Hand gesture controlled Chrome Dino Game

Sarvesh Chandra(Team Vision) IITK Email: csarvesh@iitk.ac.in

Introduction

- **Principle**

  The Chrome Dino game can be controlled by the hand using skin detection and contours(OpenCV) through a Python Code.

- **Python Libraries required**

  cv2, numpy and pyautogui

- **Overview of Code**

  The python code captures the video on the webcam and using colour detection on the frames, we detect skin. Contours are used to find the centroid of the palm. Based on the movement of the centroid above or below a certain horizontal region marked on the webcam stream, the dino jumps to dodge obstacles.

Detailed explanation of code

- **First part: importing modules and their use, description of initial variables**
  - **import numpy as np**
  - **import cv2** module containing image processing functions
  - **import pyautogui** module for linking keypad to the code
  - **ob=cv2.VideoCapture(0)** creating an object for storing the reading of camera device
  - **start=0** variable denoting the start of game
  - **flag=0** variable denoting state of centroid-whether inside or outside region between the two lines

- **Second part: extracting frames from the video stream, checking if camera is open at all times**
  - **while True:**
  - ␣ **if not ob.isOpened():** check if ob has been initialised
  - ␣␣ **ob.open()** if not, initialize ob with the reading of camera device
  - ␣ **_, im=ob.read()** extract frames from ob into im

- **Third part: Skin detection**
  - ␣ **lower=np.array([0,48,80], dtype="uint8")** setting lower hsv limit for skin pixel
  - ␣ **upper=np.array([20,255,255], dtype="uint8")** setting lower upper limit for skin pixel
  - ␣ **img=im**
  - ␣ **converted=cv2.cvtColor(img, cv2.COLOR_BGR2HSV)** image converted to HSV from BGR stored in "converted"
  - ␣ **skinmask=cv2.inRange(converted, lower, upper)** find all pixels which contain skin replace the rest by [0,0,0]
  - ␣ **kernel=cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (11,11))** define elliptical kernel of size 11*11
  - ␣ **skinmask=cv2.erode(skinmask, kernel, iterations=1)** reduce small and faulty skin detected areas using the kernel defined, done once
  - ␣ **skinmask=cv2.dilate(skinmask, kernel, iterations=2)** expand detected skin areas using the kernel defined, done twice
  - ␣ **skinmask=cv2.GaussianBlur(skinmask, (3,3), 0)** Blur the image before thresholding using a 3*3 kernel to reduce intensity of small, faulty skin detected areas(noise) and bring uniformity to true skin detected areas

- ␣ **im=cv2.bitwise_and(img, img, mask=skinmask)** keep the respective areas of original image at matching areas of the original image and ’skinmask’ in im, rest of im is black

- **Fourth part:Finding contours**

  - ␣ **imgray=cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)** convert to grayscale before finding contours
  - ␣ **ret,thresh=cv2.threshold(imgray,50,255,0)** set all pixels with values bw 50 and 255 to 255 and rest to 0 to help find contours
  - ␣ **contours, hierarchy=cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)** storing contours in ”contours”

- **Fifth part:Finding the largest contour(the hand boundaries)**

  - ␣ **max_i=0** setting index of the largest contour to 0
  - ␣ **max_area=0** setting area of the largest contour to 0
  - ␣ **for i in range(len(contours)):** loop over all indices of ’contours’
  - ␣␣␣␣ **c=contours[i]** selecting ith contour
  - ␣␣␣␣ **area_cnt=cv2.contourArea(c)** calculate area enclosed by contour
  - ␣␣␣␣ **if area_cnt >max_area:** if the area is greater than current maximum
  - ␣␣␣␣ ␣ **max_area=area_cnt** reset max_area to this area
  - ␣␣␣␣ ␣ **max_i=i** reset max_i to this i

- **Sixth part:Finding the centroid of the hand**

  - ␣ **if max_i <len(contours):** if the index obtained is not out of bounds
  - ␣␣␣␣ **max_c=contours[max_i]** Store the set of points on that contour(roughly the boundary of hand) in max_c
  - ␣␣␣␣ **moment=cv2.moments(max_c)** store the set of moments of those points in ”moment”
  - ␣␣␣␣ **if moment[’m00’] != 0:**
  - ␣␣␣␣ ␣ **gx=int(moment[’m10’]/moment[’m00’])** x coordinate of centroid
  - ␣␣␣␣ ␣ **gy=int(moment[’m01’]/moment[’m00’])** y coordinate of centroid
  - ␣␣␣␣ **g=gx,gy** the centroid as a coordinate pair

- **Seventh part:Representing the results on the video stream**

  - ␣␣␣␣ **cv2.drawContours(im, [max_c], 0, (0,255,0), 3)** Draw the boundary of hand
  - ␣␣␣␣ **cv2.circle(im, g, 5,(0,0,255), -1)** Draw the centroid
  - ␣␣␣␣ **cv2.line(im, (0, int(im.shape[0]/2)+40), (im.shape[1], int(im.shape[0]/2)+40), (255,0,0))** mark the lower boundary of the region of interest
  - ␣␣␣␣ **cv2.line(im, (0, int(im.shape[0]/2)-40), (im.shape[1], int(im.shape[0]/2)-40), (255,0,0))** mark the upper boundary of the region of interest
  - ␣␣␣␣ **cv2.imshow(”Dino”, im)** Display the frame

- **Seventh part:Configuring the link to keypad**

  - ␣ **if gy <int(im.shape[0]/2)+40 and gy >int(im.shape[0]/2)-40 :** if the centroid is in the region of interest
  - ␣␣␣␣ **if start==0 :** if game has not started
  - ␣␣␣␣ ␣ **pyautogui.press(”space”)** press space to start
  - ␣␣␣␣ ␣ **start=1** reset start to 1
  - ␣␣␣␣ **else:** if game has started
  - ␣␣␣␣ ␣ **flag=0** set flag to 0(inside the region)
  - ␣ **elif flag==0:** if the centroid has just come out of that region

- ␣␣␣␣ **pyautogui.press("space")** press space to jump
- ␣␣␣␣ **flag=1** reset flag to 1(outside the region)

- **Eighth part:End of Game**

  - ␣ **if cv2.waitKey(1) &0xFF==ord('x'):** if a key is pressed and it is 'x'
  - ␣␣␣␣ **break** stop video stream
  - **cv2.waitKey(0)** if a key is pressed
  - **ob.release()** shut down the camera device
  - **cv2.destroyAllWindows()** close the window streaming the video

Detailed explanation of OpenCV functions and their parameters

- **cv2.VideoCapture()** returns **VideoCapture** object
  parameters: ***int*** *device* ***OR const string&*** *filename*
  - **device** id of the opened video capturing device (i.e. a camera index). If there is a single camera connected, just pass 0.
  - **filename** name of the opened video file (eg. video.avi) or image sequence (eg. img_%02d.jpg, which will read samples like img_00.jpg, img_01.jpg, img_02.jpg, ...)
  - **VideoCapture object** contains readings of the camera device/video file

- **cv2.VideoCapture().isOpened()** returns **bool** variable
  parameters: ***None***
  - **Returns** true if video capturing has been initialized already.If the previous call to VideoCapture constructor or VideoCapture::open succeeded, the method returns true.

- **cv2.VideoCapture().open()** returns **None**
  parameters: ***int*** *device* ***OR const string&*** *filename*
  - **device** id of the opened video capturing device (i.e. a camera index). If there is a single camera connected, just pass 0.
  - **filename** name of the opened video file (eg. video.avi) or image sequence (eg. img_%02d.jpg, which will read samples like img_00.jpg, img_01.jpg, img_02.jpg, ...)
  - **Initializes** the VideoCapture object with a value either from the given filename or the device

- **cv2.VideoCapture().read()** returns a **tuple** (**bool** retvalue, **mat** image)
  parameters: ***None***
  - **retvalue** True if reading was successful
  - **image** instantaneous frame of the video stream

- **cv2.VideoCapture().release()** returns **None**
  parameters: ***None***
  - **Closes** video file or capturing device. The methods are automatically called by subsequent cv2.VideoCapture.open() and by VideoCapture destructor.

- **cv2.cvtcolor()** returns **mat** dst
  parameters: ***mat*** *src,* ***int*** *code[,* ***mat*** *dst[,* ***int*** *dstCn]]*
  - **src** input image: 8-bit unsigned, 16-bit unsigned (CV_16UC...), or single-precision floating-point.
  - **int** color space conversion code
  - **dst** output image of the same size and depth as src.
  - **dstCn** number of channels in the destination image; if the parameter is 0, the number of the channels is derived automatically from src and code .

- **cv2.COLOR_BGR2HSV** In case of 8-bit and 16-bit images, R, G, and B are converted to the floating-point format and scaled to fit the 0 to 1 range.
  * $V = max(R, G, B)$
  * $S \leftarrow \begin{cases} \frac{V-min(R,G,B)}{V} & \text{if V} \neq 0 \\ 0 & otherwise \end{cases}$
  * $H \leftarrow \begin{cases} \frac{60(G-B)}{(V-min(R,G,B))} & \text{if V=R} \\ \frac{120+60(B-R)}{(V-min(R,G,B))} & \text{if V=G} \\ \frac{240+60(R-G)}{(V-min(R,G,B))} & \text{if V=B} \end{cases}$
- **cv2.COLOR_BGR2GRAY** Transformations within RGB space like adding/removing the alpha channel, reversing the channel order, conversion to/from 16-bit RGB color (R5:G6:B5 or R5:G5:B5), as well as conversion to/from grayscale using:
  * RGB[A] to Gray: $\mathbf{Y} \leftarrow 0.299 \times \mathbf{R} + 0.587 \times \mathbf{G} + 0.114 \times \mathbf{B}$
  * Gray to RGB[A]: $\mathbf{R} \leftarrow \mathbf{Y}, \mathbf{G} \leftarrow \mathbf{Y}, \mathbf{B} \leftarrow \mathbf{Y}$

• cv2.inRange() — returns **mat** dst

*parameters: **mat** src, **array** lowerb, **array** upperb[, **mat** dst]*

- **src** first input array.
- **lowerb** inclusive lower boundary array or a scalar.
- **upperb** inclusive upper boundary array or a scalar.
- **dst** output array of the same size as src and CV_8U type. Obtained as a matrix containing only those pixels whose values lie between lowerb and upperb

• cv2.getStructuringElement() — returns **mat** kernel

*parameters:**int** shape,**int tuple** ksize*

- **shape** ellipse, rectangle, cross
- **cv2.MORPH_ELLIPSE** shape set to ellipse
- **ksize** specifies size of kernel
- **kernel** structuring element for morphology

• cv2.erode() — **dst**

*parameters: **mat** src, **mat** kernel[, **mat** dst[, **int tuple** anchor[,**int** iterations[, **int** borderType[, **int** borderValue]]]]]*

- **src, kernel** same as in cv2.getStructuringElement()
- **anchor** point on the kernel which moves over all the pixels of src iteratively
- **iterations** no. of times erosion happens
- **borderType** pixel extrapolation method
- **borderValue** border value in case of a constant border
- **dst** The function erodes the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the minimum is taken:
  $\texttt{dst}(x, y) = \min_{(x',y'):\,\texttt{element}(x',y')\neq 0} \texttt{src}(x + x', y + y')$

• cv2.dilate() — **dst**

*parameters: **mat** src, **mat** kernel[, **mat** dst[, **int tuple** anchor[,**int** iterations[, **int** borderType[, **int** borderValue]]]]]*

- **src, kernel** same as in cv2.getStructuringElement()
- **anchor** point on the kernel which moves over all the pixels of src iteratively
- **iterations** no. of times dilation happens
- **borderType** pixel extrapolation method
- **borderValue** border value in case of a constant border
- **dst** The function dilates the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the maximum is taken:
  $\texttt{dst}(x, y) = \max_{(x',y'):\,\texttt{element}(x',y')\neq 0} \texttt{src}(x + x', y + y')$

- **cv2.GaussianBlur()**
  *parameters: **mat** src,**int tuple** ksize, **int** sigmaX[, **mat** dst[, **int** sigmaY[, **int** borderType]]]*
  - **src** source image
  - **ksize** size of the rectangular kernel
  - **sigmaX** standard deviation in x-direction
  - **sigmaY** standard deviation in y-direction
  - **borderType** same as in cv2.erode()
  - **create kernel and set its coefficients** the ksize 1 matrix of Gaussian filter coefficients:
    $G_i = \alpha * e^{-(i-(\mathtt{ksize}-1)/2)^2/(2*\mathtt{sigma}^2)}$,
    where $i = 0..\mathtt{ksize} - 1$ and α is the scale factor chosen so that $\sum_i G_i = 1$.
  - **dst** The returned filtered image matrix. The function convolves the source image with the specified Gaussian kernel.

- **cv2.bitwise_and()** returns **mat** dst
  *parameters:**mat** src1,**mat** src2[,**mat** dst[,**mat** mask]]*
  - **src1** first input array or a scalar.
  - **src2** second input array or a scalar.
  - **mask** optional operation mask, 8-bit single channel array, that specifies elements of the output array to be changed.
  - **dst** the per-element bit-wise conjunction of two arrays or an array and a scalar, calculated as follows:
    * Two arrays when src1 and src2 have the same size:
      $\mathtt{dst}(I) = \mathtt{src1}(I) \wedge \mathtt{src2}(I) \quad \mathtt{if}\ \mathtt{mask}(I) \neq 0$
    * An array and a scalar when src2 is constructed from Scalar or has the same number of elements as src1.channels():
      $\mathtt{dst}(I) = \mathtt{src1}(I) \wedge \mathtt{src2} \quad \mathtt{if}\ \mathtt{mask}(I) \neq 0$
    * A scalar and an array when src1 is constructed from Scalar or has the same number of elements as src2.channels():
      $\mathtt{dst}(I) = \mathtt{src1} \wedge \mathtt{src2}(I) \quad \mathtt{if}\ \mathtt{mask}(I) \neq 0$

- **cv2.moments()** returns **dictionary** moments
  *parameter: **list** array[,**bool** binaryImage]*
  - **array** Raster image (single-channel, 8-bit or floating-point 2D array) or an array ($1 \times N or N \times 1$) of 2D points (Point or Point2f).
  - **binaryImage** If it is true, all non-zero image pixels are treated as 1's. The parameter is used for images only.
  - **moments** Output moments, computed as follows:
    * In case of a raster image, the spatial moments $\mathtt{Moments::m}_{ji}$ are computed as:
      $\mathtt{m}_{ji} = \sum_{x,y} \left(\mathtt{array}(x,y) \cdot x^j \cdot y^i\right)$
    * The central moments $\mathtt{Moments::mu}_{ji}$ are computed as:
      $\mathtt{mu}_{ji} = \sum_{x,y} \left(\mathtt{array}(x,y) \cdot (x-\bar{x})^j \cdot (y-\bar{y})^i\right)$
      where $(\bar{x}, \bar{y})$ is the mass center:
      $\bar{x} = \frac{\mathtt{m}_{10}}{\mathtt{m}_{00}},\ \bar{y} = \frac{\mathtt{m}_{01}}{\mathtt{m}_{00}}$
    * The normalized central moments $\mathtt{Moments::nu}_{ij}$ are computed as:
      $\mathtt{nu}_{ji} = \frac{\mathtt{mu}_{ji}}{\mathtt{m}_{00}^{(i+j)/2+1}}$
    * **Note:** The moments of a contour are defined in the same way but computed using the Green's formula. So, due to a limited raster resolution, the moments computed for a contour are slightly different from the moments computed for the same rasterized contour.

- **cv2.threshold()** returns **mat** dst
  *parameters: **mat** src,**int** thresh,**int** maxval,**int** type[,**mat** dst]*
  - **thresh** threshold value
  - **maxval** upper limit for transformation
  - **type** Five types of which THRESH_BINARY has been used here

- **cv2.THRESH_BINARY**
  $\texttt{dst}(x,y) = \begin{cases} \texttt{maxval} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$

- **cv2.findContours()** returns **tuple** contours, hierarchy
  parameters: ***mat*** *image,* ***int*** *mode,* ***int*** *method (Optional parameters:) [,* ***mat*** *contours[,* ***mat*** *hierarchy[,* ***int tuple*** *offset]]]*
  - **contours** Detected contours. Each contour is stored as a vector of points. The function retrieves contours from the binary image using the algorithm [Suzuki85].
  - **hierarchy** Optional output vector, containing information about the image topology. It has as many elements as the number of contours. For each i-th contour contours[i] , the elements hierarchy[i][0] , hiearchy[i][1] , hiearchy[i][2] , and hiearchy[i][3] are set to 0-based indices in contours of the next and previous contours at the same hierarchical level, the first child contour and the parent contour, respectively. If for the contour i there are no next, previous, parent, or nested contours, the corresponding elements of hierarchy[i] will be negative.
  - **mode** Contour retrieval modes:
    * **cv2.RETR_EXTERNAL** retrieves only the extreme outer contours. It sets hierarchy[i][2]=hierarchy[i][3]=-1 for all the contours.
    * **cv2.RETR_LIST** retrieves all of the contours without establishing any hierarchical relationships
    * **cv2.RETR_CCOMP** retrieves all of the contours and organizes them into a two-level hierarchy. At the top level, there are external boundaries of the components. At the second level, there are boundaries of the holes. If there is another contour inside a hole of a connected component, it is still put at the top level.
    * **cv2.RETR_TREE** retrieves all of the contours and reconstructs a full hierarchy of nested contours. This full hierarchy is built and shown in the OpenCV contours.c demo.
  - **method** Contour Approximation methods:
    * **cv2.CHAIN_APPROX_NONE** stores absolutely all the contour points. That is, any 2 subsequent points (x1,y1) and (x2,y2) of the contour will be either horizontal, vertical or diagonal neighbors, that is, max(abs(x1-x2),abs(y2-y1))==1.
    * **cv2.CHAIN_APPROX_SIMPLE** compresses horizontal, vertical, and diagonal segments and leaves only their end points. For example, an up-right rectangular contour is encoded with 4 points.
    * **cv2.CHAIN_APPROX_TC89_L1** applies one of the flavors of the Teh-Chin chain approximation algorithm. See [TehChin89] for details.
    * **cv2.CHAIN_APPROX_TC89_KCOS** applies one of the flavors of the Teh-Chin chain approximation algorithm. See [TehChin89] for details.
  - **offset** Optional offset by which every contour point is shifted. This is useful if the contours are extracted from the image ROI and then they should be analyzed in the whole image context.

- **cv2.drawContours()** returns **None**
  parameters: ***mat*** *image,* ***mat*** *contours,* ***int*** *contourIdx,* ***int tuple*** *color*
  Optional:[, **int** thickness[,**int** lineType[,**mat** hierarchy[,**int** maxLevel[, **int tuple** offset]]]]]
  - **contourIdx** Parameter indicating a contour to draw. If it is negative, all the contours are drawn.
  - **color, thikness, lineType** self explanatory
  - **hierarchy** Optional information about hierarchy. It is only needed if you want to draw only some of the contours
  - **maxLevel** Maximal level for drawn contours. If it is 0, only the specified contour is drawn. If it is 1, the function draws the contour(s) and all the nested contours. If it is 2, the function draws the contours, all the nested contours, all the nested-to-nested contours, and so on. This parameter is only taken into account when there is hierarchy available.
  - **offset** Optional contour shift parameter. Shift all the drawn contours by the specified $\texttt{offset} = (dx, dy)$.

- **cv2.line()** returns **None**
  parameters: ***mat*** *img,* ***int tuple*** *pt1,* ***int tuple*** *pt2 (Optional parameters:)* ***int*** *color[,* ***int*** *thickness[,* ***int*** *lineType[,* ***int*** *shift]]]*
  - **pt1** First point of the line segment.
  - **pt2** second point of the line segment.
  - **color, thickness, lineType** self-explanatory
  - **shift** Number of fractional bits in the point coordinates.

- **cv2.circle()** returns **None**
  *parameters:**mat** img,**int tuple** center,**int** radius, (Optional parameters:) **int** color[,**int** thickness[,**int** lineType[,**int** shift]]]*
  - **center** Center of the circle.
  - **radius** Radius of the circle.
  - **color, thickness, lineType, shift** same as in cv2.line()

- **cv2.imshow()** returns **None**
  *parameters:**const string&** winname,**mat** image*
  - **winname** Name of the window.
  - **image** Image to be shown.

- **cv2.waitKey()** returns **None**
  *parameters: **int** delay*
  - **delay** Delay in milliseconds. 0 is the special value that means "forever". The function waitKey waits for a key event infinitely ($when$`delay` ≤ 0) or for delay milliseconds, when it is positive. Since the OS has a minimum time between switching threads, the function will not wait exactly delay ms, it will wait at least delay ms, depending on what else is running on your computer at that time. It returns the code of the pressed key or -1 if no key was pressed before the specified time had elapsed.

Few Observations

- The number of iterations of erosion should be 1 as there are significantly large areas containing undetected skin
- The number of iterations of dilation should be at least 2 due to same reason
- Kernel size for gaussian smoothing is 3*3 as noises-falsely detected skin are quite small