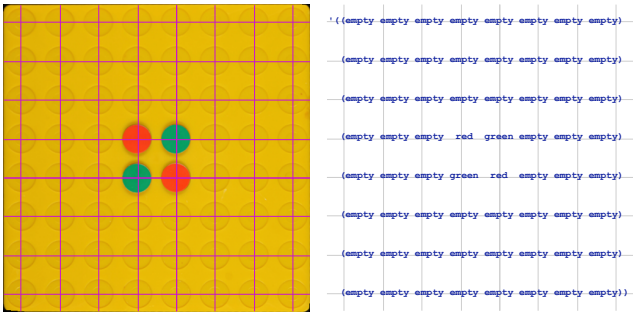


**Figure 5: Sampling of the cropped image using an equally spaced rectangular grid. Left: grid superimposed to the cropped image, right: the resulting symbolic representation superimposed on the grid.**

The sampling scheme is depicted in Fig. 5 (left). Depending on the extracted color, a classification to the tokens red, green or empty has to be made. If we assume, that red dominates the intensity for a red token, green dominates the intensity for green tokens but is comparably darker, we may express this as:

```
(define (classify-color col)
  (let* ((val (/ (apply + col) (length col))))
      (if (> (first col) (* 2 val))
            'red
            (if (> (second col) (* 1.5 val))
                  'green
                  'empty))))
```

Using these functions, we can now retrieve a list of lists to represent the game state from the image using two nested recursions, one for the rows and one for the columns of the locations of the tokens:

```
(define (board_rows image x y)
  (if (> y 7) '()
      (cons (board_cols image x y)
            (board_rows image x (add1 y)))))

(define (board_cols image x y)
  (if (> x 7) '()
      (cons (classify-color
                  (board_pos->item image x y))
            (board_cols image (add1 x) y))))

(define state (board_rows cropped_img2 0 0))
```

Since we have 8 positions per row and 8 rows, we end up with a list of 64 items in total. The resulting list is shown in Fig. 5 (right). Note that this is only one possible way of classification. Alternatively, one could switch from RGB into a HSV (hue, saturation, value) colorspace for classification.

Based on the extracted game state, the team members implemented a graphical user interface using the world framework, which is part of Racket's `2htdp/universe` module and wrote the game logic. As a result, they are able to continue the game, which was captured in the picture. Although this requires the knowledge of modeling graphical user interfaces and game logics, it can be implemented without using the VIGRACKET module.

# 5. CONCLUSIONS

We have motivated the need for interactive development methods in the field of computer vision w.r.t. research and education purposes. For many years, applicative programming has been used to solve higher AI tasks. But with a computer vision extension of such languages like Racket or Common Lisp, we are now able to offer a homogeneous, general, and highly interactive environment.

As a demonstration, we presented some of some functionalities of the VIGRACKET module in research and educational contexts. Since the extension uses a multi-layer architecture to grant access to the computer vision algorithms that are provided by the VIGRA library, a corresponding VIGRACL module (tested with Allegro Common Lisp and SBCL) is also available. We demonstrated the efficiency of the module in different aspects:

- An intuitive interface to images using shared memory,
- Generic approaches of the VIGRA, which provide great fundamental building blocks,
- High-order functions which support the development of own algorithms and
- A seamless integration into Racket by means of the built-in GUI and data types.

We have shown that the integrated high-order functions for images are really helpful in practice, since they provide wrappers for common tasks, like mapping a function on a complete image or traversing over an image in a clearly defined way. These functions in conjunction with the generic approach of the VIGRA and the easy automated installation routine make the VIGRACKET a valuable tool, not just for researchers but for teachers, too.

At the University of Hamburg, we use the VIGRACKET module as well as the VIGRACL module for research to experiment with low-level image processing tasks that have to be performed before the symbolic interpretation of the image's content.

The VIGRACKET module is also used and improved on a regular basis for a Bachelor practice at the University of Hamburg. Here, the steep learning curve and interactive experience with images and algorithms helps the undergraduate students to learn applicative programming in conjunction with computer vision during a single term. In the educational context, we found that the use of applicative programming encourages the students to understand the underlying algorithms better when compared with typical imperative low-level languages like C. This yields to an increased overall interest in computer vision.

This interest gaining of students has already resulted in many excellent Bachelor theses. In the last four years, a total of over 100 students successfully passed the practice and rated it A+. The only drawback, which has been mentioned by the students, is the limited performance of computer vision algorithms written in pure Racket.

It needs to be mentioned that, although the examples in chapter 4 where realistic, this paper cannot be more than an introduction into the interesting field of computer vision. Additionally, only a very small subset of the functionality of VIGRACKET module has been shown here. However, the examples clearly demonstrate how easy the functions of the VIGRA can be used within Racket or Common Lisp by means of the generic common VIGRA_C interface.

# 6. REFERENCES


[1] J. Bigün, G. Granlund, and J. Wiklund. Multidimensional orientation estimation with applications to texture analysis and optical flow. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(8):775 –790, aug 1991.

[2] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, November 1986.

[3] M. Felleisen. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, 2001.

[4] C. Harmon. opticl - an image processing library for common lisp: `https://github.com/slyrus/opticl`.

[5] U. Köthe. The vigra homepage: `https://github.com/ukoethe/vigra`.

[6] U. Köthe. *Reusable Software in Computer Vision*, pages 103–132. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[7] U. Köthe. *Generische Programmierung für die Bildverarbeitung*. Books on Demand GmbH, 2000.

[8] F. Ritter, T. Boskamp, A. Homeyer, H. Laue, M. Schwier, F. Link, and H.-O. Peitgen. Medical image analysis. *Pulse, IEEE*, 2(6):60–70, Nov 2011.

[9] B. Seppke. The vigracket homepage: `https://github.com/bseppke/vigracket`.

[10] B. Seppke. The vigracl homepage: `https://github.com/bseppke/vigracl`.

[11] B. Seppke. The vigra_c homepage: `https://github.com/bseppke/vigra_c`.

[12] B. Seppke and L. Dreschler-Fischer. Tutorial: Computer vision with allegro common lisp and the vigra library using vigracl. In *Proceedings of the 3rd European Lisp Symposium*, 2010.

[13] The MacPorts team. The macports project: `https://www.macports.org`.

[14] J. Weickert. Coherence-enhancing shock filters. In *Lecture Notes in Computer Science*, pages 1–8. Springer, 2003.