Kathmandu University
Department of Computer Science and Engineering
Dhulikhel, Kavre

Project Report
on

”Interactive 3D Arbitrary Rotation Visualizer”

[Code No: COMP 342]
Computer Graphics

Submitted by:

Salina Nakarmi (34)
Suniti Shrestha (51)

Submitted to:

Mr. Dhiraj Shrestha
Department of Computer Science and Engineering

Submission Date: February 5, 2026

Contents

Chapter 1. Introduction

1.1 Objective

Develop an interactive 3D visualization tool that demonstrates rotation of objects about an arbitrary axis in 3D space using transformation matrices.

1.2 Motivation

In our recent lecture on 3D transformations, the concept of rotating objects about an arbitrary axis (not aligned with X, Y, or Z) proved challenging to grasp from equations alone. This project aims to create an educational tool that visualizes the step-by-step transformation process, making the abstract mathematical concepts tangible and understandable through interactive 3D graphics.

1.3 Problem Statement

Given:

- An arbitrary rotation axis defined by two points P and P in 3D space
- A 3D object to be rotated
- A desired rotation angle

Challenge: Rotate the object about this arbitrary axis using fundamental transformation matrices (translation and rotation about coordinate axes).

Chapter 2. Project Description

2.1 Technical Approach

The project implements the standard computer graphics algorithm for arbitrary axis rotation:

The rotation of a point P about an arbitrary axis defined by points P_1 and P_2 through angle θ is achieved by composing the following transformations:

$$M_{final} = T^{-1} \cdot R_z^{-1}(\alpha) \cdot R_y^{-1}(\beta) \cdot R_x(\theta) \cdot R_y(\beta) \cdot R_z(\alpha) \cdot T \tag{2.1}$$

$$M_{final} = T^{-1} \cdot R_z(\alpha) \cdot R_y(\beta) \cdot R_x(\theta) \cdot R_y(-\beta) \cdot R_z(-\alpha) \cdot T \tag{2.2}$$

where:

- T is the translation matrix that moves P_1 to the origin
- $R_z(-\alpha)$ rotates about the Z-axis to align the axis with the XZ plane
- $R_y(-\beta)$ rotates about the Y-axis to align the axis with the X-axis
- $R_x(\theta)$ performs the desired rotation
- The remaining transformations are inverses that restore the original coordinate system

2.2 Implementation

2.2.1 Technology Stack

Implemented in Python 3.x with:

- **Pygame**: 2D rendering of 3D projections
- **NumPy**: Matrix operations and transformations
- **Math**: Trigonometric functions

Software rendering provides educational clarity and portability without requiring specialized graphics hardware.

2.2.2 Coordinate System and Projection

Uses right-handed coordinates: X (red) points right, Y (green) points up, Z (blue) points toward viewer.
Perspective projection formula:

$$\text{screen}_x = \frac{W}{2} + \frac{x \cdot \text{scale}}{z+d}, \quad \text{screen}_y = \frac{H}{2} - \frac{y \cdot \text{scale}}{z+d} \tag{2.3}$$

2.3 Key Features

2.3.1 Visualization Elements

- Color-coded coordinate axes and semi-transparent reference planes
- Depth sorting using painter's algorithm

2.3.2 Controls

- **Camera**: Pitch/yaw (K/J/H/L), zoom (Q/E), reset (R)
- **Navigation**: Arrow keys for step-by-step progression
- **Animation**: Space bar to play/pause, 60 FPS rendering
- **Views**: Toggle between perspective and orthographic projections

2.4 Source Code Highlights

This section presents key portions of the implementation that demonstrate the core functionality.

2.4.1 3D to 2D Projection

```python
def rotate_point(point, rot_x, rot_y):
    """Rotate a 3D point based on camera angles"""
    x, y, z = point

    # Convert degrees to radians
    angle_x = math.radians(rot_x)
    angle_y = math.radians(rot_y)

    # Rotate around Y-axis first (yaw)
    # FIXED: Negated sin_y to make X point RIGHT when Z points out
    cos_y = math.cos(angle_y)
    sin_y = math.sin(angle_y)
    temp_x = x * cos_y + z * sin_y  # CHANGED: was (- z * sin_y)
    temp_z = -x * sin_y + z * cos_y  # CHANGED: was (+ z * cos_y)
    x, z = temp_x, temp_z

    # Then rotate around X-axis (pitch)
    cos_x = math.cos(angle_x)
    sin_x = math.sin(angle_x)
    temp_y = y * cos_x - z * sin_x
    temp_z = y * sin_x + z * cos_x
    y, z = temp_y, temp_z

    return (x, y, z)

def project_3d(point):
    """
    Project 3D point to 2D screen coordinates using perspective projection      You, 4
    """
    # First apply camera rotation
    x, y, z = rotate_point(point, rotation_x, rotation_y)

    # Move scene away from camera
    z = z + camera_distance

    # Prevent division by zero
    if z <= 0.1:
        z = 0.1

    # Perspective projection
    factor = scale / z
    screen_x = int(WIDTH / 2 + x * factor)
    screen_y = int(HEIGHT / 2 - y * factor)  # Negative because screen Y goes down

    return (screen_x, screen_y, z)
```

Figure 2.1: Perspective Projection Implementation

2.4.2 Transformation Matrices

```python
def translation_matrix(tx, ty, tz):
    """Create translation matrix"""
    return np.array([
        [1, 0, 0, tx],
        [0, 1, 0, ty],
        [0, 0, 1, tz],
        [0, 0, 0, 1]
    ], dtype=float)

def rotation_z_matrix(angle):
    """Create rotation matrix around Z-axis"""
    c = math.cos(angle)
    s = math.sin(angle)
    return np.array([
        [c, -s, 0, 0],
        [s, c, 0, 0],
        [0, 0, 1, 0],
        [0, 0, 0, 1]
    ], dtype=float)

def rotation_y_matrix(angle):
    """Create rotation matrix around Y-axis"""
    c = math.cos(angle)
    s = math.sin(angle)
    return np.array([
        [c, 0, s, 0],
        [0, 1, 0, 0],
        [-s, 0, c, 0],
        [0, 0, 0, 1]
    ], dtype=float)

def rotation_x_matrix(angle):
    """Create rotation matrix around X-axis"""
    c = math.cos(angle)
    s = math.sin(angle)
    return np.array([
        [1, 0, 0, 0],
        [0, c, -s, 0],
        [0, s, c, 0],
        [0, 0, 0, 1]
    ], dtype=float)
```

Figure 2.2: Transformation Matrix Functions

2.4.3 Transformation Implementation:

```python
# Transfromation Steps:

def step_0_original(point, p1, p2):
    """Original point (no transformation)"""
    return point, p1, p2, None

def step_1_translate(point, p1, p2):
    """Translate so that P1 is at origin"""
    tx, ty, tz = -p1[0], -p1[1], -p1[2]
    T = translation_matrix(tx, ty, tz)
    new_point = apply_transformation(point, T)
    new_p1 = apply_transformation(p1, T)
    new_p2 = apply_transformation(p2, T)
    return new_point, new_p1, new_p2, {'T': (tx, ty, tz)}

def step_2_rotate_z(point, p1, p2):
    """Rotate around Z-axis to align P2 with XZ plane"""
    #  First apply step 1
    point, p1, p2, _ = step_1_translate(point, p1, p2)

    #Calculate alpha
    a, b, c = normalize_vector(p2)
    d = math.sqrt(a*a + b*b) #cause viewed from z axis in the XY plane

    if d > 1e-10:
        cos_alpha = a / d
        sin_alpha = b / d
        alpha = math.atan2(sin_alpha, cos_alpha)
    else:
        alpha = 0

    #Apply Rz(-alpha)
    Rz = rotation_z_matrix(-alpha)
    new_point = apply_transformation(point, Rz)
    new_p1 = apply_transformation(p1, Rz)
    new_p2 = apply_transformation(p2, Rz)

    return new_point, new_p1, new_p2, {
        'alpha': alpha, 'd': d, 'a': a, 'b': b, 'c': c
        }


def step_3_rotate_y(point, p1, p2):
    """Step 3: Rotate about Y-axis to align with X-axis"""
    # Apply steps 1 and 2
    point, p1, p2, info = step_2_rotate_z(point, p1, p2)
    alpha, d, a, b, c = info['alpha'], info['d'], info['a'], info['b'], info['c']

    # Calculate beta
    cos_beta = d
    sin_beta = -c
    beta = math.atan2(sin_beta, cos_beta)
    info['beta'] = beta

    # Apply Ry(-beta)
    Ry = rotation_y_matrix(-beta)
    new_point = apply_transformation(point, Ry)
    new_p1 = apply_transformation(p1, Ry)
    new_p2 = apply_transformation(p2, Ry)

    return new_point, new_p1, new_p2, info
```

Figure 2.3: Bringing the arbitrary line to Z-axis

```python
def step_4_rotate_x(point, p1, p2, theta):
    """Step 4: Rotate about X-axis by angle theta"""
    #Apply step 1,2,3
    point, p1, p2, info = step_3_rotate_y(point, p1, p2)

    # Apply Rx(theta)
    Rx = rotation_x_matrix(theta)
    new_point = apply_transformation(point, Rx)
    new_p1 = apply_transformation(p1, Rx)
    new_p2 = apply_transformation(p2, Rx)

    return new_point, new_p1, new_p2, info

def step_5_inverse(point, p1_orig, p2_orig, theta):
    """Step 5: Apply inverse transformations"""
    # Get to step 4
    point, p1, p2, info = step_4_rotate_x(point, p1_orig, p2_orig, theta)
    alpha, beta = info['alpha'], info['beta']

    # Apply inverse: Ry(beta), Rz(alpha), Translate back
    Ry_inv = rotation_y_matrix(beta)
    Rz_inv = rotation_z_matrix(alpha)
    T_inv = translation_matrix(p1_orig[0], p1_orig[1], p1_orig[2])

    # === INVERSE Y ===
    new_point = apply_transformation(point, Ry_inv)
    new_p1 = apply_transformation(p1, Ry_inv)
    new_p2 = apply_transformation(p2, Ry_inv)

    # === INVERSE Z ===
    new_point = apply_transformation(new_point, Rz_inv)
    new_p1 = apply_transformation(new_p1, Rz_inv)
    new_p2 = apply_transformation(new_p2, Rz_inv)

    # === INVERSE TRANSLATION ===
    new_point = apply_transformation(new_point, T_inv)
    new_p1 = apply_transformation(new_p1, T_inv)
    new_p2 = apply_transformation(new_p2, T_inv)
```

Figure 2.4: X-axis Rotation Step and reverse back to the position

2.5 Program Output and Visualization

2.5.1 User Interface

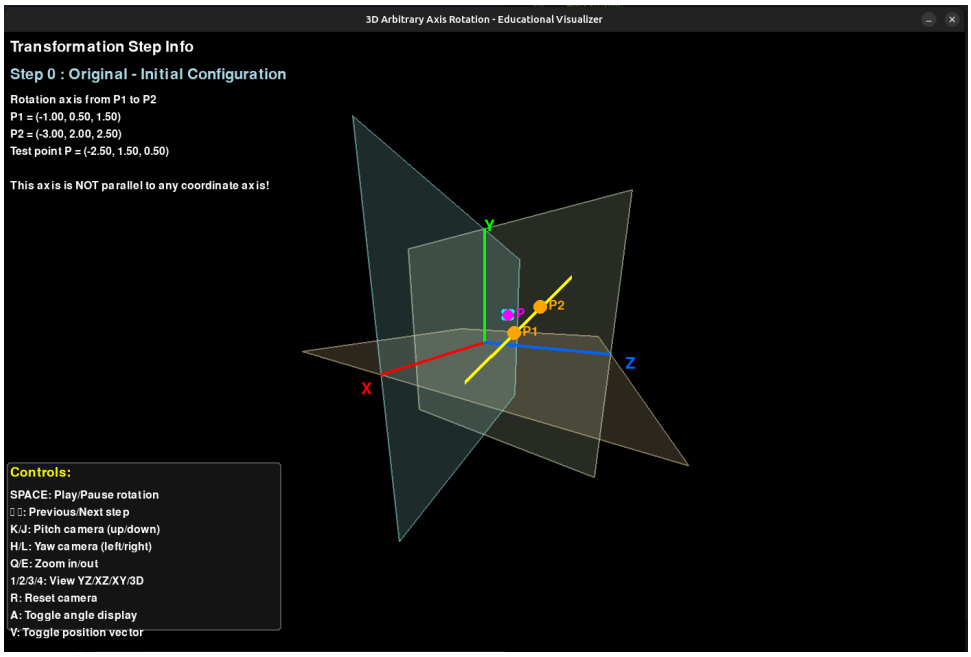

Figure 2.5: UI

The application window features a central viewport displaying the 3D scene with color-coded coordinate axes (red X, green Y, blue Z), semi-transparent reference planes, and a bright yellow rotation axis marked by orange points at P_1 and P_2. A cyan wireframe cube serves as the test object, with information and control panels on the left showing transformation details and keyboard shortcuts. All elements update in real-time during the step-by-step transformation sequence.

2.5.2 Transformation Steps Visualization

Each step provides specific visual feedback:
Step 0 - Original: Shows the initial configuration with the arbitrary axis and object in their original positions.

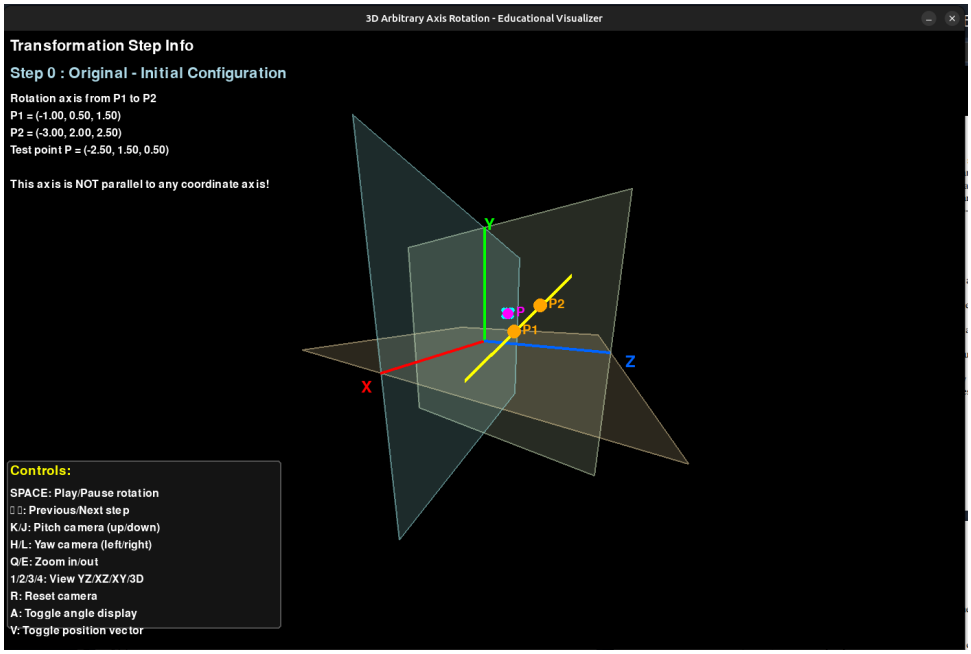

Figure 2.6: Step 0 Original

Step 1 - Translation: The axis point P_1 moves to the origin, with the entire system translated accordingly.

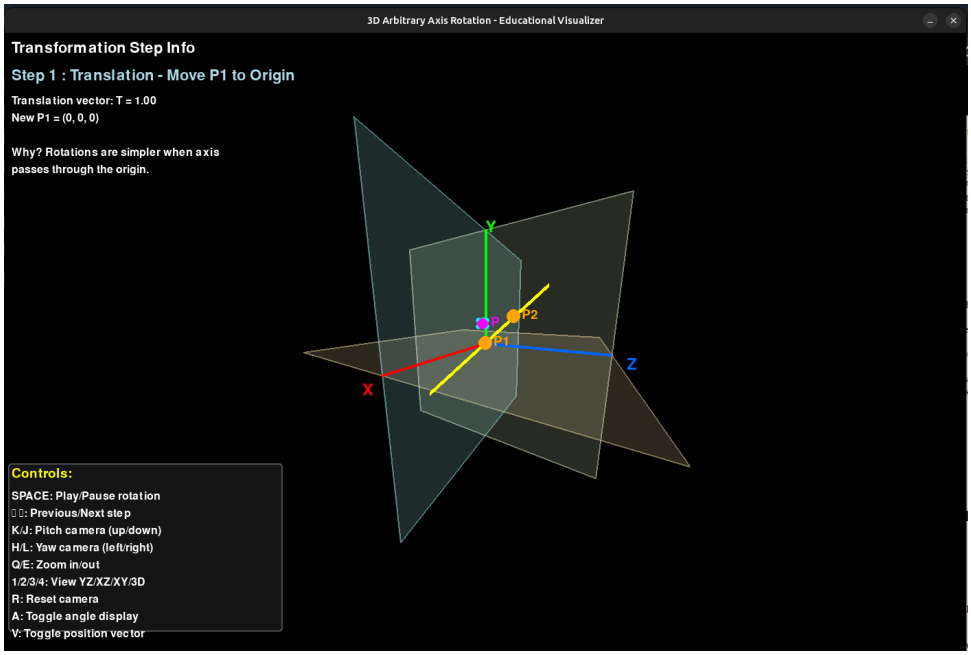

Figure 2.7: Step 1 - Translation

Step 2 - Rotate Z: Purple arc indicator shows the rotation angle α about the Z-axis as the system aligns to the XZ plane.

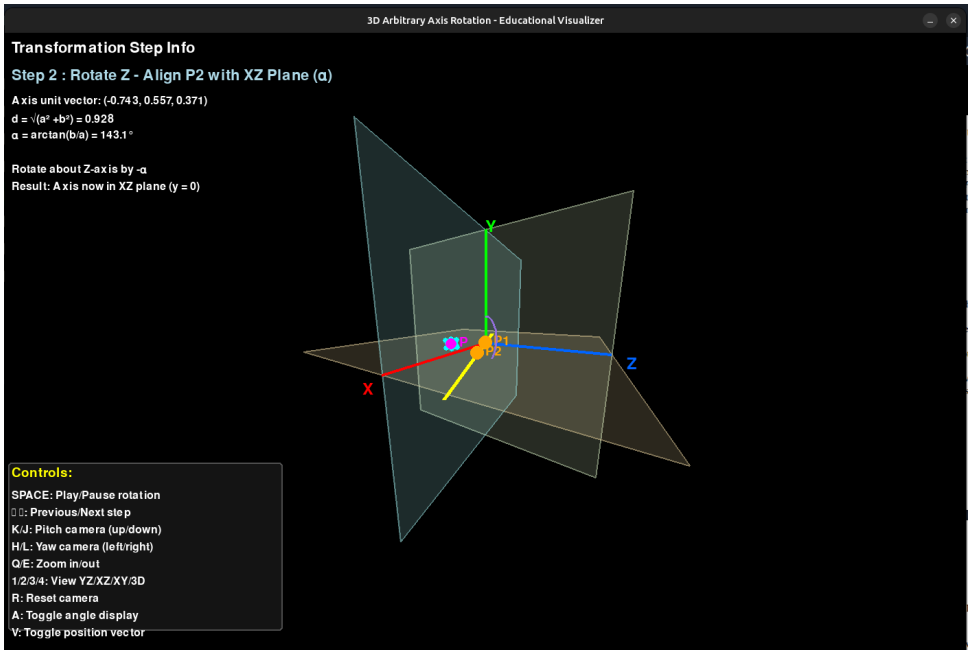

Figure 2.8: Step 2 - Rotate Z

Step 3 - Rotate Y: Cyan arc indicator shows the rotation angle β about the Y-axis as the axis aligns with the X-axis.

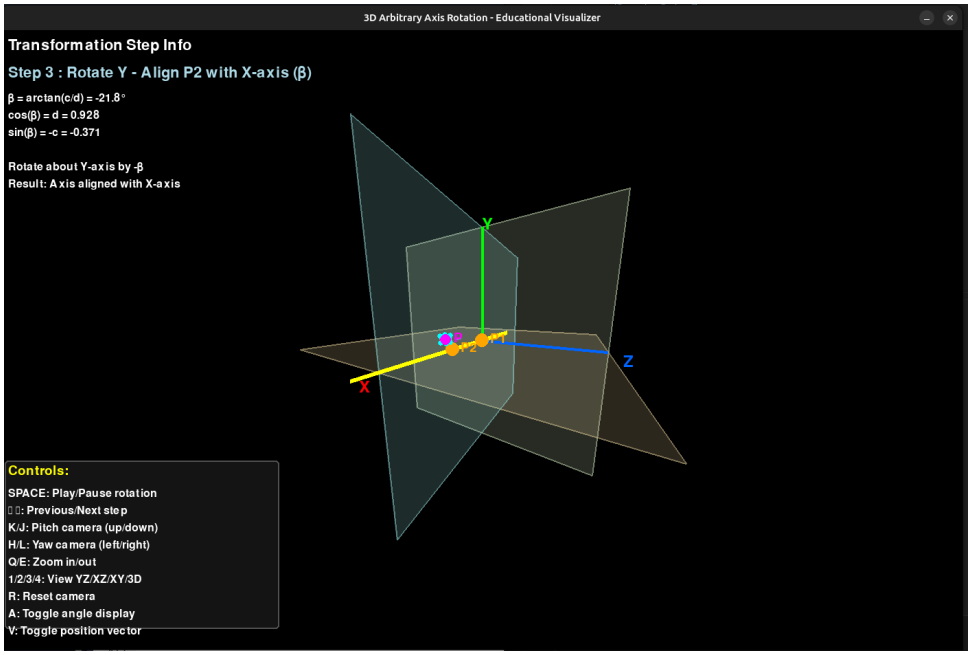

Figure 2.9: Step 3 - Rotate Y

Step 4 - Rotate X: The object rotates about the now-aligned X-axis by the desired angle θ.

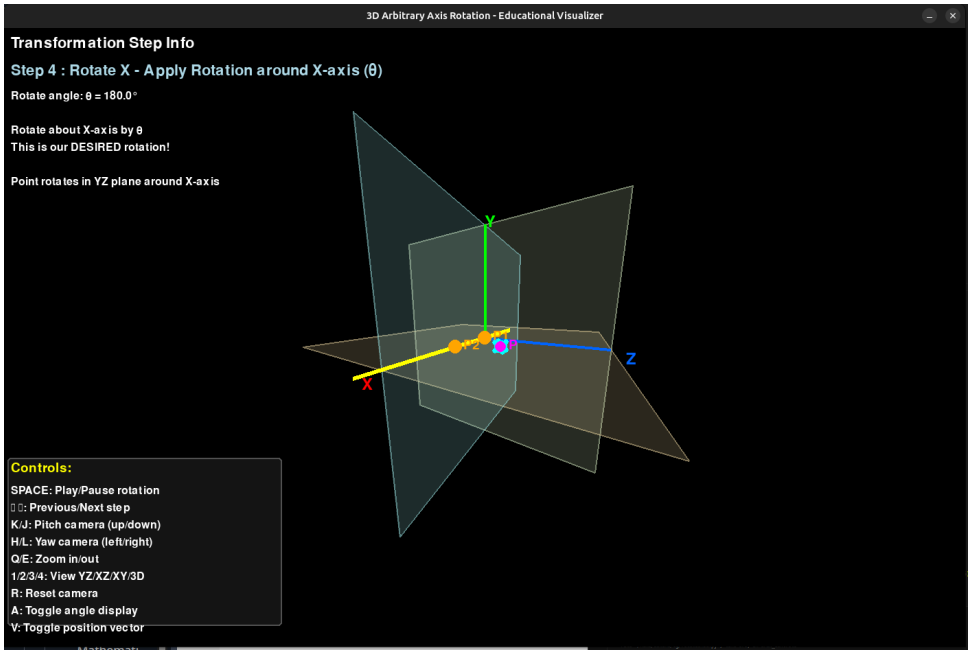

Figure 2.10: Step 4 - Rotate X

Step 5 - Inverse: All transformations are reversed, showing the final result—the object rotated about the original arbitrary axis.

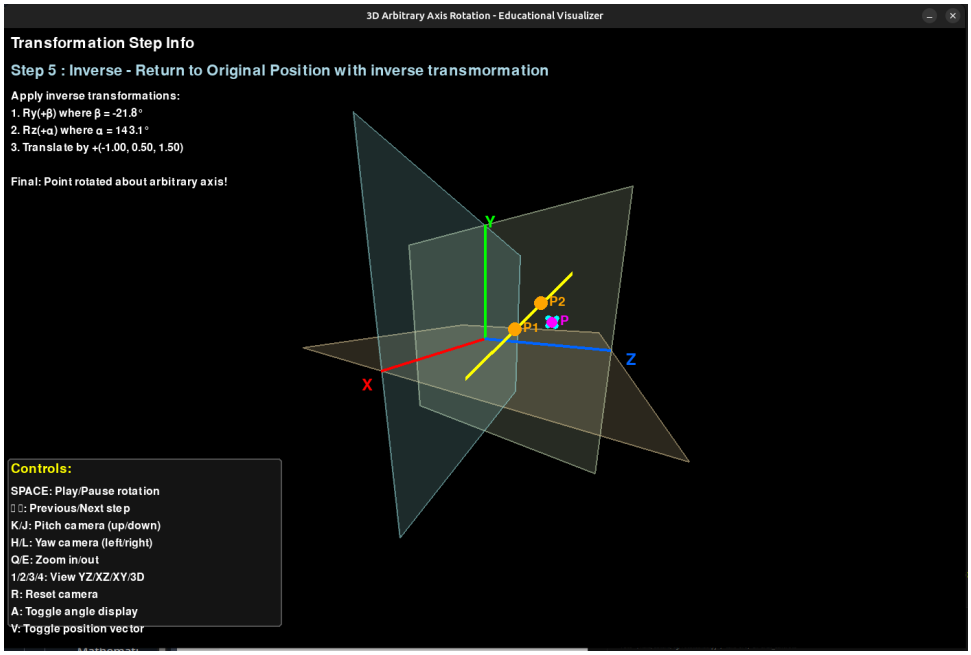

Figure 2.11: Step 5 - Inverse

2.5.3 View Modes

The application supports multiple view modes to help understand the transformations from different perspectives:

- **3D View** (default): Free-form perspective view with full camera control

- **YZ View**: Orthographic projection looking down the X-axis
- **XZ View**: Orthographic projection looking down the Y-axis
- **XY View**: Orthographic projection looking down the Z-axis

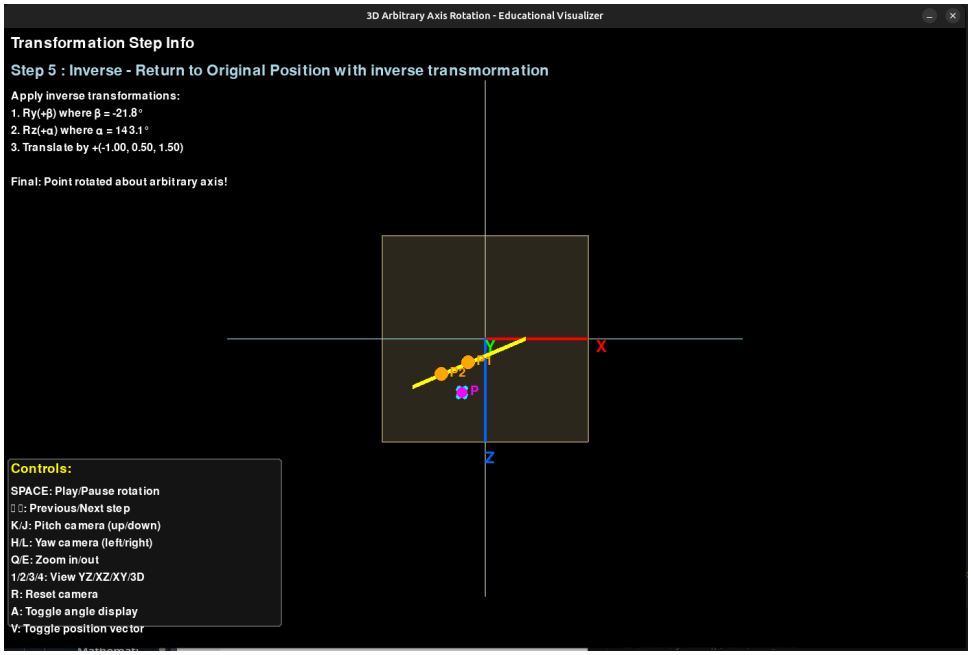

Figure 2.12: 2D - XZ View

These orthographic views are particularly useful for verifying that transformations align the rotation axis with specific coordinate planes and axes.

Chapter 3. Conclusion

The Interactive 3D Arbitrary Axis Rotation Visualizer successfully achieves its goal of making a complex computer graphics concept accessible through interactive visualization. By implementing the mathematical foundation with clarity and providing intuitive controls for exploration, the tool serves as both a learning aid for students and a demonstration of fundamental transformation techniques.

The project reinforces the importance of visualization in understanding abstract mathematical concepts. While equations and matrices provide the precise formulation needed for implementation, seeing the transformations in action and being able to manipulate them interactively creates a deeper, more intuitive understanding that complements the mathematical knowledge.

We believe this tool will be valuable for students learning computer graphics. The experience of building this project has greatly enhanced our understanding of 3D transformations and equipped us with practical skills in graphics programming that will be valuable in future work.

References

[1] Hearn, D., Baker, M. P., & Carithers, W. (2011). *Computer Graphics with OpenGL* (4th ed.). Pearson.

[2] Pygame Development Team. (2023). Pygame Documentation. Retrieved from https://www.pygame.org/docs/