Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Lab Report 5

[Course Code: COMP 342]

Submitted by:
Jatin Madhikarmi
Roll No. 32

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

1 Introduction to 3D Translation

3D Translation is a geometric transformation that moves every point of a 3D object by the same distance in a specified direction. In a Cartesian coordinate system, this involves shifting the object along the X, Y, and Z axes.

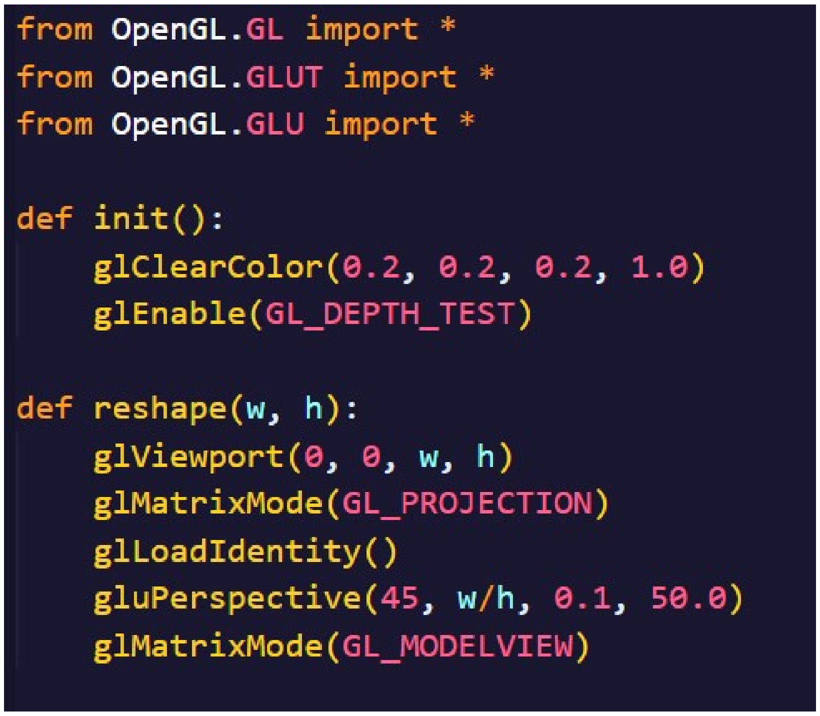
Mathematically, translation is performed by adding offsets (t_x, t_y, t_z) to the coordinates (x, y, z). In homogeneous coordinates, this is represented by the following transformation matrix T:

$$T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

When a vertex vector v is multiplied by this matrix, the new position v' is calculated. In PyOpenGL, this operation is simplified using the `glTranslatef(x, y, z)` function, which modifies the current Modelview matrix.

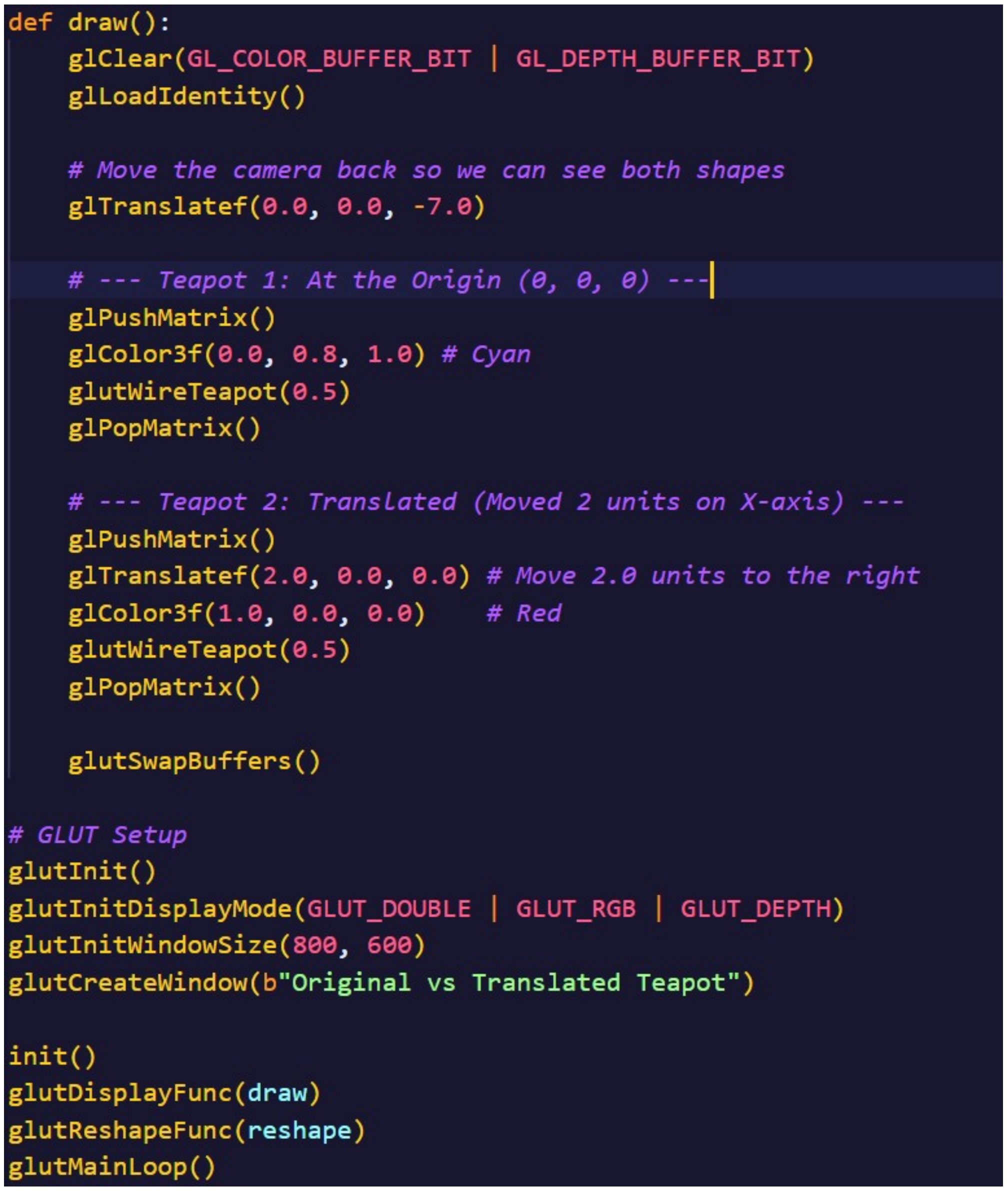
Code Implementation

The following implementation demonstrates the use of `PyOpenGL` to render two 3D teapots. The first teapot is rendered at the origin, while the second is subject to a translation transformation to illustrate the spatial shift.



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

def init():
    glClearColor(0.2, 0.2, 0.2, 1.0)
    glEnable(GL_DEPTH_TEST)

def reshape(w, h):
    glViewport(0, 0, w, h)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    gluPerspective(45, w/h, 0.1, 50.0)
    glMatrixMode(GL_MODELVIEW)
```

Figure 1: 3D Translation Source Code 1



```python
def draw():
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glLoadIdentity()

    # Move the camera back so we can see both shapes
    glTranslatef(0.0, 0.0, -7.0)

    # --- Teapot 1: At the Origin (0, 0, 0) ---
    glPushMatrix()
    glColor3f(0.0, 0.8, 1.0) # Cyan
    glutWireTeapot(0.5)
    glPopMatrix()

    # --- Teapot 2: Translated (Moved 2 units on X-axis) ---
    glPushMatrix()
    glTranslatef(2.0, 0.0, 0.0) # Move 2.0 units to the right
    glColor3f(1.0, 0.0, 0.0)    # Red
    glutWireTeapot(0.5)
    glPopMatrix()

    glutSwapBuffers()

# GLUT Setup
glutInit()
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH)
glutInitWindowSize(800, 600)
glutCreateWindow(b"Original vs Translated Teapot")

init()
glutDisplayFunc(draw)
glutReshapeFunc(reshape)
glutMainLoop()
```

Figure 2: 3D Translation Source Code 2

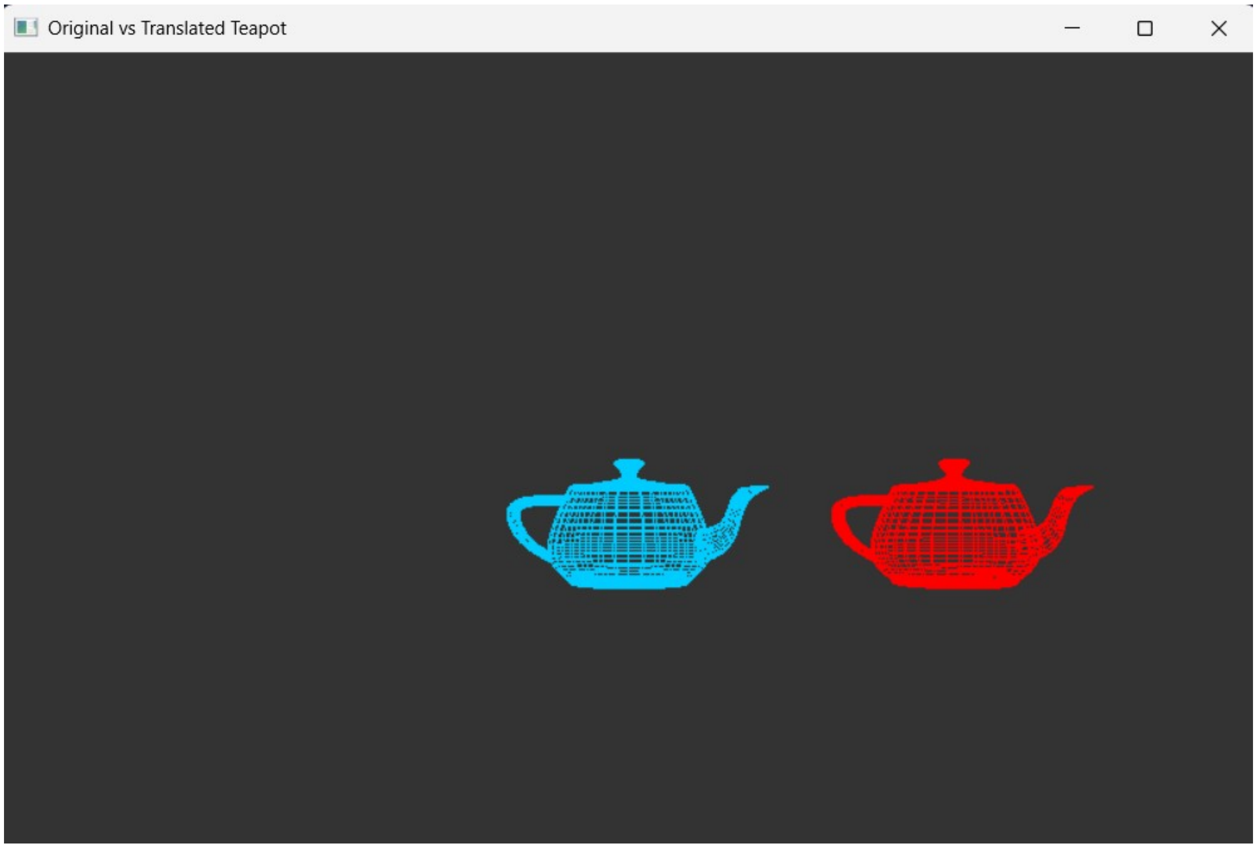


Figure 3: 3D Translation Output

Conclusion

Translation is a foundational transformation in computer graphics, essential for positioning objects within a virtual scene. By implementing translation in PyOpenGL, we observe how the coordinate system can be manipulated to create relative distances between objects. The use of the matrix stack is crucial in this process, as it allows for independent transformations of multiple objects. This report confirms that by applying the `glTranslatef` function, 3D primitives can be accurately positioned in a three-dimensional environment, forming the basis for more complex scene compositions.

2 Introduction to 3D Rotation

3D rotation is the process of changing the orientation of an object around a specified axis in a three-dimensional Cartesian coordinate system. Unlike 2D rotation, which occurs around a point, 3D rotation occurs around a line called the **axis of rotation**.

The fundamental rotations are performed around the three principal axes: X, Y, and Z. These are represented mathematically using rotation matrices R. When a point P is multiplied by these matrices, its new coordinates P' are calculated as follows:

2.1 Rotation about the X-axis

The x-coordinates remain unchanged while the y and z coordinates are transformed:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

2.2 Rotation about the Y-axis

The y-coordinates remain unchanged:

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

2.3 Rotation about the Z-axis

The z-coordinates remain unchanged (similar to 2D rotation):

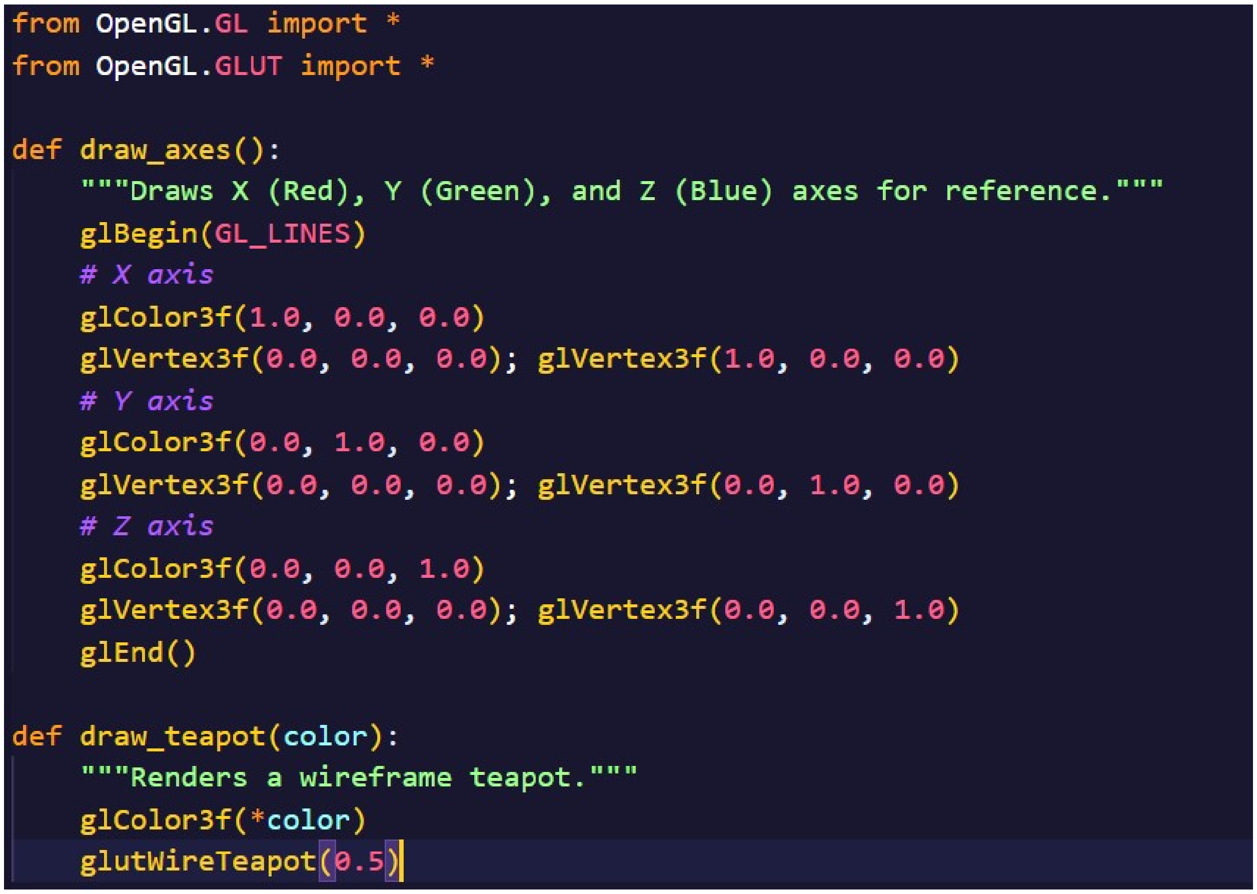
$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Implementation Strategy

The implementation is divided into two distinct modules to ensure separation of concerns and code reusability.

Geometry Module (`geometry.py`)

This file contains the definitions for the geometric primitives. It utilizes the `glutWireTeapot` function to generate the 3D mesh and includes a helper function to draw the coordinate axes for visual reference.



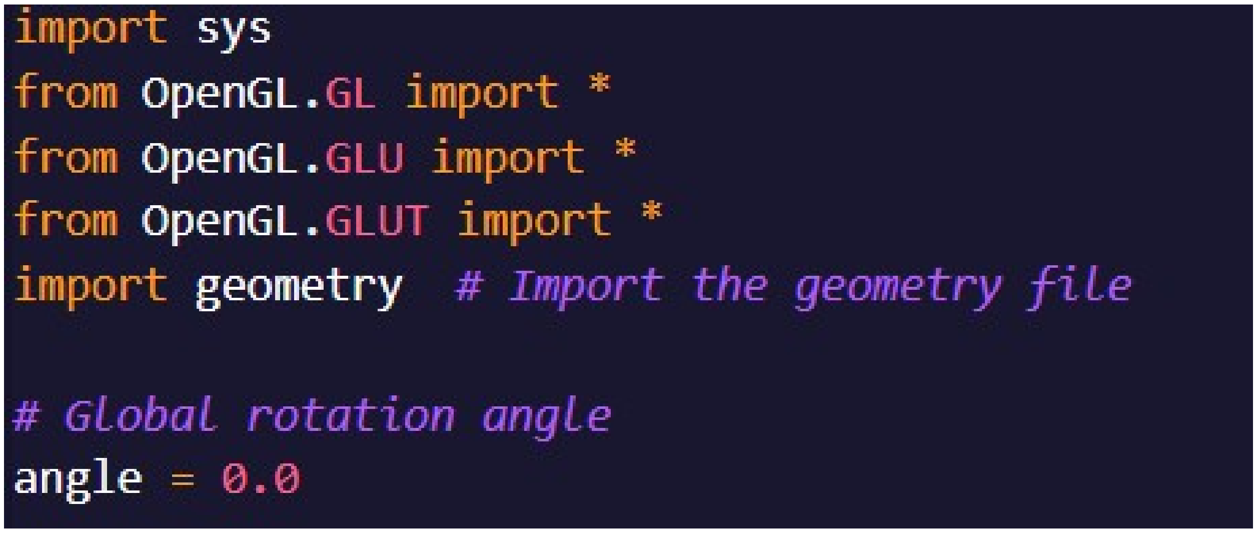
```python
from OpenGL.GL import *
from OpenGL.GLUT import *

def draw_axes():
    """Draws X (Red), Y (Green), and Z (Blue) axes for reference."""
    glBegin(GL_LINES)
    # X axis
    glColor3f(1.0, 0.0, 0.0)
    glVertex3f(0.0, 0.0, 0.0); glVertex3f(1.0, 0.0, 0.0)
    # Y axis
    glColor3f(0.0, 1.0, 0.0)
    glVertex3f(0.0, 0.0, 0.0); glVertex3f(0.0, 1.0, 0.0)
    # Z axis
    glColor3f(0.0, 0.0, 1.0)
    glVertex3f(0.0, 0.0, 0.0); glVertex3f(0.0, 0.0, 1.0)
    glEnd()

def draw_teapot(color):
    """Renders a wireframe teapot."""
    glColor3f(*color)
    glutWireTeapot(0.5)
```

Figure 4: 3D Rotation geometry.py Source Code 1

Main Application (`main.py`)

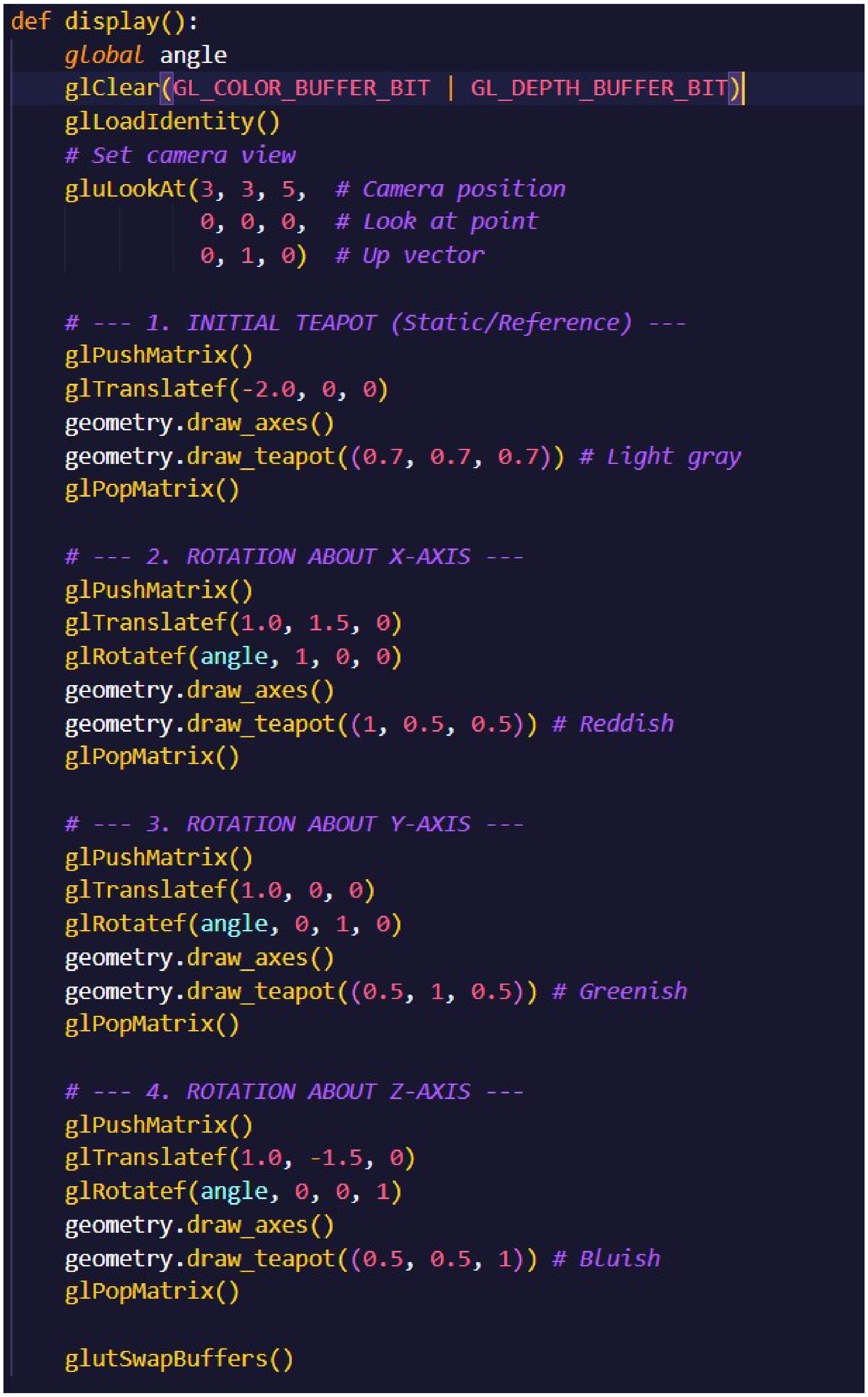
The main execution file handles:

- **Window Management:** Initializing the GLUT display mode and window.
- **Transformation Pipeline:** Using `glPushMatrix()` and `glPopMatrix()` to isolate the rotation of each teapot.
- **Animation Loop:** Utilizing the `glutIdleFunc` to increment the rotation angle over time.



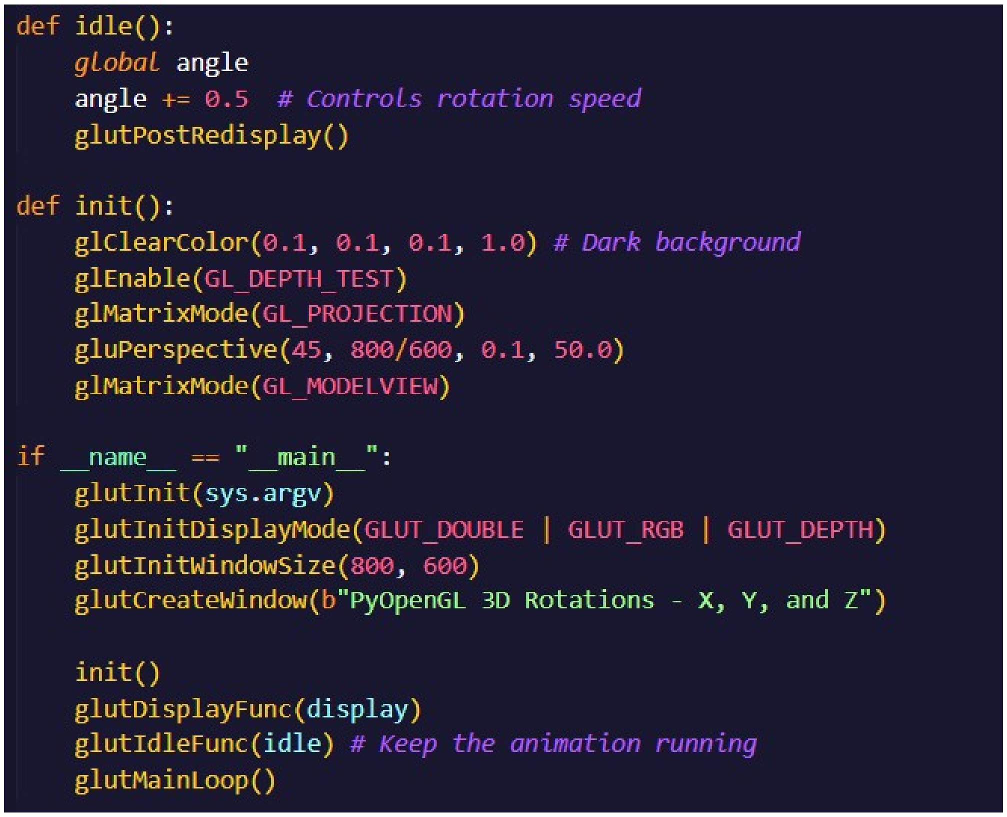
```python
import sys
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
import geometry  # Import the geometry file

# Global rotation angle
angle = 0.0
```

Figure 5: 3D Rotation Source Code 1



```python
def display():
    global angle
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glLoadIdentity()
    # Set camera view
    gluLookAt(3, 3, 5,  # Camera position
              0, 0, 0,  # Look at point
              0, 1, 0)  # Up vector

    # --- 1. INITIAL TEAPOT (Static/Reference) ---
    glPushMatrix()
    glTranslatef(-2.0, 0, 0)
    geometry.draw_axes()
    geometry.draw_teapot((0.7, 0.7, 0.7)) # Light gray
    glPopMatrix()

    # --- 2. ROTATION ABOUT X-AXIS ---
    glPushMatrix()
    glTranslatef(1.0, 1.5, 0)
    glRotatef(angle, 1, 0, 0)
    geometry.draw_axes()
    geometry.draw_teapot((1, 0.5, 0.5)) # Reddish
    glPopMatrix()

    # --- 3. ROTATION ABOUT Y-AXIS ---
    glPushMatrix()
    glTranslatef(1.0, 0, 0)
    glRotatef(angle, 0, 1, 0)
    geometry.draw_axes()
    geometry.draw_teapot((0.5, 1, 0.5)) # Greenish
    glPopMatrix()

    # --- 4. ROTATION ABOUT Z-AXIS ---
    glPushMatrix()
    glTranslatef(1.0, -1.5, 0)
    glRotatef(angle, 0, 0, 1)
    geometry.draw_axes()
    geometry.draw_teapot((0.5, 0.5, 1)) # Bluish
    glPopMatrix()

    glutSwapBuffers()
```

Figure 6: 3D Rotation Source Code 2



```python
def idle():
    global angle
    angle += 0.5  # Controls rotation speed
    glutPostRedisplay()

def init():
    glClearColor(0.1, 0.1, 0.1, 1.0) # Dark background
    glEnable(GL_DEPTH_TEST)
    glMatrixMode(GL_PROJECTION)
    gluPerspective(45, 800/600, 0.1, 50.0)
    glMatrixMode(GL_MODELVIEW)

if __name__ == "__main__":
    glutInit(sys.argv)
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH)
    glutInitWindowSize(800, 600)
    glutCreateWindow(b"PyOpenGL 3D Rotations - X, Y, and Z")

    init()
    glutDisplayFunc(display)
    glutIdleFunc(idle) # Keep the animation running
    glutMainLoop()
```

Figure 7: 3D Rotation Source Code 3

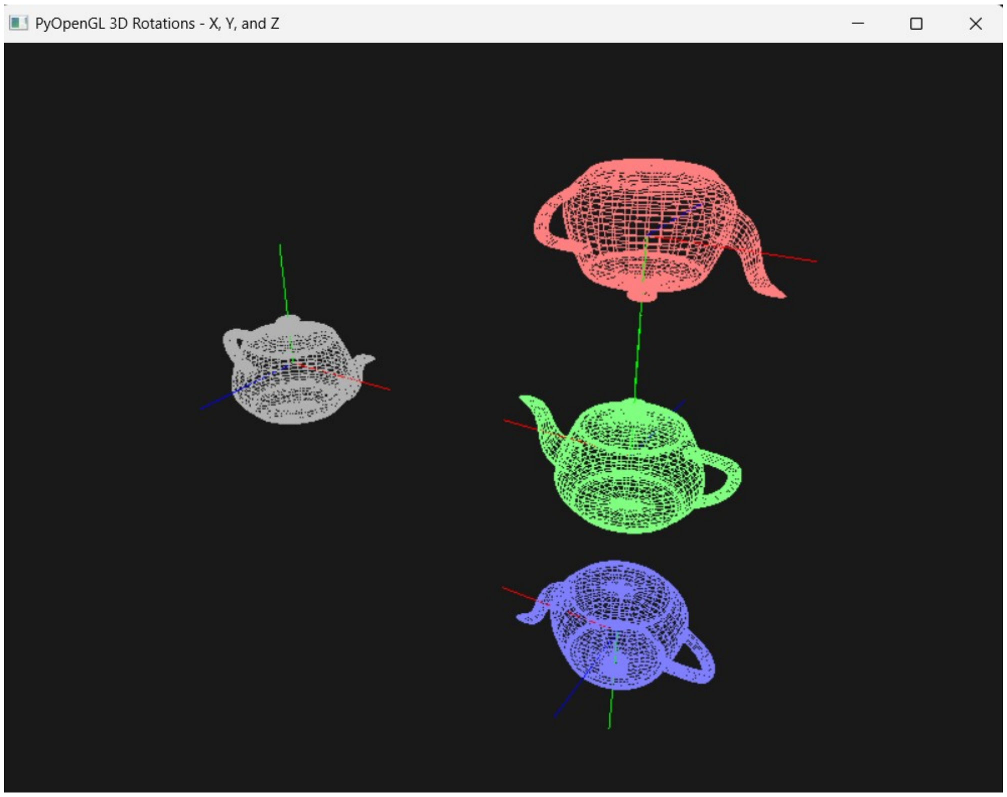


Figure 8: 3D Rotation Output

Conclusion

By implementing 3D rotations in PyOpenGL, we demonstrate how matrix transformations translate into visual orientation changes. Using the GLUT library allows for efficient rendering of complex shapes like the teapot, while the matrix stack (`glPushMatrix`) ensures that rotations around the X, Y, and Z axes can be visualized independently and simultaneously on a single canvas. This modular approach provides a foundation for more complex skeletal animations and 3D modeling applications.

3 Introduction to 3D Shearing

Shearing is a non-rigid transformation that displaces points in a given direction by an amount proportional to their perpendicular distance from a given plane or axis. While rotation and translation preserve the shape of an object, shearing ”skews” the object.

In 3D space, a shear can be applied to any pair of axes. For example, shearing along the X-axis relative to the Y-coordinate is defined by the transformation:

$$x' = x + s \cdot y, \quad y' = y, \quad z' = z$$

The general shear matrix in 3D (where s_{ab} is the shear of a relative to b) is:

$$H = \begin{bmatrix} 1 & s_{xy} & s_{xz} & 0 \\ s_{yx} & 1 & s_{yz} & 0 \\ s_{zx} & s_{zy} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Implementation Strategy

Since standard OpenGL (Fixed Function Pipeline) does not provide a specific `glShear` command, the implementation relies on manual matrix definition.

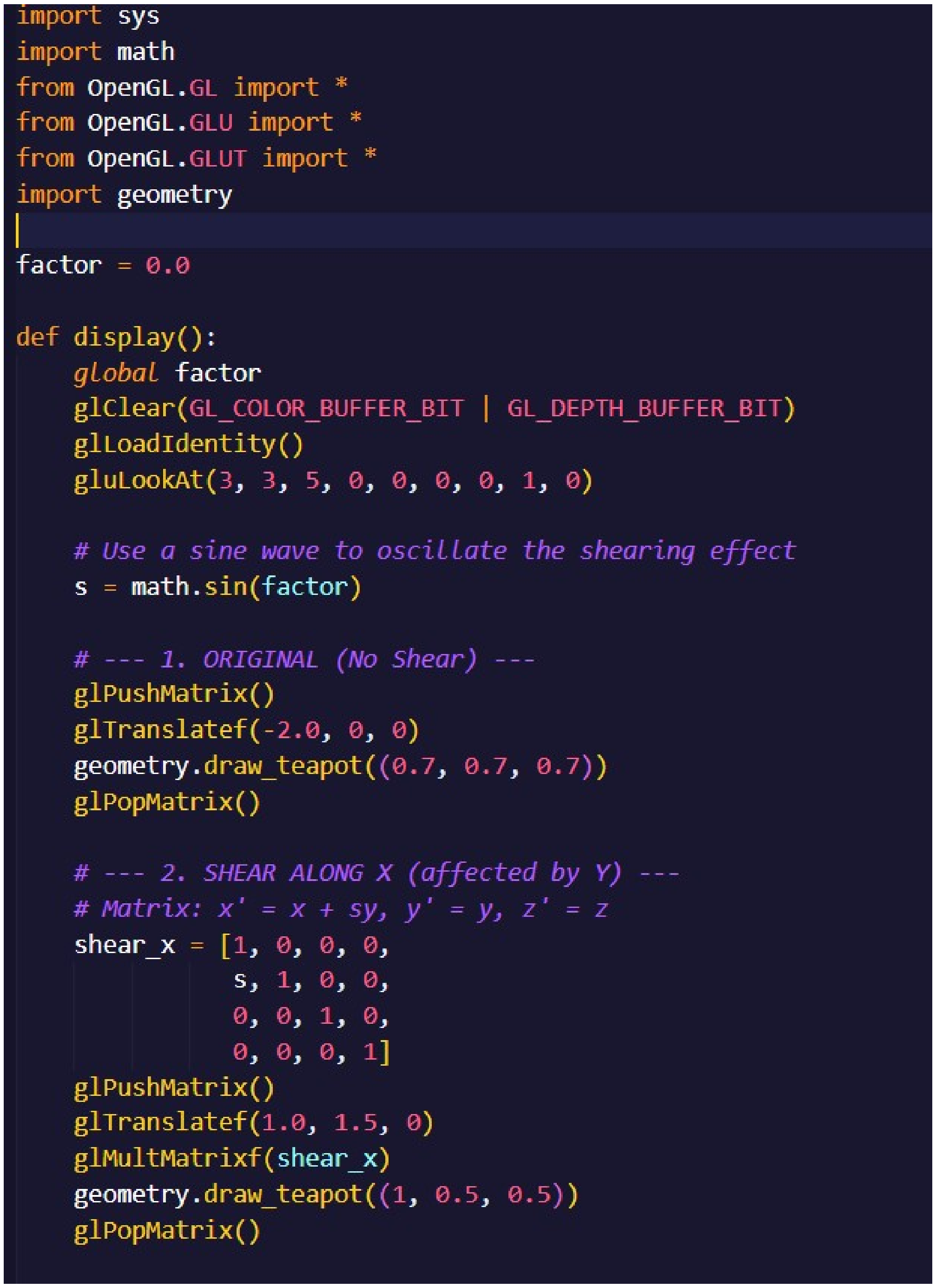
Matrix Construction

The application defines a 16-element list representing a 4x4 matrix in **column-major order**. This matrix is then passed to `glMultMatrixf()`, which multiplies the current transformation matrix by our custom shear matrix.

Modular Definition

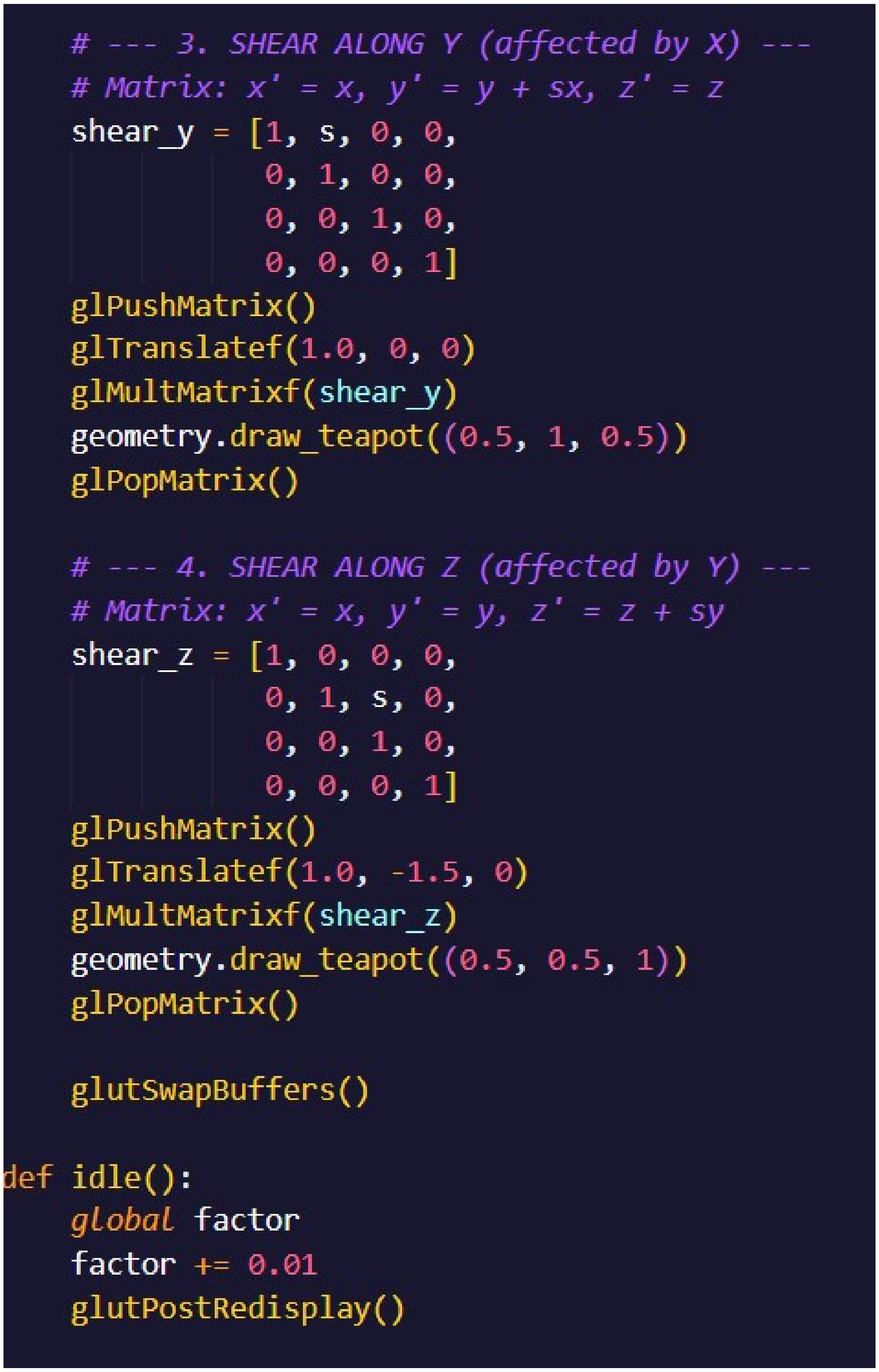
The code is structured into two files:

- **geometry.py**: Contains the `draw_teapot` and axis-drawing functions.
- **main.py**: Contains the shearing logic, where separate teapots are rendered to demonstrate shearing relative to the X, Y, and Z planes.



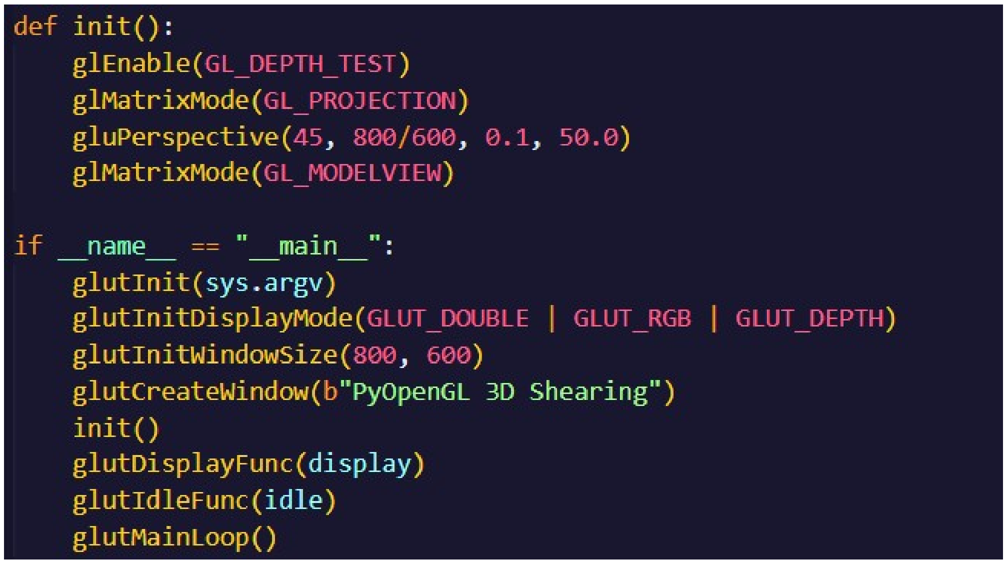
```python
import sys
import math
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
import geometry

factor = 0.0

def display():
    global factor
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glLoadIdentity()
    gluLookAt(3, 3, 5, 0, 0, 0, 0, 1, 0)

    # Use a sine wave to oscillate the shearing effect
    s = math.sin(factor)

    # --- 1. ORIGINAL (No Shear) ---
    glPushMatrix()
    glTranslatef(-2.0, 0, 0)
    geometry.draw_teapot((0.7, 0.7, 0.7))
    glPopMatrix()

    # --- 2. SHEAR ALONG X (affected by Y) ---
    # Matrix: x' = x + sy, y' = y, z' = z
    shear_x = [1, 0, 0, 0,
               s, 1, 0, 0,
               0, 0, 1, 0,
               0, 0, 0, 1]
    glPushMatrix()
    glTranslatef(1.0, 1.5, 0)
    glMultMatrixf(shear_x)
    geometry.draw_teapot((1, 0.5, 0.5))
    glPopMatrix()
```

Figure 9: 3D Shearing Source Code 1



```
    # --- 3. SHEAR ALONG Y (affected by X) ---
    # Matrix: x' = x, y' = y + sx, z' = z
    shear_y = [1, s, 0, 0,
               0, 1, 0, 0,
               0, 0, 1, 0,
               0, 0, 0, 1]
    glPushMatrix()
    glTranslatef(1.0, 0, 0)
    glMultMatrixf(shear_y)
    geometry.draw_teapot((0.5, 1, 0.5))
    glPopMatrix()

    # --- 4. SHEAR ALONG Z (affected by Y) ---
    # Matrix: x' = x, y' = y, z' = z + sy
    shear_z = [1, 0, 0, 0,
               0, 1, s, 0,
               0, 0, 1, 0,
               0, 0, 0, 1]
    glPushMatrix()
    glTranslatef(1.0, -1.5, 0)
    glMultMatrixf(shear_z)
    geometry.draw_teapot((0.5, 0.5, 1))
    glPopMatrix()

    glutSwapBuffers()

def idle():
    global factor
    factor += 0.01
    glutPostRedisplay()
```

Figure 10: 3D Shearing Source Code 2



```python
def init():
    glEnable(GL_DEPTH_TEST)
    glMatrixMode(GL_PROJECTION)
    gluPerspective(45, 800/600, 0.1, 50.0)
    glMatrixMode(GL_MODELVIEW)

if __name__ == "__main__":
    glutInit(sys.argv)
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH)
    glutInitWindowSize(800, 600)
    glutCreateWindow(b"PyOpenGL 3D Shearing")
    init()
    glutDisplayFunc(display)
    glutIdleFunc(idle)
    glutMainLoop()
```

Figure 11: 3D Shearing Source Code 3

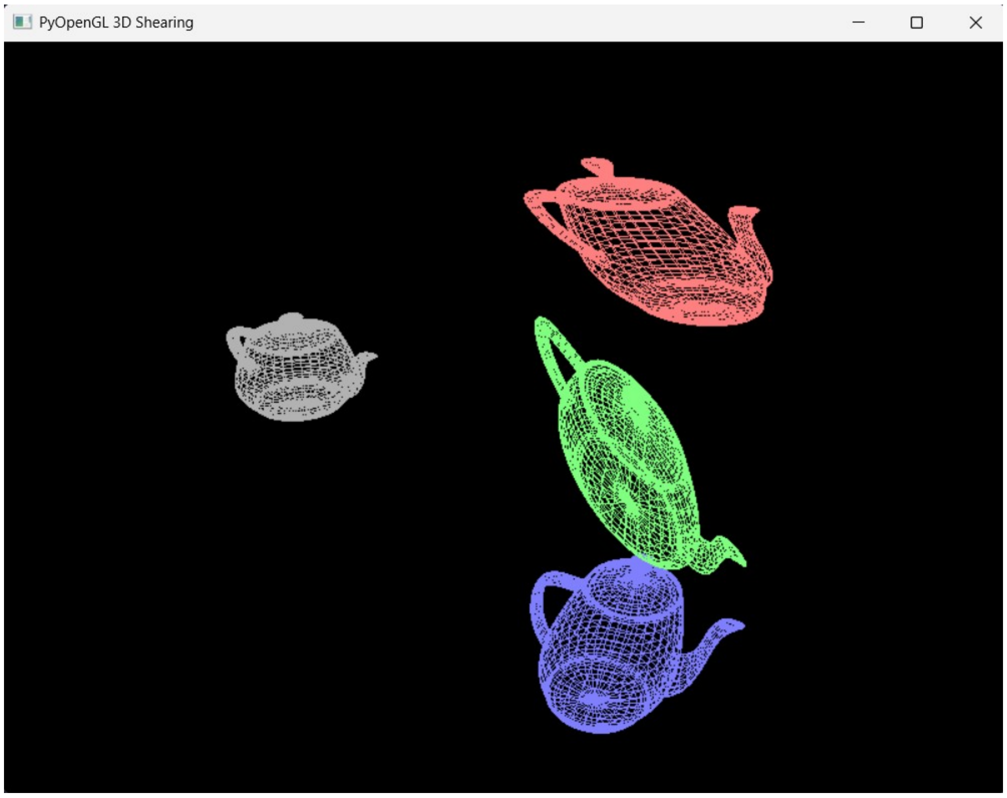


Figure 12: 3D Shearing Output

Conclusion

3D shearing is a powerful tool for simulating deformations. By manually constructing shearing matrices and applying them via `glMultMatrixf`, we can achieve complex distortions that are not possible through simple rotation or scaling. This experiment successfully demonstrates how varying a shear factor over time creates a ”jiggling” or ”slanting” effect on a 3D primitive.

4 Introduction to General 3D Scaling

Scaling is a linear transformation that enlarges or diminishes objects. In the context of 3D computer graphics, **General Scaling** (also known as Uniform Scaling) refers to applying the same scaling factor S to the X, Y, and Z coordinates simultaneously.

This transformation preserves the aspect ratio and proportions of the object, ensuring it does not appear distorted. The mathematical relationship for uniform scaling is:

$$x' = S \cdot x, \quad y' = S \cdot y, \quad z' = S \cdot z$$

The corresponding transformation matrix for a uniform scale factor S is represented as:

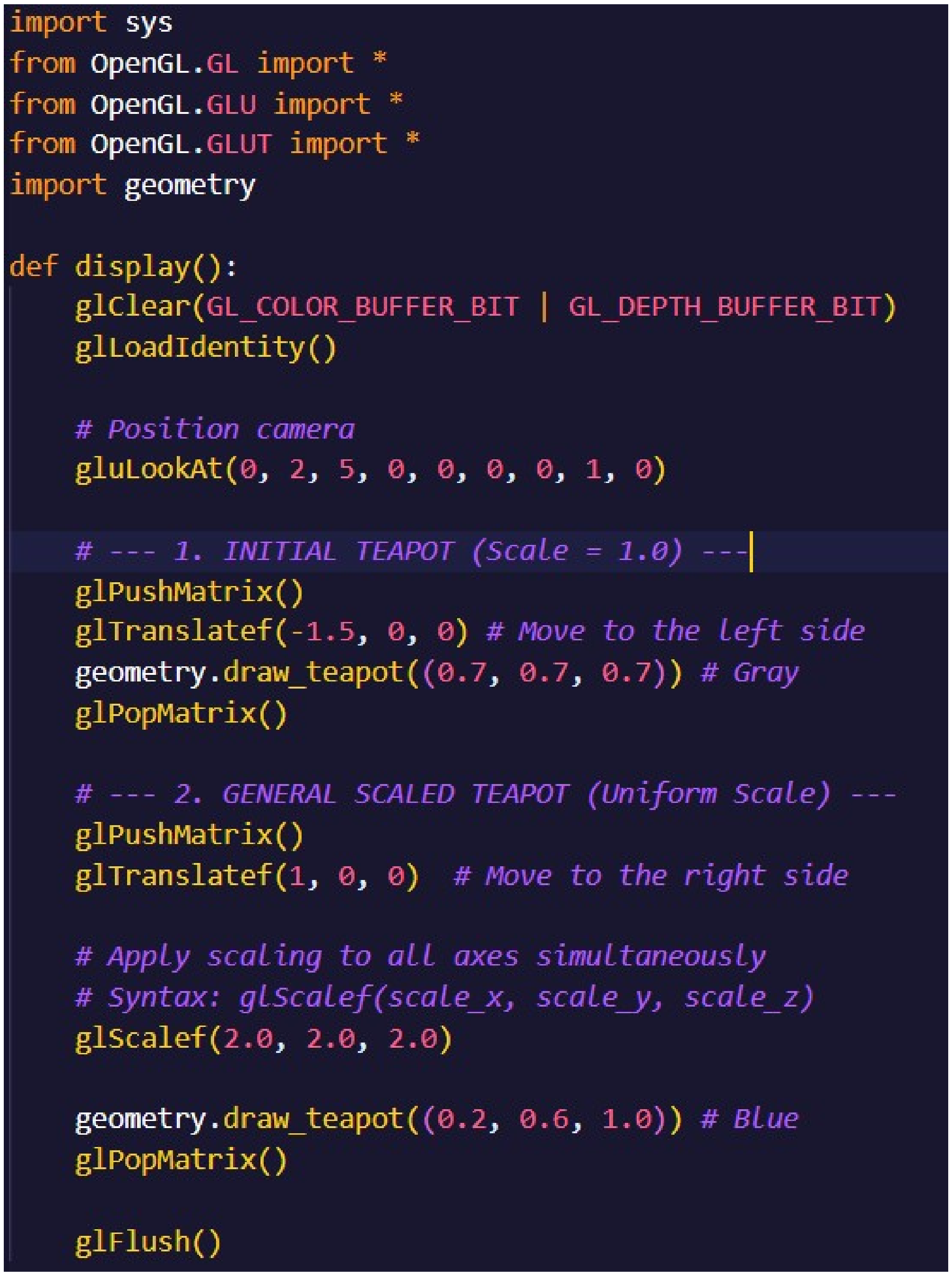
$$S_{uniform} = \begin{bmatrix} S & 0 & 0 & 0 \\ 0 & S & 0 & 0 \\ 0 & 0 & S & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Implementation Strategy

The program is implemented using a modular approach to visualize the difference between an unscaled unit and a scaled unit.

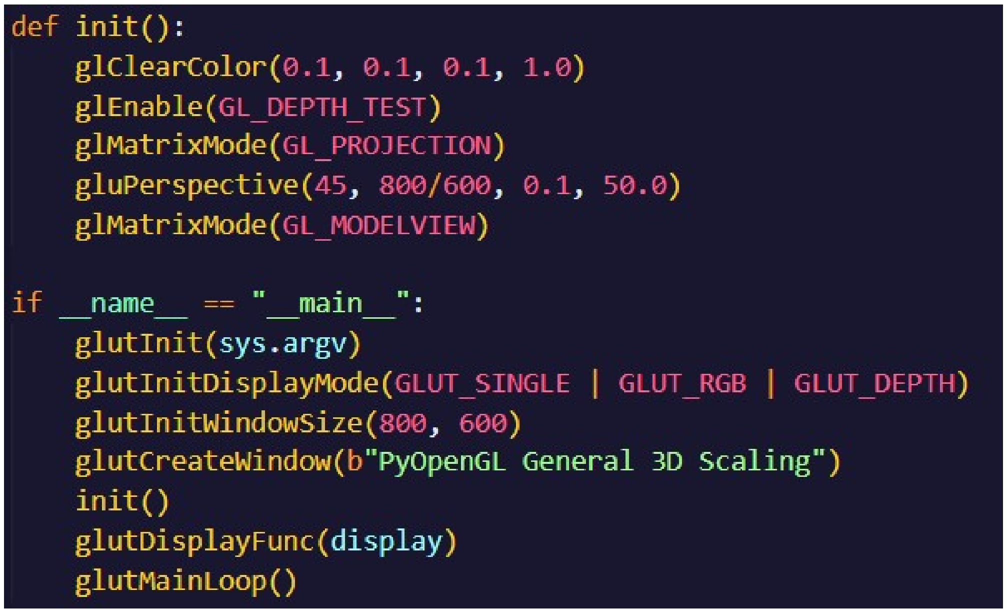
Implementation Definition

- **Geometry Module:** Provides a standardized function to render the 3D teapot primitive using the GLUT library.
- **Transformation Block:** The application uses the `glScalef(S, S, S)` function. By passing identical values for all three parameters, the OpenGL state machine applies a uniform transformation matrix to the vertices of the object.
- **Static Comparison:** Two instances of the teapot are rendered in the same scene—one with the identity matrix (original) and one with the scaled matrix—to provide a clear visual reference of the size increase.



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

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

    # Position camera
    gluLookAt(0, 2, 5, 0, 0, 0, 0, 1, 0)

    # --- 1. INITIAL TEAPOT (Scale = 1.0) ---
    glPushMatrix()
    glTranslatef(-1.5, 0, 0) # Move to the left side
    geometry.draw_teapot((0.7, 0.7, 0.7)) # Gray
    glPopMatrix()

    # --- 2. GENERAL SCALED TEAPOT (Uniform Scale) ---
    glPushMatrix()
    glTranslatef(1, 0, 0)  # Move to the right side

    # Apply scaling to all axes simultaneously
    # Syntax: glScalef(scale_x, scale_y, scale_z)
    glScalef(2.0, 2.0, 2.0)

    geometry.draw_teapot((0.2, 0.6, 1.0)) # Blue
    glPopMatrix()

    glFlush()
```

Figure 13: 3D Scaling Source Code 1



```python
def init():
    glClearColor(0.1, 0.1, 0.1, 1.0)
    glEnable(GL_DEPTH_TEST)
    glMatrixMode(GL_PROJECTION)
    gluPerspective(45, 800/600, 0.1, 50.0)
    glMatrixMode(GL_MODELVIEW)

if __name__ == "__main__":
    glutInit(sys.argv)
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH)
    glutInitWindowSize(800, 600)
    glutCreateWindow(b"PyOpenGL General 3D Scaling")
    init()
    glutDisplayFunc(display)
    glutMainLoop()
```

Figure 14: 3D Scaling Source Code 2

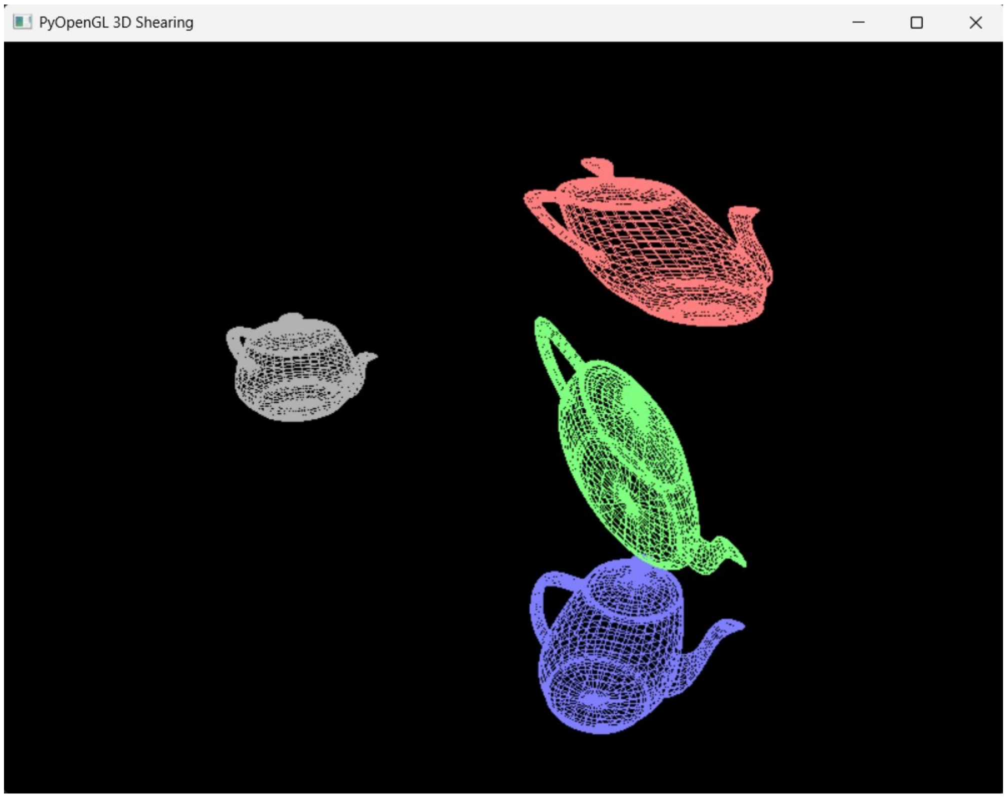


Figure 15: 3D Scaling Output

Conclusion

General 3D scaling is an essential operation for managing the relative sizes of models within a virtual environment. By applying a uniform factor across all coordinate axes, we maintain the geometric integrity of the object while adjusting its volume. This implementation demonstrates how PyOpenGL efficiently handles these matrix operations to produce predictable and proportional visual results.

5 Introduction

Orthographic projection is a means of representing three-dimensional objects in two dimensions. Unlike perspective projection, orthographic projection is a form of **parallel projection** where all the projection lines are orthogonal to the projection plane.

In computer graphics, this is primarily used for technical drawings (blueprints) or 2D games, as it preserves the actual dimensions and parallelism of objects regardless of their distance from the camera. There is no ”vanishing point” in this projection.

Mathematical Formulation

The goal of the orthographic projection matrix is to map a specified viewing volume (a box) into the Canonical View Volume, which is a cube ranging from -1 to 1 in all axes.

The transformation is defined by the parameters: $left(l)$, $right(r)$, $bottom(b)$, $top(t)$, $near(n)$, and $far(f)$. The matrix used by OpenGL for `glOrtho` is:

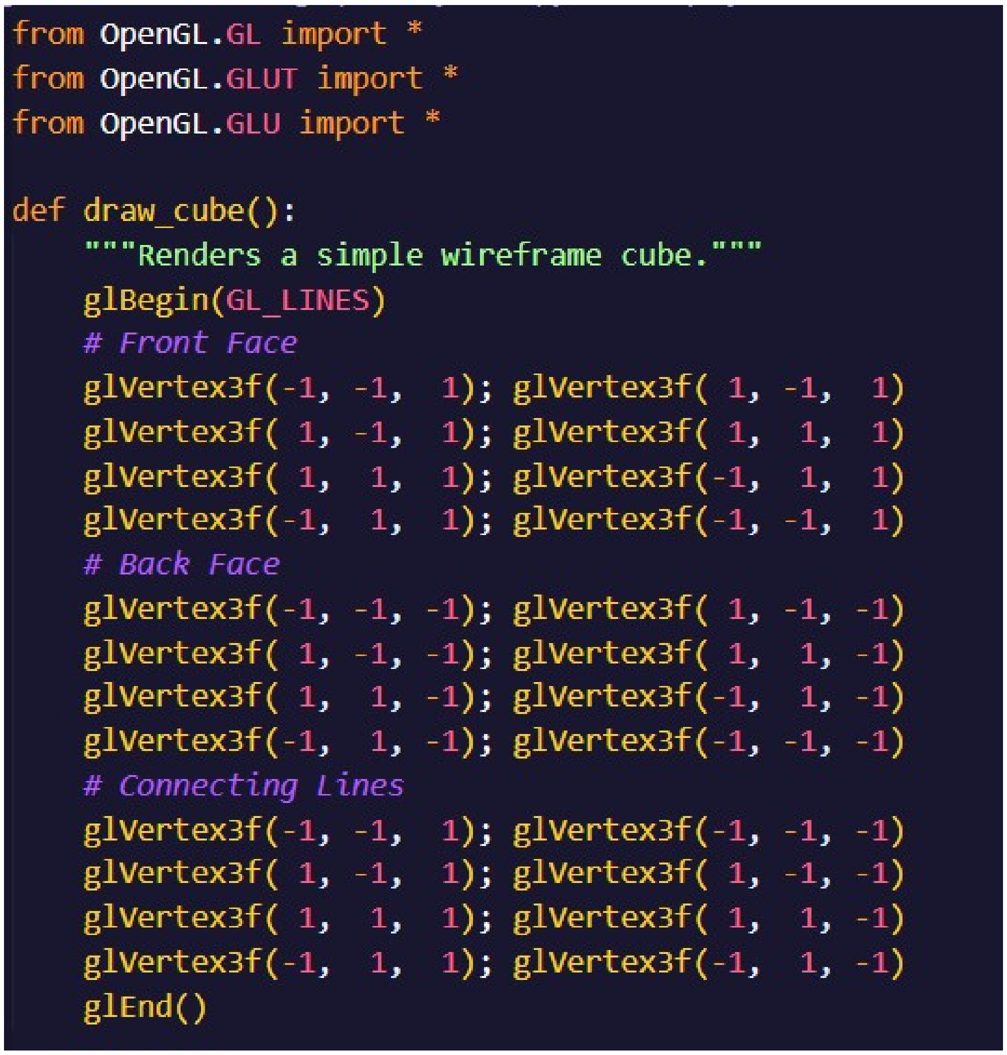
$$P = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Formula Explanation

- **Scaling Terms:** The diagonal elements $\frac{2}{r-l}$, $\frac{2}{t-b}$, and $\frac{-2}{f-n}$ scale the width, height, and depth of the user-defined box down to the size of the canonical cube (which has a total width of 2).
- **Translation Terms:** The rightmost column shifts the center of the user’s volume to the origin $(0, 0, 0)$ of the clip space.
- **Linearity:** Notice the w component (the last row) is $[0, 0, 0, 1]$. This ensures that the depth z does not affect the x and y scaling, maintaining the parallel nature of the projection.

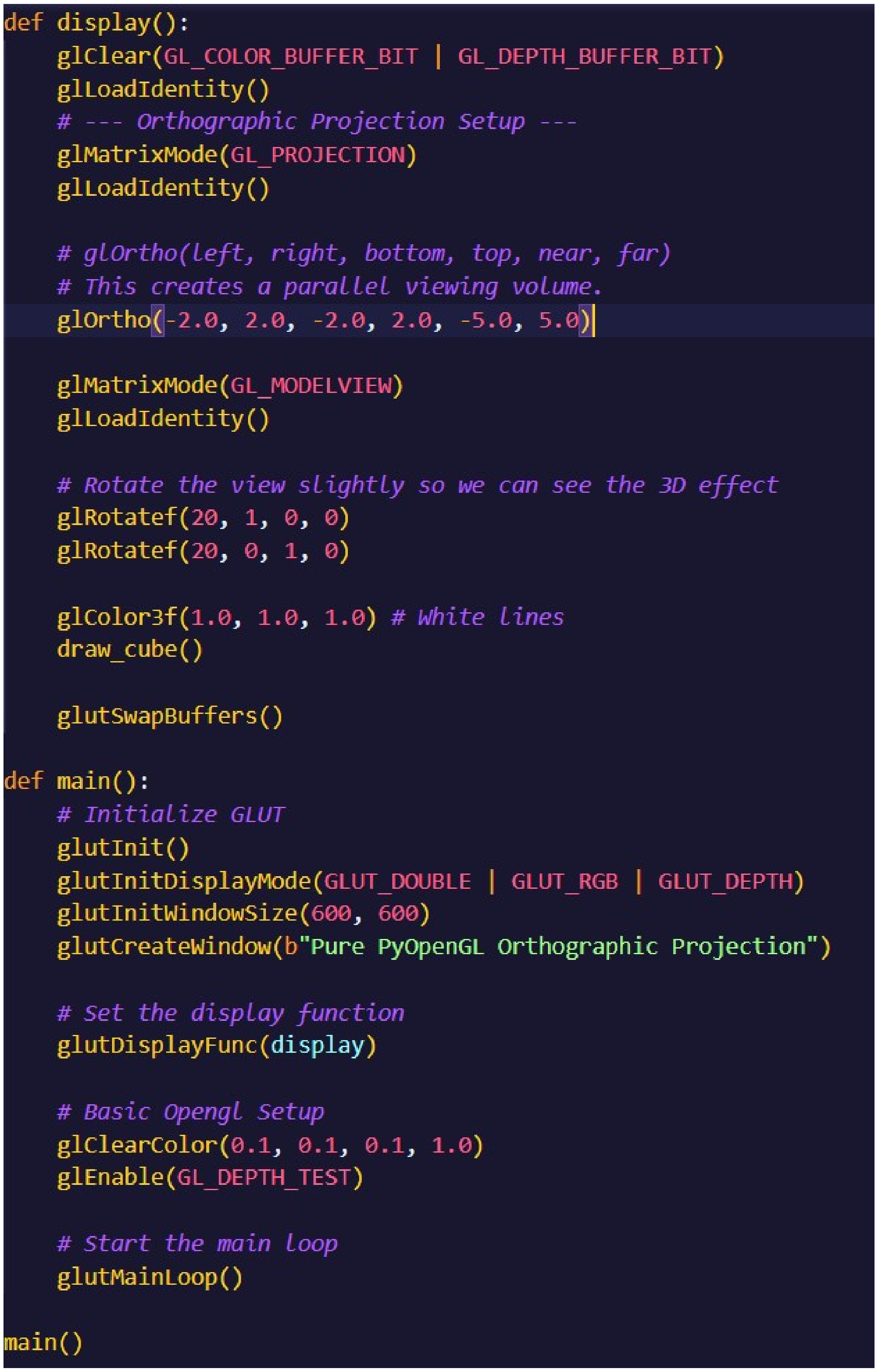
Code Implementation

The following Python implementation utilizes `PyOpenGL` to render a wireframe cube within an orthographic volume.



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

def draw_cube():
    """Renders a simple wireframe cube."""
    glBegin(GL_LINES)
    # Front Face
    glVertex3f(-1, -1,  1); glVertex3f( 1, -1,  1)
    glVertex3f( 1, -1,  1); glVertex3f( 1,  1,  1)
    glVertex3f( 1,  1,  1); glVertex3f(-1,  1,  1)
    glVertex3f(-1,  1,  1); glVertex3f(-1, -1,  1)
    # Back Face
    glVertex3f(-1, -1, -1); glVertex3f( 1, -1, -1)
    glVertex3f( 1, -1, -1); glVertex3f( 1,  1, -1)
    glVertex3f( 1,  1, -1); glVertex3f(-1,  1, -1)
    glVertex3f(-1,  1, -1); glVertex3f(-1, -1, -1)
    # Connecting Lines
    glVertex3f(-1, -1,  1); glVertex3f(-1, -1, -1)
    glVertex3f( 1, -1,  1); glVertex3f( 1, -1, -1)
    glVertex3f( 1,  1,  1); glVertex3f( 1,  1, -1)
    glVertex3f(-1,  1,  1); glVertex3f(-1,  1, -1)
    glEnd()
```

Figure 16: Orthographic Projection Source Code 1



```python
def display():
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glLoadIdentity()
    # --- Orthographic Projection Setup ---
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()

    # glOrtho(left, right, bottom, top, near, far)
    # This creates a parallel viewing volume.
    glOrtho(-2.0, 2.0, -2.0, 2.0, -5.0, 5.0)

    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()

    # Rotate the view slightly so we can see the 3D effect
    glRotatef(20, 1, 0, 0)
    glRotatef(20, 0, 1, 0)

    glColor3f(1.0, 1.0, 1.0) # White lines
    draw_cube()

    glutSwapBuffers()

def main():
    # Initialize GLUT
    glutInit()
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH)
    glutInitWindowSize(600, 600)
    glutCreateWindow(b"Pure PyOpenGL Orthographic Projection")

    # Set the display function
    glutDisplayFunc(display)

    # Basic Opengl Setup
    glClearColor(0.1, 0.1, 0.1, 1.0)
    glEnable(GL_DEPTH_TEST)

    # Start the main loop
    glutMainLoop()

main()
```

Figure 17: Orthographic Projection Source Code 2

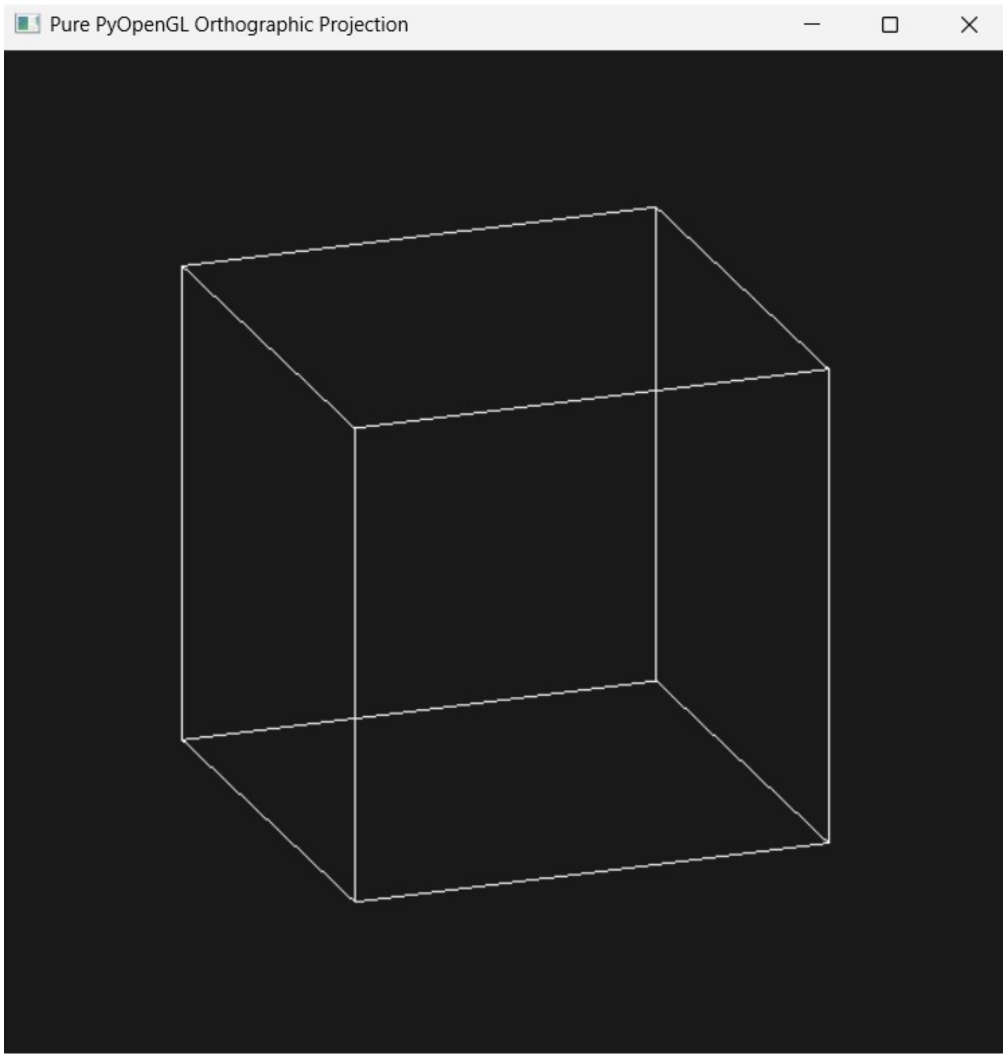


Figure 18: Orthographic Projection Output

Conclusion

The implementation demonstrates how `glOrtho` creates a parallel viewing frustum. By mapping the world coordinates to normalized device coordinates without a perspective divide, we achieve a view where object size remains constant. This is essential for applications requiring precision and accurate scale, such as CAD software and architectural visualization.