Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Lab Work 3

[Course Code: COMP 342]

Submitted by:
Jatin Madhikarmi
Roll No. 32

Submitted to:
Mr. Dhiraj Shrestha
Department of Computer Science and Engineering

1 The Translation Algorithm

To move a point $P(x, y)$ to a new position $P'(x', y')$ by translation factors (t_x, t_y), we use the following linear equations:

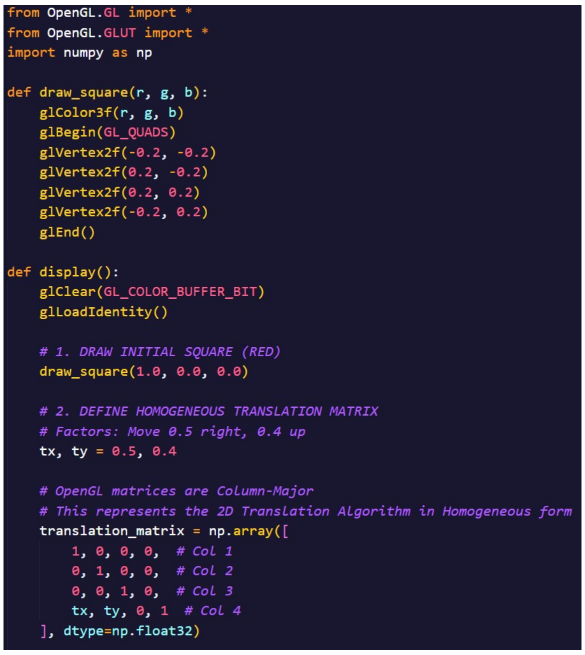
$$x' = x + t_x, \quad y' = y + t_y \tag{1}$$

In **Homogeneous Coordinates**, we represent the 2D point as a 3D vector. This allows the translation to be performed via matrix multiplication:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{2}$$

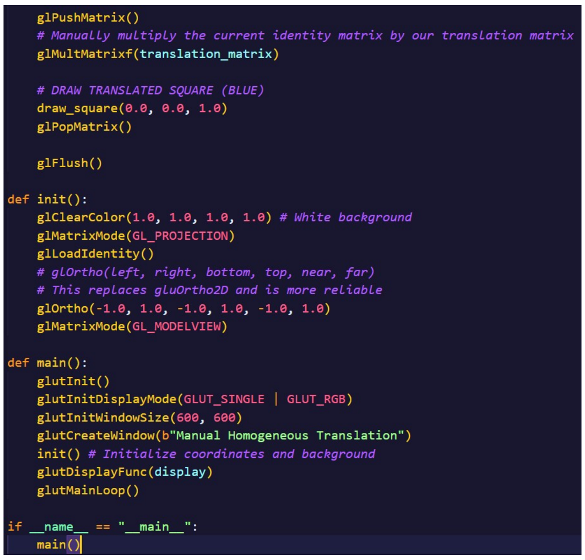
PyOpenGL Implementation

The following code renders an initial red square and its translated blue counterpart.



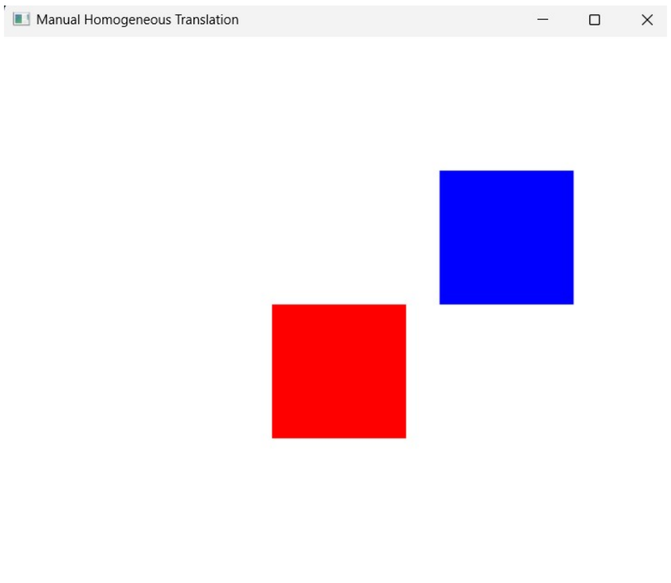
```python
from OpenGL.GL import *
from OpenGL.GLUT import *
import numpy as np

def draw_square(r, g, b):
    glColor3f(r, g, b)
    glBegin(GL_QUADS)
    glVertex2f(-0.2, -0.2)
    glVertex2f(0.2, -0.2)
    glVertex2f(0.2, 0.2)
    glVertex2f(-0.2, 0.2)
    glEnd()

def display():
    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()

    # 1. DRAW INITIAL SQUARE (RED)
    draw_square(1.0, 0.0, 0.0)

    # 2. DEFINE HOMOGENEOUS TRANSLATION MATRIX
    # Factors: Move 0.5 right, 0.4 up
    tx, ty = 0.5, 0.4

    # OpenGL matrices are Column-Major
    # This represents the 2D Translation Algorithm in Homogeneous form
    translation_matrix = np.array([
        1, 0, 0, 0,  # Col 1
        0, 1, 0, 0,  # Col 2
        0, 0, 1, 0,  # Col 3
        tx, ty, 0, 1  # Col 4
    ], dtype=np.float32)
```

(a) 2D Translation Source Code 1



```python
    glPushMatrix()
    # Manually multiply the current identity matrix by our translation matrix
    glMultMatrixf(translation_matrix)

    # DRAW TRANSLATED SQUARE (BLUE)
    draw_square(0.0, 0.0, 1.0)
    glPopMatrix()

    glFlush()

def init():
    glClearColor(1.0, 1.0, 1.0, 1.0) # White background
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    # glOrtho(left, right, bottom, top, near, far)
    # This replaces gluOrtho2D and is more reliable
    glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0)
    glMatrixMode(GL_MODELVIEW)

def main():
    glutInit()
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
    glutInitWindowSize(600, 600)
    glutCreateWindow(b"Manual Homogeneous Translation")
    init() # Initialize coordinates and background
    glutDisplayFunc(display)
    glutMainLoop()

if __name__ == "__main__":
    main()
```

(b) 2D Translation Source Code 2



(c) Program Output

Figure 1: Complete 2D Translation Implementation and Result

Conclusion

The use of homogeneous coordinates is a cornerstone of modern computer graphics. By mapping 2D points into a higher-dimensional space, we can treat translation as a matrix multiplication rather than a simple addition. This unification is vital because it enables the **concatenation of transformations**. A single complex matrix can represent a series of translations, rotations, and scales, which the GPU can process with high efficiency.

2 The Rotation Algorithm

Rotation moves a point $P(x, y)$ along a circular path centered at the origin. For a counter-clockwise rotation by an angle θ, the new coordinates $P'(x', y')$ are calculated as:

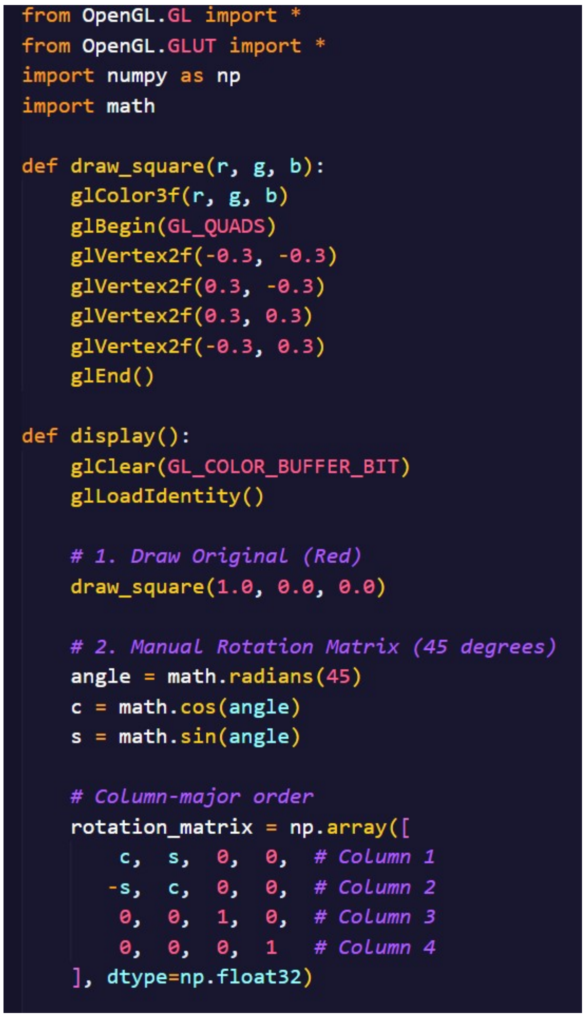
$$x' = x\cos\theta - y\sin\theta, \quad y' = x\sin\theta + y\cos\theta \tag{3}$$

In **Homogeneous Coordinates**, this is represented as the following 3×3 matrix multiplication:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{4}$$

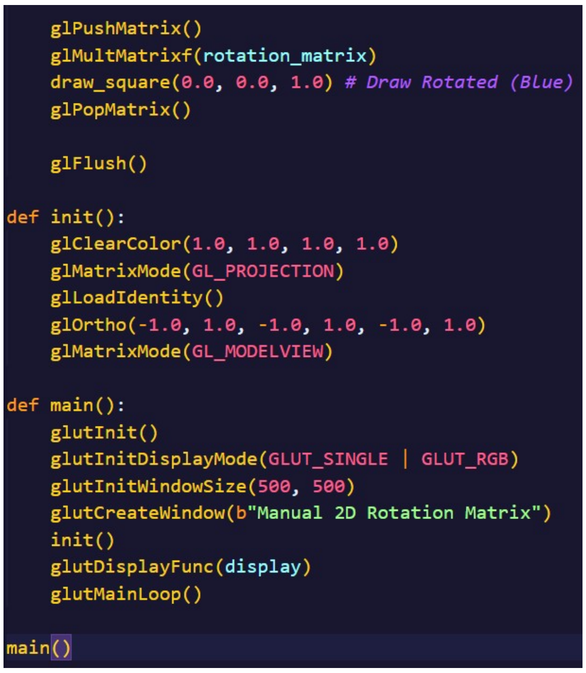
PyOpenGL Implementation

The following snippet demonstrates the rotation of a square using the `glRotatef` function.



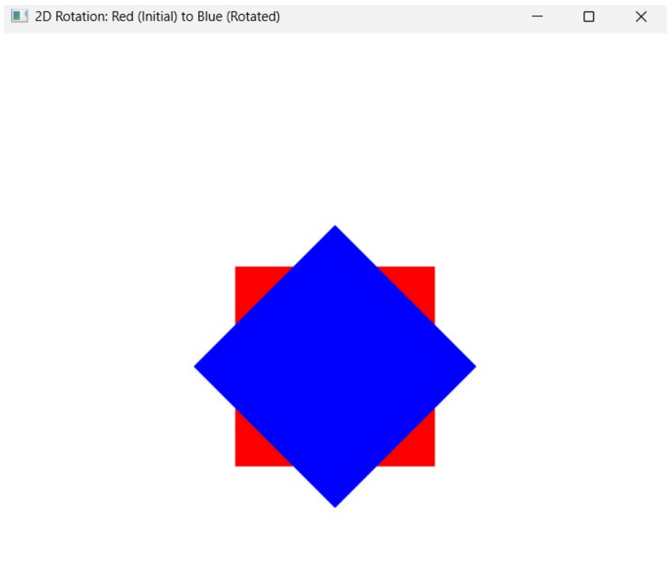
```
from OpenGL.GL import *
from OpenGL.GLUT import *
import numpy as np
import math

def draw_square(r, g, b):
    glColor3f(r, g, b)
    glBegin(GL_QUADS)
    glVertex2f(-0.3, -0.3)
    glVertex2f(0.3, -0.3)
    glVertex2f(0.3, 0.3)
    glVertex2f(-0.3, 0.3)
    glEnd()

def display():
    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()

    # 1. Draw Original (Red)
    draw_square(1.0, 0.0, 0.0)

    # 2. Manual Rotation Matrix (45 degrees)
    angle = math.radians(45)
    c = math.cos(angle)
    s = math.sin(angle)

    # Column-major order
    rotation_matrix = np.array([
        c,  s,  0,  0,  # Column 1
       -s,  c,  0,  0,  # Column 2
        0,  0,  1,  0,  # Column 3
        0,  0,  0,  1   # Column 4
    ], dtype=np.float32)
```

(a) 2D Rotation Source Code 1



```
    glPushMatrix()
    glMultMatrixf(rotation_matrix)
    draw_square(0.0, 0.0, 1.0) # Draw Rotated (Blue)
    glPopMatrix()

    glFlush()

def init():
    glClearColor(1.0, 1.0, 1.0, 1.0)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0)
    glMatrixMode(GL_MODELVIEW)

def main():
    glutInit()
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
    glutInitWindowSize(500, 500)
    glutCreateWindow(b"Manual 2D Rotation Matrix")
    init()
    glutDisplayFunc(display)
    glutMainLoop()

main()
```

(b) 2D Rotation Source Code 2



(c) Program Output

Figure 2: 2D Rotation Output

Conclusion

Rotation in 2D is a fundamental transformation that relies on trigonometric functions. By utilizing the homogeneous coordinate system, rotation becomes a linear operator. This is particularly powerful because rotation matrices are **orthogonal**, meaning their inverse is simply their transpose, which simplifies many complex geometric calculations in real-time rendering.

3 The Scaling Algorithm

Scaling is a transformation that alters the size of an object. To scale a point $P(x, y)$ by factors s_x and s_y relative to the origin, we multiply the coordinates by the respective factors:

$$x' = x \cdot s_x, \quad y' = y \cdot s_y \tag{5}$$

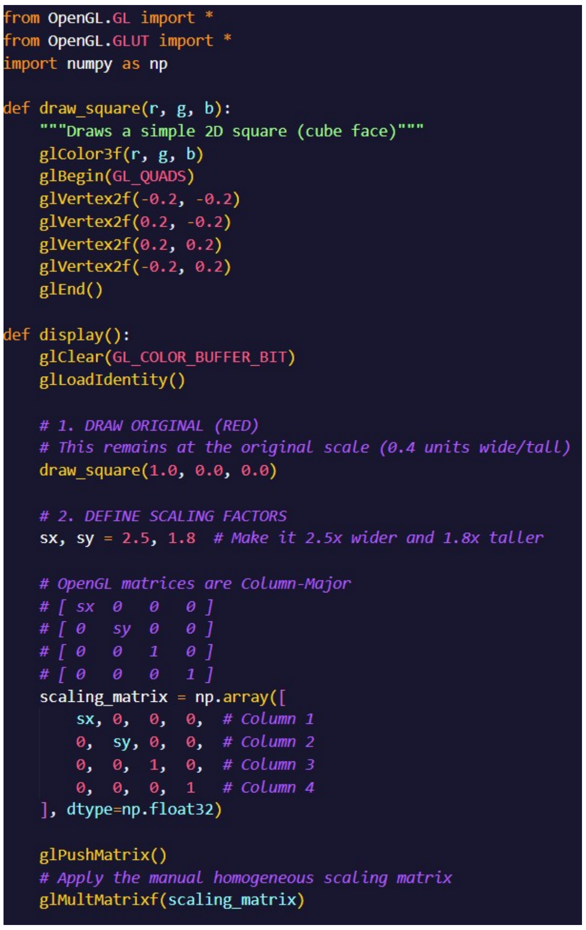
In **Homogeneous Coordinates**, we represent the 2D point as a 3D vector $[x, y, 1]^T$. This allows us to express scaling as a 3×3 matrix multiplication (or 4×4 in 3D-compatible systems like OpenGL):

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{6}$$

Where s_x and s_y are the scaling factors for the x and y axes, respectively. If $s_x = s_y$, the scaling is **uniform**; otherwise, it is **non-uniform**.

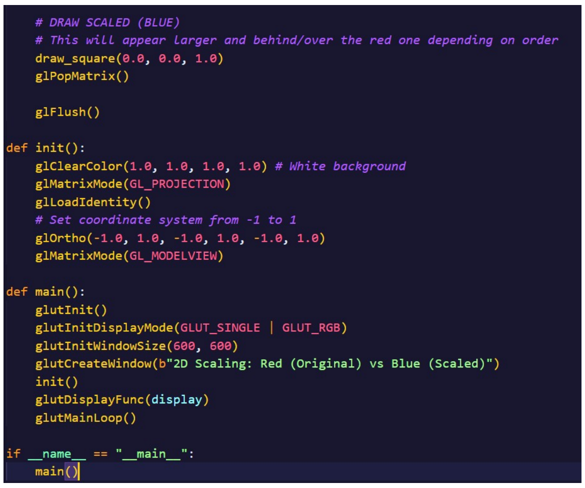
PyOpenGL Implementation

The following implementation draws an **original red square** and a **manually scaled blue square** by constructing the homogeneous matrix in column-major order.



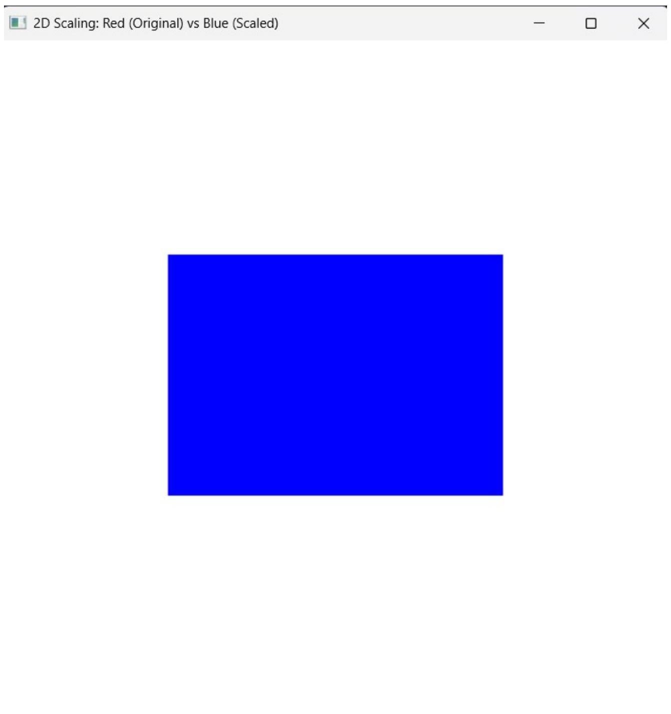
```python
from OpenGL.GL import *
from OpenGL.GLUT import *
import numpy as np

def draw_square(r, g, b):
    """Draws a simple 2D square (cube face)"""
    glColor3f(r, g, b)
    glBegin(GL_QUADS)
    glVertex2f(-0.2, -0.2)
    glVertex2f(0.2, -0.2)
    glVertex2f(0.2, 0.2)
    glVertex2f(-0.2, 0.2)
    glEnd()

def display():
    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()

    # 1. DRAW ORIGINAL (RED)
    # This remains at the original scale (0.4 units wide/tall)
    draw_square(1.0, 0.0, 0.0)

    # 2. DEFINE SCALING FACTORS
    sx, sy = 2.5, 1.8  # Make it 2.5x wider and 1.8x taller

    # OpenGL matrices are Column-Major
    # [ sx  0   0   0 ]
    # [ 0   sy  0   0 ]
    # [ 0   0   1   0 ]
    # [ 0   0   0   1 ]
    scaling_matrix = np.array([
        sx, 0,  0,  0,  # Column 1
        0,  sy, 0,  0,  # Column 2
        0,  0,  1,  0,  # Column 3
        0,  0,  0,  1   # Column 4
    ], dtype=np.float32)

    glPushMatrix()
    # Apply the manual homogeneous scaling matrix
    glMultMatrixf(scaling_matrix)
```

(a) 2D Scaling Source Code 1



```python
    # DRAW SCALED (BLUE)
    # This will appear larger and behind/over the red one depending on order
    draw_square(0.0, 0.0, 1.0)
    glPopMatrix()

    glFlush()

def init():
    glClearColor(1.0, 1.0, 1.0, 1.0) # White background
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    # Set coordinate system from -1 to 1
    glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0)
    glMatrixMode(GL_MODELVIEW)

def main():
    glutInit()
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
    glutInitWindowSize(600, 600)
    glutCreateWindow(b"2D Scaling: Red (Original) vs Blue (Scaled)")
    init()
    glutDisplayFunc(display)
    glutMainLoop()

if __name__ == "__main__":
    main()
```

(b) 2D Scaling Source Code 2



(c) Program Output

Figure 3: 2D Scaling Output

Conclusion

The implementation of 2D scaling through manual matrix construction illustrates the efficiency of the **Homogeneous Coordinate System**. By placing scaling factors on the main diagonal of the transformation matrix, we can mathematically resize objects relative to the origin. This manual approach is essential for understanding the underlying mechanics of the graphics pipeline, as it demonstrates how vertex data is transformed before rasterization. Furthermore, this unified matrix form allows for the concatenation of multiple transformations into a single calculation, which is a fundamental requirement for real-time computer graphics performance.

4 The Reflection Algorithm

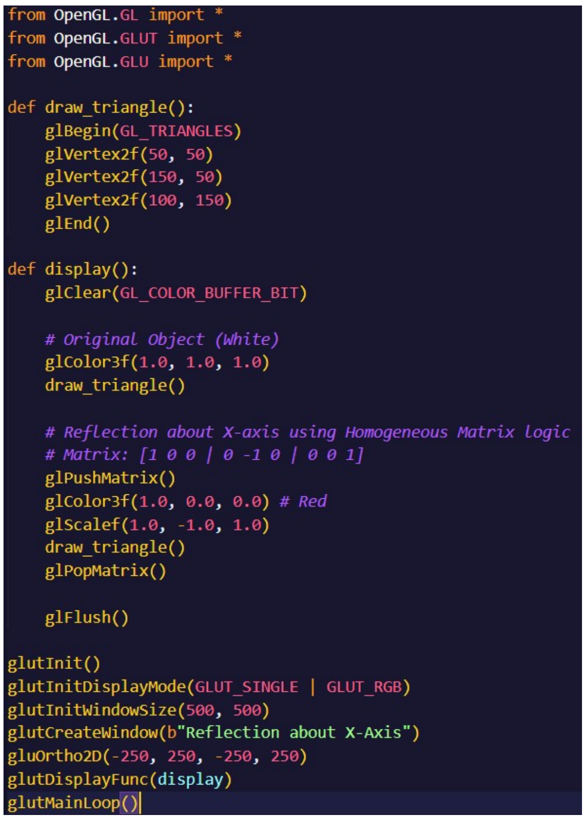
Reflection is a transformation that produces a mirror image of an object. In 2D homogeneous coordinates, a point (x, y) is represented as $(x, y, 1)$. The reflection transformation is represented by a 3×3 matrix M.

The three primary cases of reflection are defined as:

- **About X-axis:** $x' = x, y' = -y$. $M_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
- **About Y-axis:** $x' = -x, y' = y$. $M_y = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
- **About Origin:** $x' = -x, y' = -y$. $M_o = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

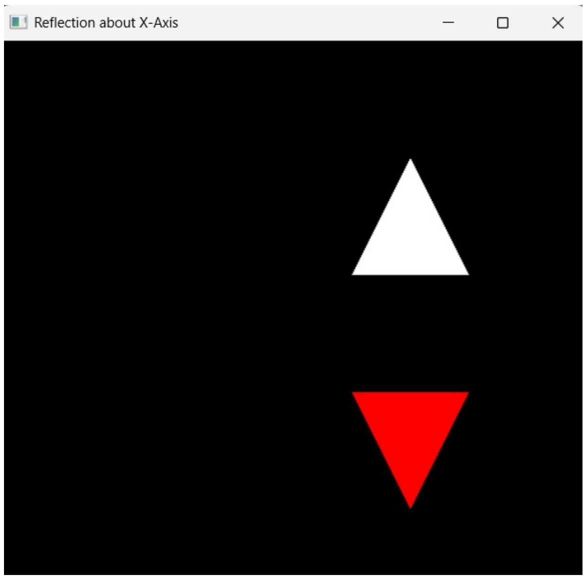
Implementations

4.1 Case A: Reflection about X-axis (ref_x.py)



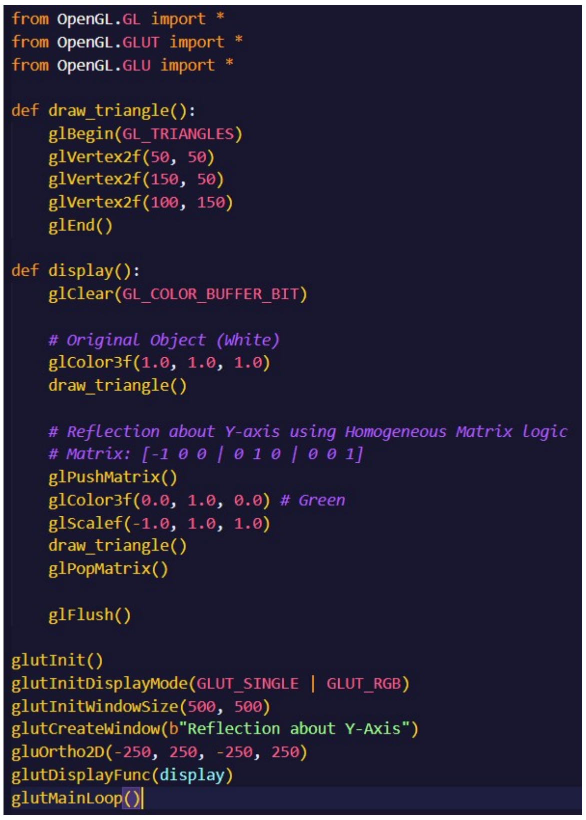
```python
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *

def draw_triangle():
    glBegin(GL_TRIANGLES)
    glVertex2f(50, 50)
    glVertex2f(150, 50)
    glVertex2f(100, 150)
    glEnd()

def display():
    glClear(GL_COLOR_BUFFER_BIT)

    # Original Object (White)
    glColor3f(1.0, 1.0, 1.0)
    draw_triangle()

    # Reflection about X-axis using Homogeneous Matrix logic
    # Matrix: [1 0 0 | 0 -1 0 | 0 0 1]
    glPushMatrix()
    glColor3f(1.0, 0.0, 0.0) # Red
    glScalef(1.0, -1.0, 1.0)
    draw_triangle()
    glPopMatrix()

    glFlush()

glutInit()
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
glutInitWindowSize(500, 500)
glutCreateWindow(b"Reflection about X-Axis")
gluOrtho2D(-250, 250, -250, 250)
glutDisplayFunc(display)
glutMainLoop()
```

(a) 2D Reflection(About X-axis) Source Code



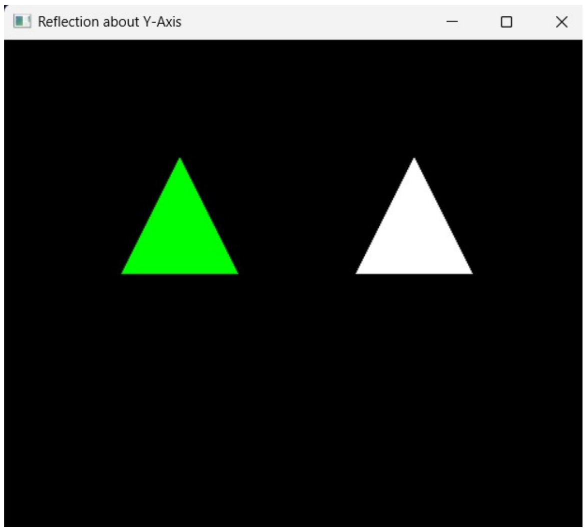
(b) 2D Reflection(About X-axis) Output

4.2 Case B: Reflection about Y-axis (ref_y.py)



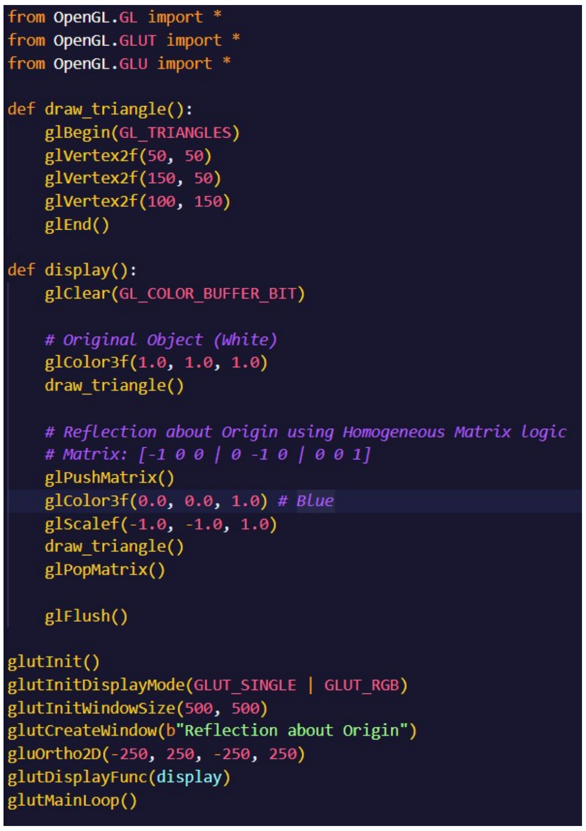
```
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *

def draw_triangle():
    glBegin(GL_TRIANGLES)
    glVertex2f(50, 50)
    glVertex2f(150, 50)
    glVertex2f(100, 150)
    glEnd()

def display():
    glClear(GL_COLOR_BUFFER_BIT)

    # Original Object (White)
    glColor3f(1.0, 1.0, 1.0)
    draw_triangle()

    # Reflection about Y-axis using Homogeneous Matrix logic
    # Matrix: [-1 0 0 | 0 1 0 | 0 0 1]
    glPushMatrix()
    glColor3f(0.0, 1.0, 0.0) # Green
    glScalef(-1.0, 1.0, 1.0)
    draw_triangle()
    glPopMatrix()

    glFlush()

glutInit()
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
glutInitWindowSize(500, 500)
glutCreateWindow(b"Reflection about Y-Axis")
gluOrtho2D(-250, 250, -250, 250)
glutDisplayFunc(display)
glutMainLoop()
```

(a) 2D Reflection(About Y-axis) Source Code



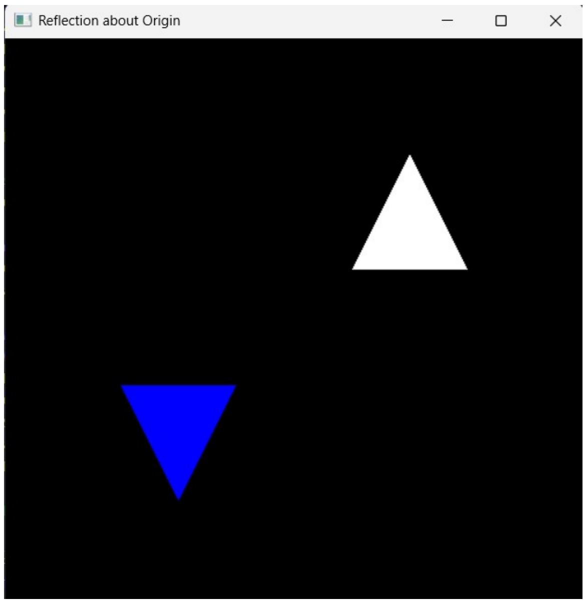
(b) 2D Reflection(About Y-axis) Output

4.3 Case C: Reflection about Origin (ref_origin.py)



```python
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *

def draw_triangle():
    glBegin(GL_TRIANGLES)
    glVertex2f(50, 50)
    glVertex2f(150, 50)
    glVertex2f(100, 150)
    glEnd()

def display():
    glClear(GL_COLOR_BUFFER_BIT)

    # Original Object (White)
    glColor3f(1.0, 1.0, 1.0)
    draw_triangle()

    # Reflection about Origin using Homogeneous Matrix Logic
    # Matrix: [-1 0 0 | 0 -1 0 | 0 0 1]
    glPushMatrix()
    glColor3f(0.0, 0.0, 1.0) # Blue
    glScalef(-1.0, -1.0, 1.0)
    draw_triangle()
    glPopMatrix()

    glFlush()

glutInit()
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
glutInitWindowSize(500, 500)
glutCreateWindow(b"Reflection about Origin")
gluOrtho2D(-250, 250, -250, 250)
glutDisplayFunc(display)
glutMainLoop()
```

(a) 2D Reflection(About Origin) Source Code



(b) 2D Reflection(About Origin) Output

Conclusion

In a homogeneous coordinate system, reflection is mathematically treated as a special case of scaling where the scaling factor is -1 along the axis of reflection. By maintaining the $w = 1$ coordinate, we ensure that these reflections can be combined with rotations and translations through simple matrix multiplication. This uniformity is what allows PyOpenGL's `glScalef()` to perform reflections seamlessly by passing negative values into the transformation pipeline.

5 The Shearing Algorithm

Shearing is a transformation that slants the shape of an object. Unlike translation or rotation, shearing changes the internal angles of the object while keeping the area constant. In a 2D plane, shearing can occur along the x-axis, the y-axis, or both. In computer graphics, 2D transformations are represented using 3x3 matrices to allow for a unified mathematical framework called **Homogeneous Coordinates**.

A point (x, y) is represented as a 3D vector $[x, y, 1]^T$. The shearing transformation is defined by the following matrix multiplication:

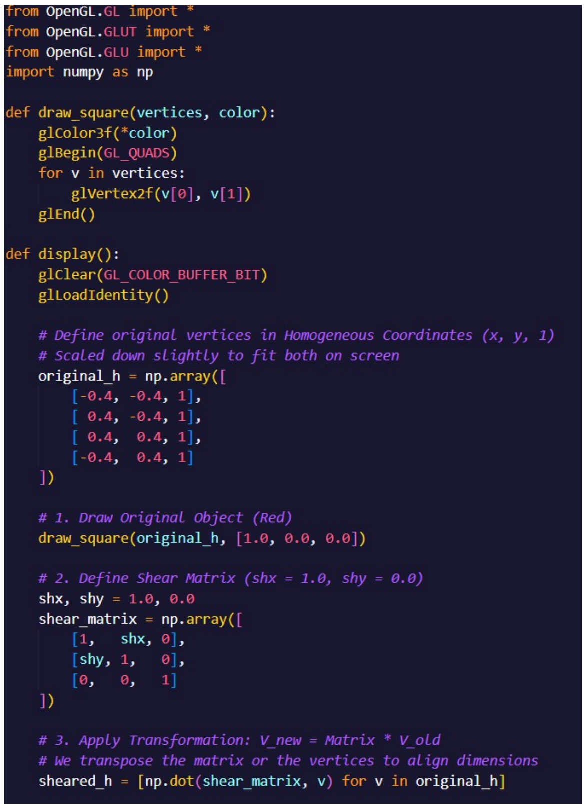
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The resulting coordinates are derived as:

- $x' = 1 \cdot x + sh_x \cdot y + 0 \cdot 1$
- $y' = sh_y \cdot x + 1 \cdot y + 0 \cdot 1$
- $w' = 0 \cdot x + 0 \cdot y + 1 \cdot 1 = 1$

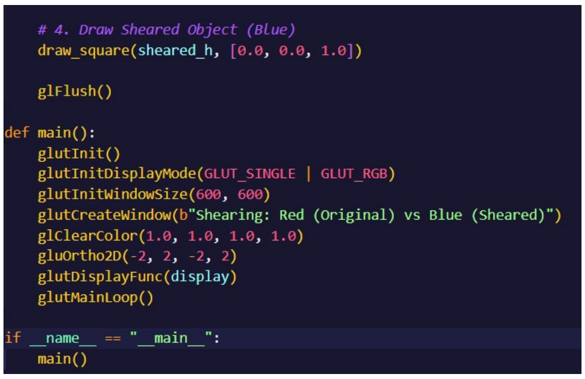
PyOpenGL Implementation

The following script renders the original object (Red) and the transformed object (Blue) using the homogeneous shearing matrix.



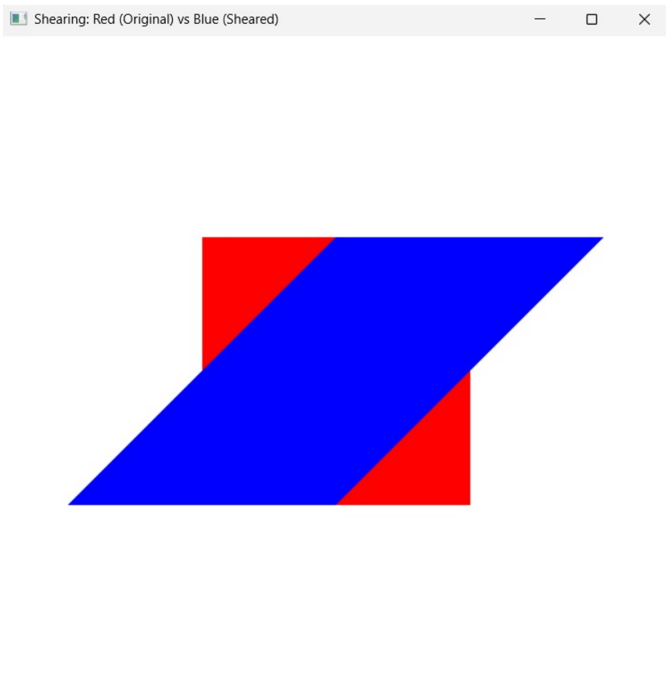
```python
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import numpy as np

def draw_square(vertices, color):
    glColor3f(*color)
    glBegin(GL_QUADS)
    for v in vertices:
        glVertex2f(v[0], v[1])
    glEnd()

def display():
    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()

    # Define original vertices in Homogeneous Coordinates (x, y, 1)
    # Scaled down slightly to fit both on screen
    original_h = np.array([
        [-0.4, -0.4, 1],
        [ 0.4, -0.4, 1],
        [ 0.4,  0.4, 1],
        [-0.4,  0.4, 1]
    ])

    # 1. Draw Original Object (Red)
    draw_square(original_h, [1.0, 0.0, 0.0])

    # 2. Define Shear Matrix (shx = 1.0, shy = 0.0)
    shx, shy = 1.0, 0.0
    shear_matrix = np.array([
        [1,   shx, 0],
        [shy, 1,   0],
        [0,   0,   1]
    ])

    # 3. Apply Transformation: V_new = Matrix * V_old
    # We transpose the matrix or the vertices to align dimensions
    sheared_h = [np.dot(shear_matrix, v) for v in original_h]
```

(a) 2D Shearing Source Code 1



```python
    # 4. Draw Sheared Object (Blue)
    draw_square(sheared_h, [0.0, 0.0, 1.0])

    glFlush()

def main():
    glutInit()
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
    glutInitWindowSize(600, 600)
    glutCreateWindow(b"Shearing: Red (Original) vs Blue (Sheared)")
    glClearColor(1.0, 1.0, 1.0, 1.0)
    gluOrtho2D(-2, 2, -2, 2)
    glutDisplayFunc(display)
    glutMainLoop()

if __name__ == "__main__":
    main()
```

(b) 2D Shearing Source Code 2



(c) Program Output

Figure 7: 2D Shearing OUtput

Conclusion

By utilizing **Homogeneous Coordinates**, shearing becomes a linear transformation within a 3-dimensional projective space. This approach is computationally efficient because it allows the shearing operation to be concatenated with other transformations like translation and rotation into a single master matrix. As demonstrated, the red original remains static while the blue object slants according to the sh_x and sh_y factors, confirming that the area remains constant while the angles are modified.

6 Composite Transformation Algorithm

A composite transformation combines multiple operations into a single 3×3 matrix. When applying transformations to a column vector, the sequence is read from right to left.

Let the sequence be: 1. Shearing (Sh), 2. Scaling (S), 3. Rotation (R), and 4. Translation (T). The composite matrix M is defined as:

$$M = T \cdot R \cdot S \cdot Sh$$

6.1 Matrix Multiplications and Resultant M

Substituting the individual matrices:

$$M = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

First, we compute the product $S \cdot Sh$:

$$M_{S,Sh} = \begin{bmatrix} s_x & s_x \cdot sh_x & 0 \\ s_y \cdot sh_y & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Next, we multiply by the rotation matrix R:

$$M_{R,S,Sh} = \begin{bmatrix} s_x \cos\theta - s_y sh_y \sin\theta & s_x sh_x \cos\theta - s_y \sin\theta & 0 \\ s_x \sin\theta + s_y sh_y \cos\theta & s_x sh_x \sin\theta + s_y \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

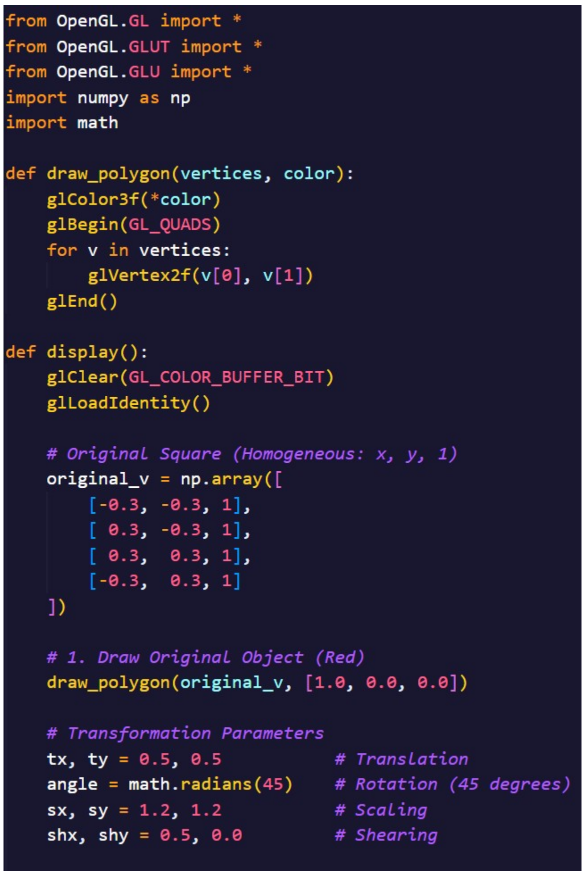
Finally, we apply the translation T. The complete composite matrix M is:

$$M = \begin{bmatrix} s_x \cos\theta - s_y sh_y \sin\theta & s_x sh_x \cos\theta - s_y \sin\theta & t_x \\ s_x \sin\theta + s_y sh_y \cos\theta & s_x sh_x \sin\theta + s_y \cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

This single matrix M now contains all transformation data. Any vertex $V = [x, y, 1]^T$ can be transformed to its final position V' via a single operation: $V' = M \cdot V$.

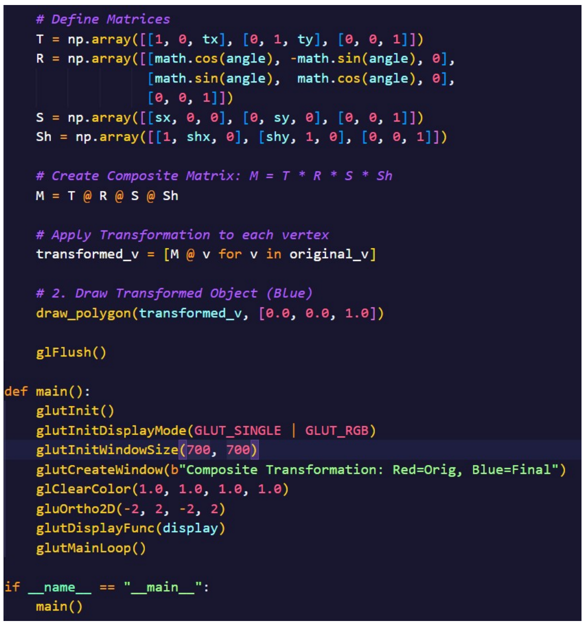
PyOpenGL Implementation

The script below demonstrates the comparison between a base square (Red) and the square after undergoing translation, rotation, scaling, and shearing (Blue).



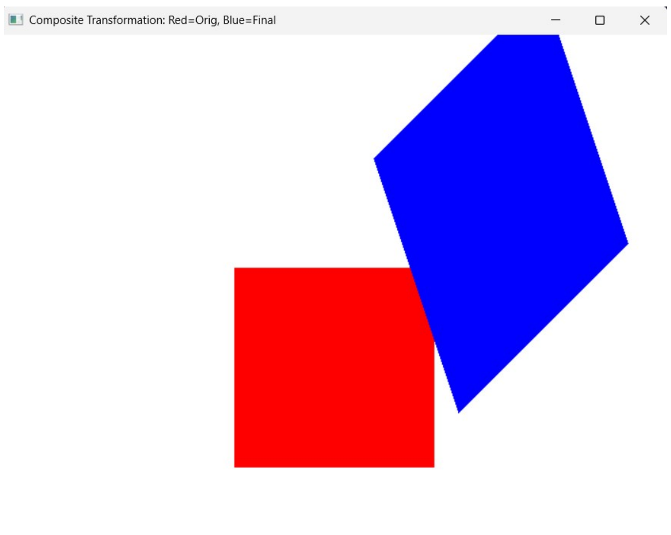
```python
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import numpy as np
import math

def draw_polygon(vertices, color):
    glColor3f(*color)
    glBegin(GL_QUADS)
    for v in vertices:
        glVertex2f(v[0], v[1])
    glEnd()

def display():
    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()

    # Original Square (Homogeneous: x, y, 1)
    original_v = np.array([
        [-0.3, -0.3, 1],
        [ 0.3, -0.3, 1],
        [ 0.3,  0.3, 1],
        [-0.3,  0.3, 1]
    ])

    # 1. Draw Original Object (Red)
    draw_polygon(original_v, [1.0, 0.0, 0.0])

    # Transformation Parameters
    tx, ty = 0.5, 0.5              # Translation
    angle = math.radians(45)       # Rotation (45 degrees)
    sx, sy = 1.2, 1.2              # Scaling
    shx, shy = 0.5, 0.0            # Shearing
```

(a) Composite Transformation Source Code 1



```python
    # Define Matrices
    T = np.array([[1, 0, tx], [0, 1, ty], [0, 0, 1]])
    R = np.array([[math.cos(angle), -math.sin(angle), 0],
                  [math.sin(angle),  math.cos(angle), 0],
                  [0, 0, 1]])
    S = np.array([[sx, 0, 0], [0, sy, 0], [0, 0, 1]])
    Sh = np.array([[1, shx, 0], [shy, 1, 0], [0, 0, 1]])

    # Create Composite Matrix: M = T * R * S * Sh
    M = T @ R @ S @ Sh

    # Apply Transformation to each vertex
    transformed_v = [M @ v for v in original_v]

    # 2. Draw Transformed Object (Blue)
    draw_polygon(transformed_v, [0.0, 0.0, 1.0])

    glFlush()

def main():
    glutInit()
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
    glutInitWindowSize(700, 700)
    glutCreateWindow(b"Composite Transformation: Red=Orig, Blue=Final")
    glClearColor(1.0, 1.0, 1.0, 1.0)
    gluOrtho2D(-2, 2, -2, 2)
    glutDisplayFunc(display)
    glutMainLoop()

if __name__ == "__main__":
    main()
```

(b) Composite Transformation Source Code 2



(c) Program Output

Figure 8: Composite Transformation Output

Conclusion

Composite transformations demonstrate the power of matrix algebra in computer graphics. By representing translation, rotation, scaling, and shearing as 3×3 homogeneous matrices, we can collapse a complex sequence of operations into a single matrix multiplication. This not only simplifies the vertex shader logic but also significantly optimizes performance by reducing the number of calculations required per vertex during rendering.