DDA:

```
#include <stdio.h>
#include <math.h>
#include <graphics.h>

// Function to implement DDA algorithm
void drawLine(int x0, int y0, int x1, int y1)
{
    int dx = x1 - x0;
    int dy = y1 - y0;

    int steps = abs(dx) > abs(dy) ? abs(dx) : abs(dy);

    float xIncrement = dx / (float) steps;
    float yIncrement = dy / (float) steps;

    float x = x0;
    float y = y0;

    // Put the first point
    putpixel(round(x), round(y), WHITE);

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

int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);

    int x0, y0, x1, y1;

    // Taking user input for start and end points
    printf("Enter the starting coordinates (x0, y0): ");
    scanf("%d %d", &x0, &y0);

    printf("Enter the ending coordinates (x1, y1): ");
    scanf("%d %d", &x1, &y1);

    // Draw the line using DDA
```

```
    drawLine(x0, y0, x1, y1);

    // Keep the graphics window open until a key is pressed
    getch();
    closegraph();

    return 0;
}
```

Bresenham's Algo:

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

void bresenham(int x0, int y0, int x1, int y1)
{
    int dx = abs(x1 - x0);
    int dy = abs(y1 - y0);
    int sx = (x0 < x1) ? 1 : -1;
    int sy = (y0 < y1) ? 1 : -1;

    int err = dx - dy;

    while (1) {
        putpixel(x0, y0, WHITE);  // Plot the pixel at (x0, y0)

        if (x0 == x1 && y0 == y1)  // If we reach the end point
            break;

        int e2 = 2 * err;

        if (e2 > -dy) {
            err -= dy;
            x0 += sx;
        }

        if (e2 < dx) {
            err += dx;
            y0 += sy;
        }
    }
}
```

```
int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);

    int x0, y0, x1, y1;

    // Taking user input for start and end points
    printf("Enter the starting coordinates (x0, y0): ");
    scanf("%d %d", &x0, &y0);

    printf("Enter the ending coordinates (x1, y1): ");
    scanf("%d %d", &x1, &y1);

    // Draw the line using Bresenham's algorithm
    bresenham(x0, y0, x1, y1);

    // Keep the graphics window open until a key is pressed
    getch();
    closegraph();

    return 0;
}
```

Midpoint circle Algorithm:

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

// Function to plot points in all octants using circle symmetry
void plotCirclePoints(int xc, int yc, int x, int y)
{
    putpixel(xc + x, yc + y, WHITE);  // 1st Octant
    putpixel(xc - x, yc + y, WHITE);  // 2nd Octant
    putpixel(xc + x, yc - y, WHITE);  // 3rd Octant
    putpixel(xc - x, yc - y, WHITE);  // 4th Octant
    putpixel(xc + y, yc + x, WHITE);  // 5th Octant
    putpixel(xc - y, yc + x, WHITE);  // 6th Octant
    putpixel(xc + y, yc - x, WHITE);  // 7th Octant
    putpixel(xc - y, yc - x, WHITE);  // 8th Octant
}

// Midpoint Circle Drawing Algorithm
void midpointCircle(int xc, int yc, int r)
{
```

```c
    int x = 0;
    int y = r;
    int p = 1 - r;  // Initial decision parameter

    // Plot the initial point
    plotCirclePoints(xc, yc, x, y);

    while (x < y) {
        x++;

        // Decision parameter update
        if (p < 0) {
            p += 2 * x + 1;
        } else {
            y--;
            p += 2 * (x - y) + 1;
        }

        plotCirclePoints(xc, yc, x, y);
    }
}

int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);

    int xc, yc, r;

    // Taking user input for center and radius of the circle
    printf("Enter the center coordinates (xc, yc): ");
    scanf("%d %d", &xc, &yc);

    printf("Enter the radius of the circle: ");
    scanf("%d", &r);

    // Draw the circle using Midpoint algorithm
    midpointCircle(xc, yc, r);

    // Keep the graphics window open until a key is pressed
    getch();
    closegraph();

    return 0;
```

```
}
```

Boundary Fill:

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

// 8-connected Boundary Fill
void boundaryFill8(int x, int y, int fill_color, int boundary_color)
{
    int current_color = getpixel(x, y);

    if (current_color != boundary_color && current_color != fill_color)
    {
        putpixel(x, y, fill_color);
        delay(1);  // To visualize the filling process

        // Recursively fill the adjacent pixels
        boundaryFill8(x + 1, y, fill_color, boundary_color);  // Right
        boundaryFill8(x - 1, y, fill_color, boundary_color);  // Left
        boundaryFill8(x, y + 1, fill_color, boundary_color);  // Down
        boundaryFill8(x, y - 1, fill_color, boundary_color);  // Up
        boundaryFill8(x + 1, y + 1, fill_color, boundary_color);  // Bottom-right diagonal
        boundaryFill8(x - 1, y - 1, fill_color, boundary_color);  // Top-left diagonal
        boundaryFill8(x + 1, y - 1, fill_color, boundary_color);  // Top-right diagonal
        boundaryFill8(x - 1, y + 1, fill_color, boundary_color);  // Bottom-left diagonal
    }
}

int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);

    // Draw a rectangle boundary
    rectangle(150, 100, 350, 300);

    // Perform 8-connected boundary fill
    boundaryFill8(200, 150, GREEN, WHITE);

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

Flood Fill :

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

// 8-connected Flood Fill
void floodFill8(int x, int y, int fill_color, int old_color)
{
    int current_color = getpixel(x, y);

    if (current_color == old_color)
    {
        putpixel(x, y, fill_color);
        delay(1);  // To visualize the filling process

        // Recursively fill the adjacent pixels
        floodFill8(x + 1, y, fill_color, old_color);  // Right
        floodFill8(x - 1, y, fill_color, old_color);  // Left
        floodFill8(x, y + 1, fill_color, old_color);  // Down
        floodFill8(x, y - 1, fill_color, old_color);  // Up
        floodFill8(x + 1, y + 1, fill_color, old_color);  // Bottom-right diagonal
        floodFill8(x - 1, y - 1, fill_color, old_color);  // Top-left diagonal
        floodFill8(x + 1, y - 1, fill_color, old_color);  // Top-right diagonal
        floodFill8(x - 1, y + 1, fill_color, old_color);  // Bottom-left diagonal
    }
}

int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);

    // Draw a rectangle with white color inside
    rectangle(150, 100, 350, 300);
    floodFill8(200, 150, YELLOW, BLACK);

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

2d Transformation:

```
#include <stdio.h>
#include <graphics.h>
#include <math.h>
```

```
#define PI 3.14159265

// Function to perform translation
void translate(int *x, int *y, int tx, int ty)
{
    *x += tx;
    *y += ty;
}

// Function to perform scaling
void scale(int *x, int *y, int sx, int sy)
{
    *x *= sx;
    *y *= sy;
}

// Function to perform rotation
void rotate(int *x, int *y, int angle, int xc, int yc)
{
    float radian = angle * (PI / 180);
    int x_new = xc + (*x - xc) * cos(radian) - (*y - yc) * sin(radian);
    int y_new = yc + (*x - xc) * sin(radian) + (*y - yc) * cos(radian);
    *x = x_new;
    *y = y_new;
}

// Function to draw a triangle
void drawTriangle(int x1, int y1, int x2, int y2, int x3, int y3)
{
    line(x1, y1, x2, y2);
    line(x2, y2, x3, y3);
    line(x3, y3, x1, y1);
}

int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);

    int choice, tx, ty, sx, sy, angle;
    int x1 = 100, y1 = 100, x2 = 150, y2 = 50, x3 = 200, y3 = 100;  // Triangle vertices

    // Original Triangle
```

```
setcolor(WHITE);
drawTriangle(x1, y1, x2, y2, x3, y3);
printf("Original Triangle drawn.\n");

// User input for the type of transformation
printf("Choose a transformation:\n");
printf("1. Translation\n");
printf("2. Scaling\n");
printf("3. Rotation\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice)
{
  case 1:
    // Translation
    printf("Enter translation factors (tx, ty): ");
    scanf("%d %d", &tx, &ty);

    // Translate the triangle
    translate(&x1, &y1, tx, ty);
    translate(&x2, &y2, tx, ty);
    translate(&x3, &y3, tx, ty);

    setcolor(RED);
    drawTriangle(x1, y1, x2, y2, x3, y3);
    printf("Triangle translated.\n");
    break;

  case 2:
    // Scaling
    printf("Enter scaling factors (sx, sy): ");
    scanf("%d %d", &sx, &sy);

    // Scale the triangle
    scale(&x1, &y1, sx, sy);
    scale(&x2, &y2, sx, sy);
    scale(&x3, &y3, sx, sy);

    setcolor(GREEN);
    drawTriangle(x1, y1, x2, y2, x3, y3);
    printf("Triangle scaled.\n");
    break;
```

```
        case 3:
            // Rotation
            printf("Enter the angle of rotation (in degrees): ");
            scanf("%d", &angle);

            // Rotate the triangle around its center (x2, y2)
            rotate(&x1, &y1, angle, x2, y2);
            rotate(&x3, &y3, angle, x2, y2);

            setcolor(BLUE);
            drawTriangle(x1, y1, x2, y2, x3, y3);
            printf("Triangle rotated.\n");
            break;

        default:
            printf("Invalid choice!\n");
            break;
    }

    getch();
    closegraph();

    return 0;
}
```

Scaling & Sheering:

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

#define MAX_POINTS 3

// Function to perform shearing
void shear(int *x, int *y, float shx, float shy)
{
    int newX = *x + (shx * *y);
    int newY = *y + (shy * *x);
    *x = newX;
    *y = newY;
}

// Function to perform reflection
void reflect(int *x, int *y, char axis)
{
```

```
    if (axis == 'x') {
        *y = -*y; // Reflection over x-axis
    } else if (axis == 'y') {
        *x = -*x; // Reflection over y-axis
    } else if (axis == 'xy') {
        *x = -*x; // Reflection over y-axis
        *y = -*y; // Reflection over x-axis
    }
}

// Function to draw a triangle
void drawTriangle(int x1, int y1, int x2, int y2, int x3, int y3)
{
    line(x1, y1, x2, y2);
    line(x2, y2, x3, y3);
    line(x3, y3, x1, y1);
}

int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);

    int x[MAX_POINTS] = {100, 150, 200}; // Triangle x-coordinates
    int y[MAX_POINTS] = {100, 50, 100};  // Triangle y-coordinates

    // Original Triangle
    setcolor(WHITE);
    drawTriangle(x[0], y[0], x[1], y[1], x[2], y[2]);
    printf("Original Triangle drawn.\n");

    // Shearing transformation
    float shx, shy;
    printf("Enter shearing factors (shx, shy): ");
    scanf("%f %f", &shx, &shy);

    // Apply shearing transformation
    for (int i = 0; i < MAX_POINTS; i++)
    {
        shear(&x[i], &y[i], shx, shy);
    }

    setcolor(RED);
    drawTriangle(x[0], y[0], x[1], y[1], x[2], y[2]);
```

```
    printf("Triangle sheared.\n");

    // Reflection transformation
    char axis;
    printf("Enter axis for reflection (x, y, or xy): ");
    scanf(" %c", &axis);

    // Apply reflection transformation
    for (int i = 0; i < MAX_POINTS; i++)
    {
        reflect(&x[i], &y[i], axis);
    }

    setcolor(GREEN);
    drawTriangle(x[0], y[0], x[1], y[1], x[2], y[2]);
    printf("Triangle reflected.\n");

    getch();
    closegraph();

    return 0;
}
```

Cohen sutherland algorithm:

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

// Define the region codes
#define INSIDE 0 // 0000
#define LEFT 1   // 0001
#define RIGHT 2  // 0010
#define BOTTOM 4  // 0100
#define TOP 8    // 1000

// Function to compute the region code for a point
int computeCode(float x, float y, float xmin, float ymin, float xmax, float ymax) {
    int code = INSIDE;

    if (x < xmin)       // to the left of the clipping window
        code |= LEFT;
    else if (x > xmax)  // to the right of the clipping window
        code |= RIGHT;
    if (y < ymin)       // below the clipping window
        code |= BOTTOM;
```

```
    else if (y > ymax)  // above the clipping window
        code |= TOP;

    return code;
}

// Cohen-Sutherland line clipping algorithm
void cohenSutherlandLineClip(float x1, float y1, float x2, float y2, float xmin, float ymin, float
xmax, float ymax) {
    int code1 = computeCode(x1, y1, xmin, ymin, xmax, ymax);
    int code2 = computeCode(x2, y2, xmin, ymin, xmax, ymax);
    int accept = 0;

    while (1) {
        if (!(code1 | code2)) {
            // Both points inside the clipping window
            accept = 1;
            break;
        } else if (code1 & code2) {
            // Both points are outside the clipping window
            break;
        } else {
            int codeOut;
            float x, y;

            // Choose one of the points outside the clipping window
            if (code1 != INSIDE) {
                codeOut = code1;
            } else {
                codeOut = code2;
            }

            // Find the intersection point
            if (codeOut & TOP) {
                // Point is above the clipping window
                x = x1 + (x2 - x1) * (ymax - y1) / (y2 - y1);
                y = ymax;
            } else if (codeOut & BOTTOM) {
                // Point is below the clipping window
                x = x1 + (x2 - x1) * (ymin - y1) / (y2 - y1);
                y = ymin;
            } else if (codeOut & RIGHT) {
                // Point is to the right of the clipping window
                y = y1 + (y2 - y1) * (xmax - x1) / (x2 - x1);
```

```
            x = xmax;
        } else if (codeOut & LEFT) {
            // Point is to the left of the clipping window
            y = y1 + (y2 - y1) * (xmin - x1) / (x2 - x1);
            x = xmin;
        }

        // Now we move the outside point to the intersection point
        if (codeOut == code1) {
            x1 = x;
            y1 = y;
            code1 = computeCode(x1, y1, xmin, ymin, xmax, ymax);
        } else {
            x2 = x;
            y2 = y;
            code2 = computeCode(x2, y2, xmin, ymin, xmax, ymax);
        }
    }
  }

  if (accept) {
    // Draw the line after clipping
    line(x1, y1, x2, y2);
  }
}

int main() {
  int gd = DETECT, gm;
  initgraph(&gd, &gm, NULL);

  float xmin = 200, ymin = 200, xmax = 400, ymax = 400; // Clipping window
  rectangle(xmin, ymin, xmax, ymax); // Draw clipping window

  float x1 = 150, y1 = 250; // Start point of the line
  float x2 = 450, y2 = 250; // End point of the line
  setcolor(WHITE);
  line(x1, y1, x2, y2); // Draw the original line

  setcolor(RED);
  cohenSutherlandLineClip(x1, y1, x2, y2, xmin, ymin, xmax, ymax); // Clip the line

  getch();
  closegraph();
```

```
    return 0;
}
```

Beizer Curve:

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

#define MAX_POINTS 10 // Maximum number of control points

// Function to compute the Bezier curve points using de Casteljau's algorithm
void bezierCurve(int controlPoints[][2], int n, int steps) {
    float t, x, y;
    for (int i = 0; i <= steps; i++) {
        t = (float)i / (float)steps;

        // Create a temporary array to hold the points for the current iteration
        int temp[MAX_POINTS][2];

        // Copy control points to temp array
        for (int j = 0; j < n; j++) {
            temp[j][0] = controlPoints[j][0];
            temp[j][1] = controlPoints[j][1];
        }

        // Perform de Casteljau's algorithm
        for (int j = 1; j < n; j++) {
            for (int k = 0; k < n - j; k++) {
                temp[k][0] = (1 - t) * temp[k][0] + t * temp[k + 1][0];
                temp[k][1] = (1 - t) * temp[k][1] + t * temp[k + 1][1];
            }
        }

        // Draw the point on the Bezier curve
        putpixel((int)temp[0][0], (int)temp[0][1], WHITE);
    }
}

// Function to draw the control points and lines
void drawControlPolygon(int controlPoints[][2], int n) {
    for (int i = 0; i < n; i++) {
        putpixel(controlPoints[i][0], controlPoints[i][1], YELLOW);
        if (i > 0) {
            line(controlPoints[i - 1][0], controlPoints[i - 1][1], controlPoints[i][0], controlPoints[i][1]);
        }
```

```
    }
}

int main() {
    int gd = DETECT, gm;
    initgraph(&gd, &gm, NULL);

    int controlPoints[MAX_POINTS][2];
    int n;

    printf("Enter the number of control points (max %d): ", MAX_POINTS);
    scanf("%d", &n);

    // Input control points
    for (int i = 0; i < n; i++) {
        printf("Enter coordinates for control point %d (x y): ", i + 1);
        scanf("%d %d", &controlPoints[i][0], &controlPoints[i][1]);
    }

    // Draw control polygon
    drawControlPolygon(controlPoints, n);

    // Draw Bezier curve
    bezierCurve(controlPoints, n, 1000); // 1000 steps for smooth curve

    getch();
    closegraph();

    return 0;
}
```