Subject Code:21CSL66

Subject: COMPUTER GRAPHICS AND IMAGE PROCESSING LABORATORY

Program-01

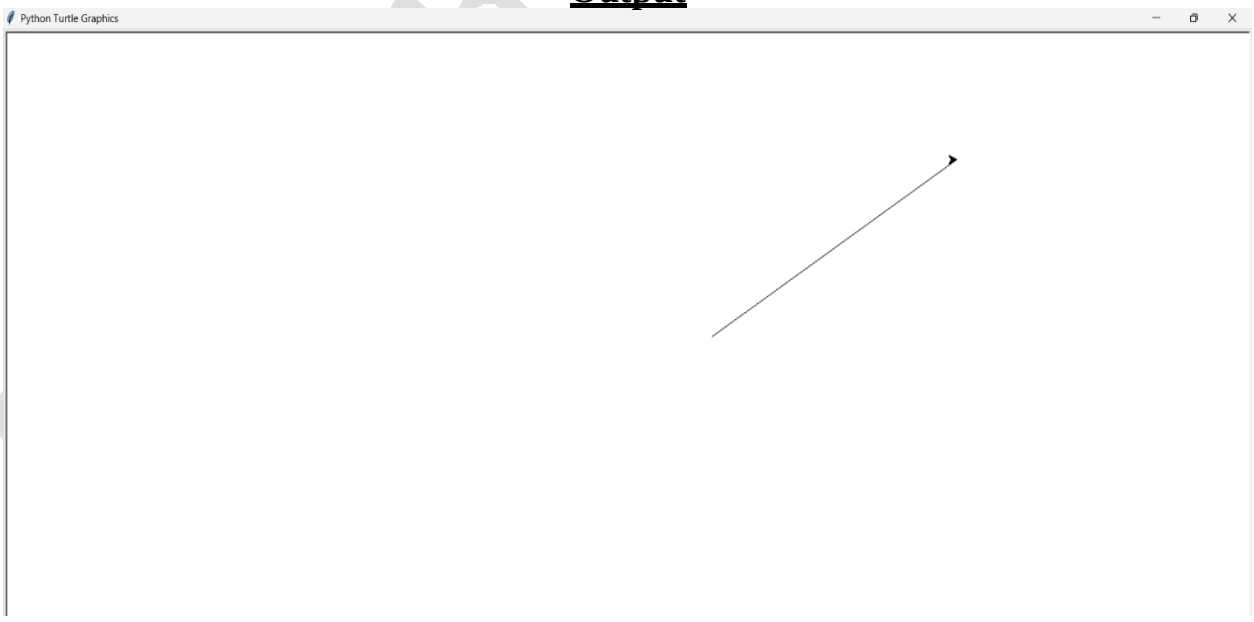
1. Develop a program to draw a line using Bresenham's line drawing technique

Program

```
import turtle

def bresenham_line(x1, y1, x2, y2):
    # Calculate the deltas
    dx = abs(x2 - x1)
    dy = abs(y2 - y1)

    # Determine the step direction for each axis
    x_step = 1 if x1 < x2 else -1
    y_step = 1 if y1 < y2 else -1

    # Initialize the error term
    error = 2 * dy - dx

    # Initialize the line points
    line_points = []

    # Start at the first point
    x, y = x1, y1

    # Draw the line
    for _ in range(dx + 1):
        # Add the current point to the line
        line_points.append((x, y))

        # Update the error term and adjust the coordinates
        if error > 0:
            y += y_step
            error -= 2 * dx
```

```
        error += 2 * dy
        x += x_step

    return line_points

# Example usage
turtle.setup(500, 500)
turtle.speed(0)  # Fastest drawing speed

x1, y1 = 100, 100
x2, y2 = 400, 300

line_points = bresenham_line(x1, y1, x2, y2)

# Draw the line
turtle.penup()
turtle.goto(x1, y1)
turtle.pendown()
for x, y in line_points:
    turtle.goto(x, y)

turtle.exitonclick()
```

Output



Explaination:

1. The bresenham_line function takes four arguments: x1, y1, x2, and y2, which represent the starting and ending points of the line segment.
2. Inside the function, it calculates the deltas (dx and dy) between the starting and ending points, and determines the step direction (x_step and y_step) for each axis.
3. It initializes the error term (error) and an empty list (line_points) to store the coordinates of the line points.
4. The function then enters a loop that iterates dx + 1 times, where in each iteration:
5. It appends the current point (x, y) to the line_points list.
6. It updates the error term (error) and adjusts the coordinates of x and y based on the Bresenham's line algorithm.
7. After the loop, the function returns the line_points list containing the coordinates of the line points.
8. In the example usage section, the code sets up a turtle graphics window, defines the starting and ending points of the line segment (x1, y1, x2, y2), calls the bresenham_line function to get the line points, and then draws the line segment by moving the turtle to each point in the line_points list.
9. The turtle.exitonclick() function keeps the graphics window open until the user clicks on it, allowing the user to view the drawn line.

2. Develop a program to demonstrate basic geometric operations on the 2D object

Program

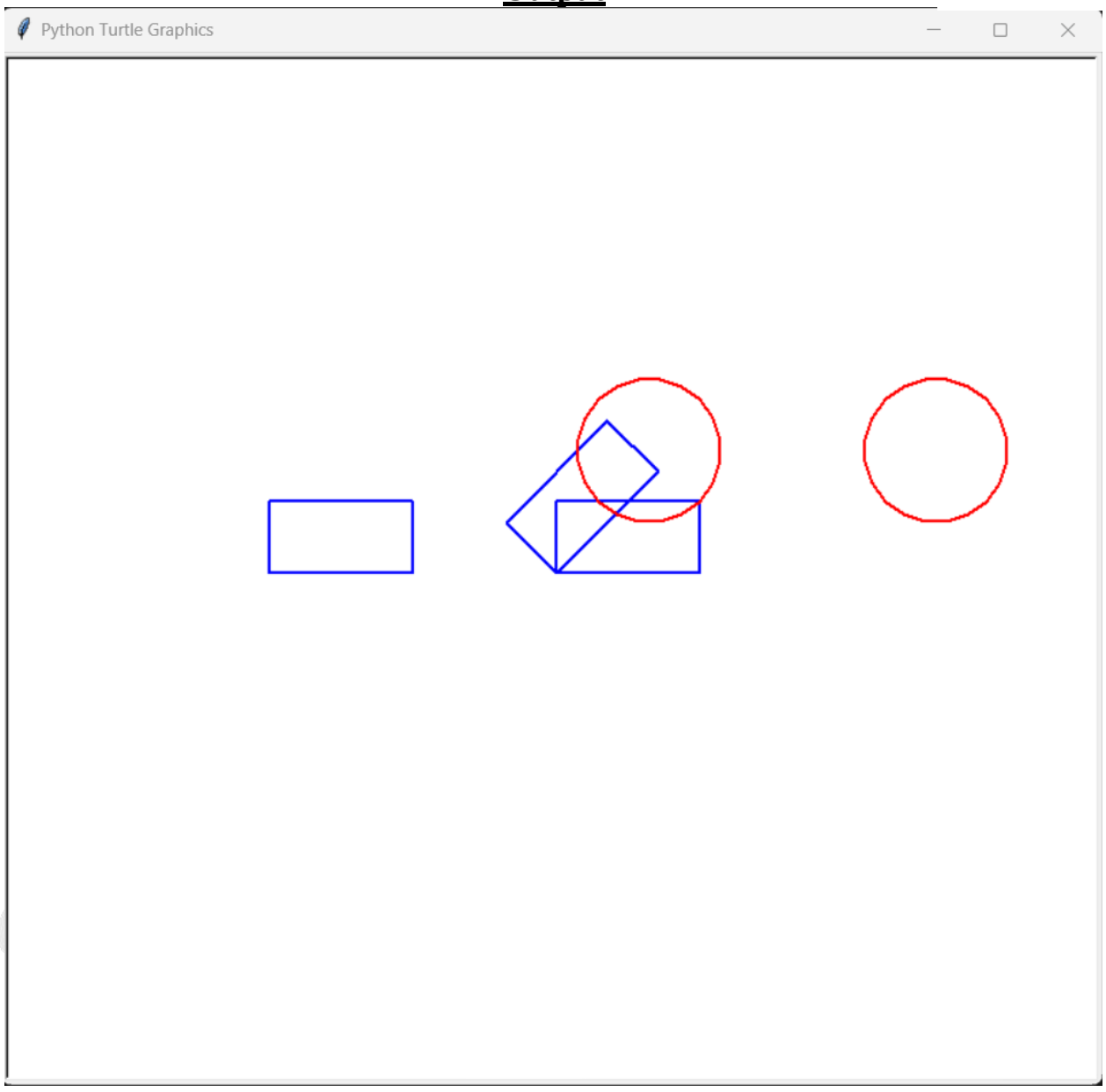
```
import turtle
import math

# Set up the turtle screen
screen = turtle.Screen()
screen.bgcolor("white")

# Create a turtle instance
t = turtle.Turtle()
t.speed(1)  # Set the drawing speed (1 is
t.pensize(2)  # Set the pen size

# Define a function to draw a rectangle
def draw_rectangle(x, y, width, height, color):
    t.penup()
    t.goto(x, y)
    t.pendown()
    t.color(color)
    for _ in range(2):
        t.forward(width)
        t.left(90)
        t.forward(height)
        t.left(90)

# Define a function to draw a circle
```

```
def draw_circle(x, y, radius, color):
    t.penup()
    t.goto(x, y - radius)
    t.pendown()
    t.color(color)
    t.circle(radius)

# Define a function to translate a 2D object
def translate(x, y, dx, dy):
    t.penup()
    t.goto(x + dx, y + dy)
    t.pendown()

# Define a function to rotate a 2D object
def rotate(x, y, angle):
    t.penup()
    t.goto(x, y)
    t.setheading(angle)
    t.pendown()

# Define a function to scale a 2D object
def scale(x, y, sx, sy):
    t.penup()
    t.goto(x * sx, y * sy)
    t.pendown()

# Draw a rectangle
```

```
draw_rectangle(-200, 0, 100, 50, "blue")

# Translate the rectangle
translate(-200, 0, 200, 0)
draw_rectangle(0, 0, 100, 50, "blue")

# Rotate the rectangle
rotate(0, 0, 45)
draw_rectangle(0, 0, 100, 50, "blue")

# Scale the rectangle
scale(0, 0, 2, 2)
draw_rectangle(0, 0, 100, 50, "blue")

# Draw a circle
draw_circle(100, 100, 50, "red")

# Translate the circle
translate(100, 100, 200, 0)
draw_circle(300, 100, 50, "red")

# Rotate the circle
rotate(300, 100, 45)
draw_circle(300, 100, 50, "red")

# Scale the circle
scale(300, 100, 2, 2)
```

```
draw_circle(600, 200, 50, "red")

# Keep the window open until it's closed
turtle.done()
```

Output



Explanation

1. The code starts by importing the necessary modules: turtle for drawing graphics and math for mathematical operations.
2. A turtle screen is set up with a white background color.
3. A turtle instance t is created, and its speed and pen size are set.
4. Two helper functions draw_rectangle and draw_circle is defined to draw rectangles and circles, respectively. These functions take the coordinates, dimensions (width, height, or radius), and color as arguments.
5. Three transformation functions are defined: translate, rotate, and scale. These functions take the coordinates and transformation parameters (translation distances, rotation angle, or scaling factors) as arguments and move the turtle's position and orientation accordingly.
6. The code then demonstrates the use of these functions by drawing and transforming a rectangle and a circle.
    - A rectangle is drawn at (-200, 0) with a width of 100 and a height of 50 in blue color.
    - The rectangle is translated 200 units to the right, and a new rectangle is drawn at (0, 0).
    - The rectangle is rotated by 45 degrees, and a new rectangle is drawn.
    - The rectangle is scaled by a factor of 2 in both dimensions, and a new rectangle is drawn.
    - A circle is drawn at (100, 100) with a radius of 50 in red color.
    - The circle is translated 200 units to the right, and a new circle is drawn at (300, 100).
    - The circle is rotated by 45 degrees, and a new circle is drawn.
    - The circle is scaled by a factor of 2 in both dimensions, and a new circle is drawn at (600, 200).
7. Finally, the turtle. done () function is called to keep the window open until it's closed by the user.

3. Develop a program to demonstrate basic geometric operations on the 3D object

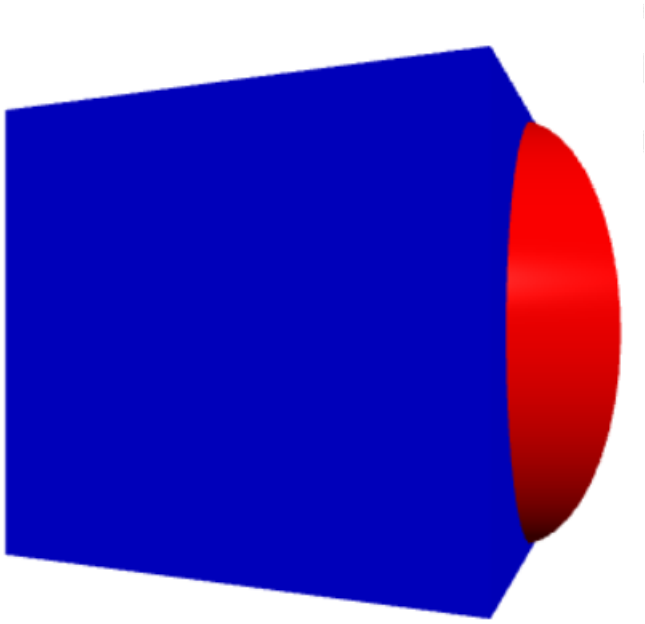
Program

```
from vpython import canvas, box, cylinder, vector, color, rate

# Create a 3D canvas
scene = canvas(width=800, height=600, background=color.white)

# Define a function to draw a cuboid
def draw_cuboid(pos, length, width, height, color):
    cuboid = box(pos=vector(*pos), length=length, width=width, height=height, color=color)
    return cuboid

# Define a function to draw a cylinder
def draw_cylinder(pos, radius, height, color):
    cyl = cylinder(pos=vector(*pos), radius=radius, height=height, color=color)
    return cyl

# Define a function to translate a 3D object
def translate(obj, dx, dy, dz):
    obj.pos += vector(dx, dy, dz)

# Define a function to rotate a 3D object
def rotate(obj, angle, axis):
    obj.rotate(angle=angle, axis=vector(*axis))
```

```
# Define a function to scale a 3D object
def scale(obj, sx, sy, sz):
    obj.size = vector(obj.size.x * sx, obj.size.y * sy, obj.size.z * sz)

# Draw a cuboid
cuboid = draw_cuboid((-2, 0, 0), 2, 2, 2,

# Translate the cuboid
translate(cuboid, 4, 0, 0)

# Rotate the cuboid
rotate(cuboid, angle=45, axis=(0, 1, 0))

# Scale the cuboid
scale(cuboid, 1.5, 1.5, 1.5)

# Draw a cylinder
cylinder = draw_cylinder((2, 2, 0), 1, 10, color.red)

# Translate the cylinder
translate(cylinder, 0, -2, 0)

# Rotate the cylinder
rotate(cylinder, angle=30, axis=(1, 0, 0))

# Scale the cylinder
scale(cylinder, 1.5, 1.5, 1.5)
```

```
# Keep the 3D scene interactive
while True:
    rate(30)  # Set the frame rate to 30 frames per second
```

Output



Explainations

1. Imports the necessary modules from the vpython library.
2. Creates a 3D canvas with a white background and dimensions of 800x600 pixels.
3. Defines a function draw_cuboid that creates a 3D cuboid (box) at a specified position, with given length, width, height, and color. It returns the created cuboid object.
4. Defines a function draw_cylinder that creates a 3D cylinder at a specified position, with given radius, height, and color. It returns the created cylinder object.
5. Defines a function translate that translates a 3D object by given dx, dy, and dz values along the x, y, and z axes, respectively.
6. Defines a function rotate that rotates a 3D object by a given angle around a specified axis.
7. Defines a function scale that scales a 3D object by given sx, sy, and sz scale factors along the x, y, and z axes, respectively.
8. Creates a blue cuboid at position (-2, 0, 0) with dimensions 2x2x2.
9. Translates the cuboid by (4, 0, 0) using the translate function.
10. Rotates the cuboid by 45 degrees around the y-axis using the rotate function.
11. Scales the cuboid by a factor of 1.5 along all axes using the scale function.
12. Creates a red cylinder at position (2, 2, 0) with radius 1 and height 10.
13. Translates the cylinder by (0, -2, 0) using the translate function.
14. Rotates the cylinder by 30 degrees around the x-axis using the rotate function.
15. Scales the cylinder by a factor of 1.5 along all axes using the scale function.
16. Enters an infinite loop to keep the 3D scene interactive, with a frame rate of 30 frames per second, using the rate function.

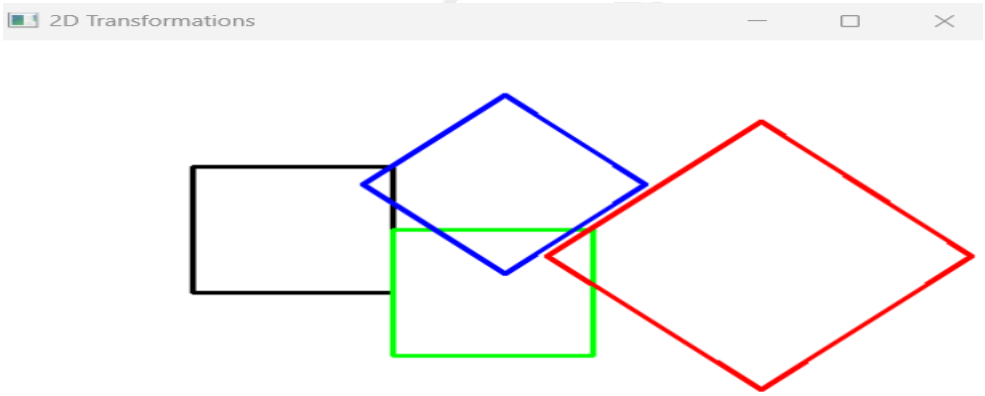
4. Develop a program to demonstrate 2D transformation on basic objects

Program

```
import cv2
import numpy as np

# Define the dimensions of the canvas
canvas_width = 500
canvas_height = 500

# Create a blank canvas
canvas = np.ones((canvas_height, canvas_width, 3), dtype=np.uint8) * 255

# Define the initial object (a square)
obj_points = np.array([[100, 100], [200, 100], [200, 200], [100, 200]], dtype=np.int32)

# Define the transformation matrices
translation_matrix = np.float32([[1, 0, 100], [0, 1, 50]])
rotation_matrix = cv2.getRotationMatrix2D((150, 150), 45, 1)
scaling_matrix = np.float32([[1.5, 0, 0], [0, 1.5, 0]])

# Apply transformations
translated_obj = np.array([np.dot(translation_matrix, [x, y, 1])[:2] for x, y in obj_points],
dtype=np.int32)
rotated_obj = np.array([np.dot(rotation_matrix, [x, y, 1])[:2] for x, y in translated_obj],
dtype=np.int32)
scaled_obj = np.array([np.dot(scaling_matrix, [x, y, 1])[:2] for x, y in rotated_obj],
dtype=np.int32)
```

```
# Draw the objects on the canvas
cv2.polylines(canvas, [obj_points], True, (0, 0, 0), 2)
cv2.polylines(canvas, [translated_obj], True, (0, 255, 0), 2)
cv2.polylines(canvas, [rotated_obj], True, (255, 0, 0), 2)
cv2.polylines(canvas, [scaled_obj], True, (0, 0, 255), 2)

# Display the canvas
cv2.imshow("2D Transformations", canvas)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output



Explainations

1. The code starts by importing the necessary libraries: cv2 for OpenCV and numpy for numerical operations.
2. It defines the dimensions of the canvas (canvas_width and canvas_height) and creates a blank white canvas using NumPy.
3. The initial object (a square) is defined as an array of four points (obj_points) representing the vertices of the square.
4. The transformation matrices are defined:
    - translation_matrix: A 2x3 matrix for translation.
    - rotation_matrix: A rotation matrix obtained using cv2.getRotationMatrix2D for rotating around a specified center point by a given angle.
    - scaling_matrix: A 2x3 matrix for scaling.
5. The transformations are applied to the initial object by performing matrix multiplication with the transformation matrices:
    - translated_obj: The object is translated by applying the translation_matrix.
    - rotated_obj: The translated object is rotated by applying the rotation_matrix.
    - scaled_obj: The rotated object is scaled by applying the scaling_matrix.
6. The original object and the transformed objects (translated, rotated, and scaled) are drawn on the canvas using cv2.polylines.
7. The canvas with the drawn objects is displayed using cv2.imshow, and the code waits for a key press (cv2.waitKey(0)) before closing the window.
8. Finally, all windows are closed using cv2.destroyAllWindows().
9. The resulting output is a window displaying the following:

- The original square (black)
- The translated square (green)
- The rotated square (red)
- The scaled square (blue)

5. Develop a program to demonstrate 3D transformation on 3D objects

Program

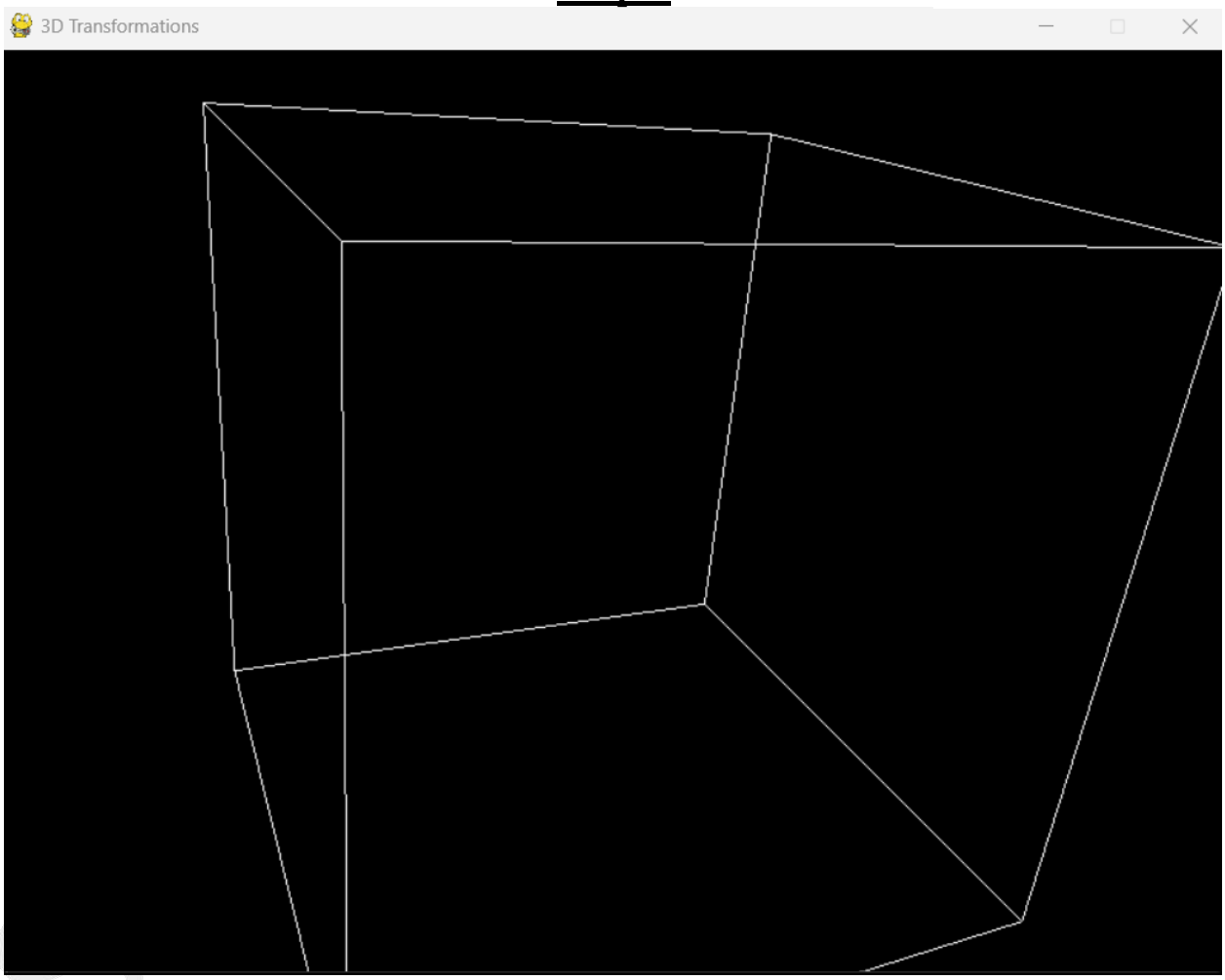
```
import pygame
from pygame.locals import *
from OpenGL.GL import *
from OpenGL.GLU import *
import numpy as np

# Initialize Pygame
pygame.init()

# Set up the display
display_width = 800
display_height = 600
display = pygame.display.set_mode((display_width, display_height), DOUBLEBUF |
OPENGL)
pygame.display.set_caption("3D Transformations")

# Set up  OpenGL
glClearColor(0.0, 0.0, 0.0, 1.0)
glEnable(GL_DEPTH_TEST)
glMatrixMode(GL_PROJECTION)
gluPerspective(45, (display_width / display_height), 0.1, 50.0)
glMatrixMode(GL_MODELVIEW)

# Define the 3D object (a cube)
vertices = np.array([
```

```
    [-1, -1, -1],
    [1, -1, -1],
    [1, 1, -1],
    [-1, 1, -1],
    [-1, -1, 1],
    [1, -1, 1],
    [1, 1, 1],
    [-1, 1, 1]
], dtype=np.float32)

edges = np.array([
    [0, 1], [1, 2], [2, 3], [3, 0],
    [4, 5], [5, 6], [6, 7], [7, 4],
    [0, 4], [1, 5], [2, 6], [3, 7]
], dtype=np.uint32)

# Set up the transformation matrices
translation_matrix = np.eye(4, dtype=np.float32)
translation_matrix[3, :3] = [0, 0, -5]

rotation_matrix = np.eye(4, dtype=np.float32)

scaling_matrix = np.eye(4, dtype=np.float32)
scaling_matrix[0, 0] = 1.5
scaling_matrix[1, 1] = 1.5
scaling_matrix[2, 2] = 1.5
```

```
# Main loop
running = True
angle = 0
while running:
    for event in  pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    # Clear the display
    glClear(GL_COLOR_BUFFER_BIT  |  GL_DEPTH_BUFFER_BIT)

    # Apply transformations
    glLoadIdentity()
    glMultMatrixf(translation_matrix)
    glRotatef(angle, 1, 1, 0)
    glMultMatrixf(rotation_matrix)
    glMultMatrixf(scaling_matrix)

    # Draw the 3D object
    glBegin(GL_LINES)
    for edge in edges:
        for vertex in edge:
            glVertex3fv(vertices[vertex])
    glEnd()

    # Update the rotation angle
    angle += 1
```

```
    # Swap the front and back buffers
    pygame.display.flip()

# Quit Pygame
pygame.quit()
```

Output



Explanation

1. Imports the necessary modules from pygame, OpenGL.GL, OpenGL.GLU, and numpy.
2. Initializes Pygame and sets up the display with a width of 800 pixels and a height of 600 pixels.
3. Sets up OpenGL by clearing the color buffer, enabling depth testing, setting up the projection matrix using gluPerspective, and switching to the ModelView matrix mode.
4. Defines the vertices of a 3D cube using a NumPy array.
5. Defines the edges of the cube as pairs of vertex indices using a NumPy array.
6. Sets up the transformation matrices for translation, rotation, and scaling:
7. translation_matrix translates the object along the negative z-axis by 5 units.
8. rotation_matrix is initially set to the identity matrix (no rotation).

scaling_matrix scales the object by a factor of 1.5 along all axes.

10. Enters the main loop, which runs until the user closes the window.
11. Inside the main loop:
    - Handles the Pygame event queue, checking for the QUIT event to exit the loop.
    - Clears the color and depth buffers using glClear.
    - Applies the transformations:
    - Loads the identity matrix using glLoadIdentity.
    - Applies the translation matrix using glMultMatrixf.
    - Rotates the object around the vector (1, 1, 0) by an angle that increases with each iteration.
    - Applies the rotation matrix using glMultMatrixf.
    - Applies the scaling matrix using glMultMatrixf.

- Draws the 3D cube by iterating over the edges and vertices, using glBegin(GL_LINES) and glVertex3fv.
- Increments the rotation angle for the next iteration.
- Swaps the front and back buffers using pygame.display.flip() to display the rendered scene.

12.After the main loop ends, the code quits Pygame.

6. Develop a program to demonstrate Animation effects on simple objects.

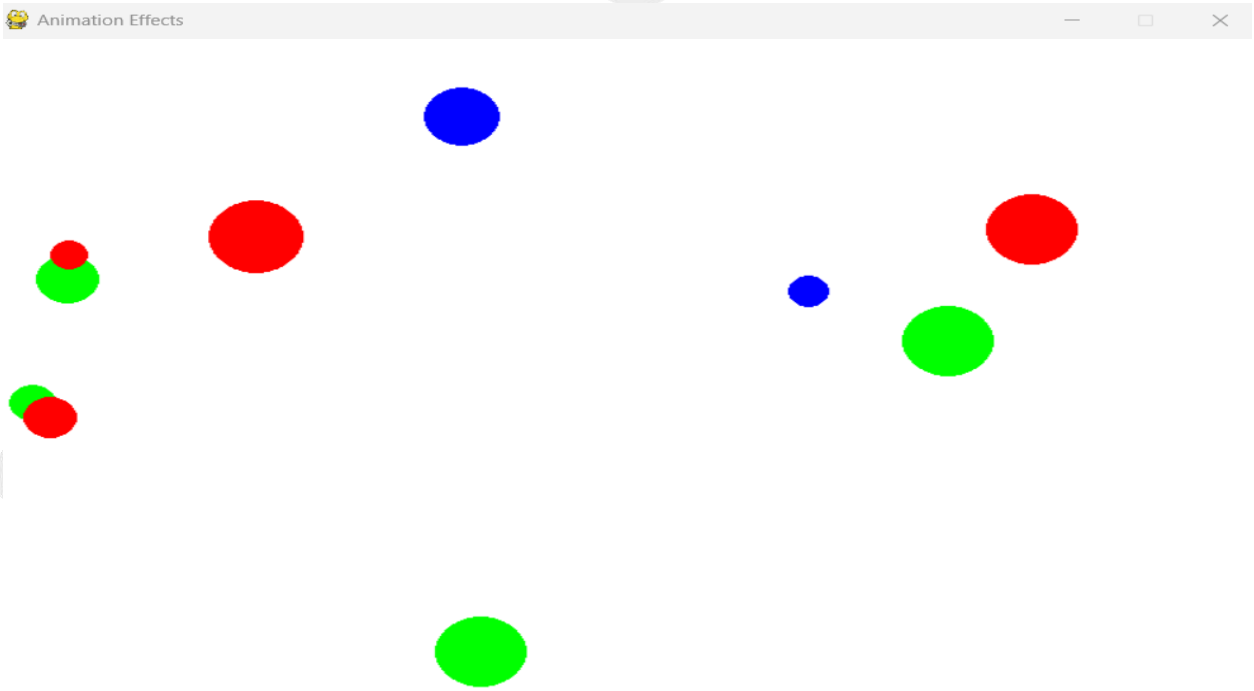
Program

```
import pygame
import random

# Initialize Pygame
pygame.init()

# Set up the display
screen_width = 800
screen_height = 600
screen = pygame.display.set_mode((screen_width, screen_height))
pygame.display.set_caption("Animation Effects")

# Define colors
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)

# Define object properties
num_objects = 10
objects = []
for _ in range(num_objects):
    x = random.randint(50, screen_width - 50)
    y = random.randint(50, screen_height - 50)
```

```
    radius = random.randint(10, 30)
    color = random.choice([RED, GREEN, BLUE])
    speed_x = random.randint(-5, 5)
    speed_y = random.randint(-5, 5)
    objects.append({"x": x, "y": y, "radius": radius, "color": color, "speed_x": speed_x, "speed_y": speed_y})

# Main loop
running = True
clock = pygame.time.Clock()
while running:
    # Handle events
    for event in  pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    # Clear the screen
    screen.fill(WHITE)

    # Update and draw objects
    for obj in objects:
        # Move the object
        obj["x"] += obj["speed_x"]
        obj["y"] += obj["speed_y"]

        # Bounce off the edges
        if obj["x"] - obj["radius"] < 0 or obj["x"] + obj["radius"] > screen_width:
```

```
        obj["speed_x"] = -obj["speed_x"]
    if obj["y"] - obj["radius"] < 0 or obj["y"] + obj["radius"] > screen_height:
        obj["speed_y"] = -obj["speed_y"]

    # Draw the object
    pygame.draw.circle(screen, obj["color"], (obj["x"], obj["y"]), obj["radius"])

# Update the display
pygame.display.flip()
clock.tick(60)  # Limit the frame rate to 60 FPS

# Quit Pygame
pygame.quit()
```

Output



Explanation

1. Imports the required modules: pygame for creating the graphical window and handling events, and random for generating random values.
2. Initializes Pygame and sets up a window with a width of 800 pixels and a height of 600 pixels.
3. Defines some colors (BLACK, WHITE, RED, GREEN, BLUE) as RGB tuples.
4. Initializes a list called objects to store the properties of each circle object. The properties include the x and y coordinates, radius, color, and velocities (speed_x and speed_y).
5. Generates num_objects (set to 10) with random positions, radii, colors, and velocities, and appends them to the objects list.
6. Enters the main loop, which runs until the user closes the window.
7. Inside the main loop:
8. Handles the Pygame event queue, checking for the QUIT event to exit the loop.
9. Clears the screen by filling it with the WHITE color.
10. Iterates over each object in the objects list:
11. Updates the x and y coordinates of the object based on its velocities.
12. Checks if the object has collided with the edges of the screen. If so, it reverses the corresponding velocity component (x or y) to make the object bounce off the edge.
13. Draws the object (circle) on the screen using pygame.draw.circle with the object's color, position, and radius.
14. Updates the display using pygame.display.flip().
15. Limits the frame rate to 60 frames per second (FPS) using clock.tick(60).
16. After the main loop ends, the code quits Pygame.

7. Write a Program to read a digital image. Split and display image into 4 quadrants, up, down, right and left.

Program

```
import cv2
import numpy as np

# Load the image
image_path = "image.jpg"  # Replace with the path to your image
img = cv2.imread(image_path)

# Get the height and width of the image
height, width, _ = img.shape

# Split the image into four quadrants
up_left = img[0:height//2, 0:width//2]
up_right = img[0:height//2, width//2:width]
down_left = img[height//2:height, 0:width//2]
down_right = img[height//2:height, width//2:width]

# Create a blank canvas to display the quadrants
canvas = np.zeros((height, width, 3), dtype=np.uint8)

# Place the quadrants on the canvas
canvas[0:height//2, 0:width//2] = up_left
canvas[0:height//2, width//2:width] = up_right
canvas[height//2:height, 0:width//2] = down_left
```

```
canvas[height//2:height, width//2:width] = down_right

# Display the canvas
cv2.imshow("Image Quadrants", canvas)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output

Explanation

1. The necessary libraries, cv2 (OpenCV) and numpy, are imported.
2. The path to the input image file is specified (image_path). In this case, it's set to "image/atc.jpg", assuming the image file named "atc.jpg" is located in a directory named "image" relative to the script's location.
3. The image is loaded using cv2.imread().
4. The height and width of the image are obtained from the shape attribute of the image.
5. The image is split into four quadrants using NumPy array slicing:
6. up_left: Top-left quadrant
7. up_right: Top-right quadrant
8. down_left: Bottom-left quadrant
9. down_right: Bottom-right quadrant
10. A blank canvas with the same dimensions as the original image is created using np.zeros(). This canvas will be used to display the four quadrants.
11. The four quadrants are placed on the canvas using NumPy array assignment.
12. The canvas containing the four quadrants is displayed using cv2.imshow().
13. The script waits for a key press (cv2.waitKey(0)) before closing the window.
14. Finally, all windows are closed using cv2.destroyAllWindows().

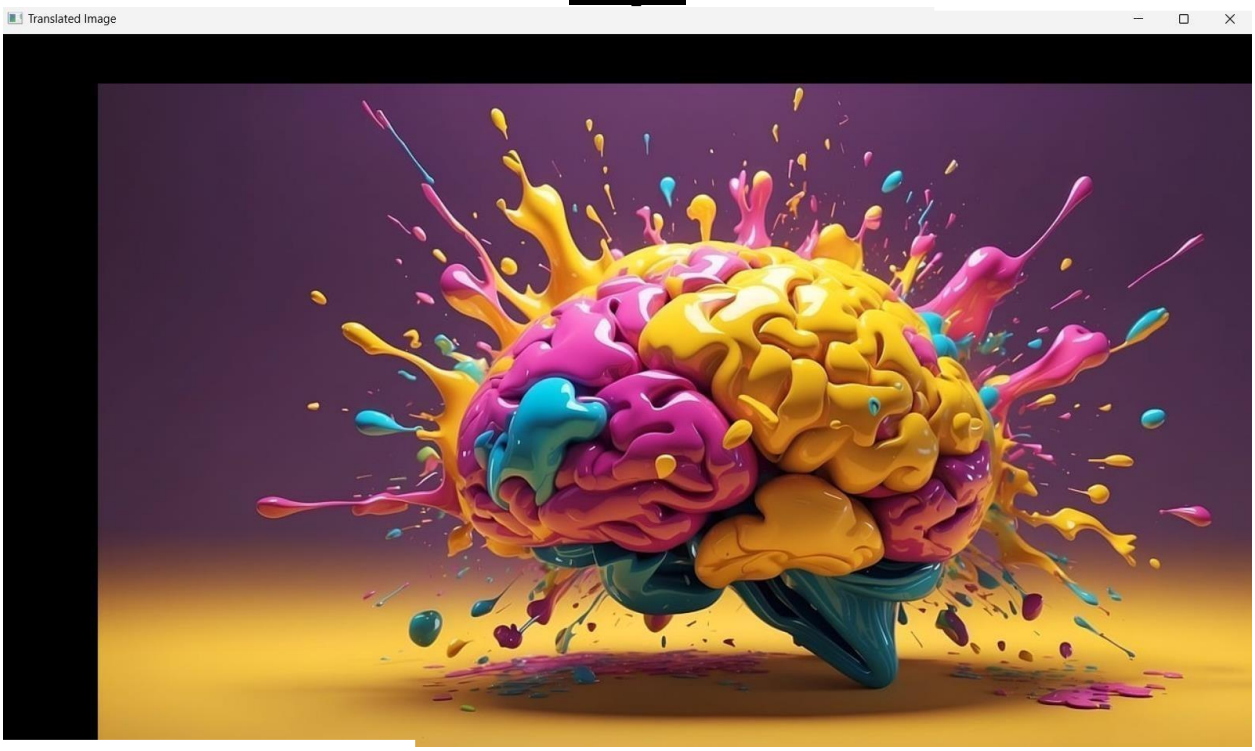
8. Write a program to show rotation, scaling, and translation on an image

Program

```
import cv2
import numpy as np

# Load the image
image_path = "image.jpg"  # Replace with the path to your image
img = cv2.imread(image_path)

# Get the image dimensions
height, width, _ = img.shape

# Define the transformation matrices
rotation_matrix = cv2.getRotationMatrix2D((width/2, height/2), 45, 1)  # Rotate by 45 degrees
scaling_matrix = np.float32([[1.5, 0, 0], [0, 1.5, 0]])  # Scale by 1.5x
translation_matrix = np.float32([[1, 0, 100], [0, 1, 50]])  # Translate by (100, 50)

# Apply transformations
rotated_img = cv2.warpAffine(img, rotation_matrix, (width, height))
scaled_img = cv2.warpAffine(img, scaling_matrix, (int(width*1.5), int(height*1.5)))
translated_img = cv2.warpAffine(img, translation_matrix, (width, height))

# Display the original and transformed images
cv2.imshow("Original Image", img)
cv2.imshow("Rotated Image", rotated_img)
cv2.imshow("Scaled Image", scaled_img)
```

```
cv2.imshow("Translated Image", translated_img)

# Wait for a key press and then close all windows
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output




Scaled Image


Rotated Image


Original Image

Explanation

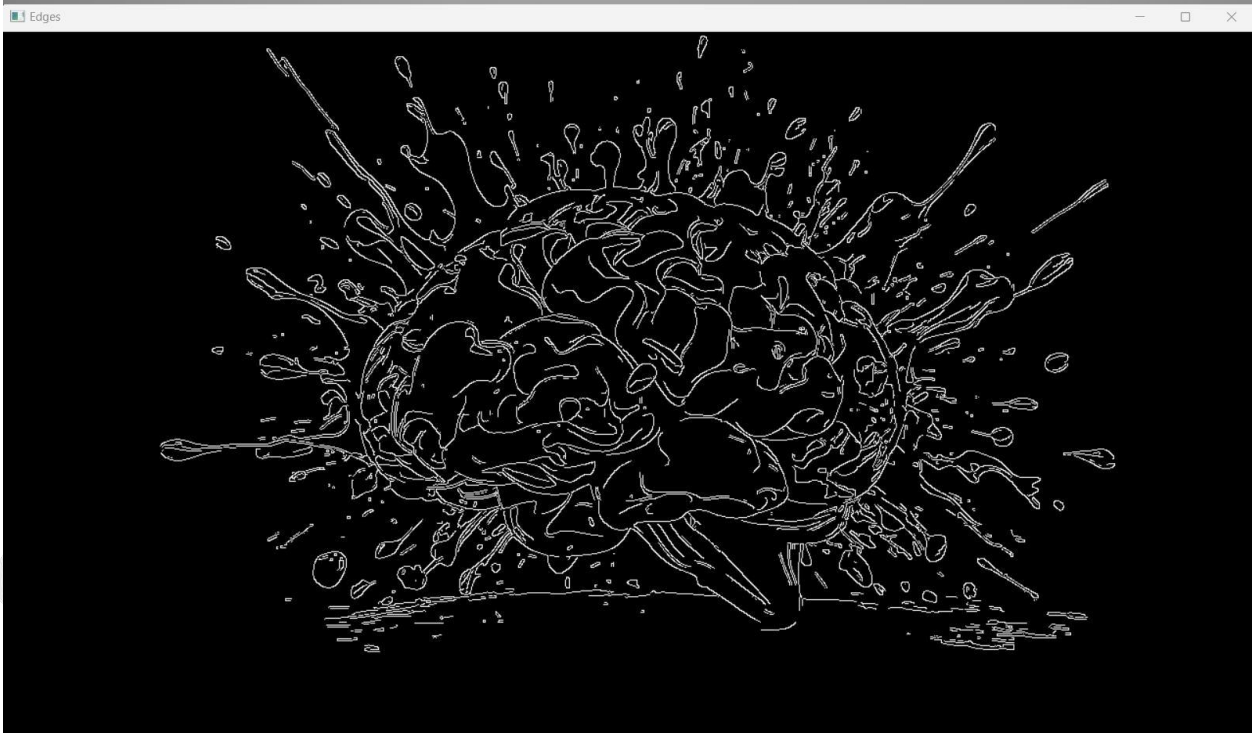
1. The necessary libraries, cv2 (OpenCV) and numpy, are imported.
2. The path to the input image file is specified (image_path). In this case, it's set to "image/atc.jpg", assuming the image file named "atc.jpg" is located in a directory named "image" relative to the script's location.
3. The image is loaded using cv2.imread().
4. The height and width of the image are obtained from the shape attribute of the image.
5. The transformation matrices for rotation, scaling, and translation are defined:
6. rotation_matrix: Obtained using cv2.getRotationMatrix2D() to rotate the image by 45 degrees around its center.
7. scaling_matrix: A 2x3 NumPy matrix to scale the image by a factor of 1.5 along both axes.
8. translation_matrix: A 2x3 NumPy matrix to translate the image by (100, 50) pixels.
9. The transformations are applied to the original image using cv2.warpAffine():
10. rotated_img: The image is rotated using the rotation_matrix.
11. scaled_img: The image is scaled using the scaling_matrix.
12. translated_img: The image is translated using the translation_matrix.
13. The original image and the transformed images (rotated, scaled, and translated) are displayed using cv2.imshow().
14. The script waits for a key press (cv2.waitKey(0)) before closing the windows.
15. Finally, all windows are closed using cv2.destroyAllWindows().

9. Read an image and extract and display low-level features such as edges, textures using filtering techniques.

Program

```
import cv2
import numpy as np

# Load the image
image_path = "image/atc.jpg"  # Replace with the path to your image
img = cv2.imread(image_path)

# Convert the image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Edge detection
edges = cv2.Canny(gray, 100, 200)  # Use Canny edge detector

# Texture extraction
kernel = np.ones((5, 5), np.float32) / 25  # Define a 5x5 averaging kernel
texture = cv2.filter2D(gray, -1, kernel)  # Apply the averaging filter for texture extraction

# Display the original image, edges, and texture
cv2.imshow("Original Image", img)
cv2.imshow("Edges", edges)
cv2.imshow("Texture", texture)

# Wait for a key press and then close all windows
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output




Original Image

Explanation

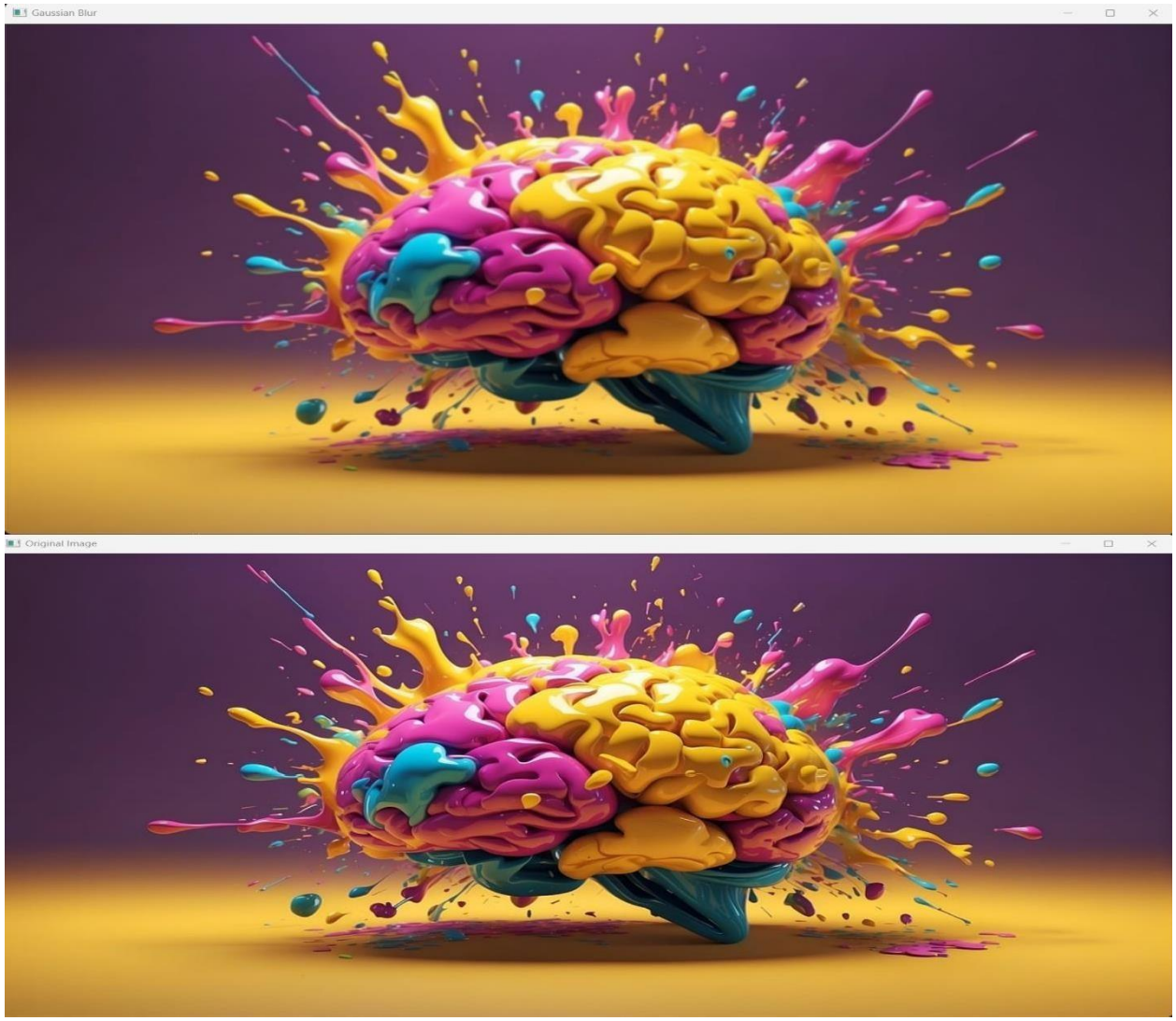
1. The necessary libraries, cv2 (OpenCV) and numpy, are imported.
2. The path to the input image file is specified (image_path). In this case, it's set to "image/atc.jpg", assuming the image file named "atc.jpg" is located in a directory named "image" relative to the script's location.
3. The image is loaded using cv2.imread().
4. The image is converted to grayscale using cv2.cvtColor(img, cv2.COLOR_BGR2GRAY). This step is necessary for edge detection and texture extraction, as these operations are typically performed on grayscale images.
5. Edge detection is performed on the grayscale image using the Canny edge detector (cv2.Canny(gray, 100, 200)). The Canny edge detector is a popular algorithm for edge detection, and the two arguments (100 and 200) are the lower and upper thresholds for hysteresis.
6. Texture extraction is performed using a simple averaging filter (cv2.filter2D(gray, -1, kernel)). A 5x5 averaging kernel (kernel = np.ones((5, 5), np.float32) / 25) is defined, where each element is set to 1/25 (the sum of the kernel elements is 1). This kernel is applied to the grayscale image using cv2.filter2D(), which performs a 2D convolution between the image and the kernel. The resulting image (texture) captures the texture information of the original image.
7. The original image (img), the detected edges (edges), and the extracted texture (texture) are displayed using cv2.imshow().
8. The script waits for a key press (cv2.waitKey(0)) before closing the windows.
9. Finally, all windows are closed using cv2.destroyAllWindows().

10. Write a program to blur and smoothing an image.

Program

```
import cv2

# Load the image
image = cv2.imread('image/atc.jpg')

# Gaussian Blur
gaussian_blur = cv2.GaussianBlur(image, (5, 5), 0)

# Median Blur
median_blur = cv2.medianBlur(image, 5)

# Bilateral Filter
bilateral_filter = cv2.bilateralFilter(image, 9, 75, 75)

# Display the original and processed images
cv2.imshow('Original Image', image)
cv2.imshow('Gaussian Blur', gaussian_blur)
cv2.imshow('Median Blur', median_blur)
cv2.imshow('Bilateral Filter', bilateral_filter)

# Wait for a key press to close the windows
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output


Gaussian Blur
Original Image

Explanation

- The cv2 library is imported from OpenCV.
- The image is loaded using cv2.imread('image/atc.jpg'). Make sure to replace'image/atc.jpg' with the correct path to your image file.
- Three different types of blurring/smoothing filters are applied to the image:
- Gaussian Blur: gaussian_blur = cv2.GaussianBlur(image, (5, 5), 0) applies a Gaussian blur filter to the image. The parameters (5, 5) specify the size of the Gaussian kernel, and 0 is the standard deviation value in the X and Y directions (which is automatically calculated from the kernel size).
- Median Blur: median_blur = cv2.medianBlur(image, 5) applies a median blur filter to the image. The parameter 5 specifies the size of the median filter kernel.
- Bilateral Filter: bilateral_filter = cv2.bilateralFilter(image, 9, 75, 75) applies a bilateral filter to the image. The parameters 9, 75, and 75 represent the diameter of the pixel neighborhood, the filter sigma in the color space, and the filter sigma in the coordinate space, respectively.
- The original image and the filtered images are displayed using cv2.imshow().
- The script waits for a key press (cv2.waitKey(0)) before closing the windows.
- Finally, all windows are closed using cv2.destroyAllWindows().

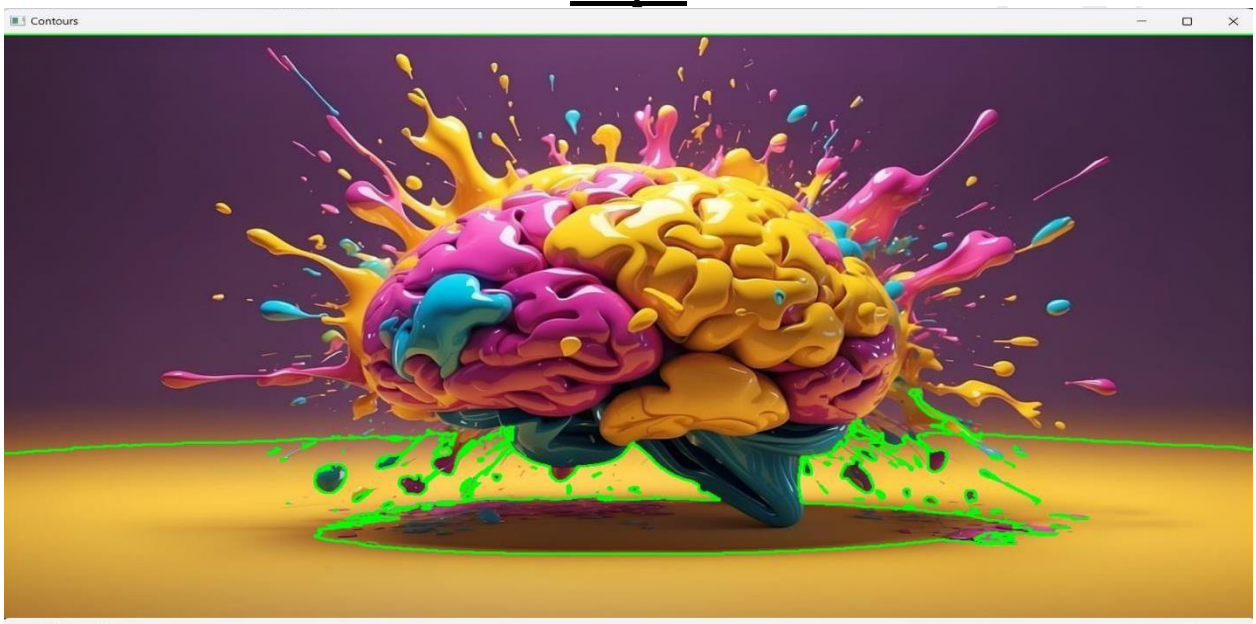
- **Write a program to contour an image.**

Program

```
import cv2
import numpy as np

# Load the image
image = cv2.imread('image/atc.jpg')

# Convert the image to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Apply binary thresholding
ret, thresh = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)

# Find contours
contours, hierarchy = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

# Create a copy of the original image to draw contours on
contour_image = image.copy()

# Draw contours on the image
cv2.drawContours(contour_image, contours, -1, (0, 255, 0), 2)

# Display the original and contour images
cv2.imshow('Original Image', image)
```

```
cv2.imshow('Contours', contour_image)

# Wait for a key press to close the windows
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output



Explanation

- The cv2 library is imported from OpenCV, and the numpy library is imported for numerical operations.
- The image is loaded using cv2.imread('image/atc.jpg'). Make sure to replace 'image/atc.jpg' with the correct path to your image file.
- The image is converted to grayscale using cv2.cvtColor(image, cv2.COLOR_BGR2GRAY). This step is necessary because contour detection is often performed on grayscale images.
- Binary thresholding is applied to the grayscale image using cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU). This operation converts the grayscale image into a binary image, where pixels are either black or white, based on a threshold value. The cv2.THRESH_OTSU flag automatically determines the optimal threshold value using Otsu's method. The cv2.THRESH_BINARY_INV flag inverts the binary image, so that foreground objects become white and the background becomes black.
- The contours are found in the binary image using cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE). The cv2.RETR_EXTERNAL flag retrieves only the extreme outer contours, and cv2.CHAIN_APPROX_SIMPLE compresses the contour data by approximating it with a simplified polygon.
- A copy of the original image is created using contour_image = image.copy(). This copy will be used to draw the contours on.
- The contours are drawn on the contour image using cv2.drawContours(contour_image, contours, -1, (0, 255, 0), 2).

The -1 argument indicates that all contours should be drawn, the (0, 255, 0) argument specifies the color (green in this case), and the 2 argument specifies the thickness of the contour lines.

- The original image and the contour image are displayed using cv2.imshow().

- The script waits for a key press (cv2.waitKey(0)) before closing the windows.

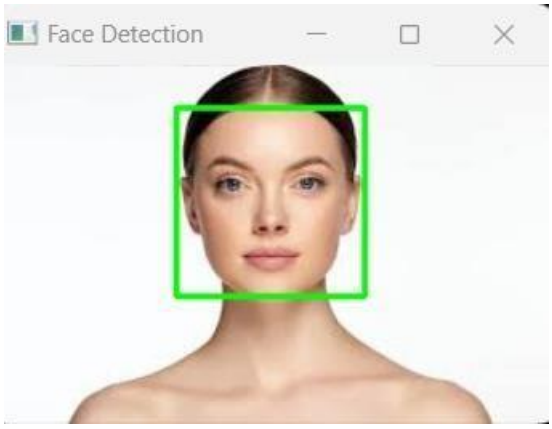
10.Finally, all windows are closed using cv2.destroyAllWindows().

- **Write a program to detect a face/s in an image.**

Program

```
import cv2

# Load the cascade classifier for face detection
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
'haarcascade_frontalface_default.xml')

# Load the image
image = cv2.imread('image/face.jpeg')

# Convert the image to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Detect faces in the grayscale image
faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5,
minSize=(30, 30))

# Draw rectangles around the detected faces
for (x, y, w, h) in faces:
    cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)
```

Output



Explanation

- This code demonstrates how to perform face detection in an image using OpenCV inPython. Here's a breakdown of what the code does:
- The cv2 library is imported from OpenCV.
- The Haar cascade classifier for face detection is loaded using cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml'). This classifier is a pre-trained model that candetect frontal faces in images.
- The image is loaded using cv2.imread('image/face.jpeg'). Make sure to replace 'image/face.jpeg' with the correct path to your image file containing faces.
- The image is converted to grayscale using cv2.cvtColor(image, cv2.COLOR_BGR2GRAY). Face detection is typically performed on grayscale images.
- The face_cascade.detectMultiScale method is used to detect faces in the grayscale image. The parameters scaleFactor=1.1, minNeighbors=5, and minSize=(30, 30) control the detection process:
- scaleFactor=1.1 specifies the scale factor used to resize the input image for differentscales.
- minNeighbors=5 specifies the minimum number of neighboring rectangles that should overlap to consider a face detection as valid.
- minSize=(30, 30) specifies the minimum size of the face to be detected.
- For each detected face, a rectangle is drawn around it using cv2.rectangle(image, (x,

y), (x + w, y + h), (0, 255, 0), 2). The rectangle coordinates are obtained from the faces list returned by detectMultiScale. The (0, 255, 0) argument specifies the color (green in this case), and the 2 argument specifies the thickness of the rectangle lines.

11. The image with the detected faces and rectangles is displayed using cv2.imshow('Face Detection', image).

12. The script waits for a key press (cv2.waitKey(0)) before closing the window.

13. Finally, the window is closed using cv2.destroyAllWindows().