GEBZE TECHNICAL UNIVERSITY

COMPUTER ENGINEERING

CSE464 DIGITAL IMAGE PROCESSING

FINAL PROJECT REPORT

TEETH SEGMENTATION WITH IMAGE PROCESSING TECHNIQUES

STUDENT NAME&SURNAME: Yağız Hakkı AYDIN

STUDENT NUMBER: 1901042612

ABSTRACT

In this study, a systematic exposition of various digital image processing techniques will be presented to elucidate the process of segmenting teeth. The step-by-step elucidation aims to provide a comprehensive understanding of the employed methodologies and their application in achieving precise segmentation results.

CONTENTS

1. INTRODUCTION

This project is dedicated to the task of teeth segmentation within facial images. While conventional approaches often resort to artificial intelligence methodologies for this purpose, the distinctive aspect of this project lies in its emphasis on achieving teeth segmentation without employing artificial intelligence. Instead, artificial intelligence is exclusively utilized for the preliminary task of mouth detection. The methodology involves an initial step of detecting the mouth region in facial images, followed by a series of carefully selected image processing techniques to facilitate teeth segmentation. Additionally, a user-friendly interface has been developed to expedite the loading of images and display the segmentation results efficiently. The choice of utilizing images featuring smiling individuals serves as a demonstrative means to illustrate the efficacy of the program's functionality.

2. PROJECT WORKFLOW

Workflow of this project will be explained on a example image (fig. 2.0).

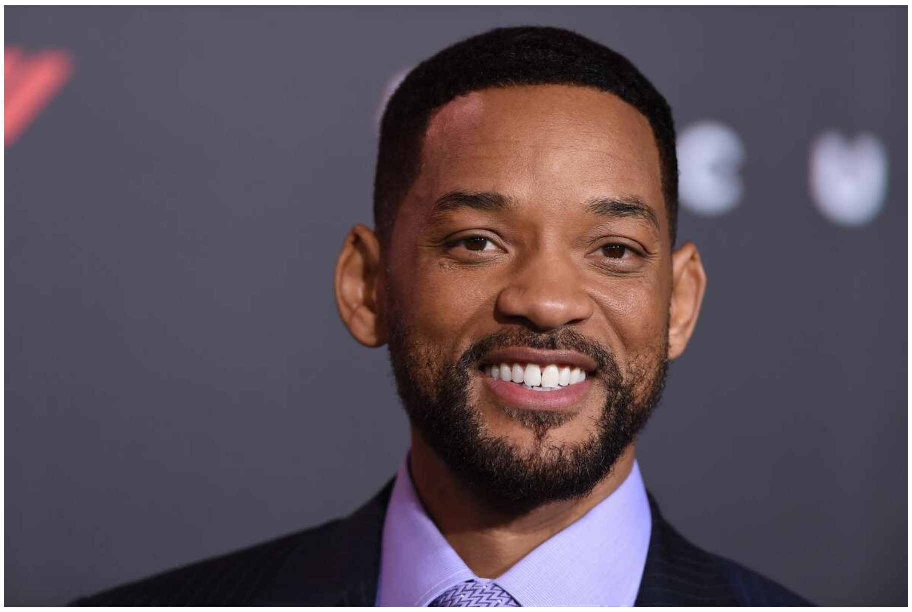


Figure 2.0

2.1 Detecting and Isolating Mouth

At the outset, the essential task involves detecting the precise position of the mouth and subsequently isolating it from the remainder of the image. This objective is accomplished through the utilization of the Python dlib library. Consequently, for the specified step, a representation of the mouth against a uniformly black background is obtained. (Example result at fig. 2.1)

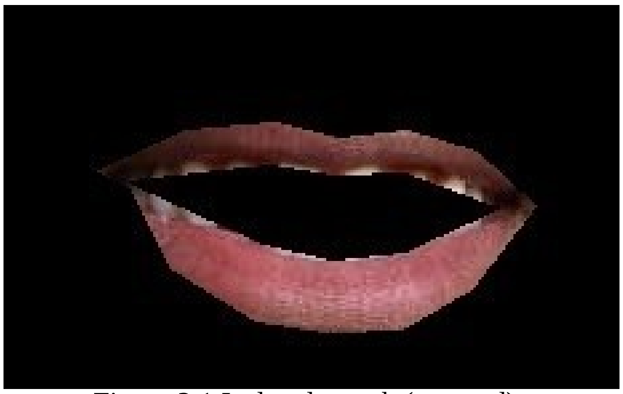


Figure 2.1 Isoletad mouth (zoomed)

2.2 Isolating Teeth

The process of isolating teeth relies on the prior isolation of the mouth, achieved by rendering the background white. This involves systematically painting each column from top to bottom until the upper border of the lips is reached, and then repeating the process from bottom to top. Subsequently, the teeth are isolated against a pitch-black background by painting the corresponding pixels of the original image black where the lips-isolated image is not black. (Example result at fig. 2.2)

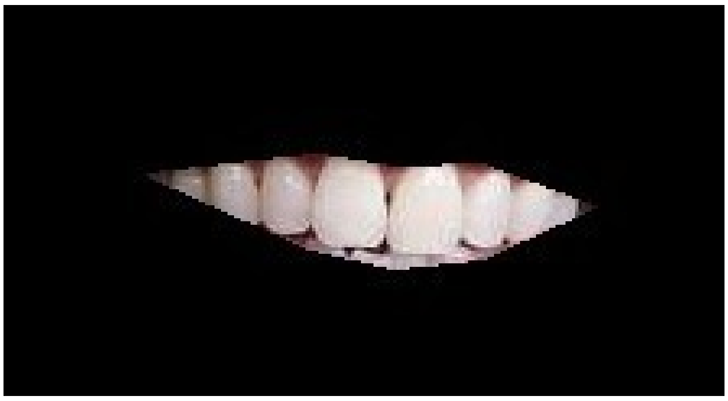


Figure 2.2 Isoletad teeth (zoomed)

2.3 Histogram Equalization on Grayscale Image

Upon the successful isolation of the teeth, the subsequent image processing commences by transforming the image into grayscale (fig 2.31). This grayscale conversion is followed by the application of histogram equalization(fig. 2.32) , enhancing the clarity of details within the image.

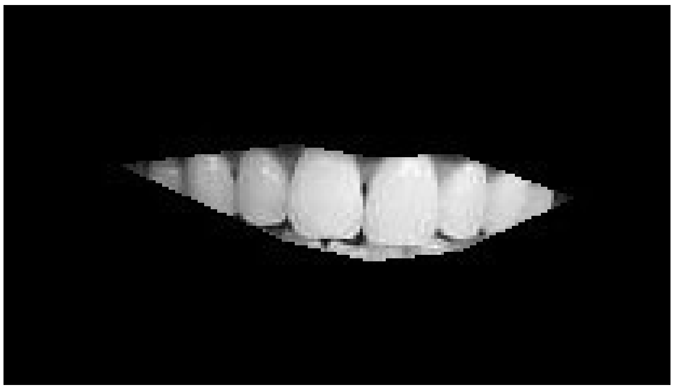


Figure 2.31 Gray Scaled

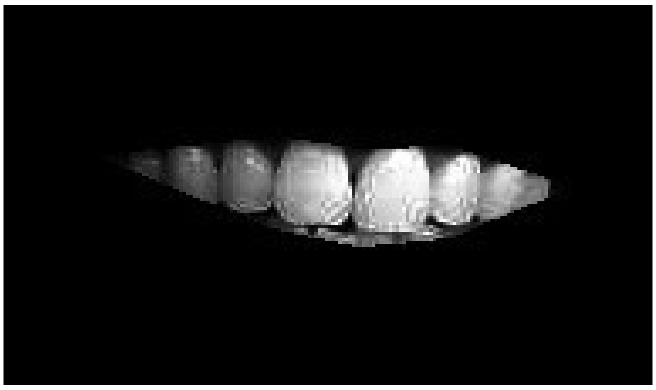


Figure 2.32 Histogram Equalization applied

2.4 Adaptive Thresholding

Following the histogram equalization process, the code applies adaptive thresholding to the grayscale image. This operation strategically assigns pixel values based on local intensity variations, resulting in a binary image where pixels exceeding a dynamically adjusted threshold are set to white, while others are set to black. The adaptive thresholding technique employed utilizes a Gaussian-weighted sum of neighborhood values, contributing to the segmentation of distinct regions in the image. (Result at fig 2.4)

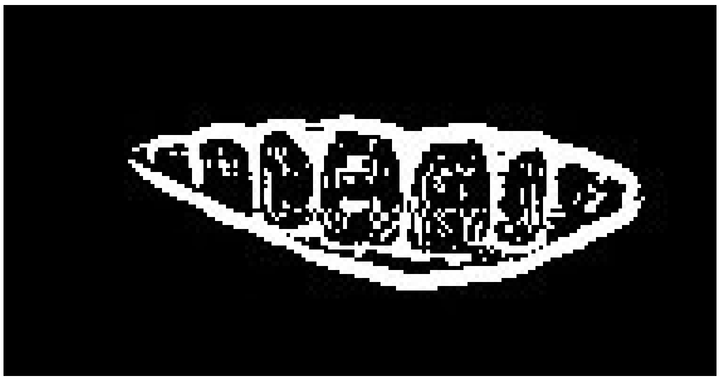


Figure 2.4 Adaptive Thresholding applied

2.5 Connected Components Labeling

Subsequent to the adaptive thresholding operation, labeling connected components within the binary image is completed. This operation assigns unique labels to contiguous regions in the binary image, facilitating the identification and differentiation of distinct objects or regions. The resulting labeled image and the total number of labeled regions are obtained through this process, setting the stage for further analysis and manipulation of segmented components. (Result is at fig. 2.51,contrast increased version of result is at fig. 2.52)

Figure 2.51 Connected Components Labeling

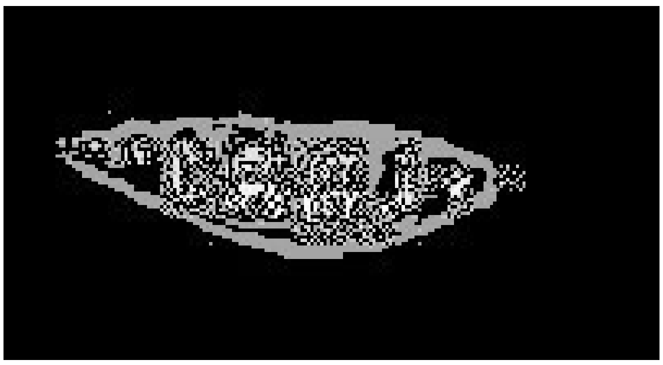


Figure 2.52 (High Contrast)

2.6 Small Object Filtering

Subsequent to the labeling of connected components, the workflow continues with morphological operations to filter out small regions in the labeled image. Specifically, regions with a pixel count below a specified threshold (`min_size=100`) gets eliminated. This operation effectively removes noise or insignificant components, enhancing the precision of the segmented regions and facilitating a more accurate representation of the distinct anatomical structures within the image. (Result is at fig. 2.61,contrast increased version of result is at fig. 2.62)

Figure 2.61 Small Object Filtering Applied

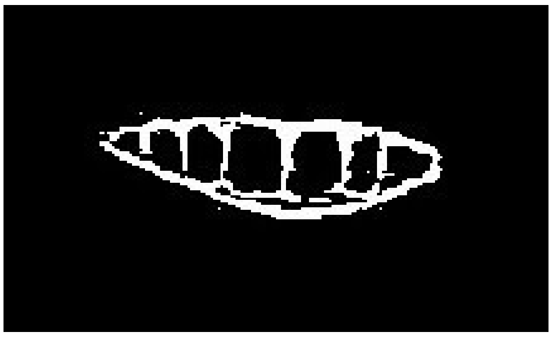


Figure 2.62 (High Contrast)

2.7 Filling Gaps In Edges

Following the removal of small objects, the code transforms the data type of the labeled image to an 8-bit unsigned integer format. Subsequently, a morphological closing operation is applied to the labeled image. This entails utilizing a square-shaped kernel with a specified size to fill gaps in the edges of the labeled regions. The kernel size is adjusted based on the size of the gaps observed in the labeled image. The morphological closing operation aids in smoothing the contours of the segmented regions, promoting connectivity and refining the overall structure of the labeled image.

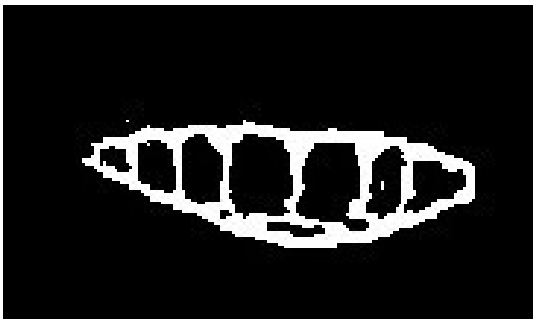


Figure 2.71 Edge Filling Applied

Figure 2.72 (High Contrast)

2.8 Connected Components Labeling and Coloring

Subsequent to the morphological closing operation, the code initiates the inversion of pixel values in the labeled image. This inversion is followed by the labeling of connected components in the binary image. Labeling considers 8-connectivity between pixels and returns the total number of labeled components. This operation is fundamental for identifying and distinguishing distinct connected components within the binary image, setting the stage for subsequent analyses or visualizations. Following those operations, each region is painted to different color. (Result at fig. 2.8)

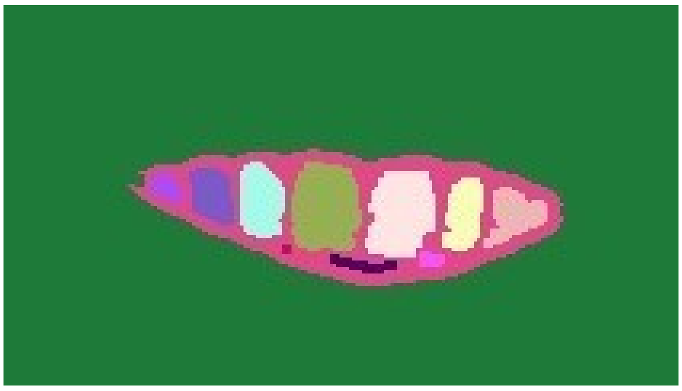


Figure 2.8 Colored Regions

2.9 Removing Background And Borders

Following the assignment of distinct colors to each labeled region, a series of operations were conducted to eliminate extraneous elements in the image. Commencing with the removal of the background, the color of the top-left pixel was selected as the reference, and all pixels sharing this color value were subsequently rendered black. This effectively eradicated the background, leaving only the delineated regions corresponding to the teeth. Subsequently, the identification and elimination of borders between teeth and the background ensued. This process involved a top-to-bottom examination of image columns, where the first encountered non-black color was attributed as the border color. All pixels exhibiting this color were then modified to black, resulting in an image wherein the regions corresponding to teeth were distinctly colored, while all other areas were rendered black. This sequence of operations serves to enhance the visual representation of teeth within the image by eliminating extraneous elements and emphasizing the structural integrity of the segmented regions.(result is at fig. 2.9)

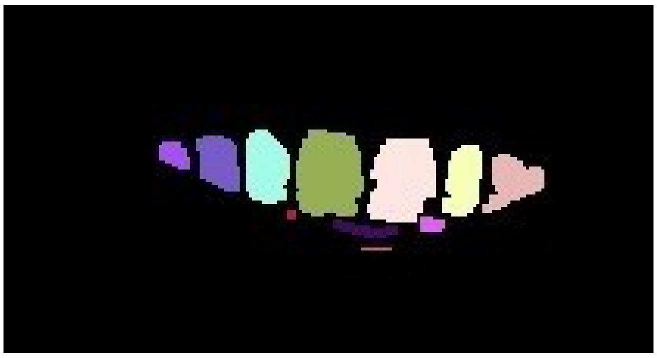


Figure 2.9 Colored Teeth

2.10 Modifying Original Image and Obtaining Final Result

Subsequent to the acquisition of distinctively colored regions corresponding to the teeth, a comparative analysis was conducted between the original image and the processed image. Each pixel in the original image underwent replacement with the corresponding pixel from the processed image, contingent upon the presence of a non-black pixel at the corresponding position in the processed image. This procedural step culminated in the completion of the teeth segmentation process. (Result is at fig. 2.10)

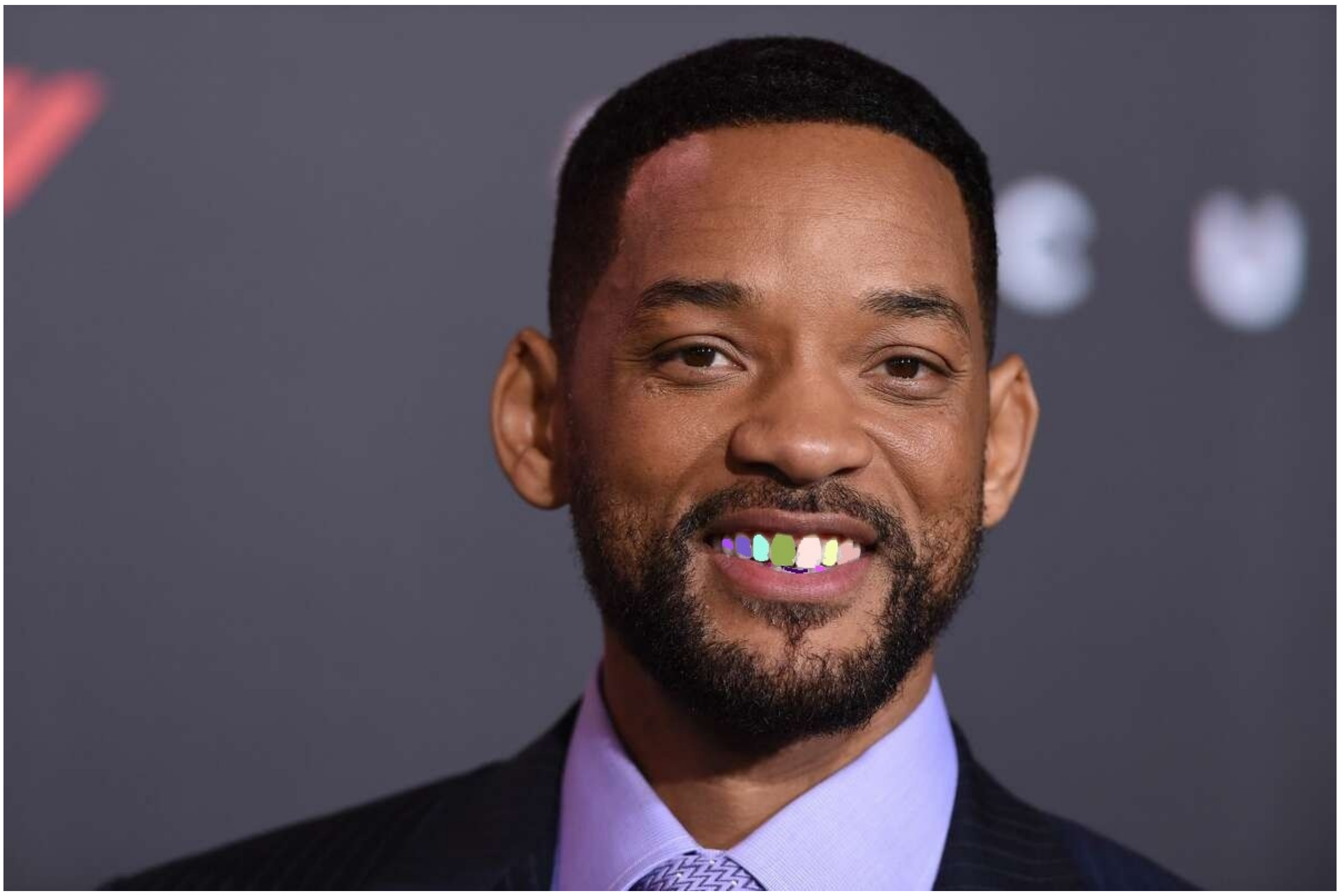


Figure 2.10 Final Result

3. CONCLUSION

In conclusion, this project undertook the task of teeth segmentation within facial images, employing a comprehensive and multi-step approach. The initial phase involved the meticulous isolation of the mouth region, accomplished through a series of image processing operations facilitated by the dlib and OpenCV libraries. Subsequent steps included the strategic refinement of the lips representation, adaptive thresholding, connected components labeling, and morphological operations to enhance the precision of teeth segmentation. The removal of small objects and the application of morphological closing further contributed to refining the segmented regions. Following the assignment of distinct colors to each labeled region, background elimination and border detection were executed to emphasize the regions corresponding to teeth. The final stage involved pixel-wise comparison between the original and processed images, culminating in the completion of teeth segmentation.

The resulting image effectively showcased the isolated teeth against a uniform background. This methodology, founded on image processing techniques and carefully selected algorithms, contributes to the nuanced segmentation of teeth without relying on artificial intelligence, underscoring the project's distinctive approach. The efficacy of the process is demonstrated through the visual representation in Figure 2.10, affirming the successful completion of the teeth segmentation objective.

APPENDIX A (TEETH DETECTION)

```
import dlib
import cv2
import numpy as np



def isolate_mouth_without_teeth(image):
    # Load the pre-trained facial landmark predictor
    predictor_path = "shape_predictor_68_face_landmarks.dat"
    detector = dlib.get_frontal_face_detector()
    predictor = dlib.shape_predictor(predictor_path)

    output_image = image.copy()  # Create a copy for the output

    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Detect faces in the image
    faces = detector(gray)

    # Loop over detected faces
    for face in faces:
        # Get facial landmarks
        landmarks = predictor(gray, face)

        # Extract full mouth region points
        mouth_points = [landmarks.part(i) for i in range(48, 68)]

        # Create a mask for the mouth region
        mouth_mask = np.zeros_like(gray)
        points = np.array([(p.x, p.y) for p in mouth_points], np.int32)
        cv2.fillPoly(mouth_mask, [points], 255)

        # Set everything outside the mouth region to black
        output_image = cv2.bitwise_and(output_image, output_image, mask=mouth_mask)

    return output_image
```

```
def isolate_teeth(image):
    lips_isolated_image = isolate_mouth_without_teeth(image)
    result = image.copy()

    # Paint white around lips
    # Leave inside of lips black

    # Top side of lips
    top_side_of_lips_reached = np.zeros(len(lips_isolated_image[0]), dtype=bool)
    for i in range(len(lips_isolated_image)):
        top_side_of_lips_reached |= ~np.all(lips_isolated_image[i] == [0, 0, 0], axis=1)
        lips_isolated_image[i][~top_side_of_lips_reached] = [255, 255, 255]

    # Bottom side of lips
    bottom_side_of_lips_reached = np.zeros(len(lips_isolated_image[0]), dtype=bool)
    for i in range(len(lips_isolated_image) - 1, -1, -1):
        bottom_side_of_lips_reached |= ~np.all(lips_isolated_image[i] == [0, 0, 0], axis=1)
        lips_isolated_image[i][~bottom_side_of_lips_reached] = [255, 255, 255]

    # Set pixels in the result image based on the lips isolation
    result[~np.all(lips_isolated_image == [0, 0, 0], axis=2)] = [0, 0, 0]
    return result

def get_teeth_cropped( image ):
    return isolate_teeth(image)
```

APPENDIX B (TEETH SEGMENTATION)

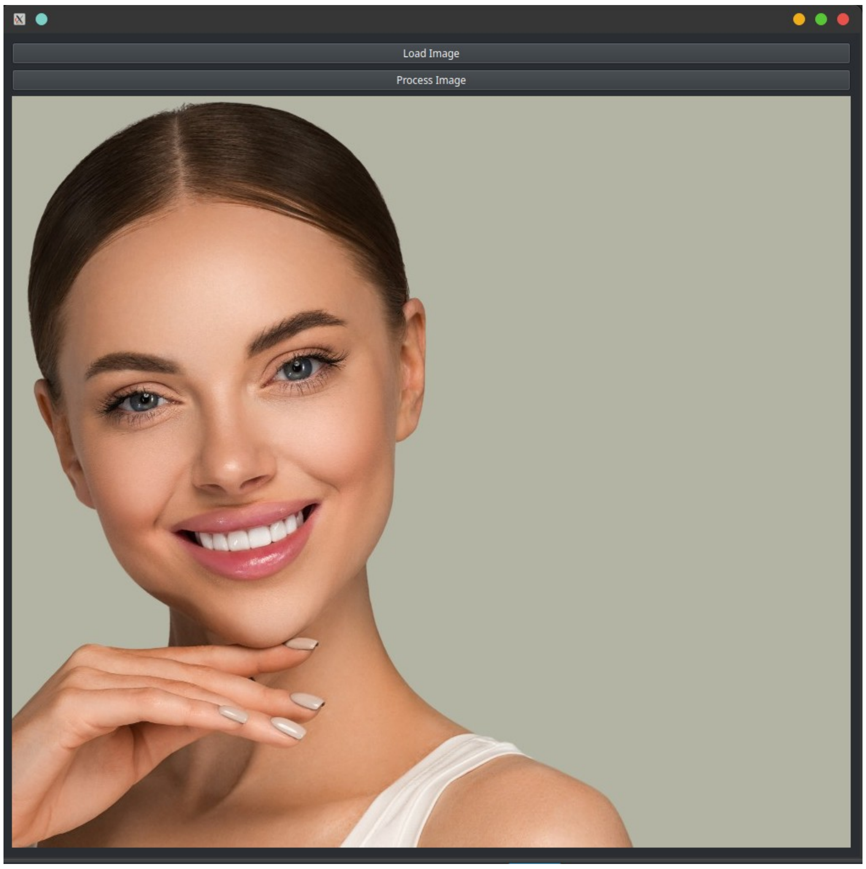
```
import numpy as np
import matplotlib.pyplot as plt
import cv2
from skimage import morphology, color, measure
from scipy import ndimage as ndi
from skimage.segmentation import mark_boundaries
import detect_teeth

def get_result(original_image):

    # Crop teeth and
    teeth = detect_teeth.get_teeth_cropped(original_image)
    teeth_gray = cv2.cvtColor(teeth, cv2.COLOR_BGR2GRAY)

    # Enhance contrast using adaptive histogram equalization
    teeth_gray = cv2.equalizeHist(teeth_gray)

    # Apply adaptive thresholding
    binary_image = cv2.adaptiveThreshold(teeth_gray, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
cv2.THRESH_BINARY_INV, 17, 2)

    # Label connected components in the binary image
    labeled_teeth, num_labels = ndi.label(binary_image)

    # Filter out small regions
    labeled_teeth = morphology.remove_small_objects(labeled_teeth, min_size=100)

    labeled_teeth = labeled_teeth.astype(np.uint8)

    # Apply morphological closing to fill gaps in edges
    kernel_size = 4  # Adjust the kernel size based on the size of the gaps
    kernel = np.ones((kernel_size, kernel_size), np.uint8)
    labeled_teeth = cv2.morphologyEx(labeled_teeth, cv2.MORPH_CLOSE, kernel)

    labeled_teeth = cv2.bitwise_not(labeled_teeth)

    # Label connected components in the binary image
    labeled_teeth, num_labels = measure.label(labeled_teeth, connectivity=2, return_num=True)
```

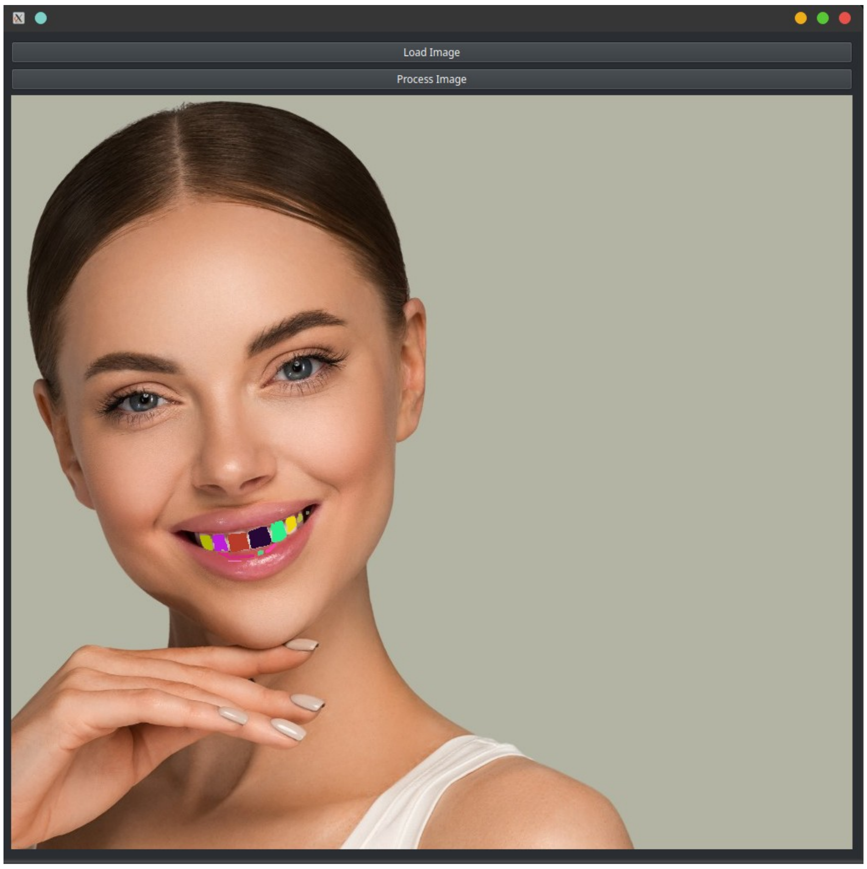
```
# Create an array to store unique colors for each region
colors = np.random.randint(0, 256, (num_labels + 1, 3), dtype=np.uint8)

# Create an empty color image
colored_teeth = np.zeros((labeled_teeth.shape[0], labeled_teeth.shape[1], 3), dtype=np.uint8)

#Paint regions
for label in range(1, num_labels + 1):
    colored_teeth[labeled_teeth == label] = colors[label]

# Specify the background color
color_of_background = colored_teeth[0, 0]

# Find pixels that exactly match the background color
background_pixels = np.all(colored_teeth == color_of_background, axis=-1)

# Set pixels matching the background color to black
colored_teeth[background_pixels] = [0, 0, 0]

#Eliminate Background
lips_color = None
found = False
for i in range(len(colored_teeth)):
    if found:
        break
    for j in range(len(colored_teeth[0])):
        if np.all(colored_teeth[i][j] != np.array([0,0,0]) ):
            lips_color = colored_teeth[i][j]
            break

#Eliminate teeth border
for i in range(len(colored_teeth)):
    for j in range(len(colored_teeth[0])):
        if np.all(colored_teeth[i][j] == lips_color):
            colored_teeth[i][j] = np.array([0,0,0])

#Obtain result
result_image = np.where(colored_teeth != [0, 0, 0], colored_teeth, original_image)

return result_image
```

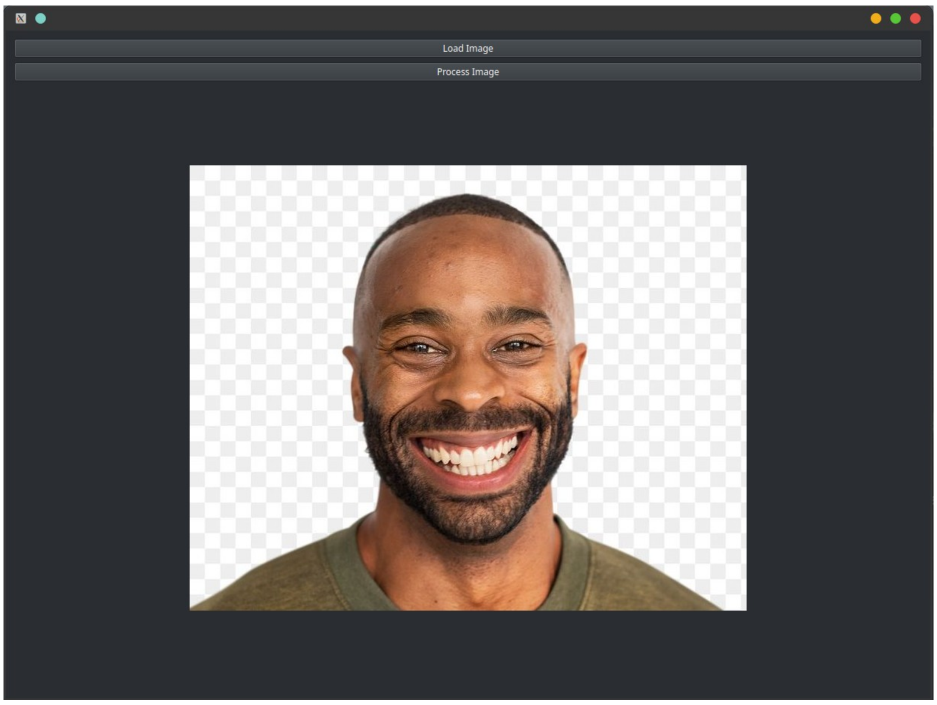
APPENDIX C (MAIN SCRIPT)

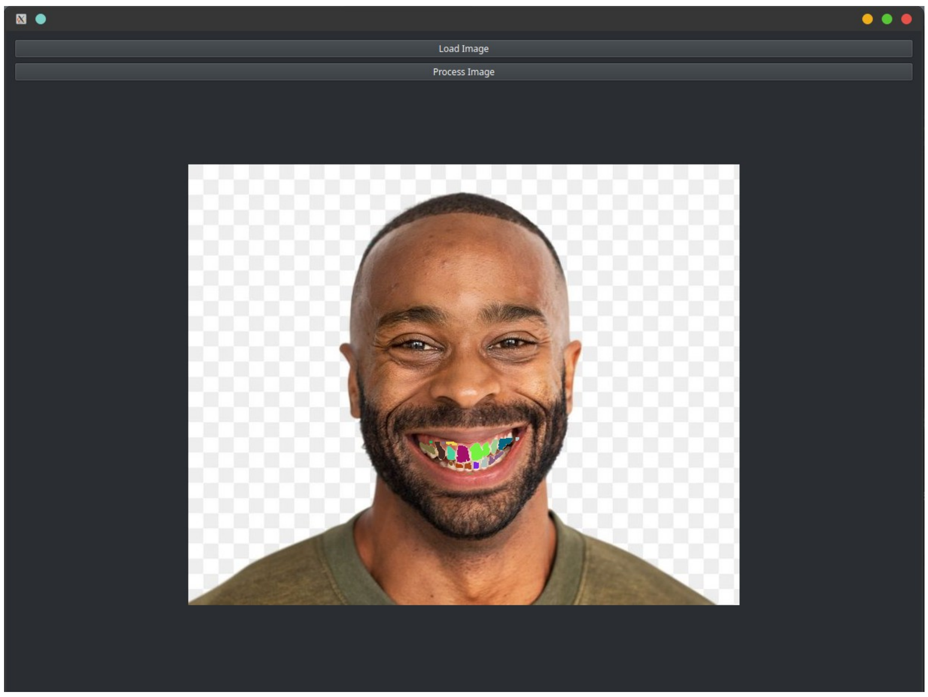
```
import cv2
import numpy as np
from PyQt5.QtCore import Qt
from PyQt5.QtGui import QPixmap, QImage, QImageReader
from PyQt5.QtWidgets import QApplication, QLabel, QVBoxLayout, QWidget, QPushButton, QFileDialog
from functools import partial
import segmentation

class ImageProcessorApp(QWidget):
    def __init__(self):
        super().__init__()

        # Initialize UI components
        self.image_label = QLabel(self)
        self.load_button = QPushButton('Load Image', self)
        self.process_button = QPushButton('Process Image', self)

        # Set up the UI layout
        self.init_ui()

    def init_ui(self):
        layout = QVBoxLayout(self)
        layout.addWidget(self.load_button)
        layout.addWidget(self.process_button)
        layout.addWidget(self.image_label)

        # Connect button signals to functions
        self.load_button.clicked.connect(self.load_image)
        self.process_button.clicked.connect(self.process_image)

    def load_image(self):
        # Open file dialog to select an image
        file_dialog = QFileDialog()
        file_dialog.setFileMode(QFileDialog.ExistingFile)
        if file_dialog.exec_():
            file_path = file_dialog.selectedFiles()[0]
            image = cv2.imread(file_path)

            # Display the original image
            self.show_image(image, self.image_label)

    def process_image(self):
        # Get the current image from the image label
        original_image = self.get_image_from_label(self.image_label)

        if original_image is not None:
            # Process the image using the segmentation.get_result function
            processed_image = segmentation.get_result(original_image)

            # Display the processed image
            self.show_image(processed_image, self.image_label)

    def show_image(self, image, label):
        # Convert the image from BGR to RGB
        image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

        # Convert the image to QImage
        h, w, ch = image_rgb.shape
        bytes_per_line = ch * w
        q_image = QImage(image_rgb.data, w, h, bytes_per_line, QImage.Format_RGB888)

        # Convert QImage to QPixmap
        pixmap = QPixmap.fromImage(q_image)

        # Set the pixmap to the label
        label.setPixmap(pixmap)
        label.setAlignment(Qt.AlignCenter)

    def get_image_from_label(self, label):
        pixmap = label.pixmap()
        if pixmap is not None:
            return self.qimage_to_cv(pixmap.toImage())
        return None
    def qimage_to_cv(self, q_image):
        width = q_image.width()
        height = q_image.height()

        buffer = q_image.bits().asstring(width * height * 4)
        image = np.frombuffer(buffer, dtype=np.uint8).reshape((height, width, 4))

        return cv2.cvtColor(image, cv2.COLOR_BGRA2BGR)


if __name__ == '__main__':
    app = QApplication([])
    window = ImageProcessorApp()
    window.setGeometry(100, 100, 800, 600)
    window.show()
    app.exec_()
```

MORE EXAMPLES ON DEMO PROGRAM WITH USER INTERFACE






Load Image
Process Image


Load Image
Process Image