DIT UoA Graphics Project with OpenGL/C++ Fall 2025-2026

Ioannis Petrakis 1115201900155

January 2026

1 Introduction

This report outlines the development of the "Moving Objects" project for the Computer Graphics I course. The assignment required the creation of a 3D simulated environment where a central planet travels in a circular orbit while acting as a light source for six orbiting satellites (cubes).

The application was written in C++ using the Legacy OpenGL (GLUT) framework and was developed and tested in a Linux (Ubuntu 24.04) environment.

2 Implementation Analysis

2.1 3D Modeling and Geometry

The scene combines external assets with procedurally generated geometry. Two different methods were used to handle these objects.

2.1.1 Custom OBJ Parser

To load the `planet.obj` file without using external model loading libraries, a custom parser was implemented in C++. The parser reads the file stream line-by-line and processes standard Wavefront OBJ prefixes:

- **v**: Vertex positions.
- **vt**: Texture coordinates (UVs).
- **vn**: Vertex normals.
- **f**: Face definitions.

A critical detail in the implementation is the handling of indices. Wavefront OBJ files use 1-based indexing, whereas C++ vectors use 0-based indexing. The loader applies the following transformation when parsing faces:

$$\text{Index}_{internal} = \text{Index}_{OBJ} - 1$$

This ensures that the vertex data matches the memory layout of the `Model` structure.

2.1.2 Procedural Cube Generation

The six cubes are generated procedurally using OpenGL Immediate Mode (`glBegin(GL_QUADS)`). To ensure the `container.jpg` texture maps correctly to the 3D geometry, explicit UV coordinates are assigned to each vertex. For every face, the coordinates $(0, 0), (1, 0), (1, 1), (0, 1)$ are mapped to the four corners, preventing texture distortion.

2.2 Animation and Hierarchical Transformations

The animation system uses a hierarchical scene graph approach implemented via the OpenGL Matrix Stack.

2.2.1 Planet Motion

The planet does not remain stationary; it orbits the world origin $(0, 0, 0)$. Its position $P(t)$ at time t is calculated using parametric circle equations:

$$P(t) = \begin{bmatrix} R \cdot \cos(0.3t) \\ 0 \\ R \cdot \sin(0.3t) \end{bmatrix} \tag{1}$$

2.2.2 Compound Cube Motion

The cubes must orbit a moving target. To achieve this, the transformation matrix M_{cube} for each satellite is constructed by multiplying matrices in a specific order:

$$M_{cube} = T_{planet}(P_x, P_z) \times R_{orbit}(\alpha) \times T_{offset}(r) \times R_{local}(\beta) \tag{2}$$

- T_{planet}**:** Translates the coordinate system to the Planet's current dynamic position.
- R_{orbit}**:** Rotates the cube around the planet.
- T_{offset}**:** Pushes the cube out to its specified orbit radius.
- R_{local}**:** Rotates the cube around its own axis.

This hierarchy ensures the cubes maintain their orbit relative to the planet, regardless of the planet's position in the world.

2.3 Lighting Physics

The project requires the planet to act as the light source. This is implemented by updating the position of `GL_LIGHT0` in every frame to match the planet's coordinates:

```
GLfloat lightPos[] = { planetX, 0.0f, planetZ, 1.0f };
glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
```

This setup satisfies the requirement for "visual contact" lighting. According to the Blinn-Phong lighting model, the intensity I of a surface fragment is:

$$I = K_a + K_d \cdot \max(0, N \cdot L) \tag{3}$$

Here, N is the surface normal and L is the light vector. When a cube face points toward the planet, the dot product $N \cdot L$ is positive, illuminating the face. When it points away, the result is zero or negative, leaving the face in shadow.

2.4 Camera Control

The camera uses a spherical coordinate system controlled by user input (Arrow Keys). The spherical angles θ (azimuth) and ϕ (elevation) are converted to Cartesian coordinates for the view matrix:

$$x = \text{zoom} \cdot \sin(\theta) \cos(\phi)$$

$$y = \text{zoom} \cdot \sin(\phi)$$

$$z = \text{zoom} \cdot \cos(\theta) \cos(\phi)$$

3 Assumptions and Dependencies

- **Texture Loading:** As Legacy OpenGL does not support native image format loading, the `stb_image.h` single-header library was used. This provides a lightweight, portable solution for decoding PNG and JPG files on Linux without requiring heavy external dependencies like OpenCV.
- **Mesh Format:** The custom OBJ parser assumes the input file uses triangulated faces.

4 Linux Deliverables

The project has been configured for compilation on Ubuntu. The submission folder contains the following files:

- **main.cpp**: The source code.
- **planet_project**: The compiled executable file/binary.
- **stb_image.h**: The texture loading library.
- **planet.obj**: The 3D model.
- **planet_Quom1200.png**: The planet texture.
- **container.jpg**: The cube texture.

4.1 Compilation Instructions

To compile the project on a system with `freeglut3-dev` installed, navigate to the folder and run:

```
g++ main.cpp -o planet_project -lGL -lGLU -lglut -lm
```

To execute the program:

```
./planet_project
```