Near-Realtime Computer Vision with Racket and the Kinect sensor

Benjamin Seppke
University of Hamburg
Dept. Informatics
Scene Analysis and Visualization Group
seppke@informatik.uni-hamburg.de

11 April 2016

Abstract

Functional programming languages, like Lisp or Racket are known to be general purpose languages with a steep learning curve and wide range of applications. They can be used interactively to solve problems and have inspired other comparably new languages with respect to functional extensions (e.g. Python or Swift). However, at least the interpreted functional languages, have a poor reputation when it comes to execution speed. In this paper, we will prove that this reputation is either wrong or at least not applicable for the field of computer vision and the language Racket. We build upon the VIGRACKET module which combines the best of the compiled and interactive worlds with respect to common tasks in computer vision, Racket and the VIGRA C++ library (see [11]). We will present the current state of the module and analyze the execution speed for the most important operation types. After the analysis we present ways to increase the processing speed and optimize the module accordingly. To demonstrate the use of the optimized library, we connect Racket to the Microsoft Kinect Sensor and present two case studies: a blue-screen simulation, and an interactive natural pointing device interface.

1 Introduction

In [11] we have presented how well functional programming and computer vision approaches may be combined by means of the VIGRACKET library. Like other state-of-the-art interactive development environments, functional languages natively offer an interactive development cycle, generic modeling and even powerful garbage collectors. So, instead of using a new language with functional extensions, like Swift, Python or others, why not simply use well-known functional languages like Lisp or Racket?

Due to the generic C-wrapper VIGRA_C [10], we are mainly independent of the target language, which uses the VIGRA functionality. We select the Racket programming language and the VIGRACKET module since it is currently the only functional wrapper for computer vision with Racket and has proven to be adequate w.r.t. research and university teaching. But in contrast to [11], our aim is to make the VIGRACKET capable of interactive (near-) realtime computer vision, which involves further analysis and optimization steps:

- **Realtime algorithms:** All of the used algorithms need to be capable of an adequate execution speed. Since we may mix up straight C-calls of computer vision algorithms with functional Racket extensions in the functional extension of the VIGRACKET module, we need to make sure that both are fast enough.
- **Visualization:** To keep the user informed about the current status or the current (near-) realtime resulting images, we need to provide an adequate solution for integrating visualization in Racket at high efficiency and, at best, at no computational costs.
- **Data sources:** To demonstrate the use on a meaningful task, we need input devices, which give us a stream of images or other data to be analyzed.

The main aim of this paper is to analyze the VIGRACKET module and the underlying VIGRA_C C-wrapper to the VIGRA computer vision library with respect to all of the constraints mentioned above. It should be highlighted, that the VIGRACKET module itself has been developed solely to introduce computer vision to Racket, neither to support interactive realtime processing nor to support fast functional development interfaces or fast visualizations.

We will first analyze critical parts of the VIGRACKET module, which prevent the use for (near-) realtime task. Afterwards, we will provide solutions to enhance the module and eventually make it applicable for interactive (near-) realtime tasks. To demonstrate this applicability of the enhanced module, we connect Racket to the Microsoft Kinect Sensor and present two case studies: a blue-screen simulation, and the an interactive natural pointing device interface.

2 VIGRA and VIGRACKET

Since we defined the VIGRACKET module as an interoperability library between Racket and the generic VIGRA_C layer, we briefly introduce the VIGRA library itself. The acronym stands for “Vision with Generic Algorithms” and its main emphasis is to provide customizable generic algorithms and data structures (see [5], [6]). This allows an easy adaption of any VIGRA component to the special needs of computer vision developers without losing speed efficiency (see [4]). Thus, with respect to making the included functions more efficient and real-time capable, we cannot perform better. We assume, that the best-possible efficiency is already achieved by VIGRA’s algorithm implementations.

Another part of the VIGRACKET module is the shared memory based design. To make the system behavior as functional as possible, a lot of memory needs to be acquired for each new image. To overcome this memory problem, the VIGRACKET module has already introduced in-place operations for some high-order functional extensions. One example is the function `(image-map func ...)`, which maps a function with n parameters to n images of equal size and channel count to generate a resulting image. This function also has a variant `image-map!`, which will not allocate new memory, but will store the results inside `img1`, meaning the result will be equal to the changed image `img1`. However, there still might be limitations due to the fact, that the shared memory type `cvector` and its accessors are being used by the image representations.

To visualize the images using DrRacket, the VIGRACKET library has additional conversion functions: `image->bitmap` and `bitmap->image`. Although they have been improved from version to version (see [11]), there still might be optimization potential. Especially the conversion to Racket bitmaps is currently too slow for realtime applications, since it takes more than 1 second for a color image of 2 mega pixels. Since the VIGRACKET module is not an acquisition library, it currently does not support real-time acquisition devices. Thus, we need to implement another grabbing module to get these data. We have selected the Microsoft Kinect sensor as the acquisition device, and to use the OpenKinect libfreenect library for access. Unfortunately, this library does provide interaction layers for Python and Java, but none for Racket (see [12]).

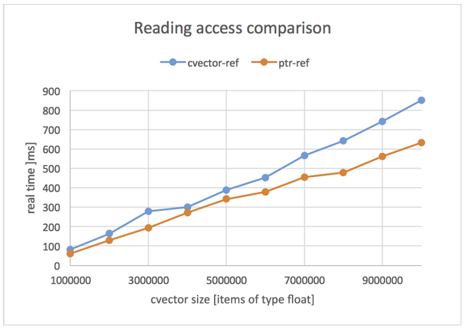
3 Enhancements for fast computer vision

We will now analyze the VIGRACKET bindings w.r.t. the execution speed and will present the necessary enhancements to support realtime computer vision applications. For sake of clarity, we distinguish between each necessary improvement in the following subsections.

3.1 Memory allocations

Modern processing architectures are highly optimized for memory allocations, since they are often occurring and time consuming tasks. It is worth mentioning, that the critical part is not just the allocation of a potentially large memory region, but the search for such a memory chunk (cf. [7]). The allocation itself may then be performed at constant time. On this low-level view of a memory allocation, everything is optimized well, and we have nearly no influence in getting faster within Racket.

However, when allocating new memory for an image, Racket allocates a managed memory region by means of a `cvector`, which also invokes the garbage collector. The use of this managed memory is highly desirable, since we don't need to take care about the lifetime of each image with respect to the VIGRACKET module in memory.



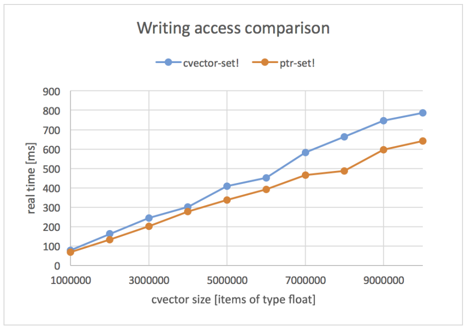


Figure 1: Comparison of the execution speeds for the `cvector` datatype. Left: comparison for read-only accesses, right: comparison for writing accesses on every item in a sequential order. Note that the advantage in computation time of using unsafe functions increases with the container size.

As a conclusion, the only way to save memory allocations is to re-use the images for certain operations. Since this approach is against the functional principle and introduces strong side-effects, it must be used with extreme caution only. Beside the `image-map!` function, we introduce functions which convert from and to already existing Racket bitmaps (cf. sec. 3.3).

The replacement of the functional with the in-place methods did not improve the execution speed in simple allocation performance tests. The reason is, that Racket is able to allocate memory at nearly zero time. However, in practical applications, we might see an advantage of the in-place methods by means of fewer garbage collector activity. This greatly influences e.g. the runtime of realtime image display.

3.2 Image accessors

Nearly all functional interfaces, like e.g. the `image-map` function, need to access the image data for each pixel on each channel for potentially all given images to compute the results. As an example, for a mapping over 2 RGB Images of size 1,000×1,000 we need 6,000,0000 read accesses on the underlying `cvector` and 3,000,000 write accesses on the target image. Since we may use the `image-map!`, we can neglect the costs of memory allocations here.

All band- and image accessor functions are currently using the `cvector-ref` and `cvector-set!` methods to read and write image data. After fruitful discussions along the European Lisp Symposium 2015 we learned, that these operations are providing safe (range checked) accesses to the managed memory space. But for some cases, e.g. the mapping of a function over equally sized images, we have prior knowledge, that all accesses (either reading or writing) are valid and can ignore the range checks. There may also be other cases, where the user wants unsafe access to the image to increase the execution speed.

For the experienced user and the internal use of the mapping functions, we introduce two alternative accessors for the underlying `cvector` and promoted them first to `carrays` and eventually to bands and images. As an example, we present the corresponding `cvector-ref` methods:

```
(define* (cvector-ref v i)
 (if (and (exact-nonnegative-integer? i)
          (< i (cvector-length v)))
   (ptr-ref (cvector-ptr v) (cvector-type v) i)
   (error 'cvector-ref ... )))
```

```
(define (cvector-ref/unsafe v i)
  (ptr-ref (cvector-ptr v)
           (cvector-type v) i))
```

Although the use of these functions has a major impact on the execution speed, it has to be handled with extreme care, because they allow access to unrestricted shared memory addresses. Fig. 1 shows the results for different vector sizes and access functions. From these we can clearly observe, that we may save about 100-200 ms for typical access counts.

3.3 Image conversion

For realtime image-driven tasks, we need to further enhance the conversion from VIGRACKET's image representation to Racket's bitmaps in order to present them inside Racket GUIs. Although the allocation of memory for images and bitmaps has already been discussed in section 3.1, there is still a need for further improvement. In [11] we have already presented one optimization strategy: the use of optimized C-functions for converting gray or RGB VIGRACKET images to ARGB-ordered C-arrays. This ordering is required for the bitmap initialization on the Racket side. Analysis of the Racket bitmap construction has shown, that the correct datatype for the initialization of a Racket bitmap is `bytes`, which stands for a simple array of type `byte` or `char`, that is located on shared memory. Thus, it is sufficient to change the former ARGB-array's datatype in the C-wrapper from `unsigned char` to `char` and to replace the list conversion by a direct initialization of each Racket bitmap with the array data. This greatly simplifies the conversion from VIGRACKET's RGB images to Racket bitmaps:

```
(define (rgbimage->argb-bytes image)
  (let* ((r_band   (first  image))
         (g_band   (second image))
         (b_band   (third  image))
         (w (band-width  r_band))
         (h (band-height r_band))
         (argb-bytes (make-bytes (* w h 4))))
    (case (vigra_convert_rgbbands_to_argb_c
             (band-data r_band)
             (band-data g_band)
             (band-data b_band)
```

```
              argb-bytes w h)
       ((0) argb-bytes)
       ((1) (error "conversion failed!")))))
```

A similar function named `grayimage->argb-bytes` exists to perform the conversion for VIGRACKET's one channel gray-images, which are also converted to the ARGB format by setting the gray values for each RGB channel. The creation of a bitmap using this function is then performed without any further conversions by means of:

```
(define (image->bitmap img [bitmap null])
  (let* ((w  (image-width img))
         (h  (image-height img))
         (bm (if (empty? bitmap)
                 (make-object bitmap% w h
                        #f #|not monochrome|#
                        #t #|alpha channel |#)
                 bitmap)))
    (case (image-numbands img)
      ((1) ;;Grayscale
         (send bm set-argb-pixels 0 0 w h
                  (grayimage->argb-bytes img))
         bm)
      ((3) ;;RedGreenBlue
         (send bm set-argb-pixels 0 0 w h
                  (rgbimage->argb-bytes img))
         bm)
      (else (error "Only RGB or gray!")))))
```

Note that this implementation also supports the use of an existing bitmap for conversion to save memory allocations. Similar functions exist for the countercase of converting Racket's images to RGB images of the VIGRACKET module. Although this may be considered as just a fine-tuning of the existing approaches, it yields to a remarkable performance enhancement. In the former VIGRACKET version, it took about 1.5-2.0 seconds to construct a Racket bitmap from the VIGRACKET representation. After the use of the optimized byte conversion, we end up with real times of 0-30 ms per image conversion. This is a necessity, if we want to present regularly updated images to the user for interactive computer vision. The computation still takes some time (cf. sec. 3.2), but we are now able to present the results at almost constant time.

3.4 Connection to the Kinect sensor

The enhancements of the previous sections are necessary for the support of fast image processing and computer vision, but we are still in need for a fast and interesting datasource, which generates data to be analyzed. For the studies presented herein, we have selected the Kinect sensor, since it provides a two dimensional image stream, like a webcam, but registered depth information, too.

One could refer to these kind of devices as RGBD (Red, Green, Blue, Depth) devices, or 2.5D devices.We are interested in the grabbing and processing of (raw) data from the sensor. Since the official "Microsoft Kinect for Windows SDK" does not provide platform-independent use (cf. [13]), we establish the connection to the Kinect sensor by means of the OpenKinect open source project [12]. Based upon the low level USB-interface via the libusb, the libfreenect library allows access to all the necessary data. In order to avoid concurrent asynchronous calls and callbacks, we use the synchronous grabbing access API. This API provides access through a specialized part of the libfreenect, which is called libfreenect_sync.

Since the image formats of the raw data used by the libfreenect is not compatible with the image representation of the VIGRACKET module, we need to introduce another small Racket/C-wrapper, to which we refer to as "rackinect". On the Racket side of this wrapper we define grabbing functions for the acquisition of the raw data. The function `(grabdepth)` grabs the depth data (in mm) and stores it inside a newly allocated one channel VIGRACKET image. The function `(grabvideo)` grabs the current RGB image and stores it by means of a new three channel VIGRACKET image. Finally the `(grabdepth+video)` grabs both data and stores it by means of new a four channel VIGRACKET image, where the first channel contains the depth data (in millimeter) and the last most three channels contain the RGB data.

If we assume a frame rate of 25 images per second, and and RGBD `carray` of type `float` with a shape of (640,480,4), we end up with allocations of approx. 120MB/s. To avoid these allocation costs, we have implemented additional functions, which use already existing images to be filled or updated with the current Kinect data. These functions may be easily recognized by the suffix "`/unsafe`" and are also called by their functional counterparts after the new image allocation.

As a demonstration, we present the use of both functions for a near real-time depth and video display GUI. Here we use the animate functionality of Racket's `2htdp/universe` module (see [1]). It takes a function with one argument and a Racket bitmap as a result type and calls the given function 28 times per second using an increasing argument `t`.

```
(define dp (grabdepth+video))
(define d  (image->bitmap (list (car dp))))
(define p  (image->bitmap (cdr dp)))

(define (live-view-combined t [update_each 4])
  (when (= (modulo t update_each) 0)
     (begin
        (grabdepth+video/unsafe dp)
        (image->bitmap (list (car dp)) d)
        (image->bitmap (cdr dp)        p)))
  (beside p (status t update_each) d))
```

The animation can then be started by:

```
(animate live-view-combined)
```

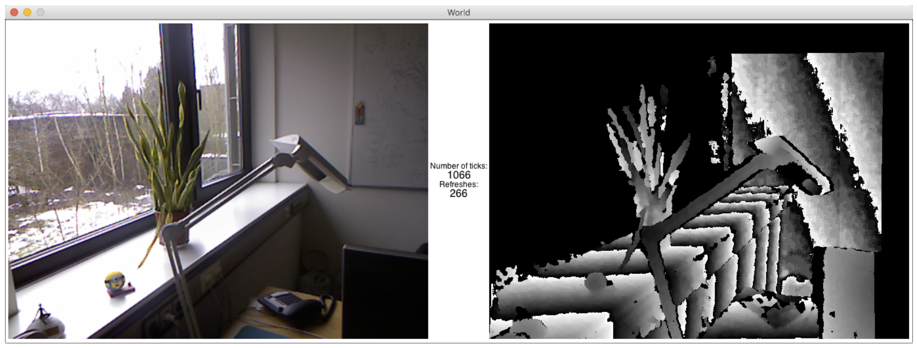


Figure 2: Image taken out of the sequence we get by calling (animate live-view-combined). Left: RGB image, center: current tick and refresh counter, right: depth image. Note that the gradients in the depth image arise from overflows w.r.t. the value range 0...255. Since the depth values are normalized to millimeters, there is one gradient each 25.5 cm. Black parts of the depth image denote regions with unknown depth.

The function `(status t update_each)` is just used to display the current tick and the count of image updates. An example output of the image is given in Fig. 2. The update_each argument controls the refresh rate of the Kinect data. Above, it is set to 4, which corresponds to a refresh rate of $4/28 = 1/7$ seconds.

4 Case Studies

To demonstrate the new near-realtime capabilities of the VIGRACKET in conjunction with the rackinect module, we choose two different scenarios: the first case study describes the simulation of a so called blue-screen (see [3]), without actually needing a blue screen. The second case study shows how to use the Kinect data to emulate an interactive pointing device.

4.1 Blue-screen simulation

The use of blue-screen (also: green-screen, see [2]) technology is currently used for many productions, like movies, TV shows or news. The aim of this technique is to cut the actors or certain objects of interest out of the current scene and place them virtually elsewhere. In TV studios, this is mainly achieved by putting the persons or objects of interest in front of a blue or green wall. Using image processing techniques, this uniform background is removed and replaced by a virtual one (see [3]). The rich data of the Kinect sensor offers us another approach. We may select a depth range, where the objects are present and then use a mask based on the depth values to cut off the objects of interest. This does not involve the RGB image and thus no blue or green background canvas is required. To determine, whether a depth-value is inside a range interval, we define the mask as follows:

```
(define (depth->mask d [front 1] [back 1000])
  (image-map! (lambda (x) (if (< front x back)
                                  255.0
                                  0.0))
              depth-img))
```

This function may then be used to combine the RGB image of the Kinect sensor with the artificial background image. Since image-map is only able to map images of corresponding sizes and channel counts, we need to create an RGB-mask based on the results of the former structure. Since images are lists of their channels, this can be achieved by:

```
(define (gray-mask->rgb-mask mask)
  (let ((first_band (car mask)))
    (list first_band first_band first_band)))
```

After the creation of an RGB-mask, we may define the cut-off of the objects from the original image using the mask as well as the filling up with the new background using the `image-map` function over all three images:

```
(define (combine-images mask old new)
    (image-map (lambda (m o n)
                  (if (> m 0.0) o n))
               mask old new))
```

First results of the application to the raw depth data of the Kinect did raise one general problem. Due to the poor depth detection quality of the Kinect sensor, there are some gaps, which have not been cut off correctly (cf. Fig. 3, left). One way to solve this is the use of morphological operations, which are already included in the VIGRACKET module. To compare the influence, we use this function in conjunction with the`2htdp/universe` module.

```
; the new background
(define img (loadimage "...")) ; some rgb image

(define (blue-screen ni t [update_each 4])
  (when (= (modulo t update_each) 0)
    (let* ((dp (grabdepth+video/unsafe dp))
           (gm (depth->mask (list (car dp))))
           (fm (closingimage gm 10))
           (m1 (gray-mask->rgb-mask gm))
           (m2 (gray-mask->rgb-mask fm))
           (rgb (cdr dp))
           (res1 (combine-images m1 rgb ni))
           (res2 (combine-images m2 rgb ni)))
      (begin
       (image->bitmap cut_off1 pic1)
       (image->bitmap cut_off2 pic2))))
  (beside pic1 (status t update_each) pic2))
```

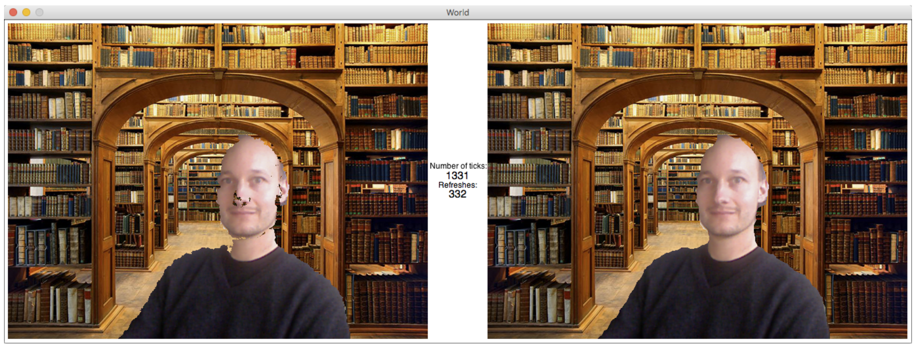


Figure 3: Image taken out of the sequence we get by calling `animate` with the `blue-screen` function. Left: Result using the raw depth data, center: current tick and refresh counter, right: Result using a morphological closing filter with radius 10. Most image discontinuities are removed by the morphological operation, but some are still visible at the objects' boundaries. (Background image: Historischer Büchersaal im Barockhaus Neißstraße 30, Copyright: René Pech, Görlitzer Anzeiger)

The blue-screen may be started by calling:

```
(animate (curry blue-screen img))
```

Fig. 3 shows the result before and after the application of a closing filter using a filter radius of 10 pixels. The quality of this result is still not perfect but sufficient for the demonstration purpose of this paper since it shows how the data of the Kinect can be used interactively.

It needs to be mentioned, that the `blue-screen` function is not realtime capable with respect to a frame rate of 25 fps. The reason is, that the different image mappings take most of the time, as described in section 3.2 and are not further optimizable. The most time-consuming part of the whole function is indeed the combine images procedure. However, the achieved execution speed of about 1 frame per second is still sufficient for general demonstration purpose.

4.2 Natural pointer interface

The computer mouse has probably been the most common pointing device for human-computer interaction for decades. Due to the electronic mobile revolution with tablets and smartphones, we are now able to use our fingers directly as pointing devices, e.g. by means of (multi-) touch displays. Although this is closer to the "natural pointing" metaphor, it is still artificial and might not be well applicable for general virtual environments. The main limitation is the two-dimensional interface, since virtual worlds are usually not as flat as the displays' touch surfaces. The Microsoft Kinect sensor with its depth image stream provides another alternative for a more natural pointer interface by tracking your fingers movements (cf. [8]).

To further simplify the task of finger detection, we make the following assumptions for this case study:

1. Only a certain range of the depth data is allowed to contain the (depth imaged) finger.
2. A finger pointer is defined as the top- and left-most point of the interval.

Using these assumptions, it is quite clear, that we will detect one pointer position if any object is inside the depth interval of interest. Since it is the top- and left-most position, it is not necessary, that the finger is put in front of other objects, but above them. The first necessary step is to threshold the raw depth image accordingly to the range of the depth interval. Here we assume, that the range consists of all valid depth values, that are closer than t millimeters:

```
(define (closerThan depth_img [t 800])
  (image-map!
     (lambda (val) (if (< 0 val t) 255.0 0.0))
     depth_img))
```

In this function we are able to use the in-place method `image-map!` to save allocation costs, because the original depth values are no longer needed in the further processing. To find the top-left part of the thresholded depth image, we introduce a higher-order function, which applies another function (with arguments `x`, `y` and `value`) over one band of an image while the function is true:

```
(define (band-while p b)
  (let* ((w (band-width b))
         (h (band-height b))
         (found  #f))
    (do ((y 0 (+ y 1)))
        ((or (pair? found) (= y h))
              (if (pair? found)
                  found
                  (cons w h)))
      (do ((x 0 (+ x 1)))
          ((or (pair? found) (= x w)) )
        (when (not (pred x y (band-ref b x y)))
            (set! found (cons x y)))))))
```

This function has also been generalized to images as they are just lists of bands. Since the image is always traversed in column-row-order, this will give us the top-left position for the thresholded depth image `work_img` in an elegant way:

```
(define (topLeft work_img)
  (band-while (lambda (x y v) (= v 0.0))
              (car work_img)))
```

We are now able to use the functions defined above to define a viewing function. This function together with the animation framework included in Racket's `2htdp/universe` module generates the GUI shown in Fig. 4:

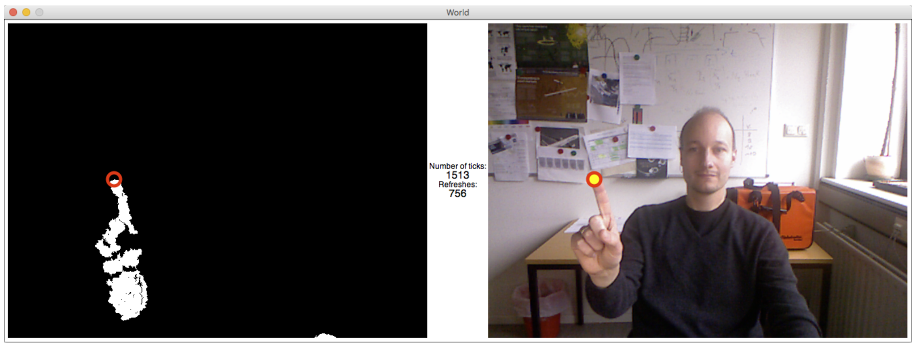


Figure 4: Image taken out of the sequence for the natural pointer approach. Left: Interval thresholded depth data with the top-most, left most point highlighted by an empty red lined circle, center: current tick and refresh counter, right: Result of the tracking overlaid on the RGB image by means of a yellow dot with a red outline.

```
(define ptr_pos '(0 0))
(define radius 10)

(define (live-pointer t [update_each 4])
  (when (= (modulo t update_each) 0)
    (let* ((foo  (grabdepth+video/unsafe dp))
           (wimg (smaller (list (car dp)) 800))
           (tl   (topLeft wimg)))
      (begin
        (image->racket-image wimg d)
        (image->racket-image (cdr dp) p)
        (set! ptr_pos tl))))
  (beside
   (underlay/xy d (- (car ptr_pos) radius)
                  (- (cdr ptr_pos) radius)
                  (marker #f))
   (status t update_each)
   (underlay/xy p (- (car ptr_pos) radius)
                  (- (cdr ptr_pos) radius)
                  (marker #t))))

(animate live-pointer)
```

Here, the function `(marker [filled #t])` is generating a filled or non-filled circle using the `2htdp/image` module. This implementation is using standard VIGRACKET functions and the newly introduced `band-while` function. Since this application does neither need many image allocations nor many reading and writing accesses, it performs comparably well with respect to the execution time. In tests, we found out, that the `live-pointer` procedure is able to refresh the pointer position without lagging 5 times per second.

If the work image and the RGB image are not needed to be shown, e.g. if the debugging is done already, one can further simplify the code to get a fast variant of the `live-pointer` procedure, which runs at realtime (28 frames per seconds):

```
(define (live-pointer-fast t)
  (let* ((foo (grabdepth+video/unsafe dp))
         (pos (band-while (lambda (x y v)
                                   (not (< 0 v 800)))
                   (car dp))))
      (beside
        (status t 1)
        (underlay/xy
          (rectangle 640 480 "solid" "white")
          (- (car pos) radius)
          (- (cdr pos) radius)
          (marker #t)))))
```

5 Conclusions

We have motivated the need for fast interactive development methods in the field of (near-) realtime computer vision. The starting point of our analysis has been the VIGRACKET module, which transforms the functional language Racket into a generic base for testing and developing interactive computer vision algorithms. Although the VIGRACKET module has already been used in research and educational contexts (see [11]), we were able to detect and analyze some few drawback with respect to the execution speed of functional and visualization parts of the module.

We have analyzed these drawbacks and categorized them into different aspects:

- Allocation of shared memory,
- Reading and writing access of shared memory,
- Racket-specific (conversion) functions, and
- Integration of realtime image sources.

We have presented solutions, that were able to improve the VIGRACKET module for each case mentioned above. These improvements have been achieved without destroying other parts of the module, but by extending and modifying mainly low-level functions and introducing new specialized functions inside the C-wrapper library. With the optimization, we are close to the maximum possible execution speed using Racket (cf. sec. 3.2). A new version of the VIGRACKET library containing all of the improvements presented herein, will be released in summer 2016 at GitHub [9].

Moreover, we have have selected the Kinect sensor as a data source to demonstrate the advantages of the new interactive real-time capabilities of the VIGRACKET library. Since the API of OpenKinect's libfreenect library was not directly compatible, we designed a light-weight wrapper, called rackinect. This wrapper provides fast grabbing of RGBD-images from the Kinect directly into VIGRACKET images. A first version of the rackinect will be released under the same account on GitHub in summer 2016, too.

The use of both modules, VIGRACKET and rackinect has been demonstrated by two case studies: Simulating a blue-screen using the Kinect's depth data and an interactive natural pointer interface. The interactive GUI execution of both case studies is only possible due to the optimizations described in this paper. Additionally, both provide a simple but powerful way on processing the image and depth data of a special kind of imaging device. Part of this work have also been performed in preparation for upcoming student practices at the University of Hamburg. Th case studies serve as computer vision examples, which can be understood without or with little knowledge of computer vision. They are suitable for introductory courses of undergraduate Bachelor students, since they are inviting the students to play with the system and explore (near-) realtime computer vision algorithms interactively.

References

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

[2] J. Foster. *The Green Screen Handbook: Real-World Production Techniques*. Taylor & Francis, 2014.

[3] J. Jackman. *Bluescreen Compositing: A Practical Guide for Video & Moviemaking*. Bluescreen Compositing: A Practical Guide for Video & Moviemaking. Focal Press, 2007.

[4] U. Köthe. The VIGRA homepage. Retrieved February 19, 2016 from: `https://github.com/ukoethe/vigra`.

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

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

[7] Z. Liu and C. Xia. *Performance Modeling and Engineering*. Springer US, 2008.

[8] P. Premaratne. *Human Computer Interaction Using Hand Gestures*. Cognitive Science and Technology. Springer Singapore, 2014.

[9] B. Seppke. The vigracket homepage. Retrieved February 19, 2016 from: `https://github.com/bseppke/vigracket`.

[10] B. Seppke. The vigra_c homepage. Retrieved February 19, 2016 from: `https://github.com/bseppke/vigra_c`.

[11] B. Seppke and L. Dreschler-Fischer. Efficient applicative programming environments for computer vision applications: Integration and use of the vigra library in racket. In *Proceedings of the 8th European Lisp Symposium*, 2015.

[12] The OpenKinect team. The OpenKinect project. Retrieved February 19, 2016 from: `https://openkinect.org`.

[13] J. Webb and J. Ashley. *Beginning Kinect Programming with the Microsoft Kinect SDK*. Expert's voice in Microsoft. Apress, 2012.