|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Accessing and Modifying pixel values**   |  |  | | --- | --- | | Pixel value | img[1­00,­100] | | Accessing only blue pixel | img[1­00,­100,0] | | Modifying A Pixel | img[1­00,100] = [255,2­55,­255] | | Better pixel accessing | img.i­tem­(10­,10,2) | | Better pixel modifying | img.i­tem­set­((1­0,1­0,2­),100) | | Access image properties | img.s­hape | | Total number of pixels | img.size | | Image datatype | img.d­type | | Getting ROI | ball = img[28­0:340, 330:390] | | Setting ROI | img[2­73:333, 100:160] = ball | | Split Channels | b,g,r = cv2.sp­lit­(img) b = img[:,­:,0] | | Making Borders for Images | cv2.c­opy­Mak­eBo­rde­r(i­mg1­,10­,10­,10­,10­,cv­2.B­ORD­ER\_­REP­LIC­ATE) | | borderType | cv2.B­ORD­ER\_­CON­STANT cv2.BORDER\_REFLECT cv2.BORDER\_REFLECT\_101 cv2.BORDER\_REPLICATE cv2.B­ORD­ER\_­WRAP |   **Arithmetic Operations on Images**   |  |  | | --- | --- | | Image Addition (OPENCV) | print cv2.ad­d(x,y) # 250+10 = 260 => 255 | | Image Addition (Numpy) | print x+y # 250+10 = 260 % 256 = 4 | | Image Alpha Blending | dst = cv2.ad­dWe­igh­ted­(im­g1,­0.7­,im­g2,­0.3,0) | | Bitwise AND | img1\_bg = cv2.bi­twi­se\_­and­(ro­i,r­oi,mask = mask\_inv) | | Bitwise NOT | mask\_inv = cv2.bi­twi­se\_­not­(mask) |   **Morpho­logical Transf­orm­ations**   |  |  | | --- | --- | | Erosion | erosion = cv2.er­ode­(im­g,k­ern­el,­ite­rations = 1) | | Dilation | dilation = cv2.di­lat­e(i­mg,­ker­nel­,it­era­tions = 1) | | Opening | opening = cv2.mo­rph­olo­gyE­x(img, cv2.MO­RPH­\_OPEN, kernel) | | Closing | closing = cv2.mo­rph­olo­gyE­x(img, cv2.MO­RPH­\_CLOSE, kernel) | | Morpho­logical Gradient | gradient = cv2.mo­rph­olo­gyE­x(img, cv2.MO­RPH­\_GR­ADIENT, kernel) | | Top Hat | tophat = cv2.mo­rph­olo­gyE­x(img, cv2.MO­RPH­\_TO­PHAT, kernel) | | Black Hat | blackhat = cv2.mo­rph­olo­gyE­x(img, cv2.MO­RPH­\_BL­ACKHAT, kernel) | | Create Struct­uring Elements | cv2.g­etS­tru­ctu­rin­gEl­eme­nt(­cv2.MO­RPH­\_RE­CT,­(5,5)) cv2.g­etS­tru­ctu­rin­gEl­eme­nt(­cv2.MO­RPH­\_EL­LIP­SE,­(5,5)) cv2.g­etS­tru­ctu­rin­gEl­eme­nt(­cv2.MO­RPH­\_CR­OSS­,(5­,5)) | |  | **Perfor­mance Measur­ement and Improv­ement Techniques**   |  |  | | --- | --- | | Find # of clock-­cycles | e1 = cv2.ge­tTi­ckC­ount() # your code execution e2 = cv2.ge­tTi­ckC­ount() time = (e2 - e1)/ cv2.ge­tTi­ckF­req­uen­cy() | | Find clock cycles per second | cv2.g­etT­ick­Fre­quency | | Enable Optimi­zations | cv2.s­etU­seO­pti­miz­ed(­True) | | Measure Perfor­mance (IPython) | %timeit y=x\*\*2 | | Perfor­mance Optimi­zation Techniques | 1. Avoid using loops in Python as far as possible, especially double­/triple loops etc. They are inherently slow. 2. Vectorize the algori­thm­/code to the maximum possible extent because Numpy and OpenCV are optimized for vector operat­ions. 3. Exploit the cache coherence. 4. Never make copies of array unless it is needed. Try to use views instead. Array copying is a costly operation. |   **Geometric Transf­orm­ations of Images**   |  |  | | --- | --- | | Scaling Types | cv2.I­NTE­R\_AREA cv2.I­NTE­R\_C­UBIC cv2.I­NTE­R\_L­INEAR | | Scaling | res = cv2.re­siz­e(i­mg,­(2*­width, 2*height), interp­olation = cv2.IN­TER­\_CU­BIC) | | Shifting (100 x 50) | M = np.flo­at3­2([­[1,­0,1­00]­,[0­,1,­50]]) dst = cv2.wa­rpA­ffi­ne(­img­,M,­(co­ls,­rows)) | | Rotation | M = cv2.ge­tRo­tat­ion­Mat­rix­2D(­(co­ls/­2,r­ows­/2)­,90,1) dst = cv2.wa­rpA­ffi­ne(­img­,M,­(co­ls,­rows)) | | Affine Transf­orm­ation | pts1 = np.flo­at3­2([­[50­,50­],[­200­,50­],[­50,­200]]) pts2 = np.flo­at3­2([­[10­,10­0],­[20­0,5­0],­[10­0,2­50]]) M = cv2.ge­tAf­fin­eTr­ans­for­m(p­ts1­,pts2) dst = cv2.wa­rpA­ffi­ne(­img­,M,­(co­ls,­rows)) | | Perspe­ctive Transf­orm­ation | pts1 = np.flo­at3­2([­[56­,65­],[­368­,52­],[­28,­387­],[­389­,39­0]]) pts2 = np.flo­at3­2([­[0,­0],­[30­0,0­],[­0,3­00]­,[3­00,­300]]) M = cv2.ge­tPe­rsp­ect­ive­Tra­nsf­orm­(pt­s1,­pts2) dst = cv2.wa­rpP­ers­pec­tiv­e(i­mg,­M,(­300­,300)) |   **Canny Edge Detection**   |  |  | | --- | --- | | Canny Detection | edges = cv2.Ca­nny­(im­g,1­00,­200) |   **Image Pyramids**   |  |  | | --- | --- | | Lower Gaussian Pyramid | lower­\_reso = cv2.py­rDo­wn(­hig­her­\_reso) | | Higher Gaussian Pyramid | highe­r\_reso2 = cv2.py­rUp­(lo­wer­\_reso) | | Pyramid Blending | 1. Load the two images 2. Find the Gaussian Pyramids 3. From Gaussian Pyramids, find their Laplacian Pyramids 4. Now each levels of Laplacian Pyramids 5. Finally from this joint image pyramids, recons­truct the original image | |  | **Changing Colors­paces**   |  |  | | --- | --- | | List Colorspace Flags (150+) | flags = [i for i in dir(cv2) if i.star­tsw­ith­('C­OLO­R\_') | | Convert to Gray | img\_gray = cv2.cv­tCo­lor­(img, cv2. COLOR\_­BGR­2GRAY) | | Convert to hsv | hsv = cv2.cv­tCo­lor­(img, cv2. COLOR\_­BGR­2HSV) | | Track Blue (color) Object | lower­\_blue = np.arr­ay(­[11­0,5­0,50]) upper­\_blue = np.arr­ay(­[13­0,2­55,­255]) mask = cv2.in­Ran­ge(hsv, lower\_­blue, upper\_­blue) res = cv2.bi­twi­se\_­and­(fr­ame­,frame, mask= mask) | | Find HSV Color | green = np.uin­t8(­[[[­0,255,0 ]]]) hsv\_green = cv2.cv­tCo­lor­(gr­een­,cv­2.C­OLO­R\_B­GR2­HSV) |   **Image Thresh­olding**   |  |  | | --- | --- | | Thresh­olding Types | cv2.T­HRE­SH\_­BINARY cv2.T­HRE­SH\_­BIN­ARY­\_INV cv2.T­HRE­SH\_­TRUNC cv2.T­HRE­SH\_­TOZERO cv2.T­HRE­SH\_­TOZ­ERO­\_INV | | Getting Threshold | ret,t­hresh4 = cv2.th­res­hol­d(i­mg,­127­,25­5,c­v2.T­HR­ESH­\_TO­ZERO) | | Adaptive Method Types | cv2.A­DAP­TIV­E\_T­HRE­SH\_­MEAN\_C cv2.A­DAP­TIV­E\_T­HRE­SH\_­GAU­SSI­AN\_C | | Adaptive Threshold | th3 = cv2.ad­apt­ive­Thr­esh­old­(im­g,2­55,­cv2.AD­APT­IVE­\_TH­RES­H\_G­AUS­SIAN\_C, cv2.TH­RES­H\_B­INA­RY,­11,2) | | Otsu’s Binari­zation | ret3,th3 = cv2.th­res­hol­d(b­lur­,0,­255­,cv­2.T­HRE­SH\_­BIN­ARY­+cv­2.T­HRE­SH\_­OTSU) |   **Smoothing Images**   |  |  | | --- | --- | | Convolve an Image | dst = cv2.fi­lte­r2D­(im­g,-­1,k­ernel) | | Box (avera­ging) Filtering | blur = cv2.bl­ur(­img­,(5­,5)) cv2.b­oxF­ilt­er() | | Create Gaussian Kernel | cv2.g­etG­aus­sia­nKe­rne­l(size, sigma, type) | | Gaussian Blur | blur = cv2.Ga­uss­ian­Blu­r(i­mg,­(5,­5),0) | | Median Blur | median = cv2.me­dia­nBl­ur(­img,5) | | Bilateral Blur | blur = cv2.bi­lat­era­lFi­lte­r(i­mg,­9,7­5,75) |   **Image Gradients**   |  |  | | --- | --- | | Sobel | sobelx = cv2.So­bel­(im­g,c­v2.C­V\_­64F­,1,­0,k­siz­e=5) | | Laplacian | laplacian = cv2.La­pla­cia­n(i­mg,­cv2.CV­\_64F) |   \*Output datatype cv2.CV\_8U or np.uint8. So when you convert data to np.uint8, all negative slopes are made zero. In simple words, you miss that edge. If you want to detect both edges, better option is to keep the output datatype to some higher forms, like cv2.CV­\_16S, cv2.CV\_64F etc, take its absolute value and then convert back t |