Dalhousie University

Department of Electrical and Computer Engineering

Task 5 (AEDV City Map)

Course: System Analysis

Title: Testing Documentation

Instructor: Dr. Larry Hughes

Date: Fall 2023

Designed and Implemented by

Anthony Okoro | Semilore Kayode
B00802417 | B00863886

1 Table of Contents

2 Design

2.1 Introduction

In an era of rapidly advancing technology, the integration of automation and sustainable transportation systems has become imperative to address the growing demands of urban logistics. The development of Autonomous Electric Delivery Vehicles (AEDVs) is a promising solution to streamline the delivery process, reduce carbon emissions, and enhance the efficiency of last-mile logistics. This software design report delves into the creation of a sophisticated simulation for an AEDV, emphasizing the design and implementation of a city map and a navigation system.

As our world continues to urbanize and online shopping becomes a fundamental part of our daily lives, the efficient transportation of goods to consumers has never been more critical. The rise of AEDVs offers a glimpse into a future where intelligent vehicles navigate bustling city streets, delivering packages with precision and care. Our simulation project reflects the essential transition to autonomous and sustainable transportation, where environmental consciousness meets cutting-edge technology.

This report provides an in-depth exploration of the software design process behind the AEDV simulation, from the initial design considerations and the creation of a city map to the implementation of navigation algorithms. The goal of this simulation is to offer an accurate representation of real-world scenarios, allowing for the testing and refinement of AEDV algorithms and systems within a controlled environment. By simulating the AEDV's interactions with the city environment, we aim to provide a platform for researchers and developers to assess, optimize, and advance the capabilities of autonomous electric delivery vehicles.

2.2 Problem Statement

The Autonomous Electric Vehicle Corporation has expressed the need for a practical demonstration of their system to visually showcase the functionality and capabilities of Autonomous Electric Delivery Vehicles (AEDVs) designed to operate within a city environment. The project's primary objective is to create a dynamic simulation of these AEDVs in action, complete with a city map that users can configure and observe AEDVs navigating through the city's streets and avenues.

The city map design includes two types of streets: those running east-west and north-south, with a variable number of buildings along each axis. While the edge streets and avenues are bi-directional, the other streets and avenues alternate as one-way. AEDVs must be capable of traversing these streets and avenues based on their respective directions. Each building within the city map possesses eight delivery points, and AEDVs must move efficiently between these points, with each movement consuming a specific amount of time or clock ticks.

In the initial phase of the project, the focus is on facilitating AEDV movement solely on the streets. These simulated vehicles are capable of traveling between buildings, and the simulation allows users to specify the origin and destination for up to four AEDVs. After an AEDV reaches its designated destination, users can specify new trips for the vehicle.
It's important to note that, in this phase, factors such as battery charge and rules-of-the-road are disregarded. The emphasis is on creating a functional visualization of AEDVs navigating the streets within the city map.

The project's software solution is expected to include a graphical map that provides a visual representation of the AEDVs' movements, with ASCII characters being an acceptable means of representation. The map can be larger than the screen, and provisions should be made for users to navigate and view different sections of the map. During idle times, users should be able to specify the vehicle number, as well as its origin and destination.

2.3 Objective

The primary objective of this project is to design and implement a simulation of an Autonomous Electric Delivery Vehicle (AEDV) system within a city environment. This simulation aims to showcase the AEDVs' capabilities by allowing users to configure a city map with different streets, buildings, and delivery points, and observe the AEDVs autonomously navigating these streets to make deliveries. The simulation will provide a visual representation of AEDVs in action, offering a practical demonstration of their movement and interactions within the city environment, serving as a valuable tool for testing and visualizing the potential of autonomous electric delivery systems.

2.4 Input Handling

The program accepts text files by drag and drop and processes the data, populating the map

2.5 Output

- The program has a graphic screen and an alternate screen for standard input.
- The graphic screen houses the simulation of the AEDV on the generated city map. While the alternate screen prompts user for input data and changes the simulation as needed without a need for recompilation.

2.6 Error Handling

- In cases where a wrong file is provided or an empty file, even a file with the wrong extension then the user will be prompted and informed about the invalidity of the file

2.7 Algorithm

The algorithm implemented in this project for simulating the Autonomous Electric Delivery Vehicle (AEDV) system is designed to create a dynamic and interactive environment where AEDVs can efficiently navigate streets and avenues to make deliveries within a virtual city. The core components of the algorithm are as follows:

City Map Generation: The algorithm will generate a city map based on user-specified parameters, including the number of buildings running east-west and north-south. It will define streets and avenues, with two-way traffic on edge streets and avenues while others alternate one-way.

AEDV Movement: Each building on the map will have eight delivery points, including west (W), east (E), north (N), south (S), northwest (NW), northeast (NE), southwest (SW), and southeast (SE). AEDVs will navigate the streets and avenues according to their direction. The algorithm will calculate the time (in clock ticks) required for AEDVs to move between different points.

User Interaction: The simulation will allow user interaction to specify the origin and destination of up to four AEDVs. Users can configure trips for each AEDV, and the vehicles will autonomously navigate the streets to reach their destinations.

Visualization: The algorithm will provide a graphical representation of the city map, displaying AEDVs' movements using ASCII characters. The map can be larger than the screen, allowing users to navigate and view different sections of the city.

Simulation Control: The algorithm will include a simulation control mechanism to slow down the output, enabling users to observe AEDVs' actions. This feature will use a "Sleep(msec)" function to control the simulation's speed.

Modular Design: The software will be designed in a modular and general-purpose manner, allowing for flexibility and easy modification of the system by changing data rather than altering the code itself. This modularity ensures that the software remains adaptable to future changes and requirements.

Testing: To validate the system's functionality, the algorithm will include a comprehensive set of tests, each described in detail, along with their expected outcomes. These tests will demonstrate that the software fulfills the specified requirements.

Overall, this algorithm provides a versatile and interactive platform for demonstrating the capabilities of AEDVs in a simulated urban environment, offering a visual representation of their movements and interactions within the city while facilitating user interaction for testing and exploration.

2.8 Design Considerations

The design of the simulation for an Autonomous Electric Delivery Vehicle (AEDV) system involves several critical considerations to ensure its effectiveness, flexibility, and user-friendliness. These design considerations are paramount in creating a robust and user-centric simulation:

User Interface (UI): The simulation must have an intuitive and visually engaging user interface. It should provide an interactive map that allows users to easily specify the origin and destination of AEDVs. The UI should be designed with user experience in mind, enabling users to navigate the virtual city efficiently.

Scalability: The simulation should be scalable to accommodate varying city sizes and the number of AEDVs. It should handle city maps of different dimensions, with the ability to expand and contract the city layout as required.

Modularity and Extensibility: A modular design is crucial to facilitate future enhancements and modifications. The software should be structured in a way that allows for easy integration of additional features, such as new vehicle types, advanced delivery strategies, or real-time data updates.

Performance Optimization: The algorithm should be optimized for performance to ensure that AEDVs' movements are smooth and responsive. Efficient data structures and algorithms should be employed for pathfinding and map rendering.

Realism and Accuracy: While the simulation may not consider all real-world complexities, it should maintain a level of realism in terms of vehicle movements and city layout. AEDVs should follow logical routes, and their actions should mimic real-world navigation, including obeying traffic rules and recognizing obstacles.

Testing and Debugging: The design should incorporate built-in testing mechanisms for debugging and validation. These mechanisms should allow developers to test various scenarios and ensure that the simulation functions as expected. Testing scenarios should be well-documented.

User Feedback: The system should provide feedback to users, allowing them to track the progress of AEDVs and understand the simulation's status. Real-time updates, such as status messages and delivery confirmations, can enhance user engagement.

Educational Value: The simulation should have educational value by providing insights into AEDV operations, navigation challenges, and urban logistics. It should serve as a tool for learning and experimentation.

Documentation: Comprehensive documentation, including user guides and developer manuals, should accompany the simulation. This documentation should explain the simulation's features, controls, and the underlying algorithms, making it accessible to users and future developers.

Security and Privacy: If the simulation includes features related to real-world data or interactions, it should prioritize data security and user privacy. Measures should be in place to protect sensitive information.

Cross-Platform Compatibility: To maximize accessibility, the simulation should be designed to run on multiple platforms and devices. Compatibility with both desktop and mobile environments may be desirable.

Real-time Simulation: Consider whether the simulation should operate in real-time or allow users to control the simulation speed. Real-time simulation provides a dynamic experience, while adjustable speed allows users to analyze details more closely.

2.9 Data Dictionary

Below is a compressed data dictionary for the software, full implementation, and dictionary available in initial design document.

```
Function print_msg(msg):
  = Display a message in the diagnostic area of the screen.
  + Move cursor to the lower-left corner of the screen
  + Clear the screen
  + Set text color to white on a blue background
  + Print the message
  + Reset text and background colors
Function box(ulx, uly, name, colour):
  = Draw a 3x4 box with a name at the specified position.
  + Set the text and background color
  + Use DEC line drawing characters
  + Draw a 3x4 box at the specified position
  + Set the text and background colors back to default
Function populate_map(xrow, ycol):
  = Populate the map with buildings.
  + Clear the screen
  + For each row:
    { For each column:
      [ Calculate the coordinates of the box
        Draw a blue box at the calculated position ]
Function terminate(msg):
  = Terminate the program in case of a fatal error.
  + Display an error message
  + Wait for user input
  + Terminate the program
Main:
  = The main program.
  + Read an input file
  + Initialize a linked list of buildings
  + Read the dimensions of the map
  + For each line in the input file:
    { Parse building information
      [ Add the building to the linked list ]
  + Initialize the VT-100 terminal
  + Set up the main screen based on the window size
  + Populate the map with buildings
  + Handle keyboard input
  + Simulate the movement of a vehicle
  + Display "Done!" message
  + Wait for user input
Function addBuilding(head, x, y, type, quad):
  = Add a new building to the linked list.
  + Create a new building node
  + Set its attributes based on the provided information
  + Add the new node to the linked list
  + Return the updated head of the linked list
```

3 Pseudo-Code (Language Agnostic)

3.1 Header File (Function Declaration)

Header File: VT100.h

Pseudo-Code:

```
1. Include necessary header files and libraries.
2. Define macros and constants
     2.1. Define ESC as "\x1b"
     2.2. Define CSI as "\x1b["
     2.3. Define TRUE as 1
     2.4. Define FALSE as 0
     2.5. Define speedforeyes as 400
     2.6. Define NUL as '\0'
3. Declare global variables
     3.1. Initialize xbuild, ybuild
     3.2. Initialize numofv
     3.3. Define an enumeration VT100_COLOURS with constants
FGWHITE, BGRED, BGGREEN, BGYELLOW, BGBLUE, and BGCYAN
     3.4. Initialize AEDV1 as a struct car
4. Define structures for vehicles, positions, and buildings
     4.1. Define a struct vehicle with fields speed, x, y, xinc,
yinc
     4.2. Define a struct xypos with fields x and y
     4.3. Define a struct car with fields originx, originy,
carsym, car (xypos), dest (xypos), xinc, and yinc
     4.4. Define a struct Building with fields x, y, type, quad,
and next (pointer to the next building)
5. Declare external data and functions
     5.1. Declare HANDLE scrout, keyin
     5.2. Declare COORD scr_size
     5.3. Declare and define a function status_window()
     5.4. Declare and define a function check_kb() that returns
an integer
     5.5. Declare and define a function move(v1: struct car)
     5.6. Declare and define a function screen_size()
     5.7. Declare and define a function print_msg(msg: string)
     5.8. Declare and define a function box(ulx, uly, name,
colour)
     5.9. Declare and define a function populate_map(xrow, ycol)
6. Define a function to add buildings to the list
     6.1. Define a function addBuilding(head: struct Building,
x, y, type, quad) -> struct Building
     6.2. Initialize buildingList as NULL
```

3.2 Function Declaration

File: Altscreen.c

Pseudo-Code:

```
1. Include the necessary header file (VT100.h).
2. Declare an external structure of type CAR called AEDV1.

3. Define a function called screen_size:
    3.1. Get the new screen size using Windows API.
    3.2. Calculate the new screen size based on the information
obtained.
    3.3. Clear the screen and hide the cursor.

4. Define a function called status_window:
    4.1. Switch to the status window (Alternate screen).
    4.2. Display information about AEDV1.
    4.3. Ask the user to specify the number of AEDVs to move.
    4.4. Prompt the user for the origin and destination
coordinates.

5. Define a function called check_kb:
    5.1. Check if a keystroke is detected.
    5.2. If the keystroke is '#' (end program), return True.
    5.3. If the keystroke is 'r' or 'R' (resize screen):
        5.3.1. Call the screen_size function to resize the
screen.
        5.3.2. Populate the map based on specified dimensions.
        5.3.3. Print a message indicating the screen has been
resized.
        5.3.4. Return False.
    5.4. If the keystroke is anything else:
        5.4.1. Call the status_window function to display
vehicle status.
        5.4.2. Return False.
```

File: MoveVehicle.c
Pseudo-Code:

```
1. Include the necessary header file "VT100.h"
2. Initialize global variables: scrout, keyin, and scr_size
3. Initialize the vehicle VEHICLE v1
4. Define an array of XYPOS structures, xy[], to represent the
route for the vehicle

5. Create a function named move(CAR* v1) for moving the vehicle
   5.1. Initialize local variables org, dst, pos, and stop
   5.2. Enter a loop to control the movement
      5.2.1. Check if the current y-coordinate of org is equal to the y-coordinate of dst
         5.2.1.1. If true, the vehicle is moving in the X direction
         5.2.1.2. Set the X and Y increments for movement
         5.2.1.3. Initialize the vehicle's position based on org
         5.2.1.4. Enter a loop for moving the vehicle in the X direction
            5.2.1.4.1. Display the vehicle at its current position
            5.2.1.4.2. Pause for a specified duration (speedforeyes) for visualization
            5.2.1.4.3. Erase the vehicle at its previous position
            5.2.1.4.4. Check for keyboard input (stop if '#' is detected)
            5.2.1.4.5. Update the vehicle's x-coordinate
         5.2.1.5. End the loop when the destination x-coordinate (dst.x) is reached or '#' is detected

      5.2.2. If the y-coordinates of org and dst are not equal, the vehicle is moving in the Y direction
         5.2.2.1. Set the Y and X increments for movement
         5.2.2.2. Initialize the vehicle's position based on org
         5.2.2.3. Enter a loop for moving the vehicle in the Y direction
            5.2.2.3.1. Display the vehicle at its current position
            5.2.2.3.2. Pause for a specified duration (speedforeyes) for visualization
            5.2.2.3.3. Erase the vehicle at its previous position
            5.2.2.3.4. Check for keyboard input (stop if '#' is detected)
            5.2.2.3.5. Update the vehicle's y-coordinate
         5.2.2.4. End the loop when the destination y-coordinate (dst.y) is reached or '#' is detected

   5.3. End the loop when the stop condition is met

6. The pseudocode outlines the procedure for moving a vehicle following the specified route with optional user input to stop the movement.
```

3.3 Main Function Pseudo-Code

File: VT100.c

Pseudo-Code:

```
1. Define Function print_msg(msg):
   1.1. Move cursor to the lower-left corner of the screen
   1.2. Clear the screen
   1.3. Set text color to white on a blue background
   1.4. Print the message
   1.5. Reset text and background colors
2. Define Function box(ulx, uly, name, colour):
   2.1. Set the text and background color
   2.2. Use DEC line drawing characters
   2.3. Draw a 3x4 box at the specified position
   2.4. Set the text and background colors back to default

3. Define Function populate_map(xrow, ycol):
   3.1. Clear the screen
   3.2. For each row:
       3.2.1. For each column:
           3.2.1.1. Calculate the coordinates of the box
           3.2.1.2. Draw a blue box at the calculated position
4. Define Function terminate(msg):
   4.1. Display an error message
   4.2. Wait for user input
   4.3. Terminate the program
5. Main:
   5.1. Read input file
   5.2. Initialize a linked list of buildings
   5.3. Read the dimensions of the map
   5.4. For each line in the input file:
       5.4.1. Parse building information
       5.4.2. Add the building to the linked list
   5.5. Initialize the VT-100 terminal
   5.6. Set up the main screen based on the window size
   5.7. Populate the map with buildings
   5.8. Handle keyboard input
   5.9. Simulate the movement of a vehicle
   5.10. Display "Done!" message
   5.11. Wait for user input
6. Function addBuilding(head, x, y, type, quad):
   6.1. Create a new building node
   6.2. Set its attributes based on the provided information
   6.3. Add the new node to the linked list
   6.4. Return the updated head of the linked list
```