Student: Aditya Sandhu

University: Colorado State University Global

Course: 25FA - CSC515 - 1 [Module 3 – Introduction to Image Processing]

Portfolio Assignment [ Blending Images in OpenCV ]

Instructor: Dr. Dong Nguyen

Date – 09/24/2025

GIT LINKS

Document Link –

Python File -

Deepfakes are synthetic media, such as images, videos, or audio recordings, created or manipulated using artificial intelligence techniques, particularly deep learning algorithms, to convincingly alter reality. Originating from a blend of "deep learning" and "fake," this technology can swap faces in photos or videos, generate realistic speech from text, or fabricate events that never occurred, often making it difficult to distinguish from authentic content. While deepfakes have creative applications in entertainment and education, they pose significant risks, including misinformation, identity theft, and cybersecurity threats, as seen in the provided code which demonstrates a basic face swap using OpenCV to replace one person's likeness with another's in an image.

Figure 1 Python function for validating and loading images with OpenCV.

A screenshot of a computer program

AI-generated content may be incorrect.

This code defines a function fetch\_images that loads two images from given file paths. It first checks if the files exist, then attempts to read them into memory using OpenCV. If either file is missing or unreadable, it raises an error; otherwise, it returns both images.

import cv2  
import numpy as np  
import os

This code begins by importing three modules - cv2 for computer vision tasks, numpy for numerical operations, and os for interacting with the file system. It then defines a function named fetch\_images that takes two arguments, origin\_img\_path and destination\_img\_path, which are expected to be file paths to images.

def fetch\_images(origin\_img\_path, destination\_img\_path):  
 """Retrieve and confirm the origin and destination images are accessible."""  
 # Verify existence of origin image file  
 if not os.path.exists(origin\_img\_path):  
 raise ValueError(f"Origin image missing at: {origin\_img\_path}")  
 # Verify existence of destination image file  
 if not os.path.exists(destination\_img\_path):  
 raise ValueError(f"Destination image missing at: {destination\_img\_path}")  
 # Attempt to read origin image into memory  
 origin\_img = cv2.imread(origin\_img\_path)  
 # Attempt to read destination image into memory  
 destination\_img = cv2.imread(destination\_img\_path)  
 # Check if origin image loaded successfully, else error

The function fetch\_images starts by checking whether the file at origin\_img\_path exists on disk using os.path.exists. If the file is not found, it immediately raises a ValueError with a descriptive message pointing to the missing path. It performs the same check for the destination\_img\_path, raising an error if the file does not exist.

Once both paths are confirmed to exist, the function attempts to load the images into memory using cv2.imread. The first call assigns the loaded data to origin\_img and the second call assigns to destination\_img. Next, the function verifies whether the origin image was successfully loaded. If origin\_img is None, meaning OpenCV could not parse the file (for example, due to an unsupported format or corruption), it raises a ValueError explaining the problem. The same logic is intended for destination\_img, although in the provided code, the if destination\_img is None: line is missing, causing the check to always raise the error for the destination image regardless.

if origin\_img is None:  
 raise ValueError(f"Unable to parse origin image at: {origin\_img\_path} (verify format or corruption)")  
 # Check if destination\_img is None:  
 raise ValueError(f"Unable to parse destination image at: {destination\_img\_path} (verify format or corruption)")  
 return origin\_img, destination\_img

If both images are successfully loaded and validated, in the function returns them as a tuple (origin\_img, destination\_img), making them ready for downstream computer vision processing.

Figure 2 Python functions for grayscale conversion, face detection, and overlay creation in OpenCV.

A screenshot of a computer program

AI-generated content may be incorrect.

This code provides helper functions for face detection and preprocessing. It includes grayscale conversion, initializing a Haar cascade recognizer, detecting facial areas, and cropping or resizing faces. Additionally, it generates an elliptical overlay for blending image elements smoothly.

def prepare\_grayscale(picture):  
 """Convert the provided picture to a single-channel grayscale version for analysis."""  
 # Use color conversion to strip RGB and retain intensity  
 return cv2.cvtColor(picture, cv2.COLOR\_BGR2GRAY)

This function prepare\_grayscale takes a color image as input and converts it into a grayscale version. It does so by stripping away the RGB color channels and retaining only intensity values using OpenCV’s cvtColor method.

def setup\_recognizer():  
 """Initialize the recognition tool with a standard pretrained model for frontal views."""  
 # Load the XML model file for detection patterns  
 return cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade\_frontalface\_default.xml')

This function setup\_recognizer initializes a face recognizer using a pre-trained Haar cascade model for frontal face detection. It loads the XML configuration file provided by OpenCV to enable recognition of facial patterns.

def find\_facial\_areas(mono\_picture, recognizer):  
 """Scan the monochrome picture to locate potential facial zones."""  
 # Apply multi-scale scanning with specified scaling and neighbor thresholds  
 return recognizer.detectMultiScale(mono\_picture, scaleFactor=1.3, minNeighbors=5)

This function find\_facial\_areas scans a grayscale image to detect possible facial regions. It uses the recognizer’s detectMultiScale method with scaling and neighbor thresholds to identify face-like zones.

def extract\_and\_adjust\_face(photo, face\_coords, new\_dimensions):  
 """Crop the face area and scale it to fit the new size."""  
 fx, fy, fw, fh = face\_coords  
 face\_region = photo[fy:fy + fh, fx:fx + fw]  
 return cv2.resize(face\_region, new\_dimensions)

This function extract\_and\_adjust\_face crops the face from an image using the provided coordinates. It then resizes the cropped region to match the given dimensions.

def create\_fusion\_overlay(vertical\_size, horizontal\_size):  
 """Build a smooth overlay shape for fusing image elements elliptically."""  
 # Initialize a blank array matching the dimensions  
 fusion\_overlay = np.zeros((vertical\_size, horizontal\_size, 3), dtype=np.uint8)  
 # Draw a filled ellipse to cover the area  
 cv2.ellipse(fusion\_overlay, (horizontal\_size // 2, vertical\_size // 2), (horizontal\_size // 2, vertical\_size // 2),  
 0, 0, 360, (255, 255, 255), -1)  
 return fusion\_overlay

The function create\_fusion\_overlay creates a blank image of the given size and draws a filled white ellipse on it. The resulting overlay can then be used to smoothly blend or mask image regions.

def perform\_seamless\_blend(source\_face, target\_photo, mask, target\_center):  
 """Apply advanced cloning to integrate the source face into the target."""  
 return cv2.seamlessClone(source\_face, target\_photo, mask, target\_center, cv2.NORMAL\_CLONE)

The function perform\_seamless\_blend blends a source face into a target photo using OpenCV’s seamlessClone. It applies advanced cloning with a mask and target center to create a natural-looking integration.

def apply\_annotation(picture, annotation\_content):  
 """Add a text label to the picture at a fixed position."""  
 # Position text near the bottom with chosen font and color  
 cv2.putText(picture, annotation\_content, (50, picture.shape[0] - 50),  
 cv2.FONT\_HERSHEY\_SIMPLEX, 1, (255, 255, 255), 2)

The function apply\_annotation places a text label onto an image at a fixed position near the bottom. It uses OpenCV’s putText with a white font for clear annotation.

code\_location = os.path.dirname(os.path.abspath(\_\_file\_\_))  
user\_portrait = os.path.join(code\_location,  
 "bearded\_selfie\_in\_car.jpeg")   
web\_portrait = os.path.join(code\_location, "online\_arms\_crossed.jpg")

This snippet first determines the absolute path of the script’s directory. It then builds two complete file paths: one pointing to a local portrait image (bearded\_selfie\_in\_car.jpeg) and another pointing to an online-sourced portrait (online\_arms\_crossed.jpg).

# Output the working location and image paths for verification  
print(f"Code location: {code\_location}")  
print(f"User portrait path: {user\_portrait}")  
print(f"Web portrait path: {web\_portrait}")

These, print statements output the script’s directory along with the full paths of the user and web portrait images to verify they are set correctly.

Figure 3 Python script implementing face detection and seamless blending workflow in OpenCV.

A screen shot of a computer program

AI-generated content may be incorrect.

try:  
 source\_photo, target\_photo = fetch\_images(user\_portrait, web\_portrait)  
 source\_mono = prepare\_grayscale(source\_photo)  
 target\_mono = prepare\_grayscale(target\_photo)  
  
 face\_detector = setup\_recognizer()  
  
 source\_detections = find\_facial\_areas(source\_mono, face\_detector)  
 target\_detections = find\_facial\_areas(target\_mono, face\_detector)  
  
 if len(source\_detections) == 0 or len(target\_detections) == 0:  
 raise ValueError("Unable to detect a face in one or both photos")

This try except block, begins by loading two portrait images and converting them into grayscale to simplify facial analysis. It then initializes a Haar cascade face recognizer and applies it to both grayscale images to detect facial regions. If no face is found in either image, the code raises an error to halt further processing.

# Select primary face from each  
 source\_coords = source\_detections[0]  
 target\_coords = target\_detections[0]  
 target\_width, target\_height = target\_coords[2], target\_coords[3]  
  
 adjusted\_source\_face = extract\_and\_adjust\_face(source\_photo, source\_coords, (target\_width, target\_height))  
  
 integration\_mask = create\_fusion\_overlay(target\_height, target\_width)  
  
 blend\_center = (target\_coords[0] + target\_width // 2, target\_coords[1] + target\_height // 2)  
 blended\_result = perform\_seamless\_blend(adjusted\_source\_face, target\_photo, integration\_mask, blend\_center)  
  
 apply\_annotation(blended\_result, "Deepfake Blend Example")  
  
 # Store the final image  
 result\_file = os.path.join(code\_location, "blended\_deepfake\_result.jpg")  
 cv2.imwrite(result\_file, blended\_result)  
  
 print(f"Generated deepfake saved to {result\_file}")  
except Exception as e:  
 print(f"Error: {e}")

This segment selects the first detected face from both the source and target images, then resizes the source face to match the target face’s width and height. An elliptical integration mask is created, and OpenCV’s seamless cloning function blends the adjusted source face into the target image at the calculated center point. Finally, the blended image is annotated with a label, saved to disk as a new file, and a confirmation message is printed, with error handling in place to catch exceptions.

Figure 4 Bearded selfie taken inside a car (bearded\_selfie\_in\_car.jpeg).

A person with a beard taking a selfie

AI-generated content may be incorrect.

Figure 5 Portrait of a smiling man with arms crossed (online\_arms\_crossed.jpg).

A person with beard and mustache smiling with arms crossed

AI-generated content may be incorrect.

Figure 6 Blended deepfake result showing face integration with annotation (blended\_deepfake\_result.jpg).

A person with a beard and mustache

AI-generated content may be incorrect.

The provided code demonstrates a complete workflow for generating a basic deepfake image using OpenCV, starting from loading and validating input images, preprocessing them through grayscale conversion and face detection, and finally performing seamless face blending with annotation. While the exercise highlights the technical steps required to manipulate media convincingly, it also underscores the dual-use nature of deepfake technology—capable of serving creative and educational purposes, yet simultaneously raising concerns about ethical misuse, misinformation, and security risks.

**References**

DataCamp. (n.d.). *Face detection in Python with OpenCV*. DataCamp. <https://www.datacamp.com/tutorial/face-detection-python-opencv>

GeeksforGeeks. (n.d.). *Reading an image in OpenCV using Python*. GeeksforGeeks. <https://www.geeksforgeeks.org/python/reading-image-opencv-using-python/>

MDPI. (2021). *Deepfake detection with computer vision*. *Sensors, 21*(21), 7367. <https://www.mdpi.com/1424-8220/21/21/7367>

OpenCV. (2018). *Image file reading and writing*. OpenCV Documentation. <https://docs.opencv.org/3.4/d4/da8/group__imgcodecs.html>

OpenCV. (2020). *Deepfake detection with computer vision*. OpenCV Blog. <https://opencv.org/blog/deepfake-detection-with-computer-vision/>