**Accessing camera :**

1. Importing the necessary modules:

* cv2 module from OpenCV is imported.

1. Opening the video capture:

* cap = cv2.VideoCapture(0) initializes the video capture object cap to capture video from the default camera (index 0). If you have multiple cameras connected, you can specify a different index accordingly.

1. Starting an infinite loop:

* while True: starts a continuous loop that will capture frames from the video feed until a break condition is met.

1. Reading frames from the video capture:

* r, f = cap.read() captures the next frame from the video capture object cap and assigns it to f. The return value r indicates whether the frame was successfully read.

1. Displaying the frame:

* cv2.imshow("Photo", f) displays the frame f in a window named "Photo". imshow() is a function from OpenCV that is used to display images.

1. Checking for the break condition:

* if cv2.waitKey(1) == ord('q'): checks if the key pressed is 'q'. waitKey() is a function that waits for a key event for the specified number of milliseconds (1 millisecond in this case). If the pressed key is 'q', the loop will break and the program will exit.

1. Releasing the video capture and closing windows:

* cap.release() releases the video capture object, freeing up any resources it was using.

**What is r doing hear in 4th line ?**

In this line of code: **r, frame = cap.read()**

The function cap.read() reads the frame from the video capture object cap and returns two values: r and frame.

* r is a boolean value that indicates whether the frame was successfully read. It will be True if the frame was read successfully and False if there was an issue or if the video capture has reached its end.
* frame is the actual frame read from the video capture. It is an image represented as a NumPy array.

**Face detection :**

1. Importing the necessary modules:

- `**cv2**` module from OpenCV is imported.

- `**os**` module is imported to handle file and directory operations.

2. Loading the face cascade classifier:

- `**h = cv2.CascadeClassifier('haarcascade\_frontalface\_default.xml')**` loads the Haar cascade classifier for face detection. This classifier is used to detect human faces in the subsequent frames.

3. Opening the camera:

- `**cam = cv2.VideoCapture(0)**` initializes the video capture object `cam` to capture video from the default camera (index 0).

4. Setting up video formatting:

- `**fou = cv2.VideoWriter\_fourcc(\*"XVID")**` sets the video format to XVID. This format code will be used when creating the video file.

5. Checking camera access:

- `**if not cam.isOpened():**` checks if the camera is not opened. If the camera access is not available, it prints a message indicating that the camera has no access.

6. Starting the main loop:

- `**while True:**` starts a continuous loop that will capture frames from the camera until a break condition is met.

* The while True loop is used in this code to **continuously process** **frames from the video capture and perform face detection. It allows the program to keep running until the user decides to exit.**
* The loop starts with while True, which means it will continue indefinitely until a break condition is met. In this case, the break condition is checking for the key 'q' to be pressed. When the user presses 'q', the loop will be terminated, and the program will continue to release the video capture and close all windows.
* Using while True is a common approach in real-time video processing tasks where you want to continuously process frames until a specific event occurs or a specific condition is met.

7. Reading frames from the camera:

- `**r, f = cam.read()**` captures the next frame from the camera and assigns it to `f`. The return value `r` indicates whether the frame was successfully read.

8. Converting frame to grayscale:

- `**g = cv2.cvtColor(f, cv2.COLOR\_BGR2GRAY)**` converts the frame `f` from BGR (color) to grayscale. This conversion is necessary for face detection.

9. Detecting human faces:

- `**hum = h.detectMultiScale(g, 1.3, 5)**` detects human faces in the grayscale frame `g` using the loaded face cascade classifier `h`. The resulting `hum` variable contains the coordinates and size of the detected faces.

* The detectMultiScale function is used to detect objects, in this case, human faces, in an image. It uses a machine learning algorithm called the Haar cascade classifier, which is trained to recognize specific patterns, such as human faces.
* The parameters 1.3 and 4 control the sensitivity and accuracy of the face detection algorithm. Here's what they represent:
* The scaleFactor parameter (1.3 in this case) specifies how much the image size is reduced at each image scale. It is used to create a scale pyramid of images, where each layer of the pyramid is a smaller version of the previous one. A smaller scaleFactor value will increase the sensitivity of the detector, but it may also increase false positives. A larger value will decrease sensitivity but may miss some faces. You can experiment with different values to find the optimal balance for your use case.
* The minNeighbors parameter (4 in this case) specifies how many neighbors each candidate rectangle should have to retain it. The algorithm detects multiple rectangles for each face, and this parameter helps filter out false positives by specifying the minimum number of overlapping rectangles needed to consider it a valid detection. Increasing the value of minNeighbors helps reduce false positives but may also remove some true detections. Again, you can adjust this value based on your requirements.

10. Drawing rectangles around detected faces:

- `**for (x, y, a, b) in hum:**` iterates over each detected face, where `(x, y)` represents the top-left corner of the face rectangle, and `(a, b)` represents the width and height of the rectangle. It draws a colored rectangle around each detected face using `cv2.rectangle()`.

11. Displaying the camera feed:

- `**cv2.imshow("camera", f)**` displays the current frame `f` with the drawn rectangles on a window named "camera". `imshow()` is a function from OpenCV used for displaying images.

12. Recording video:

- If the key 'r' is pressed **(`if cv2.waitKey(2) & 0xff == ord("r")`)**, it starts recording video:

- `**video = cv2.VideoWriter(f"vid{len(os.listdir()):03d}.mp4", fou, 20.0, (640, 480))**` creates a `VideoWriter` object to write the video frames into a file. The filename is generated based on the number of existing files in the directory.

- The program prints a message indicating that recording has started, and `rec` variable is set to `True`.

13. Stopping video recording:

- If the key 's' is pressed **(`if cv2.waitKey(2) & 0xff == ord("s")`)**, it stops the recording:

- `video.release()` releases the `VideoWriter` object, indicating the end of video recording.

**Img Color finding :**

1. Importing the necessary modules:

- `cv2` module from OpenCV is imported.

- `numpy` module is imported, typically imported as `np`, for array manipulation.

2. Defining color range values:

- `lr` and `ur` are dictionaries containing lower and upper range values respectively, for different colors. These range values are defined in the HSV color space.

- `color\_names` is a dictionary that maps color names to their corresponding BGR values.

3. Loading and reading the image:

- `img = cv2.imread('test.jpg')` reads the image file named 'test.jpg' into the `img` variable.

4. Converting the image to HSV color space:

- `hsv = cv2.cvtColor(img, cv2.COLOR\_BGR2HSV)` converts the image `img` from the BGR color space to the HSV color space. This conversion is necessary for color detection.

5. Performing color detection:

- A loop iterates over the colors and their corresponding lower and upper range values.

- `mask = cv2.inRange(hsv, np.array(lr), np.array(ur))` creates a binary mask that isolates the pixels within the specified lower and upper range values for a particular color.

- Morphological operations are performed to improve the mask:

- `kernel = np.ones((5,5), np.uint8)` defines a kernel used for morphological operations.

- `opening = cv2.morphologyEx(mask, cv2.MORPH\_OPEN, kernel)` performs morphological opening on the mask to remove noise and smoothen the detected regions.

6. Finding contours and drawing bounding rectangles:

- `c, hierarchy = cv2.findContours(opening, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)` finds contours in the binary mask. The `RETR\_EXTERNAL` flag retrieves only the external contours, and the `CHAIN\_APPROX\_SIMPLE` flag compresses horizontal, vertical, and diagonal segments.

- A loop iterates over each contour and checks its area. If the area is above a certain threshold (1000 in this case), it is considered a valid detection.

- Bounding rectangles are drawn around the detected contours using `cv2.rectangle()`. The rectangles are colored using the BGR values from the `color\_names` dictionary.

- Text labels with the color names are added to the bounding rectangles using `cv2.putText()`.

7. Displaying the image:

- `cv2.imshow("Image", img)` displays the original image with the detected color regions and labels.

- `cv2.waitKey(0)` waits for a key press.

- `cv2.destroyAllWindows()` closes any open windows.

This code demonstrates how to perform color detection and visualize the detected color regions in an image.

**Live Cam colour Detection :**

1. The code defines dictionaries **`lr`, `ur`**, and `**color\_names**` that specify the lower and upper range values for different colors, as well as the corresponding BGR color values for drawing rectangles and text.

2. The code initializes a video capture object `**cap**` to capture video from the default camera (index 0).

3. The code enters a while loop that continuously reads frames from the video capture and performs color detection on each frame.

4. Inside the loop, the code reads a frame from the video capture using `**cap.read()**`, where `**r**` is used to discard the return value indicating whether the frame was successfully read.

5. The code converts the frame from the BGR color space to the HSV color space using `**cv2.cvtColor()**`.

6. For each color specified in the dictionaries, the code creates a mask using `**cv2.inRange()**` to detect the color within the specified lower and upper range values in the HSV color space.

7. The code applies morphological operations using `**cv2.morphologyEx()**` to remove noise from the mask.

8. Contours are found in the processed mask using `**cv2.findContours()**`.

9. For each contour, the code checks if its area is greater than 1000 pixels to filter out small noise. If the area is sufficient, a bounding rectangle is drawn around the contour using `**cv2.rectangle()**` and the color name is displayed as text using `**cv2.putText()**`.

10. The resulting frame with color detections is displayed in a window named "Image" using `**cv2.imshow()**`.

11. The loop continues until the '**q**' key is pressed, at which point the video capture is released using `**cap.release()**` and all windows are closed using `**cv2.destroyAllWindows()**`.

**Color code’s :**

Here are some commonly used color codes in OpenCV:

White: (255, 255, 255)

Black: (0, 0, 0)

Red: (0, 0, 255)

Green: (0, 255, 0)

Blue: (255, 0, 0)

Yellow: (0, 255, 255)

Cyan: (255, 255, 0)

Magenta: (255, 0, 255)

Orange: (0, 165, 255)

Pink: (203, 192, 255)

Gray: (128, 128, 128)

Light Gray: (192, 192, 192)

Dark Gray: (64, 64, 64)

These are just a few examples, and you can create custom colours by specifying the intensity values for each channel (Blue, Green, Red) ranging from 0 to 255.

**Color code (lower Range to Upper Range )**

Certainly! Here are some common color codes in both upper range (UR) and lower range (LR) format for OpenCV's HSV color space:

Red:

UR: (10, 255, 255)

LR: (0, 100, 100)

Green:

UR: (70, 255, 255)

LR: (50, 100, 100)

Blue:

UR: (130, 255, 255)

LR: (110, 100, 100)

Yellow:

UR: (50, 255, 255)

LR: (30, 100, 100)

Orange:

UR: (25, 255, 255)

LR: (5, 100, 100)

Pink:

UR: (170, 255, 255)

LR: (150, 100, 100)

Skin:

UR: (25, 255, 255)

LR: (0, 50, 50)

These color codes represent the upper range (UR) and lower range (LR) values of the HSV color space. They are used to define the range of colors to be detected or filtered in applications such as color segmentation or object tracking.

**Face detection ---Task (CIE-III)**

1. **Imports:**

* **cv2**: The OpenCV library for computer vision tasks.
* **os**: The OS module for working with file paths.

1. **Loading the Face Detection Model:**

* The code loads the pre-trained face detection model called "**haarcascade\_frontalface\_default.xml**" using the **Cascade Classifier** class provided by OpenCV.

1. **Opening the Camera:**

* The code initializes the camera capture using **cv2.VideoCapture(0)**. **The argument "0" specifies the index of the camera** to use (in case there are multiple cameras connected).

1. **Face Detection Loop:**
2. The code enters an infinite while loop to continuously capture frames from the camera and perform face detection.
3. It reads the current frame from the camera using **camera.read()**, which returns the frame in the variable frame.
   1. **r is a boolean value that indicates whether the frame was successfully read**. It will be True if the frame was read successfully and False if there was an issue or if the video capture has reached its end.
   2. **frame** **is the actual frame read from the video capture**. It is an image represented as a NumPy array.
4. The captured frame is converted to grayscale using **cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)**.
5. The face detection is performed using the **detectMultiScale** method of the face detection model **(file)**. **It takes the grayscale image as input and returns a list of rectangles representing the detected faces.**
6. A rectangle is drawn around each detected face using **cv2.rectangle**, with the color **(0, 0, 255)** and a **thickness of 2**.
7. The frame with the drawn rectangles is displayed in a window named "camera" using **cv2.imshow()**.
8. If the '**q**' key is pressed, the loop is terminated with break.
9. **Saving Face Images:**

* Inside the face detection loop, if the 'r' key is pressed (currently commented out), the code captures the current frame and saves it as an image.
* It generates a timestamp for the image filename using **datetime.datetime.now().strftime("%Y-%m-%d %H-%M-%S")**.
  + **datetime**: It is a module that provides classes for working with dates and times in Python.
  + datetime.now(): The now() method is called on the datetime class and returns a datetime object representing the current date and time.

The now() method does not require any arguments. When called, it captures the current date and time information from the system's clock and creates a datetime object with that information.

* + **datetime.datetime.now()**: This retrieves the current date and time as a datetime object. It represents the current timestamp.
  + .**strftime("%Y-%m-%d %H-%M-%S")**: The strftime() method is used to format the datetime object as a string based on the specified format codes. In this case, the format codes used are:
  + %Y: Represents the four-digit year.
  + %m: Represents the two-digit month (with leading zero, if necessary).
  + %d: Represents the two-digit day of the month (with leading zero, if necessary).
  + %H: Represents the two-digit hour (in 24-hour format, with leading zero, if necessary).
  + %M: Represents the two-digit minute (with leading zero, if necessary).
  + %S: Represents the two-digit second (with leading zero, if necessary).
  + Combining these format codes in the given order, the strftime() method returns a string representing the current timestamp in the format "YYYY-MM-DD HH:MM:SS". This string is used as the filename for the saved image.
* The frame is saved as an image using cv2.imwrite() with the timestamp as the filename.

1. **Exiting the Program:**

* When the loop is terminated (by pressing 'q' key), the camera capture is released using **camera.release()**.
* Finally, all windows created by OpenCV are closed using **cv2.destroyAllWindows()**.