**Practical 1 Drawing basic shapes**
**a. Draw a line using built in graphics fucntion**
**b. Draw a circle and ellipse with specified coordinates**
**c. Create a polygon with 5 sides**
**d. Draw a fill rectangle with color.**
**Code 1a.**

```
import cv2
import numpy as np

# Create a white background image
image = np.ones((400, 600, 3), dtype=np.uint8) * 255

# Draw a line
# Syntax: cv2.line(image, start_point, end_point, color, thickness)
cv2.line(image, (50, 50), (550, 350), (0, 0, 255), 3)  # Red line

# Display the image
cv2.imshow("Line Drawing", image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

**Code 1b**

```
import cv2
import numpy as np

# Create a blank white canvas
img = np.ones((500, 700, 3), dtype=np.uint8) * 255

# ---- Draw a Circle ----
# cv2.circle(image, center_coordinates, radius, color, thickness)
center_circle = (200, 250)   # x, y coordinates of center
radius = 100                 # radius in pixels
color_circle = (255, 0, 0)   # Blue in BGR
thickness_circle = 3         # thickness (use -1 for filled)
```

```
cv2.circle(img, center_circle, radius, color_circle, thickness_circle)

# ---- Draw an Ellipse ----
# cv2.ellipse(image, center_coordinates, axes_length, angle,
startAngle, endAngle, color, thickness)
center_ellipse = (500, 250)  # x, y coordinates of center
axes_length = (150, 100)     # major axis length, minor axis length
angle = 30                   # rotation of ellipse in degrees
start_angle = 0              # starting angle of arc
end_angle = 360              # ending angle of arc (360 = full ellipse)
color_ellipse = (255, 0, 255)  # Red in BGR
thickness_ellipse = 2        # thickness (use -1 for filled)

cv2.ellipse(img, center_ellipse, axes_length, angle, start_angle,
end_angle, color_ellipse, thickness_ellipse)

# Display the result
cv2.imshow("Circle and Ellipse", img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

**Code 1c**

```
import cv2
import numpy as np

# Create a blank white canvas
img = np.ones((500, 700, 3), dtype=np.uint8) * 255

# Define points of a pentagon (x, y) coordinates
# You can adjust values to change size and position
points = np.array([
    [350, 100],   # Top vertex
    [500, 200],
    [450, 350],
```

```
    [250, 350],
    [200, 200]
], np.int32)

# Reshape for OpenCV (each point as separate row)
points = points.reshape((-1, 1, 2))

# Draw polygon
# cv2.polylines(image, [points], isClosed, color, thickness)
cv2.polylines(img, [points], True, (0, 0, 255), 3)  # Red outline

# Optional: Fill the polygon
cv2.fillPoly(img, [points], (255, 255, 0))  # Yellow fill

# Display the image
cv2.imshow("Pentagon", img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

**Code 1d**

```
import cv2
import numpy as np

# Create a blank white canvas
img = np.ones((400, 600, 3), dtype=np.uint8) * 255

# Rectangle coordinates
start_point = (100, 100)   # Top-left corner (x, y)
end_point = (400, 300)     # Bottom-right corner (x, y)

# Color in BGR format
color = (255, 0, 255)  # Green

# Draw and fill rectangle
# thickness = -1 fills the shape
```

```
cv2.rectangle(img, start_point, end_point, color, thickness=-1)

# Display the image
cv2.imshow("Filled Rectangle", img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

**Practical 2 Line Drawing algorithm**
**a. Write code for DDA line drawing algorithm**
**b. Implement Bresenhams algorithm**
**c. Draw a circle using midpoint algorithm**
**d. Accept a radius and center from user and draw a circle.**

**Code 2a**

```
import matplotlib.pyplot as plt

def dda_line(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1

    if abs(dx) > abs(dy):
        steps = abs(dx)
    else:
        steps = abs(dy)

    x_increment = dx / steps
    y_increment = dy / steps

    x = x1
    y = y1

    x_coords = []
    y_coords = []
    for _ in range(steps + 1):
        x_coords.append(int(x))
```

```
        y_coords.append(int(y))
        x += x_increment
        y += y_increment

    return x_coords, y_coords

# Draw the line
x1, y1 = 10, 10
x2, y2 = 20, 20
x_coords, y_coords = dda_line(x1, y1, x2, y2)

# Plot the line
plt.figure(figsize=(8, 8))
plt.plot(x_coords, y_coords, 'bo-')
plt.plot(x1, y1, 'go')  # starting point
plt.plot(x2, y2, 'ro')  # ending point
plt.title("DDA Line Drawing")
plt.xlabel("X")
plt.ylabel("Y")
plt.grid(True)
plt.show()
```

**Code 2b**

```
def bresenham_line(x1, y1, x2, y2):
    dx = abs(x2 - x1)
    dy = abs(y2 - y1)
    sx = 1 if x2 > x1 else -1
    sy = 1 if y2 > y1 else -1
    err = dx - dy

    points = []
    while True:
        points.append((x1, y1))
        if x1 == x2 and y1 == y2:
            break
```

```
        e2 = 2 * err
        if e2 > -dy:
            err -= dy
            x1 += sx
        if e2 < dx:
            err += dx
            y1 += sy

    return points

# Draw the line
x1, y1 = 10, 10
x2, y2 = 20, 20
line_points = bresenham_line(x1, y1, x2, y2)

# Print the points
for point in line_points:
    print(point)

# Plot the line
import matplotlib.pyplot as plt
x_coords = [point[0] for point in line_points]
y_coords = [point[1] for point in line_points]
plt.plot(x_coords, y_coords, 'bo-')
plt.plot(x1, y1, 'go')  # starting point
plt.plot(x2, y2, 'ro')  # ending point
plt.show()
```

**Code 2c**

```
import matplotlib.pyplot as plt

def midpoint_circle(x0, y0, r):
    x = 0
    y = r
    d = 1 - r
```

```
    points = []
    while x <= y:
        points.append((x0 + x, y0 + y))
        points.append((x0 - x, y0 + y))
        points.append((x0 + x, y0 - y))
        points.append((x0 - x, y0 - y))
        points.append((x0 + y, y0 + x))
        points.append((x0 - y, y0 + x))
        points.append((x0 + y, y0 - x))
        points.append((x0 - y, y0 - x))

        if d < 0:
            d += 2 * x + 3
        else:
            d += 2 * (x - y) + 5
            y -= 1
        x += 1

    return points

# Draw the circle
x0, y0 = 20, 20
r = 10
circle_points = midpoint_circle(x0, y0, r)

# Plot the circle
x_coords = [point[0] for point in circle_points]
y_coords = [point[1] for point in circle_points]
plt.plot(x_coords, y_coords, 'bo')
plt.gca().set_aspect('equal')
plt.show()
```

**Code 2d**

```
import matplotlib.pyplot as plt
```

```
def midpoint_circle(x0, y0, r):
    x = 0
    y = r
    d = 1 - r

    points = []
    while x <= y:
        points.append((x0 + x, y0 + y))
        points.append((x0 - x, y0 + y))
        points.append((x0 + x, y0 - y))
        points.append((x0 - x, y0 - y))
        points.append((x0 + y, y0 + x))
        points.append((x0 - y, y0 + x))
        points.append((x0 + y, y0 - x))
        points.append((x0 - y, y0 - x))

        if d < 0:
            d += 2 * x + 3
        else:
            d += 2 * (x - y) + 5
            y -= 1
        x += 1

    return points

def main():
    # Get user input for radius and center
    x0 = int(input("Enter the x-coordinate of the center: "))
    y0 = int(input("Enter the y-coordinate of the center: "))
    r = int(input("Enter the radius of the circle: "))

    # Draw the circle
    circle_points = midpoint_circle(x0, y0, r)

    # Plot the circle
```

```
    x_coords = [point[0] for point in circle_points]
    y_coords = [point[1] for point in circle_points]
    plt.plot(x_coords, y_coords, 'bo')
    plt.plot(x0, y0, 'ro')  # center point
    plt.gca().set_aspect('equal')
    plt.show()

if __name__ == "__main__":
    main()
```

**Practical 3 2D transformation**

**a. Draw a triangle and translate its right by 50 units. Show original and translated image**

**b. Rotate a square 45 degrees about origin.Take user input for rotation angle and direction**

**c. Scale a rectangle by 2x in both x and y direction**

**d. Reflect a triangle across the x axis and y axis**

**Code 3a**

```
import matplotlib.pyplot as plt

def draw_triangle(x, y):
    points = [(x, y), (x + 50, y + 100), (x + 100, y)]
    return points

def translate(points, tx, ty):
    translated_points = [(x + tx, y + ty) for x, y in points]
    return translated_points

def main():
    # Draw the original triangle
    original_points = draw_triangle(50, 50)

    # Translate the triangle
    translated_points = translate(original_points, 50, 0)
```

```
    # Plot the original and translated triangles
    original_x = [point[0] for point in original_points]
    original_y = [point[1] for point in original_points]
    original_x.append(original_points[0][0])
    original_y.append(original_points[0][1])

    translated_x = [point[0] for point in translated_points]
    translated_y = [point[1] for point in translated_points]
    translated_x.append(translated_points[0][0])
    translated_y.append(translated_points[0][1])

    plt.plot(original_x, original_y, 'bo-')
    plt.plot(translated_x, translated_y, 'ro-')
    plt.gca().set_aspect('equal')
    plt.show()

if __name__ == "__main__":
    main()
```

**Code 3b**

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

def rotate(points, angle):
    angle_rad = np.deg2rad(angle)
    rotation_matrix = np.array([[np.cos(angle_rad), -
np.sin(angle_rad)],
                                [np.sin(angle_rad), np.cos(angle_rad)]])
    rotated_points = [np.dot(rotation_matrix, point) for point in
points]
    return rotated_points

def draw_square(x, y, size):
```

```
    points = np.array([[x, y], [x + size, y], [x + size, y + size], [x, y
+ size]])
    return points

def main():
    # Get user input for rotation angle and direction
    angle = float(input("Enter the rotation angle (in degrees): "))
    direction = input("Enter the direction of rotation
(clockwise/anticlockwise): ")
    if direction.lower() == "clockwise":
        angle = -angle

    # Draw the original square
    original_points = draw_square(-50, -50, 100)

    # Rotate the square
    rotated_points = rotate(original_points, angle)

    # Plot the original and rotated squares
    original_x = np.append(original_points[:, 0], original_points[0,
0])
    original_y = np.append(original_points[:, 1], original_points[0,
1])

    rotated_x = np.append([point[0] for point in rotated_points],
rotated_points[0][0])
    rotated_y = np.append([point[1] for point in rotated_points],
rotated_points[0][1])

    plt.plot(original_x, original_y, 'bo-')
    plt.plot(rotated_x, rotated_y, 'ro-')
    plt.gca().set_aspect('equal')
    plt.show()

if __name__ == "__main__":
```

```
    main()
```

**Code 3c**

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

def scale(points, sx, sy):
    scaled_points = [(point[0] * sx, point[1] * sy) for point in points]
    return scaled_points

def draw_rectangle(x, y, width, height):
    points = [(x, y), (x + width, y), (x + width, y + height), (x, y +
height)]
    return points

def main():
    # Draw the original rectangle
    original_points = draw_rectangle(10, 10, 50, 20)

    # Scale the rectangle
    scaled_points = scale(original_points, 2, 2)

    # Plot the original and scaled rectangles
    original_x = [point[0] for point in original_points]
    original_y = [point[1] for point in original_points]
    original_x.append(original_points[0][0])
    original_y.append(original_points[0][1])

    scaled_x = [point[0] for point in scaled_points]
    scaled_y = [point[1] for point in scaled_points]
    scaled_x.append(scaled_points[0][0])
    scaled_y.append(scaled_points[0][1])

    plt.plot(original_x, original_y, 'bo-')
    plt.plot(scaled_x, scaled_y, 'ro-')
```

```
    plt.gca().set_aspect('equal')
    plt.show()

if __name__ == "__main__":
    main()
```

**Code 3d**

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

def reflect_x(points):
    reflected_points = [(point[0], -point[1]) for point in points]
    return reflected_points

def reflect_y(points):
    reflected_points = [(-point[0], point[1]) for point in points]
    return reflected_points

def draw_triangle(x, y, size):
    points = [(x, y), (x + size, y), (x + size / 2, y + size)]
    return points

def main():
    # Draw the original triangle
    original_points = draw_triangle(10, 10, 50)

    # Reflect the triangle across x-axis
    reflected_x_points = reflect_x(original_points)

    # Reflect the triangle across y-axis
    reflected_y_points = reflect_y(original_points)

    # Plot the original and reflected triangles
    original_x = [point[0] for point in original_points]
    original_y = [point[1] for point in original_points]
```

```
    original_x.append(original_points[0][0])
    original_y.append(original_points[0][1])

    reflected_x = [point[0] for point in reflected_x_points]
    reflected_y = [point[1] for point in reflected_x_points]
    reflected_x.append(reflected_x_points[0][0])
    reflected_y.append(reflected_x_points[0][1])

    reflected_x_y = [point[0] for point in reflected_y_points]
    reflected_y_y = [point[1] for point in reflected_y_points]
    reflected_x_y.append(reflected_y_points[0][0])
    reflected_y_y.append(reflected_y_points[0][1])

    plt.plot(original_x, original_y, 'bo-')
    plt.plot(reflected_x, reflected_y, 'ro-')
    plt.plot(reflected_x_y, reflected_y_y, 'go-')
    plt.gca().set_aspect('equal')
    plt.show()

if __name__ == "__main__":
    main()
```

**Practical 4 Composite Transformation**

**1. Translate the rotate a polygon**

**2. Rotate then scale a shape.**

**3. Apply translation , rotation and scaling sequentially.**

**4. Create animation**

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

# ----------------------------
# Transformation Functions
# ----------------------------
```

```
def translate(points, tx, ty):
    T = np.array([[1, 0, tx],
                  [0, 1, ty],
                  [0, 0, 1]])
    return T @ points

def rotate(points, angle):
    rad = np.deg2rad(angle)
    R = np.array([[np.cos(rad), -np.sin(rad), 0],
                  [np.sin(rad), np.cos(rad), 0],
                  [0, 0, 1]])
    return R @ points

def scale(points, sx, sy):
    S = np.array([[sx, 0, 0],
                  [0, sy, 0],
                  [0, 0, 1]])
    return S @ points


# ----------------------------
# 1. Translate then Rotate a Polygon
# ----------------------------
polygon = np.array([[0, 2, 1, 0],
                    [0, 0, 2, 0],
                    [1, 1, 1, 1]])

translated = translate(polygon, 3, 2)
rotated = rotate(translated, 45)

plt.figure(figsize=(6, 5))
plt.plot(polygon[0], polygon[1], 'b-', label='Original Polygon')
plt.plot(rotated[0], rotated[1], 'r-', label='Translated + Rotated')
plt.title('1. Translation followed by Rotation')
plt.legend()
```

```
plt.axis('equal')
plt.grid(True)
plt.show()


# ----------------------------
# 2. Rotate then Scale a Shape
# ----------------------------
square = np.array([[0, 2, 2, 0, 0],
                   [0, 0, 2, 2, 0],
                   [1, 1, 1, 1, 1]])

rotated_sq = rotate(square, 30)
scaled_sq = scale(rotated_sq, 1.5, 0.5)

plt.figure(figsize=(6, 5))
plt.plot(square[0], square[1], 'g-', label='Original Square')
plt.plot(scaled_sq[0], scaled_sq[1], 'r-', label='Rotated + Scaled')
plt.title('2. Rotation followed by Scaling')
plt.legend()
plt.axis('equal')
plt.grid(True)
plt.show()


# ----------------------------
# 3. Apply Translation, Rotation, and Scaling Sequentially
# ----------------------------
triangle = np.array([[1, 3, 2, 1],
                     [1, 1, 3, 1],
                     [1, 1, 1, 1]])

T1 = translate(triangle, 2, 1)
R1 = rotate(T1, 60)
S1 = scale(R1, 1.5, 1.5)
```

```
plt.figure(figsize=(6, 5))
plt.plot(triangle[0], triangle[1], 'b-', label='Original Triangle')
plt.plot(S1[0], S1[1], 'r-', label='After Translation + Rotation +
Scaling')
plt.title('3. Sequential Composite Transformation')
plt.legend()
plt.axis('equal')
plt.grid(True)
plt.show()


# ----------------------------
# 4. Animation: Combine All Transformations
# ----------------------------
fig, ax = plt.subplots(figsize=(6, 5))
ax.set_xlim(-5, 15)
ax.set_ylim(-5, 15)
ax.set_aspect('equal')
ax.grid(True)
plt.title('4. Composite Transformation Animation')

shape = np.array([[0, 2, 1, 0],
                  [0, 0, 2, 0],
                  [1, 1, 1, 1]])

line, = ax.plot([], [], 'r-', linewidth=2)

def animate(i):
    angle = i * 3
    tx, ty = i * 0.1, i * 0.05
    sx, sy = 1 + i * 0.01, 1 + i * 0.01

    p = translate(shape, tx, ty)
    p = rotate(p, angle)
```

```
    p = scale(p, sx, sy)

    line.set_data(p[0], p[1])
    return line,

ani = animation.FuncAnimation(fig, animate, frames=100,
interval=100, blit=True)
plt.show()
```

**Practical 5 Clipping algorithm**
**1. Draw a window and line segment**
**2. Clip the line segment using Cohen Sutherland**
**3. Take multiple line inputs and clip them**

```
import matplotlib.pyplot as plt
# Region codes
INSIDE = 0  # 0000
LEFT = 1    # 0001
RIGHT = 2   # 0010
BOTTOM = 4  # 0100
TOP = 8     # 1000


# Function to compute region code
def compute_code(x, y, x_min, y_min, x_max, y_max):
    code = INSIDE
    if x < x_min:
        code |= LEFT
    elif x > x_max:
        code |= RIGHT
    if y < y_min:
        code |= BOTTOM
    elif y > y_max:
        code |= TOP
    return code
```

```
# Cohen–Sutherland clipping function
def cohen_sutherland_clip(x1, y1, x2, y2, x_min, y_min, x_max,
y_max):
    code1 = compute_code(x1, y1, x_min, y_min, x_max, y_max)
    code2 = compute_code(x2, y2, x_min, y_min, x_max, y_max)
    accept = False

    while True:
        if code1 == 0 and code2 == 0:
            # Both points inside
            accept = True
            break
        elif (code1 & code2) != 0:
            # Both points share an outside zone (trivial reject)
            break
        else:
            # At least one endpoint is outside
            x, y = 1.0, 1.0
            # Pick one point outside window
            out_code = code1 if code1 != 0 else code2

            # Find intersection point
            if out_code & TOP:
                x = x1 + (x2 - x1) * (y_max - y1) / (y2 - y1)
                y = y_max
            elif out_code & BOTTOM:
                x = x1 + (x2 - x1) * (y_min - y1) / (y2 - y1)
                y = y_min
            elif out_code & RIGHT:
                y = y1 + (y2 - y1) * (x_max - x1) / (x2 - x1)
                x = x_max
            elif out_code & LEFT:
                y = y1 + (y2 - y1) * (x_min - x1) / (x2 - x1)
```

```
            x = x_min

        # Replace point outside with intersection
        if out_code == code1:
            x1, y1 = x, y
            code1 = compute_code(x1, y1, x_min, y_min, x_max,
y_max)
        else:
            x2, y2 = x, y
            code2 = compute_code(x2, y2, x_min, y_min, x_max,
y_max)

    if accept:
        return (x1, y1, x2, y2)
    else:
        return None


# ----------------------------------------------------------
# Main Program
# ----------------------------------------------------------
if __name__ == "__main__":

    # Define window boundaries
    x_min, y_min = 2, 2
    x_max, y_max = 8, 6

    # Draw window
    plt.figure(figsize=(6, 6))
    plt.plot([x_min, x_max, x_max, x_min, x_min],
             [y_min, y_min, y_max, y_max, y_min],
             'k-', linewidth=2, label='Clipping Window')

    # 1 Draw initial lines (multiple)
    lines = [
```

```
        (1, 1, 9, 7),   # Intersects window
        (3, 3, 7, 5),   # Fully inside
        (0, 5, 2, 8),   # Partially outside
        (9, 1, 11, 5)   # Fully outside
    ]

    # Plot original lines in red
    for (x1, y1, x2, y2) in lines:
        plt.plot([x1, x2], [y1, y2], 'r--', label='Original Line' if (x1, y1,
x2, y2) == lines[0] else "")

    # 2 Clip each line using Cohen–Sutherland
    for (x1, y1, x2, y2) in lines:
        clipped = cohen_sutherland_clip(x1, y1, x2, y2, x_min,
y_min, x_max, y_max)
        if clipped:
            cx1, cy1, cx2, cy2 = clipped
            plt.plot([cx1, cx2], [cy1, cy2], 'g-', linewidth=2,
label='Clipped Line' if (x1, y1, x2, y2) == lines[0] else "")

    # Window and plot setup
    plt.xlim(0, 12)
    plt.ylim(0, 10)
    plt.title("Cohen–Sutherland Line Clipping Algorithm")
    plt.xlabel("X-axis")
    plt.ylabel("Y-axis")
    plt.legend()
    plt.grid(True)
    plt.show()
```

**Practical 6 Interactive Graphics with python**
**a. Move a rectangle using arrow keys**
**b. Change the color of shape on key press**

**Code 6a**

```
import pygame
import sys

# Initialize Pygame
pygame.init()

# Screen setup
width, height = 800, 600
screen = pygame.display.set_mode((width, height))
pygame.display.set_caption("Move Rectangle with Arrow Keys")

# Colors
WHITE = (255, 255, 255)
BLUE = (0, 0, 255)

# Rectangle properties
rect_x, rect_y = width // 2, height // 2
rect_width, rect_height = 100, 50
speed = 5  # Movement speed

# Main loop
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    # Key press detection
    keys = pygame.key.get_pressed()
    if keys[pygame.K_UP]:
        rect_y -= speed
    if keys[pygame.K_DOWN]:
        rect_y += speed
    if keys[pygame.K_LEFT]:
```

```
        rect_x -= speed
    if keys[pygame.K_RIGHT]:
        rect_x += speed

    # Drawing
    screen.fill(WHITE)
    pygame.draw.rect(screen, BLUE, (rect_x, rect_y, rect_width,
rect_height))
    pygame.display.flip()
```

**Code 6b**

```
import pygame
import sys

pygame.init()

width, height = 800, 600
screen = pygame.display.set_mode((width, height))
pygame.display.set_caption("Move Rectangle and Change Color")

WHITE = (255, 255, 255)
BLUE = (0, 0, 255)
RED = (255, 0, 0)
GREEN = (0, 255, 0)
YELLOW = (255, 255, 0)

rect_x, rect_y = width // 2, height // 2
rect_width, rect_height = 100, 50
speed = 5

rect_color = BLUE  # Initial color

clock = pygame.time.Clock()

while True:
```

```
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
        elif event.type == pygame.KEYDOWN:
            # Change color only on KEYDOWN event
            if event.key == pygame.K_r:
                rect_color = RED
            elif event.key == pygame.K_g:
                rect_color = GREEN
            elif event.key == pygame.K_b:
                rect_color = BLUE
            elif event.key == pygame.K_y:
                rect_color = YELLOW

    keys = pygame.key.get_pressed()
    if keys[pygame.K_UP]:
        rect_y -= speed
    if keys[pygame.K_DOWN]:
        rect_y += speed
    if keys[pygame.K_LEFT]:
        rect_x -= speed
    if keys[pygame.K_RIGHT]:
        rect_x += speed

    screen.fill(WHITE)
    pygame.draw.rect(screen, rect_color, (rect_x, rect_y, rect_width,
rect_height))
    pygame.display.flip()
    clock.tick(60)
```

**Practical 7 Basic Animation and object movement**

**1. Animate a ball moving horizontally across the screen.**

**2. Animate multiple balls moving independently with different speeds**

**Code 7a**

```
import pygame
import sys

pygame.init()
pygame.mixer.init()

width, height = 800, 600
screen = pygame.display.set_mode((width, height))
pygame.display.set_caption("Horizontal Ball Animation")

WHITE = (255, 255, 255)
RED = (255, 0, 0)
movie_sound=pygame.mixer.Sound('C:/Users/HP/Downloads/file
_example_WAV_1MG.wav')
ball_radius = 30
x = ball_radius  # Start at left edge
y = height // 2  # Vertical center
speed = 5

clock = pygame.time.Clock()

while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    x += speed
    if x - ball_radius > width:
        x = -ball_radius  # Reset to left beyond the window for
smooth loop

    screen.fill(WHITE)
    pygame.draw.circle(screen, RED, (x, y), ball_radius)
```

```
    pygame.display.flip()
    clock.tick(60)
```

**Code 7b**

```
import pygame
import sys
import random

pygame.init()

width, height = 800, 600
screen = pygame.display.set_mode((width, height))
pygame.display.set_caption("Multiple Balls Animation")

WHITE = (255, 255, 255)
colors = [(255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 165, 0), (128, 0,
128)]

ball_radius = 20

# Create multiple balls with random starting x, fixed y, random
speeds, and colors
balls = []
for i in range(5):
    x = random.randint(ball_radius, width - ball_radius)
    y = (i + 1) * (height // 6)
    speed = random.randint(2, 8)
    color = colors[i % len(colors)]
    balls.append({'x': x, 'y': y, 'speed': speed, 'color': color})

clock = pygame.time.Clock()

while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
```

```
            pygame.quit()
            sys.exit()

    screen.fill(WHITE)

    # Update and draw each ball independently
    for ball in balls:
        ball['x'] += ball['speed']
        if ball['x'] - ball_radius > width:
            ball['x'] = -ball_radius  # Loop back to left
        pygame.draw.circle(screen, ball['color'], (int(ball['x']),
ball['y']), ball_radius)

    pygame.display.flip()
    clock.tick(60)
```

**Practical 8 Transformation and rotation in animation**

**1. Animate a rotating triangle or square around its center**

**2. create a pulsating circle or square animation**

**3. Combine rotation and translation to move and rotate an object simultaneously**

**4. Animate reflection or shearing transformation on a moving object**

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

# -------------------------
# Helper Transformation Functions
# -------------------------
def rotate(points, angle, center=(0, 0)):
    rad = np.deg2rad(angle)
    R = np.array([[np.cos(rad), -np.sin(rad)],
                  [np.sin(rad), np.cos(rad)]])
```

```
    centered = points - np.array(center).reshape(2, 1)
    rotated = R @ centered + np.array(center).reshape(2, 1)
    return rotated

def translate(points, tx, ty):
    return points + np.array([[tx], [ty]])

def scale(points, factor, center=(0, 0)):
    centered = points - np.array(center).reshape(2, 1)
    scaled = centered * factor + np.array(center).reshape(2, 1)
    return scaled

def reflect_x(points):
    M = np.array([[1, 0], [0, -1]])
    return M @ points

def shear(points, shx=0, shy=0):
    M = np.array([[1, shx], [shy, 1]])
    return M @ points



# --------------------------
# Setup Figure (2x2 Grid)
# --------------------------
fig, axs = plt.subplots(2, 2, figsize=(10, 10))
plt.subplots_adjust(wspace=0.3, hspace=0.3)
for ax in axs.flat:
    ax.set_xlim(-10, 10)
    ax.set_ylim(-10, 10)
    ax.set_aspect('equal')
    ax.grid(True)

axs[0, 0].set_title("1️⃣ Rotating Triangle")
axs[0, 1].set_title("2️⃣ Pulsating Circle")
axs[1, 0].set_title("3️⃣ Rotation + Translation")
```

```
axs[1, 1].set_title("4 Reflection + Shearing")

# -------------------------
# Define Shapes
# -------------------------
triangle = np.array([[0, 2, 1, 0],
                     [0, 0, 2, 0]])

theta = np.linspace(0, 2 * np.pi, 100)
circle = np.array([np.cos(theta), np.sin(theta)])

square = np.array([[0, 2, 2, 0, 0],
                   [0, 0, 2, 2, 0]])

shear_triangle = np.array([[0, 2, 1, 0],
                           [0, 0, 2, 0]])

# Create Line Handles for Animation
line1, = axs[0, 0].plot([], [], 'r-', linewidth=2)
line2, = axs[0, 1].plot([], [], 'b-', linewidth=2)
line3, = axs[1, 0].plot([], [], 'g-', linewidth=2)
line4, = axs[1, 1].plot([], [], 'm-', linewidth=2)


# -------------------------
# Animation Update Function
# -------------------------
def update(frame):
    # 1 Rotating Triangle
    rot = rotate(triangle, frame * 5, center=(1, 1))
    line1.set_data(rot[0], rot[1])

    # 2 Pulsating Circle
    scale_factor = 1 + 0.3 * np.sin(np.deg2rad(frame * 10))
    pulsate = circle * scale_factor
```

```
    line2.set_data(pulsate[0] * 4, pulsate[1] * 4)  # enlarge for
visibility

    # 3 Rotation + Translation
    angle = frame * 8
    tx, ty = frame * 0.1 - 5, np.sin(np.deg2rad(frame * 8)) * 3
    rotated = rotate(square, angle, center=(1, 1))
    moved = translate(rotated, tx, ty)
    line3.set_data(moved[0], moved[1])

    # 4 Reflection + Shearing
    moved2 = translate(shear_triangle, frame * 0.1 - 5, 0)
    if frame % 40 < 20:
        transformed = shear(moved2, shx=0.1 *
np.sin(np.deg2rad(frame * 10)))
    else:
        transformed = reflect_x(moved2)
    line4.set_data(transformed[0], transformed[1])

    return line1, line2, line3, line4


# -------------------------
# Animate All Together
# -------------------------
ani = animation.FuncAnimation(fig, update, frames=200,
interval=100, blit=True)
plt.show()
```

**Practical 9 Color Models and Curve Animation**

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
```

```
# Create the figure and axis
fig, ax = plt.subplots(figsize=(7, 5))
ax.set_xlim(0, 2 * np.pi)
ax.set_ylim(-1.5, 1.5)

# Initial curve data
x = np.linspace(0, 2 * np.pi, 100)
y = np.sin(x)

# Initial line object
(line,) = ax.plot(x, y, lw=2)

# Function to update curve and color for each frame
def animate(frame):
    y = np.sin(x + frame * 0.05)              # Curve animates
horizontally
    # Animate color: map frame number to RGB color
    r = (np.sin(frame * 0.03) + 1) / 2
    g = (np.sin(frame * 0.05 + 2) + 1) / 2
    b = (np.sin(frame * 0.07 + 4) + 1) / 2
    color = (r, g, b)
    line.set_ydata(y)
    line.set_color(color)
    return (line,)

# Create animation
anim = FuncAnimation(fig, animate, frames=100, interval=50,
blit=True)
plt.title("Curve Animation with Color Model (RGB)")
plt.show()
```