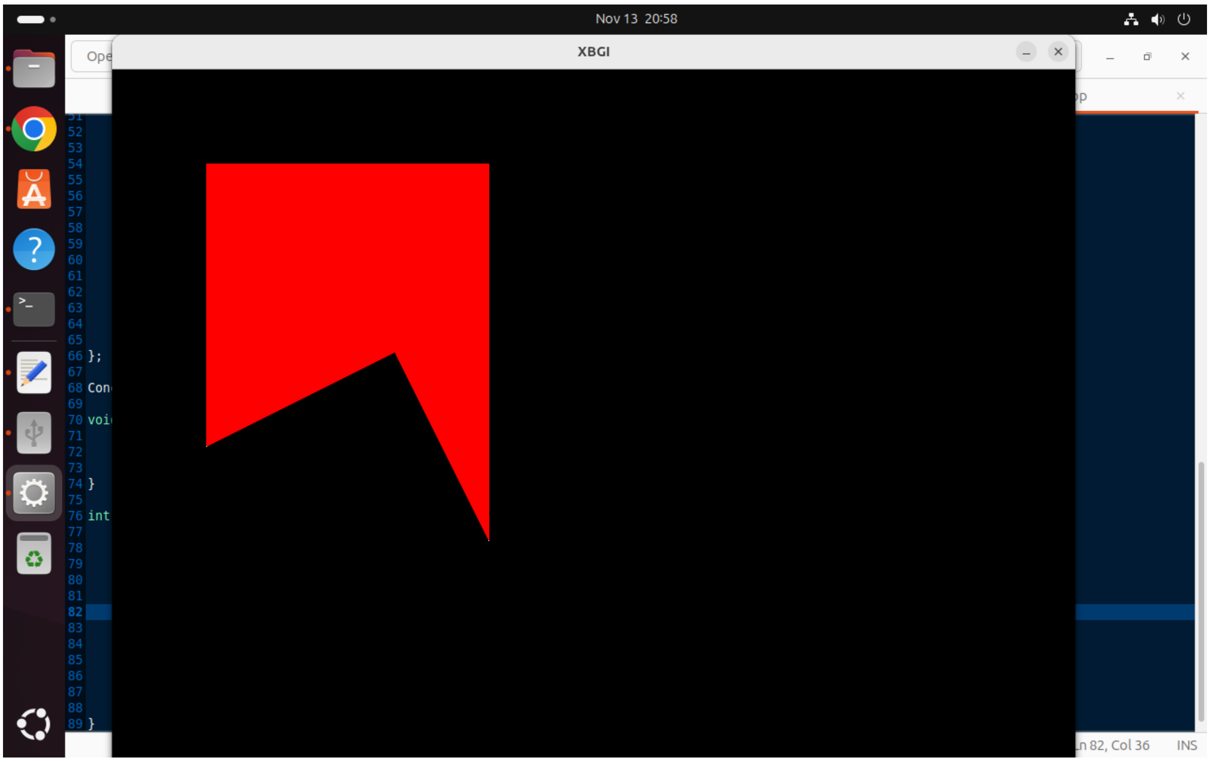
Q1. Write C++ program to draw a concave polygon and fill it with desired color using scan fill algorithm. Apply the concept of inheritance.

```
#include <graphics.h>
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

class Shape {
	public:
	virtual void draw() = 0;
	virtual void scanline() = 0;
};

class ConcavePoly : public Shape {
	protected:
	vector<pair<int, int>> vertices;
	int fillColor;

	public:
	ConcavePoly(const vector<pair<int, int>>& verts, int color) {
		vertices = verts;
		fillColor = color;
	}

	void draw() override {
		for (size_t i = 0; i < vertices.size(); ++i) {
			size_t next = (i + 1) % vertices.size();
			line(vertices[i].first, vertices[i].second, vertices[next].first, vertices[next].second);
		}
	}

	void scanline() override {
		int ymin = 10000, ymax = -10000;
		for (const auto& vertex : vertices) {
			ymin = min(ymin, vertex.second);
			ymax = max(ymax, vertex.second);
		}

		for (int y = ymin; y <= ymax; y++) {
			vector<int> xIntersect;

			for (size_t i = 0; i < vertices.size(); i++) {
				size_t next = (i + 1) % vertices.size();
				int x1 = vertices[i].first;
				int y1 = vertices[i].second;
				int x2 = vertices[next].first;
				int y2 = vertices[next].second;

				if (y1 == y2) continue;

				if (y >= min(y1, y2) && y < max(y1, y2)) {
					int x = x1 + (y - y1) * (x2 - x1) / (y2 - y1);
					xIntersect.push_back(x);
				}
			}
```

```
                        sort(xIntersect.begin(), xIntersect.end());

                        for (size_t i = 0; i < xIntersect.size(); i += 2) {
                                setcolor(fillColor);
                                line(xIntersect[i], y, xIntersect[i + 1], y);
                        }
                }
        }
};

ConcavePoly* Polygon;

void display() {
        cleardevice();
        Polygon->draw();
        Polygon->scanline();
}

int main() {
        vector<pair<int, int>> vertices = {{100, 100}, {400, 100}, {400, 500}, {300, 300}, {100, 400}};
        int color = RED; // Define a color using BGI color constants
        Polygon = new ConcavePoly(vertices, color);

        int gd = X11, gm = X11_1024x768;
  initgraph(&gd, &gm, (char*)"");
        display();

        getch(); // Wait for user input
        closegraph();
        delete Polygon;
        return 0;
}
```

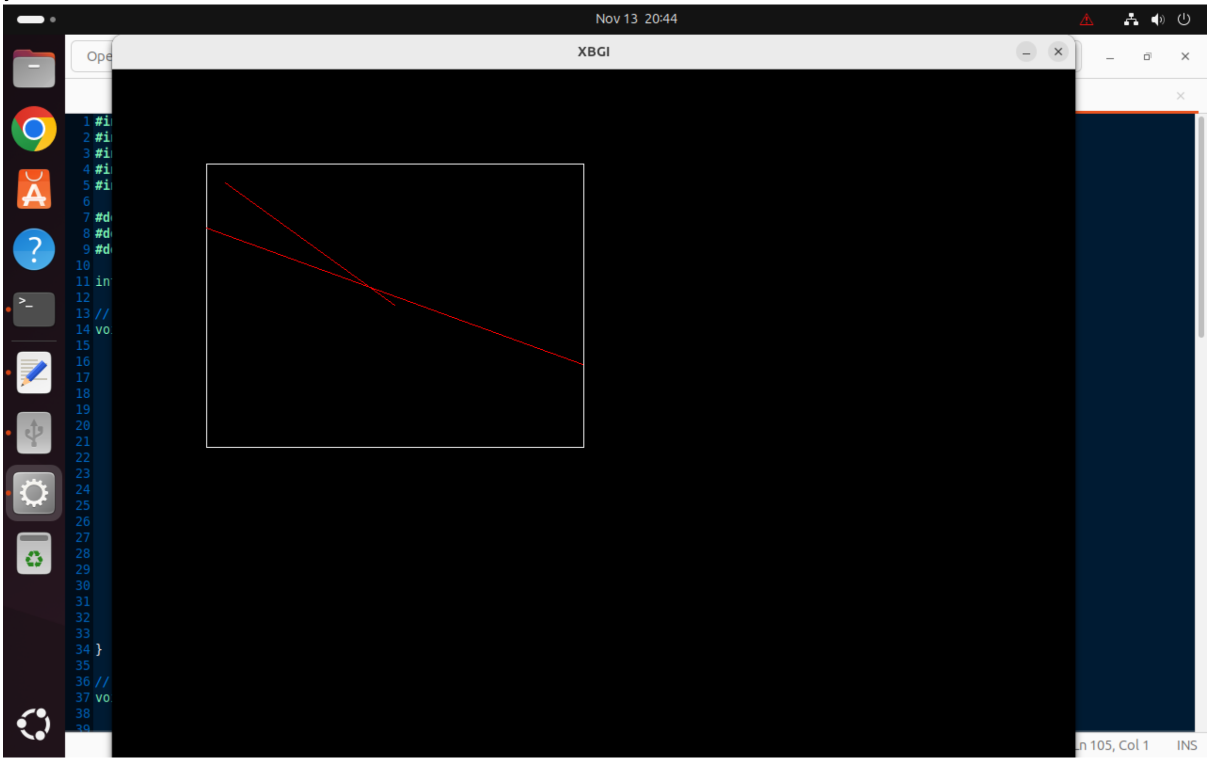


Q2. Write C++ program to implement Cohen Southerland line clipping algorithm.

```
#include <graphics.h>
#include <iostream>
using namespace std;

const int INSIDE = 0; // 0000
const int LEFT = 1;   // 0001
const int RIGHT = 2;  // 0010
const int BOTTOM = 4; // 0100
const int TOP = 8;    // 1000

int x_min, y_min, x_max, y_max;
// Function to compute region code for a point (x, y)
int computeCode(int x, int y) {
    int code = INSIDE;

    if (x < x_min)       // to the left of rectangle
        code |= LEFT;
    else if (x > x_max)  // to the right of rectangle
        code |= RIGHT;
    if (y < y_min)       // below the rectangle
        code |= BOTTOM;
    else if (y > y_max)  // above the rectangle
        code |= TOP;

    return code;
}
// Cohen-Sutherland clipping algorithm
void cohenSutherlandClip(int x1, int y1, int x2, int y2) {
    int code1 = computeCode(x1, y1);
    int code2 = computeCode(x2, y2);
    bool accept = false;

    while (true) {
        if ((code1 == 0) && (code2 == 0)) {
            accept = true;
            break;
        } else if (code1 & code2) {
            break;
        } else {
            int code_out;
            int x, y;

            if (code1 != 0)
                code_out = code1;
            else
                code_out = code2;

            if (code_out & TOP) {
                x = x1 + (x2 - x1) * (y_max - y1) / (y2 - y1);
```

```
            y = y_max;
        } else if (code_out & BOTTOM) {
            x = x1 + (x2 - x1) * (y_min - y1) / (y2 - y1);
            y = y_min;
        } else if (code_out & RIGHT) {
            y = y1 + (y2 - y1) * (x_max - x1) / (x2 - x1);
            x = x_max;
        } else if (code_out & LEFT) {
            y = y1 + (y2 - y1) * (x_min - x1) / (x2 - x1);
            x = x_min;
        }

        if (code_out == code1) {
            x1 = x;
            y1 = y;
            code1 = computeCode(x1, y1);
        } else {
            x2 = x;
            y2 = y;
            code2 = computeCode(x2, y2);
        }
      }
    }
    if (accept) {
        setcolor(RED);
        line(x1, y1, x2, y2);
    }
}

int main() {
    int gd = X11, gm = X11_1024x768;
    initgraph(&gd, &gm, (char*)"");

    // Define the clipping window
    x_min = 100, y_min = 100;
    x_max = 500, y_max = 400;

    // Draw clipping window
    rectangle(x_min, y_min, x_max, y_max);

    // Draw original line in white
    setcolor(WHITE);
    int x1 = 50, y1 = 150, x2 = 600, y2 = 350;
    int x11 = 120, y12 = 120, x21 = 300, y22 = 250;

    // Clip line and draw clipped portion in red
    cohenSutherlandClip(x1, y1, x2, y2);
    cohenSutherlandClip(x11,y12,x21,y22);

    getch();
    closegraph();
```

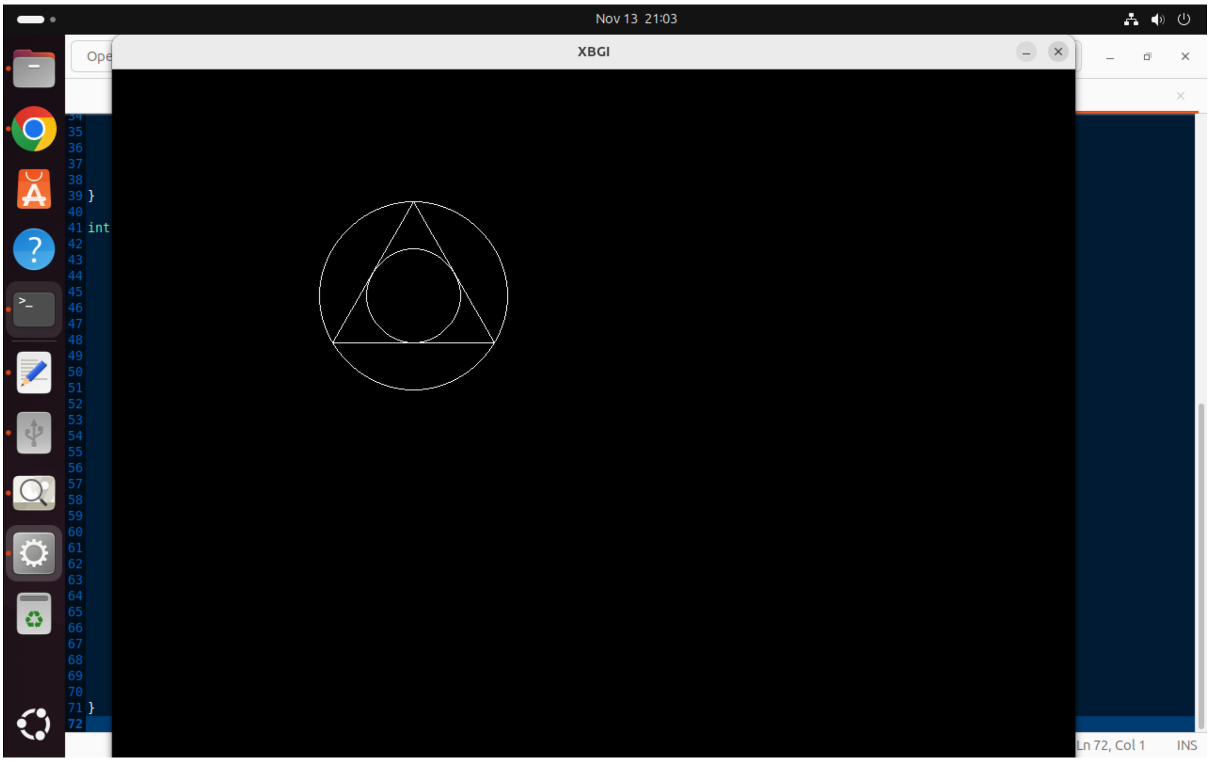
```
    return 0;
}
```



Q3. Write C++ program to draw a given pattern. Use DDA line and Bresenham's circle drawing algorithm. Apply the concept of encapsulation.

```
#include <graphics.h>
#include <cmath>

void drawDDALine(int x1, int y1, int x2, int y2) {
    int dx = x2 - x1;
    int dy = y2 - y1;
    int steps = abs(dx) > abs(dy) ? abs(dx) : abs(dy);
    float xIncrement = dx / (float) steps;
    float yIncrement = dy / (float) steps;
    float x = x1, y = y1;

    for (int i = 0; i <= steps; i++) {
        putpixel(round(x), round(y), WHITE);
        x += xIncrement;
        y += yIncrement;
    }
}

void drawBresenhamCircle(int xc, int yc, int r) {
    int x = 0, y = r;
    int d = 3 - 2 * r;
    while (y >= x) {
        putpixel(xc + x, yc + y, WHITE);
        putpixel(xc - x, yc + y, WHITE);
        putpixel(xc + x, yc - y, WHITE);
        putpixel(xc - x, yc - y, WHITE);
        putpixel(xc + y, yc + x, WHITE);
        putpixel(xc - y, yc + x, WHITE);
        putpixel(xc + y, yc - x, WHITE);
        putpixel(xc - y, yc - x, WHITE);
        x++;
        if (d > 0) {
            y--;
            d = d + 4 * (x - y) + 10;
        } else {
            d = d + 4 * x + 6;
        }
    }
}

int main() {
    int gd = X11, gm = X11_1024x768;
    initgraph(&gd, &gm, (char*)"");

    int xc = 320, yc = 240; // Center of the main circle
    int radius = 100; // Radius of the main circle

    // Draw outer circle using Bresenham's algorithm
    drawBresenhamCircle(xc, yc, radius);
```

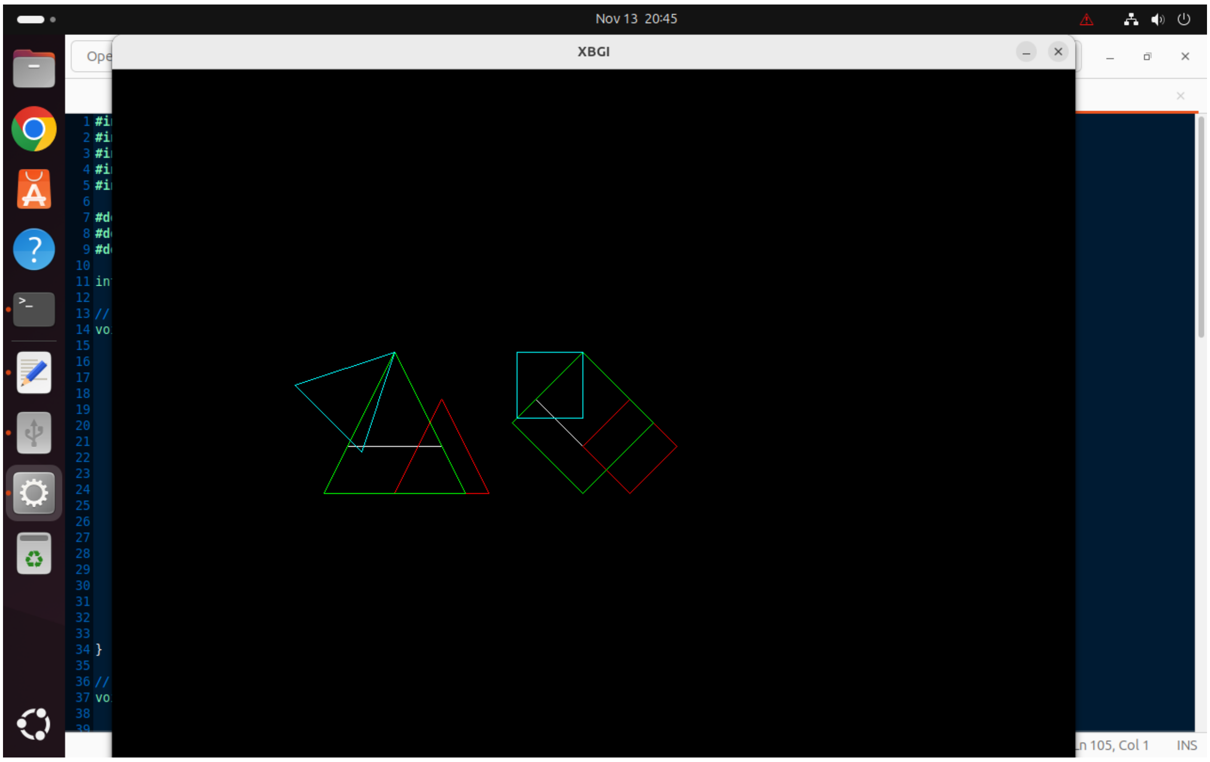
```
    // Calculate vertices of the inscribed equilateral triangle
    int x1 = xc;
    int y1 = yc - radius;
    int x2 = xc + (int)(radius * sin(M_PI / 3));
    int y2 = yc + (int)(radius * 0.5);
    int x3 = xc - (int)(radius * sin(M_PI / 3));
    int y3 = yc + (int)(radius * 0.5);

    // Draw the triangle using DDA line drawing algorithm
    drawDDALine(x1, y1, x2, y2);
    drawDDALine(x2, y2, x3, y3);
    drawDDALine(x3, y3, x1, y1);

    // Draw inner circle using Bresenham's algorithm
    int innerRadius = radius / 2;
    drawBresenhamCircle(xc, yc, innerRadius);

    getch();
    closegraph();
    return 0;
}
```



Q4. Write C++ program to implement translation, rotation and scaling transformations on equilateral triangle and rhombus. Apply the concept of operator overloading.

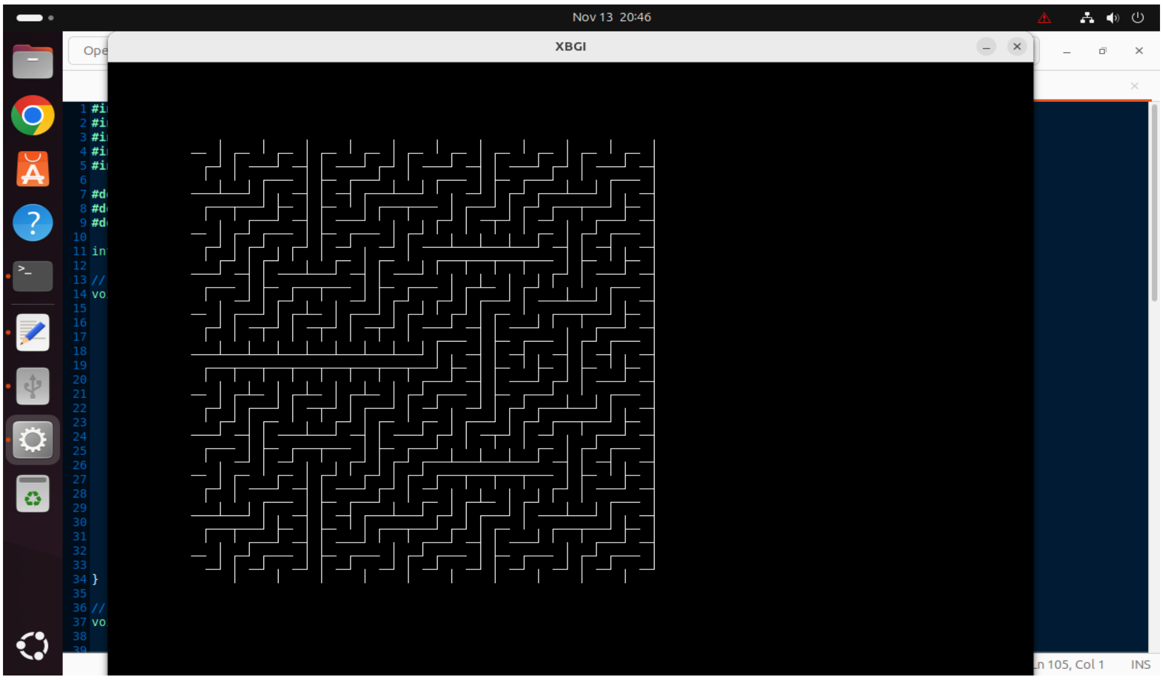
```
#include <graphics.h>
#include <cmath>

const float PI = 3.14159;

class Shape {
protected:
    int points[4][2];
    int n; // Number of vertices

public:
    Shape(int pts[][2], int vertices) {
        n = vertices;
        for (int i = 0; i < n; ++i) {
            points[i][0] = pts[i][0];
            points[i][1] = pts[i][1];
        }
    }

    void draw() {
        for (int i = 0; i < n; ++i) {
            int next = (i + 1) % n;
            line(points[i][0], points[i][1], points[next][0],
points[next][1]);
        }
    }

    Shape operator+(const int translation[2]) {
        Shape translated = *this;
        for (int i = 0; i < n; ++i) {
            translated.points[i][0] += translation[0];
            translated.points[i][1] += translation[1];
        }
        return translated;
    }

    Shape operator*(float scale) {
        Shape scaled = *this;
        int cx = points[0][0], cy = points[0][1];
        for (int i = 0; i < n; ++i) {
            scaled.points[i][0] = cx + (int)((points[i][0] - cx) *
scale);
            scaled.points[i][1] = cy + (int)((points[i][1] - cy) *
scale);
        }
        return scaled;
    }

    Shape operator^(float angle) {
        Shape rotated = *this;
        int cx = points[0][0], cy = points[0][1];
        angle = angle * PI / 180;
        for (int i = 0; i < n; ++i) {
            int x = points[i][0] - cx;
            int y = points[i][1] - cy;
```

```
            rotated.points[i][0] = cx + (int)(x * cos(angle) - y *
sin(angle));
            rotated.points[i][1] = cy + (int)(x * sin(angle) + y *
cos(angle));
        }
        return rotated;
    }
};

int main() {
    int gd = X11, gm = X11_1024x768;
    initgraph(&gd, &gm, (char*)"");

    int trianglePoints[3][2] = {{300, 300}, {350, 400}, {250, 400}};
    int rhombusPoints[4][2] = {{500, 300}, {550, 350}, {500, 400},
{450, 350}};

    Shape triangle(trianglePoints, 3);
    Shape rhombus(rhombusPoints, 4);

    triangle.draw();
    rhombus.draw();

    // Apply transformations and draw
    int translation[2] = {50, 50};
    float scale = 1.5;
    float angle = 45;

    Shape translatedTriangle = triangle + translation;
    Shape scaledTriangle = triangle * scale;
    Shape rotatedTriangle = triangle ^ angle;

    Shape translatedRhombus = rhombus + translation;
    Shape scaledRhombus = rhombus * scale;
    Shape rotatedRhombus = rhombus ^ angle;

    setcolor(RED);
    translatedTriangle.draw();
    translatedRhombus.draw();

    setcolor(GREEN);
    scaledTriangle.draw();
    scaledRhombus.draw();

    setcolor(CYAN);
    rotatedTriangle.draw();
    rotatedRhombus.draw();

    getch();
    closegraph();
    return 0;
}
```


Nov 13 20:45
XBGI
n 105, Col 1
INS

Q5. Write C++ program to generate Hilbert curve using concept of fractals.

```
#include <graphics.h>
#include <iostream>
using namespace std;

// Function to draw the Hilbert Curve
void hilbertCurve(int x, int y, int xi, int xj, int yi, int yj, int n)
{
    if (n <= 0) {
        int x1 = x + (xi + yi) / 2;
        int y1 = y + (xj + yj) / 2;
        int x2 = x + (xi - yi) / 2;
        int y2 = y + (xj - yj) / 2;
        line(x1, y1, x2, y2);
    } else {
        hilbertCurve(x, y, yi / 2, yj / 2, xi / 2, xj / 2, n - 1);
        hilbertCurve(x + xi / 2, y + xj / 2, xi / 2, xj / 2, yi / 2, yj
/ 2, n - 1);
        hilbertCurve(x + xi / 2 + yi / 2, y + xj / 2 + yj / 2, xi / 2,
xj / 2, yi / 2, yj / 2, n - 1);
        hilbertCurve(x + xi / 2 + yi, y + xj / 2 + yj, -yi / 2, -yj /
2, -xi / 2, -xj / 2, n - 1);
    }
}

int main() {
    int gd = X11, gm = X11_1024x768;
    initgraph(&gd, &gm, (char*)"");

    int n = 5; // Depth level of Hilbert curve
    int startX = 100, startY = 100;
    int length = 512;

    hilbertCurve(startX, startY, length, 0, 0, length, n);
    getch();
    closegraph();
    return 0;
}
```

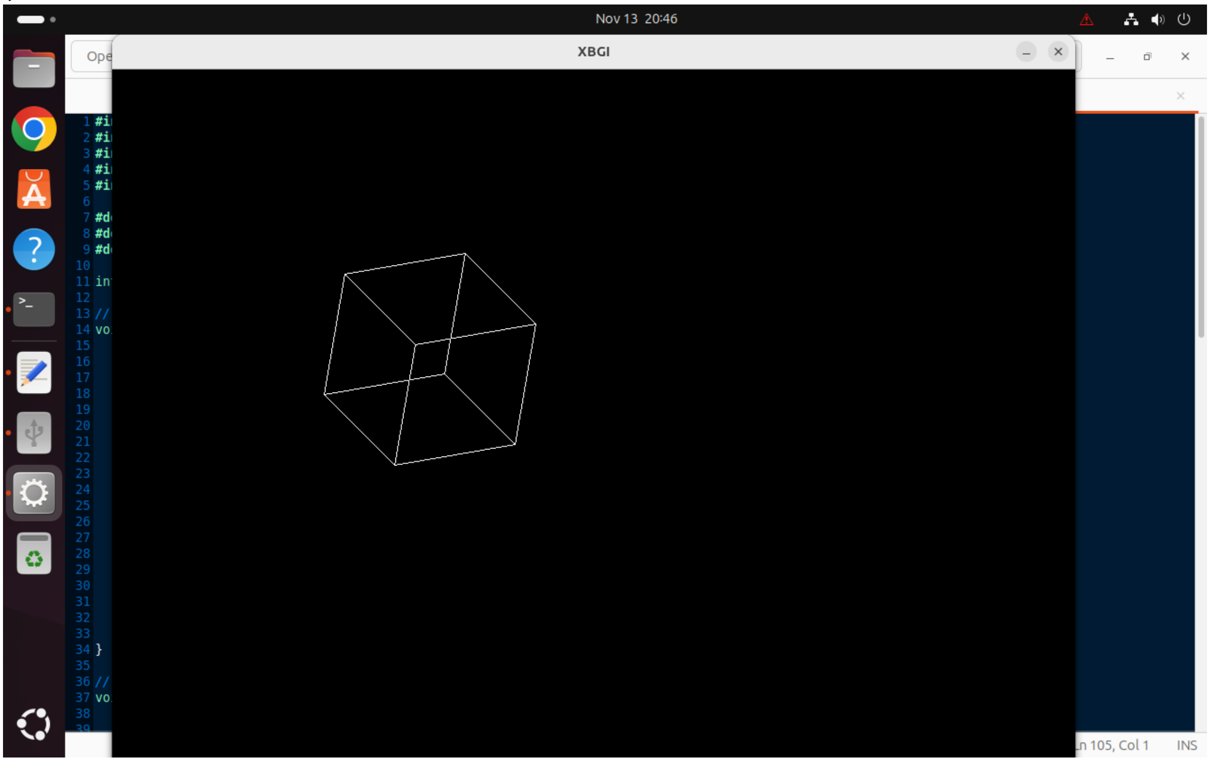

Nov 13 20:46
XBGI
Ln 105, Col 1
INS

Q6. Write C++ program to draw 3-D cube and perform following transformations on it i) Scaling ii) Translation iii) Rotation about an axis(X/Y/Z).

```
#include <graphics.h>
#include <cmath>
#include <iostream>
#include <unistd.h> // Include for sleep function

using namespace std;

struct Point3D {
    float x, y, z;
};

Point3D cube[8]; // Original points of the cube
Point3D transformedCube[8]; // Transformed points of the cube

void initializeCube(float side) {
    float halfSide = side / 2;
    cube[0] = { -halfSide, -halfSide, -halfSide };
    cube[1] = { halfSide, -halfSide, -halfSide };
    cube[2] = { halfSide, halfSide, -halfSide };
    cube[3] = { -halfSide, halfSide, -halfSide };
    cube[4] = { -halfSide, -halfSide, halfSide };
    cube[5] = { halfSide, -halfSide, halfSide };
    cube[6] = { halfSide, halfSide, halfSide };
    cube[7] = { -halfSide, halfSide, halfSide };
}

void projectCube() {
    for (int i = 0; i < 8; i++) {
        transformedCube[i].x = cube[i].x + 320;
        transformedCube[i].y = cube[i].y + 240;
    }
}

void drawCube() {
    for (int i = 0; i < 4; i++) {
        line(transformedCube[i].x, transformedCube[i].y,
transformedCube[(i + 1) % 4].x, transformedCube[(i + 1) % 4].y);
        line(transformedCube[i + 4].x, transformedCube[i + 4].y,
transformedCube[(i + 1) % 4 + 4].x, transformedCube[(i + 1) % 4 +
4].y);
        line(transformedCube[i].x, transformedCube[i].y,
transformedCube[i + 4].x, transformedCube[i + 4].y);
    }
}

void scaleCube(float scaleX, float scaleY, float scaleZ) {
    for (int i = 0; i < 8; i++) {
        cube[i].x *= scaleX;
        cube[i].y *= scaleY;
        cube[i].z *= scaleZ;
    }
}

void translateCube(float tx, float ty, float tz) {
    for (int i = 0; i < 8; i++) {
        cube[i].x += tx;
```

```
        cube[i].y += ty;
        cube[i].z += tz;
    }
}

void rotateCubeX(float angle) {
    float rad = angle * M_PI / 180;
    for (int i = 0; i < 8; i++) {
        float y = cube[i].y;
        float z = cube[i].z;
        cube[i].y = y * cos(rad) - z * sin(rad);
        cube[i].z = y * sin(rad) + z * cos(rad);
    }
}

void rotateCubeY(float angle) {
    float rad = angle * M_PI / 180;
    for (int i = 0; i < 8; i++) {
        float x = cube[i].x;
        float z = cube[i].z;
        cube[i].x = x * cos(rad) + z * sin(rad);
        cube[i].z = -x * sin(rad) + z * cos(rad);
    }
}

void rotateCubeZ(float angle) {
    float rad = angle * M_PI / 180;
    for (int i = 0; i < 8; i++) {
        float x = cube[i].x;
        float y = cube[i].y;
        cube[i].x = x * cos(rad) - y * sin(rad);
        cube[i].y = x * sin(rad) + y * cos(rad);
    }
}

int main() {
    int gd = X11, gm = X11_1024x768;
    initgraph(&gd, &gm, (char*)"");

    initializeCube(100); // Initialize a cube with side length 100
    projectCube();
    drawCube();
    sleep(1); // Replace delay(1000) with sleep(1) for 1-second pause

    cleardevice();
    scaleCube(1.5, 1.5, 1.5); // Scale the cube
    projectCube();
    drawCube();
    sleep(1);

    cleardevice();
    translateCube(50, 50, 0); // Translate the cube
    projectCube();
    drawCube();
    sleep(1);

    cleardevice();
    rotateCubeX(45); // Rotate around X-axis
```

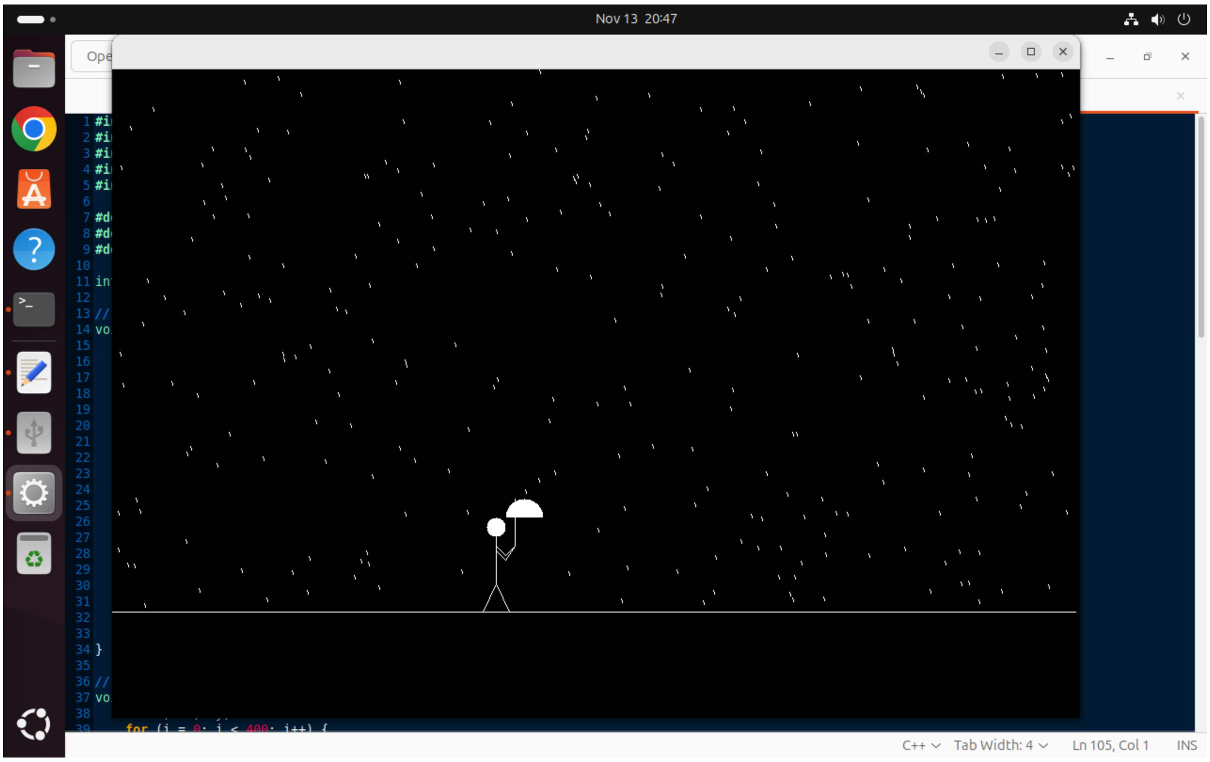
```
    projectCube();
    drawCube();
    sleep(1);

    cleardevice();
    rotateCubeY(45); // Rotate around Y-axis
    projectCube();
    drawCube();
    sleep(1);

    cleardevice();
    rotateCubeZ(45); // Rotate around Z-axis
    projectCube();
    drawCube();
    sleep(1);

    getch();
    closegraph();
    return 0;
}
```



Q7. Write C++ program to draw man walking in the rain with an umbrella. Apply the concept of polymorphism.

```
#include <X11/Xlib.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <math.h>

#define ScreenWidth 1024
#define ScreenHeight 768
#define GroundY (ScreenHeight * 0.75)

int ldisp = 0;

// Function to draw a man with an umbrella
void DrawManAndUmbrella(Display *display, Window win, GC gc, int x, int
ldisp) {
    // Draw head (circle)
    XFillArc(display, win, gc, x - 10, GroundY - 100, 20, 20, 0, 360 *
64); // Circle

    // Draw body (line)
    XDrawLine(display, win, gc, x, GroundY - 80, x, GroundY - 30);

    // Draw arms (lines)
    XDrawLine(display, win, gc, x, GroundY - 70, x + 10, GroundY - 60);
    XDrawLine(display, win, gc, x, GroundY - 65, x + 10, GroundY - 55);
    XDrawLine(display, win, gc, x + 10, GroundY - 60, x + 20, GroundY -
70);
    XDrawLine(display, win, gc, x + 10, GroundY - 55, x + 20, GroundY -
70);

    // Draw legs (lines)
    XDrawLine(display, win, gc, x, GroundY - 30, x + ldisp, GroundY);
    XDrawLine(display, win, gc, x, GroundY - 30, x - ldisp, GroundY);

    // Draw umbrella (pieslice equivalent)
    XFillArc(display, win, gc, x + 10, GroundY - 120, 40, 40, 0, 180 *
64); // Umbrella
    XDrawLine(display, win, gc, x + 20, GroundY - 120, x + 20, GroundY
- 70); // Umbrella stick
}

// Function to simulate rain
void Rain(Display *display, Window win, GC gc, int x) {
    int i, rx, ry;
    for (i = 0; i < 400; i++) {
        rx = rand() % ScreenWidth;
        ry = rand() % ScreenHeight;
        if (ry < GroundY - 4) {
            if (ry < GroundY - 120 || (ry > GroundY - 120 && (rx < x -
20 || rx > x + 60))) {
                XDrawLine(display, win, gc, rx, ry, rx + 1, ry + 4);
            }
        }
    }
}
```

```
// Driver code
int main() {
    int x = ScreenWidth / 5;

    // Initialize graphics window
    Display* display;
    int screen;
    Window win;
    XEvent report;
    display = XOpenDisplay(NULL);
    screen = DefaultScreen(display);

    win = XCreateSimpleWindow(display, RootWindow(display, screen), 0,
0, ScreenWidth, ScreenHeight, 0, 0, 0);
    XSelectInput(display, win, ExposureMask | KeyPressMask);
    XMapWindow(display, win);

    // Create Graphics Context (GC)
    GC gc = XCreateGC(display, win, 0, NULL);
    XSetForeground(display, gc, WhitePixel(display, screen));

    // Execute until any key is pressed
    while (1) {
        // Clear the screen
        XClearWindow(display, win);

        // Draw ground
        XDrawLine(display, win, gc, 0, GroundY, ScreenWidth, GroundY);

        // Simulate rain
        Rain(display, win, gc, x);

        // Update the man's umbrella position
        ldisp = (ldisp + 2) % 20;
        DrawManAndUmbrella(display, win, gc, x, ldisp);

        // Check if a key is pressed
        if (XPending(display) > 0) {
            XNextEvent(display, &report);
            if (report.type == KeyPress) {
                break; // Exit the loop if a key is pressed
            }
        }

        // Update x for animation
        x = (x + 2) % ScreenWidth;

        // Delay to control animation speed
        usleep(20000);  // Sleep for 20 milliseconds
    }

    // Close the display connection
    XCloseDisplay(display);

    return 0;
}
```


Nov 13 20:47
C++
Tab Width: 4
Ln 105, Col 1
INS

README

Important Notice

All the code provided in this project is designed to be executed on Ubuntu only. These codes cannot be run in TurboC++ or any other IDE. Ensure you are working within the Ubuntu environment for successful execution.

Requirements

The project relies on the graphics.h library to function. Follow the steps below to install graphics.h on your Ubuntu system.

1. Open terminal in Ubuntu.

2. Type '**sudo apt update**' and it will ask for password, enter the password and hit enter 3. Then type '**wget https://sourceforge.net/projects/libxbgi/files/xbgi_365-1_amd64.deb**' and then hit enter.

4. After completing step 3, type '**sudo dpkg -i xbgi_365-1_amd64.deb**' and hit enter.

This will step your Ubuntu to run graphics.h files

Other important Terminal commands

1. To compile your program :- '**g++ main.cpp -o main /usr/lib64/libXbgi.a -lX11 -lm -no-pie**' replace main.cpp with your .cpp file and main with name you want to give to .exe file.

2. To run your program:- '**./main**' replace main with name you had given to .exe file.

ALL Codes By Kokate Rushik😉