Mesh Viewer

This program reads in a 3D object file and displays it using orthographic projection and scaling. The user can rotate the object using click and drag mouse functionality.

Dependencies

This program requires the following dependencies:

- Python 3.6+
- NumPy
- pygame

OOP Design

This program utilizes OOP principles to separate the mesh data and the visualization code. There are three main classes:

- **Reader**: This class reads the object file and extracts the vertices and faces data.
- **Object**: This class represents the mesh data and provides methods to project, scale, and rotate the mesh.
- **MeshViewer**: This class handles the visualization of the mesh using the pygame library.

Usage

To use this project, follow these three simple steps:

1. Extract the zipfile and open the extracted folder.
2. Open a terminal window and navigate to the `src` folder using the following command: `cd ./src`.
3. Run the `main.py` file using the following command: `python main.py`.

Once the program is running, you will see a window displaying the 3D object. The vertices are represented as small, blue circles, and the faces are colored based on their angle with the Z-axis.

To rotate the object, click and drag the mouse within the window. Horizontal movement of the mouse will rotate the object about the Y-axis, while vertical movement will rotate the object about the X-axis. Diagonal movement will rotate the object accordingly.

Calculations

To visualize 2D object, we use orthographic projection;

For simplicity and speed, Painter's Algorithm is used to solve culling problem

1. Orthographic Projection

orthographic projection matrix is defined as a 4x4 matrix with values of 1 in the upper left 2x2 matrix and zeros everywhere else except for the lower right corner, which has a value of 1.

Before we know that ,we need several preface

- View To Projection

- Definition: Project the 3D coordinates to 2D plane. The resulting scaled vertex is a 3x1 homogeneous coordinate vector, but the z-coordinate is discarded to obtain the 2D coordinate
- Math formula

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Code

```
def orthographic_projection(self):
    # Define projection matrix
    proj_matrix = np.array([[1, 0, 0, 0],
                            [0, 1, 0, 0],
                            [0, 0, 0, 0],
                            [0, 0, 0, 1]])

    self.projected_vertices = {}
    # Define projection function
    for key, value in self.vertices.items():
        vertex = np.array(value + (1,))
        proj = np.dot(proj_matrix, vertex)[:2]
        self.projected_vertices[key] = proj
```

- World to View
  - Definition: The scaling and translation transformation matrix is defined as a 3x3 matrix that scales the vertices by a scaling factor and translates them to the center of the screen
  - Math formula

$$M = \begin{pmatrix} scale & 0 & \frac{width}{2} \\ 0 & -scale & \frac{height}{2} \\ 0 & 0 & 1 \end{pmatrix}$$

  - code

```
def scale_object(self, height, width, scale):
    # Define transformation matrices
    transformation_matrix = np.array([[scale, 0, width / 2],
                                      [0, -scale, height / 2],
                                      [0, 0, 1]])

    # Scale and translate each vertex
    self.scaled_vertices = {}
    for key, value in self.projected_vertices.items():
        # Convert to homogeneous coordinates
        vertex = np.array(tuple(value) + (1,))

        # Apply transformation matrices
        transformed_vertex = np.dot(transformation_matrix, vertex)

        # Convert back to 2D coordinates and add to dictionary
        self.scaled_vertices[key] = transformed_vertex[:2]
```

- Model to World
  - Definition: The rotation matrix is a 4x4 matrix that rotates a vector around an axis by an angle in radians. The matrix is calculated based on the axis vector and the sine and cosine of the angle. The resulting rotated vertex is a 4x1 homogeneous coordinate vector
  - Math formula

$$M = \begin{pmatrix} c + u_x^2(1-c) & u_x u_y(1-c) - u_z s & u_x u_z(1-c) + u_y s & 0 \\ u_y u_x(1-c) + u_z s & c + u_y^2(1-c) & u_y u_z(1-c) - u_x s & 0 \\ u_z u_x(1-c) - u_y s & u_z u_y(1-c) + u_x s & c + u_z^2(1-c) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, where\ c = cos(\theta)$$

  - code

```
def rotate_object(self, vector_x, vector_y, angle):
    # Convert angle from degrees to radians
    vector_z = 0
    radians = angle % (3.14159)

    # Compute sin and cosine of the angle
    c = np.cos(radians)
    s = np.sin(radians)

    # Create the rotation matrix for 3D rotation based on the given
axis vector
    norm = np.sqrt(vector_x**2 + vector_y**2 + vector_z**2)
    ux = vector_x / norm
    uy = vector_y / norm
    uz = vector_z / norm
    rotation_matrix = np.array([
        [c + ux**2 * (1 - c), ux * uy * (1 - c) - uz * s, ux * uz * (1
- c) + uy * s, 0],
        [ux * uy * (1 - c) + uz * s, c + uy**2 * (1 - c), uy * uz * (1
- c) - ux * s, 0],
        [ux * uz * (1 - c) - uy * s, uy * uz * (1 - c) + ux * s, c +
uz**2 * (1 - c), 0],
        [0, 0, 0, 1]
    ])
```

2. Painter's Algorithm

- Definition: Painter's Algorithm is a simple algorithm used to solve culling problem. It sort all polygons or objects in a scene based on their depth or distance, and then painting them onto the screen in order from back to front.
- Advantages:
  1. Simple to implement
  2. Handles complex scenes with arbitrary geometry
  3. Works well with transparency and overlapping objects
  4. Requires no more memory to store the depth buffer
- Disadvantages:
  1. Can be slow for large, complex scenes
  2. Cannot handle objects that intersect or overlap in complex way
  3. Can produce visual artifacts if the sorting algorithm is not accurate