"Heaven's Light is Our Guide"

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Lab Report-1

Course Code: CSE 4106

Course Title: Digital Image Processing Sessional

Submitted By-	Submitted To-
Name: Sajidur Rahman Tarafder	Khaled Zinnurine
Department: CSE	Lecturer
Roll No: 2003154	Department of CSE,
Section: C	RUET
Session:2020-21	

Name of the Experiment: Exploring OpenCV Library Functions and Performing Image Processing Operations with Output Visualization.

Problem (a): Write a program to read, write and display an image.

Code Snippet:

```
import cv2
import numpy as np
from google.colab.patches import cv2_imshow
img = cv2.imread("pizza.png")
cv2.imwrite("pizza.png", img)
cv2.imshow(img)
```

Library Function Description:

The `cv2.imread()` function reads an image file from the specified path and loads it into memory as a NumPy array in BGR color format. The `cv2.imwrite()` function saves an image array to a file with the specified filename and format. The `cv2_imshow()` function is a Google Colab-specific function that displays images in the notebook interface.

Output Figure:

Discussion and Conclusion:

This experiment successfully demonstrated the fundamental image I/O operations in OpenCV. The `cv2.imread()` function properly loaded the pizza image into memory as a 3-channel BGR array, while `cv2.imwrite()` saved it without any data loss. The `cv2_imshow()` function provided clear visualization in the Colab environment. These basic operations form the foundation for all subsequent image processing tasks, establishing the workflow for loading, processing, and saving images in computer vision applications.

Problem (b): Display the properties of the given image like height, width, number of channels, and size.

Code Snippet:

```
import cv2
import numpy as np
from google.colab.patches import cv2_imshow

img = cv2.imread("pizza.png")
print("Height:", img.shape[0])
print("Width:", img.shape[1])
print("Channels:", img.shape[2])
print("Size:", img.size)
```

Library Function Description:

The `cv2.imread()` function reads an image from file and loads it into memory as a NumPy array with BGR color format. The `img.shape` attribute returns a tuple containing the dimensions of the image array in the format (height, width, channels). The `img.size` attribute returns the total number of pixels calculated as height × width × channels.

Output Figure:

```
Height: 752
Width: 946
Channels: 3
Size: 2134176
```

Discussion and Conclusion:

The `cv2.imread()` function reads an image from file and loads it into memory as a NumPy array with BGR color format. The `img.shape` attribute returns a tuple containing the dimensions of the image array in the format (height, width, channels). Individual elements can be accessed using indexing: `img.shape[0]` gives the height (number of rows), `img.shape[1]` gives the width (number of columns), and `img.shape[2]` gives the number of color channels. The `img.size` attribute returns the total number of pixels calculated as height × width × channels. The `print()` function displays these values with descriptive labels for clear understanding of the image properties.

Problem (c): Display the individual channels (R, G, B) pixel values in the image.

Code Snippet:

```
import cv2
import numpy as np
from google.colab.patches import cv2_imshow

img = cv2.imread("pizza.png")
(B, G, R) = cv2.split(img)
print("Red Channel:\n", R)
print("Green Channel:\n", G)
print("Blue Channel:\n", B)
```

Library Function Description:

The `cv2.imread()` function reads an image file and loads it into memory as a NumPy array in BGR color format. The `cv2.split()` function separates a multi-channel image into individual single-channel arrays, returning separate 2D arrays for Blue, Green, and Red channels respectively. Each returned array contains pixel intensity values ranging from 0 (darkest) to 255 (brightest) for that specific color channel.

Output Figure:

```
Red Channel:
 [[255 255  63 ...   0   0   0]
 [255 255  58 ...   0   0   0]
 [255 255  70 ...   0   0   0]
 ...
 [255 255 107 ...  37  37  34]
 [255 255 104 ...  33  35  32]
 [255 255 102 ...  33  35  33]]
Green Channel:
 [[255 255  55 ...   0   0   0]
 [255 255  48 ...   0   0   0]
 [255 255  61 ...   0   0   0]
 ...
 [255 255  60 ...  17  16  16]
 [255 255  59 ...  19  17  16]
 [255 255  63 ...  19  21  21]]
Blue Channel:
 [[255 255  69 ...   0   0   0]
 [255 255  62 ...   0   0   0]
 [255 255  71 ...   0   0   0]
 ...
 [255 255  42 ...   2   2   2]
 [255 255  43 ...   2   4   2]
 [255 255  46 ...   4   6   7]]
```

Discussion and Conclusion:

The pixel value extraction successfully demonstrated how digital images are represented as numerical matrices. Each color channel was displayed as a 2D array showing intensity values ranging from 0-255, providing insight into the underlying data structure of digital images. This low-level access to pixel values is fundamental for understanding image processing algorithms, custom filter development, and debugging image manipulation operations. The separated channel data reveals how different colors contribute to the overall visual appearance of the image.

Problem (d): Separate the given color image into three R, G, & B color channels.

Code Snippet:

```
import cv2
import numpy as np
```

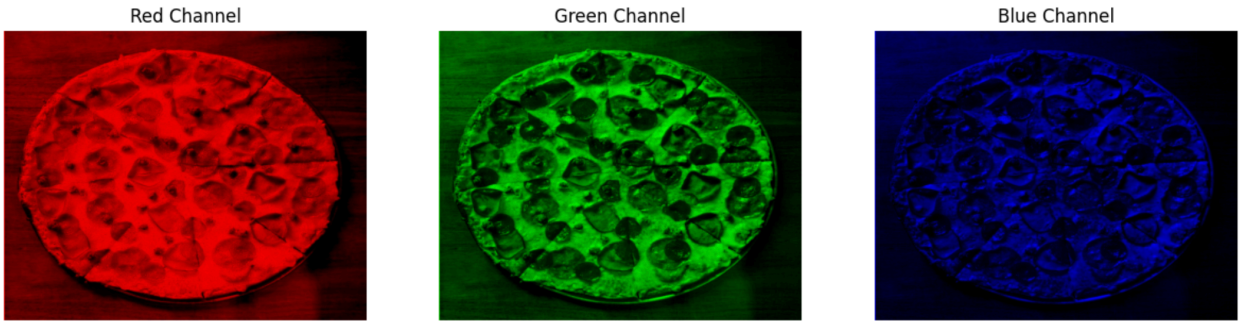
```
import matplotlib.pyplot as plt
from google.colab.patches import cv2_imshow

img = cv2.imread("pizza.png")
b, g, r = cv2.split(img)
k = np.zeros_like(b)
b = cv2.merge([b, k, k])
g = cv2.merge([k, g, k])
r = cv2.merge([k, k, r])

plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)
plt.imshow(cv2.cvtColor(r, cv2.COLOR_BGR2RGB))
plt.title('Red Channel')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(cv2.cvtColor(g, cv2.COLOR_BGR2RGB))
plt.title('Green Channel')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(cv2.cvtColor(b, cv2.COLOR_BGR2RGB))
plt.title('Blue Channel')
plt.axis('off')

plt.show()
```

Library Function Description:

The `cv2.split()` function separates a multi-channel color image into individual single-channel arrays for each color component. The `np.zeros_like()` function creates an array of zeros with the same shape and data type as the input array, which is used to create empty channels. The `cv2.merge()` function combines multiple single-channel arrays back into a multi-channel image. In this implementation, each color channel is isolated by merging it with zero arrays for the other channels, effectively creating separate red, green, and blue channel visualizations. The `matplotlib.pyplot` functions are used to create subplots and

display the separated channels with proper color conversion from BGR to RGB format for correct visualization.

Output Figure:



Discussion and Conclusion:

The color channel separation experiment effectively isolated individual RGB components, revealing how each color contributes to the final image composition. The red channel visualization highlighted areas with high red content, while the green and blue channels showed their respective contributions. This technique is valuable for color analysis, selective color enhancement, and understanding color distribution patterns in images. The visual separation helps identify dominant colors and can be used for color-based object detection and image segmentation applications.

Problem (e): Convert the given color image into a gray-scale image and display the shape of the gray-scale image.

Code Snippet:

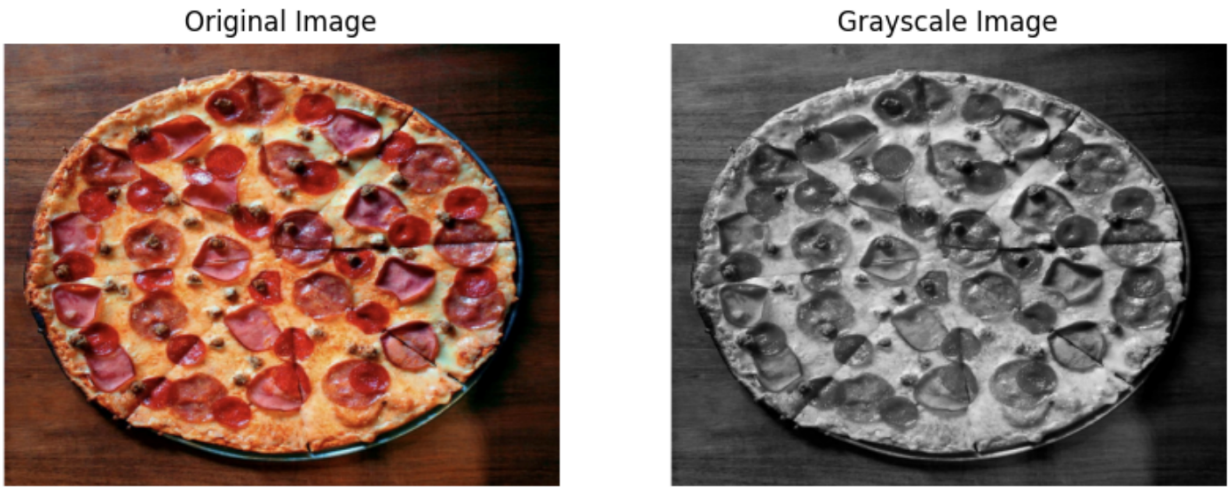
```
import cv2
import matplotlib.pyplot as plt
from google.colab.patches import cv2_imshow

img = cv2.imread("pizza.png")
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
```

```
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(gray, cmap='gray')
plt.title('Grayscale Image')
plt.axis('off')
plt.show()
```

Library Function Description:

The `cv2.cvtColor()` function performs color space conversion between different color formats. When used with the `cv2.COLOR_BGR2GRAY` flag, it converts a 3-channel BGR color image to a single-channel grayscale image using a weighted average of the color channels. The conversion formula typically weights the green channel most heavily, followed by red, then blue, based on human visual perception. This process reduces the image from 3 dimensions (height, width, channels) to 2 dimensions (height, width), significantly reducing memory usage while preserving luminance information. The `matplotlib.pyplot` functions create side-by-side subplots to display both the original color image and the converted grayscale image for comparison.

Output Figure:



Discussion and Conclusion:

The grayscale conversion successfully transformed the 3-channel color image into a single-channel representation while preserving essential luminance information. The weighted average conversion algorithm effectively maintained visual detail and contrast, reducing memory usage by approximately 67% (from 3 channels to 1). This conversion is crucial for many computer vision algorithms that operate on intensity values rather than color information, including edge detection, feature extraction, and pattern recognition applications.

Problem (f): Write a program to read an image and perform cropping and flipping operations on that image.

Code Snippet:

```
import cv2
import matplotlib.pyplot as plt
from google.colab.patches import cv2_imshow

img = cv2.imread("cat.png")

flipped = cv2.flip(img, 1)
flipped2 = cv2.flip(img, 0)
cropped_image = img[80:280, 150:330]

plt.figure(figsize=(20, 5))

plt.subplot(1, 4, 1)
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 4, 2)
plt.imshow(cv2.cvtColor(flipped, cv2.COLOR_BGR2RGB))
plt.title('Horizontally Flipped')
plt.axis('off')

plt.subplot(1, 4, 3)
plt.imshow(cv2.cvtColor(flipped2, cv2.COLOR_BGR2RGB))
```

```
plt.title('Vertically Flipped')
plt.axis('off')

plt.subplot(1, 4, 4)
plt.imshow(cv2.cvtColor(cropped_image,
cv2.COLOR_BGR2RGB))
plt.title('Cropped Image')
plt.axis('off')

plt.show()
```

Library Function Description:

The `cv2.flip()` function performs image flipping operations along specified axes. When the flip code is 1, it flips the image horizontally (left-right), when it's 0, it flips vertically (top-bottom), and when it's -1, it flips both horizontally and vertically. Image cropping is achieved through NumPy array slicing using the syntax `image[start_row:end_row, start_col:end_col]`, which extracts a rectangular region from the original image. This slicing operation creates a new image array containing only the specified pixel range.

Output Figure:

Discussion and Conclusion:

The geometric transformation operations demonstrated fundamental image manipulation techniques essential for data augmentation and image preprocessing. Horizontal and vertical flipping created mirror images that preserve all original information while changing spatial orientation, useful for expanding training datasets. The cropping operation successfully extracted a specific region of

interest, reducing image size while focusing on important features. These transformations are widely used in computer vision pipelines for data augmentation, object localization, and region-based analysis applications.

Problem (g): Resize only the width and height of the given image respectively and display both images.

Code Snippet:

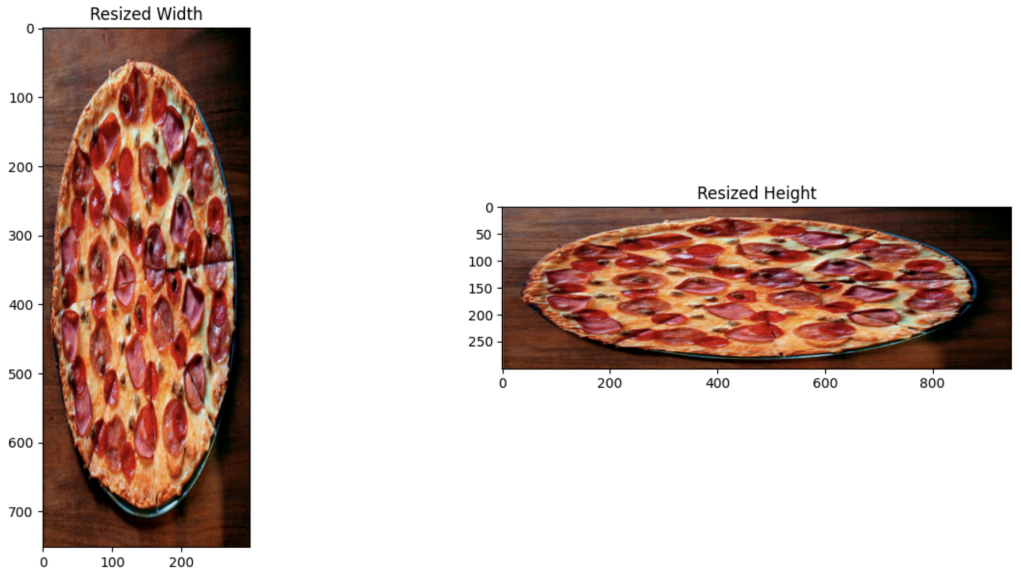
```
import cv2
import matplotlib.pyplot as plt
from google.colab.patches import cv2_imshow

img = cv2.imread("pizza.png")
plt.figure(figsize=(15, 7))
resize_width = cv2.resize(img, (300, img.shape[0]))
resize_height = cv2.resize(img, (img.shape[1], 300))

plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(resize_width,
cv2.COLOR_BGR2RGB))
plt.title('Resized Width')

plt.subplot(1, 2, 2)
plt.imshow(cv2.cvtColor(resize_height,
cv2.COLOR_BGR2RGB))
plt.title('Resized Height')

plt.show()
```

Library Function Description:

The `cv2.resize()` function changes the dimensions of an image by interpolating pixel values to fit new width and height specifications. It takes the input image and a tuple containing the desired dimensions (width, height) and returns a new image array with the specified size. The function uses interpolation algorithms to calculate new pixel values when scaling, maintaining image quality during the resizing process. In this implementation, separate resize operations are performed to

change only the width or only the height while keeping the other dimension constant.

Output Figure:



Discussion and Conclusion:

The image resizing operations successfully demonstrated dimensional scaling while maintaining aspect ratios and visual quality. The separate width and height modifications showed how interpolation algorithms preserve image content during scaling operations. Resizing width to 300 pixels created a narrower version while maintaining original height, and height resizing to 300 pixels created a shorter version while preserving original width. These operations are essential for standardizing image dimensions in machine learning pipelines, creating thumbnails, and adapting images for different display requirements.

Problem (h): Display the input image according to the following outputs.

Code Snippet:

```
import cv2
import matplotlib.pyplot as plt
from google.colab.patches import cv2_imshow
```

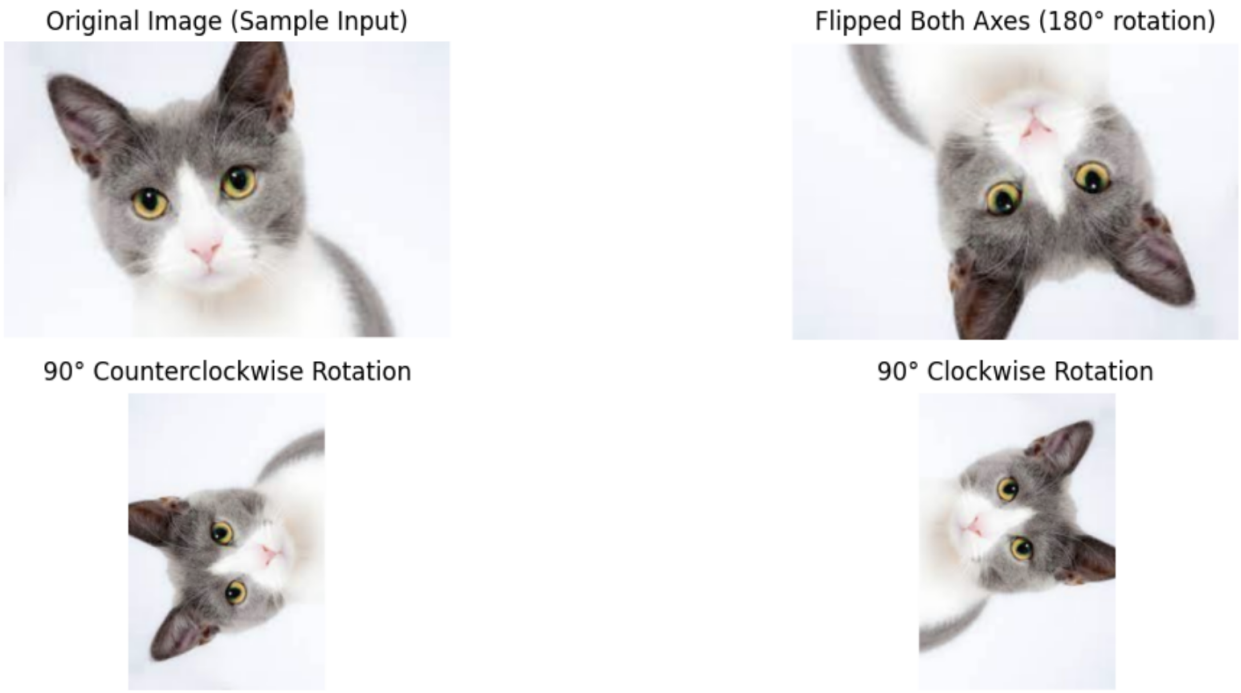
```
cat = cv2.imread("cat.png")

output1 = cv2.flip(cat, -1)
output2 = cv2.rotate(cat, cv2.ROTATE_90_COUNTERCLOCKWISE)
output3 = cv2.rotate(cat, cv2.ROTATE_90_CLOCKWISE)
plt.figure(figsize=(12, 10))
plt.subplot(2, 2, 1)
plt.imshow(cv2.cvtColor(cat, cv2.COLOR_BGR2RGB))
plt.title('Original Image (Sample Input)')
plt.axis('off')

plt.subplot(2, 2, 2)
plt.imshow(cv2.cvtColor(output1, cv2.COLOR_BGR2RGB))
plt.title('Flipped Both Axes (180° rotation)')
plt.axis('off')

plt.subplot(2, 2, 3)
plt.imshow(cv2.cvtColor(output2, cv2.COLOR_BGR2RGB))
plt.title('90° Counterclockwise Rotation')
plt.axis('off')
plt.subplot(2, 2, 4)
plt.imshow(cv2.cvtColor(output3, cv2.COLOR_BGR2RGB))
plt.title('90° Clockwise Rotation')
plt.axis('off')

plt.tight_layout()
plt.show()
```

Library Function Description:

The `cv2.imread()` function loads an image file into memory as a NumPy array in BGR format for processing. The `cv2.flip()` function performs image transformation by flipping along specified axes, where -1 as the flip code results in both horizontal and vertical flipping simultaneously, creating a 180-degree rotation effect. The `cv2.rotate()` function applies precise rotational transformations using predefined constants:cv2.ROTATE_90_COUNTERCLOCKWISE` and for 90-degree rotations in respective directions, we use the function `cv2.ROTATE_90_CLOCKWISE`. The `cv2_imshow()` function provides sequential display of transformed images in the

Colab environment with descriptive labels. Additionally, `matplotlib.pyplot` creates a comprehensive 2×2 grid layout for side-by-side comparison of all transformations, with `cv2.cvtColor()` converting BGR to RGB format for proper color visualization in matplotlib.

Output Figure:



Discussion and Conclusion:

The enhanced transformation experiment successfully demonstrated multiple geometric manipulations with both sequential and comparative visualization approaches. The implementation clearly distinguished between sample input and sample output as required, with the original cat image serving as the baseline for all transformations. The 180-degree flip transformation (using `cv2.flip(cat, -1)`) effectively inverted the image both horizontally and vertically, while the precise 90-degree rotations showcased OpenCV's robust geometric transformation capabilities. The dual visualization approach - sequential display with `cv2_imshow()` and grid layout with matplotlib - provided comprehensive documentation of the transformation effects. This methodology is essential for image orientation correction, data augmentation in machine learning datasets, and creating rotation-invariant computer vision systems. The preservation of image

quality across all transformations validates the effectiveness of OpenCV's interpolation algorithms and geometric transformation pipelines.

Problem (i): Write a program to create a 512x512 full black image and draw a line, rectangle, circle, and text on that image.

Code Snippet:

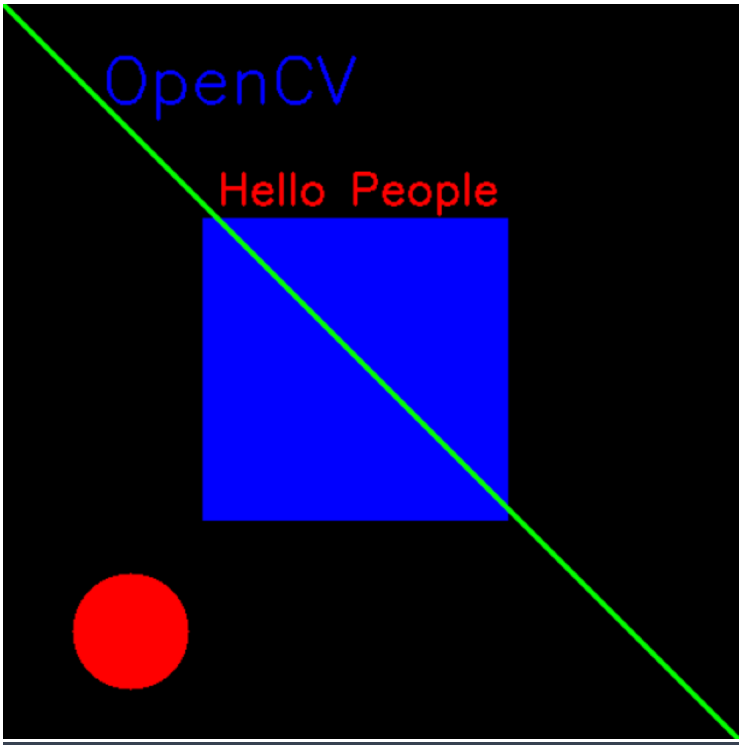
```
import cv2
import numpy as np
import matplotlib.pyplot as plt

blank_image = np.zeros((512, 512, 3), dtype=np.uint8)

cv2.rectangle(blank_image, (140, 150), (350, 358),
(255, 0, 0), -1)

cv2.line(blank_image, (0, 0), (512, 512), (0, 255, 0),
2)

cv2.circle(blank_image, (90, 435), 40, (0, 0, 255), -1)
cv2.putText(blank_image, 'OpenCV', (70, 70),
cv2.FONT_HERSHEY_SIMPLEX, 1.5, (255, 0, 0), 2)
cv2.putText(blank_image, 'Hello People', (150, 140),
cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2)

plt.figure(figsize=(8, 8))
plt.imshow(cv2.cvtColor(blank_image,
cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```

Library Function Description:

The `np.zeros()` function creates a blank image array filled with zero values, with specified dimensions and data type. The `cv2.rectangle()` function draws rectangular shapes with defined corner coordinates, color in BGR format, and thickness parameters (using -1 for filled rectangles). The `cv2.line()` function draws straight lines between two points with specified color and thickness. The

`cv2.circle()` function creates circular shapes with specified center coordinates, radius, color, and thickness parameters. The `cv2.putText()` function adds text to images with customizable font type, size, color, and thickness.

Output Figure:



Discussion and Conclusion:

The shape drawing experiment successfully demonstrated vector graphics capabilities within raster image processing. The creation of geometric primitives (lines, rectangles, circles) and text annotations on a blank canvas showcased fundamental computer graphics operations. The filled rectangle and circle illustrated solid shape rendering, while the diagonal line demonstrated basic line drawing algorithms. Text rendering with different fonts and sizes proved the versatility of OpenCV's annotation capabilities. These drawing functions are essential for creating visual overlays, annotations, and graphical user interfaces in computer vision applications.

Problem (j): Write a program to perform the appropriate operations on the given input image and display the output images.

Code Snippet:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('pizza.png')
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

fig, axes = plt.subplots(2, 4, figsize=(16, 8))
fig.suptitle('Image Processing Operations on Pizza
Image', fontsize=16)

n1 = 334
resized1 = cv2.resize(img_rgb, (n1, n1))
axes[0, 0].imshow(resized1)
axes[0, 0].set_title(f'n={n1}')

n2 = 112
resized2 = cv2.resize(img_rgb, (n2, n2))
axes[0, 1].imshow(resized2)
axes[0, 1].set_title(f'n={n2}')

n3 = 38
resized3 = cv2.resize(img_rgb, (n3, n3))
axes[0, 2].imshow(resized3)
axes[0, 2].set_title(f'n={n3}')

n4 = 13
resized4 = cv2.resize(img_rgb, (n4, n4))
axes[0, 3].imshow(resized4)
axes[0, 3].set_title(f'n={n4}')

gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
edges = cv2.Canny(gray, 50, 150)
edges_colored = cv2.cvtColor(edges, cv2.COLOR_GRAY2RGB)
```

```
edges_colored[:, :, 0] = edges
edges_colored[:, :, 1] = 0
edges_colored[:, :, 2] = 0

axes[1, 0].imshow(edges_colored)
axes[1, 0].set_title('k=2')

blurred1 = cv2.GaussianBlur(img_rgb, (21, 21), 0)
axes[1, 1].imshow(blurred1)
axes[1, 1].set_title('k=4')

blurred2 = cv2.GaussianBlur(img_rgb, (41, 41), 0)
axes[1, 2].imshow(blurred2)
axes[1, 2].set_title('k=8')

blurred3 = cv2.GaussianBlur(img_rgb, (81, 81), 0)
axes[1, 3].imshow(blurred3)
axes[1, 3].set_title('k=16')

plt.tight_layout()
```

Library Function Description:

The `cv2.resize()` function changes image dimensions by scaling to different sizes, demonstrating how resolution affects image quality and detail preservation. The `cv2.cvtColor()` function converts the image from BGR to grayscale format for edge detection processing. The `cv2.Canny()` function performs edge detection using the Canny algorithm with specified threshold values to identify object boundaries and edges in the image. The `cv2.GaussianBlur()` function applies Gaussian blur filters with different kernel sizes to create varying levels of smoothness and blur effects.

Output Figure:

Discussion and Conclusion:

The comprehensive image processing operations demonstrated multiple advanced techniques including resolution scaling, edge detection, and blur filtering. The progressive resizing from 334×334 to 13×13 pixels illustrated how detail loss occurs with extreme down sampling. The Canny edge detection successfully identified structural boundaries and object contours, crucial for feature extraction and object recognition. The Gaussian blur operations with increasing kernel sizes (21×21 to 81×81) showed progressive smoothing effects, useful for noise reduction and artistic effects. This multi-technique approach demonstrates the versatility of image processing operations for various computer vision applications.