Smart Eco City Simulation: Project Report

Course: Object-Oriented Programming (CS112)

Team Members:

Saad H. Ellahie 2024545
Abdullah Bin Adnan, 2024035
Hamza Sami, 2024212
Asad Akmal, 2024355
Zawar Fahim, 2024494

May 9, 2025

Introduction

This report details the "Smart Eco City Simulation" project, developed as a semester project for the Object-Oriented Programming (CS112) course. The simulation allows users to design, build, and manage a virtual city with a focus on ecological balance and sustainability. Players make decisions regarding infrastructure development, resource management, and environmental policies, aiming to create a thriving and eco-friendly metropolis. This document outlines the game's flow, its key features, and the application of various *OOP* concepts that form the foundation of the simulation.
https://github.com/Sel68/Smart-EcoCity-Simulation

Contents

1 Game Flow **3**
- 1.1 Create a New City . . . 3
- 1.2 Manage the City (Main Gameplay Loop) . . . 3
- 1.3 Achieve Goals and Get Rewards . . . 4
- 1.4 Compete with Others . . . 4
- 1.5 View Global Rankings . . . 4
- 1.6 Game Progression and End . . . 4

2 Features **5**

3 Object-Oriented Programming Concepts Used **7**
- 3.1 Classes and Objects . . . 7
- 3.2 Encapsulation . . . 7
- 3.3 Inheritance . . . 7
- 3.4 Polymorphism . . . 8
- 3.5 Abstraction . . . 8
- 3.6 Templates . . . 8
- 3.7 Operator Overloading . . . 9
- 3.8 Dynamic Memory Allocation . . . 9
- 3.9 Preprocessing Directives . . . 9
- 3.10 Constructors and Destructors . . . 10
- 3.11 Exception Handling . . . 10
- 3.12 File I/O . . . 10
- 3.13 Use of Standard Template Library (STL) . . . 11

4 Screenshots of Game Interface **12**

5 Conclusion **17**

1 Game Flow

The Smart Eco City Simulation offers an interactive experience where players take on the role of a city planner and manager. The typical flow of the game is as follows:

1.1 Create a New City

- Players start by creating a new city, providing a name for their city and a name for their player avatar.
- Upon creation, the player receives an initial amount of gold and some starting experience points.
- An initial eco-tip is displayed to guide the player.

1.2 Manage the City (Main Gameplay Loop)

- Once a city is created or selected, players access a City Menu with various options.
- **Earn Money (Work)**: Players can perform a "Go To Work" action to earn gold. This action has a cooldown period and also slightly increases the city's pollution. The base earnings are influenced by the player's Green Level and Experience.
- **Build Infrastructure**:
  - *Structures*: Players can spend gold to construct various buildings like Houses, Skyline Residences, Malls, and different types of Power Plants (Coal, Solar, Wind, Nuclear). Each building has a cost, land cover, and specific effects on the city (e.g., population capacity for residential buildings, energy generation and pollution for power plants).
  - *Transport*: Players can invest in transport infrastructure such as Road Networks, Railway Networks, and Airports. These have build costs, maintenance costs, and impact the city's transport network score.
  - *Vehicles*: Players can purchase Electric Vehicles (EVs) or Internal Combustion Engine (ICE) vehicles. EVs positively impact the EcoScore, while ICE vehicles negatively affect it.
- **Manage Resources and Environment**:
  - *Plant Trees*: Players can spend gold to plant trees, which improves the city's green space factor and EcoScore.
  - *Perform Maintenance*: Transport systems require maintenance, which costs gold but improves their efficiency and contribution to the transport score.
  - *Monitor City Stats*: Players need to keep an eye on the city's population, pollution level, EcoScore, power demand vs. capacity, and renewable energy percentage. These stats are influenced by the player's actions.
- **Engage in Activities**:

- *Go To Mall*: A recreational activity that costs a small amount of gold and provides a minor experience gain.
- *Drive Car*: Players can choose to drive one of their owned vehicles, which provides experience and affects pollution differently based on vehicle type (EV or ICE).
- *Eco Tips*: Players can request random eco-tips to learn about sustainable practices. These tips are read from an external file (`educationalTips.txt`).

1.3 Achieve Goals and Get Rewards

- The simulation has predefined goals, such as reaching a certain renewable energy percentage, maintaining low pollution, achieving a high EcoScore, or purchasing the first EV.
- Achieving these goals rewards the player with a Title, bonus gold and experience points.
- The player also has a "Green Level" that can increase (or decrease) based on the city's EcoScore, providing additional rewards and benefits (e.g., higher income from work).

1.4 Compete with Others

- If multiple cities are created in the simulation, players can choose to "Compete with Another City."
- This feature compares the EcoScores of the two cities, declaring a winner based on who has the higher score.

1.5 View Global Rankings

- Players can view a global ranking of all cities in the simulation based on their EcoScores or other criteria (extensible via templates).

1.6 Game Progression and End

- The game continues as the player develops their city, manages its economy and environment, and strives for higher EcoScores and Green Levels.
- The simulation can be exited from the Master Menu, and progress (like actions taken) is logged to a file specific to the city and player.

2 Features

The Smart Eco City Simulation incorporates a variety of features to create an engaging and educational experience:

- **City Creation and Management**: Players can create and manage multiple aspects of their city, including its name, finances, and development.
- **Economic System**: Players earn gold through work and spend it on buildings, transport, vehicles, and maintenance.
- **Building System**:
  - Diverse building types: Residential (`House`, `SkylineResidence`), Commercial (`Mall`), and Industrial (`PowerPlant`).
  - Power plants include conventional (Coal) and renewable (Solar, Wind) options, plus Nuclear, each with different costs, capacities, and pollution impacts.
- **Transport System**:
  - Multiple transport types: Roads, Railways, Airports.
  - Transport systems require maintenance and contribute to a transport network score.
- **Vehicle System**: Players can buy and use Electric Vehicles (EVs) or Internal Combustion Engine (ICE) vehicles, each affecting the EcoScore differently.
- **Environmental Simulation**:
  - *Pollution Tracking*: Actions like working and using certain power plants generate pollution.
  - *EcoScore*: A key metric reflecting the city's environmental health, influenced by pollution, renewable energy use, green spaces, vehicle types, and transport network quality.
  - *Green Space Factor*: Improved by planting trees, contributing positively to the EcoScore.
  - *Renewable Energy Focus*: The simulation encourages the use of renewable energy sources by rewarding players and improving the EcoScore.
- **Player Progression**:
  - *Experience Points (XP)*: Gained from various activities like building, working, and completing eco-friendly actions.
  - *Green Level*: Increases with a higher EcoScore, granting benefits and rewards.
- **Goals and Rewards System**: Players are rewarded for achieving specific environmental and developmental milestones.
- **Educational Eco Tips**: Provides players with real-world facts and tips on sustainability, read from an external file (`educationalTips.txt`).

- **Work Cooldown Mechanic**: The "Go To Work" action has a time-based cooldown to regulate income generation.
- **Dynamic City State Updates**: The city's overall state (population, power demand/capacity, pollution, EcoScore) is recalculated after significant player actions.
- **Competition Mode**: Allows comparison of EcoScores between different player-created cities.
- **Global Rankings**: Displays a leaderboard of cities based on their performance.
- **User Interface**:
  - Menu-driven navigation for master simulation actions and city-specific actions.
  - Screen clearing for a cleaner interface.
  - Formatted currency display.
- **Logging**: Player actions and significant events are logged to a text file for each city.
- **Error Handling**: Exception handling is implemented for operations like spending gold or file operations.

3 Object-Oriented Programming Concepts Used

The "Smart Eco City Simulation" extensively uses OOP principles to model the complex interactions within a city environment.

3.1 Classes and Objects

- `Player`: Represents the user controlling the city, managing gold, experience, and vehicles.
- `City`: The main orchestrator class, containing a `Player`, an `Environment`, and collections of `Building` and `Transport` objects.
- `Environment`: Models the city's ecological state, including pollution, EcoScore, population, and resource demands.
- `Building` (Base Class): Abstract representation of a structure. Derived classes include `Residential` (further derived into `House`, `SkylineResidence`), `Mall`, and `PowerPlant`.
- `Transport` (Base Class): Represents transportation infrastructure. Derived classes include `Road`, `Railways`, and `Airport`.
- `Vehicle` (Base Class): Represents vehicles. Derived classes are `EV` (Electric Vehicle) and `ICE` (Internal Combustion Engine Vehicle).
- `Utilities` (Base Class): Represents city utilities. Derived classes are `Water`, `Electricity`, and `Gas`. These are used within `Residential` buildings.
- `CityLog`: A template class for logging game events.

Objects of these classes (e.g., a specific `House` object, a particular `Player` object) are created and interact throughout the simulation.

3.2 Encapsulation

- Player's gold (`gold`) is `private` or `protected` in the `Player` class and can only be modified through public methods like `spendGold()` and `earnGold()`. This prevents direct, uncontrolled modification of sensitive data.
- The `Environment` class encapsulates `pollutionLevel` and `ecoScore`, providing methods like `modifyPollution()` and `recalculateEcoScore()` to manage these attributes in a controlled manner.
- Most class members are `protected` or `private`, with public methods (getters, setters, and action methods) providing controlled interfaces.

3.3 Inheritance

- `Residential`, `Mall`, and `PowerPlant` classes inherit from the `Building` base class. They share common properties like `cost` and `landCover` and methods like `displayDetails()` (which they often override), while also having their specific attributes and behaviors.

- `House` and `SkylineResidence` inherit from `Residential`, further specializing residential buildings.
- `Road`, `Railways`, and `Airport` inherit from `Transport`.
- `EV` and `ICE` inherit from `Vehicle`.
- `Water`, `Electricity`, and `Gas` inherit from `Utilities`.
- This hierarchy allows for treating varied objects (e.g., different building types) uniformly through their base class pointers, facilitating polymorphism.

3.4 Polymorphism

- *Virtual Functions*:
  - The `Building` class has a `virtual void operate()` method and a `virtual void displayDetails() const` method. Derived classes like `House`, `Mall`, and `PowerPlant` provide their own specific implementations (override) for these methods. This allows a collection of `Building*` pointers to call the correct `operate()` or `displayDetails()` method for the actual object type at runtime.
  - The `Transport` class has `virtual void useTransport()` and `virtual void displayDetails() const`.
  - The `Vehicle` class has `virtual void move()` and `virtual void displayDetails() const`.
  - The `Utilities` class has `virtual std::string getType() const`.
- When iterating through a `std::vector<Building*> buildings`, calling `building->displayDe` invokes the specific version for `House`, `Mall`, etc., without needing to know the exact derived type at compile time.

3.5 Abstraction

- The `Player` interacts with a `Building` object through high-level actions like "buy" or implicitly through city updates, without needing to know the intricate details of how each building type affects the environment or population internally.
- The `City::updateCityState()` method encapsulates the complex logic of recalculating various city metrics (population, power, pollution, EcoScore) based on the current buildings, transport, and player vehicles. The user of the `City` class simply calls this method.
- Base classes like `Building`, `Transport`, and `Vehicle` define a common interface (e.g., `getCost()`, `displayDetails()`) that abstract away the specific differences of their derived classes.

3.6 Templates

- `template <typename itemType> void City::BuyItemGeneric(itemType* newItem)`: This function template is used to handle the purchase logic for differ-

ent types of items like `Building`, `Transport`, and `Vehicle`. It uses `if constexpr` and `std::is_same_v` to tailor behavior based on the actual type of `itemType` at compile time, allowing for a single function to buy various game entities, reducing code duplication.

- `template <typename T> class CityLog`: This class template is used for logging. While currently used with `std::string` (`CityLog<std::string> actionLog`), it's designed to potentially log different types of entries if needed in the future.
- `template<typename T = City> void displayRankings(const std::vector<T*>& items, double(*getScore)(const T*) = ...)`: This function template provides a generic way to display rankings for different types of items (defaulting to `City` objects) based on a scoring function provided as a parameter (defaulting to `getEcoScore`).

3.7 Operator Overloading

- `friend std::ostream& operator«(std::ostream& os, const Player& p)`: Overloads the « operator to allow direct printing of `Player` objects to an output stream, displaying their status in a formatted way.
- `friend std::ostream& operator«(std::ostream& os, const Environment& env)`: Similarly, overloads « for the `Environment` class to display environmental statistics.
- `friend std::ostream& operator«(std::ostream& os, const City& city)`: Overloads « for the `City` class to print a comprehensive city report.
- `bool Player::operator<(const Player& other) const`: Overloads the < operator for the `Player` class to compare players based on their experience. This is useful if sorting players is needed (though not explicitly used for sorting in the provided main simulation loop, it's available).

3.8 Dynamic Memory Allocation

- Buildings, transport systems, and vehicles are created dynamically using `new` when the player chooses to build or buy them (e.g., `building = new House();`, `vehicle = new EV();`).
- These dynamically allocated objects are stored in `std::vector` containers of pointers (e.g., `std::vector<Building*> buildings;`).
- Destructors (`~Player()`, `~City()`, `~Building()`, etc.) are responsible for deallocating this memory using `delete` to prevent memory leaks (e.g., in `City::~City()`, it iterates through `buildings` and `transportSystems` vectors, deleting each element).

3.9 Preprocessing Directives

- **Concept**: Instructions for the compiler that are processed before the actual compilation of the source code.

- **Use Case**:
  - `#include <iostream>`, `#include <vector>`, `#include <string>`, etc.: Used to include standard library headers, providing access to functionalities like I/O operations, vectors, string manipulation, time functions, file streams, and algorithms.
  - `#ifdef _WIN32 ... #else ... #endif`: Used in the `clearScreen()` function to provide platform-specific commands for clearing the console screen (`system("cls")` for Windows, `system("clear")` for macOS/Linux).
  - `const chrono::seconds WORK_COOLDOWN(30);`: Defines global constants for game parameters.

3.10 Constructors and Destructors

- All classes have constructors to initialize their member variables to default or specified values (e.g., `Player(std::string name = "DefaultPlayer", double startGold = 1000000, ...)`).
- Parameterized constructors allow creating objects with specific initial states (e.g., `PowerPlant(std::string eType = "Coal", ...)`).
- Destructors (e.g., `~Player()`, `~City()`, `~Building()`) are implemented to delete dynamically allocated memory owned by the objects, such as vehicles in the `Player` class or buildings and transport systems in the `City` class, preventing memory leaks.

3.11 Exception Handling

- The `Player::spendGold()` method uses a `try-catch` block to `throw` and `catch` `std::runtime_error` if the amount is negative or if the player doesn't have enough gold.
- The `Utilities::setUsage()` method uses `try-catch` to handle invalid usage values.
- The `CityLog` constructor uses `try-catch` to handle potential errors during file opening.
- The generic `BuyItemGeneric` function in the `City` class has `try-catch` blocks to handle potential errors during the purchase process.
- The main simulation loop (`simulation()` function) uses `try-catch` to handle invalid input for menu choices.

3.12 File I/O

- The `CityLog` class uses `std::ofstream` to write log entries (player actions, city events) to a text file (e.g., `"cityName_playerName_log.txt"`).
- The `displayRandomTip()` function uses `std::ifstream` to read educational tips from an external text file named `educationalTips.txt`.

3.13 Use of Standard Template Library (STL)

- `std::vector`: Used extensively to manage dynamic collections of objects, such as `std::vector<Building*> buildings` in the `City` class, `std::vector<Vehicle*> vehicles` in the `Player` class, and `std::vector<City*> cities` in the main simulation.
- `std::string`: Used for handling text data like player names, city names, building types, and log entries.
- `std::chrono`: Used for time-related functionalities, specifically for the work cooldown mechanic (`std::chrono::seconds`, `std::chrono::steady_clock`).
- `std::iomanip`: Functions like `std::setprecision()`, `std::fixed`, `std::setw()` are used for formatting output, especially currency and report tables.
- `std::fstream`, `std::ifstream`, `std::ofstream`: For file input/output operations.
- `std::stringstream`: Used in `displayCurrency()` to format double values as currency strings.
- `std::numeric_limits`: Used with `std::cin.ignore()` for robust input handling.
- `std::is_same_v`: Used in template metaprogramming within `BuyItemGeneric`.
- `std::pair`: Used in `displayRankings` to store score and city name pairs for sorting.

4 Screenshots of Game Interface

Main Menu:

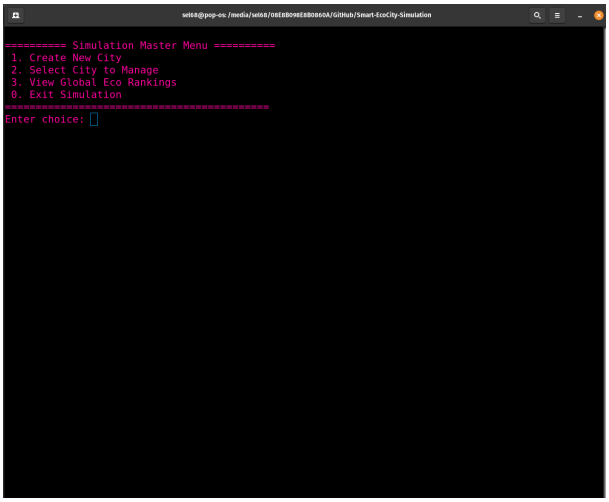

Figure 1: Main Menu

City Creation Prompt:

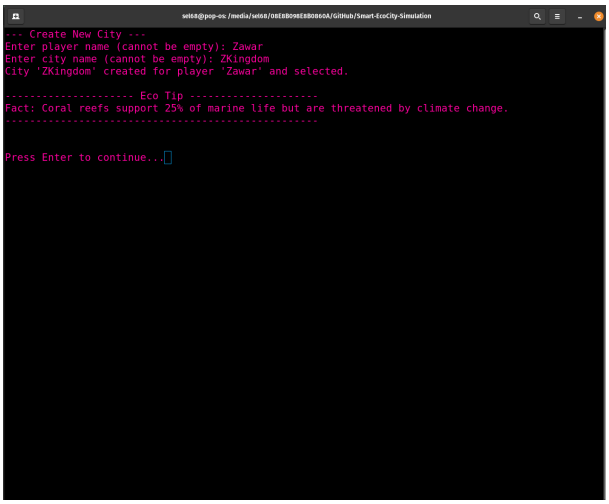

Figure 2: Creation of City

City Management Menu:

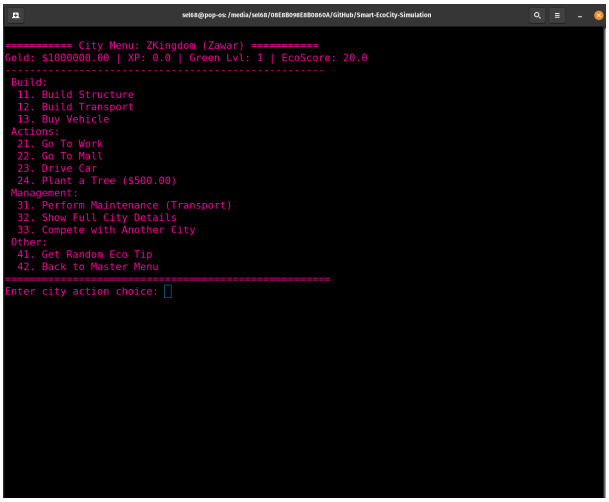

Figure 3: City Menu

Building Menu:

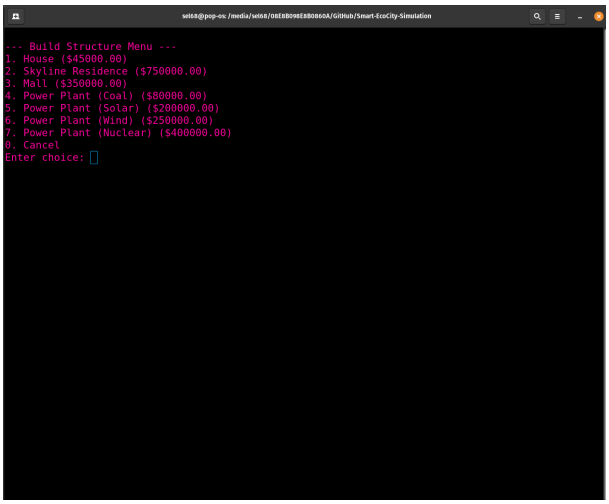

Figure 4: Buildings Menu

Transport Menu:

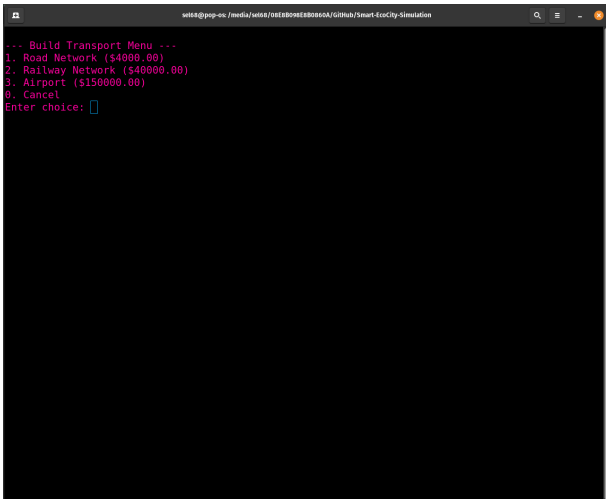

Figure 5: Build Transport Menu

Vehicle Purchase Menu:

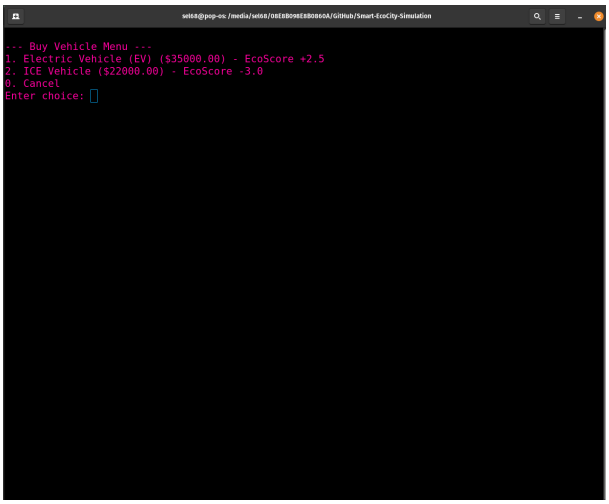

Figure 6: Vehicle Purchase Menu

City Report:

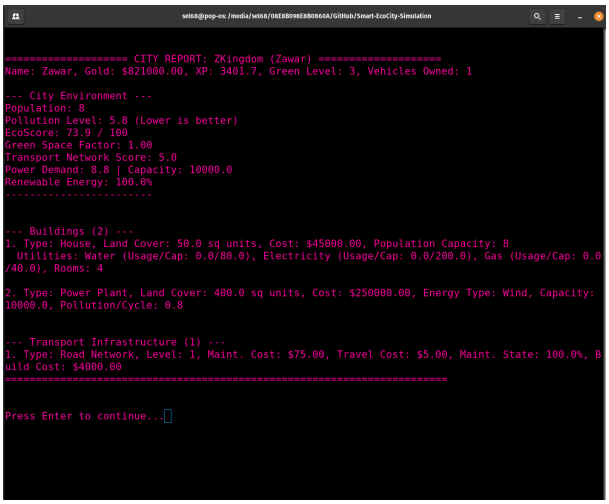

Figure 7: City Report

Work Action:

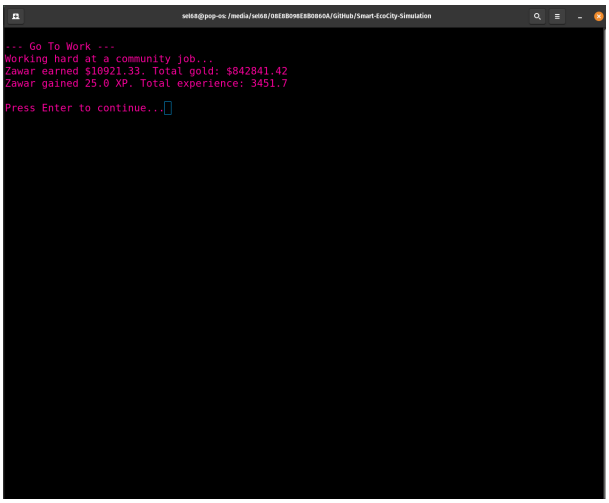

Figure 8: Work Success

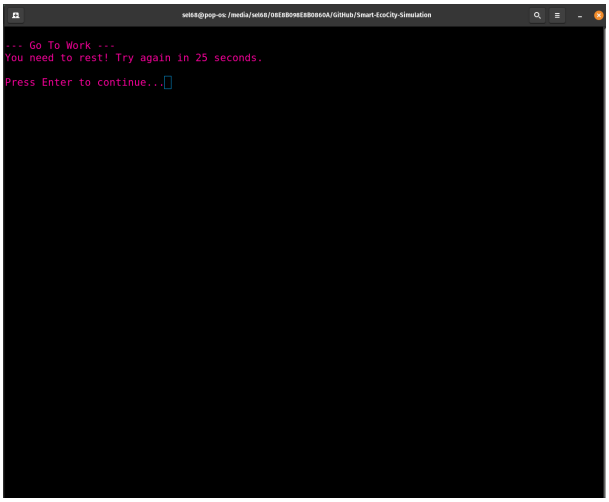

Figure 9: Work Cooldown Period

Planting a Tree:

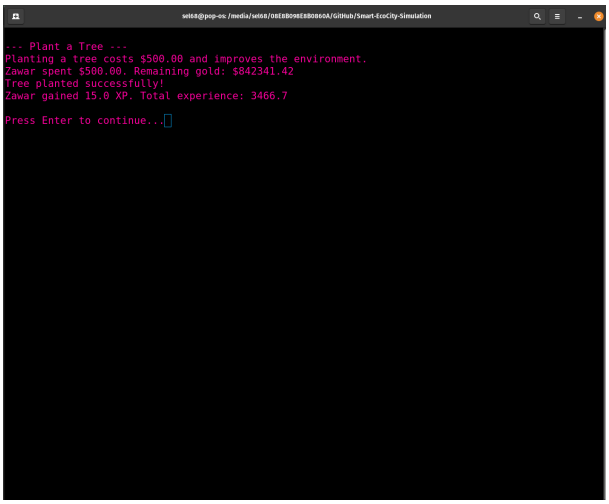

Figure 10: Planting a Tree

Eco Tip Display:

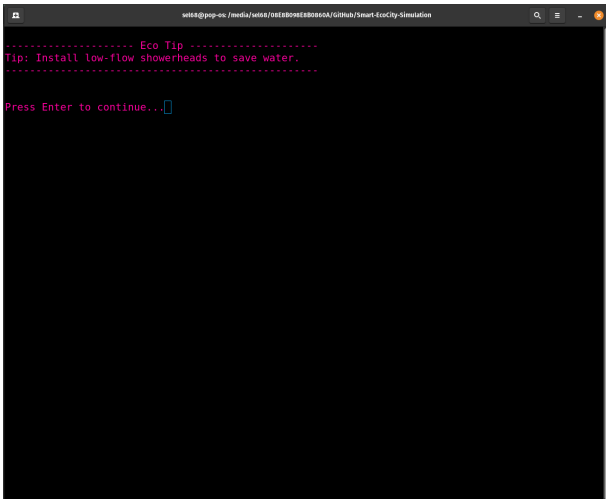

Figure 11: Display Eco Tip

Competition Screen:

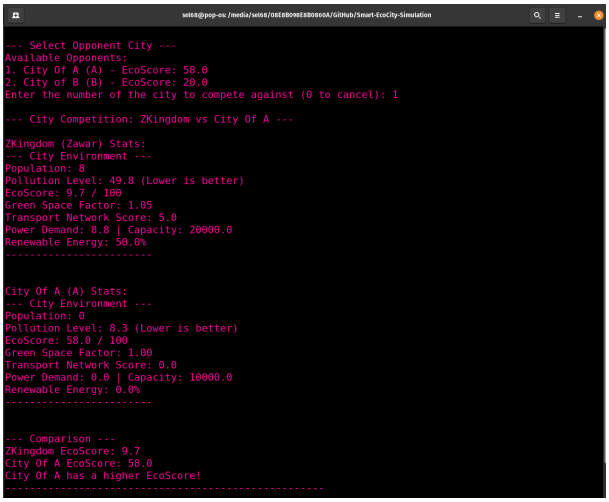

Figure 12: Compete with other cities

Global Eco Rankings Table:

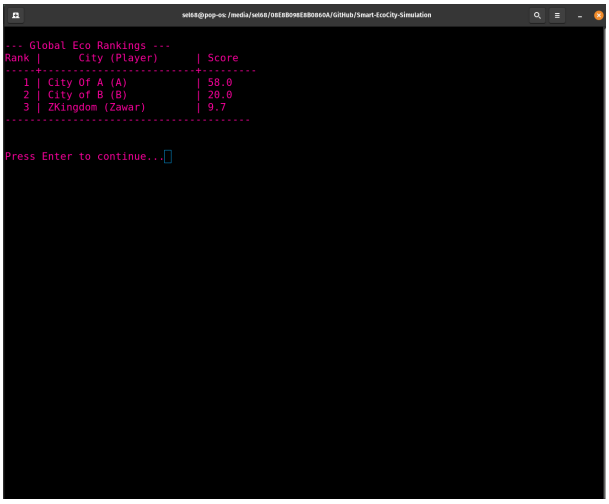

Figure 13: Global Eco Ranking

Log File Content:

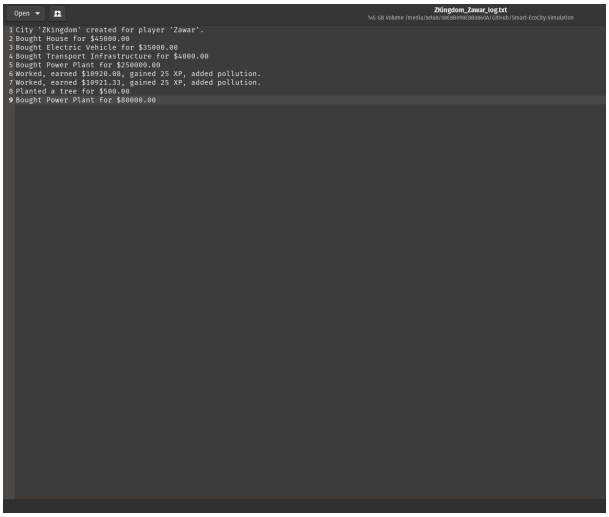

Figure 14: Log File

5 Conclusion

The "Smart Eco City Simulation" project successfully demonstrates the application of core Object-Oriented Programming principles to create a functional and interactive simulation. Through the use of classes, inheritance, polymorphism, encapsulation, and other OOP features, the project models a complex system with manageable and extensible code. The game provides an engaging way for users to learn about urban planning and environmental sustainability while navigating the challenges of managing a city's growth and ecological impact.