```
//1st

#include <iostream>
#include <string>
#include<stdio.h>
#include<math.h>
#include<gl/glut.h>

GLint X1,Y1,X2,Y2;

void LineBres(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
// The argument GL COLOR BUFFER BIT is anOpenGL symbolic constant specifying that it is
the bit values in the color buffer
    int dx=abs(X2-X1),dy=abs(Y2-Y1);
    int p=2*dy-dx;
    int twoDy=2*dy, twoDyDx=2*(dy-dx);
    int p1, p2;
    if(X1>X2)
    {
        p1=X2;
        p2=Y2;
        X2=X1;
    }
    else
    {
        p1=X1;
        p2=Y1;
        X2=X2;
    }
    glBegin(GL_POINTS);
    glVertex2i(p1,p2);

    while(p1<X2)
    {
        p1++;
        if(p<0)
            p+=twoDy;
        else
        {
            p2++; // y coordinates incremetns
            p+=twoDyDx;
        }
        glVertex2i(p1,p2);
    }
    glEnd();
    glFlush();
}

void Init()
{
    glClearColor(1.0,1.0,1.0,0);
    glColor3f(0.0,0.0,0.0);
    glPointSize(4.0);
//glViewport(0,0,50,50);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0,50,0,50);
}

int main(int argc,char **argv)
{
    printf("enter two points for draw lineBresenham:\n");
    printf("\n enter point1(X1,Y1):");
    scanf("%d%d",&X1,&Y1);
    printf("\n enter point2(X2,Y2):");
    scanf("%d%d",&X2,&Y2);
    glutInit(&argc,argv); // initializes the GLUT library.
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB); // sets the initial display mode
    glutInitWindowSize(300,400); // sets the initial window size to 500x500 pixels
//glutInitWindowPosition(0,0); //sets the initial window position to coordinates (100,
```

```
100) on the screen.
    glutCreateWindow("LineBresenham"); // creates a window with the given title
    Init();
    glutDisplayFunc(LineBres); // sets the display callback function
    glutMainLoop(); // enters the GLUT event processing loop
    return 0;
}
```

```
//2nd 2D geometric object operations

#include <GL/glut.h>
#include <math.h>

// Function to initialize OpenGL settings
void init()
{
    glClearColor(1.0, 1.0, 1.0, 1.0); // Set background color to white
    glColor3f(0.0, 0.0, 0.0); // Set drawing color to black
    glMatrixMode(GL_PROJECTION);
    gluOrtho2D(-400, 400, -400, 400); // Set the coordinate system
}

// Function to draw a rectangle
void drawRectangle(float x, float y, float width, float height, float r, float g, float b)
{
    glColor3f(r, g, b);
    glBegin(GL_QUADS);
    glVertex2f(x, y);
    glVertex2f(x + width, y);
    glVertex2f(x + width, y + height);
    glVertex2f(x, y + height);
    glEnd();
}

// Function to draw a circle
void drawCircle(float x, float y, float radius, float r, float g, float b)
{
    glColor3f(r, g, b);
    glBegin(GL_TRIANGLE_FAN);
    glVertex2f(x, y);
    for (int i = 0; i <= 100; ++i)
    {
        float angle = 2 * M_PI * i / 100;
        glVertex2f(x + cos(angle) * radius, y + sin(angle) * radius);
    }
    glEnd();
}

// Function to translate a 2D object
void translate(float dx, float dy)
{
    glTranslatef(dx, dy, 0.0f);
}

// Function to rotate a 2D object
void rotate(float angle)
{
    glRotatef(angle, 0.0f, 0.0f, 1.0f);
}

// Function to scale a 2D object
void scale(float sx, float sy)
{
    glScalef(sx, sy, 1.0f);
}

// Display callback function
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);

    // Draw initial rectangle
    glPushMatrix();
    drawRectangle(-200, 0, 100, 50, 0.0f, 0.0f, 1.0f);
    glPopMatrix();

    // Translate the rectangle
    glPushMatrix();
    translate(200, 0);
    drawRectangle(-200, 0, 100, 50, 0.0f, 1.0f, 0.0f);
```

```
    glPopMatrix();

    // Rotate the rectangle
    glPushMatrix();
    translate(0, 0);
    rotate(45);
    drawRectangle(-200, 0, 100, 50, 1.0f, 0.0f, 0.0f);
    glPopMatrix();

    // Scale the rectangle
    glPushMatrix();
    translate(0, 0);
    scale(2, 2);
    drawRectangle(-200, 0, 100, 50, 0.5f, 0.0f, 0.5f);
    glPopMatrix();

    // Draw initial circle
    glPushMatrix();
    drawCircle(100, 100, 50, 1.0f, 0.0f, 0.0f);
    glPopMatrix();

    // Translate the circle
    glPushMatrix();
    translate(200, 0);
    drawCircle(300, 100, 50, 0.0f, 1.0f, 0.0f);
    glPopMatrix();

    // Rotate the circle (rotation doesn't affect circle shape but demonstrate the
function)
    glPushMatrix();
    translate(300, 100);
    rotate(45);
    drawCircle(0, 0, 50, 0.0f, 0.0f, 1.0f);
    glPopMatrix();

    // Scale the circle
    glPushMatrix();
    translate(300, 100);
    scale(2, 2);
    drawCircle(0, 0, 50, 0.5f, 0.5f, 0.0f);
    glPopMatrix();

    glFlush();
}

// Main function
int main(int argc,char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(800, 800);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("2D Geometric Operations with FreeGLUT");

    init();
    glutDisplayFunc(display);

    glutMainLoop();
    return 0;
}
```

```
//3rd

#include <GL/glut.h>
#include <cmath>
// Function to initialize OpenGL settings
void init()
{
    glClearColor(1.0, 1.0, 1.0, 1.0); // Set background color to white
//glEnable(GL_DEPTH_TEST); // Enable depth testing for 3D rendering
    glMatrixMode(GL_PROJECTION);
    gluPerspective(45.0, 1.0, 1.0, 100.0); // Set perspective projection
    glMatrixMode(GL_MODELVIEW);
}
// Function to draw a cuboid
void drawCuboid(float x, float y, float z, float length, float width, float height, float
r, float g,
                float b)
{
    glColor3f(r, g, b); // Set the color for the cuboid using RGB values.
    glPushMatrix(); // Save the current transformation matrix.
    glTranslatef(x, y, z); // Translate the cuboid to the specified position (x, y, z).
    glScalef(length, width, height); // Scale the cuboid to the specified dimensions.
    glutSolidCube(1.0); // Draw a unit cube (1x1x1) which is then scaled to the desired
size.
    glPopMatrix(); // Restore the original transformation matrix.
}
// Function to draw a cylinder

void drawCylinder(float x, float y, float z, float radius, float height, float r, float g,
float b)
{
    glColor3f(r, g, b); // Set the color for the cylinder using RGB values.
    glPushMatrix(); // Save the current transformation matrix.
    glTranslatef(x, y, z); // Translate the cylinder to the specified position (x, y, z).
    glRotatef(-90, 1.0, 0.0, 0.0); // Rotate to align with the z-axis
    GLUquadric* quad = gluNewQuadric(); // Create a new quadric object.
    gluCylinder(quad, radius, radius, height, 32, 32); // Draw the cylinder with the
specifiedradius and height.
    gluDeleteQuadric(quad); // Delete the quadric object to free memory.
    glPopMatrix(); // Restore the original transformation matrix.
}

// Function to translate a 3D object
void translate(float dx, float dy, float dz)
{
    glTranslatef(dx, dy, dz);
}
// Function to rotate a 3D object
void rotate(float angle, float x, float y, float z)
{
    glRotatef(angle, x, y, z);
}
// Function to scale a 3D object
void scale(float sx, float sy, float sz)
{
    glScalef(sx, sy, sz);
}
// Display callback function
void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt(0.0, 0.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0); // Set the camera view

// Draw a cuboid
    glPushMatrix();
    drawCuboid(-2, 0, 0, 2, 2, 1, 0.0f, 0.0f, 1.0f);

// Translate the cuboid
    translate(4, 0, 0);
    drawCuboid(-2, 0, 0, 2, 2, 1, 0.0f, 1.0f, 0.0f);
```

```
// Rotate the cuboid
    glPushMatrix();
    rotate(45, 0, 1, 0);
    drawCuboid(-2, 0, 0, 2, 2, 1, 1.0f, 0.0f, 0.0f);
    glPopMatrix();

// Scale the cuboid
    glPushMatrix();
    scale(1.5, 1.5, 1.5);
    drawCuboid(-2, 0, 0, 2, 2, 1, 0.5f, 0.0f, 0.5f);
    glPopMatrix();

    glPopMatrix();

// Draw a cylinder
    glPushMatrix();
    drawCylinder(2, 2, 0, 1, 10, 1.0f, 0.0f, 0.0f);

// Translate the cylinder
    translate(0, -2, 0);
    drawCylinder(2, 2, 0, 1, 1, 0.0f, 1.0f, 0.0f);

// Rotate the cylinder
    glPushMatrix();
    rotate(30, 1, 0, 0);
    drawCylinder(2, 2, 0, 1, 1, 0.0f, 0.0f, 1.0f);
    glPopMatrix();

// Scale the cylinder
    glPushMatrix();
    scale(1.5, 1.5, 1.5);
    drawCylinder(2, 4, 0, 1, 1, 0.5f, 0.5f, 0.0f);
    glPopMatrix();
    glPopMatrix();

    glutSwapBuffers(); // Swap the buffers to display the rendered frame
}

// Timer callback function
void timer(int value)
{
    glutPostRedisplay(); // Redraw the scene
    glutTimerFunc(33, timer, 0); // Set the timer to call itself again after 33 ms
(approx 30 FPS)
}
// Main function
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(800,800);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("3D Geometric Operations with FreeGLUT");
    init();
    glutDisplayFunc(display);
//glutTimerFunc(33, timer, 0); // Start the timer
    glutMainLoop();
    return 0;
}
```

```
//4th

#include <GL/glut.h>
#include <iostream>

// Define the dimensions of the canvas
const int canvas_width = 500;
const int canvas_height = 500;

// Define the initial object (a square)
GLfloat obj_points[][2] = { {100, 100}, {200, 100}, {200, 200}, {100, 200} };
const int num_points = sizeof(obj_points) / sizeof(obj_points[0]);

// Define the transformation parameters
GLfloat translation_x = 100;
GLfloat translation_y = 50;
GLfloat rotation_angle = 45;
GLfloat scaling_factor = 1.5;

// Function to initialize OpenGL settings
void init() {
    glClearColor(1.0, 1.0, 1.0, 1.0); // Set background color to white
    glMatrixMode(GL_PROJECTION);
    gluOrtho2D(0, canvas_width, 0, canvas_height); // Set the coordinate system
}

// Function to draw the objects on the canvas
void drawObjects() {
    // Draw the initial object (a square)
    glColor3f(0.0, 0.0, 0.0);
    glBegin(GL_LINE_LOOP);
    for (int i = 0; i < num_points; ++i) {
        glVertex2f(obj_points[i][0], obj_points[i][1]);
    }
    glEnd();

    // Apply transformations and draw translated object
    glColor3f(0.0, 1.0, 0.0);
    glPushMatrix();
    glTranslatef(translation_x, translation_y, 0.0);
    glBegin(GL_LINE_LOOP);
    for (int i = 0; i < num_points; ++i) {
        glVertex2f(obj_points[i][0], obj_points[i][1]);
    }
    glEnd();
    glPopMatrix();

    // Apply transformations and draw rotated object
    glColor3f(1.0, 0.0, 0.0);
    glPushMatrix();
    glTranslatef(obj_points[0][0], obj_points[0][1], 0.0);
    glRotatef(rotation_angle, 0.0, 0.0, 1.0);
    glTranslatef(-obj_points[0][0], -obj_points[0][1], 0.0);
    glBegin(GL_LINE_LOOP);
    for (int i = 0; i < num_points; ++i) {
        glVertex2f(obj_points[i][0], obj_points[i][1]);
    }
    glEnd();
    glPopMatrix();

    // Apply transformations and draw scaled object
    glColor3f(0.0, 0.0, 1.0);
    glPushMatrix();
    glTranslatef(obj_points[0][0], obj_points[0][1], 0.0);
    glScalef(scaling_factor, scaling_factor, 1.0);
    glTranslatef(-obj_points[0][0], -obj_points[0][1], 0.0);
    glBegin(GL_LINE_LOOP);
    for (int i = 0; i < num_points; ++i) {
        glVertex2f(obj_points[i][0], obj_points[i][1]);
    }
    glEnd();
    glPopMatrix();
```

```
}

// Display callback function
void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    // Draw the objects on the canvas
    drawObjects();

    glutSwapBuffers();
}

// Main function
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(canvas_width, canvas_height);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("2D Transformations with FreeGLUT");

    init();
    glutDisplayFunc(display);

    glutMainLoop();
    return 0;
}
```

```
//5th

#include <GL/glut.h>
#include <iostream>
#include <cmath>

// Define the dimensions of the display
const int display_width = 800;
const int display_height = 600;
// Define the 3D object (a cube)
GLfloat vertices[][3] =
{
    {-1, -1, -1},
    { 1, -1, -1},
    { 1,  1, -1},
    {-1,  1, -1},
    {-1, -1,  1},
    { 1, -1,  1},
    { 1,  1,  1},
    {-1,  1,  1}
};

GLuint edges[][2] =
{
    {0, 1}, {1, 2}, {2, 3}, {3, 0},
    {4, 5}, {5, 6}, {6, 7}, {7, 4},
    {0, 4}, {1, 5}, {2, 6}, {3, 7}
};

GLfloat angle = 0;

// Function to initialize OpenGL settings
void init()
{
    glClearColor(0.0, 0.0, 0.0, 1.0); // Set background color to black
    glEnable(GL_DEPTH_TEST); // Enable depth testing
    glMatrixMode(GL_PROJECTION);
    gluPerspective(45, (GLfloat)display_width / (GLfloat)display_height, 0.1, 50.0);
    glMatrixMode(GL_MODELVIEW);
}

// Function to draw the 3D object
void drawCube()
{
    glBegin(GL_LINES);
    for (int i = 0; i < 12; ++i)
    {
        glVertex3fv(vertices[edges[i][0]]);
        glVertex3fv(vertices[edges[i][1]]); //1
    }
    glEnd();
}

// Display callback function
void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
// Apply transformations
    glTranslatef(0.0f, 0.0f, -5.0f); // Translation matrix
    glRotatef(angle, 1.0f, 1.0f, 0.0f); // Rotation matrix
    glScalef(1.5f, 1.5f, 1.5f); // Scaling matrix
// Draw the 3D object
    drawCube();

    glutSwapBuffers();
}

// Timer callback function
void timer(int value)
{
    angle += 1.0f;
```

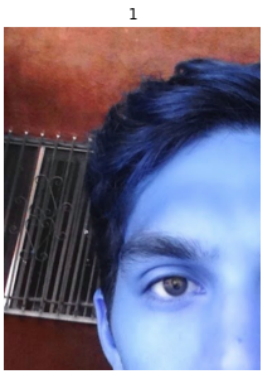
```
    if (angle > 360)
    {
        angle -= 360;
    }
    glutPostRedisplay(); // Redraw the scene
    glutTimerFunc(50, timer, 0); // Set the timer to call itself again after 33 ms
(approx 30 FPS)
}

// Main function
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(display_width, display_height);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("3D Transformations with FreeGLUT");

    init();
    glutDisplayFunc(display);
    glutTimerFunc(50, timer, 0);// Start the timer

    glutMainLoop();
    return 0;
}
```

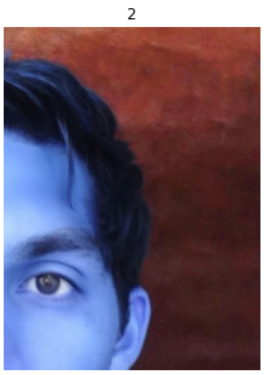
```
//6th

#include <GL/freeglut.h>
#include <cstdlib>
#include <ctime>
#include <cmath>

// Define object properties
const int num_objects = 10;
struct Object
{
    float x;
    float y;
    float radius;
    float speed_x;
    float speed_y;
    float color[3];
};
Object objects[num_objects];

// Initialize objects
void initObjects()
{
    srand(time(NULL));
    for (int i = 0; i < num_objects; ++i)
    {
        objects[i].x = rand() % (800 - 100) + 50;
        objects[i].y = rand() % (600 - 100) + 50;
        objects[i].radius = rand() % 21 + 10;
        objects[i].speed_x = rand() % 11 - 5;
        objects[i].speed_y = rand() % 11 - 5;
        objects[i].color[0] = (float)rand() / RAND_MAX; // Random value between 0 and 1
        objects[i].color[1] = (float)rand() / RAND_MAX;
        objects[i].color[2] = (float)rand() / RAND_MAX;
    }
}

// Draw objects
void drawObjects()
{
    for (int i = 0; i < num_objects; ++i)
    {
        glPushMatrix();
        glTranslatef(objects[i].x, objects[i].y, 0.0);
        glColor3fv(objects[i].color);
        glBegin(GL_POLYGON);
        for (float angle = 0; angle < 360; angle += 10)
        {
            float x = objects[i].radius * cos(angle * 3.14159 / 180.0);
            float y = objects[i].radius * sin(angle * 3.14159 / 180.0);
            glVertex2f(x, y);
        }
        glEnd();
        glPopMatrix();
    }
}

// Update objects
void updateObjects()
{
    for (int i = 0; i < num_objects; ++i)
    {
        objects[i].x += objects[i].speed_x;
        objects[i].y += objects[i].speed_y;
        if (objects[i].x - objects[i].radius < 0 || objects[i].x + objects[i].radius > 800)
        {
            objects[i].speed_x = -objects[i].speed_x;
        }
        if (objects[i].y - objects[i].radius < 0 || objects[i].y + objects[i].radius > 600)
        {
            objects[i].speed_y = -objects[i].speed_y;
        }
```

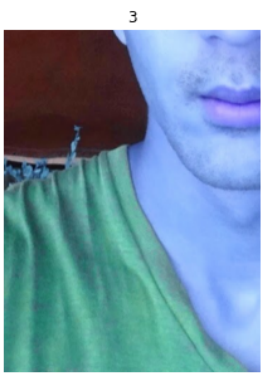
```
    }
}

// Display function
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    drawObjects();
    glutSwapBuffers();
    updateObjects();
}

// Reshape function
void reshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, 800, 0, 600);
    glMatrixMode(GL_MODELVIEW);
}

// Main function
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(800, 600);
    glutCreateWindow("Animation Effects");
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    initObjects();
    glutMainLoop();
    return 0;
}
```

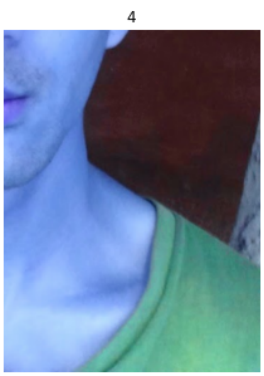
In [4]:

```
#7th

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

# Read the image
img = cv2.imread('C:\\Users\CSELAB2\Desktop\pictures\9.jpeg')

# Get the height and width of the image
height, width = img.shape[:2]
# Split the image into four quadrants
quad1 = img[:height//2, :width//2]
quad2 = img[:height//2, width//2:]
quad3 = img[height//2:, :width//2]
quad4 = img[height//2:, width//2:]
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(quad1)
plt.title("1")
plt.axis("off")

plt.subplot(1, 2, 2)
plt.imshow(quad2)
plt.title("2")
plt.axis("off")

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(quad3)
plt.title("3")
plt.axis("off")

plt.subplot(1, 2, 2)
plt.imshow(quad4)
plt.title("4")
plt.axis("off")

plt.show()
```









In [8]:

```
#8th

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

def translate_image(image, dx, dy):
    rows, cols = image.shape[:2]
    translation_matrix = np.float32([[1, 0, dx], [0, 1, dy]])
    translated_image = cv2.warpAffine(image, translation_matrix, (cols, rows))
    return translated_image

# Read the image
image = cv2.imread('C:\\Users\CSELAB2\Desktop\pictures\9.jpeg')
##double slash has to be given

# Get image dimensions
height, width = image.shape[:2]

# Calculate the center coordinates of the image
center = (width // 2, height // 2)
rotation_value = int(input("Enter the degree of Rotation:"))
scaling_value = int(input("Enter the zooming factor:"))

# Create the 2D rotation matrix
rotated = cv2.getRotationMatrix2D(center=center, angle=rotation_value, scale=1)
rotated_image = cv2.warpAffine(src=image, M=rotated, dsize=(width, height))
scaled = cv2.getRotationMatrix2D(center=center, angle=0, scale=scaling_value)
scaled_image = cv2.warpAffine(src=rotated_image, M=scaled, dsize=(width, height))
h = int(input("How many pixels you want the image to be translated horizontally? "))
v = int(input("How many pixels you want the image to be translated vertically? "))

translated_image = translate_image(scaled_image, dx=h, dy=v)
cv2.imwrite('Final_image.png', translated_image)
```

```
Enter the degree of Rotation:2
Enter the zooming factor:45
How many pixels you want the image to be translated horizontally? 34
How many pixels you want the image to be translated vertically? 34
```

Out[8]:

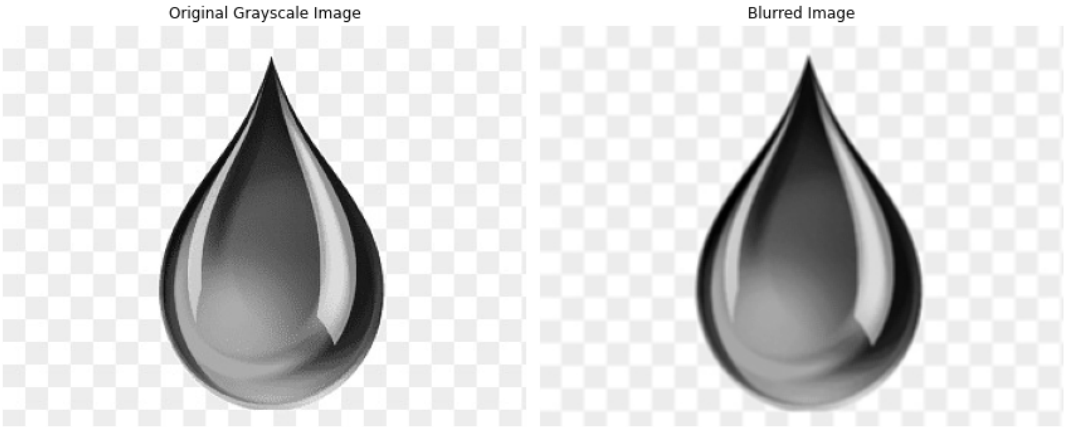
```
True
```

In [29]:

```
#9th

import cv2
import numpy as np

# Load the image
image_path = "C://Users/CSELAB2/Desktop/pictures/6.jpeg"  # 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()
```

In [26]:

```
#10th

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

# Load the image in grayscale
img = cv2.imread("C://Users/CSELAB2/Desktop/pictures/6.jpeg", cv2.IMREAD_GRAYSCALE)

def blur():
    return np.ones((3, 3), np.float32) / 9  # 3x3 averaging kernel for blur

def filtering(image, kernel):
    m, n = kernel.shape
    if (m == n):
        y, x = image.shape
        y = y - m + 1
        x = x - m + 1
        new_image = np.zeros((y, x))
        for i in range(y):
            for j in range(x):
                new_image[i][j] = np.sum(image[i:i+m, j:j+m] * kernel)
    return new_image

# Apply blur filter
blurred_image = filtering(img, blur())

# Display the original and blurred images
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.imshow(img, cmap='gray')
plt.title("Original Grayscale Image")
plt.axis("off")

plt.subplot(1, 2, 2)
plt.imshow(blurred_image, cmap='gray')
plt.title("Blurred Image")
plt.axis("off")

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



In [30]:

```
#11tg

import cv2
import numpy as np

# Specify the path to your image (adjust this path according to your system)
image_path = r'C:\Users\CSELAB2\Desktop\pictures\4.jpeg'

# Read the image
image = cv2.imread(image_path)

# Convert the image to grayscale (contours work best on binary images)
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Apply thresholding (you can use other techniques like Sobel edges)
_, binary_image = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY)

# Find contours
contours, _ = cv2.findContours(binary_image, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

# Draw all contours on the original image
cv2.drawContours(image, contours, -1, (0, 255, 0), 3)

# Display the result
cv2.imshow('Contours', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

In [*]:

```
#12th

import cv2

# Load the pre-trained Haar Cascade classifier for face detection
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')
eye_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_eye.xml')

# Read the input image (replace 'your_image.jpg' with the actual image path)
image_path = "C:\\Users\CSELAB2\Desktop\pictures\9.jpeg"
image = cv2.imread(image_path)

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

# Detect faces in the image
faces = face_cascade.detectMultiScale(gray, scaleFactor=1.3, minNeighbors=5)

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

# Save or display the result
cv2.imwrite('detected_faces.jpg', image)  # Save the result
cv2.imshow('Detected Faces', image)  # Display the result
cv2.waitKey(0)
cv2.destroyAllWindows()
```

In [ ]:

```

```