Lab 2 — Camera & Table (Computer Graphics Lab)

Author: [Your Name]
Date: 2025-12-01

Objective

The objective of this lab is to demonstrate a small interactive 3D scene using OpenGL: implement a camera that can be controlled with keyboard + mouse, construct and render a simple table model, and provide runtime diagnostics (FPS, camera state). The final deliverable shows a camera-controlled view of a table sitting on a grid floor and provides both in-terminal runtime status and well-structured project documentation.

Introduction

This exercise is designed to teach fundamental 3D computer graphics concepts — camera transforms, projection matrices, basic geometry construction, GLSL shaders, and rendering pipelines — using Python bindings (PyOpenGL/glfw). Students will gain practical experience composing model, view and projection matrices, creating buffer objects and writing very small vertex/fragment shaders.

Environment

- Python 3.8+ (3.11+ tested) — this lab was built and validated with Python 3.13 on Windows.
- Libraries required (see requirements.txt):
- PyOpenGL
- PyOpenGL_accelerate (optional, speeds up GL interop)
- glfw
- numpy
- Tools used to produce documentation:
- reportlab (for PDF generation)
- python-docx (for Word / .docx generation)

Setup on Windows (bash shell):

```
bash
python -m venv .venv
source .venv/Scripts/activate
pip install -r requirements.txt
pip install reportlab python-docx # optional for report generation
```

Implementation (what is used & how it works)

What is used:
- PyOpenGL: OpenGL bindings for Python to issue GL calls and work with shaders and buffers.
- glfw: cross-platform window and input library used to create the OpenGL context and capture keyboard/mouse events.
- numpy: for numerical arrays and matrix construction.

How it works (high-level):
1. Camera and transforms
- The Camera class holds position, yaw and pitch. It exposes a get_view_matrix() method that uses a look-at style calculation to produce the view matrix from the camera pose.
- A perspective projection matrix is generated by create_perspective_matrix(fov, aspect, near, far).
- The vertex shader multiplies model, view and projection matrices to move vertex positions to clip space.

2. Geometry & rendering
- Geometry is built on the CPU using numpy arrays: a floor grid (GL_LINES) and the table model assembled from axis-aligned cuboids for the tabletop and four legs.
- Interleaved vertex buffers hold position (vec3) followed by color (vec3). VBOs/VAOs are created and bound once at startup.
- GLSL shaders (vertex + fragment) perform the per-vertex transform and pass color to the fragment shader.

3. Application & input
- App initializes GLFW, creates the window and GL context, sets input callbacks (mouse + window resize), and runs the main loop.
- Keyboard/mouse input is processed to update the camera's position and orientation; delta-time is tracked for framerate-independent movement.

4. Runtime diagnostics
- Optionally, a terminal-facing real-time table prints FPS, delta-time, current camera position/rotation and a small object list. This is printed in-place using ANSI control sequences to avoid terminal spam.

Files changed / added

- Lab2.py — main program. Implements:
- Camera class (position, yaw/pitch, look-at view matrix)
- create_perspective_matrix() — projection matrix
- Basic GLSL shaders (vertex + fragment) for transform + color
- Table model generator: create_table_data() (table top + legs)
- App class with GLFW window, input handling and render loop
- Optional real-time terminal table printing technique (shows FPS, camera and per-object info)

Design / Implementation notes

- The program uses a classic OpenGL 3.3 core pipeline: vertex shader applies model, view, projection; fragment shader outputs interpolated color.
- Geometry uses interleaved vertex attributes: 3 floats for position followed by 3 floats for color.
- The floor grid is a simple line grid centered on the origin, floor plane at y = -1.0.
- The table model is constructed from several axis-aligned cuboids (top + 4 legs) generated at runtime.

How to run

1. Create and activate a virtual environment (Windows bash):

bash
python -m venv .venv
source .venv/Scripts/activate

2. Install dependencies:

bash
pip install -r requirements.txt

3. Run the lab program:

bash
python Lab2.py

Controls shown in the terminal while running:

- WASD — move camera (forward/back/left/right)
- Mouse — look around
- Space — move up
- Shift — move down
- ESC — exit

Live status table

If enabled, Lab2.py prints a small terminal table containing FPS, delta time, window size, camera position/orientation and (optionally) per-object rows. This is printed using ANSI sequences to a single-place, real-time table that refreshes in the terminal.

Notes / Future improvements

- On-screen HUD (text overlay) could be implemented for in-window stats
- Add textured materials, lighting and shadows for more realism
- Add an interactive UI to adjust table dimensions at runtime

Output

When the program runs, it opens a GLFW window and displays the following:

- A grid floor (lines) representing the scene ground (floor plane at y=-1.0).
- A wooden table (single model) placed on the floor, centered at the origin. The table is composed of a rectangular tabletop and four legs.
- Camera controls let you move and look around the scene using keyboard and mouse; movement is smooth and frame-rate independent.
- The terminal prints a compact, live-updating table with FPS, delta time, camera position/orientation, window size and (optionally) per-object rows.

Screenshot(s) could be added to this section for a visual demonstration (if available).

Conclusion

This lab demonstrates essential interactive 3D graphics-building blocks: camera math (view / projection), CPU-side geometry creation, minimal GLSL pipeline and GLFW-driven interactivity. It provides a concise but complete workflow for rapidly prototyping and documenting small OpenGL scenes in Python, and lays groundwork for enhancements such as textured materials, lighting models and UI-driven scene editing.

This file was used to render Lab2_report.pdf in the project workspace.