Department of CSE
SSN College of Engineering

Vishakan Subramanian - 18 5001 196 - Semester VII

05 September 2021

UCS 1712 - Graphics And Multimedia Lab

Exercise 6: 2D Composite Transformations and Windowing in C++ using OpenGL

Aim:

- To compute the composite transformation matrix for any 2 transformations given as input by the user and applying it on the object. The transformation can be any combination of the following:
  - Translation
  - Rotation
  - Scaling
  - Reflection
  - Shearing

  Display the original and the transformed object. Calculate the final transformation matrix by multiplying the two individual transformation matrices and then apply it to the object.

  Note: Use Homogeneous coordinate representations and matrix multiplication to perform transformations. Divide the output window into four quadrants. (Use LINES primitive to draw x and y axis)

- Create a window with any 2D object and a different sized viewport. Apply window to viewport transformation on the object. Display both window and viewport.

Code: 2D Composite Transformations:

```
/*
To compute the composite transformation matrix for any 2 transformations
    given as input by
the user and applying it on the object. The transformation can be any
    combination of the following:
Translation, Rotation, Scaling, Reflection & Shearing.
*/

#include <stdio.h>
#include <math.h>
#include <GL/glut.h>
#include <iostream>     //for cin, cout
#include <cstring>      //for memcpy

using namespace std;

const int WINDOW_WIDTH = 800;
const int WINDOW_HEIGHT = 800;
const int X_MIN = -400;
const int X_MAX = 400;
const int Y_MIN = -400;
const int Y_MAX = 400;
const int FPS = 60;

enum Axes {xAxis = 0, yAxis = 1};
enum Lines {XAxis = 0, YAxis = 1, Origin = 2, XEqualsY = 3};
enum Transforms {Translation = 1, Rotation = 2, Scaling = 3, Reflection =
    4, Shearing = 5};

class Point{
private:
    GLdouble x, y, h;

public:
    Point(){
        x = y = 0;
        h = 1;
    }

    Point(GLint xCoord, GLint yCoord){
        x = xCoord;
        y = yCoord;
        h = 1;
    }

    Point(GLint xCoord, GLint yCoord, GLint H){
        x = xCoord;
```

```cpp
        y = yCoord;
        h = H;
    }

    void setCoords(GLdouble xCoord, GLdouble yCoord){
        x = xCoord;
        y = yCoord;
    }

    void setHomogeneousCoords(GLdouble xCoord, GLdouble yCoord, GLdouble H){
        x = xCoord;
        y = yCoord;
        h = H;
    }

    GLdouble getX() const{
        return x;
    }

    GLdouble getY() const{
        return y;
    }

    GLdouble getH() const{
        return h;
    }

    GLdouble getHomogenousX() const{
        return x * h;

    }

    GLdouble getHomogenousY() const{
        return y * h;
    }
};


class PolygonShape{
private:
    int numVertices;
    Point *points;
    bool matrix1Flag, matrix2Flag;
    double matrix1[3][3], matrix2[3][3], compositeMatrix[3][3];

public:
    PolygonShape(){
        numVertices = 0;
        matrix1Flag = false;
        matrix2Flag = false;
    }

    PolygonShape(int noVertices){
        numVertices = noVertices;
        points = new Point[numVertices];
        matrix1Flag = false;
        matrix2Flag = false;
    }

    int getVertexCount() const{
        return numVertices;
    }

    Point getPoint(int i){
        return points[i];
    }

    void setVertices(int noVertices){
        numVertices = noVertices;
        points = new Point[numVertices];
    }

    void setPoint(int i, GLdouble x, GLdouble y, GLdouble h = 1){
        points[i].setHomogeneousCoords(x, y, h);
    }

    void clearMatrices(){
       //Clear the transformation matrices to identity matrices

        matrix1Flag = false;
        matrix2Flag = false;

        for(int i = 0; i < 3; i++){
            for(int j = 0; j < 3; j++){
                if(i == j){
                    matrix1[i][j] = 1;
                    matrix2[i][j] = 1;
                } else{
                    matrix1[i][j] = 0;
                    matrix2[i][j] = 0;
                }
            }
        }
    }

    void setTranslationMatrix(Point translationVector){
        //Sets a translation matrix to one of the transformation matrices

        double translationMatrix[3][3] = {  {1, 0, translationVector.
   getHomogenousX()},
                                            {0, 1, translationVector.getHomogenousY()},
                                            {0, 0, 1}};

        if(!matrix1Flag){
            memcpy(matrix1, translationMatrix, sizeof(translationMatrix));
            matrix1Flag = true;
        } else{
            memcpy(matrix2, translationMatrix, sizeof(translationMatrix));
            matrix2Flag = true;
        }
    }

    void setRotationMatrix(int rotationAngle, Point pivot = Point(0, 0, 1)){
        //Sets a rotation matrix to one of the transformation matrices

        double rotationAngleInRadians = rotationAngle * 3.14159/180;
        double cosAngle = cos(rotationAngleInRadians);
        double sinAngle = sin(rotationAngleInRadians);

        double xPivotValue = (pivot.getX() * (1 - cosAngle)) + (pivot.getY() * sinAngle);
        double yPivotValue = (pivot.getY() * (1 - cosAngle)) - (pivot.getX() * sinAngle);

        double rotationMatrix[3][3] = { {cosAngle, -sinAngle, xPivotValue},
                                        {sinAngle, cosAngle, yPivotValue},
                                        {0, 0, 1}};

        if(!matrix1Flag){
            memcpy(matrix1, rotationMatrix, sizeof(rotationMatrix));
            matrix1Flag = true;
        } else{
            memcpy(matrix2, rotationMatrix, sizeof(rotationMatrix));
            matrix2Flag = true;
        }
    }

    void setReflectionMatrix(int line){
        //Sets a reflection matrix to one of the transformation matrices

        double reflectionMatrix[3][3];

        switch(line){
            case XAxis:{
                double temp[3][3] = {   {1, 0, 0},
                                        {0, -1, 0},
                                        {0, 0, 1}};

```

```cpp
            memcpy(reflectionMatrix, temp, sizeof(temp));
            break;
        }


        case YAxis:{
            double temp[3][3] = {    {-1, 0, 0},
                                     {0, 1, 0},
                                     {0, 0, 1}};

            memcpy(reflectionMatrix, temp, sizeof(temp));
            break;
        }


        case Origin:{
            double temp[3][3] = {    {-1, 0, 0},
                                     {0, -1, 0},
                                     {0, 0, 1}};

            memcpy(reflectionMatrix, temp, sizeof(temp));
            break;
        }


        case XEqualsY:{
            double temp[3][3] = {    {0, 1, 0},
                                     {1, 0, 0},
                                     {0, 0, 1}};

            memcpy(reflectionMatrix, temp, sizeof(temp));
            break;
        }
    }

    if(!matrix1Flag){
        memcpy(matrix1, reflectionMatrix, sizeof(reflectionMatrix));
        matrix1Flag = true;
    } else{
        memcpy(matrix2, reflectionMatrix, sizeof(reflectionMatrix));
        matrix2Flag = true;
    }
}

void setScaleMatrix(double ScaleX, double ScaleY, Point fixed = Point(0, 0, 1)){
    //Sets a scale matrix to one of the transformation matrices

    double xFixedValue = fixed.getX() * (1 - ScaleX);
    double yFixedValue = fixed.getY() * (1 - ScaleY);

    double scaleMatrix[3][3] = {  {ScaleX, 0, xFixedValue},
                                  {0, ScaleY, yFixedValue},
                                  {0, 0, 1}};

    if(!matrix1Flag){
        memcpy(matrix1, scaleMatrix, sizeof(scaleMatrix));
        matrix1Flag = true;
    } else{
        memcpy(matrix2, scaleMatrix, sizeof(scaleMatrix));
        matrix2Flag = true;
    }
}

void setShearMatrix(double shearParam, int axis, double refConst = 0){
    //Sets a shear matrix to one of the transformation matrices

    double shearMatrix[3][3];

    switch(axis){
        case xAxis:{
            double temp[3][3] = {   {1, shearParam, -shearParam *
refConst},
                                    {0, 1, 0},
                                    {0, 0, 1}};

            memcpy(shearMatrix, temp, sizeof(temp));
            break;
        }


        case yAxis:{
            double temp[3][3] = {   {1, 0, -shearParam * refConst},
                                    {shearParam, 1, 0},
                                    {0, 0, 1}};

            memcpy(shearMatrix, temp, sizeof(temp));
            break;
        }
    }

    if(!matrix1Flag){
        memcpy(matrix1, shearMatrix, sizeof(shearMatrix));
        matrix1Flag = true;
    } else{
        memcpy(matrix2, shearMatrix, sizeof(shearMatrix));
        matrix2Flag = true;
    }
}

void setCompositeMatrix(){
    //Sets the composite matrix based on matrix multiplication
```

```
        //of the two transformation matrices

        if(!matrix1Flag || !matrix2Flag){
            //if any one matrix is not set, don't multiply
            return;
        }

        for(int i = 0; i < 3; i++){
            for(int j = 0; j < 3; j++){
                double tempSum = 0;

                for(int k = 0; k < 3; k++){
                    tempSum += matrix1[i][k] * matrix2[k][j];
                }

                compositeMatrix[i][j] = tempSum;
            }
        }
    }

    PolygonShape getTransformedPolygon(){
        //Obtain the transformed polygon based upon the composite transformation

        PolygonShape polyDash(numVertices);
        double values[3];

        for(int i = 0; i < numVertices; i++){
            Point p = getPoint(i);

            //[3 x 3] x [3 x 1] = [3 x 1] matrix
            for(int j = 0; j < 3; j++){
                values[j] = compositeMatrix[j][0] * p.getHomogenousX() +
                            compositeMatrix[j][1] * p.getHomogenousY() +
                            compositeMatrix[j][2] * p.getH();
            }

            polyDash.setPoint(i, values[0]/p.getH(), values[1]/p.getH(), values[2]);
        }

        return polyDash;
    }
};


void initializeDisplay();
void plotComponents();
void dummyFunction();
void renderContents();
void mainLoop(int val);
void setTransformMatrices();
void plotTransformation();
void drawAxes();
void drawPolygon(PolygonShape polygon, bool transformed = false);

PolygonShape polygon;                   //Global PolygonShape object to be plotted on the graph
int transform1 = 0, transform2 = 0;     //Global variable to keep track of chosen transformation

int main(int argc, char **argv){
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowPosition(0, 0);
    glutInitWindowSize(WINDOW_WIDTH, WINDOW_HEIGHT);
    glutCreateWindow("2D Composite Transformations - Examples");

    printf("\n------[2D COMPOSITE TRANSFORMATIONS]------\n");
    printf("\nUsage:\tSelect the required transformations in the console.");
    printf("\n\tEnter the parameters for the specified transformations.");
    printf("\n\tView the output in the GLUT window.");
    printf("\n\n------[2D COMPOSITE TRANSFORMATIONS]------\n\n");

    //Set the initial default polygon for the graph
    polygon.setVertices(4);
    polygon.setPoint(0, 0, 0);
    polygon.setPoint(1, 0, 50);
    polygon.setPoint(2, 100, 50);
    polygon.setPoint(3, 100, 0);

    initializeDisplay();

    glutDisplayFunc(dummyFunction);

    //important - to refresh screen periodically
    glutTimerFunc(1000/FPS, mainLoop, 0);

    glutMainLoop();

    return 1;
}

void initializeDisplay(){
    //Initialize the display parameters

    glClearColor(1, 1, 1, 0);
    glMatrixMode(GL_PROJECTION);
    gluOrtho2D(X_MIN, X_MAX, Y_MIN, Y_MAX);
    glClear(GL_COLOR_BUFFER_BIT);   //Clear the display window

    glEnable(GL_BLEND);     //enable blending (translucent colors)
    glDepthMask(GL_FALSE);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);  //set the blend function for translucency
}

void plotComponents(){
    //Plot the axes and the base polygon

    glClear(GL_COLOR_BUFFER_BIT);   //Clear the display window
    drawAxes();
    drawPolygon(polygon);
    glFlush();
}

void dummyFunction(){
    //Placeholder function to be called in glutDisplayFunc
}

void mainLoop(int val){
    //Function to be called within the glutTimerFunc periodically to refresh screen at 60FPS
    renderContents();
}


void renderContents(){
    //to render the graph along with a user-defined composite transformation

    plotComponents();

    while(true){
        //Await user input

        cout << "\nChoose Transformation 1: " << endl;
        cout << "\t1 for Translation" << endl;
        cout << "\t2 for Rotation" << endl;
        cout << "\t3 for Scaling" << endl;
        cout << "\t4 for Reflection" << endl;
        cout << "\t5 for Shearing" << endl;
        cout << "\t0 to Exit" << endl;
        cout << "\tYour Option -> ";
        cin >> transform1;

        if(!transform1){    //user chooses to exit
            exit(0);
        }

        cout << "\nChoose Transformation 2: " << endl;
        cout << "\t1 for Translation" << endl;
```

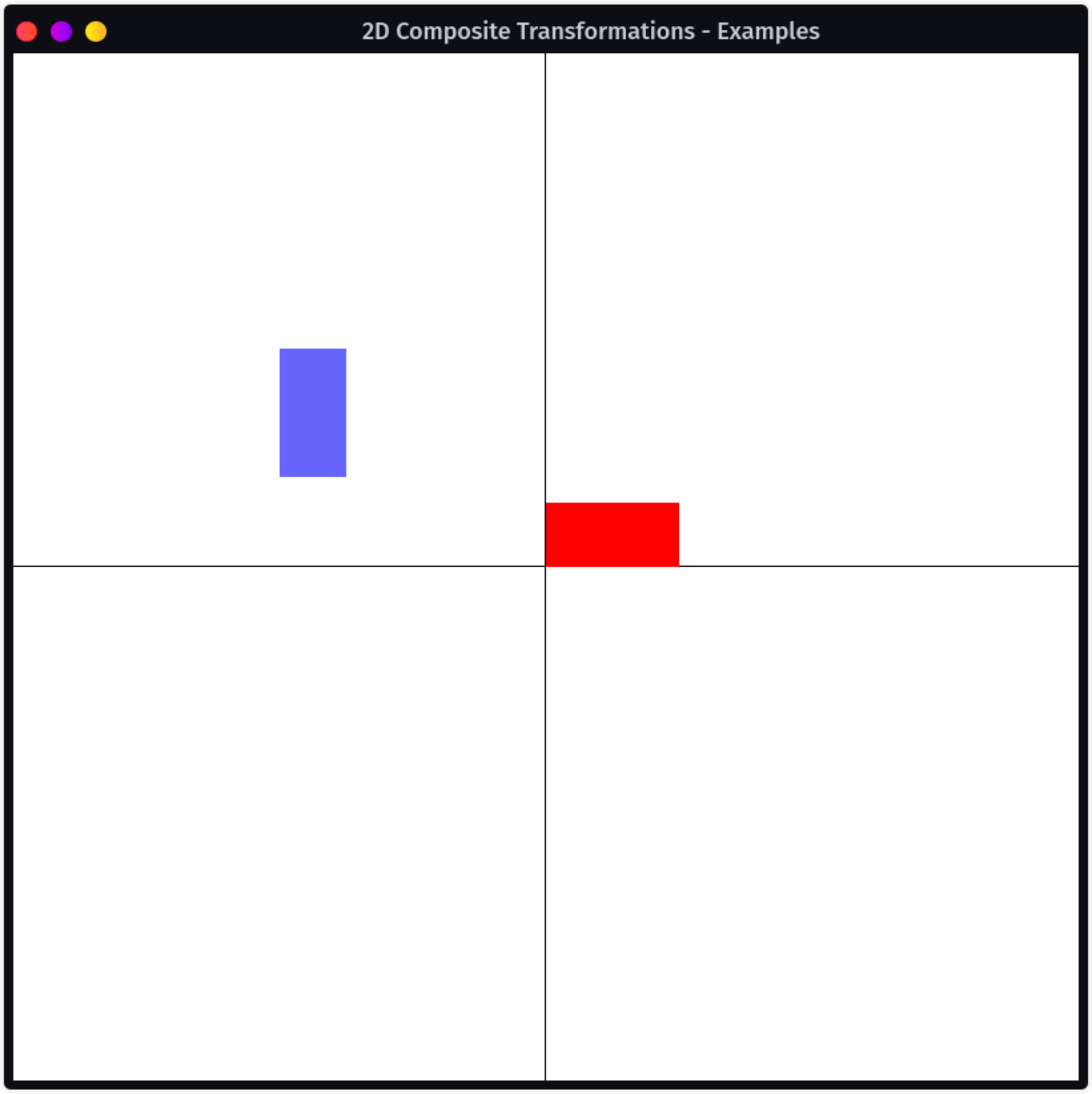
```cpp
        cout << "\t2 for Rotation" << endl;
        cout << "\t3 for Scaling" << endl;
        cout << "\t4 for Reflection" << endl;
        cout << "\t5 for Shearing" << endl;
        cout << "\tYour Option -> ";
        cin >> transform2;

        plotComponents();
        polygon.clearMatrices();        //clear previous transformations
        setTransformMatrices();         //set the new transform matrices
        polygon.setCompositeMatrix();   //multiply to form composite matrix

        PolygonShape polygonDash;
        polygonDash = polygon.getTransformedPolygon();

        //To print the transformed polygon's coordinates
        // for(int i = 0; i < polygonDash.getVertexCount(); i++){
        //     Point pDash = polygonDash.getPoint(i);
        //     cout << "(" << pDash.getX() << ", " << pDash.getY() << ")" << endl;
        // }

        drawPolygon(polygonDash, true);
        glFlush();
        //glutPostRedisplay();
    }
}

void setTransformMatrices(){
    int i = 0;

    while(i < 2){
        int currentTransform = (i == 0) ? transform1 : transform2;
        i++;

        switch(currentTransform){
            case Translation:{
                double x, y;
                cout << "\n\n------[TRANSLATION]------" << endl;
                cout << "\n\tEnter the Translation Vector Magnitude: ";
                cout << "\n\t\tX Component: "; cin >> x;
                cout << "\n\t\tY Component: "; cin >> y;

                polygon.setTranslationMatrix(Point(x, y, 1));
                cout << "\n\n------[TRANSLATION NOTED]------" << endl;
                break;
            }

            case Rotation:{
                double rotationAngle, x = 0, y = 0;
                int pivot = 0;
                cout << "\n\n------[ROTATION]------" << endl;
                cout << "\n\tEnter the Rotation Angle: ";
                cin >> rotationAngle;

                cout << "\n\tEnter 1 for Rotating about Pivot, else enter 0.";
                cout << "\n\t\tYour Choice -> "; cin >> pivot;

                if(pivot){
                    cout << "\n\tEnter Pivot Point: ";
                    cout << "\n\t\tEnter X Coordinate: "; cin >> x;
                    cout << "\n\t\tEnter Y Coordinate: "; cin >> y;
                }

                polygon.setRotationMatrix(rotationAngle, Point(x, y, 1));
                cout << "\n\n------[ROTATION NOTED]------" << endl;
                break;
            }

            case Scaling:{
                double xScale, yScale, xFixed = 0, yFixed = 0;
                int uniform = 0, fixed = 0;
                cout << "\n\n------[SCALING]------" << endl;
                cout << "\n\tEnter an option:";
                cout << "\n\t\t0 for Uniform Scaling";
                cout << "\n\t\t1 for Differential Scaling";
                cout << "\n\t\tYour Choice -> "; cin >> uniform;

                if(uniform){
                    cout << "\n\tEnter the Scaling Factors: ";
                    cout << "\n\t\tX Scale Factor: "; cin >> xScale;
                    cout << "\n\t\tY Scale Factor: "; cin >> yScale;
                } else{
                    cout << "\n\tEnter the Scaling Factor: "; cin >> xScale;
                    yScale = xScale;
                }

                cout << "\n\tEnter 1 for Scaling about Fixed Point, else enter 0.";
                cout << "\n\t\tYour Choice -> "; cin >> fixed;

                if(fixed){
                    cout << "\n\tEnter Fixed Point: ";
                    cout << "\n\t\tEnter X Coordinate: "; cin >> xFixed;
                    cout << "\n\t\tEnter Y Coordinate: "; cin >> yFixed;
                }

                polygon.setScaleMatrix(xScale, yScale, Point(xFixed, yFixed, 1));
```

```
                cout << "\n\n------[SCALING NOTED]------" << endl;
                break;
            }

            case Reflection:{
                int reflectionOption = 4;
                cout << "\n\n------[REFLECTION]------" << endl;

                while(reflectionOption < 0 || reflectionOption > 3){
                    cout << "\n\tEnter an option:";
                    cout << "\n\t\t0 for Reflection About X Axis";
                    cout << "\n\t\t1 for Reflection About Y Axis.";
                    cout << "\n\t\t2 for Reflection About Origin.";
                    cout << "\n\t\t3 for Reflection About Line X = Y.";
                    cout << "\n\t\tYour Choice -> "; cin >> reflectionOption;
                }

                polygon.setReflectionMatrix(reflectionOption);

                cout << "\n\n------[REFLECTION NOTED]------" << endl;
                break;
            }

            case Shearing:{
                double shearParam, refConst = 0;
                int axis = 0, refLine = 0;
                cout << "\n\n------[SHEARING]------" << endl;

                cout << "\n\tEnter an option:";
                cout << "\n\t\t0 for Shearing About X Axis";
                cout << "\n\t\t1 for Shearing About Y Axis";
                cout << "\n\t\tYour Choice -> "; cin >> axis;

                cout << "\n\tEnter the Shearing Parameter: "; cin >> shearParam;

                cout << "\n\tEnter 1 for Shearing About Reference Line, else enter 0.";
                cout << "\n\t\tYour Choice -> "; cin >> refLine;

                if(refLine){
                    if(!axis){
                        cout << "\n\tEnter c for Ref. Line Y = c: ";
                        cin >> refConst;
                    } else{
                        cout << "\n\tEnter c for Ref. Line X = c: ";
                        cin >> refConst;
                    }
                }

```

```cpp
                polygon.setShearMatrix(shearParam, axis, refConst);

                cout << "\n\n------[SHEARING NOTED]------" << endl;
                break;
            }
        }

        if(i == 1){
            cout << "\n\n------[TRANSFORMATION 1 NOTED]------" << endl;
        } else{
            cout << "\n\n------[TRANSFORMATION 2 NOTED]------" << endl;
        }
    }
}

void drawAxes(){
    //To draw the X and Y axes

    glColor3d(0, 0, 0); //Black color
    glBegin(GL_LINES);

    //X-axis
    glVertex2f(X_MIN, 0);
    glVertex2f(X_MAX, 0);

    //Y-axis
    glVertex2f(0, Y_MIN);
    glVertex2f(0, Y_MAX);

    glEnd();
}

void drawPolygon(PolygonShape polygon, bool transformed){
    //To draw a given polygon

    if(!transformed){
        glColor3d(1, 0, 0); //Red color
    } else{
        glColor4f(0, 0, 1, 0.6); //Blue Color
    }

    glBegin(GL_POLYGON);

    for(int i = 0; i < polygon.getVertexCount(); i++){
        Point p = polygon.getPoint(i);
        glVertex2f(p.getX(), p.getY());
    }

    glEnd();
}
```

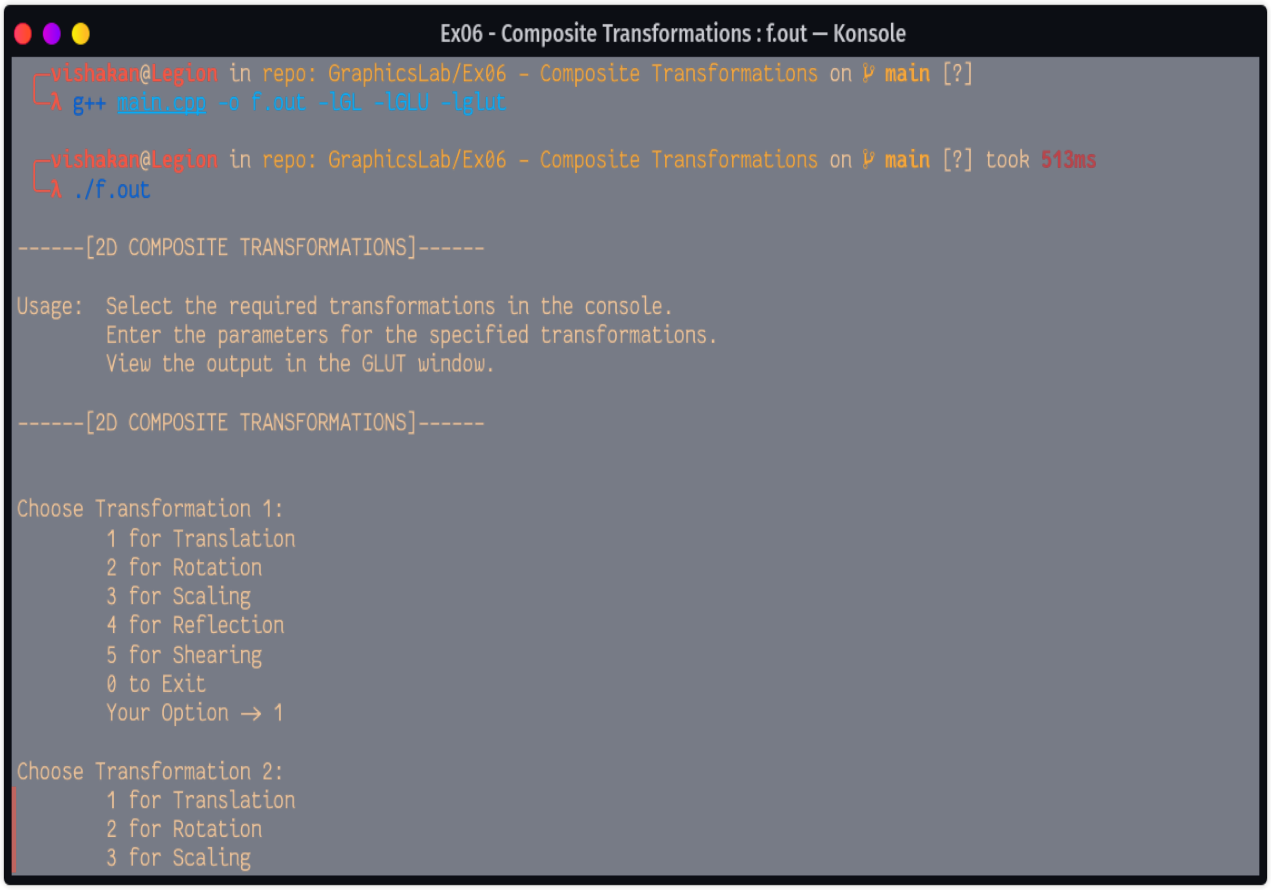
Output: Plot With Translation and Rotation

Figure 1: Plot With Translation and Rotation.



Output: Console

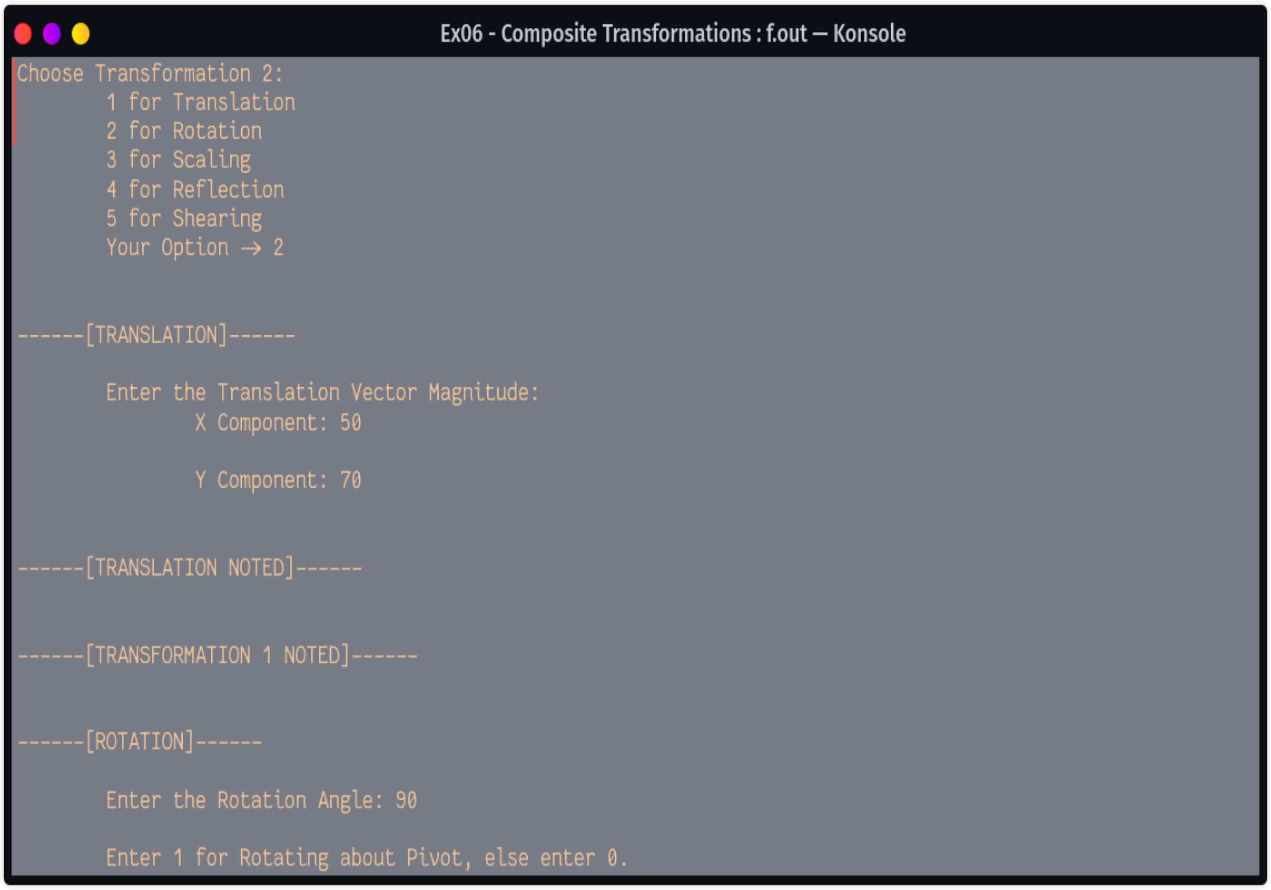
Figure 2: Output: Console.



```
┌─vishakan@Legion in repo: GraphicsLab/Ex06 - Composite Transformations on  main [?]
└─λ g++ main.cpp -o f.out -lGL -lGLU -lglut

┌─vishakan@Legion in repo: GraphicsLab/Ex06 - Composite Transformations on  main [?] took 513ms
└─λ ./f.out

------[2D COMPOSITE TRANSFORMATIONS]------

Usage:  Select the required transformations in the console.
        Enter the parameters for the specified transformations.
        View the output in the GLUT window.

------[2D COMPOSITE TRANSFORMATIONS]------


Choose Transformation 1:
        1 for Translation
        2 for Rotation
        3 for Scaling
        4 for Reflection
        5 for Shearing
        0 to Exit
        Your Option -> 1

Choose Transformation 2:
        1 for Translation
        2 for Rotation
        3 for Scaling
```

Output: Console

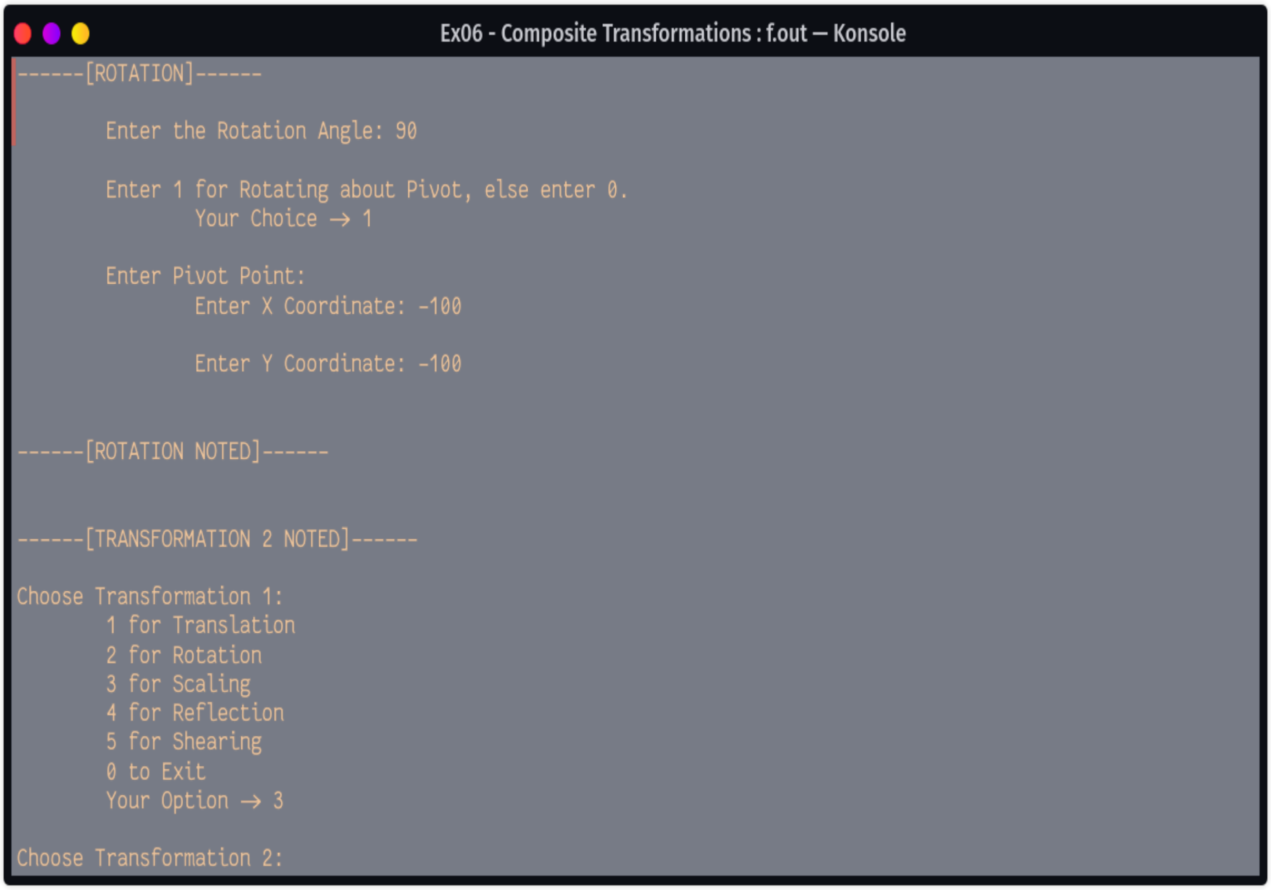
Figure 3: Output: Console.



```
Choose Transformation 2:
        1 for Translation
        2 for Rotation
        3 for Scaling
        4 for Reflection
        5 for Shearing
        Your Option → 2


------[TRANSLATION]------

        Enter the Translation Vector Magnitude:
                X Component: 50

                Y Component: 70


------[TRANSLATION NOTED]------


------[TRANSFORMATION 1 NOTED]------


------[ROTATION]------

        Enter the Rotation Angle: 90

        Enter 1 for Rotating about Pivot, else enter 0.
```

Output: Console

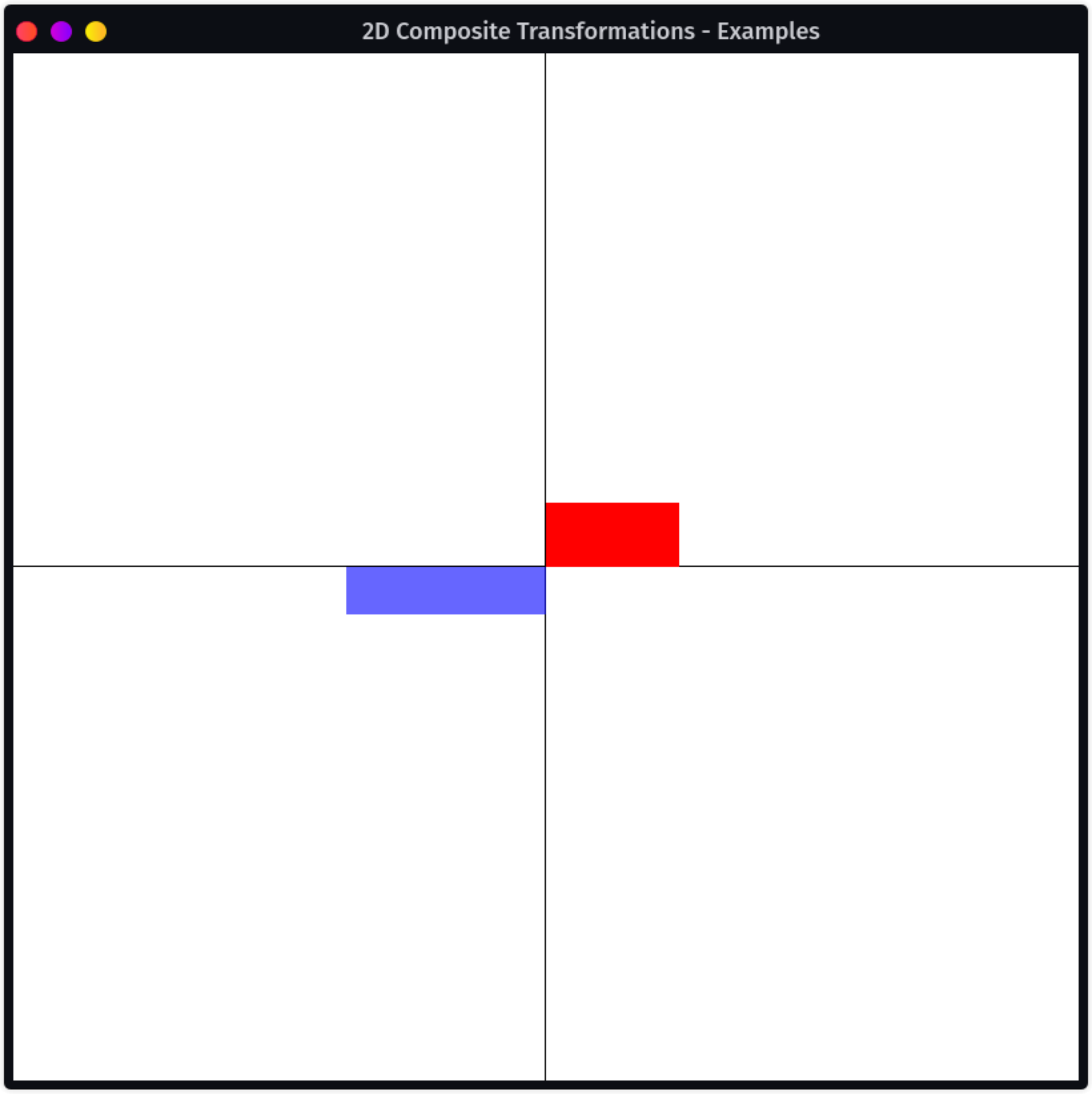
Figure 4: Output: Console.



```
------[ROTATION]------

        Enter the Rotation Angle: 90

        Enter 1 for Rotating about Pivot, else enter 0.
                Your Choice → 1

        Enter Pivot Point:
                Enter X Coordinate: -100

                Enter Y Coordinate: -100


------[ROTATION NOTED]------


------[TRANSFORMATION 2 NOTED]------

Choose Transformation 1:
        1 for Translation
        2 for Rotation
        3 for Scaling
        4 for Reflection
        5 for Shearing
        0 to Exit
        Your Option → 3

Choose Transformation 2:
```

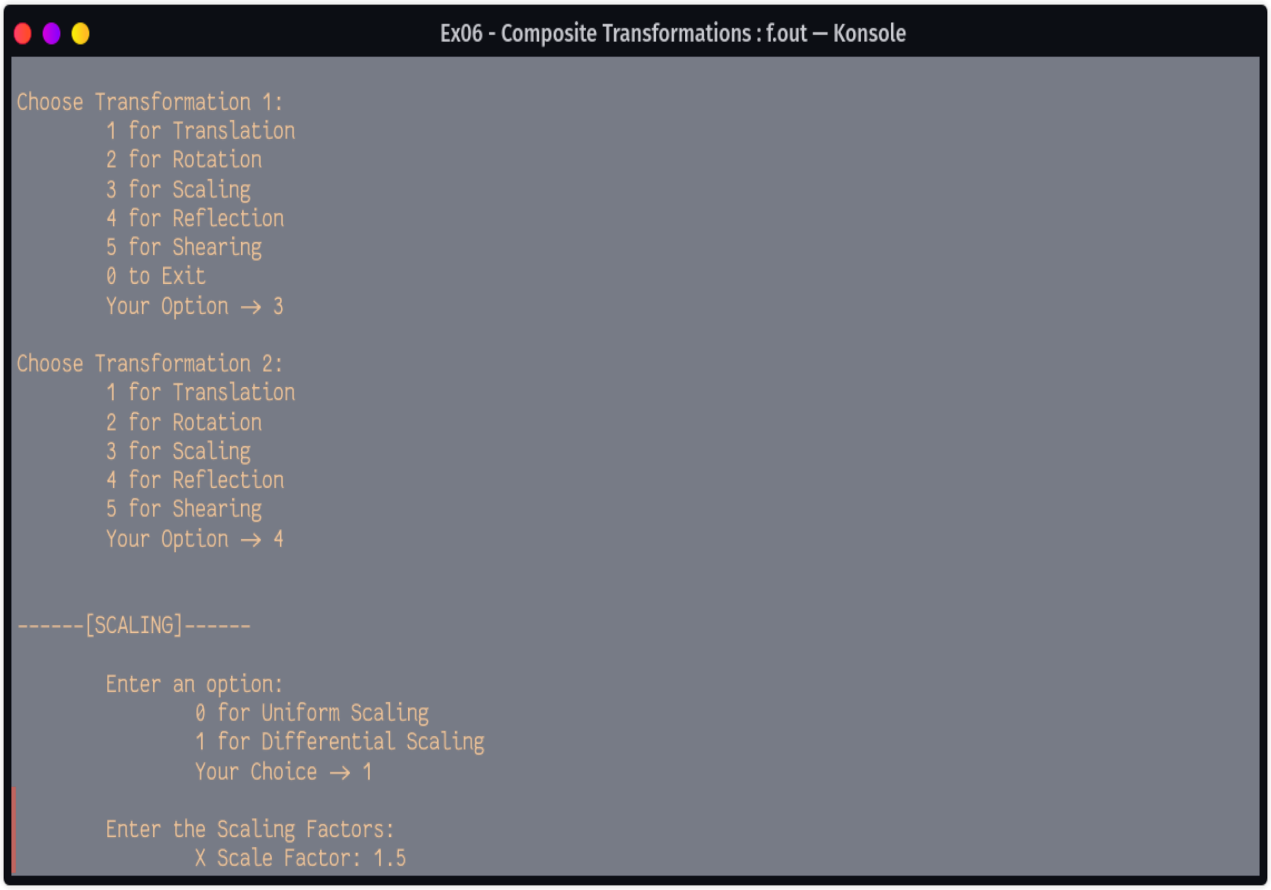
Output: Plot With Scaling and Reflection

Figure 5: Plot With Scaling and Reflection.



Output: Console

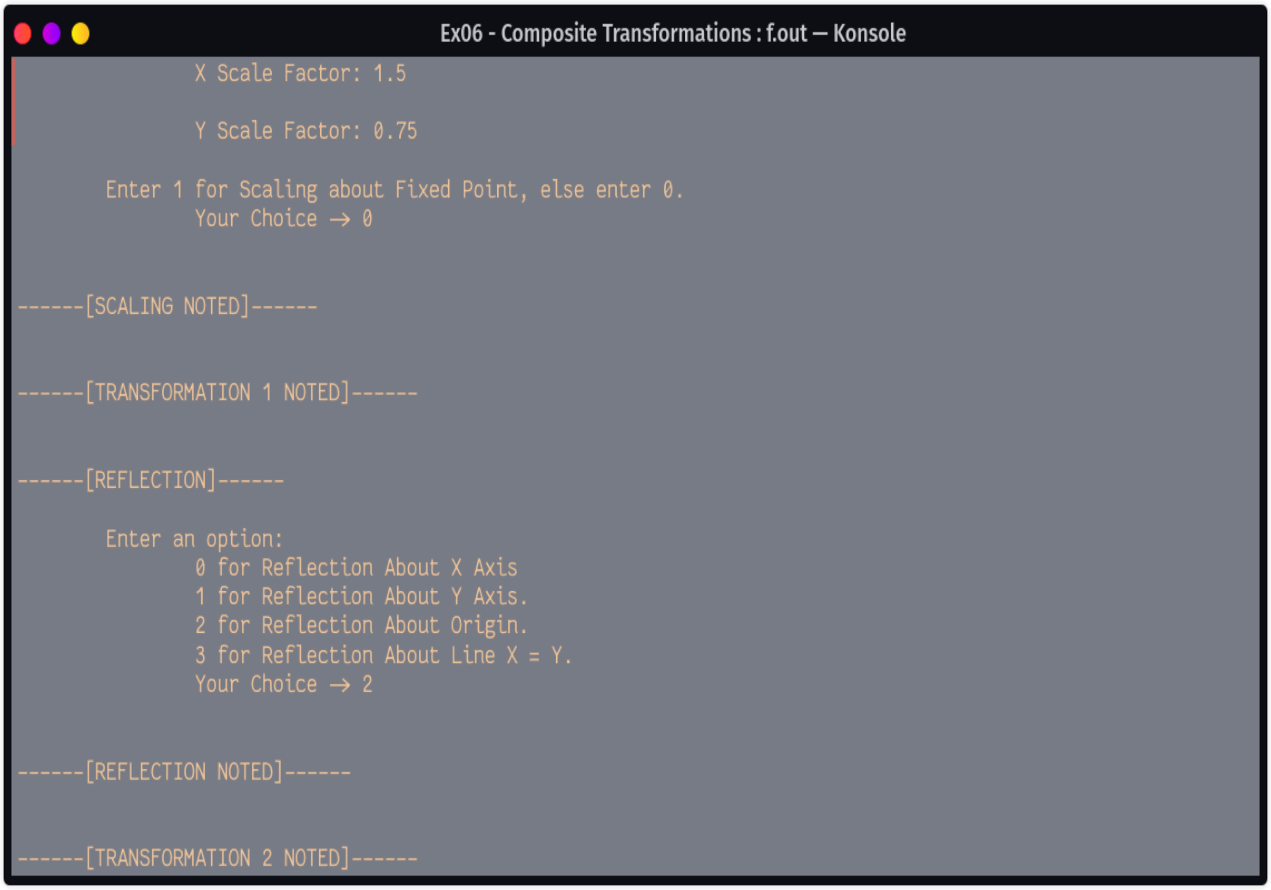
Figure 6: Console.



```

Choose Transformation 1:
        1 for Translation
        2 for Rotation
        3 for Scaling
        4 for Reflection
        5 for Shearing
        0 to Exit
        Your Option -> 3

Choose Transformation 2:
        1 for Translation
        2 for Rotation
        3 for Scaling
        4 for Reflection
        5 for Shearing
        Your Option -> 4


------[SCALING]------

        Enter an option:
                0 for Uniform Scaling
                1 for Differential Scaling
                Your Choice -> 1

        Enter the Scaling Factors:
                X Scale Factor: 1.5
```

Output: Console

Figure 7: Console.



```
                X Scale Factor: 1.5

                Y Scale Factor: 0.75

        Enter 1 for Scaling about Fixed Point, else enter 0.
                Your Choice → 0


------[SCALING NOTED]------


------[TRANSFORMATION 1 NOTED]------


------[REFLECTION]------

        Enter an option:
                0 for Reflection About X Axis
                1 for Reflection About Y Axis.
                2 for Reflection About Origin.
                3 for Reflection About Line X = Y.
                Your Choice → 2


------[REFLECTION NOTED]------


------[TRANSFORMATION 2 NOTED]------
```

Code: Window To Viewport Transformation:

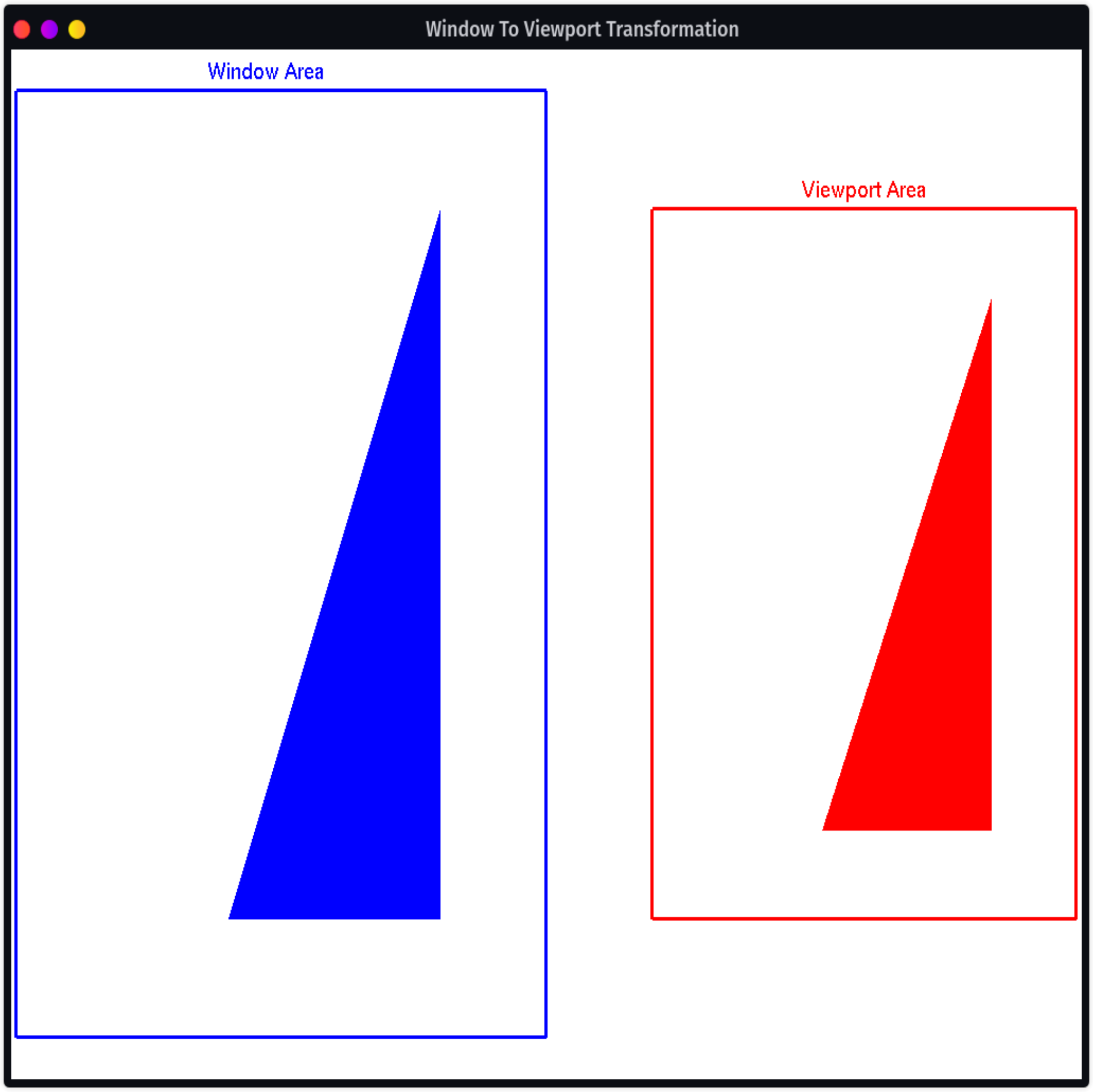
```
/*
Create a window with any 2D object and a different sized viewport. Apply window to viewport
transformation on the object. Display both window and viewport.
*/

#include <stdio.h>
#include <math.h>
#include <GL/glut.h>
#include <iostream>     //for cin, cout
#include <cstring>      //for memcpy

using namespace std;

const int HEIGHT = 870;
const int WIDTH = 1010;
const int WINDOW_XMIN = -500;
const int WINDOW_XMAX = 0;
const int WINDOW_YMIN = -400;
const int WINDOW_YMAX = 400;
const int VIEWPORT_XMIN = 100;
const int VIEWPORT_XMAX = 500;
const int VIEWPORT_YMIN = -300;
const int VIEWPORT_YMAX = 300;

class Point{
private:
    GLdouble x, y, h;

public:
    Point(){
        x = y = 0;
        h = 1;
    }

    Point(GLint xCoord, GLint yCoord){
        x = xCoord;
        y = yCoord;
        h = 1;
    }

    Point(GLint xCoord, GLint yCoord, GLint H){
        x = xCoord;
        y = yCoord;
        h = H;
    }

```

```cpp
    void setCoords(GLdouble xCoord, GLdouble yCoord){
        x = xCoord;
        y = yCoord;
    }

    void setHomogeneousCoords(GLdouble xCoord, GLdouble yCoord, GLdouble H){
        x = xCoord;
        y = yCoord;
        h = H;
    }

    GLdouble getX() const{
        return x;
    }

    GLdouble getY() const{
        return y;
    }

    GLdouble getH() const{
        return h;
    }

    GLdouble getHomogenousX() const{
        return x * h;

    }

    GLdouble getHomogenousY() const{
        return y * h;
    }
};

class PolygonShape{
private:
    int numVertices;
    Point *points;
    double transformMatrix[3][3];

public:
    PolygonShape(){
        numVertices = 0;
    }

    PolygonShape(int noVertices){
        numVertices = noVertices;
        points = new Point[numVertices];
    }

    int getVertexCount() const{
        return numVertices;
    }

    Point getPoint(int i){
        return points[i];
    }

    void setVertices(int noVertices){
        numVertices = noVertices;
        points = new Point[numVertices];
    }

    void setPoint(int i, GLdouble x, GLdouble y, GLdouble h = 1){
        points[i].setHomogeneousCoords(x, y, h);
    }

    void setTransformMatrix(){
        //Perform Window->Viewport Transformation using Translation and Scaling

        double xShift = VIEWPORT_XMIN - WINDOW_XMIN;
        double yShift = VIEWPORT_YMIN - WINDOW_YMIN;

        double translateMatrix[3][3] = {    {1, 0, xShift},
                                            {0, 1, yShift},
                                            {0, 0, 1}};

        double xScale = (double) (VIEWPORT_XMAX - VIEWPORT_XMIN) / (WINDOW_XMAX - WINDOW_XMIN);
        double yScale = (double) (VIEWPORT_YMAX - VIEWPORT_YMIN) / (WINDOW_YMAX - WINDOW_YMIN);

        Point pivot(VIEWPORT_XMIN, VIEWPORT_YMIN);

        double scaleMatrix[3][3] = {    {xScale, 0, pivot.getHomogenousX() * (1 - xScale)},
                                        {0, yScale, pivot.getHomogenousY() * (1 - yScale)},
                                        {0, 0, 1}};

        double product = 0;

        //Composite Transformation = Scaling * Translation
        for(int i = 0; i < 3; i++){
            for(int j = 0; j < 3; j++){
                product = 0;

                for(int k = 0; k < 3; k++){
                    product += scaleMatrix[i][k] * translateMatrix[k][j];
                }

```

```cpp
                transformMatrix[i][j] = product;
            }
        }
    }

    PolygonShape getViewportPolygon(){

        PolygonShape polyDash(numVertices);
        double values[3];

        for(int i = 0; i < numVertices; i++){
            Point p = getPoint(i);

            //[3 x 3] x [3 x 1] = [3 x 1] matrix
            for(int j = 0; j < 3; j++){
                values[j] = transformMatrix[j][0] * p.getHomogenousX() +
                            transformMatrix[j][1] * p.getHomogenousY() +
                            transformMatrix[j][2] * p.getH();
            }

            polyDash.setPoint(i, values[0]/p.getH(), values[1]/p.getH(), values[2]);
        }

        return polyDash;
    }
};


void initializeDisplay();
void plotComponents();
void plotBoundaries();
void plotPolygon(PolygonShape polygon, bool transformed = false);

PolygonShape polygon;

int main(int argc, char **argv){
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowPosition(0, 0);
    glutInitWindowSize(WIDTH, HEIGHT);
    glutCreateWindow("Window To Viewport Transformation");

    polygon.setVertices(3);
    polygon.setPoint(0, -300, -300);
    polygon.setPoint(1, -100, -300);
    polygon.setPoint(2, -100, 300);
    //polygon.setPoint(3, -300, 300);

    initializeDisplay();
    glutDisplayFunc(plotComponents);

    glutMainLoop();

    return 1;
}

void initializeDisplay(){
    //Initialize the display parameters

    glClearColor(1, 1, 1, 0);
    glMatrixMode(GL_PROJECTION);
    gluOrtho2D(-WIDTH/2, WIDTH/2, -HEIGHT/2, HEIGHT/2);
    glClear(GL_COLOR_BUFFER_BIT);
}

void plotComponents(){
    //Plot the window, viewport and transformations

    plotBoundaries();
    plotPolygon(polygon);
    polygon.setTransformMatrix();
    PolygonShape polygonDash = polygon.getViewportPolygon();
    plotPolygon(polygonDash, true);
}

void plotBoundaries(){
    //Plot the window and viewport boundaries

    glLineWidth(3);

    //Title of window area
    glColor3d(0, 0, 1); //Blue color
    unsigned char windowString[] = "Window Area";
    glutBitmapLength(GLUT_BITMAP_HELVETICA_18, windowString);
    glRasterPos2d(-320, 410);

    for(int i = 0; i < strlen((const char *)windowString); i++) {
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18, windowString[i]);
    }

    //Plot the window area
    glBegin(GL_LINE_LOOP);
    glVertex2f(WINDOW_XMIN, WINDOW_YMIN);
    glVertex2f(WINDOW_XMAX, WINDOW_YMIN);
    glVertex2f(WINDOW_XMAX, WINDOW_YMAX);
    glVertex2f(WINDOW_XMIN, WINDOW_YMAX);
    glEnd();

    //Title of viewport area
    glColor3d(1, 0, 0); //Red color
    unsigned char viewportString[] = "Viewport Area";
    glutBitmapLength(GLUT_BITMAP_HELVETICA_18, viewportString);
    glRasterPos2d(240, 310);

    for(int i = 0; i < strlen((const char *)viewportString); i++) {
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18, viewportString[i]);
    }

    //Plot the viewport area
    glBegin(GL_LINE_LOOP);
    glVertex2f(VIEWPORT_XMIN, VIEWPORT_YMIN);
    glVertex2f(VIEWPORT_XMAX, VIEWPORT_YMIN);
    glVertex2f(VIEWPORT_XMAX, VIEWPORT_YMAX);
    glVertex2f(VIEWPORT_XMIN, VIEWPORT_YMAX);
    glEnd();

    glFlush();
}

void plotPolygon(PolygonShape polygon, bool transformed){
    //To draw a given polygon

    if(!transformed){
        glColor3d(0, 0, 1); //Blue color
    } else{
        glColor3d(1, 0, 0); //Red Color
    }

    glBegin(GL_POLYGON);

    for(int i = 0; i < polygon.getVertexCount(); i++){
        Point p = polygon.getPoint(i);
        glVertex2f(p.getX(), p.getY());
    }

    glEnd();
}
```

Output: Window To Viewport Transformation

Figure 8: Window To Viewport Transformation.



Learning Outcome:

- I understood how to perform **composite transformations** using OpenGL and C++ programming.
- I learnt to perform proper matrix multiplication of the base transformation matrices to get the appropriate composite transformation matrix.
- I applied the composite transformation on the base polygon and displayed the transformed polygon on the graph window.
- I learnt how to overcome the screen refreshing issue/asynchronous event handling while getting user I/O with the help of **glutTimerFunc()** and setting a **60 FPS** refresh rate.
- I was able to perform composite transformations based on translation, rotation, scaling, reflection and shearing.
- I learnt about **Window to Viewport Transformation**.
- I learnt how to set up 2 windows, each to simulate a window and a viewport output window for the purpose of demonstrating the transformation.
- I learnt to perform the transformation from window to viewport using **Translation & Scaling** transformations.
- I learnt the formula for performing the specific translation and scaling required for viewport transformation.
- I understood how to display raster text using the **glutBitmapCharacter() and glRasterPos2d()** methods.
- I refreshed my C/C++ concepts regarding **cin, cout and memcpy()** methods.
- I learnt how to configure **OpenGL** in **Linux**.