OpenCV meets Clojure: Motivation

Short Sample

Interactive OpenCV Introduction

Color Maps

Cartoon Cats

More Api Examples

Find shapes with hough circles

Using hough techniques to find lines and circles

Finding only pockets on a pool table

Akaze

Simple Foreground Background Diff

Blur Detection

Tennis Ball

Using Hough Circles

Method2: Using Find Contours

Pencil Sketch

Load the picture that will be sketched

Apply a Canvas effect

Landscape Art Adventures

smoothing the picture using a bilateral filter

detect and enhance edges

Combine color image with edge mask

Playing with contours

turning all this in functions

OpenCV meets Clojure: Motivation

OpenCV is quite well known in the domain of image manipulation. It has all those secret boots you can wear and use to apply filters, resize, rotate, find shapes, remove background etc ...

In those days of neural networks, image pre-analysis is actual a fundamental step in preparing a training set.

While OpenCV is fantastic to achieve all those tasks, its also quite a pain to re-compile and make the install manually on each machine you want to run it, and I do feel this is a gigantic wall for most of the people out there who just want to use it.

Short Sample

Welcome to this tutorial. This is a short sample to show how to use this fantastic API to do OpenCV stuff in the browser.

```
(ns composed-pine
   (:require
    [opencv3.utils :as u]
    [opencv3.core :refer :all]))
```

```
nil
```

```
(defn load-cvt-resize-show[url]
(-> url
  (u/mat-from-url)
  (cvt-color! COLOR_RGB2GRAY)
  (resize! (new-size 150 200))
  (u/mat-view)))
```

```
#'composed-pine/load-cvt-resize-show
```

Interactive OpenCV Introduction

Manipulate images with clojure and opencv

```
(ns spacial-dusk
  (:require
    [opencv3.core :refer :all]
    [opencv3.utils :as u]))
```

```
nil
```

```
(def neko
  (u/mat-from-url "https://s-media-cache-
ak0.pinimg.com/236x/10/2c/75/102c756d7e808deff666f3edf540abba.jpg"))
(u/mat-view neko)
```

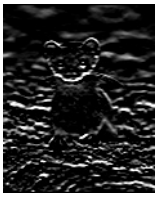
```
(def gray (new-mat))
(cvt-color neko gray COLOR_RGB2GRAY)
(u/mat-view gray)
```

```
(def small-gray-neko
  (u/resize-by gray 0.5))
(u/mat-view small-gray-neko)
```

```
(def equalized (new-mat))
(equalize-hist small-gray-neko equalized)
(u/mat-view equalized)
```

```
(def dilated (u/mat-from small-gray-neko))
(def dilation-size 2)
(def element (get-structuring-element  MORPH_RECT
  (new-size (inc (* 2 dilation-size)) (inc (* 2 dilation-size)))))
(dilate small-gray-neko dilated element)
(u/mat-view dilated)
```

```
(def sobelled (u/mat-from small-gray-neko))
(sobel small-gray-neko sobelled -1 0 1)
(u/mat-view sobelled)
```



```
(def bit_not (u/mat-from small-gray-neko))
(bitwise-not small-gray-neko bit_not)
(u/mat-view bit_not)
```

Color Maps

This short tutorial will show you how to change the color maps of loaded pictures. A colormap is defined by a m-by-3 matrix of real numbers between 0.0 and 1.0. Each row is an RGB vector that defines one color.

This follows the story of kutsushita, the cat that likes to change color.

```
; first we import the usual namespaces.
(ns affectionate-thorns
  (:require
    [opencv3.utils :as u]
    [opencv3.colors.rgb :as color]
    [opencv3.core :refer :all]))
```

```
nil
```

Changing color space for a default color space is done through the function **apply-color-map!** from opencv3.core

The mat as well as the colormap id is passed as parameter.

Using the chaining method we can apply the BONE color map using the code below:

```
(-> "resources/images/cats/onsofa.jpg"
  imread
  (u/resize-by 0.5)
  (apply-color-map! COLORMAP_BONE )
  (u/mat-view))
```

You just saw one colormap, but a few other default color maps are available by default. The array below is a list of all the available color maps through OpenCV core.

```
(def colors-maps
[
 "COLORMAP_HOT"
 "COLORMAP_HSV"
 "COLORMAP_JET"
 "COLORMAP_BONE"
 "COLORMAP_COOL"
 "COLORMAP_PINK"
 "COLORMAP_RAINBOW"
 "COLORMAP_OCEAN"
 "COLORMAP_WINTER"
 "COLORMAP_SUMMER"
 "COLORMAP_AUTUMN"
 "COLORMAP_SPRING"])
```

```
#'affectionate-thorns/colors-maps
```

```
(defn watermark [ source text]
  (put-text! source
    text
    (new-point 30 30) FONT_HERSHEY_PLAIN 1 (new-scalar 255 255 255) 2))

(defn change-color [ source color-map-string ]
  (->
  source
  clone
  (apply-color-map! (eval (read-string (str "opencv3.core/" color-map-string))))
  (watermark color-map-string)))

(defn apply-all-colormaps [ source ]
   (cons
    (-> source clone (watermark "ORIGINAL"))
    (map (partial change-color source) colors-maps)) )
```

```
#'affectionate-thorns/apply-all-colormaps
```

```
(def source
  (-> "http://sites.psu.edu/siowfa15/wp-content/uploads/sites/29639/2015/10/cat.jpg"
    (u/mat-from-url)
    (u/resize-by 0.1)))
```

```
#'affectionate-thorns/source
```

```
(def targets (apply-all-colormaps source))
(u/mat-view (vconcat! targets))
```

Defining your own color space is also possible using the function **transform!**. transform needs a matrix doing the mapping from rgb to some mapping. The first line of the matrix is the blue value, the second line is the green value, the third value is the red value.

The below matrix makes so that all red pixels are turned to green.

```
(def custom (u/matrix-to-mat [
  [1 0 0] ; blue
  [0 1 0] ; green
  [0 1 0] ; red
  ]))

(-> img
  clone
  (transform!  custom)
  (u/mat-view ))
```

You could use the same technique to define your own sepia filter.

```
(def img (-> "resources/images/cats/onsofa.jpg" imread (u/resize-by 0.5)))

(def blue-sepia (u/matrix-to-mat [
  [0.393 0.769 0.189] ; blue
  [0.349 0.686 0.168] ; green
  [0.272 0.534 0.131] ; red
  ]))

(-> img
  clone
  (transform!  blue-sepia)
  (u/mat-view ))
```

And a regular sepia is done using a similar matrix.

```
(def sepia-2 (u/matrix-to-mat [
  [0.131 0.534 0.272]
  [0.168 0.686 0.349]
  [0.189 0.769 0.393]
  ]))

(-> img
  clone
  (transform!  sepia-2)
  (u/mat-view ))
```

Cartoon Cats

Because we all love cats and cartoon ...

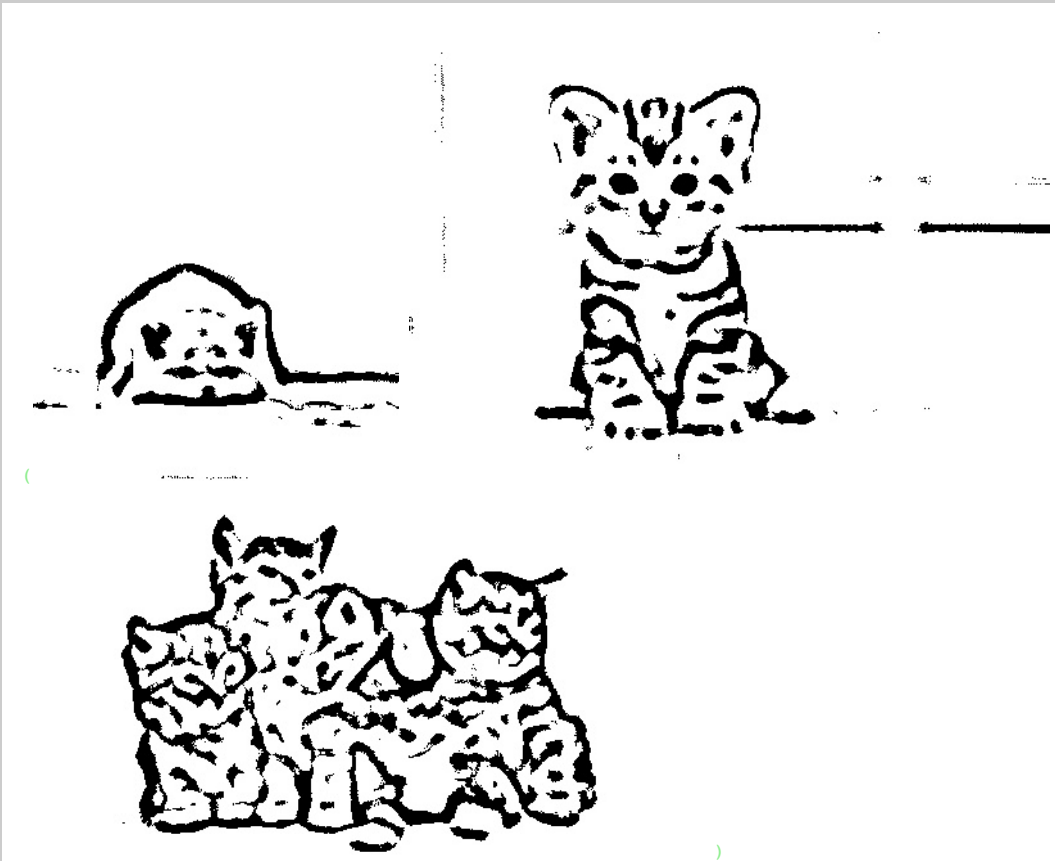
```
(ns opencv3.cartoon2
  (:require
   [opencv3.core :refer :all]
   [opencv3.utils :as u]))
```

```
nil
```

```
(defn cartoon
  [buffer]
  (->
    (cvt-color! buffer COLOR_RGB2GRAY))
    (let [ c (clone buffer)]
    ; (bilateral-filter buffer c 9 9 7)
    (bilateral-filter buffer c 10 250 50)
    (-> c
     (median-blur! 7)
     (adaptive-threshold! 255 ADAPTIVE_THRESH_MEAN_C THRESH_BINARY 9 3)
     (cvt-color! COLOR_GRAY2RGB))))

(defn cartoon-me[url]
 (-> url
  u/mat-from-url
  (u/resize-by 0.5)
  cartoon
  u/mat-view))
```

```
#'opencv3.cartoon2/cartoon-me
```

```
(map
  cartoon-me
  ["http://www.petmd.com/sites/default/files/sleepy-cat-125522297.jpg"
   "http://www.readersdigest.ca/wp-content/uploads/2011/01/4-ways-cheer-up-depressed-
cat.jpg"
   "https://hyatoky.com/wp-content/uploads/cute-cat-wallpapers-hd-1080x580.jpg"])
```



```
```

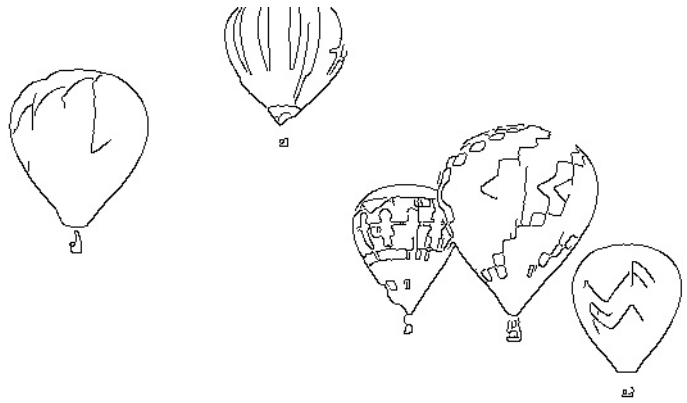
More Api Examples

We have seen very simple syntax usage, now its time to make use of those pipelines.

```
(ns opencv3.dolphin
(:require
  [opencv3.utils :as u]
  [opencv3.core :refer :all]))
```

```
nil
```

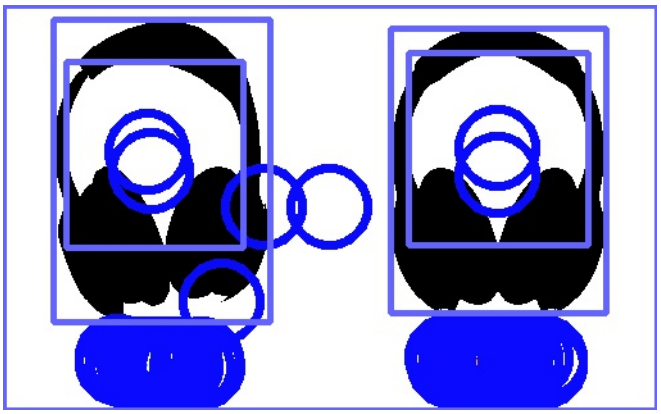
```
(-> "http://www.v3wall.com/wallpaper/1366_768/0912/1366_768_20091223010850201138.jpg"
 u/mat-from-url
 (u/resize-by 0.4)
 (median-blur! 3)
 (cvt-color! COLOR_BGR2GRAY)
 (canny! 300.0 100.0 3 true)
 (bitwise-not!)
 (u/mat-view))
```



```
(-> "resources/images/cat.jpg"
  imread
  (clone)
  (u/resize-by 0.3)
  (median-blur! 7)
  (cvt-color! COLOR_BGR2GRAY)
  bitwise-not!
  u/mat-view)
```

```
(def headphones (->
    "resources/morph/headphone.png"
    imread
    clone
   (cvt-color! COLOR_BGR2GRAY)))

(def contours (new-arraylist))
(find-contours headphones contours (new-mat) RETR_LIST CHAIN_APPROX_SIMPLE)

(doseq [c contours]
  (if (> (contour-area c) 100 )
    (let [ rect (bounding-rect c)]
      (if (> (.height rect) 28)
      (rectangle
        image-c
        (new-point (.x rect) (.y rect))
        (new-point (+ (.width rect) (.x rect)) (+ (.y rect) (.height rect)))
        (new-scalar 255 100 100)
        3)))))
```

```
(u/mat-view image-c)
```



Find shapes with hough circles

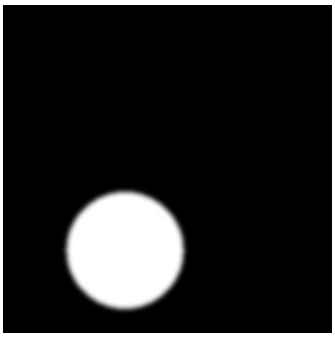
This tutorial shows how to find shape with a given color within an image.

```
(ns combative-atoll
  (:require
   [opencv3.core :refer :all]
   [opencv3.utils :as u]))
```

```
nil
```

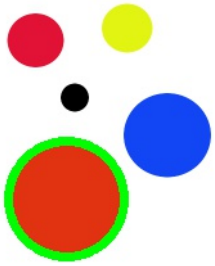
```
(def bgr-image
  (-> "resources/detect/circles.jpg" imread (u/resize-by 0.5) ))
(u/mat-view bgr-image)
```

```
(def ogr-image
  (-> bgr-image
   (clone)
   (median-blur! 3)
   (cvt-color! COLOR_BGR2HSV)
   (in-range! (new-scalar 0 100 100) (new-scalar 10 255 255))
   (gaussian-blur! (new-size 9 9) 2 2)))
(u/mat-view ogr-image)
```



Let's find the circles using the **hough-circles** function from opencv core.

```
(def circles (new-mat))
(hough-circles ogr-image circles CV_HOUGH_GRADIENT 1 (/ (.rows bgr-image) 8) 100 20 0 0)
(dotimes [i (.cols circles)]
  (let [ circle (.get circles 0 i) x (nth circle 0) y (nth circle 1) r (nth circle 2)  p
(new-point x y)]
  (opencv3.core/circle bgr-image p (int r) (new-scalar 0 255 0) 5)))
(u/mat-view bgr-image)
```



Using hough techniques to find lines and circles

Here we learn how to use opencv's hough lines

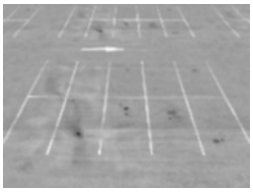
```
(ns opencv3.hough
  (:require
    [opencv3.utils :as u]
    [opencv3.colors.rgb :as color]
    [opencv3.core :refer :all]))
```

```
nil
```

Let's load the target image, and convert it to gray with a bit of blur, so that canny can work on it.

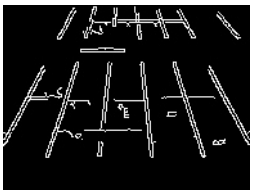
```
(def parking (-> "resources/images/lines/parking.png" imread (u/resize-by 0.3)))
(u/mat-view parking)
```

```
(def gray       (-> parking clone (cvt-color! COLOR_BGR2GRAY) (gaussian-blur! (new-size 3
3) 0 ) ))
(u/mat-view gray)
```



Edges detection is done using canny.

```
(def edges      (-> gray clone (canny! 50 150 )))
(u/mat-view edges)
```



Let's setup the parameters for the hough lines detection.

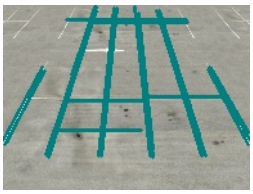
```
(def rho 1) ; distance resolution in pixels of the Hough grid
(def theta  (/ Math/PI 180)) ;   angular resolution in radians of the Hough grid
(def min-intersections 15) ;   minimum number of votes (intersections in Hough grid cell)
(def min-line-length  50) ;   minimum number of pixels making up a line
(def max-line-gap  20 ) ;  maximum gap in pixels between connectable line segments

(def lines (new-mat))
(hough-lines-p edges lines rho theta min-intersections min-line-length max-line-gap)
```

```
nil
```

Now that we have the lines, let's draw them on a clone of the original picture.

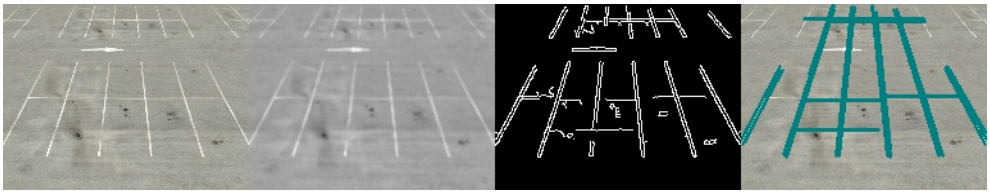
```
(def result (clone parking))
(dotimes [ i (.rows lines)]
(let [ val (.get lines i 0)]
  (line result
    (new-point (nth val 0) (nth val 1))
    (new-point (nth val 2) (nth val 3))
    color/teal
    2)))
```

```
(u/mat-view result)
```



Finally, output all the intermediate images used in this guide. Let's not forget that hconcat and vconcat need all the pictures to be in the same format, in our case, the same number of channels. We convert back *gray* and *edges* with the proper number of channels.

```
(u/mat-view (hconcat! [
  parking
  (-> gray clone (cvt-color! COLOR_GRAY2RGB))
  (-> edges clone (cvt-color! COLOR_GRAY2RGB))
  result] ))
```



Finding only pockets on a pool table

```
(def pool
  (->
    "https://raw.githubusercontent.com/badlogic/opencv-fun/master/data/topdown-6.jpg"
    u/mat-from-url
    (u/resize-by 0.5)))
(u/mat-view pool)
```

```
(def gray (-> pool clone (cvt-color! COLOR_BGR2GRAY)))
(def minRadius 14)
(def maxRadius 18)
(def circles (new-mat))
(hough-circles gray circles CV_HOUGH_GRADIENT 1 minRadius 120 10 minRadius maxRadius)
```

```
nil
```

```
(def output (clone pool))
(dotimes [i (.cols circles)]
  (let [ circle (.get circles 0 i) x (nth circle 0) y (nth circle 1) r (nth circle 2)  p
(new-point x y)]
  (opencv3.core/circle output p (int r) color/red-2 1)))
(u/mat-view output)
```

Akaze

Find an image inside another image.

```
(ns wandering-moss
  (:require
    [opencv3.core :refer :all]
    [opencv3.utils :as u])
  (:import
    [org.opencv.features2d Features2d DescriptorExtractor DescriptorMatcher
FeatureDetector]))

(def detector (FeatureDetector/create FeatureDetector/AKAZE))
(def extractor  (DescriptorExtractor/create DescriptorExtractor/AKAZE))
```

```
#'wandering-moss/extractor
```

```
(def original (-> "resources/images/cat.jpg" imread (u/resize-by 0.3)))
(def mat1 (clone original))
(def points1 (new-matofkeypoint))
(.detect detector mat1 points1)

(def show-keypoints1 (new-mat))
(Features2d/drawKeypoints mat1 points1 show-keypoints1 (new-scalar 255 0 0) 0)
(u/mat-view show-keypoints1)
```

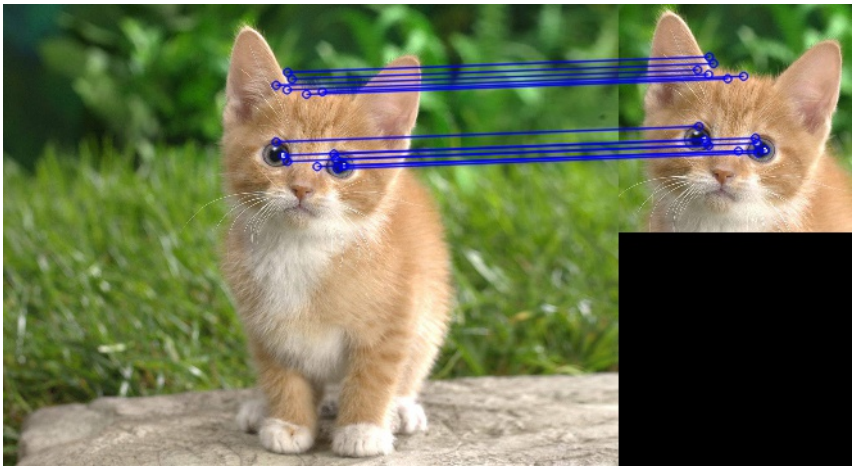
```
(def desc1 (new-mat))
(.compute extractor mat1 points1 desc1)
(u/mat-view desc1)
```

```
(def mat2 (-> "resources/images/cat_face.jpg" imread (u/resize-by 0.3)))
(def points2 (new-matofkeypoint))
(.detect detector mat2 points2)

(def show-keypoints2 (new-mat))
(Features2d/drawKeypoints mat2 points2 show-keypoints2 (new-scalar 255 0 0) 0)
(u/mat-view show-keypoints2)

(def desc2 (new-mat))
(.compute extractor mat2 points2 desc2)
```

```
nil
```

```
(def matcher
  (DescriptorMatcher/create DescriptorMatcher/BRUTEFORCE_HAMMINGLUT))
(def matches
  (new-matofdmatch))
(.match matcher desc1 desc2 matches)

(defn best-n-dmatches2[dmatches]
  (new-matofdmatch
    (into-array org.opencv.core.DMatch
      (filter #(< (.-distance %) 10) (.toArray dmatches)))))

(defn draw-matches [_mat1 _points1 _mat2 _points2 _matches draw-method]
  (let[ _mat (new-mat (* 2 (.rows _mat1)) (* 2 (.cols _mat1)) (.type _mat1))
        _sorted-matches (best-n-dmatches2 _matches)]

    (Features2d/drawMatches
              _mat1
              _points1
              _mat2
              _points2
```

```
               _sorted-matches
               _mat
               (new-scalar 255 0 0)
               (new-scalar 0 0 255)
               (new-matofbyte)
               draw-method)
      _mat))

(defn is-a-match [dmatches]
  (> (count (filter #(< (.-distance %) 10) (.toArray dmatches))) 0))

; (is-a-match matches)

(u/mat-view
  (draw-matches mat1 points1 mat2 points2 matches Features2d/NOT_DRAW_SINGLE_POINTS))
```



Simple Foreground Background Diff

Here we have a background picture and we would like to discover new objects on top of this background.

We will use the simple opencv3.core function **absdiff** and then apply a simple and large **threshold** to pick up all the new objects.

The reference clojure code file can be found here: bgdiff.clj

Let's see first what our background looks like.

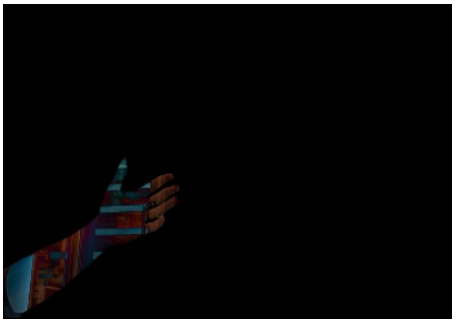
```
(ns scenic-iceberg
  (:require
    [opencv3.core :refer :all]
    [opencv3.utils :as u]))

(def bg (-> "resources/images/bgdiff/header.png" imread (u/resize-by 0.5)))
(u/mat-view bg)
```

Then put our hand in front of that background.

```
(def fg (-> "resources/images/bgdiff/front.png" imread (u/resize-by 0.5)))
(u/mat-view fg)
```

The **absdiff** function from OpenCV core is used, and we can directly see the output.

```
(def output (new-mat))
(absdiff bg fg output)
(u/mat-view output)
```



```
; diff in color
(def fg-1
  (-> output
  clone
  (threshold! 10 255 1)))
(u/mat-view fg-1)
```

You can then use a very permissive threshold to find out relevant shapes.

```
; diff in gray
(def fg-2
 (-> output
  clone
  (cvt-color! COLOR_RGB2GRAY)
  (threshold! 10 255 1)))
(u/mat-view fg-2)
```

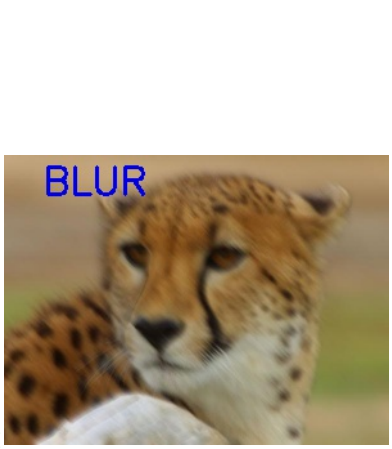
Blur Detection

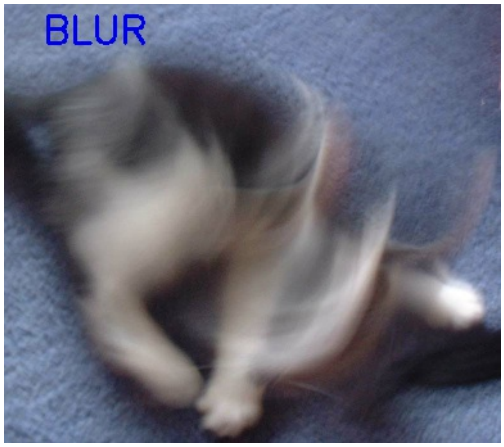
Is my image blurred ?

```
(ns wealthy-darkness
 (:require
     [opencv3.core :refer :all]
     [opencv3.colors.rgb :as color]
     [opencv3.utils :as u]))


;
; laplacian variation
;
; https://stackoverflow.com/questions/36413394/opencv-variation-of-the-laplacian-java
; http://www.pyimagesearch.com/2015/09/07/blur-detection-with-opencv/
;
; used to detect blur in an image

(def img
  (-> "resources/images/cat.jpg" imread))

(def kernel
  (u/matrix-to-mat
  [ [0 -1 0]
    [-1 4 -1]
    [0 -1 0]]))

(filter-2-d! img -1 kernel)
(def std (new-matofdouble))
(def median (new-matofdouble))
(mean-std-dev img median std)

(Math/pow (first (.get std 0 0)) 2)


;
; implementation using a function
;

(def laplacian-kernel (u/matrix-to-mat
[ [0 -1 0]
  [-1 4 -1]
  [0 -1 0]]))

(defn std-laplacian [img]
  (let [ std (new-matofdouble)]
    (filter-2-d! img -1 laplacian-kernel)
    (mean-std-dev img (new-matofdouble) std)
    (Math/pow (first (.get std 0 0)) 2)))

(defn is-image-blurred?[img]
  (< (std-laplacian (clone img)) 100))

(defn mark-blurred! [ _img ]
  (let [ _text   (if (is-image-blurred? _img) "BLUR" "STILL") ]
    (put-text _img _text (new-point 30 30) FONT_ITALIC 1.0 color/blue-2 2)
  _img))
```

```
#'wealthy-darkness/mark-blurred!
```

```
(map #(-> % imread (u/resize-by 0.5) mark-blurred! u/mat-view)
     ["resources/images/tiger-blur.gif"
      "resources/blurred/blurred_cat.jpg"
      "resources/nico.jpg"])
```






STILL
PICK UP
HERE

Tennis Ball

This tutorial proposes options to find a tennis ball in a picture using two different techniques. This will also play with parameters, to show how to avoid finding false positive, i.e objects that are nowhere near the one we are looking for.

First, let's load the image that will be the base for our image finding exercice.

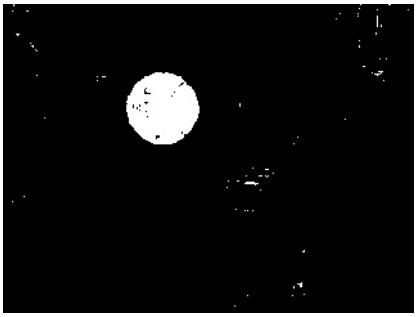
```
(ns divine-briars
  (:require
    [opencv3.utils :as u]
    [opencv3.colors.rgb :as color]
    [opencv3.core :refer :all]))
```

```
nil
```

```
(def img (-> "http://i.imgur.com/uoNRu60.jpg" (u/mat-from-url) (u/resize-by 0.5)))
(u/mat-view img)
```

Using Hough Circles

```
(def hsv (-> img clone (cvt-color! COLOR_RGB2HSV)))
(def thresh-image (new-mat))
(in-range hsv (new-scalar 50 100 0) (new-scalar 95 255 255) thresh-image)

; the ball is here
(u/mat-view thresh-image)
```

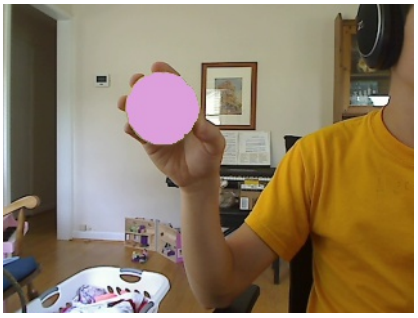


```
(let[ circles (new-mat) output (clone img) minRadius 25 maxRadius 40 ]

(hough-circles thresh-image circles CV_HOUGH_GRADIENT 1 minRadius 120 15 minRadius
maxRadius)

(dotimes [i (.cols circles)]
  (let [ circle (.get circles 0 i) x (nth circle 0) y (nth circle 1) r (nth circle 2)  p
(new-point x y)]
  (opencv3.core/circle output p (int r) color/plum 2)))

; the ball is detected
(u/mat-view output))
```

Method2: Using Find Contours

Find object with find contours gives less chance to parameters, the **find-contours** function from core does pretty much all for us.

```
(let[ hsv (-> img clone (cvt-color! COLOR_RGB2HSV))
      thresh-image (new-mat)
      contours (new-arraylist)
      output (clone img)]

(in-range hsv
          (new-scalar 50 100 0)
          (new-scalar 95 255 255)
          thresh-image)

(find-contours
  thresh-image
  contours
  (new-mat)  ; mask
  RETR_LIST
  CHAIN_APPROX_SIMPLE)

(dotimes [ci (.size contours)]
 (if (> (contour-area (.get contours ci)) 100 )
  (draw-contours output contours ci color/plum FILLED)))

(u/mat-view output))
```



Pencil Sketch

http://www.askaswiss.com/2016/01/how-to-create-pencil-sketch-opencv-python.html

Using OpenCV and Python, an RGB color image can be converted into a pencil sketch in four simple steps:

- Convert the RGB color image to grayscale.
- Invert the grayscale image to get a negative.
- Apply a Gaussian blur to the negative from step 2.
- Blend the grayscale image from step 1 with the blurred negative from step 3 using a color dodge.

Load the picture that will be sketched

```
(ns opencv3.cartoon2
  (:require
   [opencv3.core :refer :all]
   [opencv3.utils :as u]))

(def img
  (-> "https://cdn.theculturetrip.com/wp-content/uploads/2016/01/canals2.jpg"
   u/mat-from-url
   (u/resize-by 0.10)))
(u/mat-view img)
```

```
(def gray (-> img clone (cvt-color! COLOR_BGR2GRAY)))
(u/mat-view gray)

(def inverted
  (-> gray clone (bitwise-not!)))
(u/mat-view inverted)

(def gaussed
  (-> inverted clone (gaussian-blur! (new-size 21 21) 0.0 0.0)))
(u/mat-view gaussed)
```

```
(defn dodge-v2! [img_ mask]
  (let [ output (clone img_) ]
  (divide img_ (bitwise-not! (-> mask clone)) output 256.0)
  output))

(u/mat-view (dodge-v2! gray gaussed))
```

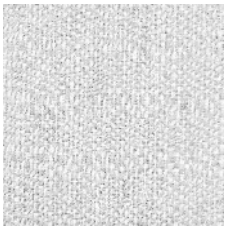
```
(defn burn-v2! [ image mask]
  (bitwise-not! (dodge-v2! image mask)))
(u/mat-view (burn-v2! gray gaussed))
```

Apply a Canvas effect

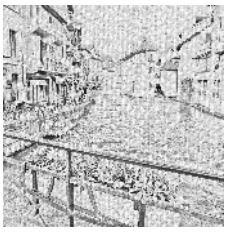
Now that the main picture has been turned to a crayon styled art form, it would be nice to lay this out on a canvas looking mat.

This is done using the **multiply** function from OpenCV core.

```
(def canvas (imread "resources/canvas.jpg" 0))
(resize! canvas (new-size (.cols gray) (.rows gray)))
(u/mat-view canvas)
```



```
(def output (new-mat))
(multiply (dodge-v2! gray gaussed) canvas output  (/ 1 256.0))
(u/mat-view output)
```

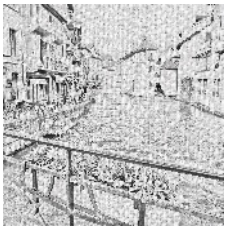


Let's make a function out of the above, so we can apply a few different canvas and see the output and effect of each of them.

```
(defn apply-canvas! [ sketch canvas]
  (let [ output (new-mat)]
    (resize! canvas (new-size (.cols sketch) (.rows sketch)))
    (multiply (-> sketch clone (cvt-color! COLOR_GRAY2RGB)) canvas output  (/ 1 256.0))
    output  ))
```

```
#'opencv3.cartoon2/apply-canvas!
```

```
(def sketch (dodge-v2! gray gaussed))
(def canvas (imread "resources/canvas.jpg"))
(u/mat-view (apply-canvas! sketch canvas))
```



```
(def canvas (imread "resources/canvas/oldcanvas.jpg"))
(u/mat-view (apply-canvas! sketch canvas))
```

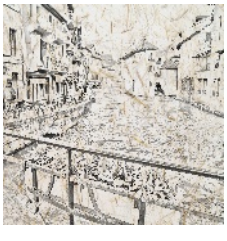
```
(def canvas (imread "resources/canvas/dottedcanvas.jpg"))
(u/mat-view (apply-canvas! sketch canvas))
```

```
(u/mat-view (apply-canvas! sketch  (imread "resources/canvas/oldcanvastexture.jpg")))
```

```
(u/mat-view (apply-canvas! sketch (imread "resources/canvas/vintage-old-brown-canvas-
texture.jpg")))
```

```
(u/mat-view (apply-canvas! sketch (imread "resources/canvas/grunge-parchment-background-
canvas.jpg")))
```

```
(u/mat-view (apply-canvas! sketch (imread "resources/canvas/japanese_paper.jpg")))
```



Landscape Art Adventures

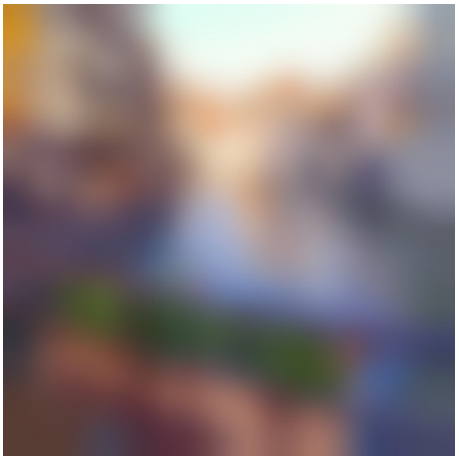
```
(ns opencv3.cartoon3
  (:require
   [opencv3.core :refer :all]
   [opencv3.utils :as u]))
```

```
nil
```

```
(def img
  (-> "https://cdn.theculturetrip.com/wp-content/uploads/2016/01/canals2.jpg"
  u/mat-from-url
  (u/resize-by 0.2)))
(u/mat-view img)
```

smoothing the picture using a bilateral filter

```
(def factor 4)
(def output (new-mat))
(def work (clone img))

(dotimes [_ factor] (pyr-down! work))
(bilateral-filter work output 9 9 7)
(dotimes [_ factor] (pyr-up! output))

(u/mat-view output)
```



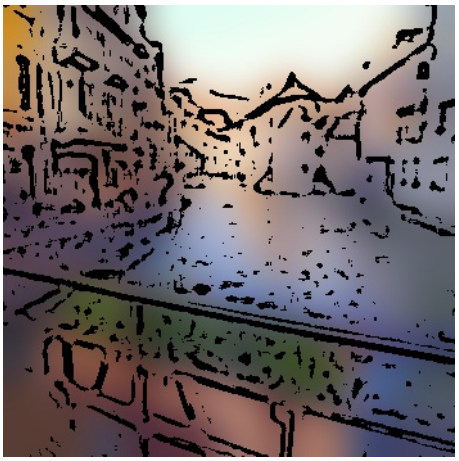
detect and enhance edges

- Reduce noise using a median filter
- Create an edge mask using adaptive thresholding

```
(def edge
  (-> img
    clone
    (resize! (new-size (.cols output) (.rows output)))
    (cvt-color! COLOR_RGB2GRAY)
    (median-blur! 7)
    (adaptive-threshold! 255 ADAPTIVE_THRESH_MEAN_C THRESH_BINARY 9 7)
    (cvt-color! COLOR_GRAY2RGB)))
(u/mat-view edge)
```

Combine color image with edge mask

```
(let [result (new-mat) ]
  (bitwise-and output edge result)
  (u/mat-view result))
```



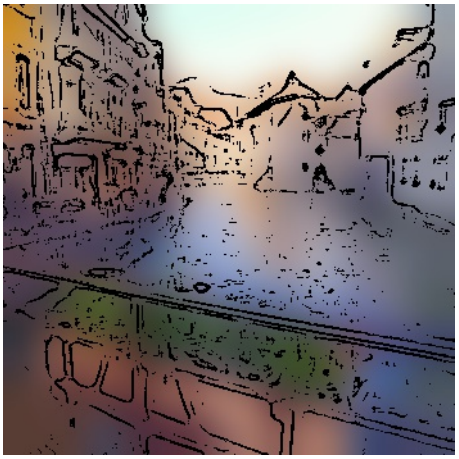
Playing with contours

Of course you can play with the contours size and numbers. The adaptive-threshold! does that with its last two parameters.

The **edges-thickmess** parameter controls how thick the contours will be, and the **edges-number** controls how many of them should be drawn.

```
(def edges-thickness 5)
(def edges-number 5)
(def edge
  (-> img
    clone
    (resize! (new-size (.cols output) (.rows output)))
    (cvt-color! COLOR_RGB2GRAY)
    (median-blur! 7)
    (adaptive-threshold! 255 ADAPTIVE_THRESH_MEAN_C THRESH_BINARY edges-thickness edges-
number)
    (cvt-color! COLOR_GRAY2RGB)))

(u/mat-view edge)
```

```
(let [result (new-mat) ]
  (bitwise-and output edge result)
  (u/mat-view result))
```



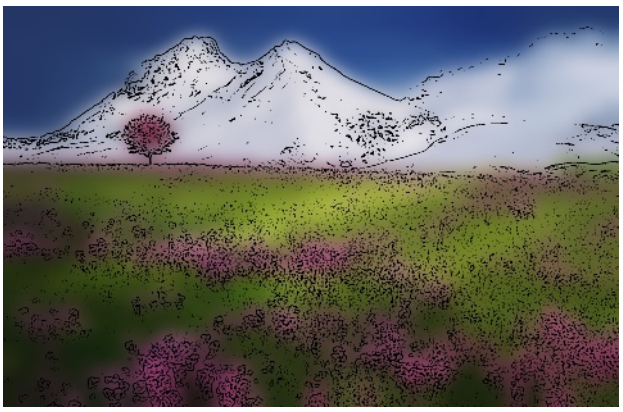
turning all this in functions

```
(defn smoothing!
  [img factor filter-size filter-value]
  (let [ work (clone img) output (new-mat)]
  (dotimes [_ factor] (pyr-down! work))
  (bilateral-filter work output filter-size filter-size filter-value)
  (dotimes [_ factor] (pyr-up! output))
  (resize! output (new-size (.cols img) (.rows img)))))

(defn edges!
  [img e1 e2 e3]
  (-> img
    clone
    (cvt-color! COLOR_RGB2GRAY)
    (median-blur! e1)
    (adaptive-threshold! 255 ADAPTIVE_THRESH_MEAN_C THRESH_BINARY e2 e3)
    (cvt-color! COLOR_GRAY2RGB)))

(defn cartoonize!
  [ img s1 s2 s3 e1 e2 e3]
    (let [ output (smoothing! img s1 s2 s3) edge (edges! img e1 e2 e3)]
    (bitwise-and output edge output)
    output))
```

```
#'opencv3.cartoon3/cartoonize!
```

```
(->
  "resources/landscape/landscape-nature-sky-blue.jpg"
  imread
  (u/resize-by 0.2)
  (cartoonize! 6 9 7  7 9 11)
  (u/resize-by 0.5)
  (u/mat-view ))
```

```
(->
  "resources/landscape/amazing-beautiful-beauty-blue.jpg"
  imread
  (cartoonize! 5 9 7  7 7 5)
  (u/resize-by 0.25)
  (u/mat-view))
```



```
(->
  "resources/landscape/amazing-beautiful-beauty-blue.jpg"
  imread
  (u/resize-by 0.25)
  (cartoonize! 5 9 7  7 7 5)
  (u/mat-view))
```