Summer Internship Report

on

Osdag Bridge Module

Custom Vehicle Diagrams, Dynamic Girder Highlighting & Robustness Improvements

---

Submitted by

Anushka Pareek

VIT Bhopal University

Under the Guidance of

Nidhi Khare

Osdag Project, FOSSEE, IIT Bombay

July 2026

Project Website: `https://osdag.fossee.in/`

GitHub Repository: `https://github.com/Aditya-Donde/OsdagBridge`

Acknowledgement

I would like to take this opportunity to express my heartfelt gratitude to everyone who played a pivotal role in making my summer internship at Osdag, FOSSEE, IIT Bombay an enriching and memorable experience.

I extend my deepest thanks to my mentor, **Nidhi Khare**, for believing in my abilities and guiding me throughout the development of the Osdag Bridge module. Her unwavering support, patient reviews, and constant encouragement have been the driving force behind my progress on this project.

I am also grateful to the Osdag and FOSSEE team at IIT Bombay for providing me with the opportunity to work on a real-world, open-source structural engineering tool and for their constructive feedback throughout the internship.

Finally, I would like to thank my peers and collaborators on the OsdagBridge repository, whose reviews and suggestions helped improve the quality of my contributions.

Anushka Pareek

Declaration

I declare that this written submission represents my ideas in my own words, and wherever others' ideas or words have been included, I have adequately cited and referenced the original sources. I declare that I have properly and accurately acknowledged all sources used in the production of this report. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented, fabricated, or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources which have not been properly cited or from whom proper permission has not been taken when needed.

Anushka Pareek

Abstract

OsdagBridge is a PySide6-based desktop module of the open-source **Osdag** platform (FOSSEE, IIT Bombay) used to design and analyse steel bridge structures. This report documents my contributions during a summer internship with the Osdag project, delivered through two pull requests on the official repository. The first (PR #52) introduced three custom, `QPainter`-based vehicle load diagrams – for tracked/bogie vehicles, wheeled axle trains, and clear carriageway width – replacing static placeholder text in the Custom Vehicle dialog and making the live-load input workflow visually self-explanatory. The second (PR #66) implemented real-time, synchronised girder highlighting across all cross-section CAD views, fixed a blank-CAD-on-open rendering bug, and made several heavyweight optional dependencies (`ospgrillage`, `openseespy`, `pythonocc-core`, `navcube`) import-safe so the application degrades gracefully rather than failing to launch. Together, these changes improve the accuracy, interactivity, and robustness of the OsdagBridge design workflow. This report describes the problem context, design decisions, implementation details, and impact of each contribution, and outlines directions for future work.

Contents

List of Figures

List of Tables

List of Abbreviations

CAD	Computer-Aided Design
FOSSEE	Free/Libre and Open Source Software for Education
IRC	Indian Roads Congress
OCC	Open CASCADE Technology (`pythonocc-core`)
PR	Pull Request
SFD/BMD	Shear Force Diagram / Bending Moment Diagram
UI/UX	User Interface / User Experience
VIT	Vellore Institute of Technology

Chapter 1

Introduction

1.1 About FOSSEE and Osdag

FOSSEE (Free/Libre and Open Source Software for Education) is a project of **IIT Bombay**, funded by the Ministry of Education, Government of India, that promotes the use of open-source software in academia and industry. **Osdag** (Open Source Design and Analysis of Steel Structures) is one of FOSSEE's flagship engineering tools, providing free, open, and standards-compliant design software for the structural engineering community.

OsdagBridge is a dedicated desktop module of the Osdag ecosystem, built with **PySide6**, that allows engineers to design, analyse, and visualise steel and composite bridge structures – including plate girder bridges – through an interactive graphical interface, live-load vehicle modelling, 2D cross-section CAD views, and 3D CAD generation.

1.2 Problem Statement

At the start of this internship, the OsdagBridge desktop application had two categories of gaps between its current state and its intended design:

1. The **Custom Vehicle dialog**, used to define live-load vehicles for bridge analysis, displayed only static `QLabel` placeholders (e.g. "Axle Layout Diagram") instead of the accurate, dimensioned diagrams specified in the platform's Figma designs.

2. The **cross-section CAD visualisation** had no way to visually confirm which girder was currently selected in the Member Properties → Girder Details table, the CAD view rendered blank until the window was resized, and several optional third-party dependencies were imported unconditionally – causing the entire application to fail to start on machines where those packages were not installed.

1.3 Objectives

- Design and implement spec-accurate, dynamically drawn vehicle load diagrams for the Custom Vehicle dialog.
- Implement real-time, synchronised girder highlighting across all cross-section

CAD views.

- Diagnose and fix the blank-CAD-on-open rendering bug.
- Harden the application against missing optional dependencies (`ospgrillage`, `openseespy`, `pythonocc-core`, `navcube`) so it degrades gracefully instead of crashing.
- Polish surrounding UI/UX issues encountered while implementing the above (scroll support, signal wiring, encoding bugs).

1.4 Scope of the Internship

This work was carried out as a **summer internship** with the **Osdag Project, FOSSEE, IIT Bombay**, under the guidance of **Nidhi Khare**, contributing directly to the `OsdagBridge` desktop application (`https://github.com/Aditya-Donde/OsdagBridge`). The work spanned understanding the existing PySide6-based codebase, replicating Figma design specifications in custom-drawn Qt widgets, debugging CAD rendering issues, and hardening the application against missing optional dependencies. All changes were submitted, reviewed, and merged through the standard GitHub pull-request workflow, primarily via:

- **PR #52** – *"feat: complete UI enhancements matching new specs"*
- **PR #66** – *"Member properties Dynamic girder fixes"*

1.5 Report Structure

Chapter 2 gives a brief overview of the OsdagBridge application architecture relevant to this work. Chapter 3 details the Custom Vehicle diagram implementation (PR #52). Chapter 4 details the girder-highlighting, CAD-rendering, and dependency-robustness work (PR #66). Chapter 5 lists the technical stack and files touched. Chapters 6–8 present the summary, conclusion, and future work, followed by references and an appendix of representative code listings.

Chapter 2

OsdagBridge Application Overview

Before describing the contributions, it is useful to outline how the relevant parts of OsdagBridge are structured.

- **Desktop UI layer** (`osdagbridge.desktop`) – Built with PySide6; contains dialogs (e.g. `CustomVehicleDialog`, `AdditionalInputs`), docks (e.g. `cad_cross_section.py`, `input_dock.py`), and reusable widgets/utilities (e.g. `custom_3dviewer.py`, `mpl_plot_widget.py`
- **Core analysis layer** (`osdagbridge.core`) – Contains bridge-type-specific analysis code, such as `plate_girder/analyser.py`, `analysis_results.py`, and `plot_generator.py`, which optionally depend on external structural-analysis packages.
- **Tab-based dialog composition** – Complex dialogs such as the Member Properties dialog are composed of multiple tabs (e.g. Section Properties, Girder Details, Typical Section Details), each emitting Qt signals so that changes in one tab (e.g. girder selection, cross-section parameters) can update live previews in sibling tabs.
- **Optional heavyweight dependencies** – Full structural analysis (`openseespy`, `ospgrillage`) and 3D CAD generation (`pythonocc-core`) rely on external packages that are non-trivial to install on some platforms, making them natural candidates for optional, guarded imports.

This architecture – signal-driven tab communication plus a layered core/UI split – directly shaped the implementation approach for both contributions described in the following chapters.

Chapter 3

Custom Vehicle Load Diagrams

(Pull Request #52 – "feat: complete UI enhancements matching new specs")

3.1 Problem

The **Custom Vehicle dialog** in OsdagBridge allows an engineer to define custom live-load vehicles for bridge analysis, entering axle loads and spacings. Prior to this contribution, the dialog only showed static `QLabel` placeholders (e.g. "Axle Layout Diagram", "Tracked Layout Diagram") with no actual visual representation of the vehicle geometry the user was configuring, making the input form harder to interpret correctly.

3.2 Design and Implementation

A new module, `custom_vehicle_diagrams.py`, was created containing three custom `QWidget` subclasses that render vehicle diagrams live using `QPainter`, plus a shared arrow-drawing helper.

3.2.1 Shared Arrow Helper

The function `draw_arrow_down()` draws a vertical force line terminated by a filled triangular arrowhead (built from a `QPolygonF`), and is reused by all three diagram widgets so that every force arrow across the dialog is visually consistent.

3.2.2 TrackedBogieDiagram

Draws force arrows, a dashed horizontal reference line, and a labelled distance dimension for **tracked vehicles** and **bogie cars**, switching its title text ("Tracked Vehicles" vs. "Bogie Cars") based on the supplied force label.

3.2.3 WheeledAxlesDiagram

Draws a full axle train $(P_1, P_2, \ldots, P_n)$ with per-axle force arrows, dimension lines with double-sided arrowheads, and ellipsis ("...") continuity markers so that long axle sequences can be represented compactly without drawing every intermediate axle.

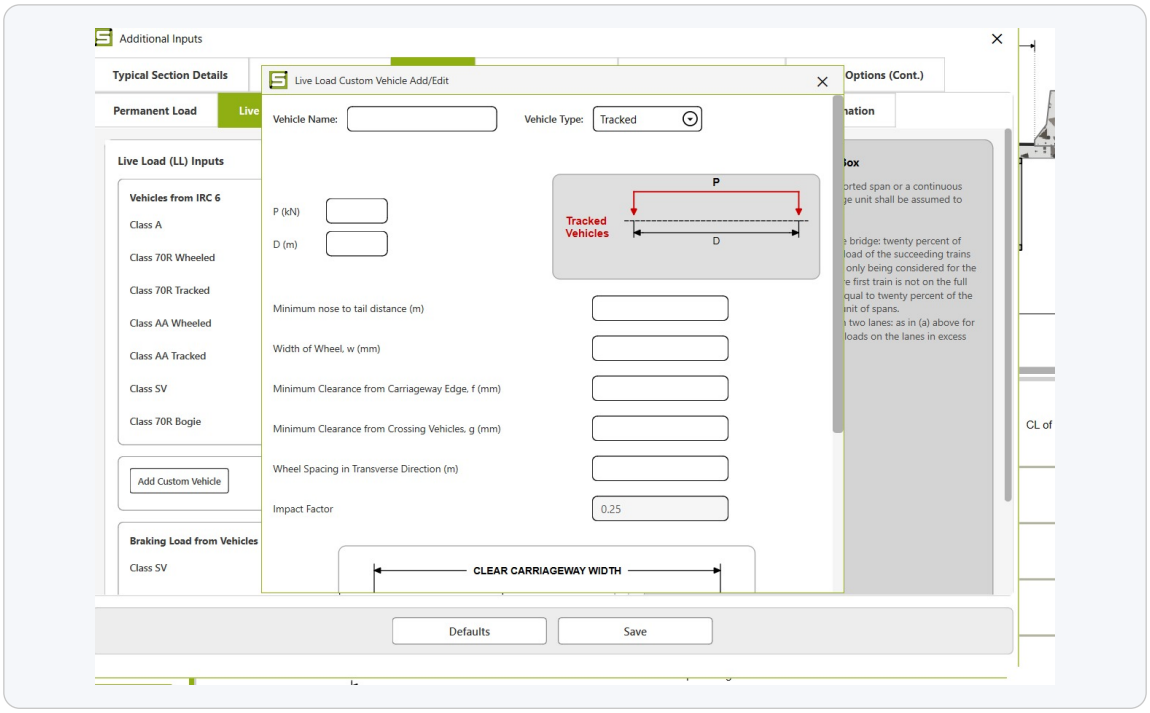

Figure 3.1: `TrackedBogieDiagram` – Tracked Vehicles variant, showing the single force P over distance D, rendered live via `QPainter` inside the Live Load Custom Vehicle Add/Edit dialog.

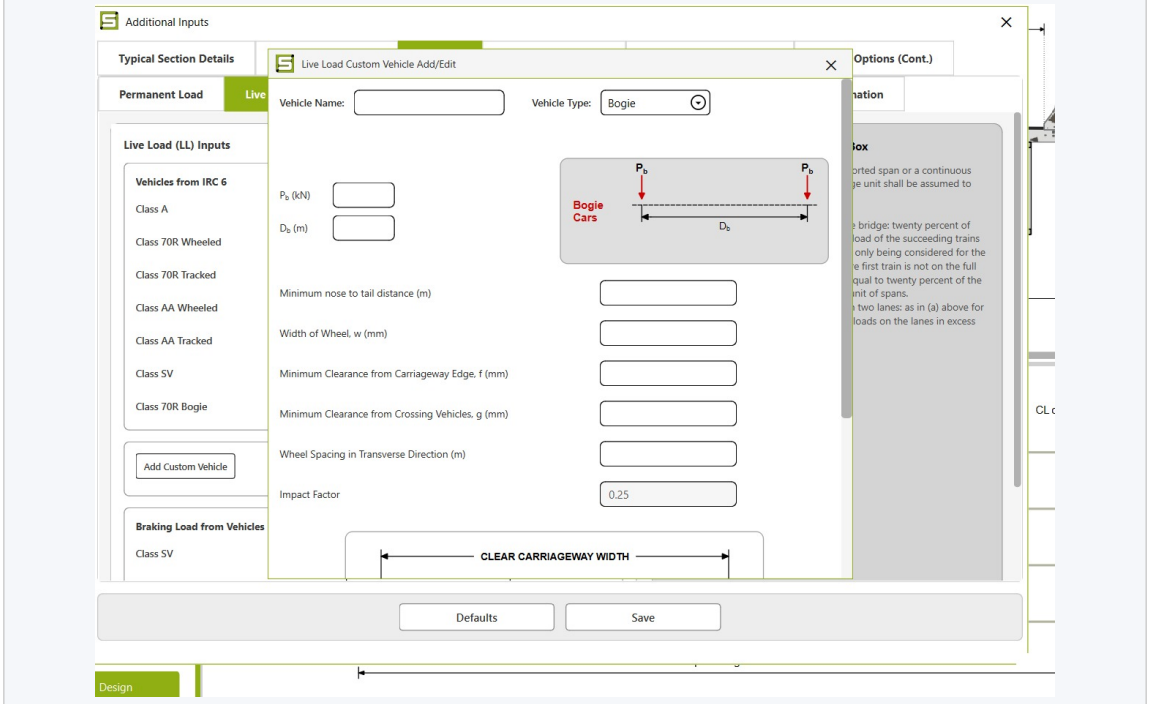

Figure 3.2: `TrackedBogieDiagram` – Bogie Cars variant, showing the P_b/D_b labelling used for bogie vehicles.

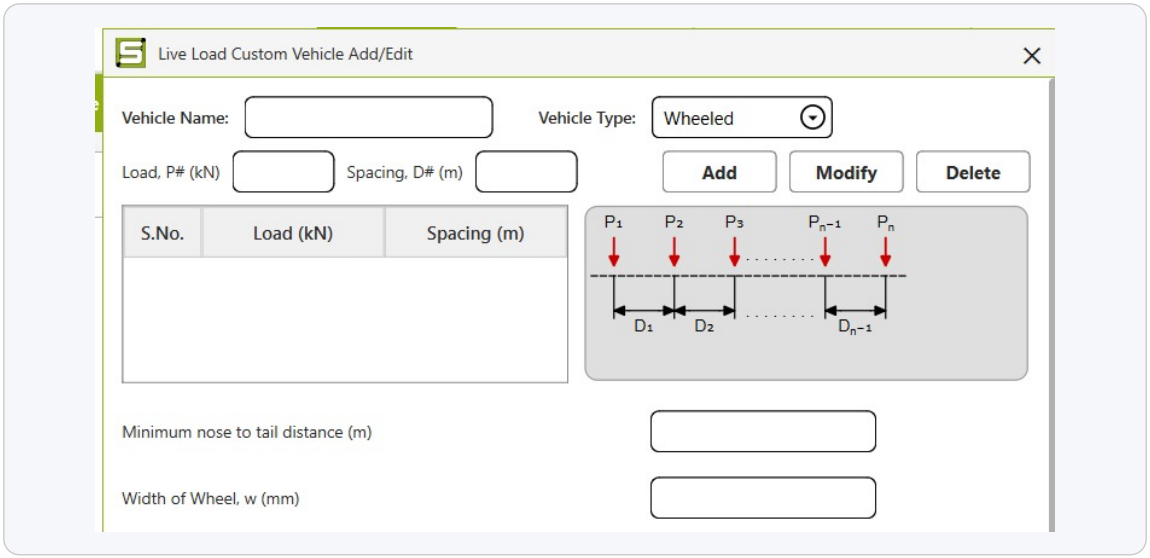

Figure 3.3: Wheeled Axles diagram: the axle load/spacing table (S.No, Load, Spacing) alongside the live-rendered $P_1 \ldots P_n$ diagram with Add/Modify/Delete controls.

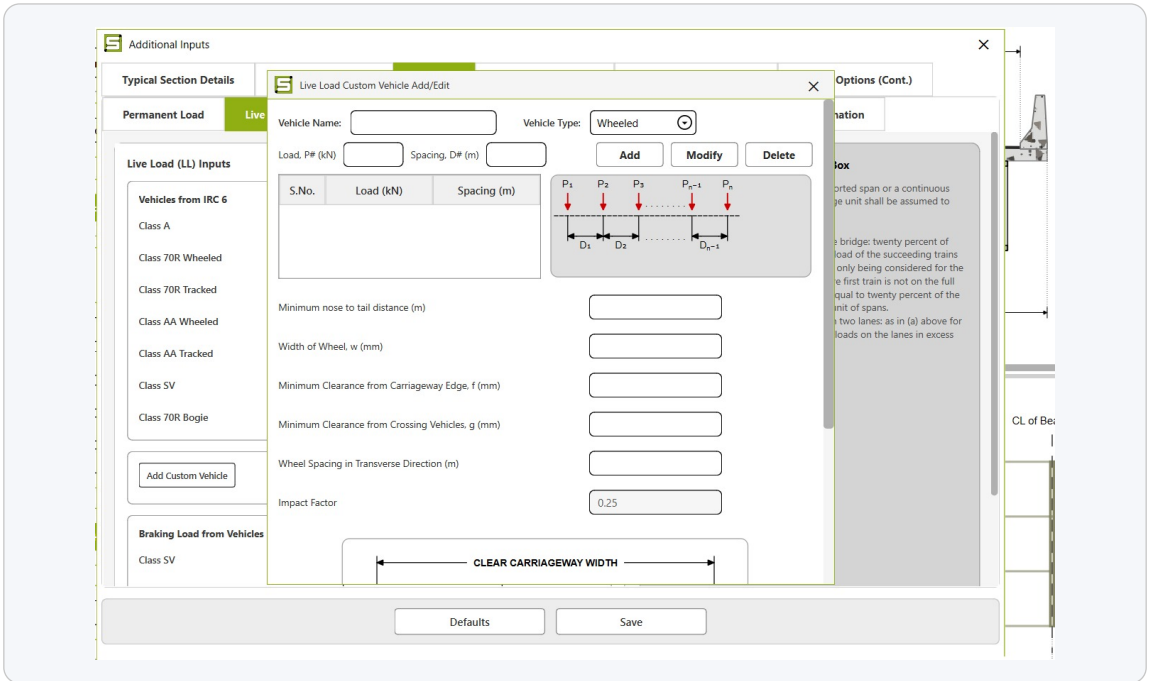

Figure 3.4: The full Live Load Custom Vehicle Add/Edit dialog (Wheeled type) as embedded within the scrollable Additional Inputs window.

3.2.4 ClearCarriagewayWidthDiagram

Draws the clear carriageway width diagram, including vehicle silhouettes, ground hatching, and labelled dimension spans (f, w, g) matching the Figma reference drawing, with a bold title rendered in the platform's accent colour.

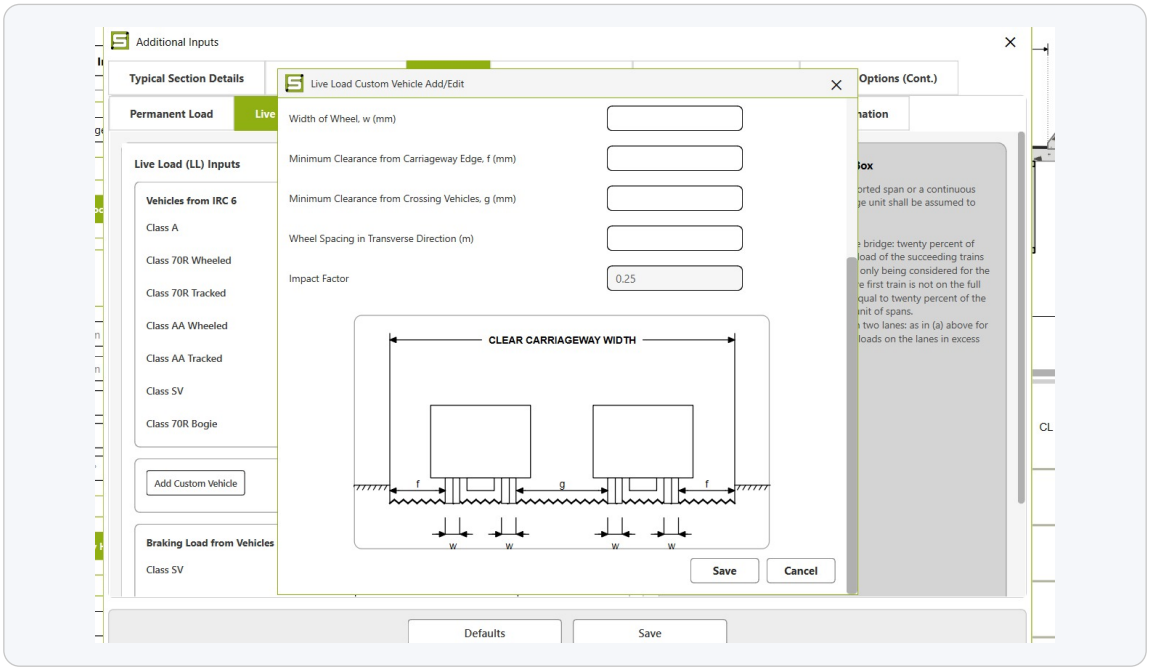

Figure 3.5: `ClearCarriagewayWidthDiagram` rendered inside the Live Load Custom Vehicle dialog, alongside the wheel-clearance (f, g, w) input fields.

3.2.5 Dialog Integration

The static placeholder `QLabel` widgets in `custom_vehicle_dialog.py` were replaced with live instances of these diagram widgets, and a `QStackedWidget` was introduced so the dialog can switch between the Tracked and Bogie diagrams based on the selected vehicle type without rebuilding the widget tree. Colours, fonts, and line styles (red force arrows, black dimension lines, dashed reference lines) were matched to the platform's Figma design specification.

3.3 Impact

Replacing static text placeholders with accurate, dynamically drawn diagrams gives engineers immediate visual confirmation of the vehicle geometry they are defining, reducing input errors and making the Custom Vehicle dialog self-explanatory without needing external reference documents. Because the diagrams are drawn procedurally rather than

as static images, they remain crisp at any dialog size and can later be parameterised to reflect the user's actual entered values.

Chapter 4

Dynamic Girder Highlighting and Robustness Fixes

(Pull Request #66 – "Member properties Dynamic girder fixes")

4.1 Dynamic Girder Highlighting

4.1.1 Problem

In a multi-girder bridge cross-section, there was previously no visual link between a row selected in the **Member Properties** → **Girder Details** table and the corresponding girder drawn in the CAD views, making it difficult to identify which physical girder a given row of properties referred to, especially as girder count increased.

4.1.2 Implementation

- A `highlighted_girders` list attribute was added to the cross-section widget, and the girder-drawing loop was updated to check whether each girder's identifier (by index, e.g. `"G{i+1}"`, or by label) is present in this list before choosing its fill colour.
- The I-section outline pen was updated to draw a bold, high-contrast width-3 border in **Osdag green** (`RGB(144, 175, 19)`) around the selected girder, instead of the default thin black outline, so the highlight is unambiguous at a glance.
- Girder selection now updates the `highlighted_girders` list and triggers a repaint across **all three** CAD views simultaneously: the sub-tab cross-section, the Typical Section Details preview, and the main window cross-section CAD.
- The **side-view selected segment highlight** was upgraded from a lighter overlay to a full Osdag-green fill with opacity 120 and a width-3 border, for visual consistency with the new cross-section highlighting.

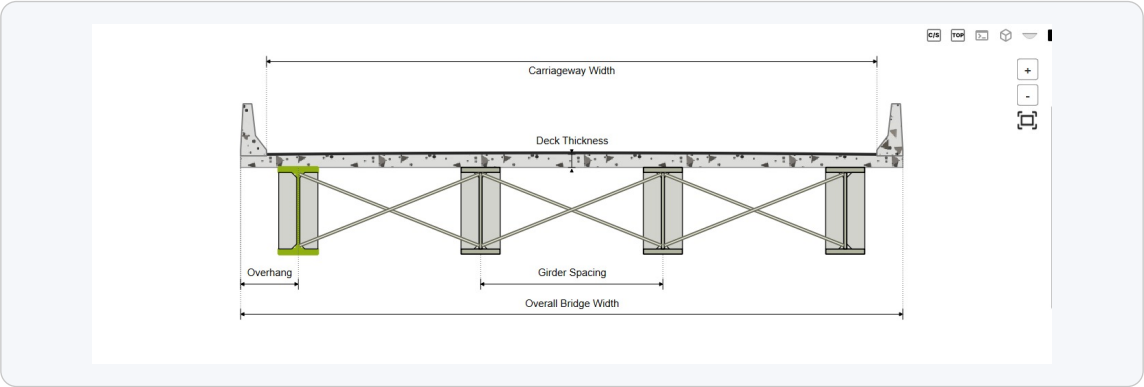

Figure 4.1: Selected girder highlighted in Osdag green in the cross-section CAD dock view, with Overhang, Girder Spacing, and Overall Bridge Width dimensions shown.

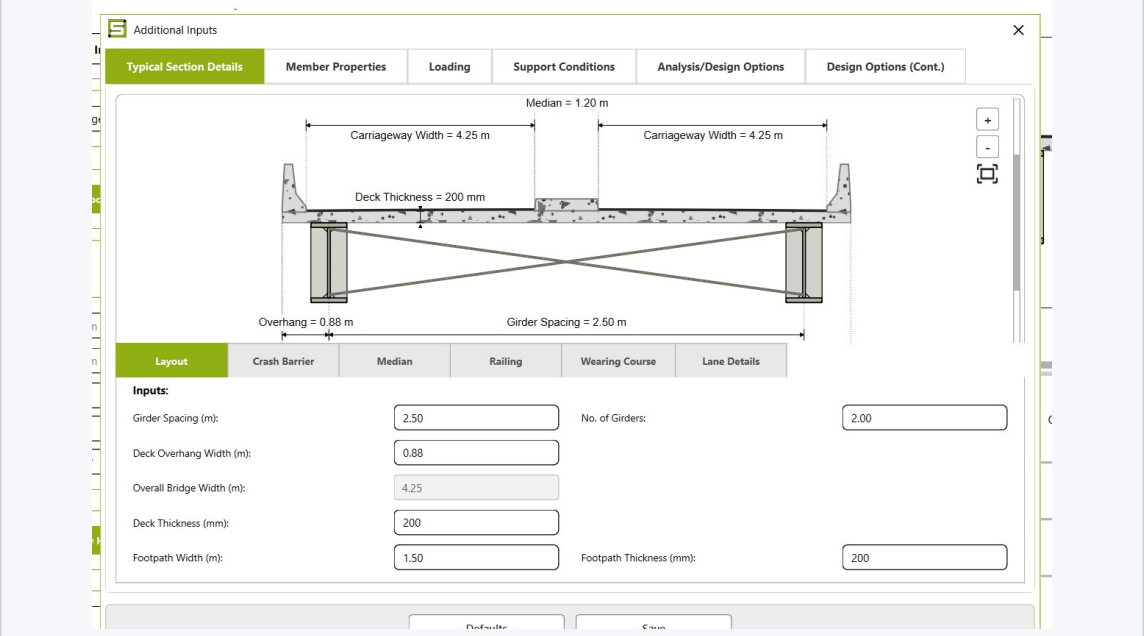

Figure 4.2: Cross-section preview shown in the Typical Section Details tab of the Additional Inputs dialog, driven by the same `cad_params_changed` signal used for girder highlighting.

4.2 Fixing the Blank CAD View on Open

4.2.1 Problem

The cross-section CAD would render **blank** when the Member Properties dialog was first opened, and only appeared correctly after the window was resized or minimized/restored – an unintuitive first impression for new users.

4.2.2 Root Cause and Fix

The CAD widget's viewport-fitting logic (`fit_to_screen()`) depended on the widget having a valid, finalised size, which was not yet available at construction time. This was fixed by overriding `showEvent()` on the CAD widget and scheduling a `fit_to_screen()` call via `QTimer.singleShot(120, ...)` shortly after the widget becomes visible, ensuring the viewport is fitted and the geometry is drawn immediately once the widget has a real size, without requiring any manual resize.

4.3 Optional Dependency Handling and Robustness

4.3.1 Problem

Several core and desktop modules in OsdagBridge directly imported heavyweight, optional third-party packages (`ospgrillage`, `openseespy`, `pythonocc-core` / `OCC`, and `navcube`) at module load time. If any of these were missing from a developer's or user's Python environment, the **entire application would fail to start**, even for features unrelated to that dependency.

4.3.2 Implementation

- In `analyser.py`, `analysis_results.py`, and `plot_generator.py`, the direct `import ospgrillage as og` / `import openseespy.opensees as ops` statements were wrapped in `try/except (ImportError, ModuleNotFoundError)` blocks, each setting a module-level flag (`HAS_OSP_GRILLAGE`, `HAS_OPENSEES`) that downstream code checks before using the corresponding functionality.
- In `cad_3d.py`, the `pythonocc-core` (`OCC`) import and the custom 3D viewer import were similarly guarded with a `HAS_OCC` flag. When `pythonocc-core` is unavailable, the `CAD3DWindow` now displays a clear, styled fallback message instead of crashing, and all CAD initialisation/render methods (`init_display`, `_is_display_ready`) short-circuit safely.

- In `mpl_plot_widget.py`, the `navcube` package import was wrapped with a `HAS_NAVCUBE` flag, and every NavCube-related method now checks this flag before touching the NavCube overlay, falling back to `self._navcube = None` when unavailable.
- In `custom_3dviewer.py`, a call to `self.navcube.mark_ready()` was guarded with `hasattr()` to avoid an `AttributeError` if the NavCube overlay was not fully initialised.

See Listing A.2 and Listing A.3 in Appendix A for representative code.

4.3.3 Impact

These changes let contributors and users run and develop core parts of OsdagBridge (2D cross-section views, live-load modelling, plate girder design) even without installing the full scientific/CAD stack (`openseespy`, `ospgrillage`, `pythonocc-core`), which can be difficult to install on some platforms. The application now fails gracefully feature-by-feature instead of refusing to launch entirely.

4.4 Additional UI/UX Fixes

- Added a `QScrollArea` wrapping the Custom Vehicle dialog's content widget, so the dialog remains usable on smaller screens or with many axle rows, instead of clipping content.
- Wired the `cad_params_changed` signal from the Typical Section Details tab to a new `update_cad_params()` slot on the Section Properties tab (and its Girder Details sub-tab), keeping the girder-details CAD preview in sync with cross-section parameter changes, including emitting the initial CAD state on dialog setup.
- Fixed the `AdditionalInputs` dialog constructor call to correctly pass the `parent` argument, resolving a parenting/ownership bug.
- Added two new optional cross-section rendering modes – `show_minimal_dimensions` (overhang and one girder-spacing dimension only, to reduce clutter) and `show_girder_spacing_o` (spacing between every adjacent girder pair) – controlled via boolean flags so different dialogs can request the dimensioning detail they need.
- Fixed several UI strings across `template_page.py` where Unicode arrow/chevron characters had been corrupted into mojibake due to an encoding mismatch, restoring the correct toggle-button glyphs and comment text.

Chapter 5

Technical Stack and Files Touched

5.1 Technologies Used

- **Python 3** – Primary implementation language for both the core analysis engine and the desktop UI.
- **PySide6 (Qt for Python)** – Used for the entire desktop application, including `QWidget`, `QPainter`, `QDialog`, `QStackedWidget`, and `QScrollArea` for the Custom Vehicle dialog and cross-section CAD views.
- **QPainter / QPen / QBrush / QPolygonF** – Used to hand-draw all custom vehicle diagrams and girder-highlight overlays directly onto widget canvases.
- **Matplotlib** – Used for SFD/BMD plot generation and the cross-section/3D plotting canvas in `mpl_plot_widget.py`.
- **ospgrillage** *(optional)* – Grillage analysis backend for plate girder bridges.
- **openseespy** *(optional)* – Structural analysis engine used by the analyser and analysis-results modules.
- **pythonocc-core (OCC)** *(optional)* – Used for 3D CAD generation and rendering of the bridge model.
- **navcube** *(optional)* – Provides the interactive 3D navigation cube overlay on plot/-CAD canvases.
- **Git & GitHub** – Version control and pull-request-based collaboration on the `Aditya-Donde/OsdagBridge` repository.

5.2 Key Files Touched

Table 5.1: Key files touched/added during this contribution

File	**Purpose**
`.../custom_vehicle_diagrams.py`	New module: `TrackedBogieDiagram`, `WheeledAxlesDiagram`, `ClearCarriagewayWidthDiagram` widgets
`.../custom_vehicle_dialog.py`	Custom Vehicle dialog – integrated live diagrams, added scroll area, fixed vehicle-type switching
`.../plate_girder/analyser.py`	Made `ospgrillage` import optional with `HAS_OSP_GRILLAGE` flag
`.../plate_girder/analysis_results.py`	Made `openseespy` import optional with `HAS_OPENSEES` flag
`.../plate_girder/plot_generator.py`	Made `openseespy` import optional
`.../section_properties/girder_details_tab.py`	Girder selection wired to CAD highlighting
`.../section_properties_tab.py`	Added `cad_params_changed` signal and `update_cad_params()`
`.../typical_section_details.py`	Emits CAD parameter change signal on preview update
`.../docks/cad_cross_section.py`	Girder highlighting logic, `showEvent` fix for blank CAD, minimal-dimension modes
`.../docks/input_dock.py`	Fixed `AdditionalInputs` dialog parenting
`.../utils/custom_3dviewer.py`	Guarded `OCC` import, added fallback "3D CAD Viewer Unavailable" message
`.../ui/cad_3d.py`	Guarded `OCC` import with `HAS_OCC` flag across CAD init/render methods
`.../ui/mpl_plot_widget.py`	Guarded `navcube` import with `HAS_NAVCUBE` flag
`.../ui/template_page.py`	Fixed corrupted Unicode chevron glyphs and comment mojibake

Chapter 6

Summary of Contributions

This internship contribution focused on making the **OsdagBridge** desktop application more visually accurate, more interactive, and more robust. The key outcomes were:

- **Custom Vehicle Diagrams**: Three spec-accurate, live-rendered `QPainter` diagrams (Tracked/Bogie, Wheeled Axles, Clear Carriageway Width) replacing static placeholder labels in the live-load vehicle dialog.
- **Dynamic Girder Highlighting**: Real-time, synchronised Osdag-green highlighting of the selected girder across all three cross-section CAD views.
- **Blank CAD View Fix**: Resolved a first-open rendering bug so the cross-section CAD now appears correctly without requiring a window-resize workaround.
- **Optional Dependency Hardening**: Made `ospgrillage`, `openseespy`, `pythonocc-core`, and `navcube` imports fail gracefully, keeping the core application usable even when these optional packages are not installed.
- **UI Polish**: Scroll-area support, CAD parameter signal wiring between tabs, and fixed corrupted Unicode UI glyphs.

Chapter 7

Conclusion

This internship has been a rewarding and technically enriching experience. Working on **OsdagBridge** gave me the opportunity to translate Figma design specifications into precise, hand-drawn Qt graphics, to reason about real-time state synchronisation across multiple CAD views, and to think carefully about software robustness when a project depends on several heavyweight, platform-sensitive scientific and CAD libraries.

I am sincerely grateful to my mentor, Nidhi Khare, and the Osdag/FOSSEE team at IIT Bombay for their constant guidance, code review, and encouragement throughout the internship. Their feedback was instrumental in shaping both the technical direction and the quality of my contributions.

I believe the improvements made during this internship – from accurate live-load vehicle diagrams to synchronised girder highlighting and safer dependency handling – will make OsdagBridge easier to use for practising engineers and easier to set up and contribute to for future developers.

Chapter 8

Future Work

Building on the foundation laid during this internship, several promising directions remain open for future work on OsdagBridge:

- **Extending girder highlighting to plots**: Reflect the selected girder highlight inside the SFD/BMD and other analysis plots generated by `plot_generator.py`, not just the cross-section CAD.
- **Graceful 3D fallback improvements**: Extend the "dependency unavailable" pattern used for `pythonocc-core` to offer an optional lightweight/alternate 3D preview when the full OCC-based viewer cannot be used.
- **Additional vehicle diagram types**: Extend the custom vehicle diagram library to cover more IRC-specified live-load vehicle configurations beyond tracked, bogie, and wheeled axle types.
- **Automated visual regression testing**: Add snapshot-based tests for the `QPainter`-based diagrams and CAD views to catch unintended rendering regressions as the UI evolves.
- **Packaging optional dependencies**: Provide clearer installation profiles (e.g. "core", "full-analysis", "full-cad") so users can install only the optional dependencies relevant to the features they need.

References

1. Osdag Project – https://osdag.fossee.in/
2. FOSSEE, IIT Bombay – https://fossee.in/
3. OsdagBridge Repository (GitHub) – https://github.com/Aditya-Donde/OsdagBridge
4. Pull Request #52 – "feat: complete UI enhancements matching new specs" – https://github.com/Aditya-Donde/OsdagBridge/pull/52
5. Pull Request #66 – "Member properties Dynamic girder fixes" – https://github.com/Aditya-Donde/OsdagBridge/pull/66
6. PySide6 (Qt for Python) Documentation – https://doc.qt.io/qtforpython/

Appendix A

Representative Code Listings

```
def draw_arrow_down(painter: QPainter, x: float, start_y: float,
    end_y: float):
    # Draw vertical line
    painter.drawLine(QPointF(x, start_y), QPointF(x, end_y))
    # Draw arrow head
    head_size = 6
    polygon = QPolygonF([
        QPointF(x, end_y),
        QPointF(x - head_size / 2, end_y - head_size),
        QPointF(x + head_size / 2, end_y - head_size)
    ])
    painter.setBrush(painter.pen().color())
    painter.drawPolygon(polygon)
```

Listing A.1: Shared arrow-drawing helper used by all vehicle diagrams.

```
HAS_OSP_GRILLAGE = True
try:
    import ospgrillage as og
except (ImportError, ModuleNotFoundError):
    HAS_OSP_GRILLAGE = False
```

Listing A.2: Guarded optional import pattern applied to `ospgrillage`.

```
if not HAS_OCC:
    self.layout = QVBoxLayout(self)
    self.layout.setAlignment(Qt.AlignCenter)

    self.label = QLabel(
        "<h3>3D CAD Viewer Unavailable</h3>"
        "<p>The 3D viewer requires <b>pythonocc-core</b>, "
        "which is not installed in your Python environment.</p>"
        "<p>To enable 3D viewing, please install it via conda:<br>
           "
        "<code>conda install -c conda-forge pythonocc-core=7.6.2</
           code></p>"
    )
```

```
12      self.label.setTextFormat(Qt.RichText)
13      self.label.setAlignment(Qt.AlignCenter)
14      self.layout.addWidget(self.label)
15      return
```

Listing A.3: Graceful fallback UI when `pythonocc-core` (OCC) is unavailable.

```
1  def showEvent(self, event):
2      """Fit diagram to viewport size once the widget becomes
           visible."""
3      super().showEvent(event)
4      QTimer.singleShot(120, self.fit_to_screen)
```

Listing A.4: Fix for the blank CAD view on first open, using a deferred `fit_to_screen()` call.