City Simulation Report

Ankit Kumar, Dhruv Duggal, Gagandeep Singh, and **Naman Salwan**

University of Lethbridge

Computer Graphics (CPSC-3710)

Mr. Howard Cheng

December 01, 2024

1. Program Organization

Object Organization

The program follows an object-oriented design, breaking the application into modular, manageable components, each with clearly defined roles:

a. **Car Class:** Represents the car object, which includes its physical structure (body, tires, headlights, tail lights, and windows) and behaviors (movement, rotation, collision detection). The car's movements are tightly controlled, and its visual appearance dynamically adapts based on interaction. The class also implements features like tire rotation and lighting changes during reversing.

b. **Cuboid Class:** Represents all buildings in the city. This class defines the cuboid’s geometry (vertices and faces), colors, and additional features such as windows. It manages the rendering process for both the building and its windows, ensuring proper alignment and scaling.

c. **TrafficLight Class:** Handles traffic lights at intersections. This includes cycling between green, yellow, and red states at defined intervals. Each traffic light maintains its position and state independently, and all instances are stored in a vector for easy management.

d. **BoundingBox Class:** Implements collision detection by defining the minimum and maximum bounds of objects like buildings and the car.

2. Interaction Organization

a) Keyboard Controls:

- ***w / UP Arrow* ⬆: *Move the car forward.***
- ***s / DOWN Arrow* ⬇: *Move the car backward.***
- ***a / LEFT Arrow* ⬅: *Turn the car left (if moving).***
- ***d / RIGHT Arrow* ➡: *Turn the car right (if moving).***

b) Camera Views:

- **F1:** Default view from behind the car.
- **F2:** Overhead view.
- **F3:** Top-corner view.
- **F4:** Driver view from inside the car.

c) Special Keys:

- **ESC:** Exit the program.

3. Collision Handling:

The car detects collisions using bounding boxes around buildings. If the car collides with a building, movement is halted to simulate a real-world obstacle.

A limitation in collision detection (discussed in Section 3) allows some buildings to be passable, which reduces realism.

4. View Organization

Camera Views:

Multiple camera perspectives are implemented using OpenGL’s ***LookAt transformation***, allowing seamless transitions between views:

1. **Default View (F1):** Positioned behind and above the car, following its movement and angle dynamically.
2. **Overhead View (F2):** Directly above the car, providing a top-down perspective of the city layout.
3. **Top-Corner View (F3):** Positioned at an angle, giving a balanced perspective of the car and its surroundings.
4. **Driver View (F4):** Simulates the perspective from within the car’s cabin, looking straight ahead.

The camera adjusts its position relative to the car in real time, ensuring that the chosen perspective always aligns with the car's position and orientation.

5. Key Learnings from the Project

Transformations:

Learned to use transformations (Translate, Rotate, Scale) to define and animate 3D objects in space.

Understood how to apply hierarchical transformations, where object components (e.g., tires, headlights) are transformed relative to the main object (e.g., car body).

Projection and Viewing:

Gained a deeper understanding of perspective and orthogonal projections, their matrix representations, and their impact on rendering.

Implemented the LookAt function to create dynamic and interactive camera views.

Buffer Management:

Used Vertex Array Objects (VAOs) and Vertex Buffer Objects (VBOs) to optimize the rendering pipeline.

Applied shaders for custom rendering, enabling flexibility in object appearance.

Window Rendering:

Designed and rendered dynamic windows on both the car and buildings. Learned to calculate and position window vertices programmatically based on object dimensions.

Object-Oriented Programming:

Structured the program into reusable and modular classes, enabling scalability and easier debugging.

Implemented clear abstractions for different object types, reducing code duplication.

Dynamic Data Management:

Used vectors and pointers to manage dynamically created objects (e.g., buildings, traffic lights) efficiently.

Debugging and Optimization:

Tackled challenges like misaligned windows, flying objects, and collision bugs, reinforcing problem-solving skills.

6. Use of Valgrind for Debugging

In this project, Valgrind was utilized to identify and debug memory-related issues. Valgrind is a powerful tool that helps detect memory leaks, invalid memory accesses, and improper use of dynamic memory in C++ programs. Its use ensured the stability and reliability of the project, especially in handling dynamic memory management for complex objects like the car, buildings, traffic lights, and their associated data structures.

7. Limitations and Bugs

Collision Detection:

While the car successfully detects collisions with most buildings, certain buildings allow the car to pass through. This issue arises due to inconsistencies in the bounding box calculations, likely caused by random building dimensions and positions.

Possible Solutions:

1. Adjust bounding box dimensions dynamically based on building height and width.
2. Refine collision detection logic to handle edge cases.

Window Placement:

Windows on buildings are generated programmatically based on building dimensions. However, highly irregular building shapes or extreme dimensions may result in misaligned or disproportionately large windows.

Possible Solutions:

1. Introduce a constraint system to standardize building shapes.
2. Dynamically adjust window sizes and spacing for extreme cases.

City Layout:

The city layout is static and predefined, lacking dynamic or procedural generation. This limits replayability and realism.

Possible Enhancements

1. Introduce interactive traffic lights that respond to the car's position and speed.
2. Implement advanced lighting and shadows for improved visual appeal.

8. Instructions to Run the Project

This section provides detailed steps to set up and run the project using either a provided zip file or directly from the GitHub repository.

Option 1: Using the Provided Zip File

1. Download the Zip File:
2. Locate the zip file provided with this report.

3. Download it to your local machine.
4. Extract the Files: Extract the zip file to a directory of your choice using any archive tool (e.g., WinRAR, 7-Zip, or the default extractor in your OS).
5. Navigate to the Project Directory:
6. **Build the Project**:
7. Ensure you have the required OpenGL development libraries installed (see **Prerequisites**).
8. Compile the project using the provided Makefile:
    a. **make main**
    b. Then run **./main**

Option 2: Cloning the GitHub Repository

1. Open a terminal or command prompt.
2. Clone the repository using the following command:
   -> git clone https://github.com/yourusername/yourprojectname.git
3. Navigate to the Project Directory:
4. Enter the cloned repository:
5. Build the Project:
6. Ensure you have the required OpenGL development libraries installed (see Prerequisites).
7. Compile the project using the Makefile:
    a. **make main**
    b. Then run **./main**

Prerequisites

Before running the project, ensure that the following dependencies are installed on your system:

1. OpenGL Libraries:
2. Install OpenGL development packages. For example:

- **On Ubuntu:**

   a. sudo apt-get install freeglut3 freeglut3-dev
   b. sudo apt-get install libglew-dev

- **GLEW and GLUT:** Ensure GLEW and GLUT libraries are properly installed as they are required for the project to compile and run.
- **C++ Compiler:** Ensure you have a modern C++ compiler, such as g++.

This project has been an excellent opportunity to explore the principles of computer graphics, OpenGL, and programming. By working on a city simulation, I gained practical experience in rendering 3D objects, managing interactions, and implementing complex behaviors. The multi-camera view system provided valuable insights into how transformations can be used creatively to enhance user experience. While the program has limitations, such as collision inaccuracies and static city layouts, these challenges present opportunities for further improvement and learning.