Demonstrator

User Manual



(V0.1)

In collaboration with David Crampen and Marcel Hein

Table of Contents

1 Introduction 1

2 Functions 2

2.1 CPM 2

2.2 WIM 2

2.3 PRS 3

2.4 Gantry 4

2.5 ALE 4

3 Installation and Commissioning 5

3.1 Unreal Engine 5

3.2 Python 6

3.3 Visual Studio 6

3.4 Houdini Engine 6

3.5 Project Creation and Setup 6

4 Operation 9

4.1 CPM 10

4.2 WIM 11

4.3 Gantry 12

4.4 PRS 13

4.5 Pressure Mat 14

4.6 Aggregation Mat 14

4.7 Python Scripts for Sending Data 14

5 Settings 17

5.1 CPM 18

5.2 WIM 18

5.3 Gantry 19

5.4 PRS 19

5.5 Pressure Mat 21

5.6 Aggregation Mat 22

6 OSC Ports and Data Structure 23

7 Notes 25

List of Figures

Fig. 2.1: Visualisation of CPM data using OBU (vehicle model), OSN (blue box), RSU (orange box), and V2I (pink area) 2
Fig. 2.2: Visualisation of WIM data on the corresponding measurement fields 3
Fig. 2.3: Visualisation of PRS data for a vehicle 4
Fig. 2.4: Visualisation of a gantry with freely selectable illuminated and traffic signs 4
Fig. 2.5: Visualisation of a predefined ALE scenario 5
Fig. 3.1: Unreal Engine project creation window 7
Fig. 3.2: Creating the Plugins folder for Unreal Engine 7
Fig. 3.3: Opening the Unreal Engine plugin manager 8
Fig. 3.4: Enabling Show Plugin Content in Unreal Engine 8
Fig. 3.5: Unreal Engine GameMode settings in World Settings 9
Fig. 4.1: Demonstrator information panel 10
Fig. 4.2: Demonstrator options panel 10
Fig. 4.3: Structure of a JSON file for the OBU category in CPM 11
Fig. 4.4: Structure of a JSON file for the gantry 12
Fig. 4.5: Traffic signs from left to right: 276-black, 274-80-black, 282-black 13
Fig. 4.6: Structure of a JSON file for the PRS 13
Fig. 4.7: Python interface for operating the CPM, PRS, and pressure mat functions 15
Fig. 4.8: Python interface for operating the gantry 16
Fig. 4.9: Python interface for operating the WIM function 17
Fig. 5.1: Unreal Engine CPM settings 18
Fig. 5.2: Unreal Engine WIM measurement base settings 19
Fig. 5.3: Unreal Engine WIM measurement field settings 19
Fig. 5.4: Unreal Engine PRS settings (1) 20
Fig. 5.5: Unreal Engine PRS settings (2) 20
Fig. 5.6: Unreal Engine PRS settings (3) 21
Fig. 5.7: Unreal Engine pressure mat settings 22
Fig. 5.8: Unreal Engine Aggregation Mat settings (1) 22
Fig. 5.9: Unreal Engine Aggregation Mat settings (2) 23
Fig. 6.1: Python example for sending data via OSC 24

1 Introduction

This manual provides an overview of how to set up and operate the Demonstrator.

The Demonstrator was developed as a plugin for Unreal Engine. It brings together several independent functions and subprojects in a shared environment, allowing different components to be displayed, controlled, and combined centrally.

The aim is to make these components available within a shared Demonstrator environment rather than using them separately. This allows scenarios to be prepared, visualised, and controlled during operation.

This documentation describes the basic setup of the Demonstrator, its key settings, and how to operate its individual functions. It also explains the required project and system settings and provides relevant information on ports, media paths, cameras, and other configuration options.

This manual is intended for anyone who wants to use, set up, extend, or prepare the Demonstrator for presentations. It provides support for initial setup as well as subsequent use and troubleshooting.

2 Functions

The Demonstrator comprises several functional areas brought together within the Unreal Engine plugin. The individual functions represent different subprojects and can be used separately or together, depending on the application.

The integrated functional areas include CPM, WIM as a Weigh-in-Motion system, PRS as a Pavement Response System, and the gantry function. The Demonstrator also includes ALE, which represents loads and their effects on the road within the Demonstrator environment. Local aggregation of load data is also supported, allowing load information within specific areas to be combined and processed further.

2.1 CPM

CPM stands for Cyber-Physical Mobility Lab. In the Demonstrator, it refers to the functional area for representing vehicle positions within the Unreal Engine environment. This function allows external vehicle data to be transmitted to the Demonstrator and assigned to the corresponding vehicles. Real movement data can therefore be displayed in the virtual environment and used for different scenarios.

The CPM function supports several operating modes, including OBU, OSN, RSU, SSL, and V2I. These modes represent different applications within the CPM Lab. Data can be processed and visualised differently in the Demonstrator depending on the mode.

In addition to position and rotation data, uncertainty information is processed and displayed. This makes it possible to represent not only a vehicle's position but also its orientation and any potential deviation or inaccuracy in the transmitted data.

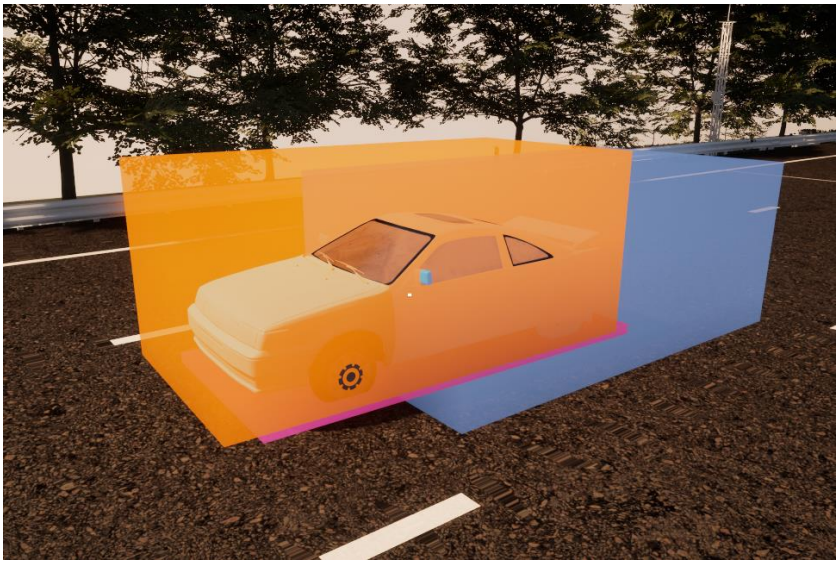


Fig. 2.1: Visualisation of CPM data using OBU (vehicle model), OSN (blue box), RSU (orange box), and V2I (pink area)

2.2 WIM

The WIM function represents a Weigh-in-Motion system in the Demonstrator. It is a digital twin of a real installation. This function aims to display measurement data from real vehicles within the Unreal Engine environment in an understandable visual form.

The WIM function processes position data for individual vehicles, particularly the measured wheel or tyre positions. These position data provide the basis for reproducing vehicle movements in the Demonstrator.

The corresponding vehicles can be interpolated from the tyre positions and displayed in the virtual environment. This produces a realistic representation of the recorded vehicle movements and the associated load distribution.

The WIM function thus links real measurement data with the visual representation in the Demonstrator, making vehicle loads and movements easy to illustrate within a scenario.

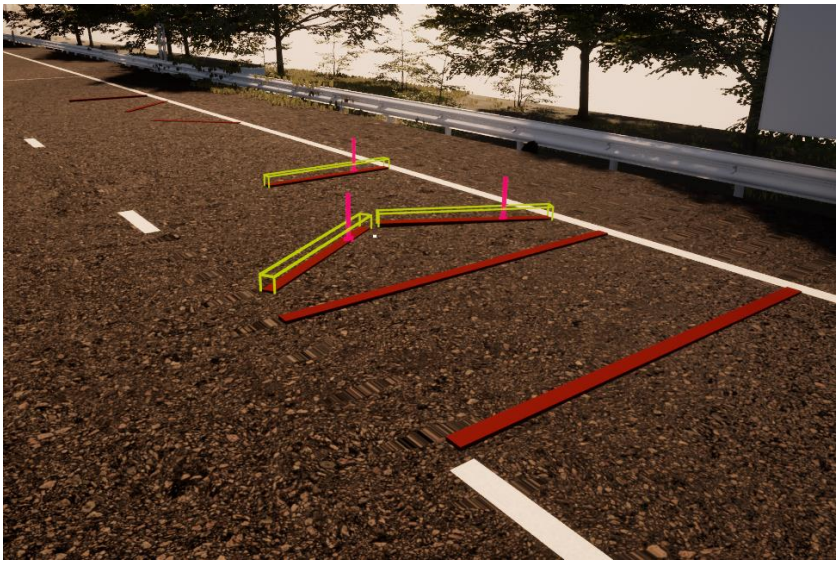


Fig. 2.2: Visualisation of WIM data on the corresponding measurement fields

2.3 PRS

PRS stands for Pavement Response System. In the Demonstrator, it displays the calculated effects of loads on the road surface, visualising how loads within a specific area can affect the pavement structure or surface.

The visualisation uses a three-dimensional grid model that divides the road area under consideration into individual segments. The calculated load values can be displayed separately for each segment. Their magnitude is represented by visual parameters such as height and colour, making differences within the area immediately visible.

This visual representation clearly illustrates local loads and their spatial distribution within the Demonstrator environment. Areas with greater effects can therefore be clearly distinguished from less heavily loaded areas.

The PRS provides a clear and intuitive visualisation of load effects on the road surface and can be used in combination with other Demonstrator functions.

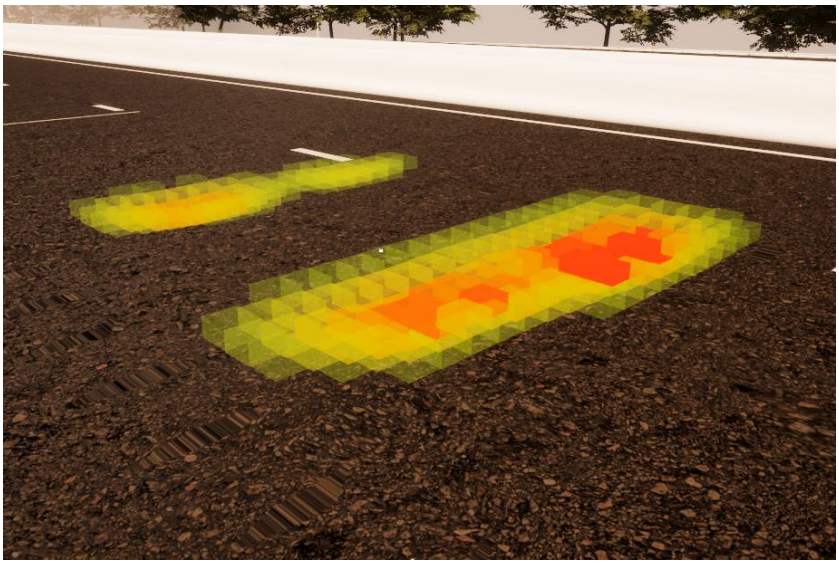


Fig. 2.3: Visualisation of PRS data for a vehicle

2.4 Gantry

The gantry function provides a visual representation of overhead sign gantries within the Demonstrator. Gantries can be displayed with freely selectable traffic signs and adapted to the scenario.

The signs shown and their arrangement can be selected freely. This allows different traffic situations or sign configurations to be reproduced within the Unreal Engine environment.

In addition to conventional traffic signs, other image files or illuminated signs can be included as required. The gantry function primarily provides a flexible visual addition to a scenario and enables a more realistic representation of the road environment.



Fig. 2.4: Visualisation of a gantry with freely selectable illuminated and traffic signs

2.5 ALE

ALE is a local tool within the Demonstrator for representing the simulation of a wheel or wheel load. Its purpose is to display the effect of this load on the road and make it visible within the Unreal Engine environment.

This function primarily illustrates loading scenarios and can be used to examine their effects on the road surface within the Demonstrator.

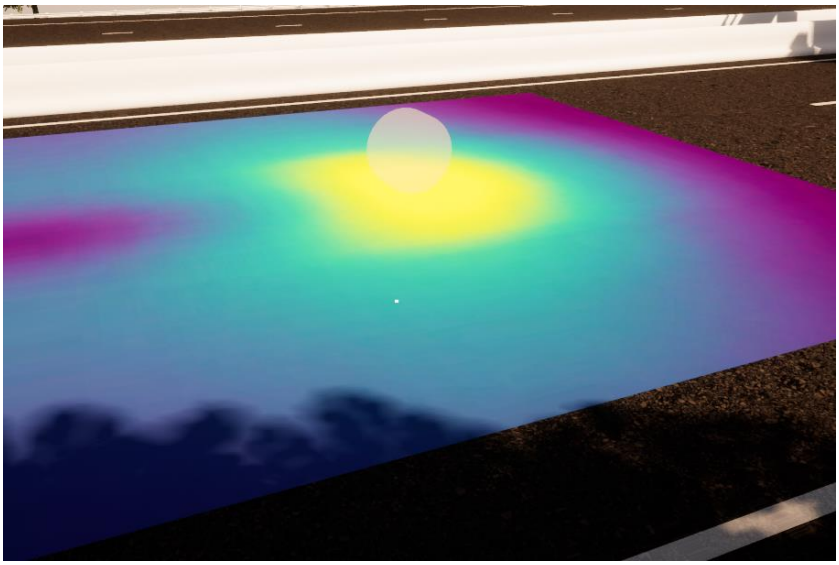


Fig. 2.5: Visualisation of a predefined ALE scenario

3 Installation and Commissioning

Before using the Demonstrator, a small number of programs and plugins must be installed and configured. These include Unreal Engine and the Demonstrator plugin.

Depending on the application, additional programs such as Python or Houdini Engine may be required. These allow, for example, scripts to process data for the Demonstrator or roads to be imported and created within the Demonstrator environment.

The following sections describe the required programs and their installation prerequisites.

3.1 Unreal Engine

Unreal Engine is required as the foundation of the entire Demonstrator. The Demonstrator was developed as an Unreal Engine plugin and is integrated and run within an Unreal Engine project.

The Demonstrator was developed and tested using Unreal Engine 5.2.1. This version is therefore recommended. Other Unreal Engine versions may also work, but have not been tested. Correct operation of the Demonstrator cannot therefore be guaranteed with other versions.

Plugins

In addition to installing Unreal Engine, several plugins must be enabled within it. These include OSC, MQTT, and the Demonstrator plugin itself. These plugins provide the interfaces and functions required for correct operation of the Demonstrator.

Before the Demonstrator plugin can be enabled in Unreal Engine, it must be added to the Unreal Engine project. First create an Unreal Engine project, then create a folder named "Plugins" in the project directory and move the extracted Demonstrator folder into it.

Once the plugin has been added, reopen the project. The required plugins can then be enabled in Unreal Engine's plugin manager. After enabling OSC, MQTT, and the Demonstrator plugin, the project usually needs to be restarted so that all functions load correctly.

3.2 Python

Python is one way to send data to Unreal Engine via OSC. Suitable Python scripts can transmit prepared, recorded, or custom data to the Demonstrator for further processing or visualisation.

The Python scripts are currently used with Python 3.12.6. Other Python versions may also work, but compatibility has not yet been fully checked. It is therefore recommended to use Python 3.12.6 initially.

Additional libraries are required to run the scripts. Installation using a package manager such as pip is recommended. The required packages are listed in a requirements file supplied with the Python scripts and can be installed using that file.

The python-osc package is typically used for OSC. It provides functions for sending and receiving OSC messages and can be installed using pip.

The required packages can be installed using the following command:

```
pip install -r requirements.txt
```

3.3 Visual Studio

The Visual Studio IDE was used to develop the Demonstrator plugin. Visual Studio is primarily used to edit and develop C++ code for Unreal Engine projects. Visual Studio 2022 was used. Other compatible versions, such as Visual Studio 2019, can also be used in principle.

Visual Studio is not essential for simply using the Demonstrator. The Demonstrator plugin is already fully integrated into the Unreal Engine project as a plugin and normally requires no further modification. Visual Studio is therefore only required to edit, extend, or recompile the plugin's source code or other C++ files.

As Visual Studio is optional for normal use of the Demonstrator, its installation and setup are not covered further in this manual. Other suitable development environments or code editors may also be used, provided they support Unreal Engine C++ projects.

3.4 Houdini Engine

Houdini Engine is optional and is not required for basic use of the Demonstrator.

However, the road used in the Demonstrator was created in Houdini and transferred to Unreal Engine using a dedicated Houdini plugin. This workflow requires Houdini 19.5.1231 and the corresponding Unreal Engine plugin with the same version number.

Houdini Engine is therefore only required to create or modify the road in Houdini, or to transfer it to Unreal Engine again. Existing road models or other suitable 3D models can also be imported directly into Unreal Engine. In this case, Houdini Engine is not needed to use the Demonstrator.

As Houdini Engine is entirely optional, installation and setup of the Houdini plugin in Unreal Engine are not covered further in this manual. If required, the appropriate Houdini and plugin versions must be configured separately.

3.5 Project Creation and Setup

To use the Demonstrator plugin, first create an Unreal Engine project. Start Unreal Engine to open the Project Browser, where a new project can be created. Select "Games" on the left side of the window, then "First Person" in the centre selection panel. Other presets can also be used, although additional

settings may need to be adjusted later. All other settings can be selected as desired and have little effect on the subsequent steps. Note the project path selected at the bottom left of Fig. 3.1 for the next step.

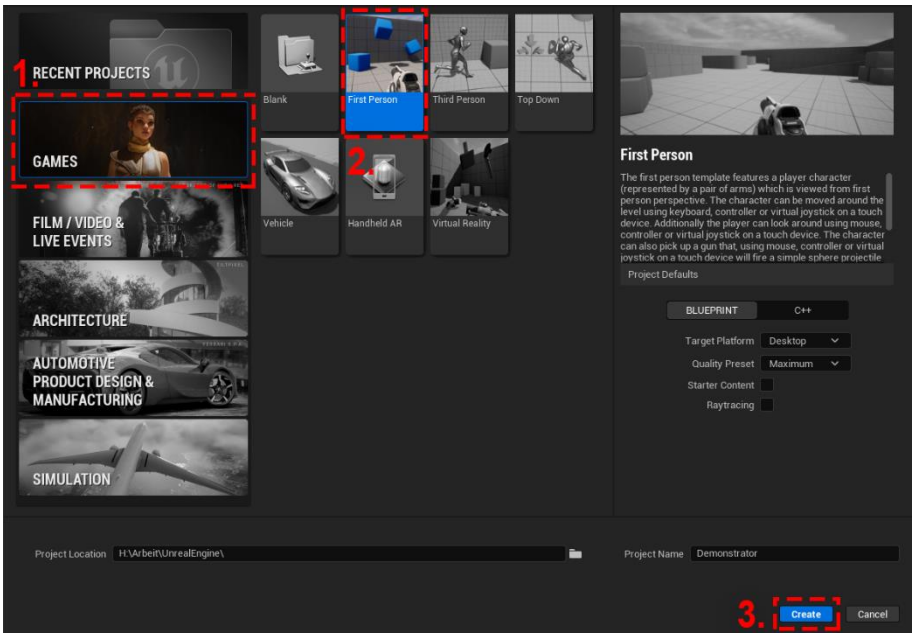


Fig. 3.1: Unreal Engine project creation window

Next, install the Demonstrator plugin. First create a folder named "Plugins" in the project's root directory. Extract the Demonstrator archive, then move or copy the entire extracted folder into the Plugins folder. The Plugins folder should now contain a folder named "Demonstrator".

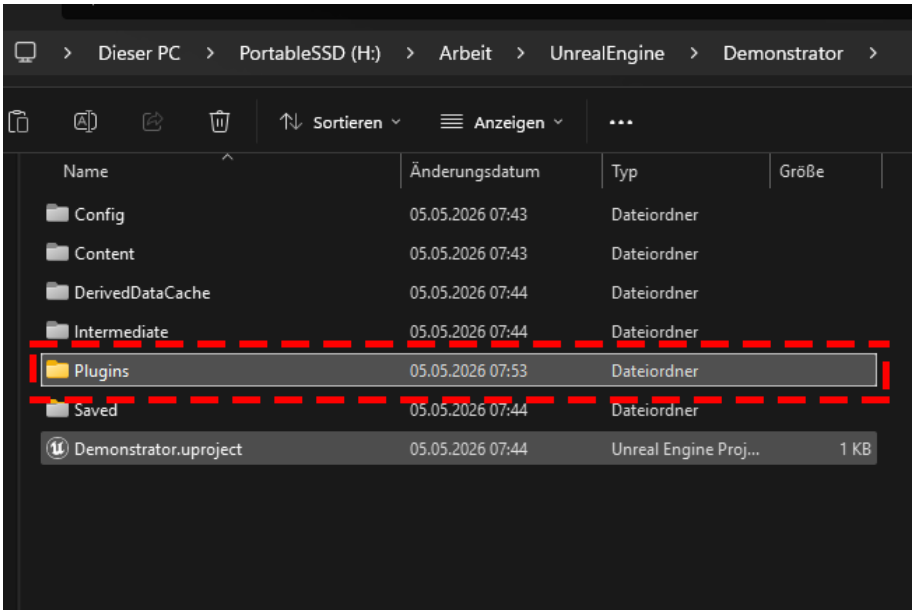


Fig. 3.2: Creating the Plugins folder for Unreal Engine

Once this is complete, start the project. The Demonstrator must then be enabled in the plugin manager. Open the plugin manager in the Unreal Engine project as shown in Fig. 3.3.

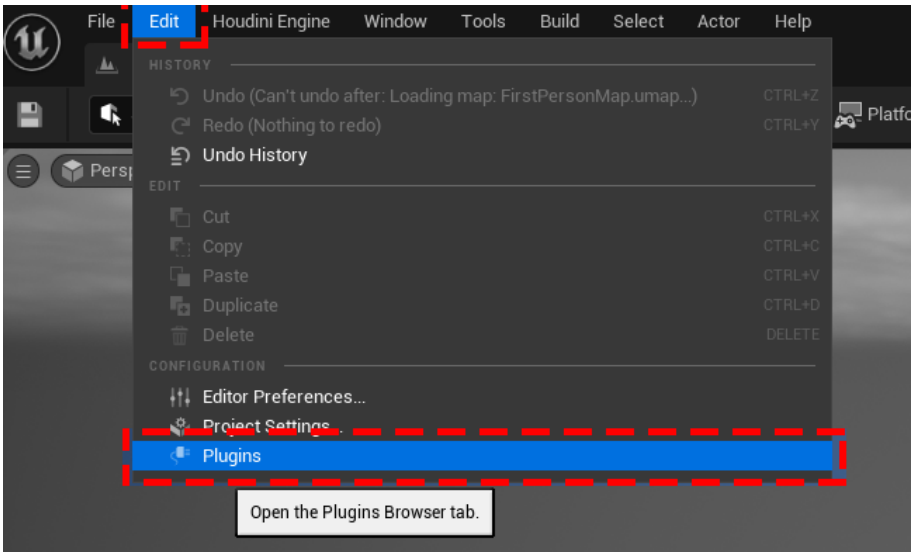


Fig. 3.3: Opening the Unreal Engine plugin manager

Search for the Demonstrator using the search bar at the top and enable it. If not already enabled, also enable OSC and MQTT here. Unreal Engine usually needs to be restarted after activation. After the restart, only a few steps remain before the Demonstrator is ready to use. Next, enable access to plugin content in the Content Browser. Open it using "CTRL + SPACE" and enable "Show Plugin Content" as shown in Fig. 3.4.

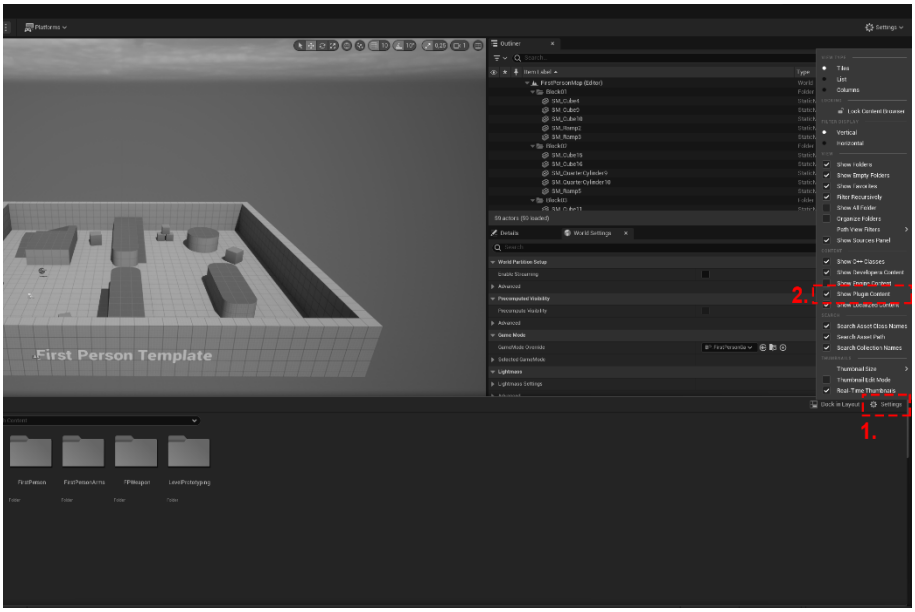


Fig. 3.4: Enabling Show Plugin Content in Unreal Engine

The "Demonstrator Content" folder is now available in the Content Browser's Sources panel. It contains all the important tools and files for the following steps. Another setting required to use the Demonstrator is in "World Settings". By default, this tab is on the right, next to the Details panel. Ensure that "GameMode Override" is set to "Demonstrator_Gamemode". The other two settings should be populated automatically. If they are not, adjust them as shown in Fig. 3.5.

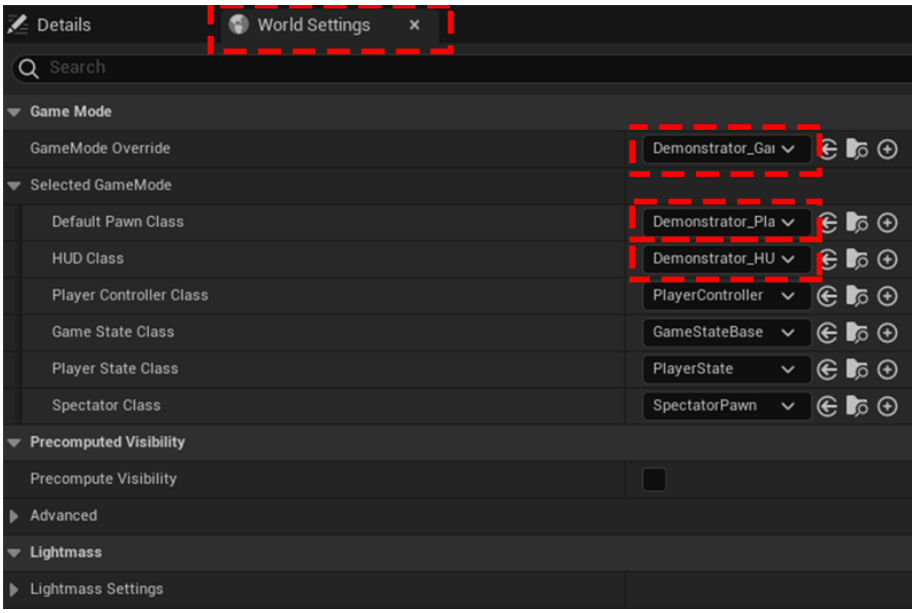


Fig. 3.5: Unreal Engine GameMode settings in World Settings

Once all settings have been configured, the Demonstrator is fully functional and ready to use. The default map, containing a preconfigured road with all available functions, is located in the Content Browser under:

"Demonstrator Content" -> "Maps" -> "Demonstrator"

This map contains all preconfigured functions, including CPM and WIM. The next chapter, "Operation", explains general operation, the functions, and how to use them.

4 Operation

This chapter first covers general operation and controls within the Demonstrator, then describes the individual features. Basic knowledge of Unreal Engine operation is assumed and is not covered here.

Selecting the Game Mode has already defined the Demonstrator's interface and controls. These normally require no further adjustment. Only a few basic inputs and menus are relevant during operation.

The Demonstrator has a graphical user interface, also known as a HUD. It displays the main keyboard shortcuts and controls in the lower-left corner. In addition to normal movement using W, A, S, D, and the space bar, and control using the mouse, further functions are available.

Press F to toggle flight mode. Holding the Ctrl key increases movement speed during both normal movement and flight. The Ctrl key therefore acts as a sprint control.

Press I to open the information menu. This menu provides an overview of the individual Demonstrator features. It also shows which OSC ports the functions listen on and whether they are currently enabled.

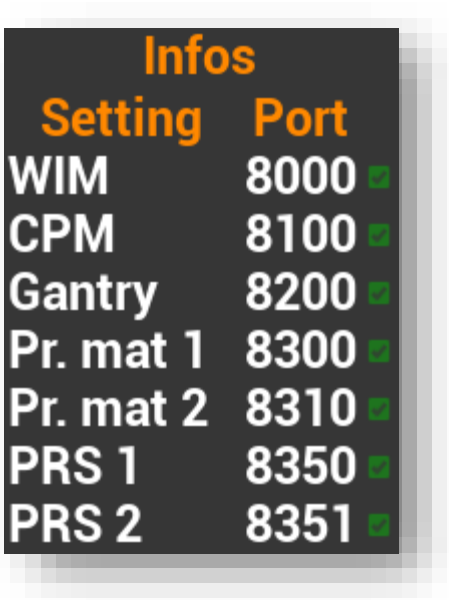


Fig. 4.1: Demonstrator information panel

Press O to open the options menu. Advanced settings can be adjusted here during operation, including enabling and disabling individual features. For CPM, individual subcategories can also be enabled or disabled, and covariance visualisation can be toggled.

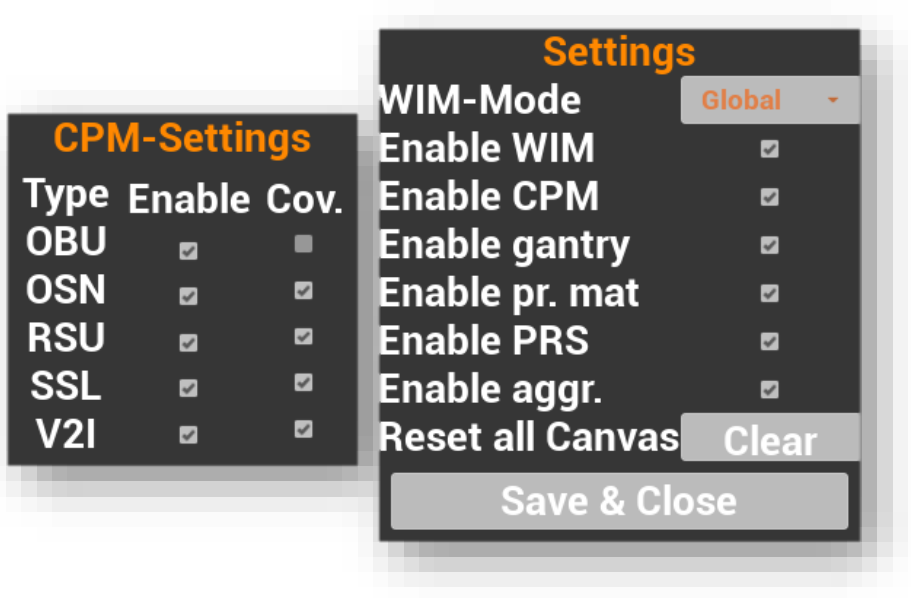


Fig. 4.2: Demonstrator options panel

4.1 CPM

Enable or disable the CPM module by selecting or clearing its checkbox in the options menu. When selected, the CPM feature is enabled and the corresponding OSC server is started or the data are processed. Incoming position data can then be received, processed, and used within the Demonstrator. Clearing the checkbox disables the feature.

A separate CPM settings section is also available. Individual categories, including OBU, OSN, RSU, SSL, and V2I, can be enabled or disabled here. The covariance visualisations for each category can also be

enabled or disabled individually. This allows precise control over which data types the CPM module considers and processes.

Data are received via OSC and use a JSON structure. The basic structure is shown in Fig. 4.3 below. Incoming data may contain, for example, position information and other associated values. Ensure that the data match the expected structure so that the system can interpret them correctly.


```
{
  "object_list": {
    "header": {
      "seconds": 1764518877.471214,
      "frame_id": "map"
    },
    "objects": [
      {
        "id": 13,
        "existence_probability": 0.99,
        "state": {
          "x": 29.180837,
          "y": 40.967461,
          "v_x": -13.434225,
          "v_y": 0.419641,
          "a_x": 0.0,
          "a_y": 0.0,
          "yaw": -3.172819,
          "yaw_rate": 0.0,
          "covariance": [ ···
          ]
        },
        "dimension": {
          "length": 3.96,
          "width": 1.89,
          "height": 1.08,
          "covariance": [ ···
          ]
        },
        "classification": { ···
        },
        "features": { ···
        },
        "axles": []
      }
    ]
  }
}
```


Fig. 4.3: Structure of a JSON file for the OBU category in CPM

The current state of the CPM module can be viewed in the information panel. Its status is currently displayed under Tracker. This shows whether the module is active and whether incoming data are being processed.

4.2 WIM

The WIM module integrates data from a Weigh-in-Motion system into the Demonstrator. Enable or disable the WIM feature using its checkbox in the options menu. Once enabled, incoming measurement data can be received via OSC and visualised within the application.

The corresponding settings section controls the level of detail in the visualisation. Global and fine visualisation modes are available. Global mode shows which measurement field is loaded by a wheel, indicating where the wheel is located within the system. Fine mode adds a more precise tyre position within the measurement area.

The data transfer structure differs from the Demonstrator's other OSC-based functions. While other modules use their own data formats, the WIM module uses values from a CSV file. An example extract

is shown below. An external script reads the file and then transmits the data to the Demonstrator via OSC.

	time	x	y	sensor	vehicle	axle	wheel	wheelload	…
1	0.5	0	1	1	1	1	1	30.5	…
2	1	1	1	2	1	1	1	31.2	…
3	1.5	2	1	3	1	1	1	30.9	…
4	1.5	2	2	4	1	1	2	35.1	…
5	2	3	1	5	2	1	1	20.0	…

Fig. 4: Example CSV file for the WIM system

In addition to the CSV file, a suitable sending script is therefore required. This script prepares the values and sends them to the intended OSC address. The current script also allows measurement values to be interpolated. Interpolated values are transmitted via the OSC address /interp and are used to display an estimated tyre position.

A simplified geometry can also be displayed. It is intended only for visual orientation and does not represent a realistic wheel size. The tyre's position on the Weigh-in-Motion system is the key information here, rather than an exact geometric representation of the wheel.

4.3 Gantry

The overhead sign gantry, referred to below as the Gantry, represents and integrates infrastructure elements within the Demonstrator. For example, it can visualise sensors, display elements, or communication components above the carriageway. This allows typical traffic infrastructure, such as that used in real measurement or detection systems, to be reproduced.

Enable or disable the Gantry feature in the options menu. Once enabled, the gantry is displayed in the scene and can be linked to other Demonstrator functions, depending on the settings.

The current Gantry function includes five signs. These can be extended, replaced, or removed as required, allowing the displayed traffic signs or display elements to be adapted to the scenario.

The signs are controlled exclusively via OSC using a dedicated JSON structure.

```
{
  "object_list": {
    "header": {
      "seconds": 1764518877.471214,
      "frame_id": "signs"
    },
    "objects": [
      {
        "id": 1,
        "type": "150-schild-black"
      },
      {
        "id": 2,
        "type": "141-schild2-red"
      }
    ]
  }
}
```

Fig. 4.4: Structure of a JSON file for the gantry

This structure specifies which sign is displayed at which position or for which ID. A list of available signs is provided for this purpose. Copy the required names from this list into the JSON structure. The list follows the official numbering of traffic signs.



Fig. 4.5: Traffic signs from left to right: 276-black, 274-80-black, 282-black

The data are then sent to the corresponding receiver in the Demonstrator. The signs do not validate the content of the transmitted data. In particular, the system does not check whether the selected combination of sign, position, or displayed content makes sense from a traffic engineering perspective. The specified image or sign is displayed directly. This behaviour is intentional and allows the Gantry display to be configured as freely and flexibly as possible.

4.4 PRS

Like the other Demonstrator functions, the PRS can be enabled or disabled in the options menu. Once enabled, its grid is displayed in the scene and can receive external data.

Multiple PRS instances can be integrated into the Demonstrator. The current setup uses two PRS systems. Their assigned ports are shown in the information panel, which indicates the OSC port each instance listens on and whether it is currently enabled. As with the other functions, data are transferred via OSC using a defined JSON format.

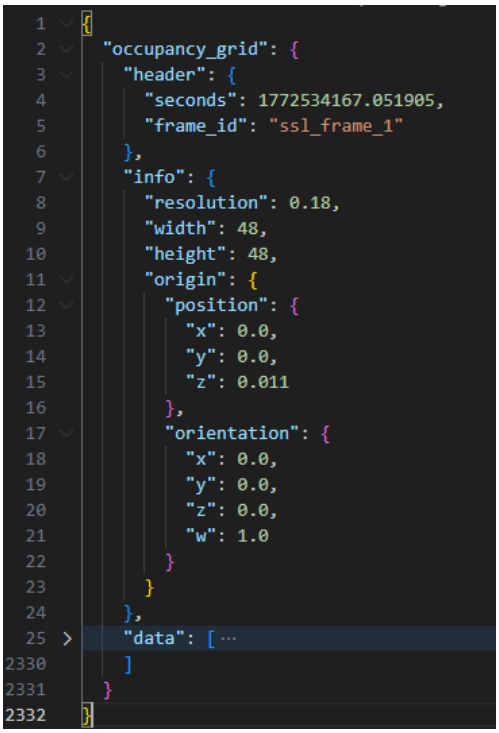


Fig. 4.6: Structure of a JSON file for the PRS

For processing, the PRS expects an array in the "data" field representing a 48 × 48 grid. This grid is passed as a flat list in the JSON structure, containing a total of 2304 values. Each value is assigned to a PRS grid element and then visualised in three dimensions within the scene.

For correct visualisation, the specified value ranges or limits must be observed. These limits define the range within which incoming data are interpreted. Values within this range are then mapped to the visual representation, for example to the height and colour of individual grid elements.

The relevant limits and their meaning are explained in more detail in the following chapter on settings.

4.5 Pressure Mat

The pressure mat can generally be used in the same way as the PRS. It is also controlled through the settings and information panels, receives data via OSC, and expects the same JSON structure. The values are therefore also passed as an array in the "data" field.

However, the pressure mat uses a different visual representation. While the PRS visualises incoming values as a three-dimensional grid with height and colour changes, the pressure mat displays them on a two-dimensional surface.

The value range and specified limits may also differ from those of the PRS. These limits determine how incoming data are visualised and should be adjusted or observed according to the pressure mat's settings.

4.6 Aggregation Mat

The Aggregation Mat is visually similar to the pressure mat. It also uses a grid-based system, but is calculated locally within the Demonstrator and does not require separate data transmission via OSC. This feature can also be enabled or disabled through the settings panel.

The Aggregation Mat records and displays vehicle passes within a specific area. It evaluates how often individual grid areas are driven over. As with the pressure mat, the results are displayed on a two-dimensional surface.

The main difference from the pressure mat is the meaning of the displayed values. Whereas the pressure mat visualises load data, the Aggregation Mat's colours represent the number or frequency of vehicle passes. Areas driven over more frequently can therefore be distinguished visually from areas with fewer passes.

The Aggregation Mat can be reset through the settings panel if required. This deletes the vehicle passes recorded so far and restarts the count.

4.7 Python Scripts for Sending Data

Various Python scripts are available for sending prepared JSON files to Unreal Engine. A direct live connection is therefore not required. Instead, the scripts can be used to transmit recorded, prepared, or custom datasets to the Demonstrator as needed.

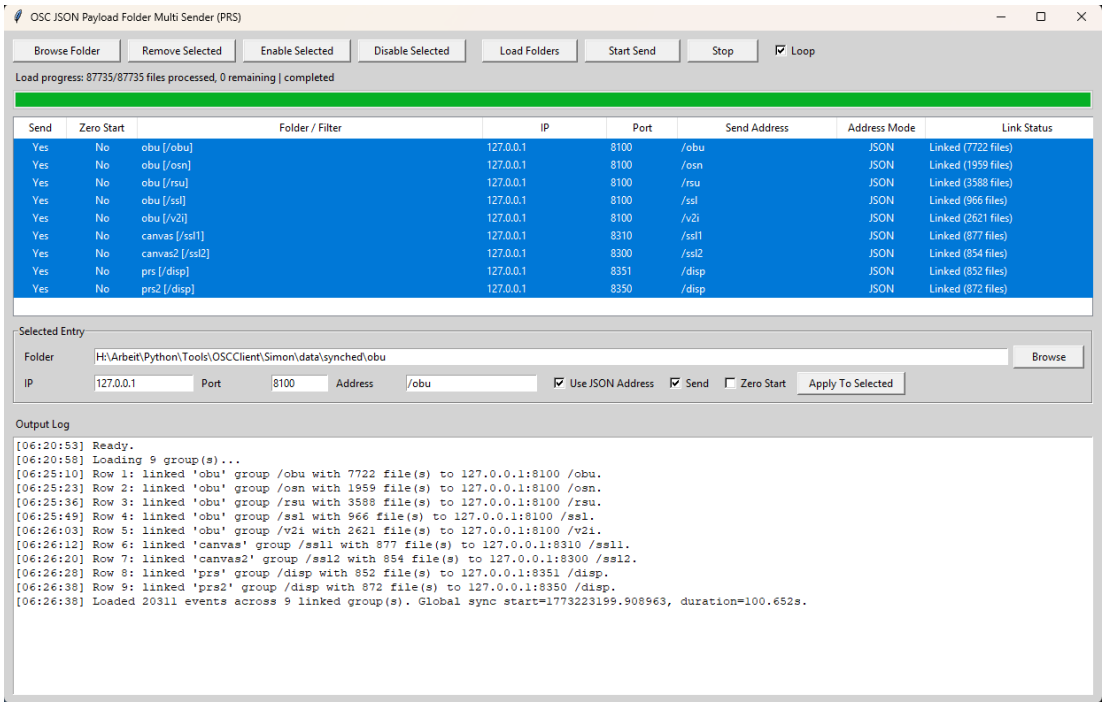


Fig. 4.7: Python interface for operating the CPM, PRS, and pressure mat functions

The script shown in Fig. 4.7 sends CPM, PRS, and pressure mat data. One or more folders containing the JSON files to be sent can be added to the script. The loaded datasets are then displayed in a list, where they can be enabled or disabled individually.

Click an entry in the list to adjust its settings, including the destination IP address, port, and other sending options. Changes only take effect when "Apply to Selected" is pressed. This allows individual datasets to be adjusted without directly modifying the original files.

An important option is the Zero-Start function. Many datasets contain timestamps defining when each item of data is sent. Without adjustment, transmission may therefore begin only after a waiting period. Zero-Start shifts the starting time so that data playback begins immediately when sending starts. This is particularly useful for testing and debugging, as it allows a quick check that the data are displayed correctly in the Demonstrator.

After selecting the required folders, load the data using "Load Folders". The files in those folders are searched and prepared. Then select the datasets to be sent. Use "Start Send" and "Stop" to start or stop transmission. Loop mode can also be used to replay the data after reaching the end.

Another Python script is the Gantry Client. It can send existing JSON files to the Demonstrator or create custom sign combinations for the Gantry and transmit them directly.



Fig. 4.8: Python interface for operating the gantry

First check the IP address and port at the top of the script interface. The OSC address must also be specified, although it is not otherwise central to the current usage. The preset default values can generally be retained if the settings in the Demonstrator have not been changed.

Custom rows of signs can be prepared in the Presets section. These presets are saved locally whenever they are changed, so the entered values remain available after closing and reopening the script. Select a row of signs using the radio buttons on the left, then send it using the corresponding button.

The script also allows presets to be exported or imported. A preview of the JSON structure generated for transmission is shown at the bottom. Use this preview to check the generated data or as a basis for custom JSON files. A JSON file can also be sent directly using "Send File". This is not essential for normal use of the Gantry Client, as sign rows can be sent directly from the presets.

An "Image File Picker" is also located at the bottom of the script interface. Select a folder containing the available image files. Only image names that have also been added to Unreal Engine may be used. Therefore, use only the image files from the signs' corresponding Style-Images folder. The Demonstrator cannot correctly match other filenames.

The displayed image files can be selected in the Gantry Client and added to the prepared sign rows. These rows can then be sent directly to the Gantry and displayed within the Demonstrator.

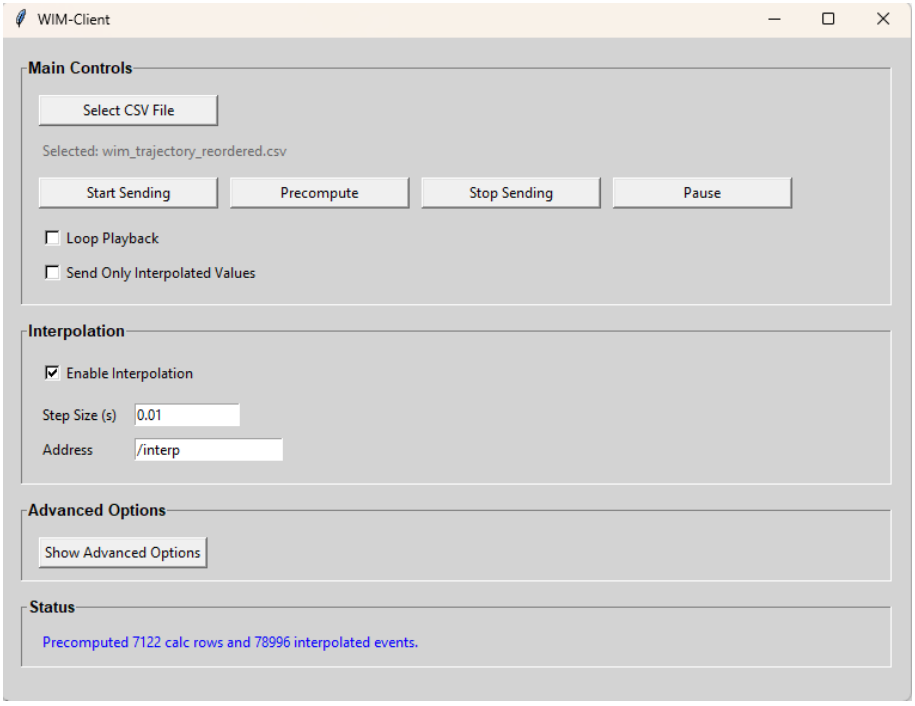


Fig. 4.9: Python interface for operating the WIM function

The WIM Client sends WIM data to the Demonstrator. First select a suitable CSV file using "Select CSV File" at the top. Ensure that the columns in this file are in the expected order. Each value must be in its intended position within the CSV file so that the script can read it correctly and transmit it to the Demonstrator.

A key function is "Enable Interpolation". When enabled, WIM data are not displayed only at the original sensor positions. Additional intermediate values are calculated, allowing vehicle movement to be displayed between recorded positions. This produces a smoother and more understandable representation of movement.

Use "Step Size" to set the interpolation increment. A smaller value produces more intermediate steps and a finer visualisation. A larger value reduces the number of calculated intermediate positions.

After selecting the CSV file and setting the required options, run "Precompute" first. This prepares the required data and calculates interpolation values if enabled. Playback can then be started, stopped, or paused using the corresponding buttons.

Further settings are available under "Advanced Options". They are not described in detail in this manual, as they are primarily intended for testing and debugging or include experimental functions.

5 Settings

This chapter describes the main settings for the individual Demonstrator functions. It does not explain every available option in detail, as some settings are primarily intended for debugging, advanced configuration, or experimental functions.

The focus is on parameters relevant to basic use, adaptation, and control of each function. This provides a clear overview without unnecessary detail on internal options or those that are not yet fully developed.

5.1 CPM

The CPM function generally requires three elements in the Demonstrator map: the CPM Actor, an Origin, and a parking position.

Both the Origin and the parking position are of type Centerpoint. This class defines reference points within the scene. The use of Centerpoints is explained in more detail below.

Add the CPM Actor to the level if it is not already present. Its exact position in the scene is generally not important, as the orientation and scaling of incoming data are controlled through its settings.

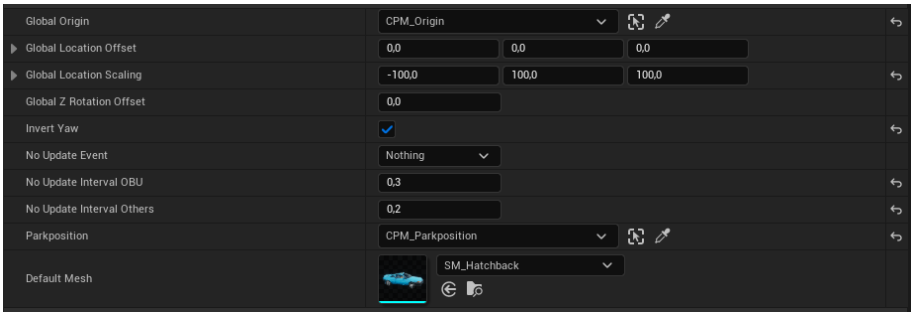


Fig. 5.1: Unreal Engine CPM settings

First specify a "Global Origin" in the CPM settings, using the previously created Origin Actor of type Centerpoint. It serves as the reference point for incoming position data and defines the origin from which vehicle positions are interpreted in the scene.

A "Position Offset" and "Scaling" can also be set. Position Offset shifts the incoming data spatially. Scaling increases or decreases the scale of the position data. This is particularly useful when incoming data use different units or scaling, for example metres instead of centimetres. "Z-Rotation Offset" also allows the orientation around the Z-axis to be adjusted.

The "Invert Yaw" option reverses the vehicles' direction of rotation so that it can be set correctly. Another important setting is "No Update Event". It determines what happens to a vehicle if no new data are received for its ID within a specified period. Selecting "None" leaves the vehicle at its last position. Alternatively, the vehicle can be moved to a defined parking position or removed from the scene entirely.

The associated time parameters define the period after which this event is triggered. To park a vehicle when updates stop, a suitable parking position must also be specified. This position is also defined using a Centerpoint.

The CenterPoint Actor is used solely to reference specific scene positions in other settings. For example, it can define an origin, a parking position, or other fixed reference points for Demonstrator functions. To create a Centerpoint, drag the corresponding Blueprint into the level and place it at the required position. Give the Actor a meaningful name so that it can be clearly identified in the relevant settings later.

Finally, specify a "Default Mesh" in the CPM settings. This mesh is used for vehicles created or displayed from incoming OBU data.

5.2 WIM

Using the WIM system first requires a measurement station. The number of associated measurement fields can vary depending on the setup and is not fixed.

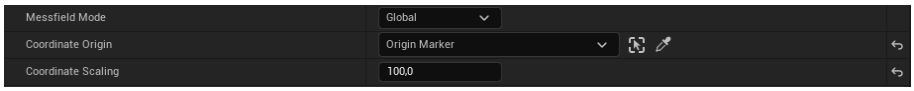


Fig. 5.2: Unreal Engine WIM measurement base settings

The measurement station settings include "Messfield Mode", which can be set to Global or Fine. This setting is not critical during basic setup, as the mode can also be changed later through the options menu during operation.

The measurement station also requires a coordinate origin of type Centerpoint. This serves as the reference point for incoming measurement data, which use their own coordinate system. The associated Scaling setting also allows the measurement data to be adapted to the scale of the Unreal Engine environment.



Fig. 5.3: Unreal Engine WIM measurement field settings

Each measurement field also requires its own ID. These IDs are OSC addresses and must therefore always begin with a slash. In the current setup, measurement fields use addresses such as /id1, /id2, through to /id10.

The measurement fields also have their own mode, which is inherited from the measurement station. Changing the station's mode passes the setting on to the individual measurement fields.

5.3 Gantry

The Gantry function generally uses two types of objects: the Gantry Server and the individual Gantry signs.

The Gantry Server controls the gantry and has only a few relevant settings. The main option is "Enable Server", which enables or disables the server. As the same function can also be controlled through the options menu during operation, no further adjustment is generally required here.

Each Gantry sign has its own ID. This ID allows data sent via OSC to be assigned to the correct sign. When adding signs, ensure that each receives a unique ID.

In the current Demonstrator map, sign numbering starts at ID 0. Assignment begins above the right-hand lane and continues towards the middle and other lanes. Each ID can therefore be used to specify which sign at which gantry position is addressed and changed.

5.4 PRS

The PRS offers more settings than the other functions. Many are not essential for normal operation and can generally be left unchanged or copied from the figures.

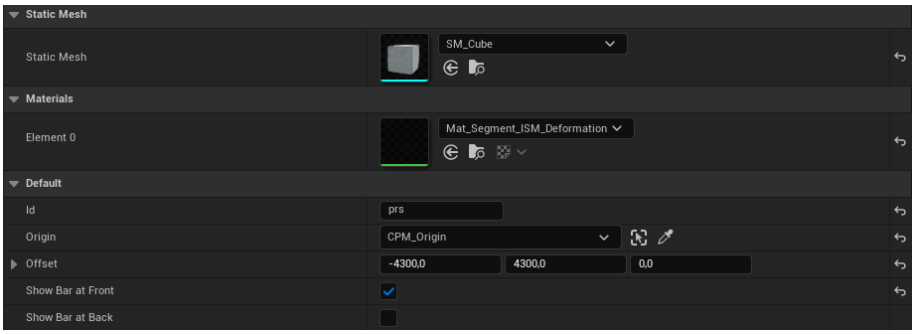


Fig. 5.4: Unreal Engine PRS settings (1)

First ensure that the correct values are entered for "Static Mesh" and "Material". In principle, any object can be used as the Static Mesh from which the grid is built. The Demonstrator uses cubes. The material is particularly important: select the intended Deformation material, as it controls the visual representation of the values.

Under "ID", assign a unique name to each PRS. For example, the Demonstrator map uses prs. Additional PRS instances can be named “prs2”, “prs3”, or similar. The name must be unique and contain “prs”.

Specify an Origin of type Centerpoint for spatial referencing. An offset relative to the Origin can also be set. In the current Demonstrator map, this reference has been chosen to match the real CPM Lab and its coordinate origin, allowing incoming data to be positioned correctly in the virtual representation.

The "Show Bar at Front" and "Show Bar at Back" options provide visual orientation only. They display markers at the start and end of the mat. This is useful because the PRS position is otherwise difficult to identify when no values are active.

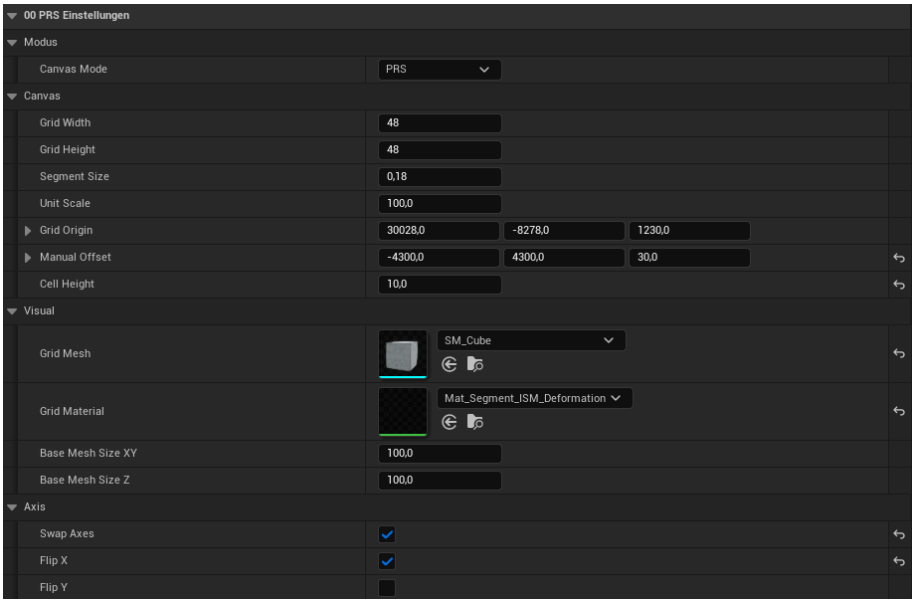


Fig. 5.5: Unreal Engine PRS settings (2)

Select the appropriate mode in the "00-PRS" settings section. For use as a PRS, select the corresponding PRS mode. Other modes, such as Aggregation, are covered in a separate section.

"Grid Width" and "Grid Height" define the grid dimensions. The Demonstrator uses a grid of 48 × 48 elements, matching the original laboratory setup. These values can be adjusted for other mats or grid sizes. However, the JSON structure sent via OSC must also be adjusted accordingly. The number of transmitted values must match the configured grid dimensions.

"Segment Size" defines the displayed size of individual grid elements. "Unit Scale" can generally remain at 100 unless a different scale is required.

A "Grid Origin" must also be set manually. In the current configuration, it corresponds to the CPM Origin and its position. The offset below it can shift the entire grid further. Among other adjustments, a height offset has been entered so that the PRS is displayed above the carriageway, as the actual origin is at carriageway level.

"Cell Height" determines the extent to which the individual cells can vary in height. It therefore affects the three-dimensional representation of incoming values.

The "Visual" section normally requires no further adjustment if the mesh and material settings at the top have already been set correctly. The "Grid Mesh" and "Grid Material" entries here are largely redundant and provide alternative or additional settings.

Individual axes can be mirrored in the "Axis" section if required. These options are only relevant if the orientation of incoming data does not match the representation in the Unreal Engine scene.

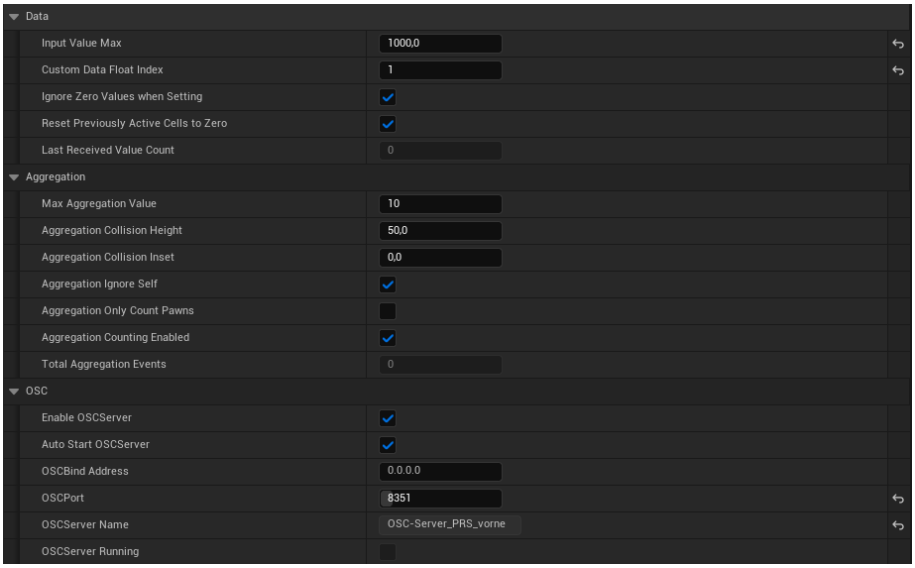


Fig. 5.6: Unreal Engine PRS settings (3)

The "Data" section is particularly important. It defines the upper limit for incoming data, determining how the data are interpreted and interpolated for visualisation. For example, if incoming values range from 0 to 1, the upper value must be set to 1. The current Demonstrator configuration uses PRS values from 0 to 1000. "Custom Data Float Index" should remain at 1. The other settings can generally be left unchanged.

The "Aggregation" section can be ignored for normal PRS use, as its settings are only relevant to the local aggregation function.

The default settings in the "OSC" section can generally be retained. If required, the port on which the PRS receives data can be changed here. Ensure that this port matches the sending script or corresponding data source.

5.5 Pressure Mat

The pressure mat settings are similar to the PRS settings in many respects. Basic parameters such as grid dimensions, segment size, origin, offset, axis orientation, and data range can therefore generally be configured in the same way as for the PRS.

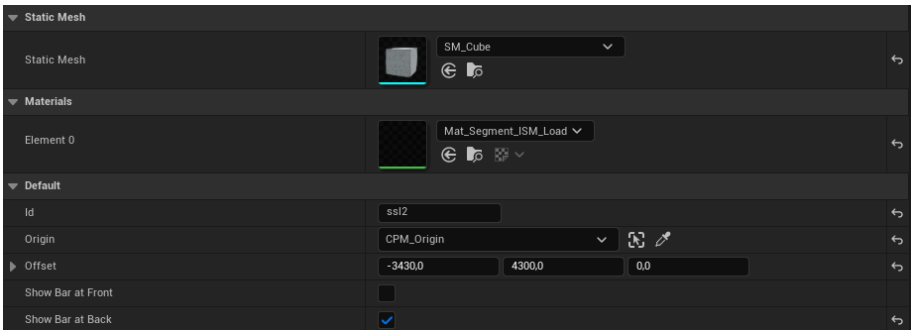


Fig. 5.7: Unreal Engine pressure mat settings

The main difference is the ID. While the PRS uses an ID containing “prs”, the pressure mat's ID must contain ssl. Ensure that the name is in lower case.

There is also an important difference in the material. Do not use the Deformation material for the pressure mat. Instead, select the corresponding Load material, as the pressure mat visualises data differently from the PRS.

The OSC port must also be adjusted. Each pressure mat instance requires the appropriate port for receiving incoming data. This port must match the sending script or corresponding data source.

All other settings can generally follow those of the PRS unless a specific adjustment to visualisation or positioning is required.

5.6 Aggregation Mat

The Aggregation Mat settings generally follow those of the pressure mat. It also uses the Load material, as it provides a two-dimensional colour visualisation.

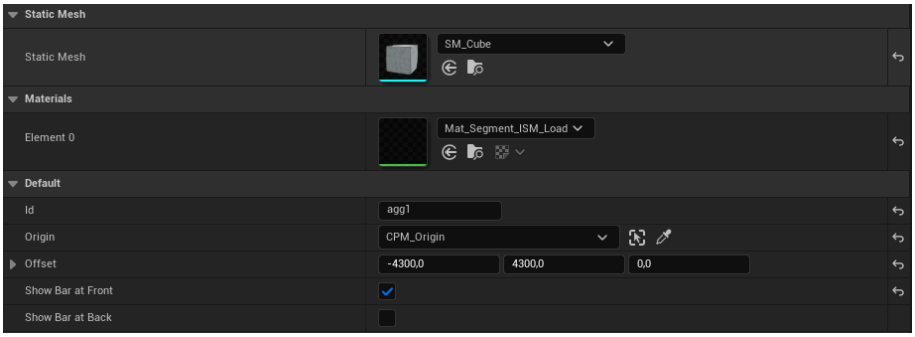


Fig. 5.8: Unreal Engine Aggregation Mat settings (1)

Assign a unique ID containing agg. This allows the Aggregation Mat to be correctly distinguished from the other grid-based functions.

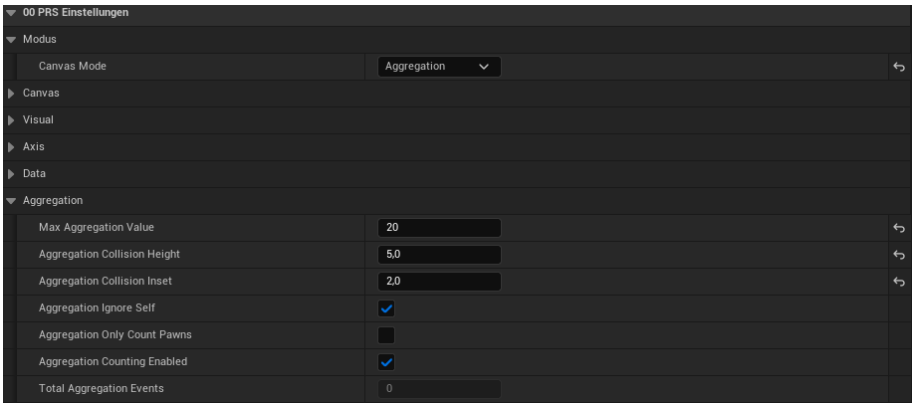


Fig. 5.9: Unreal Engine Aggregation Mat settings (2)

The mode must also be set to "Aggregation". Only in this mode are values calculated locally from vehicle passes rather than taken from external load data.

The main parameter in the Aggregation settings is "Max Aggregation Value". It defines the number of vehicle passes at which the maximum display value is reached. A lower value makes the mat reach its maximum indication sooner. A higher value requires more passes before the maximum value is displayed.

The other Aggregation settings, particularly collision or hitbox parameters, can generally be copied from the existing template. They do not need to be adjusted for normal use.

All other settings can generally follow those of the pressure mat or PRS unless specific adjustments to positioning, scaling, or visualisation are required.

6 OSC Ports and Data Structure

OSC is used to transmit data to the Demonstrator. OSC stands for Open Sound Control and serves as the interface between external data sources and the Unreal Engine environment in this project. It supplies data to different Demonstrator functions at runtime.

An OSC message generally consists of a target address and content. Transmission requires an IP address, a port, and an OSC address. The IP address identifies the computer to which the data are sent. The port determines which function or receiver within the Demonstrator is addressed. The OSC address then describes the type of data more precisely. An OSC address usually starts with a slash, for example /ALE or /CPM. The address is followed by the message content. Depending on the function, this can include numeric values, text, status values, or structured data.

```python
# Beispiel zum Senden von OSC-Daten an Unreal Engine
# Installation der benötigten Bibliothek:
# pip install python-osc

from pythonosc.udp_client import SimpleUDPClient

# Zieladresse: IP-Adresse und Port des Empfängers
# 127.0.0.1 bedeutet: derselbe Rechner, auf dem auch Unreal läuft
ip = "127.0.0.1"
port = 7800

# OSC-Client erstellen
client = SimpleUDPClient(ip, port)

# OSC-Adresse und zu sendende Werte
# OSC-Adressen beginnen üblicherweise mit einem Slash "/"
address = "/address"

# Beispielwerte, z. B. eine einfache Position im Raum
x = 100.0
y = 0.0
z = 50.0

# Nachricht senden
client.send_message(address, [x, y, z])
```

Fig. 6.1: Python example for sending data via OSC

The Demonstrator mainly uses structured data. These are often transmitted as JSON data or generated from JSON files. This allows related information to be sent together, such as position data, rotation values, load data, vehicle classes, activation states, or other parameters. The exact data structure depends on the feature and is described in more detail in the corresponding subsections.

The IP address, port, OSC address, and data format must match the relevant function. If they do not, the Demonstrator cannot receive or process the data correctly. For custom implementations, first check which address and data structure the feature expects.

Communication takes place through defined ports. Each feature can listen on its own port, allowing incoming data to be clearly assigned to a specific function. Multiple data sources can therefore be used in parallel without the individual functional areas interfering with one another.

The following table provides an overview of the OSC ports used. Port assignments can be adjusted as required.

Function	**Port**
WIM	8000
CPM	8100
Gantry	8200
PRS	8300 / 8310
Pressure Mat	8350 / 8351

7 Notes

The Demonstrator is currently under development. Individual functions, settings, or workflows may therefore still be adapted, extended, or changed. Although the basic functions are already usable, occasional errors or unexpected behaviour cannot be ruled out.

Some features are not yet fully fault-tolerant. Invalid inputs, incomplete data, incorrect ports, or unexpected data structures are therefore not always handled automatically. For reliable operation, ensure that the required programs, plugins, data formats, and settings are configured correctly.

Some names, structures, and conventions are also still being standardised. As several functions were originally developed as independent subprojects and later combined in the Demonstrator, naming, settings, or internal workflows may still differ in places. These will be revised gradually during further development and aligned with a common scheme.

This manual describes the current state of the Demonstrator and is intended to support its setup and operation. As the project develops, individual descriptions, screenshots, or available functions may change. If problems occur, first check the relevant settings, data sources, and connections.

We used the following prop-Packs, whose creators are greatly acknowledged:

- **Vehicle Variety Pack - [Switchboard Studios]**
- **Megascans Meadow Pack - [Megascans]**
- **Industry Props Pack 6 - [SilverTm]**
- **European Beech - [Quixel]**