introduction to easyVision (draft)

Alberto Ruiz[1] Timothy D. Sears[2]

[1]Dept. Informática y Sistemas
Universidad de Murcia

[2]Pingwell
Palo Alto, CA

September 27, 2012

motivation

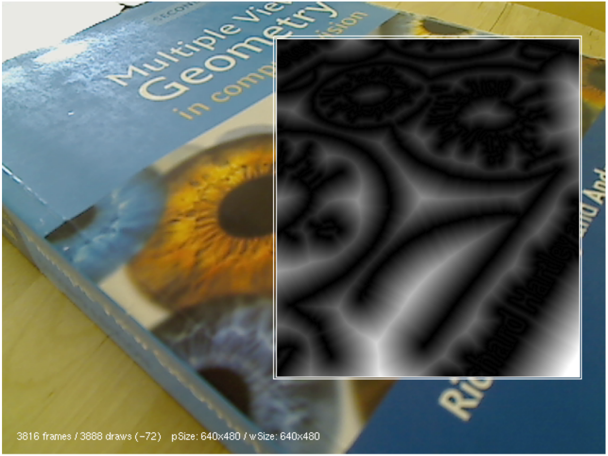
- fast prototyping of computer vision, pattern recognition, and machine learning applications [3, 5, 1, 2].
- supported by low level optimized libraries: **IPP**, **BLAS/LAPACK** (hmatrix [4]), **OpenGL**, and custom **C code** for a few critical performance routines.
- the goal is elegant code: simple and efficient definitions
- basic **Haskell**: most of the time we only need composition of pure functions.

installation

- Haskell platform
- standard cabal packages
- available from github
- soon in Hackage

Installation instructions can be found in the `repository`

real time image processing



3816 frames / 3888 draws (-72) pSize: 640x480 / wSize: 640x480

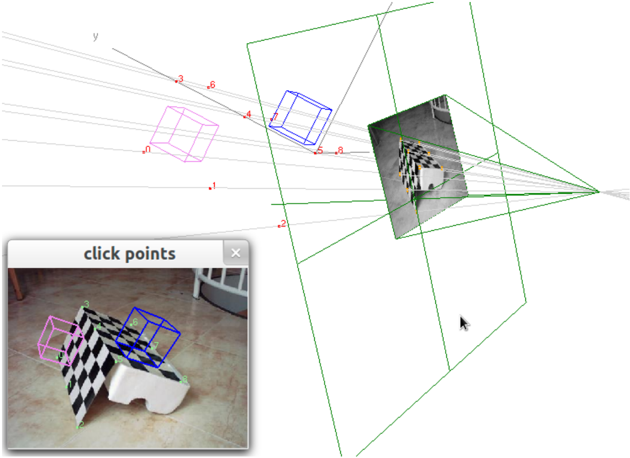
`projects/demos/imagproc.hs`

interface to standard libs

SIFTGPU (Chang Chang Wu)

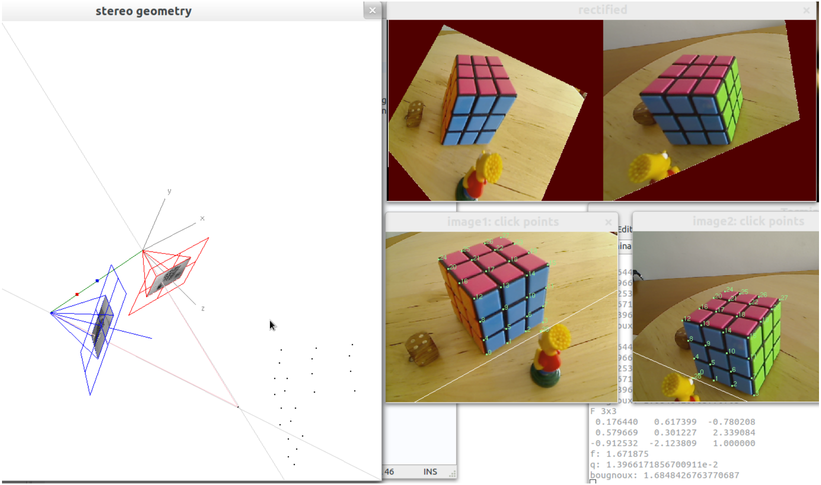
`projects/gpu`

live visual demos



`projects/vision/geom/resection.hs`

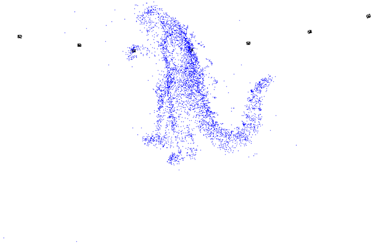
live visual demos

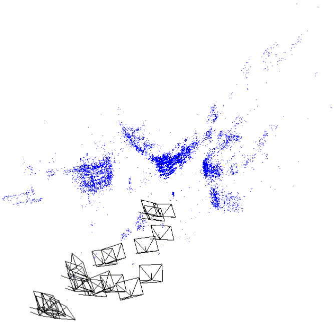


```
F 3x3
 0.176440   0.617399  -0.780208
 0.579669   0.301227   2.339084
-0.912532  -2.123809   1.000000
f: 1.671875
q: 1.3966171856700911e-2
bougnoux: 1.6848426763770687
```

`projects/vision/geom/stereo.hs`

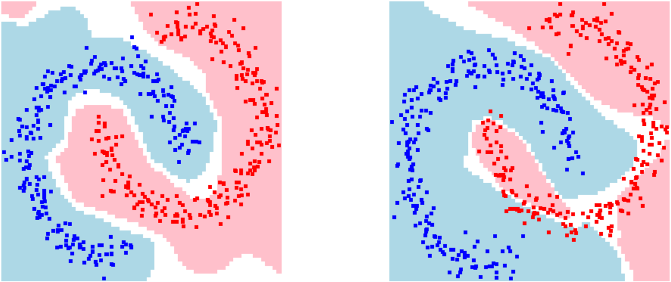
multiview reconstruction





`projects/vision/multiview`

pattern recognition / machine learning



gaussian mixture, 15db NN [10,10,5], 5db

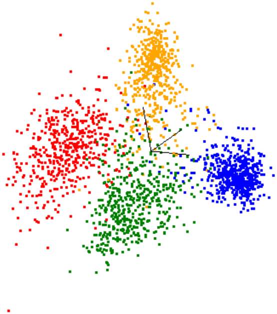
`projects/patrec`

using probability monad

data visualization

MNIST digits (788 dimensions)

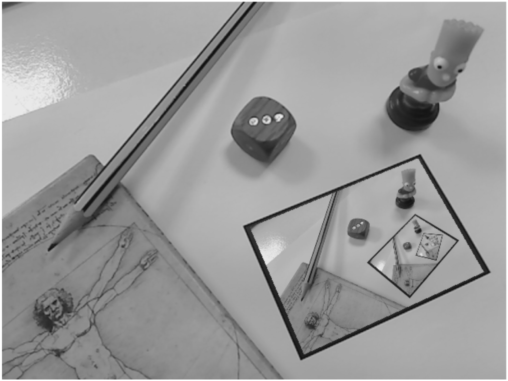
PCA



MDF

press M to autorotate the 3D view

scanl



can this be a Haskell one-liner?

hello world!

play.hs

```haskell
import Vision.GUI (observe, run)
import ImagProc (rgb)

main = run (observe "image" rgb)
```

```
$ ghc --make -O -threaded play.hs
$ ./play
$ ./play video.avi

$ ghci play.hs
  > main
$ runhaskell play.hs
```

Get ready the local documentation

Use `ghci` to see the types (`:t name`, `:i name`)

standard arrow notation

play1.hs

```haskell
import Vision.GUI
import ImagProc

main = run p

p = observe "RGB" rgb >>> arr grayscale >>> observe "inverted" notI
```

- $(\ggg) =$ composition of transformations
- *arr* maps a pure function on the infinite list of objects generated by the camera.

processing pipeline vs observation functions

play1.hs

```
import Vision.GUI
import ImagProc

main = run p

p = observe "RGB" rgb ⋙ arr grayscale ⋙ observe "inverted" notI
```

the processing pipeline *p* produces grayscale images.

the observation windows show some features of the processed objects, but the results are not sent forward.

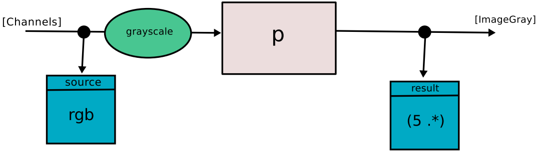
arrow syntax

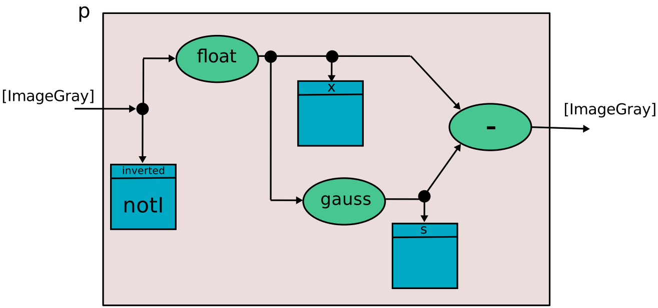
arrows.hs

```
 {-# LANGUAGE Arrows #-}
import Vision.GUI
import ImagProc

main = run  $  observe "source" rgb
                ⋙ arr grayscale
                ⋙ p
                ⋙ observe "result" (5.∗)

p = proc g → do
    let f = float g
    x ← observe "x" id ≺ f
    s ← (observe "s" id ⋘ arr (gaussS 5)) ≺ f
    observe "inverted" notI ≺ g
    returnA ≺ x ⊟ s
```

≺ = `-<` ⊟ = `|-|` (image difference)

diagram


[Channels]
grayscale
p
[ImageGray]
source
rgb
result
(5 .*)


p
[ImageGray]
float
x
-
[ImageGray]
inverted
notI
gauss
s

arrow syntax *

conditional processing paths

choice.hs

```
{-# LANGUAGE Arrows #-}
import Vision.GUI
import ImagProc

main = run $ arrL (zip [0 ..])
           ⋙ separ
           ⋙ observe "final" rgb

separ = proc (k, img) → do
  if odd (k ‘div‘ 25)
    then observe "monochrome" grayscale ≺ img
    else observe "negated" (notI ∘ grayscale) ≺ img
```

(see also `demos/save.hs`)

arrow syntax *

feedback

loop.hs

```
{-# LANGUAGE Arrows #-}
import Vision.GUI
import ImagProc

main = run $ observe "source" rgb
           ⋙ f
           ⋙ observe "result" (5.∗)

f = proc img → do
    let x = (float ∘ grayscale) img
    p ← delay′ ≺ x
    returnA ≺ x ⊟ p
```

`tour/scanl1.hs` and `tour/circuit.hs` are possible solutions to the recursive “image inside image” previous example

arrow type *

$$Generator\ a = IO\ (IO\ (Maybe\ a))$$

$$ITrans\ a\ b = IO\ (IO\ (Maybe\ a) \rightarrow IO\ (Maybe\ b))$$

$$runS :: Generator\ a \rightarrow ITrans\ a\ b \rightarrow IO\ [b]$$

this allows for initialization of windows, user interaction, etc.

similar, but not equivalent to `Kleisli IO`

TO DO: parallel processing

arrL

apply a pure function to the whole (possibly infinite) input list

`grid.hs`

```
import Vision.GUI
import ImagProc
import Util.Misc (splitEvery)
import Data.List (tails)

grid n = map (blockImage ∘ splitEvery n ∘ take (n ∗ n)) ∘ tails

main = run  $  arr (resize (Size 96 120) ∘ rgb)
            ⋙ arrL (grid 5)
            ⋙ observe "grid" id
```

```
arr f = arrL (map f)
```

capture options *

(another example of *arrL*)

play3.hs

```haskell
import Vision.GUI
import ImagProc
import Util.Misc (splitEvery)

main = run $ arrL f >>> observe "RGB" rgb >>> wait (100 `div` 30)

f = concatMap (\x -> x ++ reverse x ++ x) . splitEvery 30
```

- Check on a video, and try `--live` and `--chan`

```
$ ./play3 ../../data/videos/rot4.avi --live
```

```
$ ./play3 ../../data/videos/rot4.avi --chan
```

- Check again the effects with a live webcam:

```
$ ./play3
```

multiple display functions

selected with the mouse wheel

smon.hs

```
import Vision.GUI
import Contours.Base
import ImagProc

main = run $ sMonitor "result" f

f roi x = [ msg "grayscale"        [Draw g]
          , msg "gaussian filter " [Draw smooth]
          , msg "canny edges"      [Draw (notI edges)]]
  where
    img    = rgb x
    g      = setRegion roi (grayscale x)
    smooth = gauss Mask5x5 ∘ float $ g
    edges  = canny (0.1, 0.3) ∘ gradients $ smooth

    msg s t = Draw [Draw img, Draw t, color yellow $ text (Point 0.9 0.65) s]
```

the display function also receives the region of interest

see also `demos/imagproc.hs`

window controls

In all windows you can:

- Zoom (CTRL-wheel / CTRL-Left click drag)
- Freely resize the window
- Pause (SPACE)
- Change the region of interest (CTRL-Right click drag).

(see below for advanced options)

check on the previous example

async mode *

`play1.hs`

```
import Vision.GUI
import ImagProc

main = run p

p = observe "RGB" rgb ⋙ arr grayscale ⋙ observe "inverted" notI
```

- The ‘main’ process grabs and transform **all** frames, as fast as possible.
- Window display is made in a separate thread. Some frames can be dropped for display if they come too fast.
- Press F10 to display at 5Hz (async mode).

test with

```
$ ./play1 '../../data/videos/rot4.avi -benchmark'
```

freqMonitor *

freqMonitor shows the achieved frame rate

play4.hs

```
import Vision.GUI
import ImagProc

main = run $ observe "RGB" rgb >>> freqMonitor
```

(built-in in *run* with command line option `--freq`, check on the previous example)

CTRL-F3 to disable any window

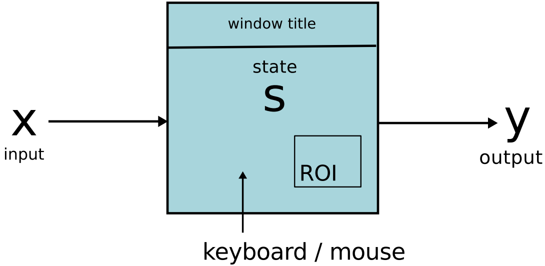
more general cameras *

We can process streams of any object, not necessarily images.

play5.hs

```
import Vision.GUI
import ImagProc.Base
import Data.Time (getCurrentTime, UTCTime)
import Control.Concurrent (threadDelay)

main = runT_ clock see

see :: Show x ⇒ ITrans x x
see = observe "time" (text (Point 0.9 0) ∘ show)

clock :: IO (IO (Maybe UTCTime))
clock = return (threadDelay 10000 ≫ Just ‘fmap‘ getCurrentTime)
```

interface: generic stateful UI window



Defined by

$$\begin{aligned} &\mathit{result} :: \mathit{ROI} \to s \to x \to (s, y) \\ &\mathit{show} :: \mathit{ROI} \to s \to x \to y \to \mathit{Drawing} \\ &\mathit{updts} :: [\mathit{Key} \to \mathit{ROI} \to \mathit{Point} \to s \to s] \end{aligned}$$

(There are also IO actions and first-time initialization)

example

add points clicked by the user and invert the image

`interface.hs`

```
import Vision.GUI hiding (clickPoints)
import ImagProc

main = run clickPoints

clickPoints :: ITrans Channels ([Point], ImageGray)
clickPoints = transUI $ interface (Size 240 320) "click points"
                               state0 firsttime updts acts result display
  where
    state0 = []
    firsttime _ _ = return ()
    updts = [(key (MouseButton LeftButton), \_roi pt pts → pt : pts)]
    acts = []
    result _roi pts input = (pts, (pts, notI ∘ grayscale $ input))
    display _roi _pts _input (pts, x) = Draw [Draw x, drwpts]
      where drwpts = (color green ∘ pointSz 3) pts
```

`observe`, `sMonitor`, etc., are special cases

interactive parameters

param2.hs

```
{-# LANGUAGE TemplateHaskell, RecordWildCards #-}
import Vision.GUI
import ImagProc

autoParam "SParam" "g-" [("sigma","Float",realParam 3 0 20)
                        ,("scale","Float",realParam 1 0 5)]

main = run $ arr grayscale
           ⋙ withParam g
           ⋙ observe "gauss" id

g SParam {..} = (scale.∗) ∘ gaussS sigma ∘ float
```

autoParam automagically creates the parameter record, the interactive window, and support for command line arguments.

withParam supplies the 'current' value to a pure function

command line arguments

The initial value of any parameter can be set in the command line:

```
$ ./ param --g-sigma=1.5
```

all parameter windows can be removed:

```
$ ./ param --default
```

`observe` and `sMonitor` windows can be selectively removed:

```
$ ./ param --no-gauss
```

`--options` shows all options recognized by a program

standard input video sequences

argument	**type**	**implementation**
uvc0, uvc1, . . .	UVC webcams	native
tv://	webcam (/dev/video0)	via mplayer
'tv:// -tv device=/dev/video3'	webcam (/dev/video3)	
path/to/any/video.avi	video file	
'quoted mplayer commands'	any mplayer source	
- -photosmp=/path/to/folder/	jpg or png images	
- -photos=folder	jpg or png images	imagemagick
- -sphotos=folder	jpg or png images	imagemagick

alias in `$EASYVISION/cameras.def`
the first one is used by default

Options:

- -size=n	×32 (aspect ratio 4/3, 20=640x480)
- -rows=r - -cols=c	in pixels
- - live	get most recent frame
- - chan	read from channel, no frame is lost

standard key bindings

ESC	exit program
I	save screenshot
SPACE	pause
Ctrl-Mouse-Wheel	zoom (also Ctrl-Up/Down)
Ctrl-Mouse-Left	move zoom
Ctrl-0	reset zoom
M	auto rotate (3D window)
O	reset view (3D window) (FIXME)
Shift-Wheel	rotate view (3D window)
Ctrl-Mouse-Right	set ROI
Alt-0	reset ROI
F11	toggle show ROI and display stats
F3	toggle window resize mode
F10	toggle sync display
Shift-SPACE	pause (pass through)
S	step (advance one frame)
Ctrl-ESC	exit main loop only (useful in ghci)
Shift-ESC	leave loop and kill mplayer (FIXME)
Ctrl-F3	minimize (and inhibit display)

pause

pause modes

- normal pause (SPACE): the incoming process is stopped.
- *drawing* pause (SHIFT-SPACE): the data flow is not interrupted, but the window keeps the current drawing.
- step by step (S): the data flow advances one single frame.

batch processes

`run = runT_ camera`

`runT_` runs the arrow process inside the GUI, discarding the output

`runS` works without any GUI

runS.hs

```
import Vision.GUI
import ImagProc

main = do
   r ← runS camera $ arr (size ∘ grayscale)
   print $ take 10 r
```

batch processes

the result is produced lazily

nogui.hs

```
import Vision.GUI
import ImagProc

main = do
  putStrLn "Working without GUI..."
  x ← runS camera  $  arr (sum8u ∘ grayscale)
                   ⋙ arrL (zip [1 . .] ∘ take 1000)
  print x
```

```
$ ./nogui ’../../data/videos/rot4.avi -benchmark’
Working without GUI...
YUV4MPEG2 W640 H480 F30:1 Ip A1:1
[(1,3.1425134e7),(2,3.1350479e7),(3,3.1274583e7), ...
...,(999,3.3603773e7),(1000,3.3621203e7)]
```

batch processes *

The process can also be monitorized:

batch.hs

```
import Vision.GUI
import ImagProc

main = do
  x ← runT camera $ observe "image" rgb
                  ⋙ arr (sum8u ∘ grayscale)
  print (sum $ take 1000 x)
```

```
$ ./batch '../../data/videos/rot4.avi -benchmark'
YUV4MPEG2 W640 H480 F30:1 Ip A1:1
3.25630728e10
batch: GraphicsziUIziGLUTziCallbacksziWindow_dhXC: interrupted
```

batch processes *

working with individual images:

single.hs

```
import Vision.GUI
import ImagProc
import ImagProc.Camera
import System.Environment

f = sum8u ∘ grayscale

main = getArgs ≫= readImages ≫= runITrans (arr f) ≫= print
```

```
$ ./single ../../data/images/transi/dscn2070.jpg ../../data/images/tra
[3.29684731e8,3.25189723e8]
```

arrIO *

lift an IO action

arrIO.hs

```
import Vision.GUI
import ImagProc

main = run $ observe "img" rgb >>> arrIO (print . size . grayscale)
```

IMPORTANT:

- `arrIO` should be used in a pipeline with at least one window. Otherwise the GLUT main loop invoked by `run` exits immediately.
- alternative for no GUI apps: `runS` or `runITrans`

standalone windows

stand1.hs

```
import Vision.GUI
import ImagProc

main = runIt win

win = standalone (Size 100 400) "click to change" x0 updts [] sh
  where
    x0 = 7
    sh = text (Point 0 0) ∘ show
    updts = [(key (MouseButton LeftButton), λroi pt → (+1))]
```

- contains a state that can be interactively modified
- the update function receives the ROI of the window and mouse position

browser

a standalone window for showing a finite list of objects

stand2.hs

```
import Vision.GUI
import ImagProc

main = runIt win

win = browser "odd numbers" xs sh
  where
    xs = [1, 3 .. 21]
    sh _k = text (Point 0 0) ∘ show
```

select the element in the list with the mouse wheel[1]

---

[1] or with Key Up / Down

editor

a standalone window for modification of a list of objects

stand3.hs

```
import Vision.GUI
import ImagProc
import Util.Misc (replaceAt)

main = runIt win

win = editor update save "editor" [2, 4 . . 10] sh
  where
    sh k x = Draw [color white $ text (Point 0 0) (show x)
                  , color yellow $ text (Point 0.9 0.8) ("#  " ++ show k)]
    update = [op (Char '+') succ
             , op (Char '-') pred]
    save = [(ctrlS, \_roi _pt (_k, xs) → print xs)]
    ctrlS = kCtrl (key (Char '\DC3'))
    op c f = updateItem (key c) (const ∘ const $ f)
```

connectWith

state changes can be propagated:

`connect.hs`

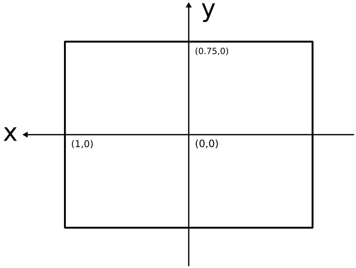
```
import Vision.GUI
import ImagProc

main = runIt $ do
    p ← click "click points"
    w ← browser "work with them" [] (const Draw)
    connectWith g p w

g _ pts = (0, [map (Segment (Point 0 0)) pts])

click name = standalone (Size 400 400) name [] updts [] sh
  where
    updts = [(key (MouseButton LeftButton), \_p ps → ps ⧺ [p])]
    sh = color yellow ∘ drawPointsLabeled
```

see also `tour/clickPoints.hs` and `vision/geom`

graphic primitives

we `Draw` different objects (using an *existential* type and a `Renderable` class):

- `Image`
- `Polyline = Closed [Point] | Open [Point]`
- `[Point]`, `[Segment]`, `HLine`, etc.
- `text Point String`
- with attributes: `color`, `pointSz`, `lineWd`



unusual coordinate system, $z =$ depth (for eqs. in [3])

drawing example

draw.hs

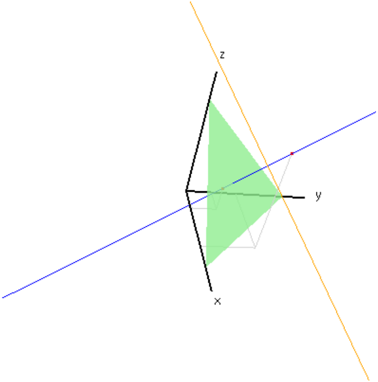
```
import Vision.GUI
import ImagProc

main = runIt $ browser "points & lines" xs (const id)
  where
    xs = [drawing1]

drawing1 :: Drawing
drawing1 = Draw [color yellow (HLine 0.1 1 0)
                ,pointSz 5 [Point 0.5 0.5, Point 0 (−0.2)]
                ]
```

see `demos/conjrot.hs` for an example of a drawing with interactive parameters

3D

- `viewPoint`
- `monitor3D`
- `browse3D`

strongly typed geometry



type safe composition and application of transformations, meet and join of points, lines, and planes, etc.
(`draw3DParam.hs`)

demos

../demos/mirror.hs

```
import Vision.GUI
import ImagProc

main = run $ observe "Mirror" (mirror ∘ grayscale)

mirror im = blockImage [[im1, mirror8u 1 im1]] where
    Size h w = size im
    roi = (theROI im) {c2 = div w 2}
    im1 = resize (Size h (div w 2)) (modifyROI (const roi) im)
```

demos

../demos/points.hs

```
import Vision.GUI
import ImagProc

main = run $ arr (grayscale ⋙ id &&& interest)
           ⋙ observe "Corners" sh

sh (im, pts) = [Draw im, pointSz 5 ∘ color red $ pts]

interest :: ImageGray → [Point]
interest g = pixelsToPoints (size g) ∘ getPoints32f 300 ∘ localMax 1
           ∘ thres 0.5 ∘ salience 2 4 ∘ float $ g
  where
    thres r im = thresholdVal32f (mx ∗ r) 0 IppCmpLess im
      where (_, mx) = minmax im
    salience s1 s2 = gaussS s2 ∘ sqrt32f ∘ abs32f ∘ hessian ∘ gradients ∘ gaussS s1
```

demos

run `make; make demo` in `/projects/demos`

`demos/warp.hs`

`demos/transi.hs`

`demos/hessharr.hs`

`demos/imagproc.hs`

`demos/pose.hs`

`demos/spline.hs`

adding new IPP functions

1. add function name to `functions.txt`

```
ippi.h  ippiXor_8u_C1R
```

2. `runhaskell adapter.hs`
3. create pure interface with desired ROI policy in `Pure.hs`

```
-- | image XOR, pixel by pixel
xorI :: ImageGray -> ImageGray -> ImageGray
xorI = mkInt ioXor_8u_C1R
```

- `adapter.hs` uses a custom C header parser using *Parsec*
- some functions may need an `AdHoc.hs` interface or `C code`

low level processing in C

`contrib.hs`

```
import Vision.GUI
import ImagProc
import ImagProc.Contrib.Examples

main = run $ observe "C wrapper test" (f ∘ grayscale)

f x = Draw [Draw (invertInC x)
           , text (Point 0 0) (show (sum8u x, sumInC x))]
```

see package `contrib/examples` for examples of FFI with images

export a Haskell library to C

a simple example is in $\boxed{\texttt{contrib/export}}$

modules overview

Util
Classifier
Vision
Vision.GUI
ImagProc

future work

- WiP: static checking of geometric types
- parallel processing
- reorganize documentation, improve function names
- mobile version (MeV)
- FRP
- repa
- OpenCV
- vlfeat, other packages . . .

references

- C.M. Bishop.
  Pattern Recognition and Machine Learning.
  Springer, 2006.
- R.C. Gonzalez and R.E. Woods.
  Digital Image Processing (2nd).
  Prentice Hall, 2002.
- Richard Hartley and Andrew Zisserman.
  Multiple View Geometry in Computer Vision.
  Cambridge University Press, 2 edition, 2003.
- A. Ruiz.
  Introduction to *hmatrix*, 2010.
  `http://perception.inf.um.es/hmatrix`.
- Richard Szeliski.
  Computer Vision: Algorithms and Applications.
  Springer, 2010.