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

```
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)
```

```python
turtle.pendown()
for x, y in line_points:
    turtle.goto(x, y)

turtle.exitonclick()
```

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

```python
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 slowest, 10 is fastest)
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()
```

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

```
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 3Dn 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, color.blue)

# 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)

while True:
    rate(30)  # Set the frame rate to 30 frames per second
```

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

```
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()
```

5) **Develop a program to demonstrate 3D transformation on 3D objects**

```
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

# Define a 3D object (modify these vertices for different shapes)
vertices = np.array([
    [1, 1, 1],  # Top front right
    [-1, 1, 1],  # Top back right
    [-1, -1, 1], # Bottom back right
    [1, -1, 1],  # Bottom front right
    [1, 1, -1],  # Top front left
    [-1, 1, -1], # Top back left
    [-1, -1, -1],# Bottom back left
    [1, -1, -1],  # Bottom front left
])

# Define functions for transformations (modify these for different effects)
def translate(vertices, tx, ty, tz):
  """
  Translates the object by the specified amounts in x, y, and z directions.
  """
  return vertices + np.array([tx, ty, tz])

def rotate_x(vertices, angle):
  """
  Rotates the object around the X-axis by the given angle in degrees.
  """
  theta = np.radians(angle)
  rotation_matrix = np.array([[1, 0, 0], [0, np.cos(theta), -np.sin(theta)], [0, np.sin(theta),
np.cos(theta)]])
  return vertices.dot(rotation_matrix)

def rotate_y(vertices, angle):
  """
  Rotates the object around the Y-axis by the given angle in degrees.
  """
  theta = np.radians(angle)
  rotation_matrix = np.array([[np.cos(theta), 0, np.sin(theta)], [0, 1, 0], [-np.sin(theta), 0,
np.cos(theta)]])
  return vertices.dot(rotation_matrix)

def rotate_z(vertices, angle):
  """
  Rotates the object around the Z-axis by the given angle in degrees.
```

```
  """
  theta = np.radians(angle)
  rotation_matrix = np.array([[np.cos(theta), -np.sin(theta), 0], [np.sin(theta),
np.cos(theta), 0], [0, 0, 1]])
  return vertices.dot(rotation_matrix)

def scale(vertices, sx, sy, sz):
  """
  Scales the object by the specified factors in x, y, and z directions.
  """
  return vertices * np.array([sx, sy, sz])

# Apply transformations (replace with desired operations)
transformed_vertices = translate(vertices, 2, 1, 0)  # Translate object
transformed_vertices = rotate_y(transformed_vertices, 60)  # Rotate around Y-axis

# Define viewing parameters (optional, adjust for better visualization)
view_elev = 15  # Elevation angle for viewing (in degrees)
view_azim = -60  # Azimuth angle for viewing (in degrees)

# Plot the original and transformed object
fig = plt.figure(figsize=(10, 6))

ax = fig.add_subplot(121, projection='3d')
ax.scatter(vertices[:, 0], vertices[:, 1], vertices[:, 2], c='red', marker='o',
label='Original')

ax.set_xlabel("X Label")
ax.set_ylabel("Y Label")
ax.set_zlabel("Z Label")

ax = fig.add_subplot(122, projection='3d')
ax.scatter(transformed_vertices[:, 0], transformed_vertices[:, 1],
transformed_vertices[:, 2], c='blue', marker='o', label='Transformed')

ax.set_xlabel("X Label")
ax.set_ylabel("Y Label")
ax.set_zlabel("Z Label")

# Set viewing angles (optional)
ax.view_init(elev=view_elev, azim=view_azim)

plt.title("3D Transformation Demonstration")
plt.legend()
plt.show()
```

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

```
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()
```

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

```
import cv2

import numpy as np

# Define image path (replace with your image path)
image_path = "atc.jpg"

# Load the image
img = cv2.imread(image_path)

# Check if image loading was successful
if img is None:
   print("Error: Could not load image from", image_path)
   exit()

# Get the image height, width, and number of channels
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()
```

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

```
import cv2

import numpy as np

# Load the image
image_path = "atc.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()
```

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

```
import cv2

import numpy as np

# Load the image
image_path = "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()
```

10) **Write a program to blur and smoothing an image**

```
import cv2


# Load the image
image = cv2.imread('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)
```

```python
# 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()
```

11) **Write a program to contour an image.**

```python
import cv2

import numpy as np

# Load the image
image = cv2.imread('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()
```

12) Write a program to detect a face/s in an image

```python
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('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)

# Display the image with detected faces
cv2.imshow('Face Detection', image)

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