DHAKA INTERNATIONAL UNIVERSITY

Department of Computer Science & Engineering

Lab Report

Course Name : Computer Graphics & Multimedia Lab

Course Code : CSE-406

Report No. : 02

Report Name : Rendering a 3D Table Using OpenGL with Camera, Model–View–Projection Transformations.

SUBMITTED BY	SUBMITTED TO
NAME : Md. Soumik Hasan **ROLL : 34** **BATCH : D-72**	**Umme Niraj Mahi** **Lecturer** **Department of CSE** **Dhaka International University**

DATE OF SUBMISSION : 02-12-2025

Objective

The objective of this experiment is to design and render a **3D table model** using OpenGL by implementing:

- Custom **cuboid-based geometry** for the table top and four legs
- **Model–View–Projection (MVP)** transformations
- A **camera system** with keyboard and mouse control
- **OpenGL buffers** (VAO, VBO, shaders) for rendering
- A complete 3D scene where the user can navigate around the table

Introduction

Modern OpenGL uses a programmable pipeline where all rendering is performed through shaders. In this experiment, I modeled a 3D table by composing multiple cuboids and rendered it inside an interactive 3D environment.

To navigate and view the table from different angles, I implemented a camera system using:

- Position, front, up vectors
- Yaw and pitch angles
- Keyboard movement (W, A, S, D, Space, Ctrl)
- Mouse-driven rotation

The project demonstrates how OpenGL handles 3D geometry, transformations, color attributes, and real-time camera control. It also helps understand how complex objects can be built from simple primitives such as cuboids.

Implementation

Tools & Technologies Used

- Python
- PyOpenGL
- GLFW (window + input handling)
- NumPy (matrix & vector operations)
- OpenGL 3.3 Core Profile

Geometry Construction (Table Modeling)

The table was created using **5 separate cuboids**:

1. **Table Top** – a wide and thin rectangular cuboid.

2. **4 Legs** – four long cuboids positioned at the corners.

Each cuboid is generated procedurally by computing the 8 vertices using:

```
(x ± width/2, y ± height/2, z ± depth/2)
```

Then expanded into 12 triangles (36 vertices). All cuboid vertices were concatenated into a single vertex array for rendering.

OpenGL Buffers and Shader Pipeline

Vertex Shader

- Receives each vertex position & color
- Applies the **Model, View, and Projection** matrices
- Outputs color to fragment shader

Fragment Shader

- Produces the final pixel color for each fragment

VAO & VBO Usage

- **VBO** stores vertex data (positions + colors)
- **VAO** stores attribute pointers
- Helps render the table efficiently in each frame

Data flow:
CPU → VBO → Vertex Shader → Fragment Shader → Screen

Model–View–Projection (MVP) Transformations

Model Matrix

Each cuboid (table part) uses a different model matrix for:

- Scaling (making legs thinner, top wider)
- Translation (placing legs at corners)

View Matrix

Generated using a **custom Camera class**, using:

- Position
- Front direction
- Up vector
- LookAt function

This allows the user to look around the scene.

Projection Matrix

A manually calculated Perspective Projection is used to simulate realistic depth.

Camera System (How It Works)

The camera supports:

Keyboard Controls

- WASD - Move forward/left/backward/right
- Mouse - Look around
- Space - Move up
- Shift - Move down
- ESC - Exit

Mouse Controls

- Changes yaw and pitch
- Updates camera direction vector
- Allows looking around the table naturally

The combination of vectors (`front`, `right`, `up`) ensures smooth movement in 3D space.

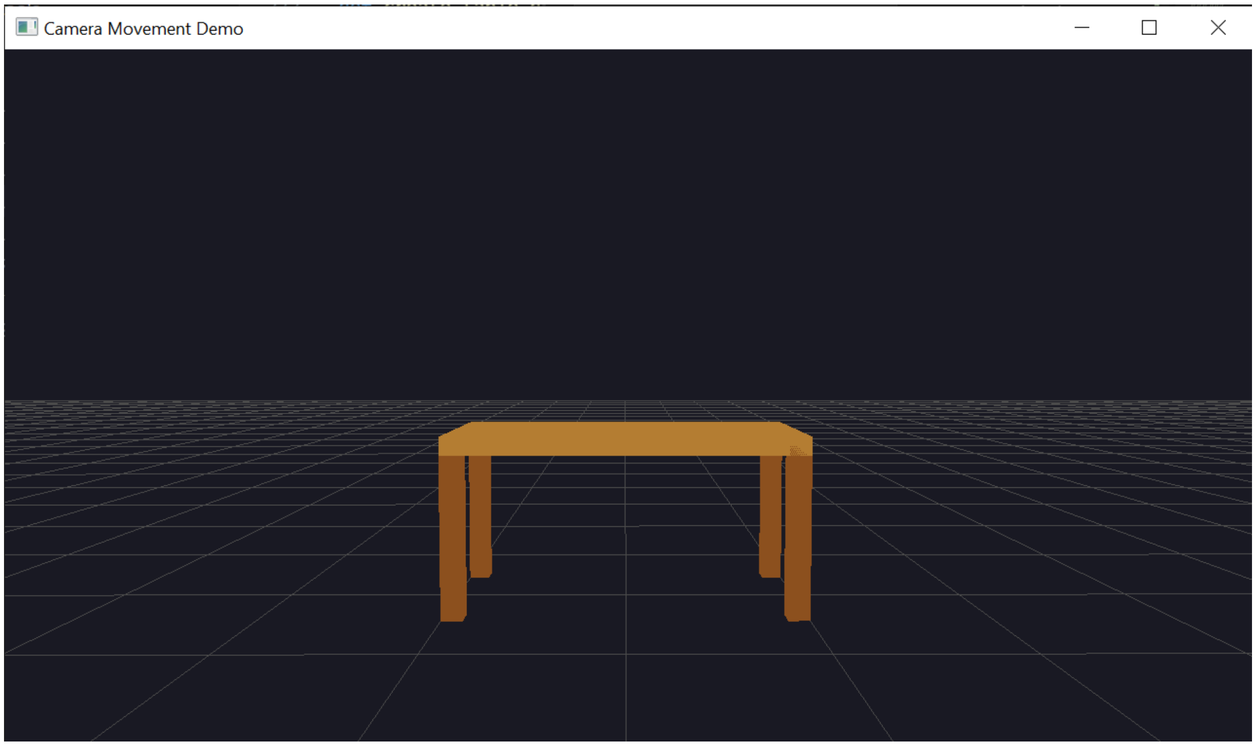
Render Loop

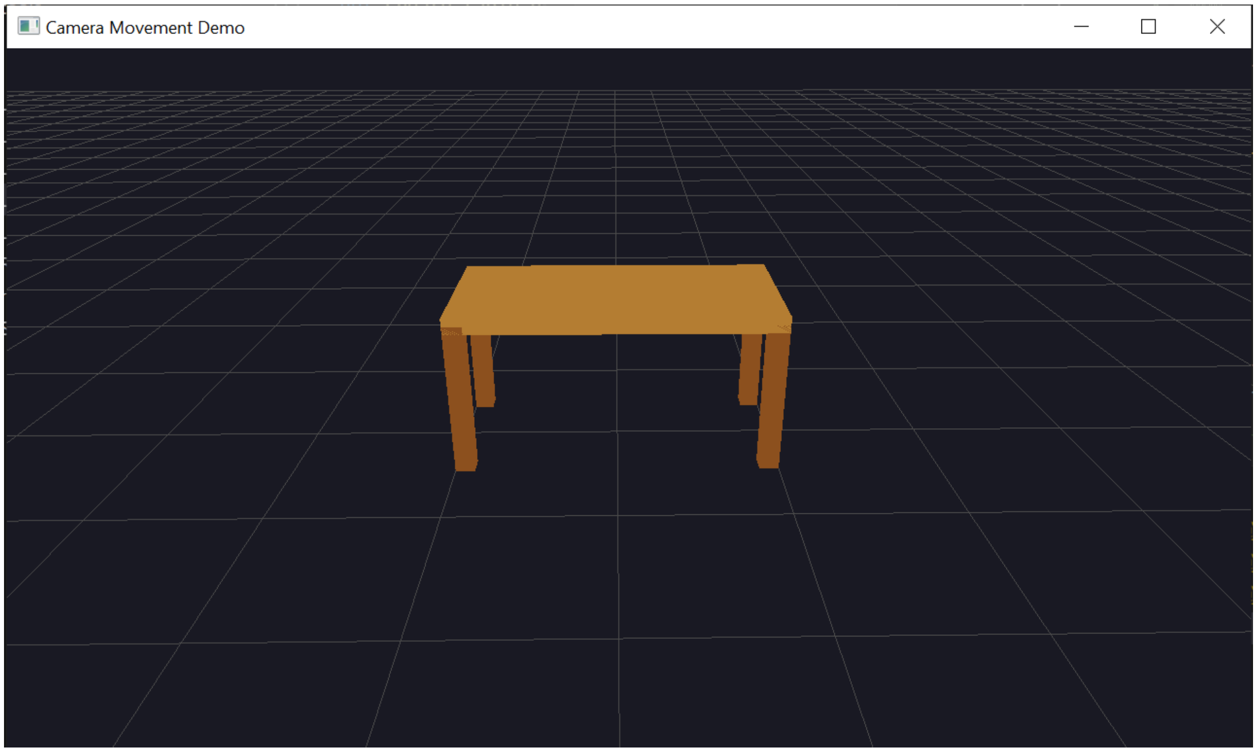
Each frame:

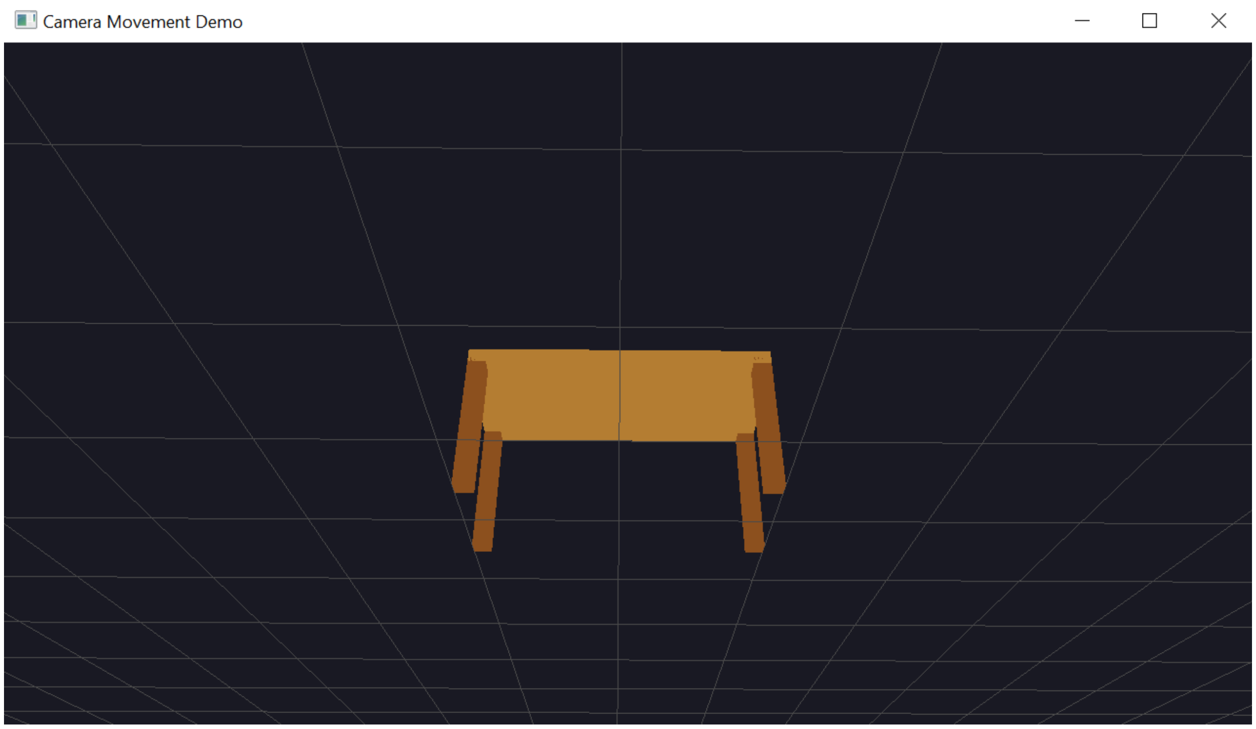
1. Clear screen
2. Update camera and projection matrices
3. Bind VAO
4. Draw all table vertices
5. Swap buffers (display result)

Real-time rendering allows free movement around the 3D table.

Output


Camera Movement Demo


Camera Movement Demo



Conclusion

This experiment significantly deepened my understanding of modern OpenGL and real-time 3D graphics. By constructing a complete 3D table model using multiple cuboids, I learned how vertex data, normals, and colors are combined inside VBOs, and how VAOs simplify the rendering process. Implementing camera movement helped me clearly grasp view and projection matrices, while working with shaders taught me how lighting and surface orientation affect final visual output. Overall, the project strengthened my conceptual and practical knowledge of the graphics pipeline and gave me confidence in transforming theoretical concepts into an interactive 3D application.