Technische Universität München
24/25

Mapping for a Sustainable World – Seminar

SoSe 2025

Crowdsensing of parking lots with static and dynamic Lidar sensors

Student: Marko Mrsa
Matriculation number: 03799093
Mail contact: marko.mrsa@tum.de
Supervisor: Dr. Yu Feng y.feng@tum.de

Chair of Cartography and Visual Analytics

Abstract

Urban parking remains a continuous challenge in modern cities, contributing to traffic congestion, driver frustration, and unnecessary CO_2 emissions. This project presents a browser-based simulation platform that uses crowdsensed LiDAR data from static infrastructure and vehicle-mounted sensors to explore a different approach to real-time parking detection. The system integrates point cloud processing, SUMO-based traffic simulation, and interactive 3D visualization to create a digital twin of urban parking dynamics.

While the current implementation is limited to a local simulation and does not yet connect to live sensors or a backend system, it serves as a functional proof-of-concept for LiDAR-driven occupancy detection and client-side spatial logic. The platform demonstrates the technical possibility of integrating multi-source spatiotemporal data to simulate real-time parking scenarios, detect space occupancy, and visualize user parking behavior at a large scale.

Apart from Simulation, the project introduces a broad idea focused on improving navigation systems by adding real-time parking, guidance, dynamic route adjustments, and allowing users to reserve parking spots in advance. This idea creates a starting point for building parking assistance systems that can work at different scales, do not depend on specific sensors, and use existing technologies to support smarter and more flexible urban mobility in the future.

Contents

Table of Figures:

1. Introduction

This project presents a browser-based 3D simulation platform for real-time parking detection and vehicle tracking in urban areas. It combines point cloud data with a SUMO.[1]-based traffic simulation [1], and modern visualization techniques to create an interactive environment where users can analyze vehicle movement and parking behavior within a defined zone.

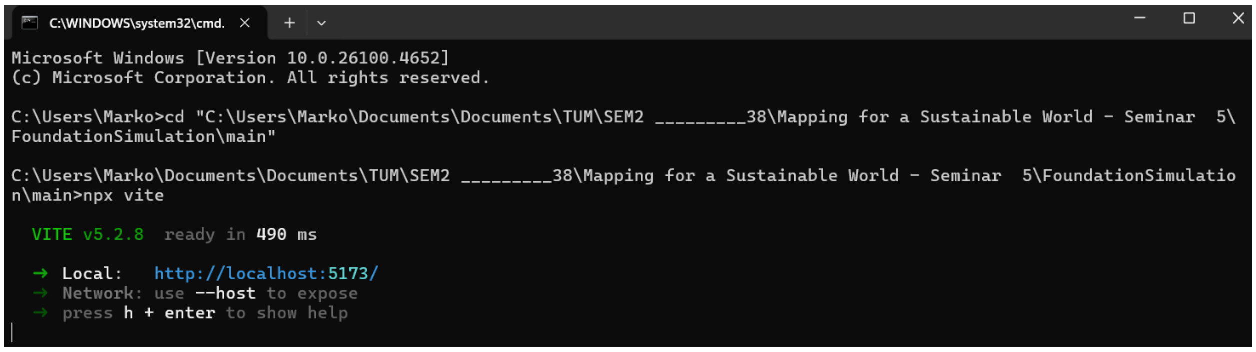
The high cost of infrastructure is one of the biggest challenges in developing real-world systems like this. Equipping urban areas with enough sensors and reliable communication networks requires a significant investment. While the idea and software solutions are conceptually in place, the hardware side, especially the large-scale deployment of sensors, is still a limiting factor in feasibility and affordability.

The current project builds upon a previous bachelor's thesis, which introduced an early version of a parking simulation. That version consisted of a single, unstructured codebase with around 1500 lines, making it difficult to understand, modify, or extend. Despite investing significant time into analyzing its logic, the results remained inconsistent, and the codebase was unsuitable for direct reuse. As a result, a new simulation was developed from scratch, inspired by the original concept but redesigned with improved modular structure, visual presentation, and customized traffic logic. While certain parts of the previous implementation were reviewed for general understanding, all necessary components were either rebuilt or significantly modified before integration.

[1] "**S**imulation of **U**rban **M**obility" (SUMO) is an open-source, highly portable, microscopic, and continuous traffic simulation package designed to handle large networks.

2. Initial Implementation (Version 0)

The initial simulation that was provided as a foundation for this project was to simulate a mobile laser scanning of the area, detect occupied parking spaces, display the number of free parking spaces, simulate real-time traffic with different car sizes, and provide tracking for some vehicles. For future references, this version of the simulation will be referenced as Version 0. So, version 0 is being initialized by running it from the command prompt.



```
Microsoft Windows [Version 10.0.26100.4652]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Marko>cd "C:\Users\Marko\Documents\Documents\TUM\SEM2 _________38\Mapping for a Sustainable World - Seminar  5\FoundationSimulation\main"

C:\Users\Marko\Documents\Documents\TUM\SEM2 _________38\Mapping for a Sustainable World - Seminar  5\FoundationSimulation\main>npx vite

  VITE v5.2.8  ready in 490 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: use --host to expose
  ➜  press h + enter to show help
```

Figure 1 Opening Version 0

Following the HTTP link opens the HTML in a web browser.

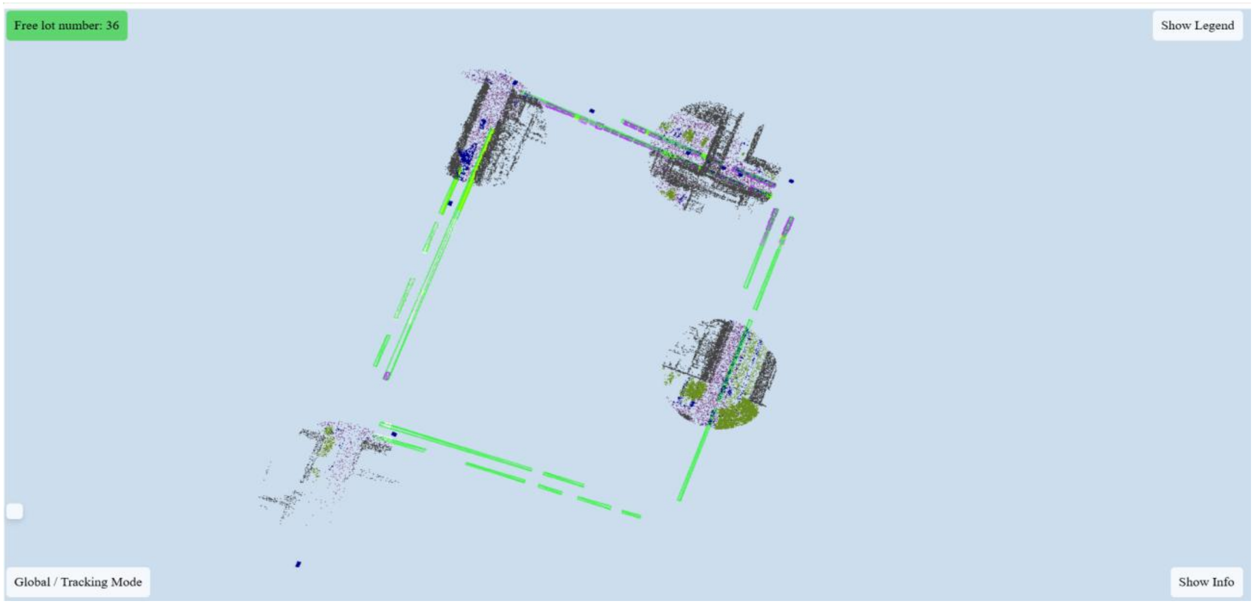


Figure 2 Initial Preview of Version 0

The first thing noticed was that the free lot number is 36, while it's the start of the simulation, only a few vehicles have been detected, and basically none of the free parking spots. Indicating that the free lot number is not properly connected with the real-time data. This simulation uses parking zones and not parking spaces; it uses different vehicle sizes. Also, the number of vehicles exponentially grows and at some point, overflows the scene.

Secondly, the detection logic of the point cloud is none. In this simulation, the vehicle does all the sensing, point clouds only render, and sometimes overlap with vehicles, looking like it's an MLS. In

Figure 3, when looking at the street on the east side of the simulation block, we don’t see any parked or free parking spaces, only the point cloud, and this is because none of the vehicles have passed this street yet.

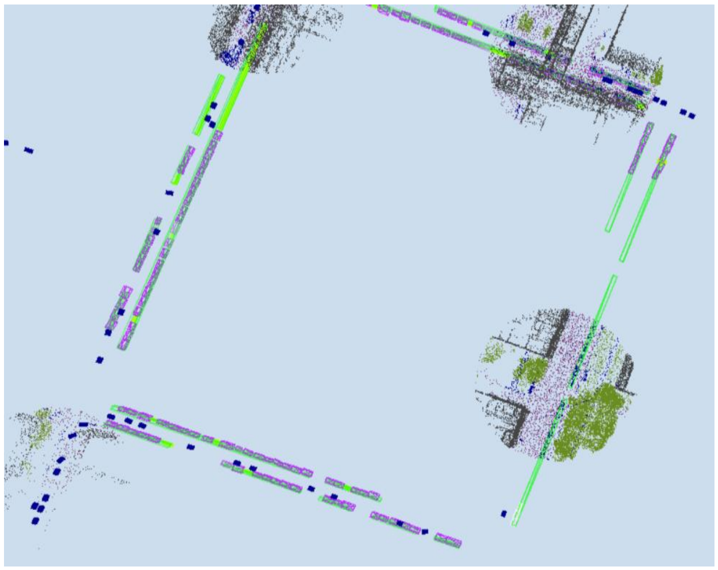


Figure 3 Version 0 Detect Logic

As seen below, the green wireframe covers the whole parking area. The purple wireframe is constructed every time a vehicle senses a parked vehicle. Also, if it senses enough free space, it constructs a green cube with proper dimensions and orientation.

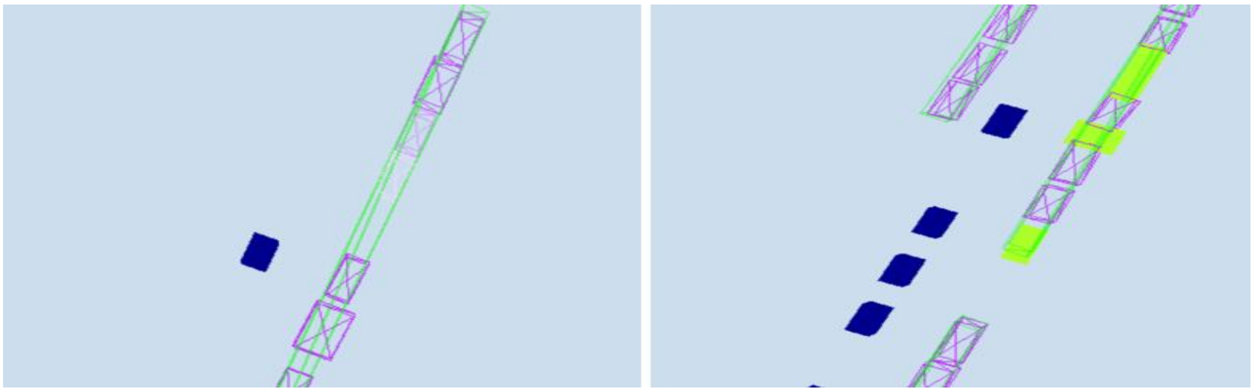
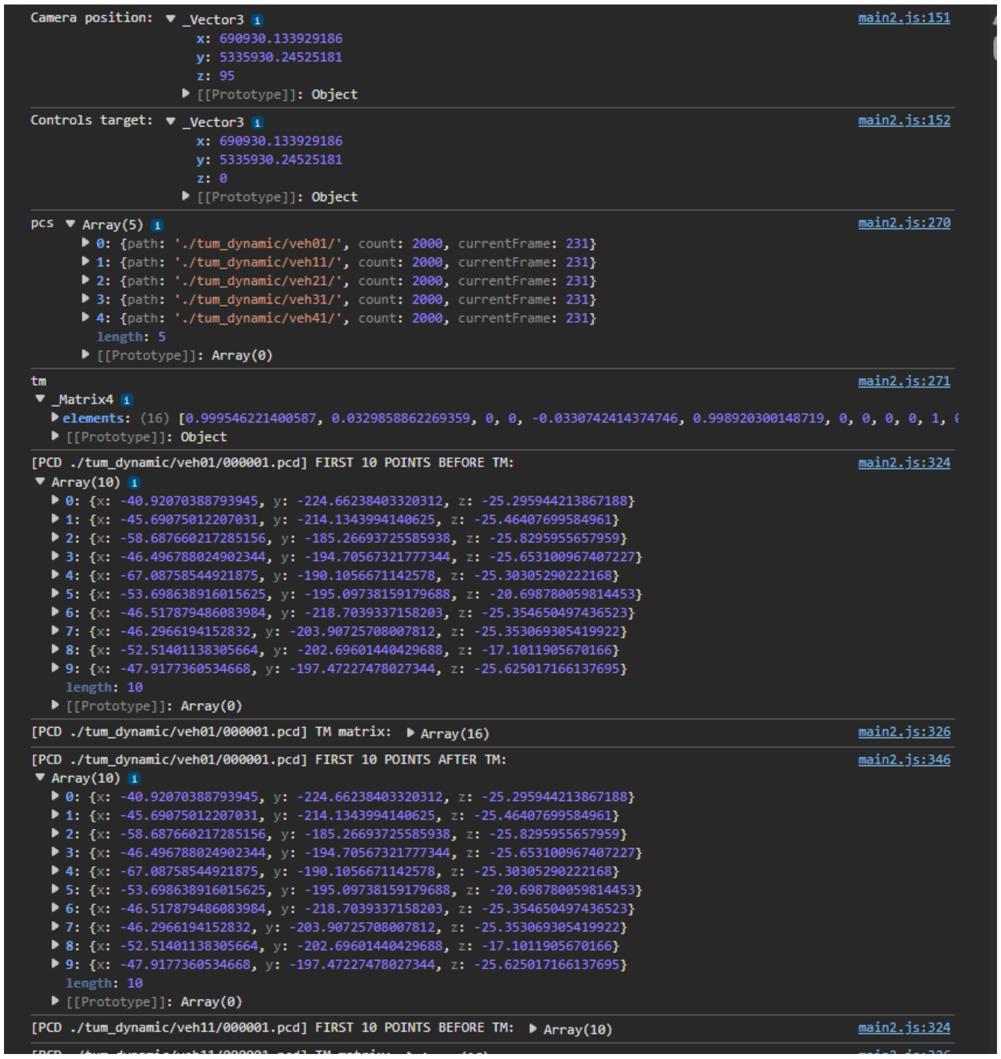


Figure 4 Version 0 Parking spaces Zoom In

The camera and target coordinates are set in the UTM 32 coordinate system (e.g., x ≈ 690,930, y ≈ 5,335,930), and for the point clouds, there is a proper transformation matrix set in the configuration matrix. But, despite applying a non-identity transformation matrix (TM) intended to move point clouds into the UTM coordinate system, the point clouds' coordinates remain clustered around the origin (near (0,0,0)), both before and after the transformation. This is confirmed by multiple console log outputs. Despite this observation, the camera and target coordinates are set in the UTM zone, meaning that point clouds shouldn’t be visible, but they are. Code review and logging confirm that no other translation or scaling is applied to the point clouds after loading. This issue remains unexplained.



```
Camera position:  ▼ _Vector3                                                   main2.js:151
                      x: 690930.133929186
                      y: 5335930.24525181
                      z: 95
                    ▶ [[Prototype]]: Object
Controls target:  ▼ _Vector3                                                   main2.js:152
                      x: 690930.133929186
                      y: 5335930.24525181
                      z: 0
                    ▶ [[Prototype]]: Object
pcs  ▼ Array(5)                                                                main2.js:270
       ▶ 0: {path: './tum_dynamic/veh01/', count: 2000, currentFrame: 231}
       ▶ 1: {path: './tum_dynamic/veh11/', count: 2000, currentFrame: 231}
       ▶ 2: {path: './tum_dynamic/veh21/', count: 2000, currentFrame: 231}
       ▶ 3: {path: './tum_dynamic/veh31/', count: 2000, currentFrame: 231}
       ▶ 4: {path: './tum_dynamic/veh41/', count: 2000, currentFrame: 231}
         length: 5
       ▶ [[Prototype]]: Array(0)
tm                                                                             main2.js:271
▼ _Matrix4
  ▶ elements: (16) [0.999546221400587, 0.0329858862269359, 0, 0, -0.0330742414374746, 0.998920300148719, 0, 0, 0, 0, 1, 
  ▶ [[Prototype]]: Object
[PCD ./tum_dynamic/veh01/000001.pcd] FIRST 10 POINTS BEFORE TM:                main2.js:324
▼ Array(10)
  ▶ 0: {x: -40.92070388793945, y: -224.66238403320312, z: -25.295944213867188}
  ▶ 1: {x: -45.69075012207031, y: -214.1343994140625, z: -25.46407699584961}
  ▶ 2: {x: -58.687660217285156, y: -185.26693725585938, z: -25.8295955657959}
  ▶ 3: {x: -46.496788024902344, y: -194.70567321777344, z: -25.653100967407227}
  ▶ 4: {x: -67.08758544921875, y: -190.1056671142578, z: -25.30305290222168}
  ▶ 5: {x: -53.698638916015625, y: -195.09738159179688, z: -20.698780059814453}
  ▶ 6: {x: -46.517879486083984, y: -218.7039337158203, z: -25.354650497436523}
  ▶ 7: {x: -46.2966194152832, y: -203.90725708007812, z: -25.353069305419922}
  ▶ 8: {x: -52.51401138305664, y: -202.69601440429688, z: -17.1011905670166}
  ▶ 9: {x: -47.9177360534668, y: -197.47227478027344, z: -25.625017166137695}
    length: 10
  ▶ [[Prototype]]: Array(0)
[PCD ./tum_dynamic/veh01/000001.pcd] TM matrix:  ▶ Array(16)                   main2.js:326
[PCD ./tum_dynamic/veh01/000001.pcd] FIRST 10 POINTS AFTER TM:                 main2.js:346
▼ Array(10)
  ▶ 0: {x: -40.92070388793945, y: -224.66238403320312, z: -25.295944213867188}
  ▶ 1: {x: -45.69075012207031, y: -214.1343994140625, z: -25.46407699584961}
  ▶ 2: {x: -58.687660217285156, y: -185.26693725585938, z: -25.8295955657959}
  ▶ 3: {x: -46.496788024902344, y: -194.70567321777344, z: -25.653100967407227}
  ▶ 4: {x: -67.08758544921875, y: -190.1056671142578, z: -25.30305290222168}
  ▶ 5: {x: -53.698638916015625, y: -195.09738159179688, z: -20.698780059814453}
  ▶ 6: {x: -46.517879486083984, y: -218.7039337158203, z: -25.354650497436523}
  ▶ 7: {x: -46.2966194152832, y: -203.90725708007812, z: -25.353069305419922}
  ▶ 8: {x: -52.51401138305664, y: -202.69601440429688, z: -17.1011905670166}
  ▶ 9: {x: -47.9177360534668, y: -197.47227478027344, z: -25.625017166137695}
    length: 10
  ▶ [[Prototype]]: Array(0)
[PCD ./tum_dynamic/veh11/000001.pcd] FIRST 10 POINTS BEFORE TM:  ▶ Array(10)     main2.js:324
```

Figure 5 Version 0 Coordinate Confusion

The results of this sensing are of high accuracy. The vehicle detection logic determines if there is a parked vehicle or not in the parking zone by comparing the position of preprocessed JSON files that were manually created for each parked car and the parking zone positions.

This project started with version 0, tried to upgrade the functionalities and correct the logic, but ended up as a completely different project. For future reference, the final project will be referred to as the final version.

The basis remains the same, the Point cloud data is the same, and the goal is to detect free parking lots. Major changes in the final simulation are as follows:

- 236 defined parking spots, instead of 20 parking zones
- The vehicle detection logic is encoded only for the Point clouds
- There is only one SUMO vehicle used for testing the navigation and parking.
- Point clouds are transformed into UTM32 before loading into three.js [2]
- Improved Visualization and UI of the scene
- Modularization using ES6 modules (18 js script instead of 1)

3. Data Sources and Preprocessing

3.1 Point Clouds

Point Clouds have been extracted from a TUM2TWIN [8] point cloud dataset for the Arcisstrasse 21 · 80333 München University building. The dataset consists of a static point cloud representing the entire environment within a local coordinate system. In addition to this, the author received five vehicle-specific folders, each containing approximately 2,000 PCD files. Each file captures a circular scan centered on the street axis, representing data collected through Mobile Laser Scanning (MLS). [2]On a moving vehicle. This principle has been used to simulate the environment. So, the final Simulation loads the point clouds from each vehicle file simultaneously, one by one, in a closed loop.
The Simulation occurs in UTM 32 EPSG (25832) projection, hence the need for georeferencing the point clouds.

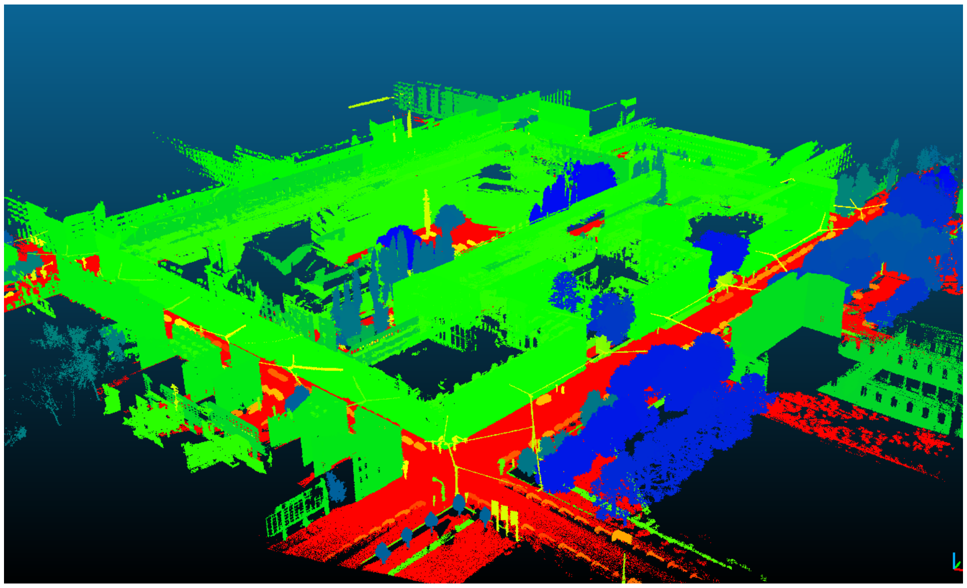
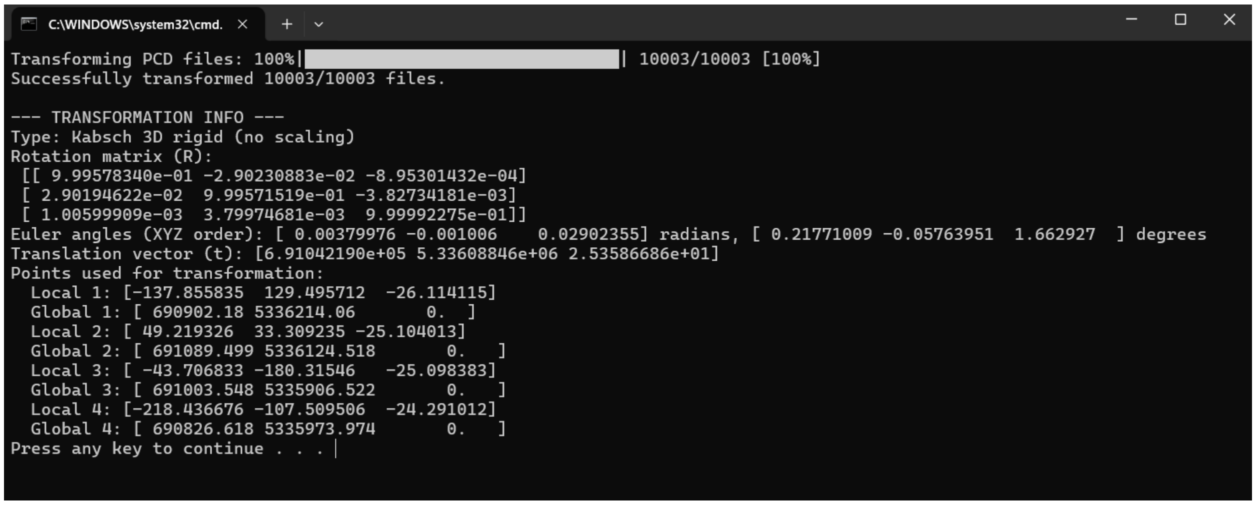


Figure 6 The Main PCD File

Preprocessing:
As no specific methodology for the registration of this large point cloud dataset was available to the author, a Python script was developed as an improvised solution, which takes 8 point(x,y,z) from a local system and four corresponding points from the global system, in this case specifically Cloud Compare has been used [6] for identifiyng the corner of the buildings in the local system and QGIS open street maps layer for identifying the same corners in the UTM32 corrdinate system, the script then calculates six parameteres for the rigib body transformation (3 rotations and 3 translations). It applies it to the whole dataset, creating new files named tumdynamic_utm. The resulting .pcd files are placed in the appropriate directory. The Python Script can be found under the name transform_pcd_to_utm_auto.py

[2] MLS- Mobile Laser Scanning



C:\WINDOWS\system32\cmd.

```
Transforming PCD files: 100%|██████████████████████████████| 10003/10003 [100%]
Successfully transformed 10003/10003 files.

--- TRANSFORMATION INFO ---
Type: Kabsch 3D rigid (no scaling)
Rotation matrix (R):
 [[ 9.99578340e-01 -2.90230883e-02 -8.95301432e-04]
 [ 2.90194622e-02  9.99571519e-01 -3.82734181e-03]
 [ 1.00599909e-03  3.79974681e-03  9.99992275e-01]]
Euler angles (XYZ order): [ 0.00379976 -0.001006    0.02902355] radians, [ 0.21771009 -0.05763951  1.662927  ] degrees
Translation vector (t): [6.91042190e+05 5.33608846e+06 2.53586686e+01]
Points used for transformation:
  Local 1: [-137.855835  129.495712  -26.114115]
  Global 1: [ 690902.18 5336214.06       0.  ]
  Local 2: [ 49.219326  33.309235 -25.104013]
  Global 2: [ 691089.499 5336124.518       0.   ]
  Local 3: [ -43.706833 -180.31546   -25.098383]
  Global 3: [ 691003.548 5335906.522       0.   ]
  Local 4: [-218.436676 -107.509506  -24.291012]
  Global 4: [ 690826.618 5335973.974       0.   ]
Press any key to continue . . .
```

Figure 7 PCD Georeferencing Report

3.2 Parking Spaces

The logic behind creating the full cuboid (boxes) for each parking spot starts with manually digitizing point by point for each edge corner of the parking spot (This has been done using QGIS and open street map parking zones, in the proper coordinate system) [5].

Preprocessing:
By making points in the same pattern (4 points construct a parking lot; number of parking spots * 4 = total number of points), after sampling all the parking lots, calculate the x and y coordinates of the points and export the shapefile as a CSV file. Then load the CSV file into a Python script, which creates a proper order of coordinates for each parking lot (number of parking spots = number of tumparkingXX.csv files).

Configuration
The final folder containing multiple CSV parking files is set in the root folder *main/parkingspaces//* and is ready for loading.

3.3 City GML

City GML data has been downloaded via the Bavarian Open Data Website, the 3D-Gebäudemodelle (LoD2) [3]. This resource had to be converted into a .glb file using the FME software [7], for use in the three.js environment. Using a JS function for GLTF [3]Loading the building into the model successfully.

3.4 Satellite Imagery

Satellite imagery has been downloaded via the Bavarian Open Data Website, the Digitales Orthophoto RGB 20cm [4]. This resource had to be manually clipped in QGIS so that the three.js server can handle the load. This imagery has been used for the ground surface to add more semantics to the Simulation. With it, the roads, parking spaces, vegetation, and urban environment are clear. It lacks vertical semantics but perfectly covers the horizontal. Of course, the whole Simulation assumes that the area is completely flat.

[3] **glTF** (Graphics Library Transmission Format) is a standard file format for three-dimensional scenes and models. A glTF file uses one of two possible file extensions: .gltf (JSON/ASCII) or **.glb** (binary)

4. Simulation Framework

The Simulation's primary goal is to detect and update the status of each parking lot. Other goals include visualizing the environment, real-time change detection, and navigation.

4.1 Architecture Overview

- Utilizes Three.js [4] For rendering point clouds, vehicles, parking spots, and buildings in a realistic 3D environment.
- Detects and tracks the occupancy status of individual parking spots using vehicle detection data and spatial algorithms.
- Imports and animate vehicle routes generated from OpenStreetMap (OSM) and SUMO, providing ground-truth vehicle trajectories.
- Loads and visualizes large-scale point cloud datasets (PCD [5]Format).
- Kalman filters are applied to smooth noisy vehicle detection data and improve vehicle tracking accuracy [11].
- Offers real-time controls for simulation speed, point cloud size, navigation to specific parking spots, and manual override of parking statuses.
- Tracks and displays system performance metrics such as FPS and render time.

The system's architecture is frontend only, meaning the entire application runs in the browser. There is no backend server; all logic, data processing, and rendering are performed client-side using JavaScript modules.

The codebase is organized into ES6. [6]Modules, each responsible for a specific aspect of the Simulation (e.g., data processing, parking logic, animation, UI controls). *(Modular JavaScript)*

4.2 Technologies Used

- **Three.js:** 3D rendering and scene management.
- **Math.js:** Advanced mathematical operations and matrix calculations.
- **SUMO:** External tool for traffic simulation and route generation.
- **JavaScript (ES6 Modules):** Core application logic and data processing.
- **HTML/CSS:** User interface and layout.

[4] **Three.js** is an open-source **JavaScript library** designed to create and render **3D graphics** directly in web browsers. It leverages **WebGL** (Web Graphics Library) to utilize the GPU for rendering high-performance 3D content. This library simplifies the process of working with WebGL by providing an intuitive API for creating 3D scenes, animations, and interactive graphics

[5] **PCD**- **Organized point cloud** dataset is the name given to point clouds that resemble an organized image (or matrix) like structure, where the data is split into rows and columns

[6] **ES6 modules** enhance **JavaScript** by allowing developers to modularize code, breaking it into manageable pieces that can be imported and reused across different parts of an application.

4.3 Workflow

1. **Initialization:**

Loads configuration from a text file (Config.txt).
Sets up the Three.js scene, camera, lighting, and UI elements.
Loads parking spot definitions and initializes the parking system.

2. **Data Loading:**

Loads point cloud frames and vehicle detection data for each simulation step.
Loads and parses SUMO route files for ground-truth vehicle animation.

3. **Simulation Loop:**

For each frame, Point Clouds:
Update vehicle positions using detection logic and Kalman filtering.
Checks parking spot occupancy by matching vehicle positions to parking areas.
Updates the 3D scene and UI to reflect the current parking and vehicle status.
Animates vehicles along SUMO routes for comparison and validation.

4. **User Interaction:**

Users can adjust simulation parameters (vehicle speed, pointcloud point size), navigate to specific parking spots, and manually override parking statuses.
Real-time feedback is provided through tables, legends, and performance indicators.

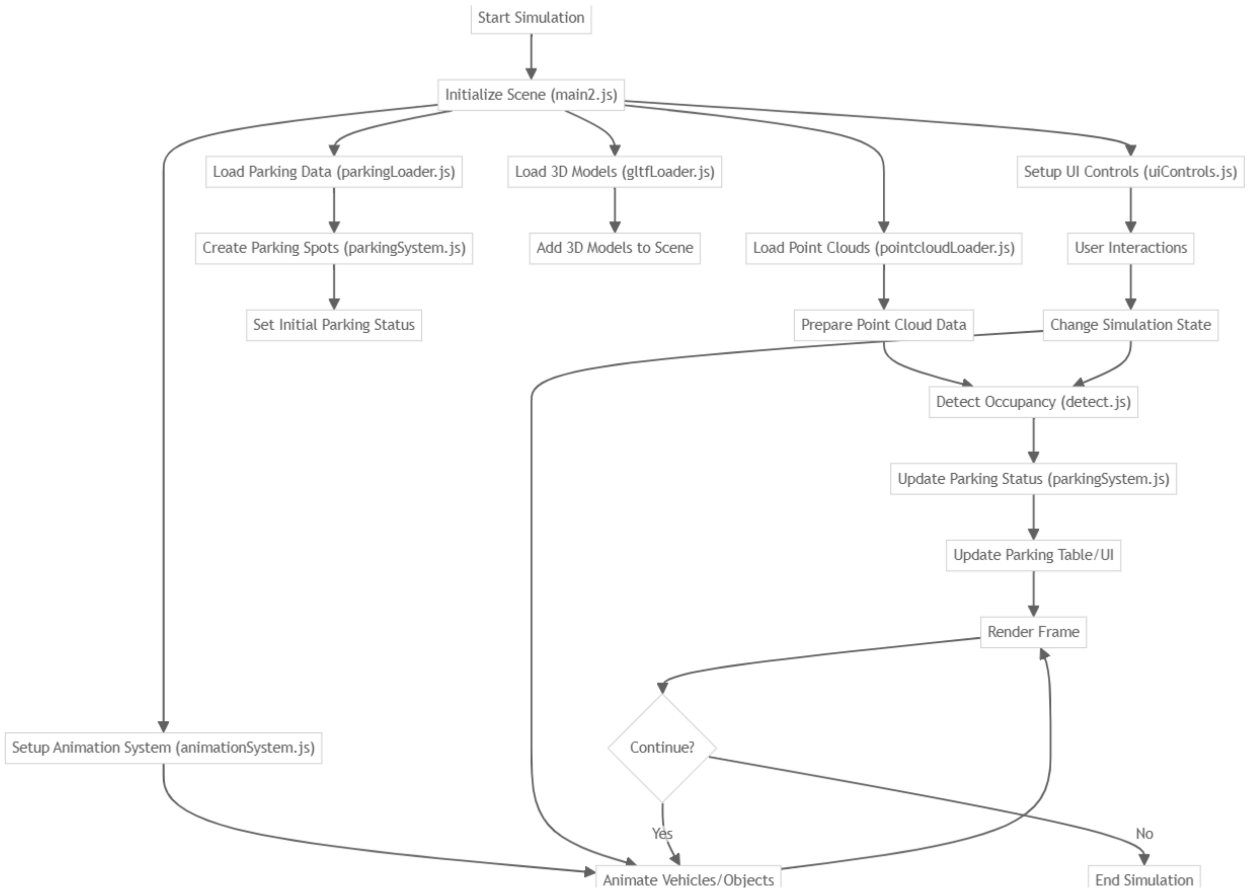

Start Simulation
Initialize Scene (main2.js)
Load Parking Data (parkingLoader.js)
Load 3D Models (gltfLoader.js)
Setup UI Controls (uiControls.js)
Create Parking Spots (parkingSystem.js)
Add 3D Models to Scene
Load Point Clouds (pointcloudLoader.js)
User Interactions
Set Initial Parking Status
Prepare Point Cloud Data
Change Simulation State
Detect Occupancy (detect.js)
Update Parking Status (parkingSystem.js)
Update Parking Table/UI
Render Frame
Setup Animation System (animationSystem.js)
Continue?
Yes
No
Animate Vehicles/Objects
End Simulation

Figure 8 Workflow diagram of the simulation process.

5. Frontend Part of the Simulation: Module Breakdown

5.1 Parking and GLTF Loading

5.1.1Parking Loader (parkingLoader.js)

LoadCSV (filepath, scene, TM)

- Reads a CSV file describing a single parking area (usually 4 points for a rectangle).
- Parses the coordinates for the four corners of the parking spot.
- Optionally applies a transformation matrix (TM) to the coordinates (if provided).
- Creates a 3D mesh (a wireframe rectangle/box) using THREE.js to represent the parking spot.
- Adds the mesh to the scene at a fixed height (e.g., z = 2.0).
- Returns an object containing the area coordinates and the mesh.

Initializepark (scene, parkingAreas, parkinglots, parkingUIDMap, parkingStatusMap, TM, freescene)

- Loops through all parking CSV files (e.g., tumparking1.csv to tumparking236.csv).
- Loads each parking area using loadCSV.
- Stores the area and mesh in parkingAreas and parking lots.
- Assigns a unique ID (UID) to each parking spot.
- Calculates the center of each parking spot and stores it in parkingUIDMap.
- Initializes the status of each parking spot as "free" in parkingStatusMap.
- Optionally calculates the angle of each parking spot for further use.
- Handles errors and logs warnings for problematic parking areas.

3. updateParkingSpotColor (mesh, status)

- Changes the color of a parking spot mesh based on its status:
- Green for free
- Red for occupied
- Green as default

5.1.2 GLTF Loader (gltfLoader.js)

LoadFacultyModels (scene, camera, controls, renderer)

- Creates a new GLTFLoader instance.
- Defines a counter to track how many models have been loaded.
- Defines a helper function (checkAllLoaded) to resolve the central promise when all models are loaded (or failed).

Loads GLTF models:

1. cityGML.glb
    - Loads the model.
    - Sets its scale, rotation, and position.
    - Adds it to the scene.
    - Calls checkAllLoaded () when done or on error.

Handles window resize events to update the camera aspect ratio and renderer size.
Returns a Promise that resolves when all three models are loaded (or have failed to load).

5.2 Parking System (parkingSystem.js)

Manages the logic for parking spots: their creation, status (free/occupied), and updates based on detection (e.g., point cloud or simulation). It starts with …

Initialization:

- Loads parking spot definitions (positions, sizes) from a loader or data file.
- Creates 3D meshes (usually rectangles/boxes) for each parking spot and adds them to the scene.

Status Tracking:

- Each parking spot has a status property (e.g., "free", "occupied").
- The system keeps a list or map of all parking spots and their current status.

Detection Update:

- Receives input from detection logic (e.g., point cloud analysis or simulated vehicles).
- Check whether each parking spot should be marked as "occupied" or "free" based on the detection results (e.g., number of points inside the box).

Visual Update:

- Updates the color or appearance of each parking spot mesh based on its status (e.g., green for free, red for occupied).
- Optionally updates labels or UI elements to reflect the current status.

Interaction:

- May handle user interactions (e.g., clicking a spot to get info or override status).

Reset/Unlock:

- Provides a method to reset all spots to "free" or unlock them when switching detection modes.

5.3 Detect Module

The detect.js module is responsible for updating the status of each parking spot based on the detected positions of vehicles (or objects) in the scene. It does not render any visual objects, but only updates the status data structure.

Step-by-step logic:

1. **Reset update flags for all parking spots**
   - For every parking spot in the status map, set its updated flag to false at the start of each detection cycle.
2. **Count vehicles in each parking area**
   - For each detected vehicle:
     - For each parking area:
       - Calculate the center of the parking area.
       - If the vehicle is within 5 meters of the center, increment the vehicle count for that parking area and stop checking further areas for this vehicle.
3. **Update the status of each parking spot**
   - For each parking area:
     - Get the current time.
     - Check if the area is currently occupied (i.e., at least one vehicle detected in step 2).
     - Retrieve the current status and lock state of the parking spot.
   - **Status update logic:**
   - If currently occupied: set status to "occupied".

- If not currently occupied but status was "occupied":
  - If the spot is locked, keep it "occupied".
  - If not locked, but it was occupied within the last 5 seconds, keep it "occupied".
  - Otherwise, set status to "free".
- If none of the above, set status to "free".
- Update the status and last change time only if the status changed.
- Mark the spot as updated.

4. **Debug output for a specific spot (UID 83)**

- If the spot UID is 83, print debug information about its status update.

5. **Set status to "unknown" for any spot that did not update this frame**

- For any parking spot whose updated flag is still false, set its status to "unknown".

6. **Count and display statistics**

- Count the number of spots that are "free", "occupied", and "unknown".
- Update the UI element (with id occupancyRate) to show the number of free spots.

5.4 Point Cloud Loading and Vehicle Detection

During the Simulation, the point cloud files are loaded "as is" and visualized in the 3D scene. The pointcloudLoader firstly loads the PCDs frame by frame, removing each previous frame as the new one loads, storing the frame into the pcdExistArray.

Then performs the detection using the detect.js module and the inside function from the utils.js module. If the parking zones (parkingAreas) are forwarded, detection of parking space occupancy is started:

- It initializes a point counter for each parking zone.
- For each point from the point cloud:
  - It is transformed by the TM matrix.
  - First, check if there is a parking zone in the XY bounding box (AABB optimization).
  - Then check if it is inside the polygon of the parking zone (inside).
  - Then check if it is inside the 3D box of the parking space in Z (use mesh height and position).
  - If so, increase the counter for that zone.
- If the number of points in the zone exceeds the threshold (adjustedThreshold), the parking space is set to 'occupied'.
- If it is less, it is set to 'free' (but only if it is not locked or targeted).
- Illuminates the parking space with the appropriate color (red for occupied, green for free).
- Updates the parking table and visualization, including the updatePointSize(newSize) function

5.5 Animation System (animationSystem.js)

Handles animations in the simulation, such as moving vehicles, updating object positions, or animating UI elements. It starts with …

Initialization:

- Sets up animation parameters and references to objects that will be animated (e.g., vehicles, parking spot highlights).

Animation Loop:

- Defines an update function on every frame (usually from the main render loop).
- Calculates new positions, rotations, or other properties for animated objects based on time, simulation state, or user input.

Vehicle Movement:

- Updates vehicle positions along predefined paths or according to simulation logic.
- Handles smooth transitions, acceleration, and deceleration.

Parking Spot Animation:

- Animates parking spot highlights or status changes (e.g., fading color when a spot becomes occupied).

UI Animation:

- Optionally animates UI elements (e.g., counters, status bars) in sync with simulation events.

Integration:

- Exposes functions to start, stop, or reset animations as needed by other modules.

5.6 Main Entry (main2.js)

Acts as the main entry point for the simulation, initializing the scene, loading data, and starting the render/animation loop. It starts with:

Scene Setup:

- Initializes the Three.js scene, camera, renderer, and lighting.
- Sets up the main canvas and attaches it to the HTML page.

Module Initialization:

- Loads and initializes other modules: parking system, animation system, loaders for point clouds, GLTF models, etc.

Data Loading:

- Loads required data files (e.g., parking definitions, point clouds, 3D models).
- Waits for all assets to be loaded before starting the simulation.

Object Creation:

- Adds all objects (parking spots, vehicles, models) to the scene.

Event Listeners:

- Sets up user input handlers (e.g., mouse, keyboard, UI buttons).

Main Loop:

- Starts the animation/render loop.
- On each frame:
- Updates the animation system.
- Updates the parking system (e.g., checks for status changes).
- Renders the scene.

UI Updates:

- Updates UI elements (e.g., parking table, status bar) in sync with the simulation state.

Cleanup:

- Handles cleanup or reset logic when restarting or stopping the simulation.

5.7 UI Features (uiControls.js)

Manages user interface controls and interactions for the simulation, connecting UI elements (buttons, sliders, checkboxes) to simulation logic. It starts with …

UI Element Setup:

- Selects or creates HTML elements (buttons, sliders, checkboxes, etc.) to control the simulation (e.g., play/pause, toggle point clouds, adjust parameters).

Event Listener Registration:

- Attaches event listeners to these UI elements (e.g., onClick, onChange).
- Each listener triggers a function in the simulation (e.g., starting/stopping animation, toggling visibility of objects, changing detection thresholds).

State Synchronization:

- Updates UI elements to reflect the current state of the simulation (e.g., disables buttons when not available, updates labels or counters).

Callback Integration:

- Provides callback functions that other modules (like main2.js or animationSystem.js) can use to update the UI in response to simulation events.

User Feedback:

- Optionally displays messages, warnings, or status updates to the user (e.g., “Loading...”, “Simulation paused”).

5.7.1 Real-time Parking Table

The real-time parking table is a dynamic user interface component that displays the current occupancy status of all parking spots in the Simulation. It updates live as the simulation runs, reflecting changes in parking spot status (free/occupied). It allows user interaction, such as zooming to a specific spot or manually overriding its status.

5.7.2 Simulation Controls

Simulation controls are interactive UI elements (sliders, buttons) that let users adjust simulation parameters. All controls update the simulation state instantly via JavaScript event listeners.

- **Vehicle Speed Slider:** Changes how fast vehicles move along their routes.
- **Point Cloud Size Slider:** Adjusts the visual size of point cloud points.
- **Tracking Toggle:** Enables/turns off camera following a vehicle or MLS.
- **Pause Point Clouds:** Freezes or resumes point cloud animation.
- **MLS Controls:** Switches between different point cloud datasets.

5.8 Navigation panel

The Navigation Panel allows users to select a target parking spot for navigation.

- Users enter a parking spot number.
- The system calculates and displays the estimated route, time, and distance to the spot.
- The camera can animate to the selected spot.
- Real-time status (e.g., “en Route”, “Parked”) is shown.

5.9 Bottom Status Bar

The Status Bar is a fixed UI element at the bottom of the screen showing real-time system metrics, providing immediate feedback on simulation performance and context.

Metrics being provided:

- FPS: Current frames per second.
- Render Time: Time to render a frame.
- Point Cloud Count: Number of point clouds loaded.
- Data Sources: Shows which raster and 3D model sources are active.
- Camera Position: Displays current camera coordinates and coordinate system.

5.10 Legend Panel

The Legend Panel provides a visual key for all symbols and colors used in the Simulation. It explains the meaning of icons (e.g., free/occupied parking, vehicles, buildings, point clouds). It helps users quickly interpret the 3D scene.

6. User Experience and Navigation

The simulation project is organized with a specific folder structure. Not all files need to be in the root directory. The main entry point is main/index.html, JavaScript modules, assets folder, point cloud data, and configuration files are organized in subdirectories (e.g., main/jssimulation/, main/assets/, main/PointCloudDatasets/). Simulation reads a configuration file (Config.txt) that specifies:
- The directories and filenames of the point cloud datasets are to be used.
- The number of frames and other relevant parameters.

The application can be opened using Vite (if configured). **Vite** is a modern frontend build tool and dev server. Executing npm run dev

- initiates a development server via Vite, which serves files over HTTP to resolve browser security
- constraints, enables real-time monitoring and automatic browser
- reloading upon file changes, and opens the Simulation at a local address using the system's default web browser. (e.g., http://localhost:5173/).

All logic runs entirely in the browser; no server-side code or database is required. The simulation is initialized by double-clicking the RUNSIMULATION.bat file. It takes approximately 15 seconds to load the server and a few more seconds to render the scene.

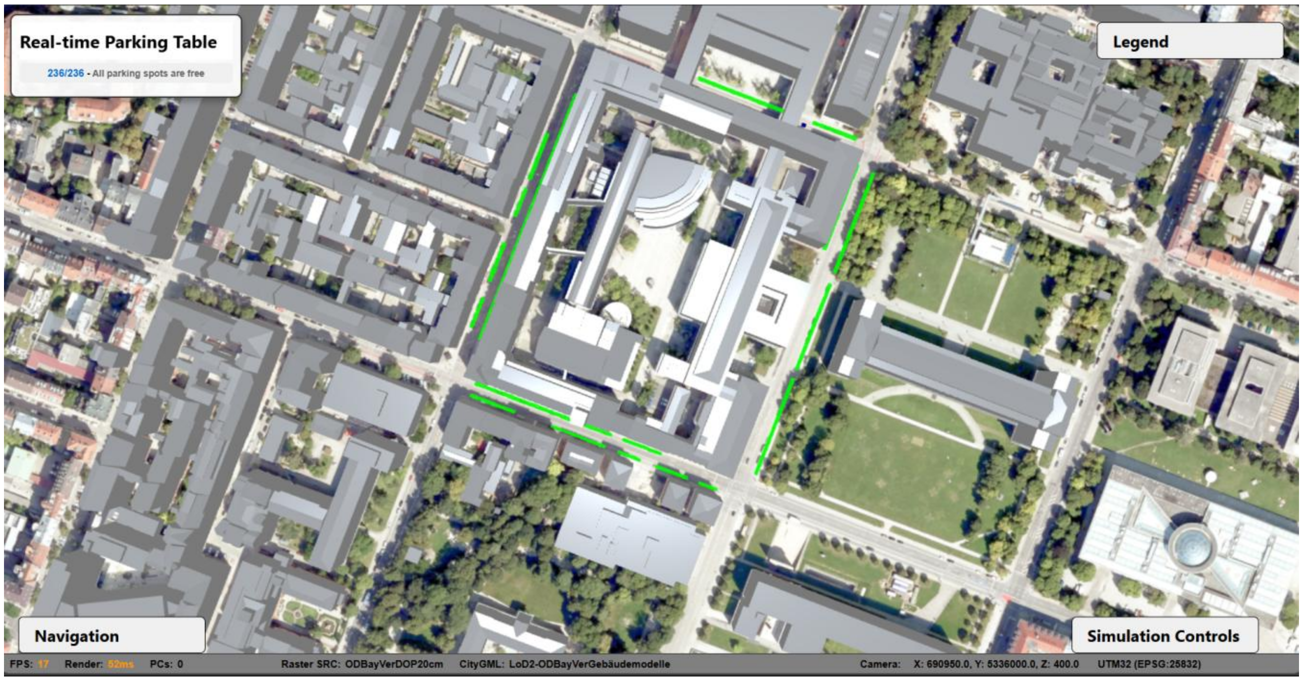
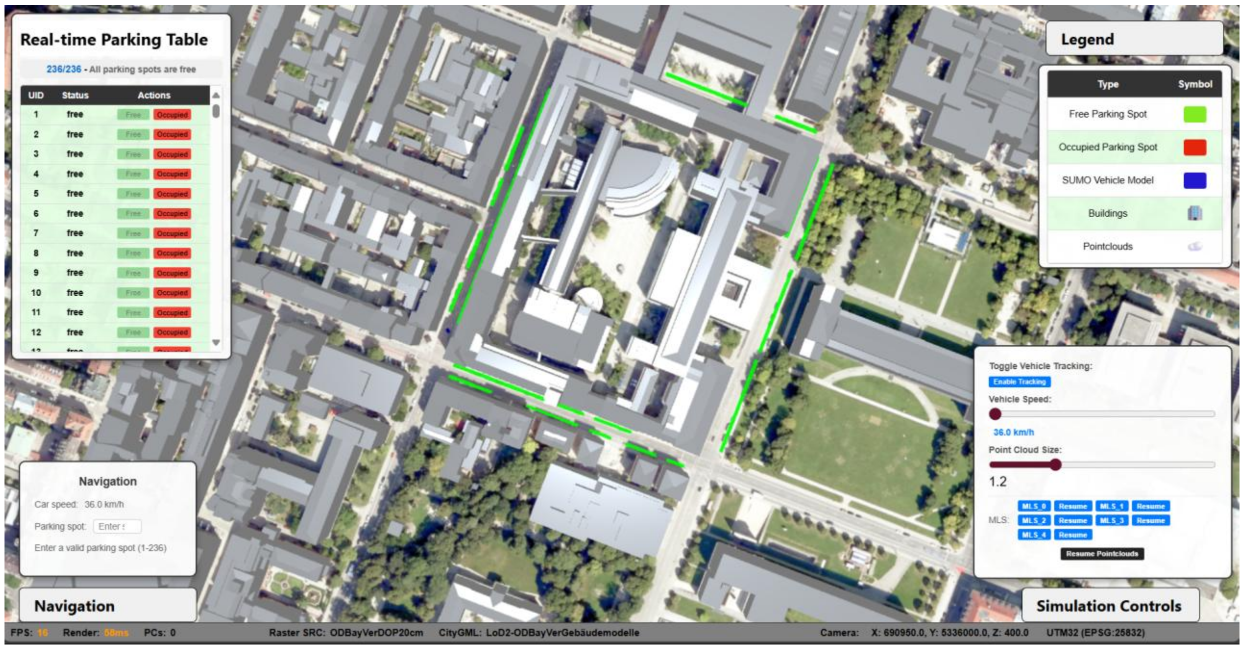

Real-time Parking Table
236/236 - All parking spots are free
Legend
Navigation
Simulation Controls
FPS: 17 Render: 52ms PCs: 0
Raster SRC: ODBayVerDOP20cm CityGML: LoD2-ODBayVerGebäudemodelle
Camera: X: 690950.0, Y: 5336000.0, Z: 400.0 UTM32 (EPSG:25832)

Figure 9 Initial Preview of Final Version

Starting with the simulation, 236 out of 236 parking spaces are empty since the simulation has not initialized its point clouds yet. To detect the state of parking spots, the user should click on the Simulation Controls panel and click on Resume Point Clouds. After which, we can observe the real-time parking status occupancy changes in its corresponding panel in the upper left corner by clicking on it for more information.



236/236 - All parking spots are free

UID	Status	Actions
1	free	Free Occupied
2	free	Free Occupied
3	free	Free Occupied
4	free	Free Occupied
5	free	Free Occupied
6	free	Free Occupied
7	free	Free Occupied
8	free	Free Occupied
9	free	Free Occupied
10	free	Free Occupied
11	free	Free Occupied
12	free	Free Occupied

Type	Symbol
Free Parking Spot	🟩
Occupied Parking Spot	🟥
SUMO Vehicle Model	🟦
Buildings	▨
Pointclouds	●

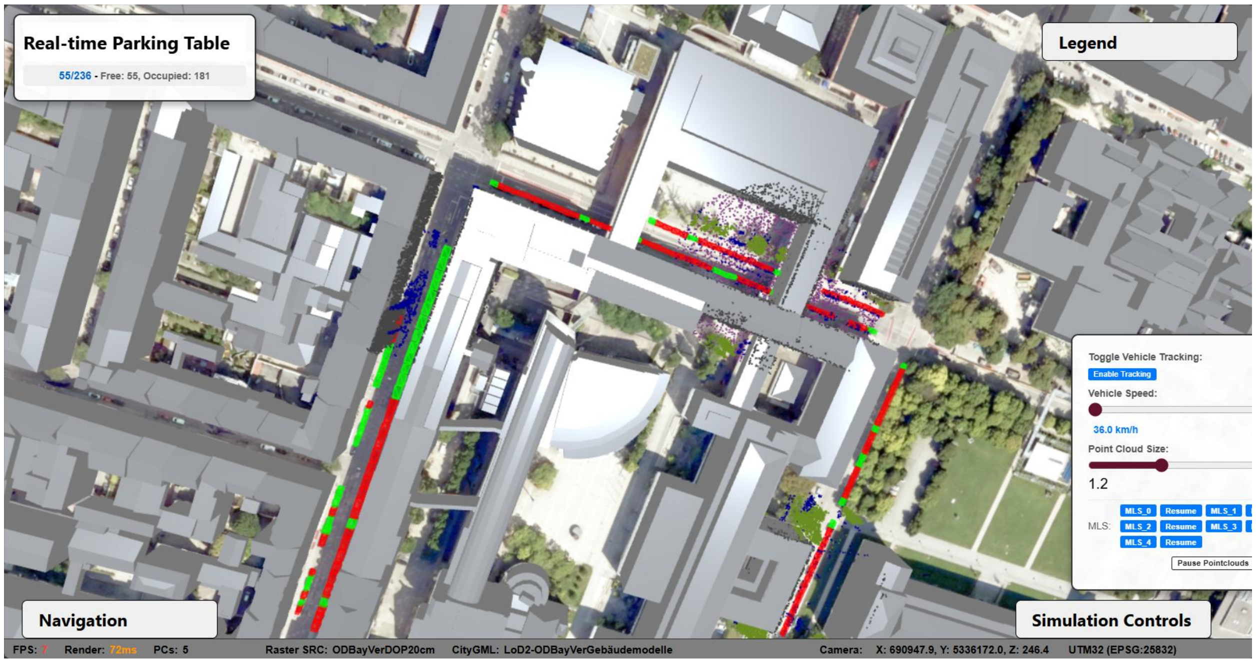
Navigation
Car speed: 36.0 km/h
Parking spot: Enter
Enter a valid parking spot (1-236)

Vehicle Speed: 36.0 km/h
Point Cloud Size: 1.2

FPS: 16 Render: 58ms PCs: 0 Raster SRC: ODBayVerDOP20cm CityGML: LoD2-ODBayVerGebäudemodelle Camera: X: 690950.0, Y: 5336000.0, Z: 400.0 UTM32 (EPSG:25832)

Figure 10 Final Version: All panels open

The user initializes the point cloud by clicking on the Resume point cloud from the simulation control panel, and then the actual parking detection starts.



55/236 - Free: 55, Occupied: 181

Vehicle Speed: 36.0 km/h
Point Cloud Size: 1.2

FPS: 7 Render: 72ms PCs: 5 Raster SRC: ODBayVerDOP20cm CityGML: LoD2-ODBayVerGebäudemodelle Camera: X: 690947.9, Y: 5336172.0, Z: 246.4 UTM32 (EPSG:25832)

Figure 11 Sensing of the environment

Initial parking spaces are 4 points from which a bounding box is created, since the parking spots were manually sampled without a bounding box of sorts.

Point clouds count the number of points that fall within the cube defined by the bounding box of a parking space; currently, this threshold is set at 10 points.Three.js has its own visualisation methods for points and geometry, whose size we can adjust in the simulation control panel.

The logic has been tested multiple times by lowering the threshold, resulting in more parking spaces being occupied and vice versa. But also, with visual verification within Cloud Compare software, comparing the results of the main Point cloud dataset with the Simulation results. In Figures 10,11,12,13, and 13, the main PCD file is loaded in Cloud Compare, where one can see parked vehicles.

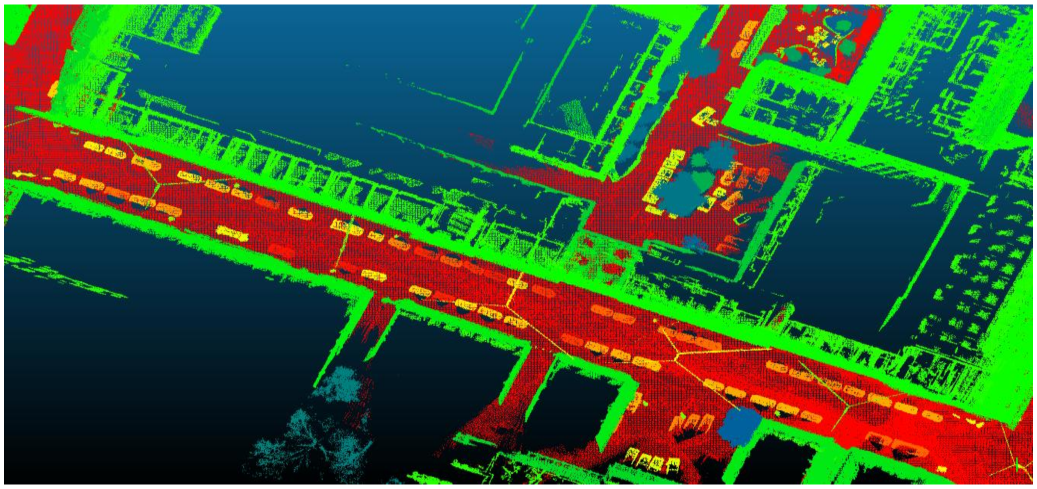
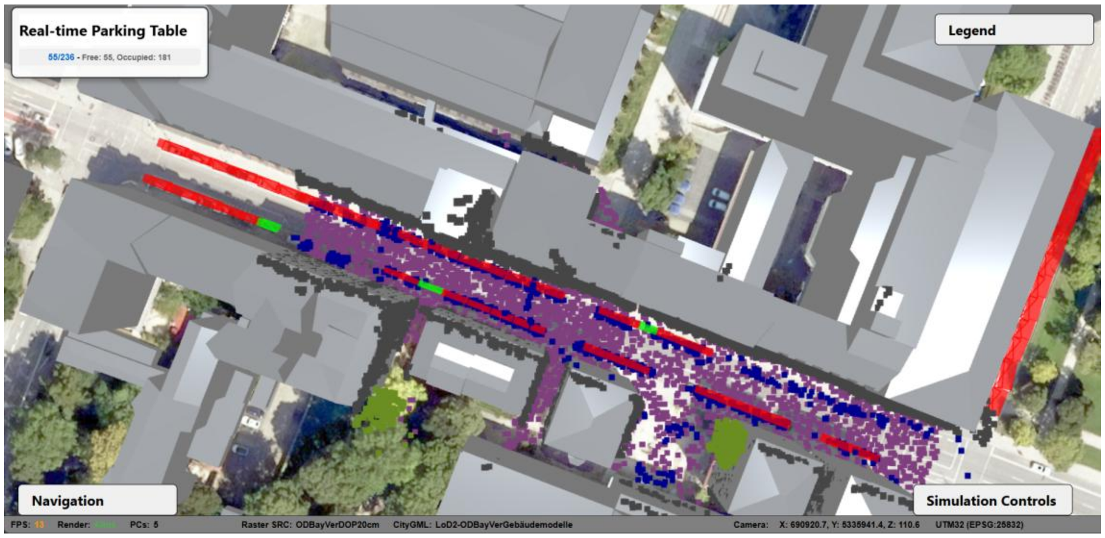


Figure 12 CC Main PCD South Side



55/236 - Free: 55, Occupied: 181

FPS: 13 Render: PCs: 5 Raster SRC: ODBayVerDOP20cm CityGML: LoD2-ODBayVerGebäudemodelle Camera: X: 690920.7, Y: 5335941.4, Z: 110.6 UTM32 (EPSG:25832)

Figure 13 A closer look at the sensed parking spaces on the South side

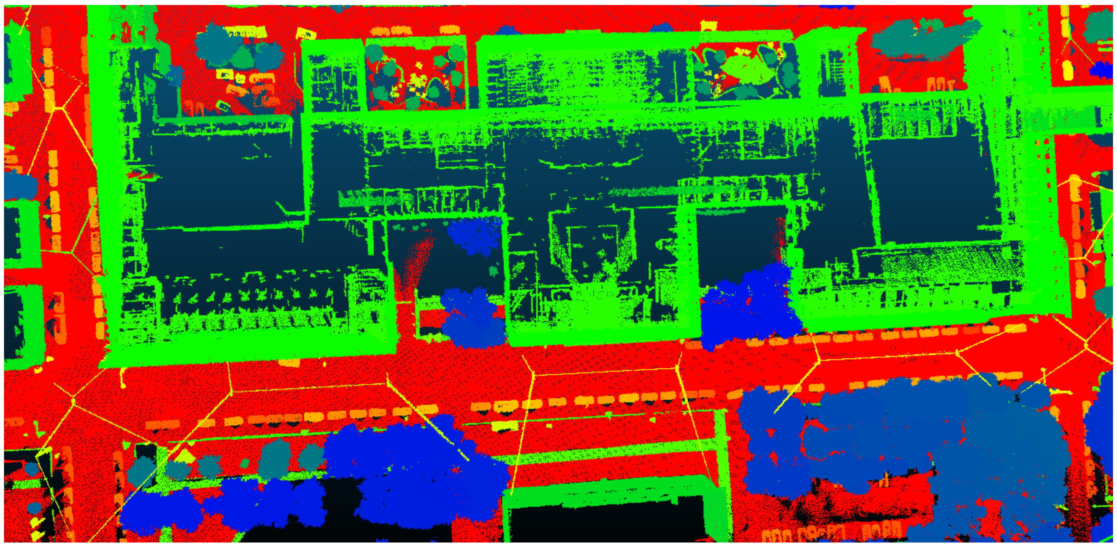
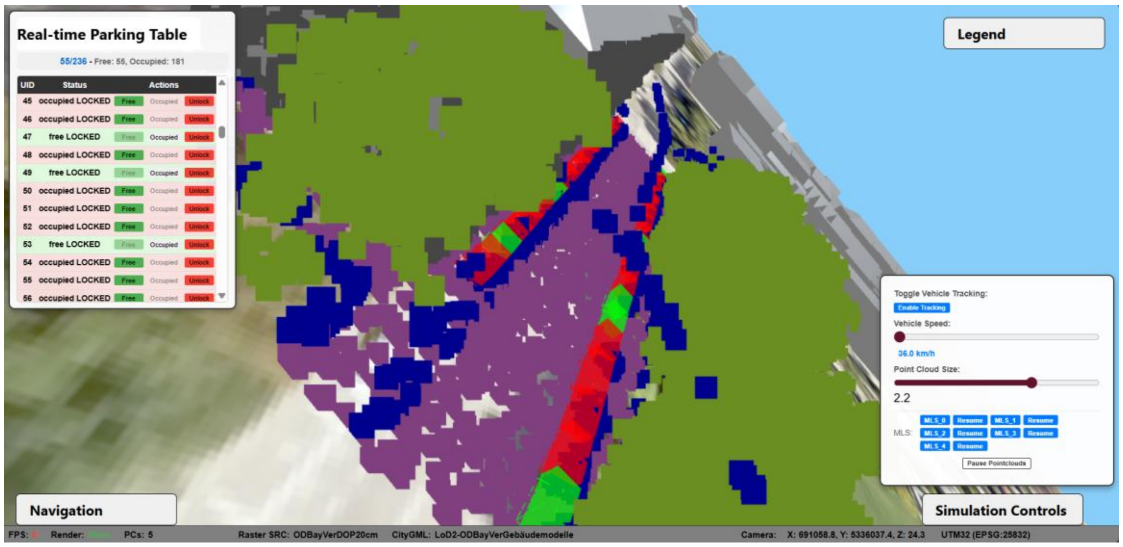


Figure 14 CC Main PCD East Side



Real-time Parking Table

55/236 - Free: 55, Occupied: 181

UID	Status	Actions		
45	occupied LOCKED	Free	Occupied	Unlock
46	occupied LOCKED	Free	Occupied	Unlock
47	free LOCKED	Free	Occupied	Unlock
48	occupied LOCKED	Free	Occupied	Unlock
49	free LOCKED	Free	Occupied	Unlock
50	occupied LOCKED	Free	Occupied	Unlock
51	occupied LOCKED	Free	Occupied	Unlock
52	occupied LOCKED	Free	Occupied	Unlock
53	free LOCKED	Free	Occupied	Unlock
54	occupied LOCKED	Free	Occupied	Unlock
55	occupied LOCKED	Free	Occupied	Unlock
56	occupied LOCKED	Free	Occupied	Unlock

FPS: Render: PCs: 5 Raster SRC: ODBayVerDOP20cm CityGML: LoD2-ODBayVerGebäudemodelle Camera: X: 691058.8, Y: 5336037.4, Z: 24.3 UTM32 (EPSG:25832)

Figure 15 A closer look at the sensed parking spaces on the East side

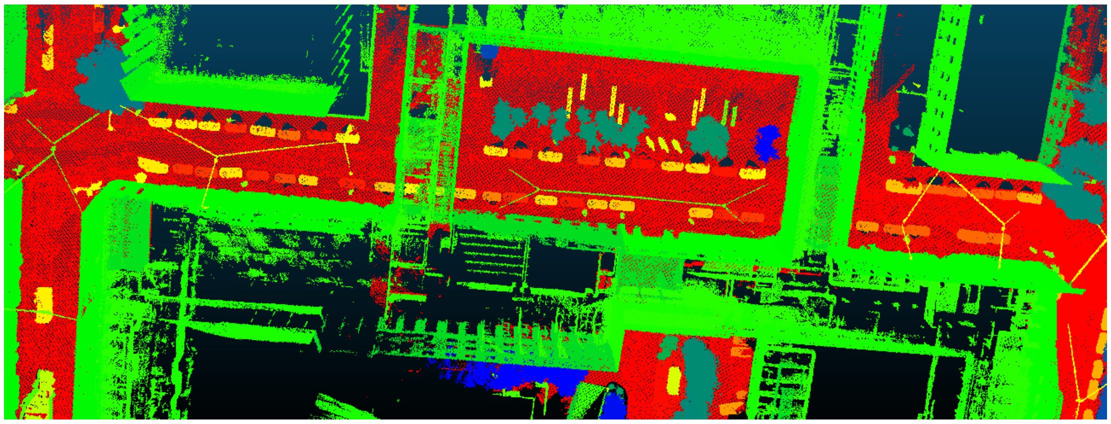


Figure 16 CC Main PCD North Side

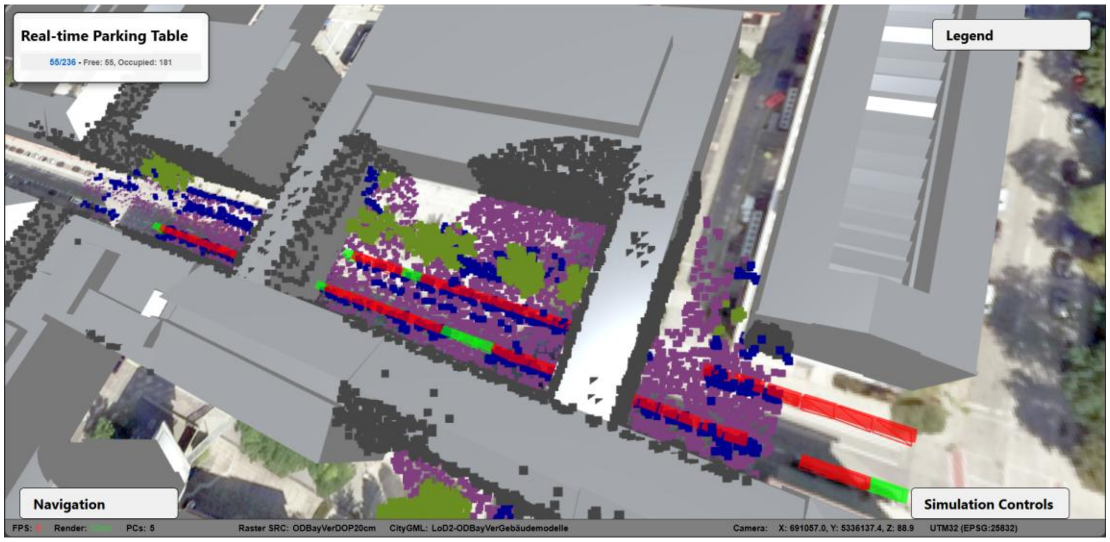

Real-time Parking Table
55/236 - Free: 55, Occupied: 181
Legend
Navigation
Simulation Controls
FPS:
Render:
PCs: 5
Raster SRC: ODBayVerDOP20cm
CityGML: LoD2-ODBayVerGebäudemodelle
Camera: X: 691057.0, Y: 5336137.4, Z: 88.9
UTM32 (EPSG:25832)

Figure 17 A closer look at the sensed parking spaces on the North side

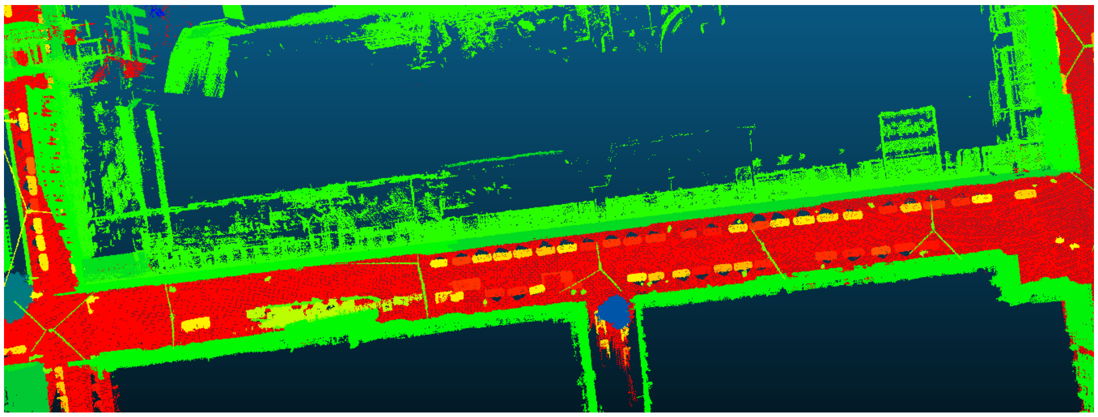
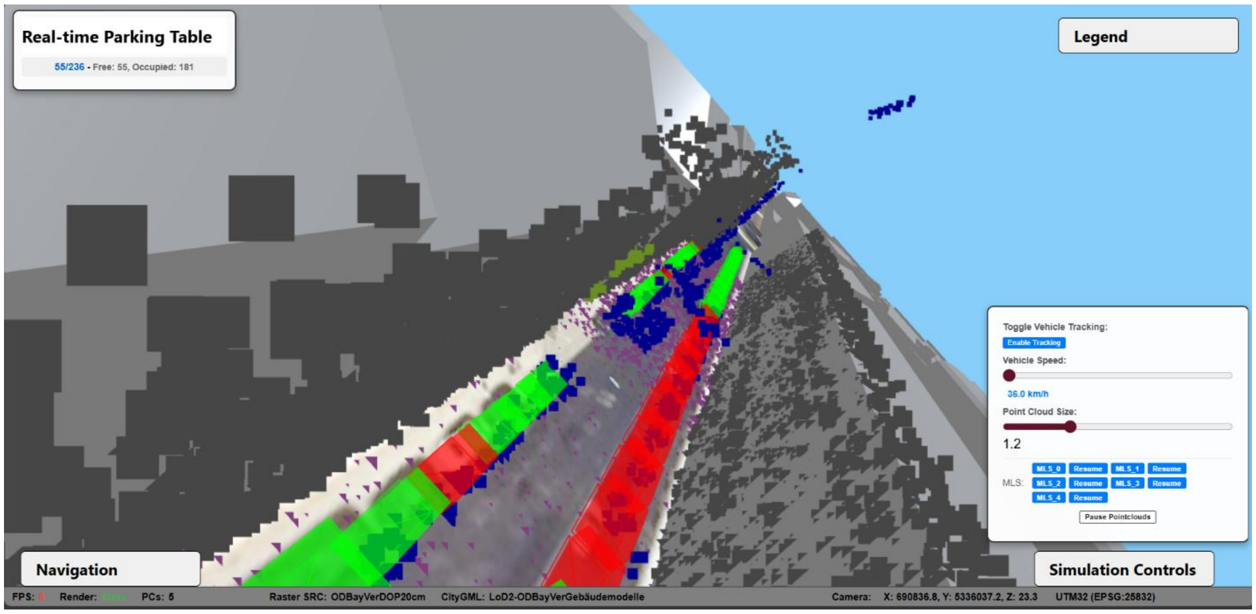


Figure 18 CC Main PCD West Side


Real-time Parking Table
55/236 - Free: 55, Occupied: 181

Toggle Vehicle Tracking:
Vehicle Speed: 36.0 km/h
Point Cloud Size: 1.2

FPS: 8 Render: PCs: 5 Raster SRC: ODBayVerDOP20cm CityGML: LoD2-ODBayVerGebäudemodelle Camera: X: 690836.8, Y: 5336037.2, Z: 23.3 UTM32 (EPSG:25832)

Figure 19 A closer look at the sensed parking spaces on the West side

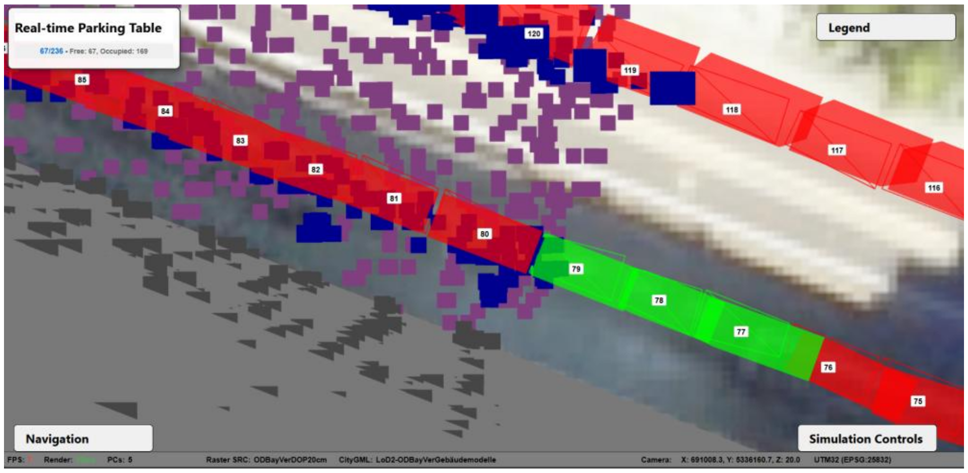
Once the first loop has been made, the second loop starts with 55 free spaces and finishes with the same number of free spaces. Therefore, the user is advised to reload the simulation to monitor change detection via some tracking options provided in the Toggle Vehicle Tracking from the Simulation Controls Panel, and/or try visualising the scene with different point sizes.

Afterward, the user is advised to pause and remove the point cloud from the scene by clicking on pause and MLS_ii so that the user can implement changes in the current parking occupancy state. See the next chapter.

6.1 Manual Status Override

From here, the user can manually make some parking spaces occupied or free by manipulating the real-time parking table panel. By double-clicking on the parking UID or parking spot directly in the scene, the camera automatically glides above (z=20m) the center parking spot. Also, the parking spots' UIDs show only if the camera is below or at 20m.

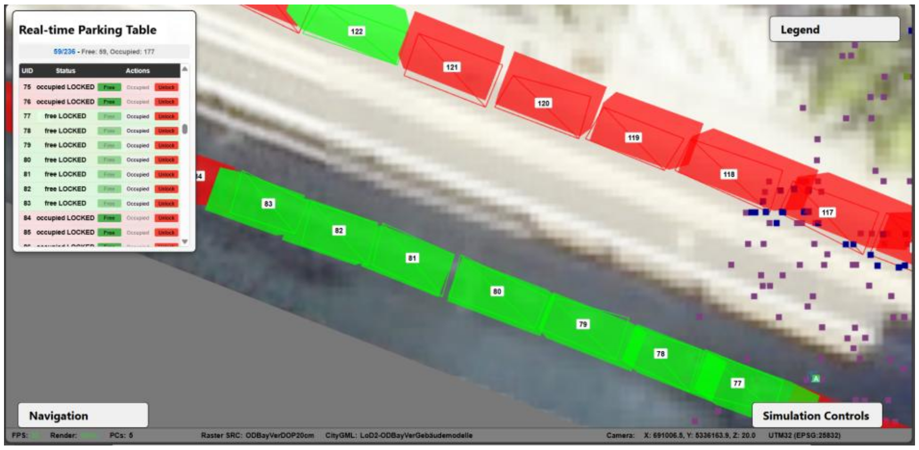
When the point clouds scan through the area, a parking spot is manually turned free, and the spots are returned to occupied. As shown in the figures below:



Real-time Parking Table

67/236 - Free: 67, Occupied: 169

Figure 20 Sensed Parked Vehicles



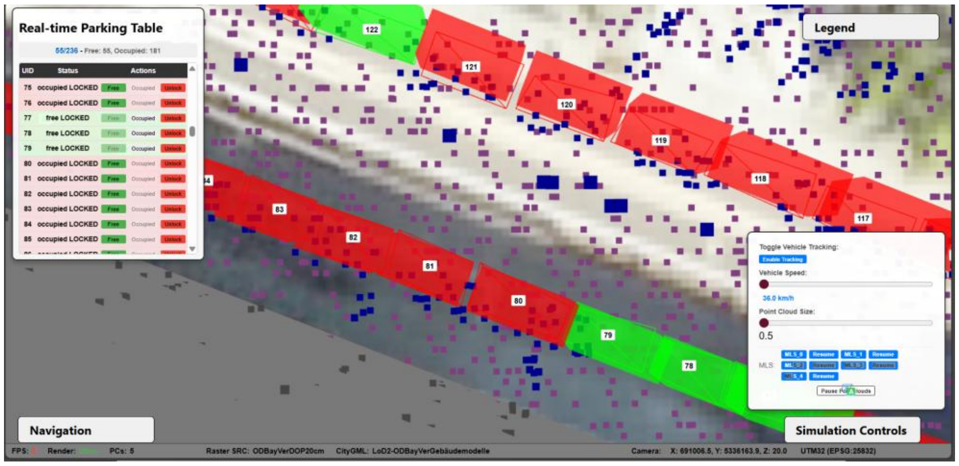
Real-time Parking Table

59/236 - Free: 59, Occupied: 177

UID	Status	Actions		
75	occupied LOCKED	Free	Occupied	Unlock
76	occupied LOCKED	Free	Occupied	Unlock
77	free LOCKED	Free	Occupied	Unlock
78	free LOCKED	Free	Occupied	Unlock
79	free LOCKED	Free	Occupied	Unlock
80	free LOCKED	Free	Occupied	Unlock
81	free LOCKED	Free	Occupied	Unlock
82	free LOCKED	Free	Occupied	Unlock
83	free LOCKED	Free	Occupied	Unlock
84	occupied LOCKED	Free	Occupied	Unlock
85	occupied LOCKED	Free	Occupied	Unlock

Legend
122
121
120
119
118
117
83
82
81
80
79
78
77
Navigation
Simulation Controls
FPS:
Render:
PCs: 5
Raster SRC: ODBayVerDOP20cm
CityGML: LoD2-ODBayVerGebäudemodelle
Camera: X: 691006.5, Y: 5336163.9, Z: 20.0
UTM32 (EPSG:25832)

Figure 21 Manually changed Parking Status.



Real-time Parking Table

55/236 - Free: 55, Occupied: 181

UID	Status	Actions		
75	occupied LOCKED	Free	Occupied	Unlock
76	occupied LOCKED	Free	Occupied	Unlock
77	free LOCKED	Free	Occupied	Unlock
78	free LOCKED	Free	Occupied	Unlock
79	free LOCKED	Free	Occupied	Unlock
80	occupied LOCKED	Free	Occupied	Unlock
81	occupied LOCKED	Free	Occupied	Unlock
82	occupied LOCKED	Free	Occupied	Unlock
83	occupied LOCKED	Free	Occupied	Unlock
84	occupied LOCKED	Free	Occupied	Unlock
85	occupied LOCKED	Free	Occupied	Unlock

Toggle Vehicle Tracking: Enable Tracking

Vehicle Speed: 36.0 km/h

Point Cloud Size: 0.5

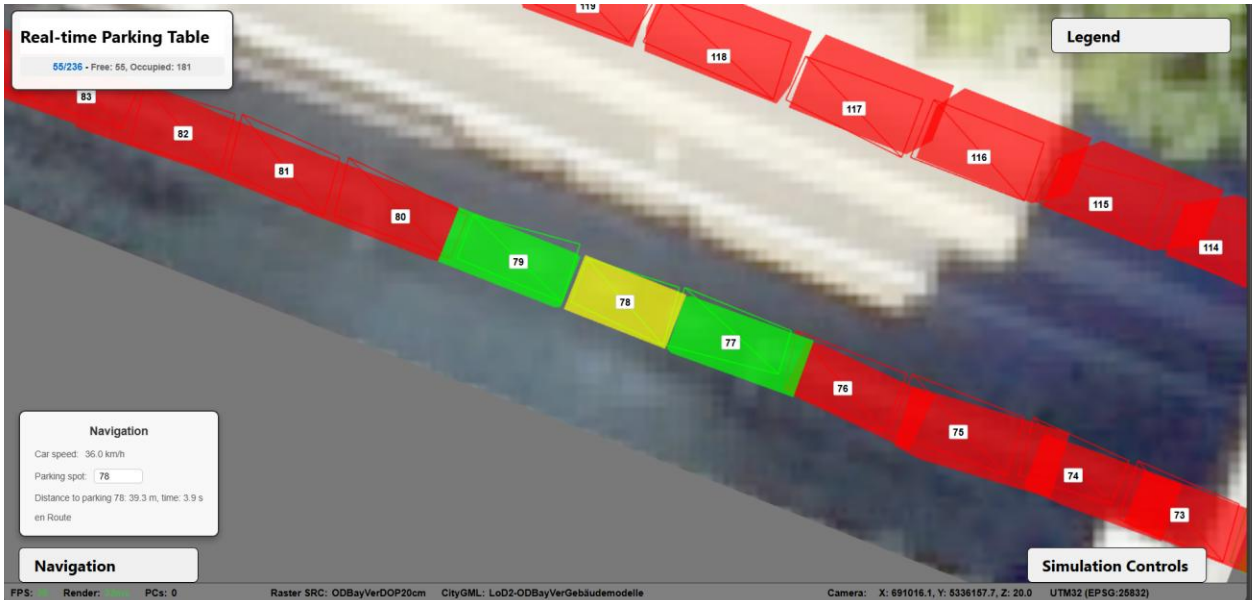
Figure 22 Next Point Cloud Iteration as a new detection

6.2 Navigation

The logic behind this type of simulation comes from the fact that the provided point cloud data sets all come from one big point cloud data set, meaning only one time epoch and no change detection. Therefore, if the user were to import a new directory, for example, of vehicle 01, which currently has 2000 PCD files with scans of exactly one circle around the TUM, if the user were to scan again, they should place this scan in the same folder with proper number indexing. As the simulation would load one by one, making the visualization. The user could observe one final parking result at the 2000th PCD file load, and afterward, all the new changes would be made by the new point cloud, visualizing change detection. Also, this is not constrained to just one vehicle as the simulation runs simultaneously all 5-vehicle folders, unless manually stopped from the simulation control panel.

This project contains PCD files from one epoch, hence the navigation panel for visualising **change detection**. By clicking on this panel, it displayes the curent speed of the vehicle, there is a slot for enntering a target parking spot by its uid, distance and time needed for the vehicle to reach this parking spot and the status of the navigation with fiew options: En route meaning the vehicle is on its way, Parking Ocuppied meaning the vehicle came to the parking spot and saw that its already occupied and the third option Parked Succesfully meaning the car went to the parking spot read from the real time parking table that its free and parked, instantly changing the status on scene and in the table. If the car is parked successfully after 3 seconds of not moving (parking), the vehicle respawns at its starting point; otherwise, the vehicle does not stop and continues in a closed loop of its trajectory.

Another way of visualising change detection in this simulation is by Navigation. In Figure 23, a zoom in on a parking section is shown after the sensing is over and Navigation is turned on.


Real-time Parking Table
55/236 - Free: 55, Occupied: 181

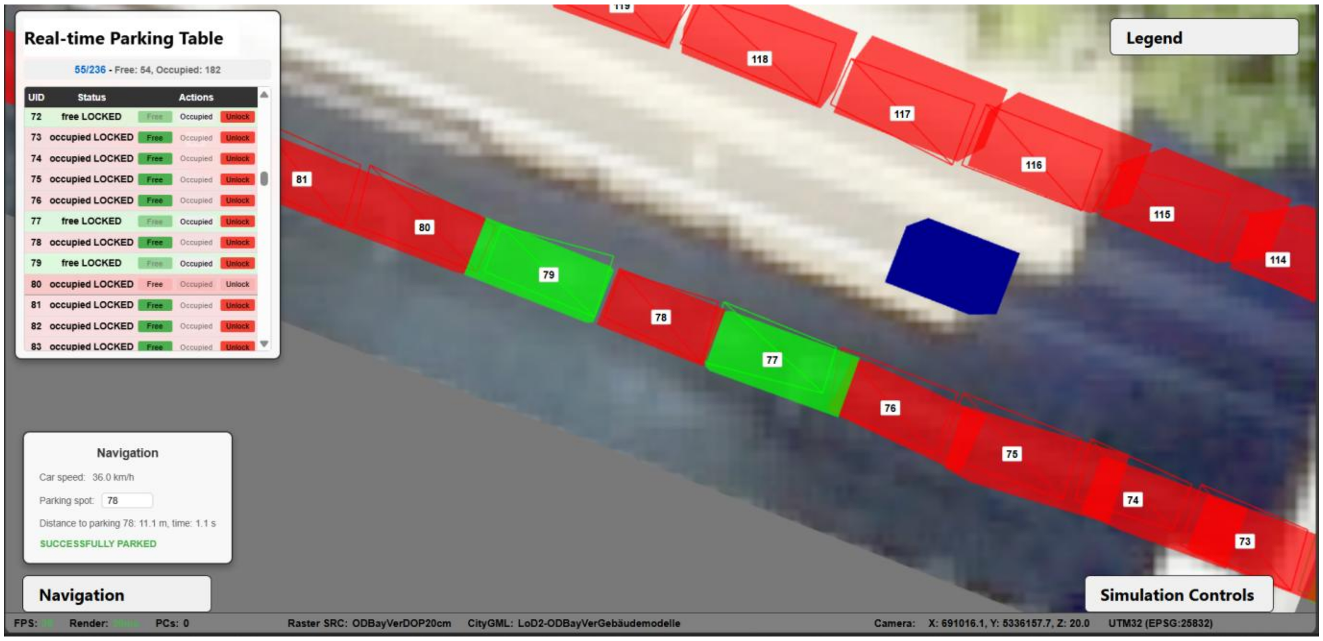
Navigation
Car speed: 36.0 km/h
Parking spot: 78
Distance to parking 78: 39.3 m, time: 3.9 s
en Route

FPS: Render: PCs: 0 Raster SRC: ODBayVerDOP20cm CityGML: LoD2-ODBayVerGebäudemodelle Camera: X: 691016.1, Y: 5336157.7, Z: 20.0 UTM32 (EPSG:25832)

Figure 23 Example of Change Detection nr1

Therefore, the user can select the navigation to the parking spot as shown in Figure 23. The vehicle is en route, the parking space turns yellow, and its status changes to locked in the real-time parking table.

When the vehicle senses the locked UID within a 10m radius of itself, it reads the UID's status and decides whether to park or not. Since the parking space was free, the vehicle parks and starts over from the initial starting point of its OSM route, as shown in Figure 24.



Real-time Parking Table

55/236 - Free: 54, Occupied: 182

UID	Status	Actions		
72	free LOCKED	Free	Occupied	Unlock
73	occupied LOCKED	Free	Occupied	Unlock
74	occupied LOCKED	Free	Occupied	Unlock
75	occupied LOCKED	Free	Occupied	Unlock
76	occupied LOCKED	Free	Occupied	Unlock
77	free LOCKED	Free	Occupied	Unlock
78	occupied LOCKED	Free	Occupied	Unlock
79	free LOCKED	Free	Occupied	Unlock
80	occupied LOCKED	Free	Occupied	Unlock
81	occupied LOCKED	Free	Occupied	Unlock
82	occupied LOCKED	Free	Occupied	Unlock
83	occupied LOCKED	Free	Occupied	Unlock

Navigation

Car speed: 36.0 km/h

Parking spot: 78

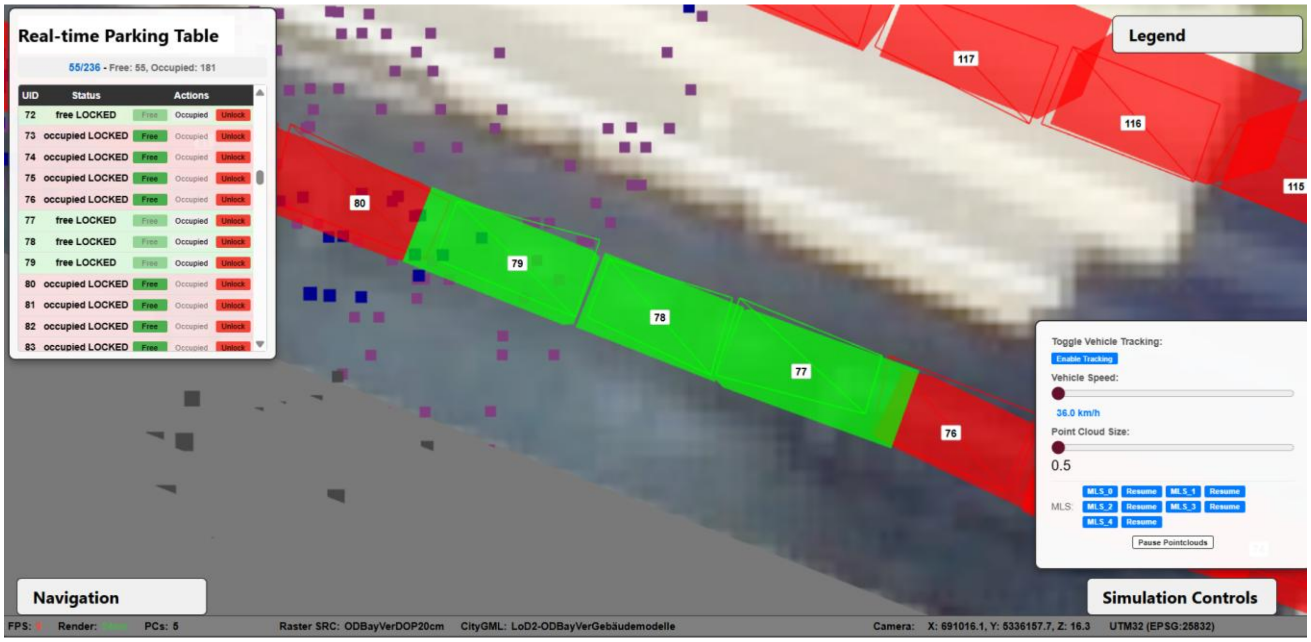
Distance to parking 78: 11.1 m, time: 1.1 s

SUCCESSFULLY PARKED

FPS: Render: PCs: 0 Raster SRC: ODBayVerDOP20cm CityGML: LoD2-ODBayVerGebäudemodelle Camera: X: 691016.1, Y: 5336157.7, Z: 20.0 UTM32 (EPSG:25832)

Figure 24 Example of Change Detection nr2

Now, the total number of parking spaces is down to 54, and the user can set the point cloud detection in motion again. Users must now click unlock on the corresponding parking spot so that the point clouds start considering this space again, not simply avoiding it as they do when it's reserved. Figure 25 shows that the parking space with UID 78 has been detected as empty and has returned to free status.



Real-time Parking Table

55/236 - Free: 55, Occupied: 181

UID	Status	Actions		
72	free LOCKED	Free	Occupied	Unlock
73	occupied LOCKED	Free	Occupied	Unlock
74	occupied LOCKED	Free	Occupied	Unlock
75	occupied LOCKED	Free	Occupied	Unlock
76	occupied LOCKED	Free	Occupied	Unlock
77	free LOCKED	Free	Occupied	Unlock
78	free LOCKED	Free	Occupied	Unlock
79	free LOCKED	Free	Occupied	Unlock
80	occupied LOCKED	Free	Occupied	Unlock
81	occupied LOCKED	Free	Occupied	Unlock
82	occupied LOCKED	Free	Occupied	Unlock
83	occupied LOCKED	Free	Occupied	Unlock

Toggle Vehicle Tracking:

Vehicle Speed: 36.0 km/h

Point Cloud Size: 0.5

MLS:

FPS: Render: PCs: 5 Raster SRC: ODBayVerDOP20cm CityGML: LoD2-ODBayVerGebäudemodelle Camera: X: 691016.1, Y: 5336157.7, Z: 16.3 UTM32 (EPSG:25832)

Figure 25 Example of Change Detection nr3

7. Typical Use Cases

- Research and Development

Testing and validating parking detection algorithms using real or synthetic point cloud data.

- Simulation and Visualization

Demonstrating the impact of vehicle movement and parking occupancy in a realistic 3D environment.

- Performance Analysis

Evaluating the efficiency and accuracy of detection and tracking methods under various conditions.

8. Challenges and Limitations

The project encountered several significant challenges, both technical and structural. The first issue was the lack of modularity and documentation in the original Version 0 simulation, which made it challenging to trace logic, modify components, or implement new features. Additionally, the original simulation consumed excessive resources and lacked scalability.

Midway through the project, it became evident that continuing development on the initial codebase would be inefficient and would not support the intended functionality. As a result, the project was restructured from scratch, adopting a modular ES6 architecture, optimized point cloud handling, and a more transparent data processing workflow.

A major challenge was efficiently rendering large point cloud datasets in the browser. This was addressed by developing custom loaders and optimizing the rendering pipeline, allowing the simulation to handle larger datasets than before. Performance bottlenecks were identified and mitigated using a dedicated performance monitoring module, which guided further optimizations. The lack of separation between data processing, UI controls, and simulation logic in the original code made adding new features such as parking management and performance monitoring difficult. By refactoring into ES6 modules, these features were implemented with minimal code duplication and improved maintainability.

Another challenge was providing a user-friendly and interactive simulation experience. This was achieved by introducing dedicated modules for UI controls and legend rendering, making the interface more intuitive and accessible. Despite these improvements, limitations remain. The entire simulation runs in the browser, which imposes strict constraints on memory usage (RAM and GPU). Point cloud sizes, texture quality, and the number of rendered objects must be managed carefully to avoid performance degradation or crashes. There is also no backend integration, which limits real-time data sourcing, multi-user functionality, and historical analysis.

Nevertheless, the new implementation allows for future optimization and functional expansion, including backend integration, better memory handling, and broader geographic coverage.

9. Summary

This project is an entirely client-side, modular, and interactive 3D simulation tool for parking detection and vehicle tracking. It combines point cloud analytics, SUMO-based traffic simulation, and advanced visualization, all running directly in the browser. No backend or database is required, as all logic executes in the browser environment. The Simulation provides a functional prototype for analyzing parking space occupancy and vehicle movement using real-time or simulated input.

However, as a browser-based solution, it is constrained by the memory limits of the browser environment (WebGL) and the available GPU resources. All point cloud data and textures must fit into system RAM and GPU memory. Browsers typically impose memory limits per tab (often between 2–4 GB), and WebGL has hardware-dependent constraints like the number of vertices, size of textures, and objects rendered simultaneously. Once those limits are reached, performance degradation or browser crashes may occur.

These limitations highlight the importance of optimizing memory usage by loading only required assets at a time, disposing of unused objects, downsampling point clouds, and using lower-resolution textures. Despite those constraints, the architecture lays a strong foundation for scalable and efficient 3D parking simulations.

10. Outlook

This work proposes a solid foundation for a crowdsensing-based system that integrates data from various sensors (vehicle-mounted, stationary, or user-reported) to provide real-time updates on parking occupancy.

The next logical step would be to expand the model. Currently, it includes only the TUM building in Arcisstraße. Expanding to a broader area, such as the inner part of Munich, would require new ground surfaces, CityGML models, point cloud data, and updated parking files. Ground surfaces and CityGML models can be collected from the OpenData Bayern Measurement website. Extracting parking spaces would need to be at least semi-optimized for larger zones and more accurate spot geometry. Point cloud data would also need to be correctly georeferenced.

To support further expansion, it will be essential to load only the required point clouds and textures at a given time and to properly dispose of unused objects from the scene and memory. Downsampling point clouds, using lower-resolution textures, and applying efficient memory management will be key to scaling the Simulation and maintaining stable performance.

11. Future Development and Application Potential

Future goals and directions include establishing a dedicated server infrastructure with sufficient GPU resources to support more complex simulations and real-time data processing. A backend system would also be necessary to ensure scalability and enable integration with external data sources. Introducing a time variable opening options for time series, enhancing traffic analysis and urban development. Consider using a digital terrain model for ground representation. Available for Muich on the tum2twin website. Adding material textures to its facets for better visualization. Finding new ways to calculate the accuracy of detection.

Once these foundations are in place, expanding the sensor network, such as increasing the number of LiDAR units or incorporating other data collection methods, could significantly enhance coverage and accuracy.
Eventually, the system could be extended with a navigation application or an integration with existing platforms like Google Maps, allowing users to receive precise, real-time directions to available parking spaces within the covered area.

The ultimate idea is to enhance existing navigation applications by suggesting available parking spots near the target location and automatically rerouting users to the closest free space.
Challenges naturally arise. For instance, multiple users driving to the same area at the same time (e.g., coworkers arriving at 7 a.m.) might all be directed to the same spot. In such cases, the one who comes first gets the parking space, while others are rerouted to the next closest options. This can lead to increased traffic and the search radius expanding around the initial destination.
To manage such situations, the system could include user registration, allowing features such as parking spot reservation, temporarily hiding available spots from other users, and predicting user reliability based on their app usage history. Still, the risk remains that an unregistered driver might occupy a reserved spot. If that happens, the system will notify the user instantly and recalculate a new route.

Currently there is a lot of already developed systems like google maps which can display in some cities parking availability as easy, medium or limited, or Parkopedia [10] or SpotHero, Best parking, Park Mobile, but all of them are based on parking data in general and connected with public and private garages and parking zones, these applications are the middle man between the user and parking allowing users to see all their options and pricing before parking at a location. There is the Bosch & Siemens Smart Parking Project from Germany, which includes sensors on streets sensing real-time data, transferring it to their database, and updating the application in real time, which best reflects the ideas in this project. None of the startups use lidar as the source of data collection; they mostly use radar sensors, cameras, and advanced machine learning algorithms for detecting parking spaces. As hardware and technology become more accessible and affordable, so will our everyday parking struggles one day be reduced when we find the perfect solution.

12. References

[1] Eclipse Foundation. (2024). *SUMO - Simulation of Urban Mobility*. Retrieved from https://www.eclipse.dev/sumo

[2] Three.js Authors. (2024). *Three.js Documentation*. Retrieved from https://threejs.org/docs

[3] Bayerische Vermessungsverwaltung. (2024). *3D-Gebäudemodelle LoD2* [Open Data]. Retrieved from https://geodaten.bayern.de

[4] Bayerische Vermessungsverwaltung. (2024). *Digitales Orthophoto (DOP) RGB 20 cm* [Open Data]. Retrieved from https://geodaten.bayern.de

[5] OpenStreetMap Contributors. (2024). *OpenStreetMap* [Data set]. Retrieved from https://www.openstreetmap.org

[6] CloudCompare Development Team. (2024). *CloudCompare (Version 2.12)* [GPL Software]. Retrieved from https://www.danielgm.net/cc/

[7] Safe Software Inc. (2024). *FME Desktop (Feature Manipulation Engine)* [Software]. Retrieved from https://www.safe.com/fme/

[8] Technische Universität München – TUM2TWIN Project Team. (2024). *TUM2TWIN Point Cloud Dataset*. [Internal/unpublished dataset shared for academic use].

[9] Bosch GmbH. (2023). *Smart Parking Solutions by Bosch*. Retrieved from https://www.bosch-mobility.com

[10] Parkopedia Ltd. (2024). *Parkopedia - Parking Availability App*. Retrieved from https://www.parkopedia.com

[11] Welch, G., & Bishop, G. (1995). *An Introduction to the Kalman Filter*. University of North Carolina at Chapel Hill. Retrieved from https://www.cs.unc.edu/~welch/media/pdf/kalman_intro.pdf