



# FOSSEE Winter Internship Report

On

## Parametric 3D CAD Modeling and Development of Steel Bridge Components in OsdagBridge

**Submitted by**

**Faizan Khan**

*3rd Year B.Tech Student, Department of Computer Engineering*

*Zakir Husain College of Engineering and Technology*

Aligarh Muslim University, Aligarh

**Under the Guidance of**

**Prof. Siddhartha Ghosh**

Department of Civil Engineering

Indian Institute of Technology Bombay

**Mentors:**

Nidhi Khare

Aditya Donde

Parth Karia

February 15, 2026

# Acknowledgments

I express my sincere gratitude to all who supported me during this internship at FOSSEE, IIT Bombay. I am deeply thankful to **Prof. Siddhartha Ghosh**, Principal Investigator of the Osdag project, Department of Civil Engineering, IIT Bombay, for providing this invaluable opportunity to work on cutting-edge structural engineering software development.

I am immensely grateful to my mentors **Nidhi Khare**, **Aditya Donde**, and **Parth Karia** for their patient guidance, technical expertise, and constant support. Their insights into pythonOCC, 3D modeling, and software development were invaluable in completing this project successfully. I also thank my fellow interns at FOSSEE for their collaboration and shared learning experiences.

I acknowledge the **National Mission on Education through ICT, Ministry of Education, Government of India** for funding this project. Finally, I am grateful to **Zakir Husain College of Engineering and Technology** and the **Department of Computer Engineering** for their encouragement in pursuing this internship opportunity.

**Faizan Khan**

*Department of Computer Engineering*

*Zakir Husain College of Engineering and Technology*

# Contents

# Chapter 1

# Introduction

## 1.1 National Mission in Education through ICT

The National Mission on Education through ICT (NMEICT) is a scheme under the Department of Higher Education, Ministry of Education, Government of India. It aims to leverage the potential of ICT to enhance teaching and learning in Higher Education Institutions in an anytime-anywhere mode.

The mission aligns with the three cardinal principles of the Education Policy—**access, equity, and quality**—by:

- Providing connectivity and affordable access devices for learners and institutions.
- Generating high-quality e-content free of cost.

NMEICT seeks to bridge the digital divide by empowering learners and teachers in urban and rural areas, fostering inclusivity in the knowledge economy. Key focus areas include:

- Development of e-learning pedagogies and virtual laboratories.
- Online testing, certification, and mentorship through accessible platforms like EduSAT and DTH.
- Training and empowering teachers to adopt ICT-based teaching methods.

For further details, visit the official website: www.nmeict.ac.in.

### 1.1.1 ICT Initiatives of MoE

The Ministry of Education (MoE) has launched several ICT initiatives aimed at students, researchers, and institutions. The table below summarizes the key details:

| No. | Resource | For Students/Researchers | For Institutions |
|---|---|---|---|
| **Audio-Video e-content** | | | |
| 1 | SWAYAM | Earn credit via online courses | Develop and host courses; accept credits |
| 2 | SWAYAMPRABHA | Access 24x7 TV programs | Enable SWAYAMPRABHA viewing facilities |
| **Digital Content Access** | | | |
| 3 | National Digital Library | Access e-content in multiple disciplines | List e-content; form NDL Clubs |
| 4 | e-PG Pathshala | Access free books and e-content | Host e-books |
| 5 | Shodhganga | Access Indian research theses | List institutional theses |
| 6 | e-ShodhSindhu | Access full-text e-resources | Access e-resources for institutions |
| **Hands-on Learning** | | | |
| 7 | e-Yantra | Hands-on embedded systems training | Create e-Yantra labs with IIT Bombay |
| 8 | FOSSEE | Volunteer for open-source software | Run labs with open-source software |
| 9 | Spoken Tutorial | Learn IT skills via tutorials | Provide self-learning IT content |
| 10 | Virtual Labs | Perform online experiments | Develop curriculum-based experiments |
| **E-Governance** | | | |
| 11 | SAMARTH ERP | Manage student lifecycle digitally | Enable institutional e-governance |
| **Tracking and Research Tools** | | | |
| 12 | VIDWAN | Register and access experts | Monitor faculty research outcomes |
| 13 | Shodh Shuddhi | Ensure plagiarism-free work | Improve research quality and reputation |
| 14 | Academic Bank of Credits | Store and transfer credits | Facilitate credit redemption |

Table 1.1: Summary of ICT Initiatives by the Ministry of Education

## 1.2 FOSSEE Project

The FOSSEE (Free/Libre and Open Source Software for Education) project promotes the use of FLOSS tools in academia and research. It is part of the National Mission on Education through Information and Communication Technology (NMEICT), Ministry of Education (MoE), Government of India.

### 1.2.1 Projects and Activities

The FOSSEE Project supports the use of various FLOSS tools to enhance education and research. Key activities include:

- **Textbook Companion**: Porting solved examples from textbooks using FLOSS.
- **Lab Migration**: Facilitating the migration of proprietary labs to FLOSS alternatives.
- **Niche Software Activities**: Specialized activities to promote niche software tools.
- **Forums**: Providing a collaborative space for users.
- **Workshops and Conferences**: Organizing events to train and inform users.

### 1.2.2 Fellowships

FOSSEE offers various internship and fellowship opportunities for students:

- Winter Internship
- Summer Fellowship
- Semester-Long Internship

Students from any degree and academic stage can apply for these internships. Selection is based on the completion of screening tasks involving programming, scientific computing, or data collection that benefit the FLOSS community. These tasks are designed to be completed within a week.

For more details, visit the official FOSSEE website.


Figure 1.1: FOSSEE Projects and Activities

## 1.3 Osdag Software

Osdag (Open steel design and graphics) is a cross-platform, free/libre and open-source software designed for the detailing and design of steel structures based on the Indian Standard IS 800:2007. It allows users to design steel connections, members, and systems through an interactive graphical user interface (GUI) and provides 3D visualizations of designed components. The software enables easy export of CAD models to drafting tools for construction/fabrication drawings, with optimized designs following industry best practices [1, 2, 3]. Built on Python and several Python-based FLOSS tools (e.g., PyQt and PythonOCC), Osdag is licensed under the GNU Lesser General Public License (LGPL) Version 3.

### 1.3.1 Osdag GUI

The Osdag GUI is designed to be user-friendly and interactive. It consists of

- **Input Dock**: Collects and validates user inputs.
- **Output Dock**: Displays design results after validation.
- **CAD Window**: Displays the 3D CAD model, where users can pan, zoom, and rotate the design.
- **Message Log**: Shows errors, warnings, and suggestions based on design checks.

### 1.3.2 Features

- **CAD Model**: The 3D CAD model is color-coded and can be saved in multiple formats such as IGS, STL, and STEP.
- **Design Preferences**: Customizes the design process, with advanced users able to set preferences for bolts, welds, and detailing.
- **Design Report**: Creates a detailed report in PDF format, summarizing all checks, calculations, and design details, including any discrepancies.

For more details, visit the official Osdag website.

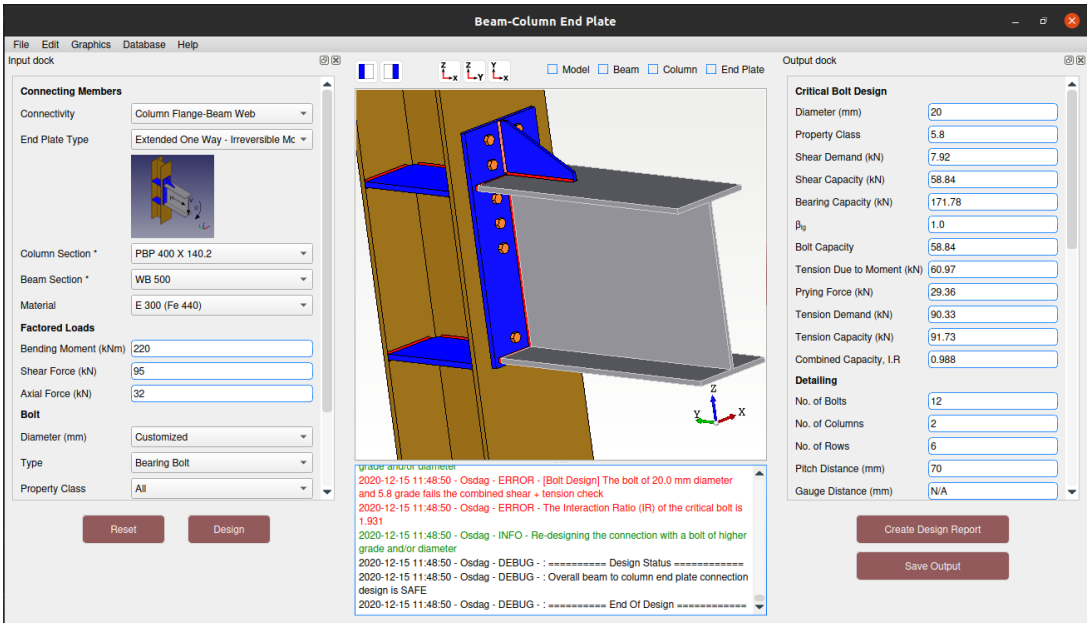



Figure 1.2: Osdag GUI

# Chapter 2

# Screening Task

## 2.1 Web-Based UI Development for Group Design Module

The screening task involved developing a web-based graphical user interface for the Osdag Bridge Module's Group Design feature. The objective was to create an application that handles user input, performs validation checks, manages field dependencies, and loads reference values from engineering data tables.

## 2.2 Tasks Done

### 2.2.1 Requirements Overview

The module required implementation of:

- Two-panel layout: Left panel for inputs, right panel for bridge cross-section reference image
- Basic Inputs tab with Type of Structure, Project Location, Geometric Details, and Material Inputs
- Additional Inputs tab (placeholder)
- Modify Additional Geometry pop-up with interdependent field calculations
- Comprehensive validation and error feedback

### 2.2.2 Frontend Development (React)

#### 2.2.2.1 Type of Structure Implementation

Implemented structure type selection with conditional logic that disables all fields when "Other" is selected. This ensures users can only proceed with "Highway Bridge" configuration, which is the currently supported design scope.

#### 2.2.2.2 Project Location - Dual Mode System

Two mutually exclusive modes were implemented:

**Mode 1: Enter Location Name**

- Hardcoded data for 5 major cities (Mumbai, Delhi, Chennai, Bangalore, Kolkata)
- Automatically displays Basic Wind Speed, Seismic Zone & Factor, and Temperature range
- State and district dropdown selection for location hierarchy
- Client-side lookup from preset database

**Mode 2: Tabulate Custom Loading Parameters**

- Spreadsheet-style interface for manual parameter entry
- User inputs wind speed, seismic zone and factor, and shade temperatures
- Persistent storage using browser localStorage
- Row selection to choose active parameter set

#### 2.2.2.3 Geometric Details Validation

Implemented real-time validation for all geometric parameters:

| Field | Validation Rule | Error Message |
|---|---|---|
| Span (m) | $20 \leq \text{span} \leq 45$ | "Outside the software range" |
| Carriageway Width (m) | $\geq 4.25$ and $< 24$ | Range violation message |
| Skew Angle (°) | $-15° \leq \theta \leq +15°$ | "IRC 24 (2010) requires detailed analysis" |
| Footpath | None / Single-sided / Both | Dropdown selection |

#### 2.2.2.4 Material Inputs

Dropdown selections for material grades were implemented with industry-standard options:

- Girder Steel: E250, E350, E450 (IS 2062)
- Cross Bracing Steel: E250, E350, E450 (IS 2062)
- Deck Concrete: M25, M30, M35, M40, M45, M50, M55, M60 (IS 456)

#### 2.2.2.5 Modify Additional Geometry Pop-Up

Implemented interdependent field calculations based on the constraint:

$$\frac{\text{Overall Width} - \text{Overhang}}{\text{Spacing}} = \text{No. of Girders} \tag{2.1}$$

where Overall Width = Carriageway Width + 5 meters.

**Features:**

- Auto-calculation: modifying any field automatically updates the other two
- Real-time constraint validation to ensure structural feasibility
- Circular update prevention using source field tracking
- Clear error messages for invalid configurations
- Minimum girder count enforcement (typically $\geq 3$)

### 2.2.3 Technology Stack

- **Frontend:** React 18.x with functional components and hooks
- **State Management:** React useState and useEffect hooks
- **Styling:** Custom CSS with responsive design
- **Development Environment:** Osdag-on-cloud platform
- **Version Control:** Git and GitHub

### 2.2.4 Challenges and Solutions

**Environment Setup:** Initial dependency conflicts were resolved by creating a clean virtual environment and consulting the project's Discord helpdesk for platform-specific configurations.

**Interdependent Calculations:** The primary challenge was implementing the three-way interdependent field calculation without causing infinite update loops. This was solved by tracking which field was the source of changes and only updating the dependent fields.

**State Management:** Managing complex state dependencies across multiple components required careful planning of data flow and prop passing to maintain synchronization between the UI and application state.

### 2.2.5 Application Screenshots

### 2.2.6 Deliverables

- **Video Demonstration:** `https://youtu.be/fcoeMBYUXuU?si=GA0wkkC060vZnuGP`

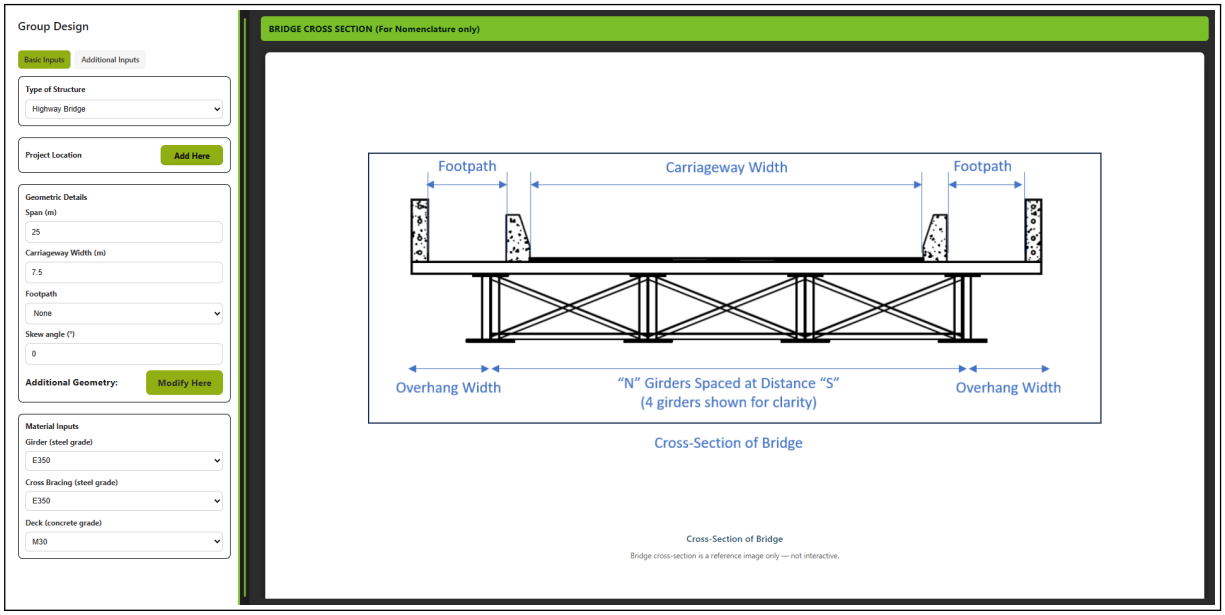



Figure 2.1: Osdag Bridge Group Design Module - Complete Interface

# Chapter 3

# Internship Task - 1: 3D CAD Modeling of Steel Girder Bridge

## 3.1 Problem Statement

Develop a parametric, modular 3D CAD model of a short-span steel girder bridge superstructure using the pythonOCC (pythonocc-core) library. The model must be assembled from configurable component factories including plate girders, deck slab, cross bracing, crash barriers, median barriers, and railings, with all parameters adjustable for different bridge configurations.

## 3.2 Objectives

- Implement a reusable, parameter-driven script for complete bridge superstructure assembly
- Create semi-transparent deck visualization to display internal steel reinforcement
- Deliver visual output using OpenCascade display with proper materials and colors
- Provide export capability to industry-standard STEP/BREP formats
- Ensure compliance with IRC:5-2015 specifications for all safety components

## 3.3 Component Design and Implementation

### 3.3.1 Plate Girder Component

The plate girder serves as the primary longitudinal load-carrying element, consisting of a vertical web connecting top and bottom flange plates to form an I-section that efficiently resists bending moments and shear forces.

#### 3.3.1.1 Geometry Generation Workflow

The girder geometry follows a five-stage parametric pipeline: web creation, flange positioning, stiffener distribution, support modeling, and global coordinate transformation. This modular approach enables rapid geometric updates through parameter modification.

**Generic Plate Creation** All girder elements are derived from a reusable rectangular plate generator based on face-extrusion workflow:

Listing 3.1: Generic Parametric Plate Creation

```
def _make_plate(origin, length, width, thickness, u_dir, w_dir):
    v_dir = np.cross(w_dir, u_dir)

    # Define corner points
    a1 = origin + (thickness/2)*u_dir + (length/2)*v_dir
    a2 = origin - (thickness/2)*u_dir + (length/2)*v_dir
    a3 = origin - (thickness/2)*u_dir - (length/2)*v_dir
    a4 = origin + (thickness/2)*u_dir - (length/2)*v_dir

    wire = _make_wire_from_points([gp_Pnt(*a1), gp_Pnt(*a2),
                                   gp_Pnt(*a3), gp_Pnt(*a4)])
    face = BRepBuilderAPI_MakeFace(wire).Face()
    return BRepPrimAPI_MakePrism(face, gp_Vec(*(width*w_dir))).
       Shape()
```

**I-Section Assembly** The complete girder section positions flanges with vertical offsets accounting for web depth and flange thickness:

Listing 3.2: Web and Flange Assembly

```
u_dir = np.array([0., 0., 1.])    # Vertical
w_dir = np.array([0., 1., 0.])    # Longitudinal

web = _make_plate(np.array([0.,0.,0.]), tw, length, D, u_dir,
   w_dir)
top_flange = _make_plate(np.array([0.,0.,(D + T_ft)/2]),
                         B_ft, length, T_ft, u_dir, w_dir)
bottom_flange = _make_plate(np.array([0.,0.,-(D + T_fb)/2]),
                            B_fb, length, T_fb, u_dir, w_dir)
```

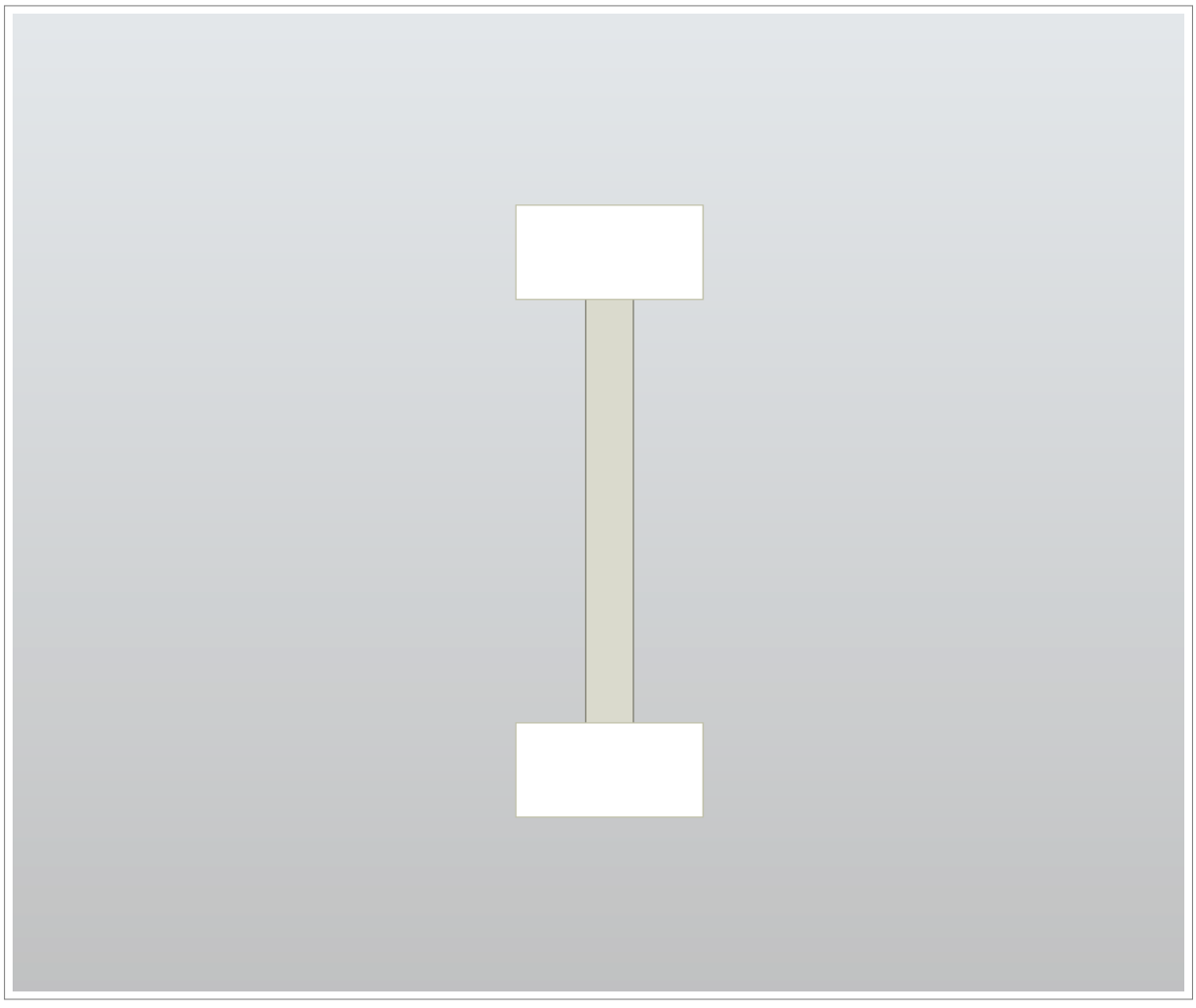


Figure 3.1: Plate girder I-section showing web and flange assembly

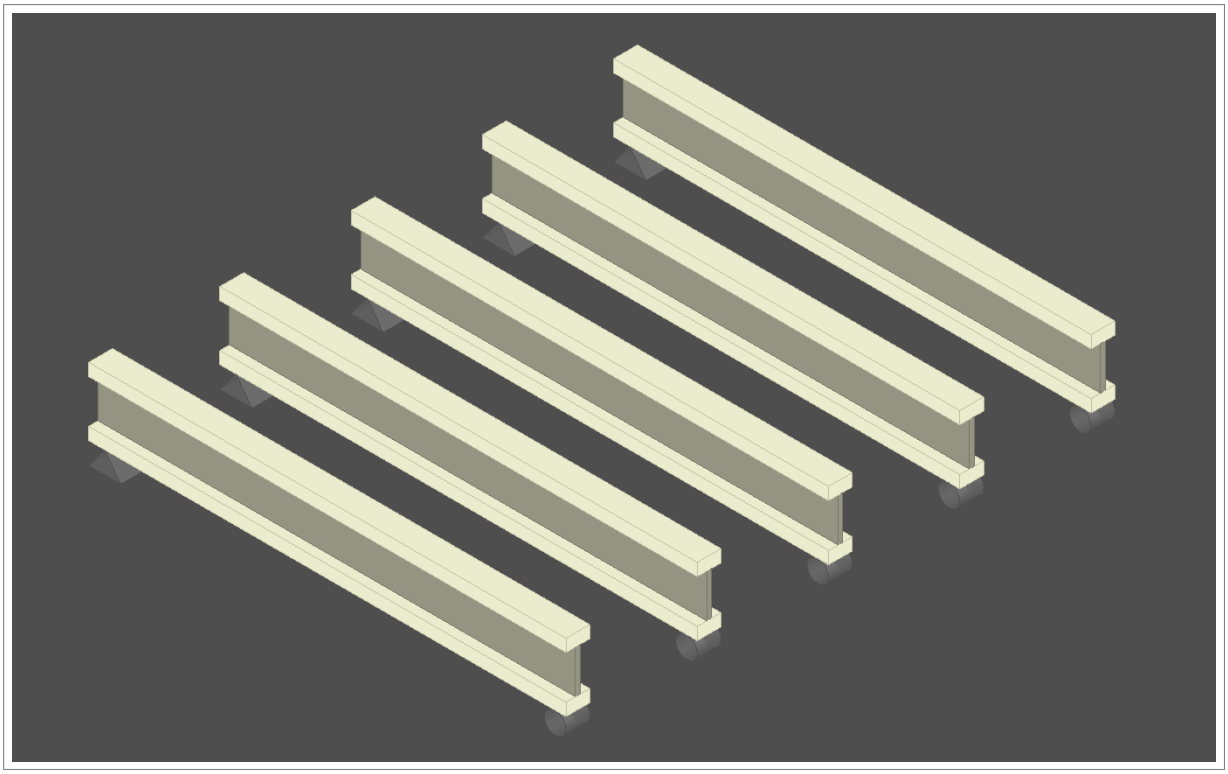


Figure 3.2: Plate girder I-section Iso-view

**Intermediate Stiffeners** Intermediate stiffeners use chamfered hexagonal profiles to prevent local web buckling. Each stiffener consists of two symmetric plates positioned on either side of the web.

Listing 3.3: Chamfered Hexagonal Stiffener Plate

```
def _create_stiffener_plate(position, width, height, thickness,
    chamfer, side):
    """Creates single stiffener plate with chamfered hexagonal
        profile."""
    x, y, z = map(float, position)
    c = chamfer

    # Define hexagonal profile points based on side
    if side == "right":
        pts = [gp_Pnt(0, 0, height/2 - c), gp_Pnt(c, 0, height/2)
            ,
               gp_Pnt(width, 0, height/2), gp_Pnt(width, 0, -
                   height/2),
```

```
            gp_Pnt(c, 0, -height/2), gp_Pnt(0, 0, -height/2 +
                c)]
    else:  # left (mirrored)
        pts = [gp_Pnt(0, 0, height/2 - c), gp_Pnt(0, 0, -height/2
            + c),
               gp_Pnt(-c, 0, -height/2), gp_Pnt(-width, 0, -
                   height/2),
               gp_Pnt(-width, 0, height/2), gp_Pnt(-c, 0, height
                   /2)]

    # Create wire, extrude to solid, and position
    wire = _make_wire_from_points(pts)
    face = BRepBuilderAPI_MakeFace(wire).Face()
    solid = BRepPrimAPI_MakePrism(face, gp_Vec(0, thickness, 0)).
       Shape()

    return _translate_to_position(solid, x, y, z)
```

The chamfered corners (typically 20-30 mm) reduce stress concentration and facilitate welding operations.

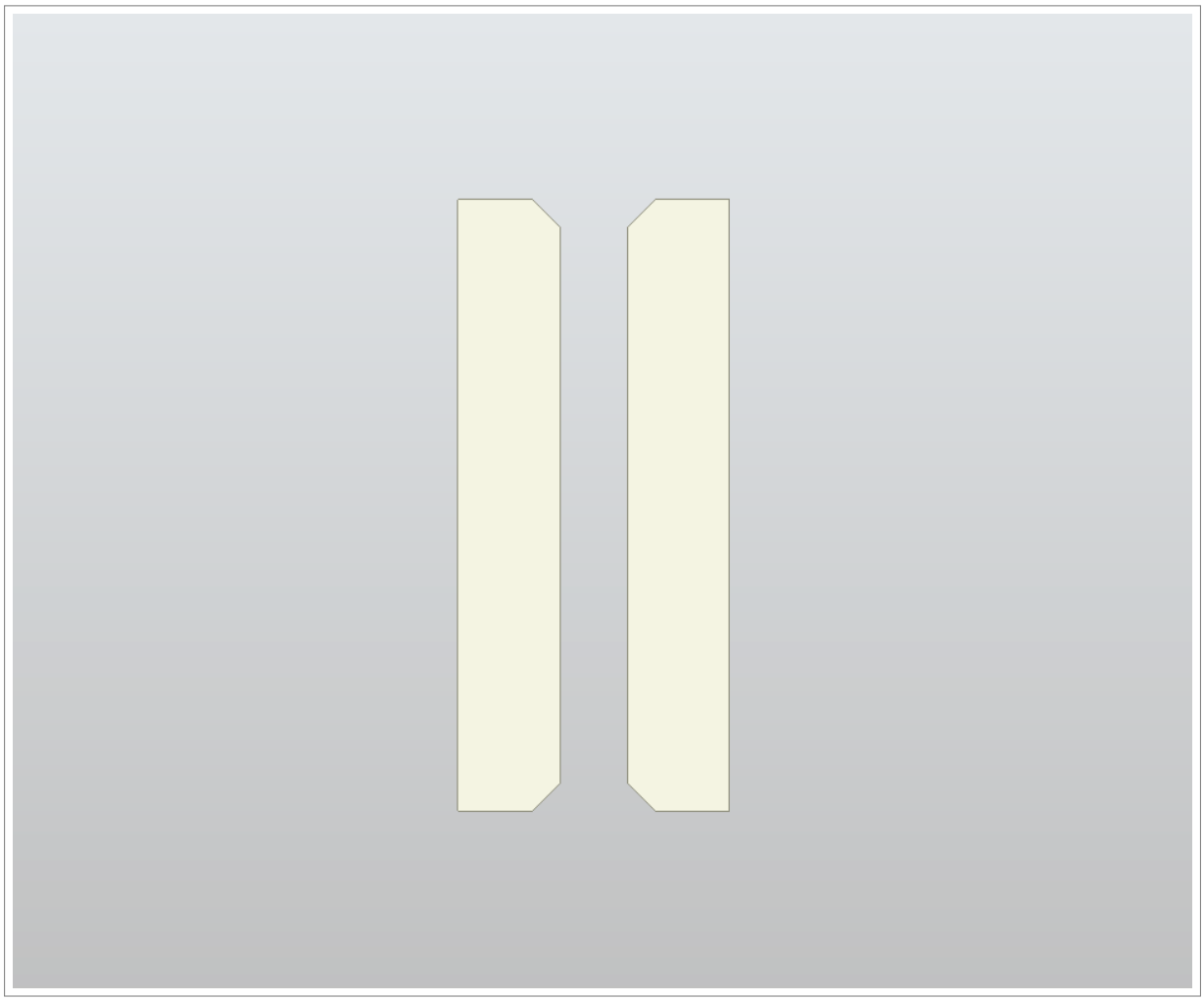


Figure 3.3: Stiffeners Front-view

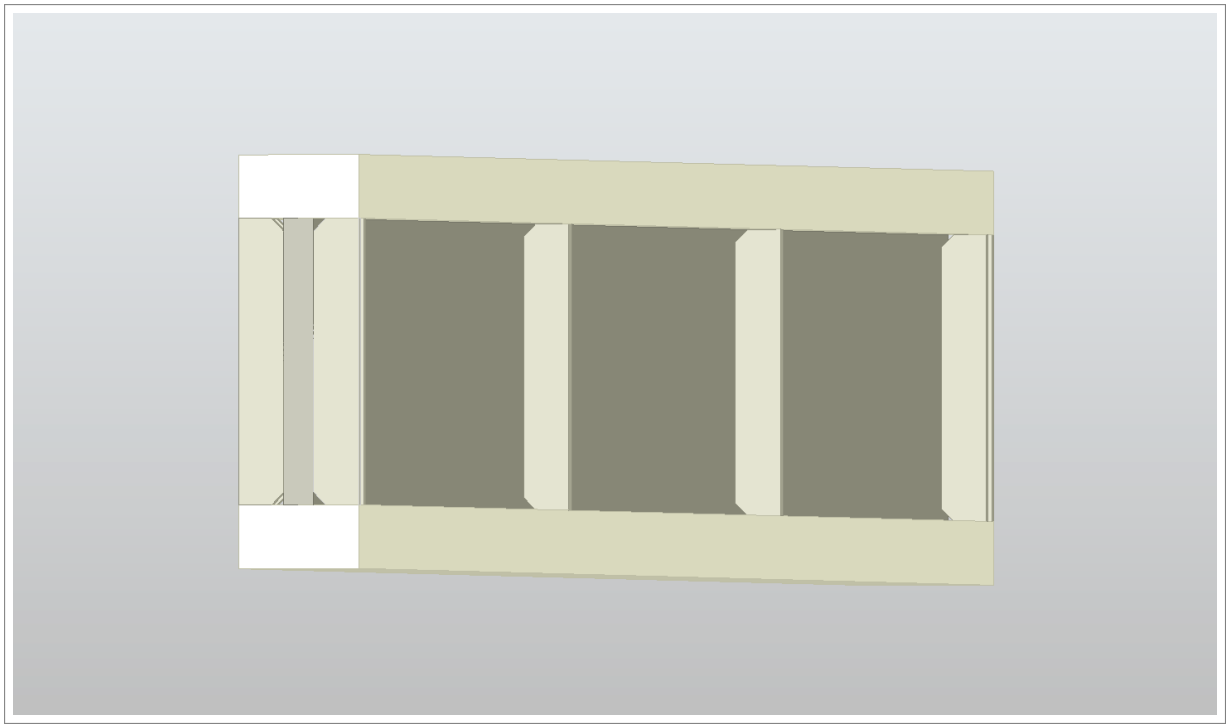


Figure 3.4: Stiffeners longitudional-view

**Support Modeling** Triangular roller supports (allowing thermal expansion) and cylindrical pin supports (providing restraint) simulate realistic bearing conditions:

Listing 3.4: Support Geometry Creation

```
# Triangular roller support (left end)
tri_height, tri_base = 200, 250
tri_pts = [
    gp_Pnt(0, -tri_base/2, -girder_depth/2 - tri_height),
    gp_Pnt(0, tri_base/2, -girder_depth/2 - tri_height),
    gp_Pnt(0, 0, -girder_depth/2)
]
tri_face = BRepBuilderAPI_MakeFace(make_wire_from_points(tri_pts)
   ).Face()
tri_support = BRepPrimAPI_MakePrism(tri_face, gp_Vec(
   support_width,0,0)).Shape()

# Cylindrical pin support (right end)
r_cyl = 100
cyl_support = BRepPrimAPI_MakeCylinder(
```

```
    gp_Ax2(gp_Pnt(0, 0, -girder_depth/2 - r_cyl), gp_Dir(1,0,0)),
    r_cyl, support_width
).Shape()
```

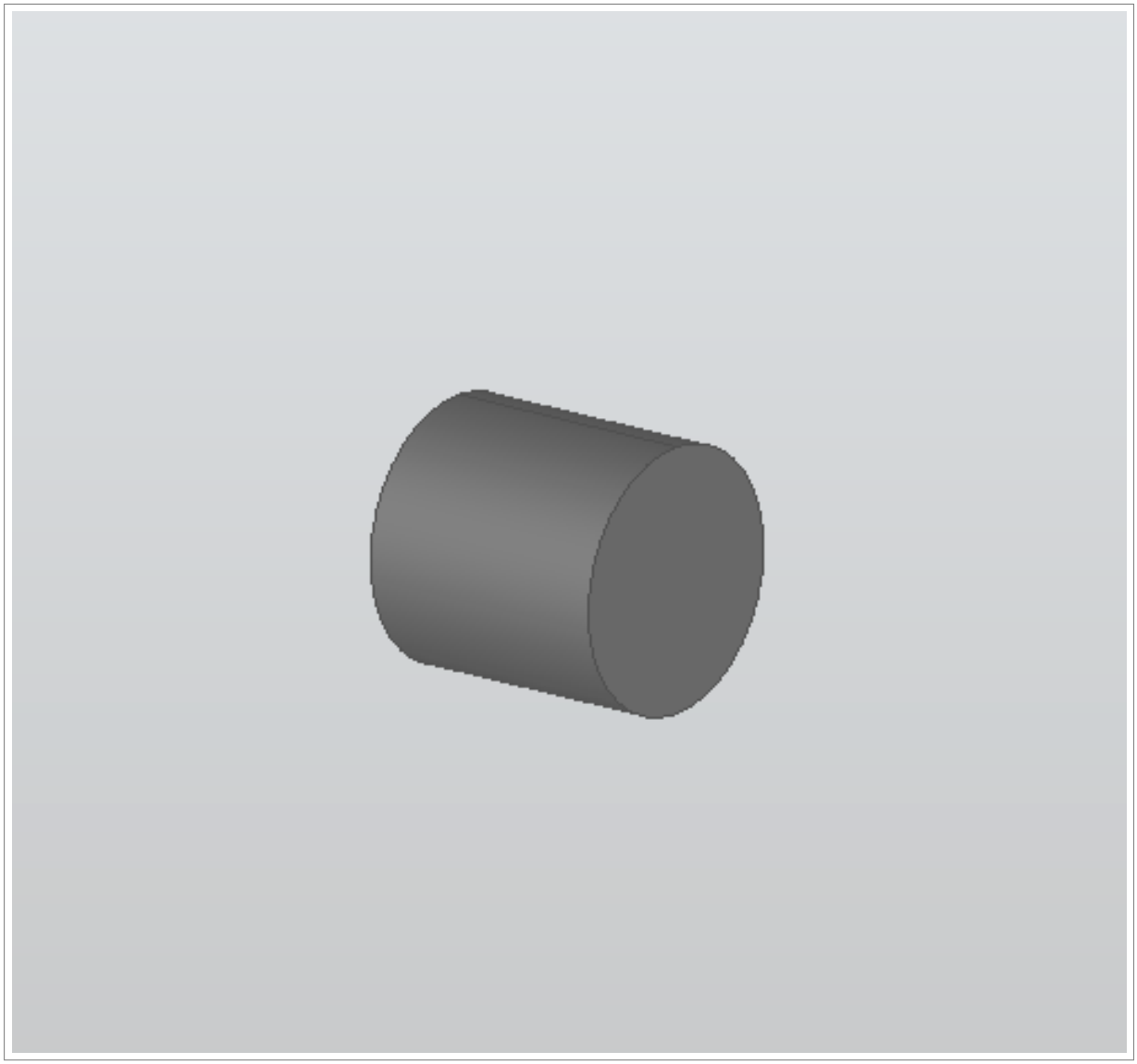


Figure 3.5: Cylindrical support

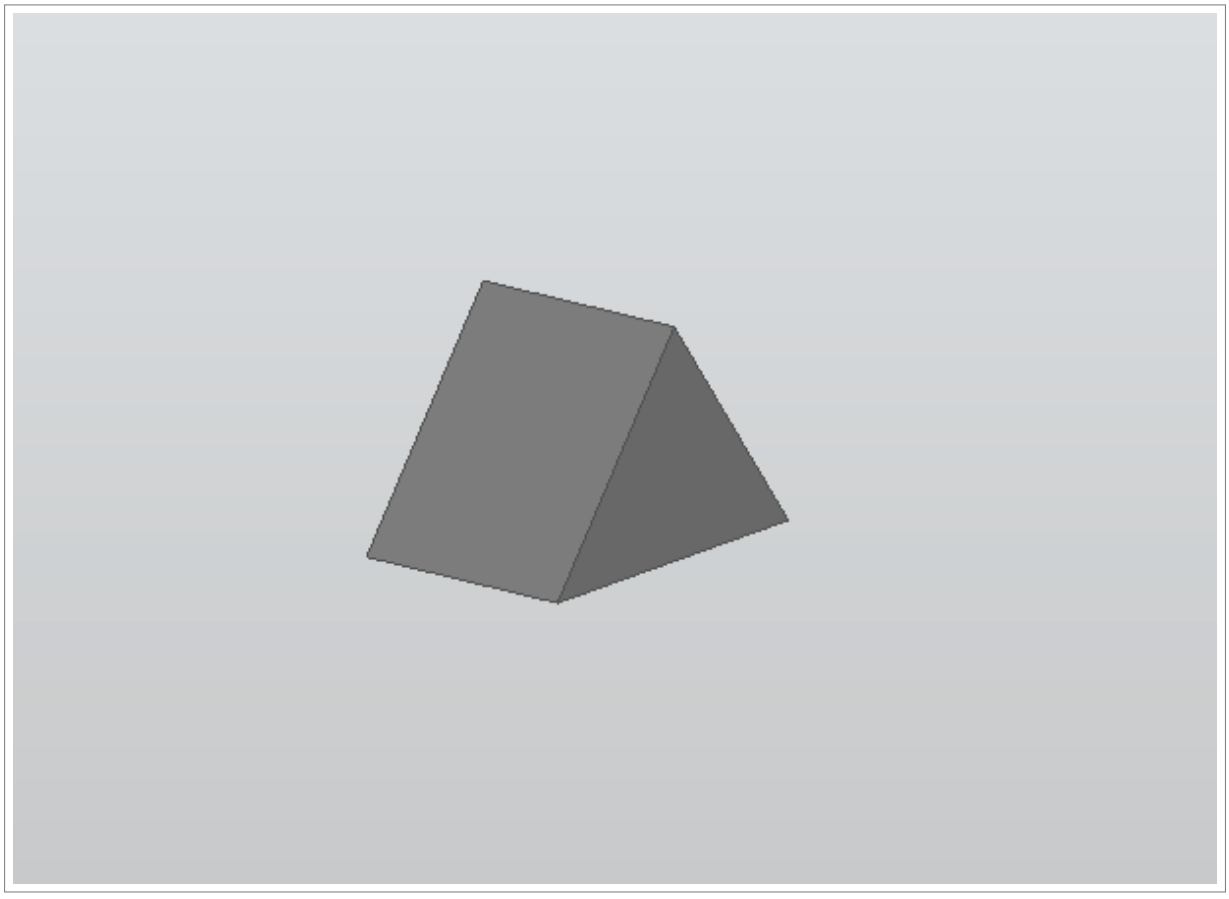


Figure 3.6: Triangular Support

**Skew Girder Modeling** To represent bridges with non-orthogonal alignment relative to the support axis, skew geometry was incorporated into the plate girder model. Skew bridges are commonly used where the roadway crosses an obstacle at an oblique angle, resulting in inclined girder ends rather than perpendicular terminations.

The skew configuration was implemented by applying a rotational transformation to the generated girder components about the global vertical axis. This transformation preserves the geometric integrity of individual components while altering the overall orientation of the girder assembly, enabling realistic representation of skew bridge behavior in the CAD model.

Listing 3.5: Skew Transformation of Plate Girder Geometry

```
from OCC.Core.gp import gp_Trsf, gp_Ax1, gp_Dir, gp_Pnt
from OCC.Core.BRepBuilderAPI import BRepBuilderAPI_Transform
import math

def apply_skew(shape, skew_angle_deg):
    """
    Rotates geometry about global Z-axis to create skew.
    """
    trsf = gp_Trsf()

    # rotation about vertical axis
    axis = gp_Ax1(gp_Pnt(0, 0, 0), gp_Dir(0, 0, 1))

    trsf.SetRotation(axis, math.radians(skew_angle_deg))

    return BRepBuilderAPI_Transform(shape, trsf, True).Shape()


# Example: apply skew to girder components
skew_angle = 20  # degrees

skewed_web = apply_skew(web, skew_angle)
skewed_flange = apply_skew(top_flange, skew_angle)
```

**Complete Plate Girder Assembly** The complete plate girder model is obtained by integrating the web, flange plates, intermediate stiffeners, and support components within a unified parametric framework. All elements are positioned using a consistent global coordinate system to ensure geometric accuracy and structural continuity. The assembled three-dimensional model provides a realistic representation of the girder configuration while retaining full parametric adaptability for design updates.

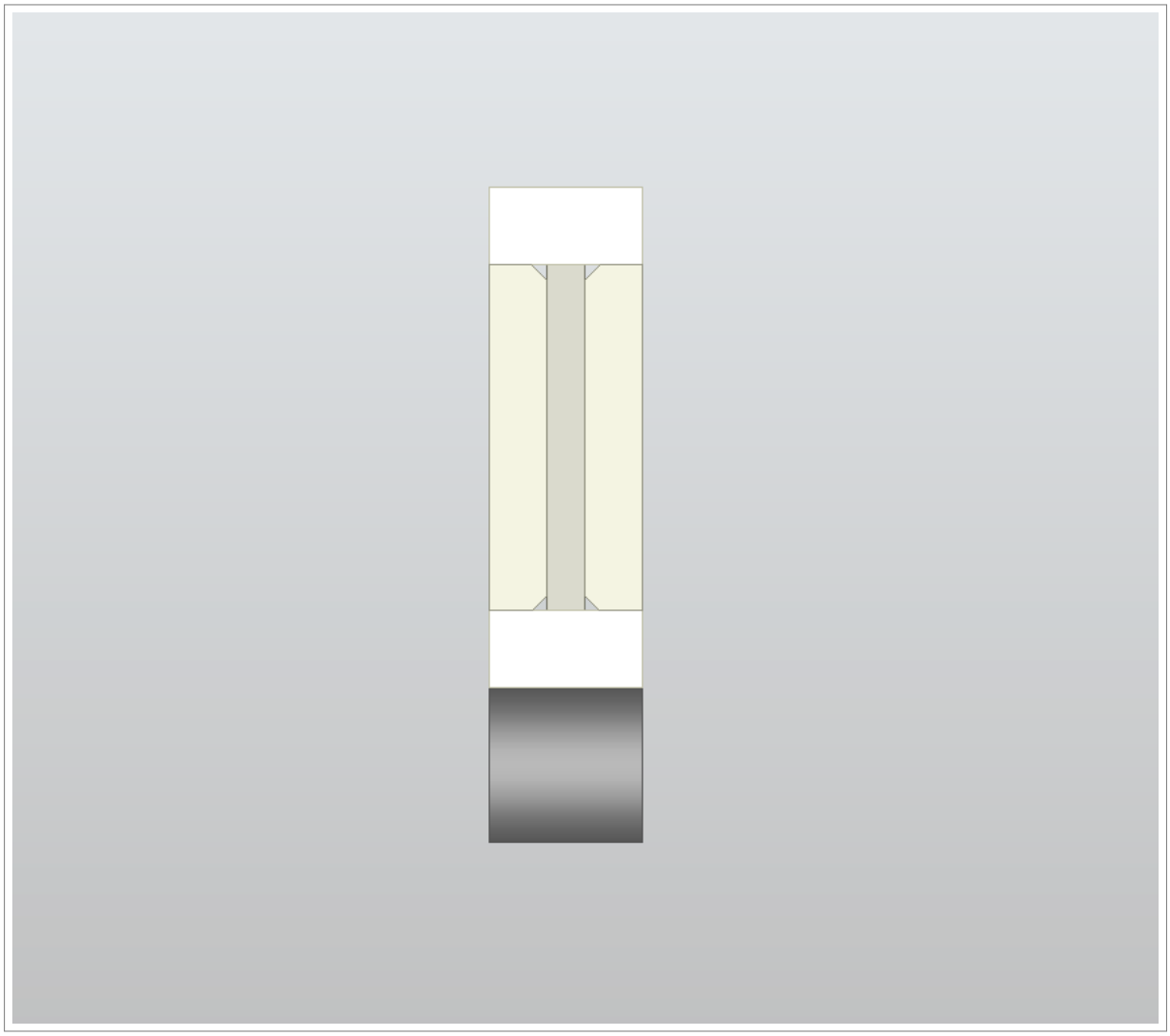


Figure 3.7: Plate Girder Front View

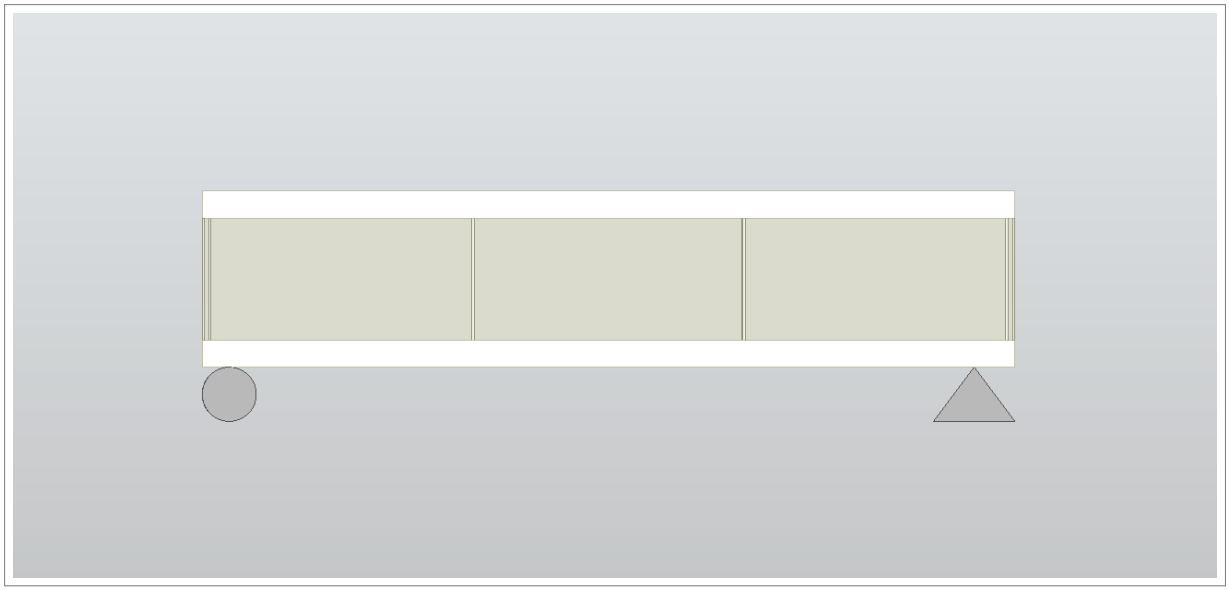


Figure 3.8: Complete plate girder assembly with supports

### 3.3.2 Deck Component

The deck is a horizontal concrete slab spanning transversely across girders, transferring vehicle loads to the supporting structure. Deck width is calculated dynamically based on carriageway requirements, crash barriers, footpaths, and railings.

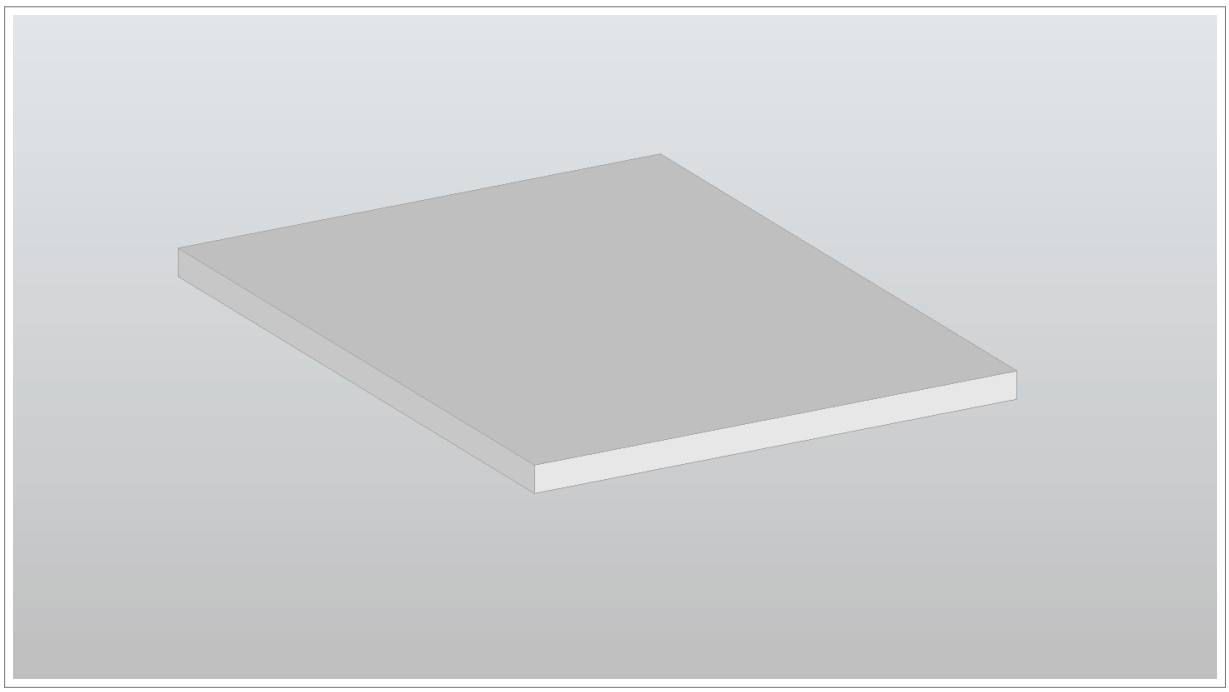


Figure 3.9: Deck

#### 3.3.2.1 Deck Width Calculation

Total deck width varies based on footpath configuration:

Listing 3.6: Dynamic Deck Width Calculation

```
def calculate_deck_width(*, footpath_config, carriageway_width,
                         crash_barrier_base_width, footpath_width,
                             railing_width):
    if footpath_config == "NONE":
        return carriageway_width + 2 * crash_barrier_base_width
    elif footpath_config in ("LEFT", "RIGHT"):
        return (carriageway_width + 2 * crash_barrier_base_width
                + footpath_width + railing_width)
    elif footpath_config == "BOTH":
        return (carriageway_width + 2 * crash_barrier_base_width
                + 2 * footpath_width + 2 * railing_width)
```

#### 3.3.2.2 Carriageway Center Positioning

Carriageway centerline offset depends on footpath configuration:

Listing 3.7: Carriageway Center Calculation

```
def calculate_carriageway_center_y(*, footpath_config,
   total_deck_width,
                                   carriageway_width,
                                      crash_barrier_base_width,
                                   footpath_width, railing_width)
                                      :
    # Start from LEFT edge of deck
    y = -total_deck_width / 2

    # Left crash barrier always exists
    y += crash_barrier_base_width

    # Add left footpath + railing if present
    if footpath_config in ("LEFT", "BOTH"):
```

```
        y += footpath_width + railing_width

    # Move to center of carriageway
    y += carriageway_width / 2
    return y
```

#### 3.3.2.3 Skewed Deck Geometry

For skewed bridges, deck edges are angled to the girder centerline, requiring parallelogram footprints. The X-coordinate shifts based on Y-distance from centerline: `x_shift = y_local * tan(skew_angle)`.

Listing 3.8: Skewed Deck Slab Creation

```
skew_rad = math.radians(skew_angle)
y_half = deck_width / 2

# Calculate X shifts for edges
shift_top = y_half * math.tan(skew_rad)
shift_bottom = -y_half * math.tan(skew_rad)

# Define parallelogram vertices
p1 = gp_Pnt(shift_bottom, -y_half, 0)
p2 = gp_Pnt(span_length + shift_bottom, -y_half, 0)
p3 = gp_Pnt(span_length + shift_top, y_half, 0)
p4 = gp_Pnt(shift_top, y_half, 0)

# Create and extrude face
poly = BRepBuilderAPI_MakePolygon()
for p in (p1, p2, p3, p4): poly.Add(p)
poly.Close()

slab = BRepPrimAPI_MakePrism(BRepBuilderAPI_MakeFace(poly.Wire())
   .Face(),
                             gp_Vec(0, 0, deck_thickness)).Shape
                                ()
```

#### 3.3.2.4 Surface Texture

Realistic concrete appearance is achieved using randomly distributed cylindrical dots and triangular patterns on all six faces. Different faces use appropriately oriented primitives:

Listing 3.9: Deck Texture Primitive Creation and Distribution

```
# Texture constants
DOT_RADIUS, DOT_HEIGHT = 6, 3
TRI_SIZE_MAX, TRI_HEIGHT = 120, 0.4

def _create_dot_z(x, y, z, up=True):
    """Create vertical cylinder for top/bottom face texture."""
    h = DOT_HEIGHT if up else -DOT_HEIGHT
    dot = BRepPrimAPI_MakeCylinder(DOT_RADIUS, abs(h)).Shape()
    return _translate(dot, x, y, z)

def _create_triangle_xy(x, y, z, up=True):
    """Create triangular prism for top/bottom face texture."""
    size = random.uniform(80, TRI_SIZE_MAX)
    p1 = gp_Pnt(0, 0, 0)
    p2 = gp_Pnt(size, random.uniform(10, size), 0)
    p3 = gp_Pnt(random.uniform(10, size), size, 0)

    poly = BRepBuilderAPI_MakePolygon()
    for p in (p1, p2, p3): poly.Add(p)
    poly.Close()

    face = BRepBuilderAPI_MakeFace(poly.Wire()).Face()
    h = TRI_HEIGHT if up else -TRI_HEIGHT
    prism = BRepPrimAPI_MakePrism(face, gp_Vec(0, 0, h)).Shape()
    return _translate(prism, x - size / 2, y - size / 2, z)

# Front/Back faces: 120 dots + 15 triangles each
```

```
# Left/Right faces: 100 dots + 20 triangles each
# Top/Bottom faces: 150 dots + 50 triangles each
```

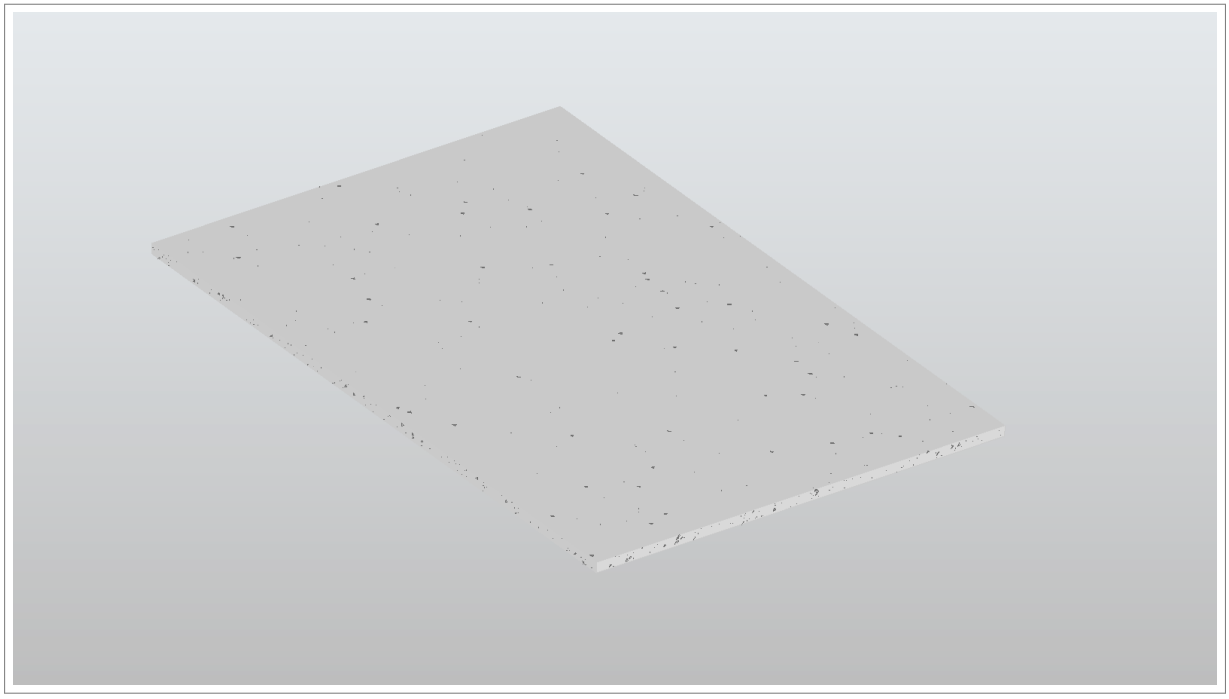


Figure 3.10: Deck with texture

#### 3.3.2.5 Deck–Girder Assembly

The deck slab is positioned above the plate girders at the top flange level to enable effective load transfer to the supporting steel members. Its vertical location is determined by the girder depth and flange thickness, ensuring proper structural alignment. This assembly represents the interaction between the concrete deck and steel girder system, forming a continuous superstructure while maintaining compatibility with girder spacing and skew geometry.

Figure 3.11: Deck slab positioned on plate girders showing complete deck–girder assembly - Front View

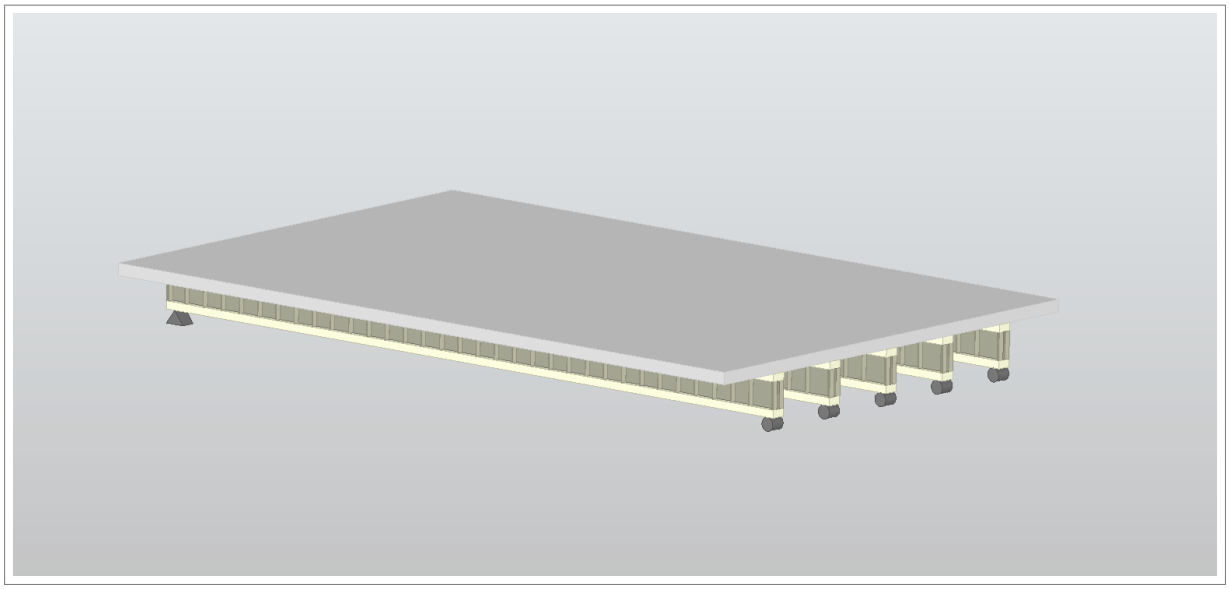


Figure 3.12: Deck slab positioned on plate girders showing complete deck–girder assembly - Side View

### 3.3.3 Cross Bracing Component

Cross bracing connects adjacent girders to provide lateral stability and maintain structural alignment. The system is generated parametrically by creating section profiles, forming diagonal members, and assembling bracing configurations between girders.

#### 3.3.3.1 Section Geometry Creation

Cross bracing members are modeled using standard steel sections such as angle, channel, double angle, double channel, and I-sections. These section profiles are generated using primitive solids and Boolean operations.

Listing 3.10: Section Geometry Creation

```
def _create_section_solid(section_type, length, thickness, dims):
    if section_type == "ANGLE":
        return _create_angle_section(length, h, w, thickness)

    if section_type == "CHANNEL":
        return _create_channel_section(length, d, wf, tw, tf)

    if section_type == "DOUBLE_ANGLE":
        return _create_double_angle_section(length, h, w,
           thickness)

    if section_type == "DOUBLE_CHANNEL":
        return _create_double_channel_section(length, d, wf, tw,
           tf)

    if section_type == "I_SECTION":
        return _create_i_section(length, d, wf, tw, tf)
```

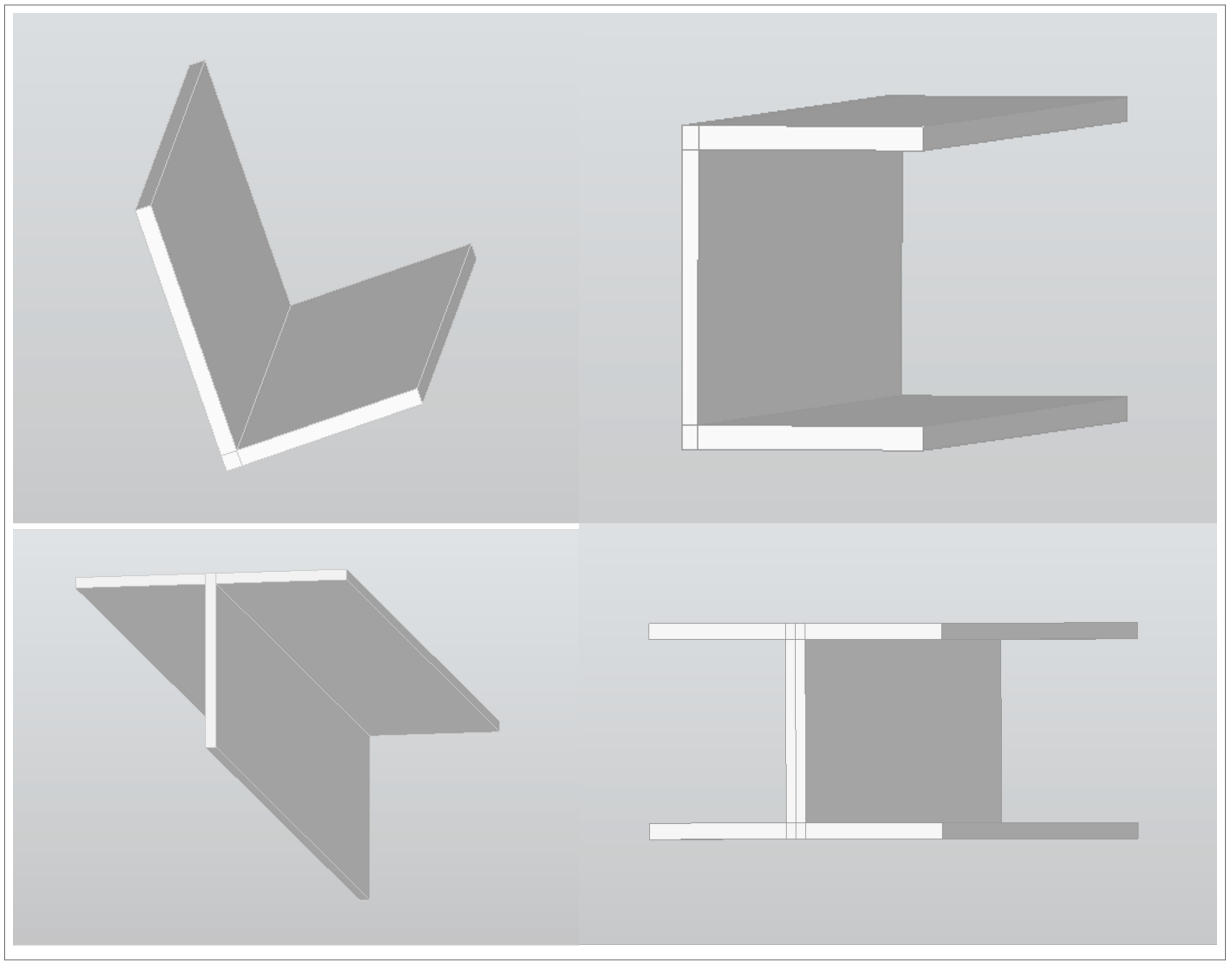


Figure 3.13: Steel section profiles used for cross bracing members.

#### 3.3.3.2 Diagonal Member Generation

Diagonal bracing members are created between two connection points by computing the member vector, aligning the section profile with the target direction, and positioning the member through geometric transformations.

Listing 3.11: Diagonal Member Creation

```
def _create_diagonal_member(p1, p2, thickness, section_type, dims
    , roll_sign):
    vec = gp_Vec(p1, p2)
    length = vec.Magnitude()

    solid = _create_section_solid(section_type, length, thickness
        , dims)

    # rotation and roll alignment
    # translation to midpoint

    return positioned_member
```

#### 3.3.3.3 X-Bracing Configuration

X-bracing consists of two diagonal members intersecting between adjacent girders to form a cross pattern. Optional horizontal members can be added at the top and bottom locations.

Listing 3.12: X-Bracing Pattern

```
def _x_bracing(x, yL, yR, depth, tf, thickness, flange_w,
               section_type, dims, bracket, skew_angle=0):

    braces = [
        _create_diagonal_member(gp_Pnt(x_l, yL, z_top),
                                gp_Pnt(x_r, yR, z_bot), ...),
        _create_diagonal_member(gp_Pnt(x_l, yL, z_bot),
                                gp_Pnt(x_r, yR, z_top), ...)
    ]
```

```
    return braces
```

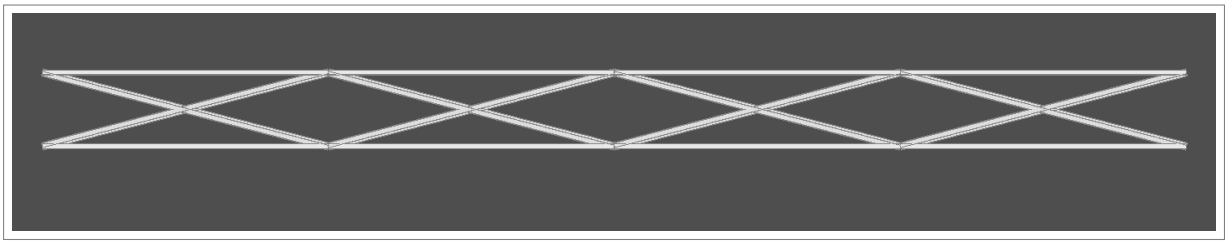


Figure 3.14: X-bracing configuration between adjacent girders.

#### 3.3.3.4 K-Bracing Configuration

K-bracing consists of diagonal members converging at an intermediate node between girders. A bottom horizontal member is included with an optional top member.

Listing 3.13: K-Bracing Pattern

```
def _k_bracing(x, yL, yR, depth, tf, thickness, flange_w,
               section_type, dims, top_bracket, skew_angle=0):

    braces = [
        _create_diagonal_member(gp_Pnt(x_l, yL, z_top),
                                gp_Pnt(x_m, ym, z_bot), ...),
        _create_diagonal_member(gp_Pnt(x_r, yR, z_top),
                                gp_Pnt(x_m, ym, z_bot), ...),
        _create_diagonal_member(gp_Pnt(x_l, yL, z_bot),
                                gp_Pnt(x_r, yR, z_bot), ...)
    ]
    return braces
```

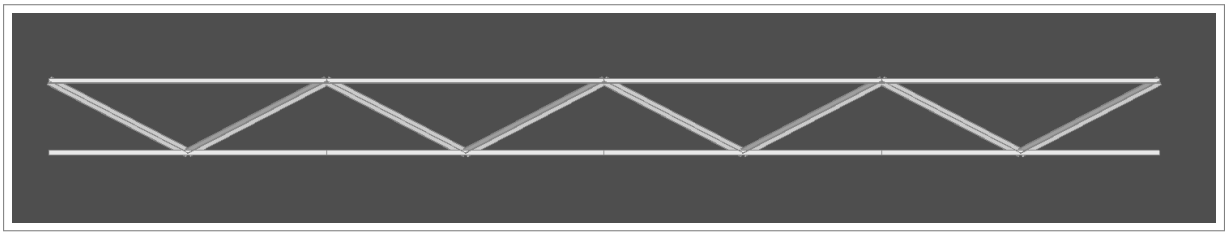


Figure 3.15: K-bracing configuration with mid-height convergence.

#### 3.3.3.5 Cross Bracing Placement Between Girders

Cross bracing assemblies are placed between adjacent girders at predefined longitudinal locations along the bridge span. The system generates bracing frames automatically based on girder spacing and bridge geometry.

Listing 3.14: Cross Bracing Placement Along Bridge Span

```
def build_cross_bracings(span_length_L, num_girders,
   girder_spacing,girder_depth, section_type, section_dims,
   thickness, panel_spacing):

    # compute panel locations
    # loop through girder bays
    # generate X or K bracing

    return bracings
```

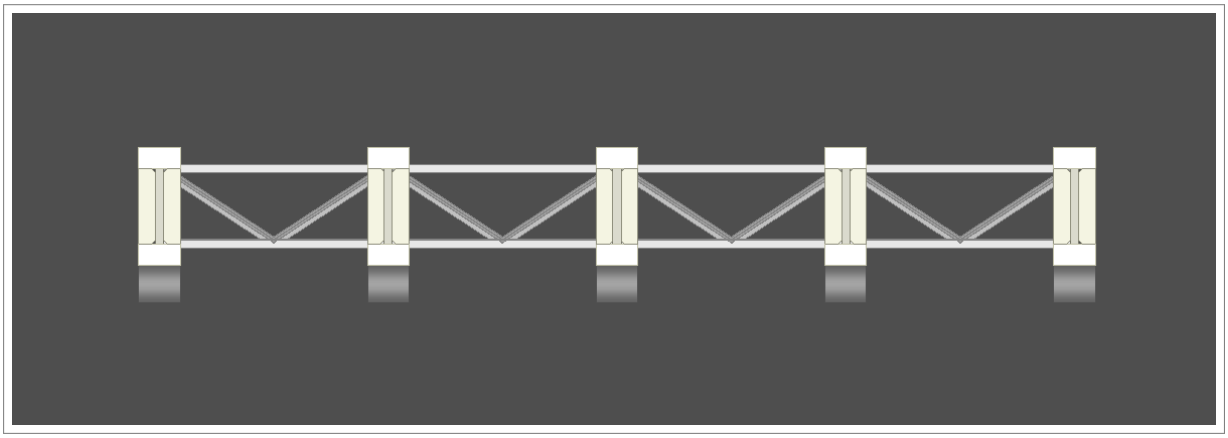


Figure 3.16: Cross bracing system placed between adjacent plate girders.

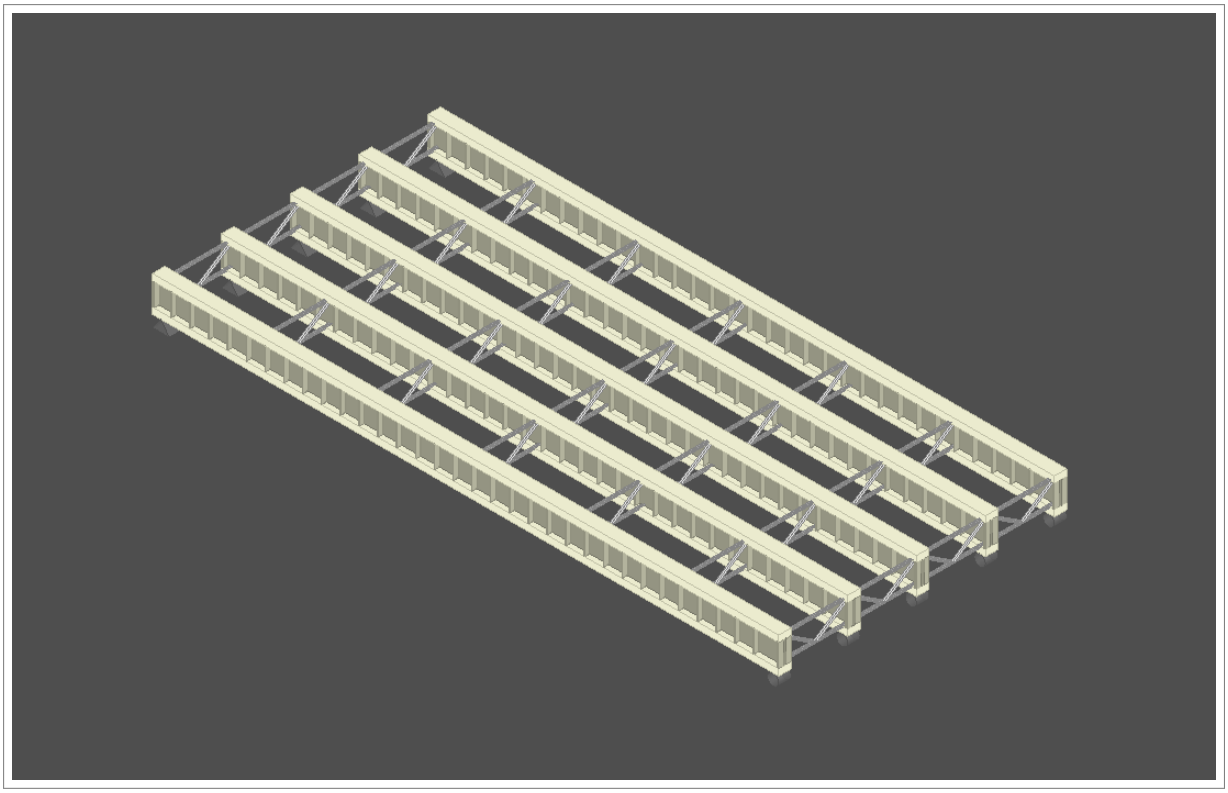


Figure 3.17: Cross bracing system placed between adjacent plate girders.

### 3.3.4 Crash Barrier Component

Crash barriers prevent vehicles from leaving the roadway. Two types are supported per IRC:5-2015: rigid RCC barriers and semi-rigid metallic barriers with W-beam rails.

#### 3.3.4.1 Rigid RCC Crash Barrier

Rigid reinforced cement concrete (RCC) crash barriers are provided along the bridge edges to ensure vehicle containment and enhance structural safety. The model supports IRC-compliant barrier configurations, including the standard IRC:5 profile and high-containment variants, by defining trapezoidal cross-sectional geometry using parametric dimensions. The barrier profile is generated using sectional height and width parameters, skew-adjusted boundary points, and longitudinal extrusion along the bridge span.

Listing 3.15: Parametric Rigid RCC Crash Barrier Generation

```
# Read IRC geometric parameters
H_total = design_dict.get("crash_barrier_height")
W_base  = design_dict.get("crash_barrier_width")
W_top   = design_dict.get("crash_barrier_top_notch")
```

```
# Define sectional height levels
z0 = 0.0
z1 = H_base + wearing
z2 = z1 + H_transition
z3 = H_total

# Create skew-adjusted trapezoidal profile points
poly = BRepBuilderAPI_MakePolygon()
for p in profile_points:
    poly.Add(p)

# Extrude profile along bridge span
solid = BRepPrimAPI_MakePrism(face, gp_Vec(length, 0, 0)).Shape()
```

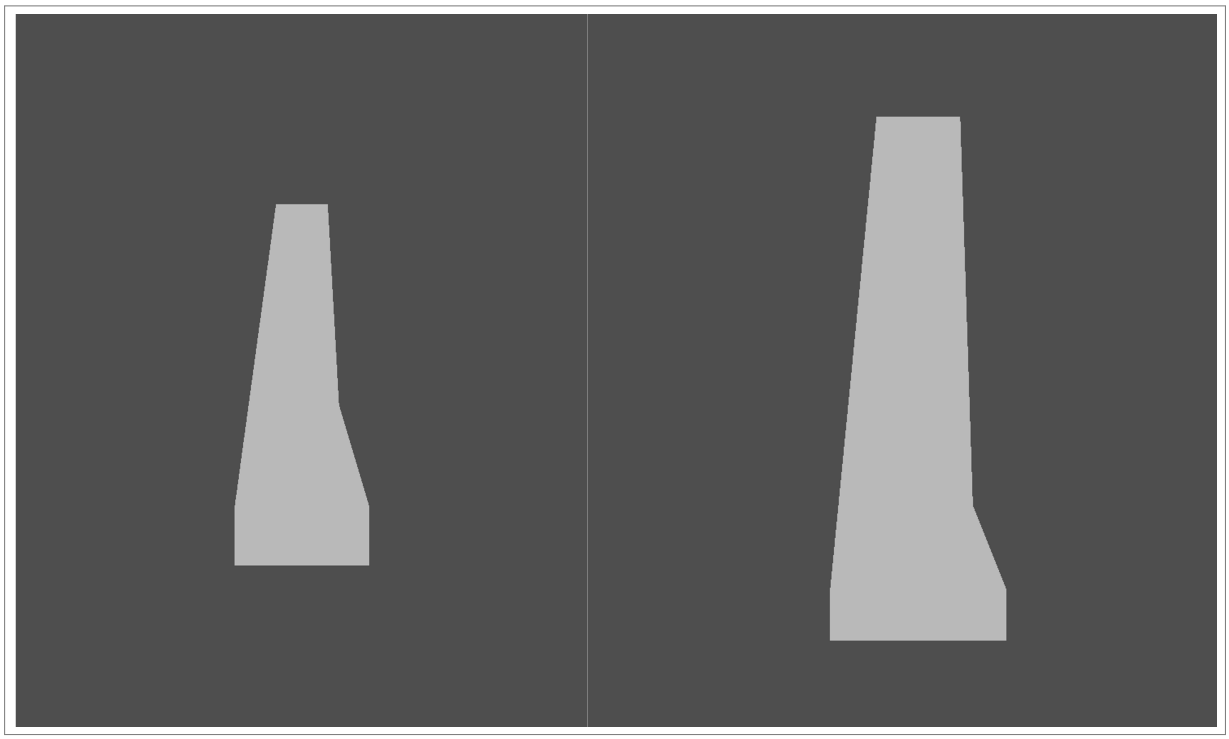


Figure 3.18: Rigid RCC crash barrier showing IRC:5 and high-containment trapezoidal profiles

#### 3.3.4.2 Semi-Rigid Metallic Crash Barrier

The semi-rigid metallic crash barrier consists of an RCC kerb base, steel posts (ISMC channels), horizontal spacer elements, and corrugated W-beam rails. The system supports both single and double W-beam configurations, where multiple beams are positioned vertically with specified spacing to improve impact energy absorption. The W-beam profile is generated parametrically using Gaussian wave functions to model the characteristic double-corrugated geometry.

Listing 3.16: Corrugated W-Beam Profile Generation

```
# Gaussian wave parameters for double corrugation
sigma = W_BEAM_HEIGHT / 10.0
mu1 = W_BEAM_HEIGHT * 0.25
mu2 = W_BEAM_HEIGHT * 0.75

# Generate corrugated beam profile
for i, z in enumerate(zs, start=1):
    y_wave = (amp * math.exp(-((z - mu1)**2) / (2 * sigma**2)) +
              amp * math.exp(-((z - mu2)**2) / (2 * sigma**2)))
    outer_pts.SetValue(i, gp_Pnt(get_skew_x(y_wave), y_wave, z))

# Create smooth beam surface
outer_curve = GeomAPI_PointsToBSpline(outer_pts).Curve()
```

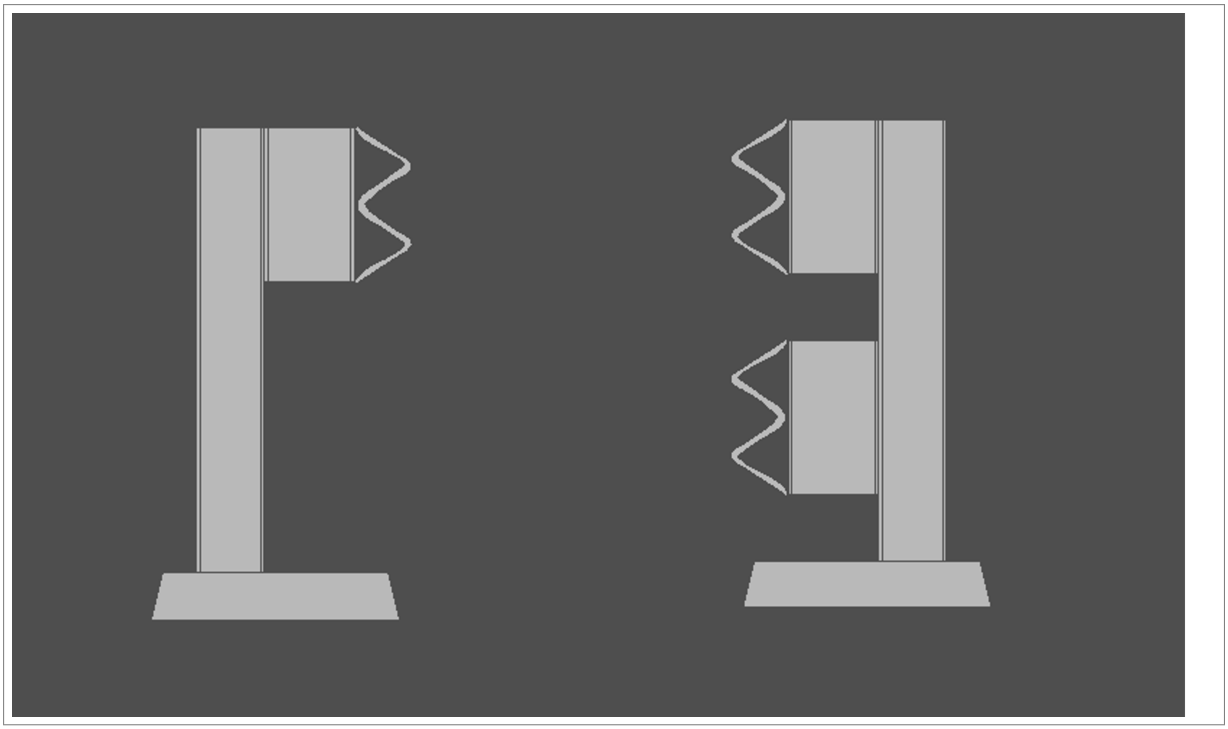


Figure 3.19: Semi-rigid metallic crash barrier showing RCC kerb, steel posts, spacers, and single/double W-beam rails

#### 3.3.4.3 Barrier Positioning Logic

Barriers are positioned based on footpath configuration:

Listing 3.17: Barrier Positioning Based on Footpath Configuration

```
def calculate_carriageway_offset(footpath_config, footpath_width,
    railing_width):
    if footpath_config in ("NONE", "BOTH"):
        return 0.0  # Symmetric - carriageway centered
    elif footpath_config == "LEFT":
        return (footpath_width + railing_width) / 2.0  # Shifted
            right
    elif footpath_config == "RIGHT":
        return -(footpath_width + railing_width) / 2.0  # Shifted
            left


# Calculate barrier positions
total_deck_width = calculate_deck_width(...)
carriageway_offset = calculate_carriageway_offset(...)
```

```
if footpath_config == "NONE":
    y_r = deck_half - crash_barrier_base_width / 2
    y_l = -deck_half + crash_barrier_base_width / 2
elif footpath_config == "LEFT":
    y_r = deck_half - crash_barrier_base_width / 2.0
    y_l = cw_left + crash_barrier_base_width / 2.0
# [Additional configurations...]
```

### 3.3.5 Median Barrier Component

Median barriers separate opposing traffic streams per IRC:5-2015. Three types are supported: raised kerb (Fig 5a), RCC crash barrier (Fig 5b), and metallic crash barrier (Fig 5c).

#### 3.3.5.1 Raised Kerb Median

Simple trapezoidal concrete barrier providing basic lane separation:

Listing 3.18: Raised Kerb Median with Skew Support

```
def create_raised_kerb(length, design_dict, skew_angle=0):
    skew_rad = math.radians(skew_angle)
    def get_skew_x(y):
        return y * math.tan(skew_rad)

    # Extract dimensions
    kerb_height = design_dict.get("kerb_height")
    kerb_top_width = design_dict.get("kerb_top_width")
    kerb_bottom_width = design_dict.get("kerb_bottom_width")

    # Define trapezoidal profile with skew
    y_bottom_l, y_bottom_r = -kerb_bottom_width / 2.0,
       kerb_bottom_width / 2.0
    y_top_l, y_top_r = -kerb_top_width / 2.0, kerb_top_width /
       2.0
```

```
    p1 = gp_Pnt(get_skew_x(y_bottom_l), y_bottom_l, 0.0)
    p2 = gp_Pnt(get_skew_x(y_bottom_r), y_bottom_r, 0.0)
    p3 = gp_Pnt(get_skew_x(y_top_r), y_top_r, kerb_height)
    p4 = gp_Pnt(get_skew_x(y_top_l), y_top_l, kerb_height)

    # Create and extrude profile
    poly = BRepBuilderAPI_MakePolygon()
    for p in (p1, p2, p3, p4): poly.Add(p)
    poly.Close()

    face = BRepBuilderAPI_MakeFace(poly.Wire()).Face()
    return BRepPrimAPI_MakePrism(face, gp_Vec(length, 0, 0)).
       Shape()
```

Figure 3.20: Raised kerb median with trapezoidal profile

#### 3.3.5.2 RCC Crash Barrier Median

Dual New Jersey-profile barriers separated by median width:

Listing 3.19: RCC Crash Barrier Median Profile Creation

```
def create_rcc_crash_barrier_median(length, design_dict,
   skew_angle=0):
```

```
    # Extract IRC-compliant dimensions
    barrier_height = design_dict.get("barrier_height", 900)
    barrier_base_h = design_dict.get("barrier_split_h3", 100)
    H_transition = 250

    # Define Z levels and Y widths
    z0, z1 = 0.0, wearing + barrier_base_h
    z2 = z1 + H_transition
    z3 = barrier_height

    y_base = barrier_bottom_width / 2.0
    y_top = barrier_top_width / 2.0
    y_trans = (barrier_bottom_width - 200) / 2.0

    # Create New Jersey profile with skew offset
    # [Profile points with get_skew_x() applied]

    return BRepPrimAPI_MakePrism(face, gp_Vec(length, 0, 0)).
       Shape()
```

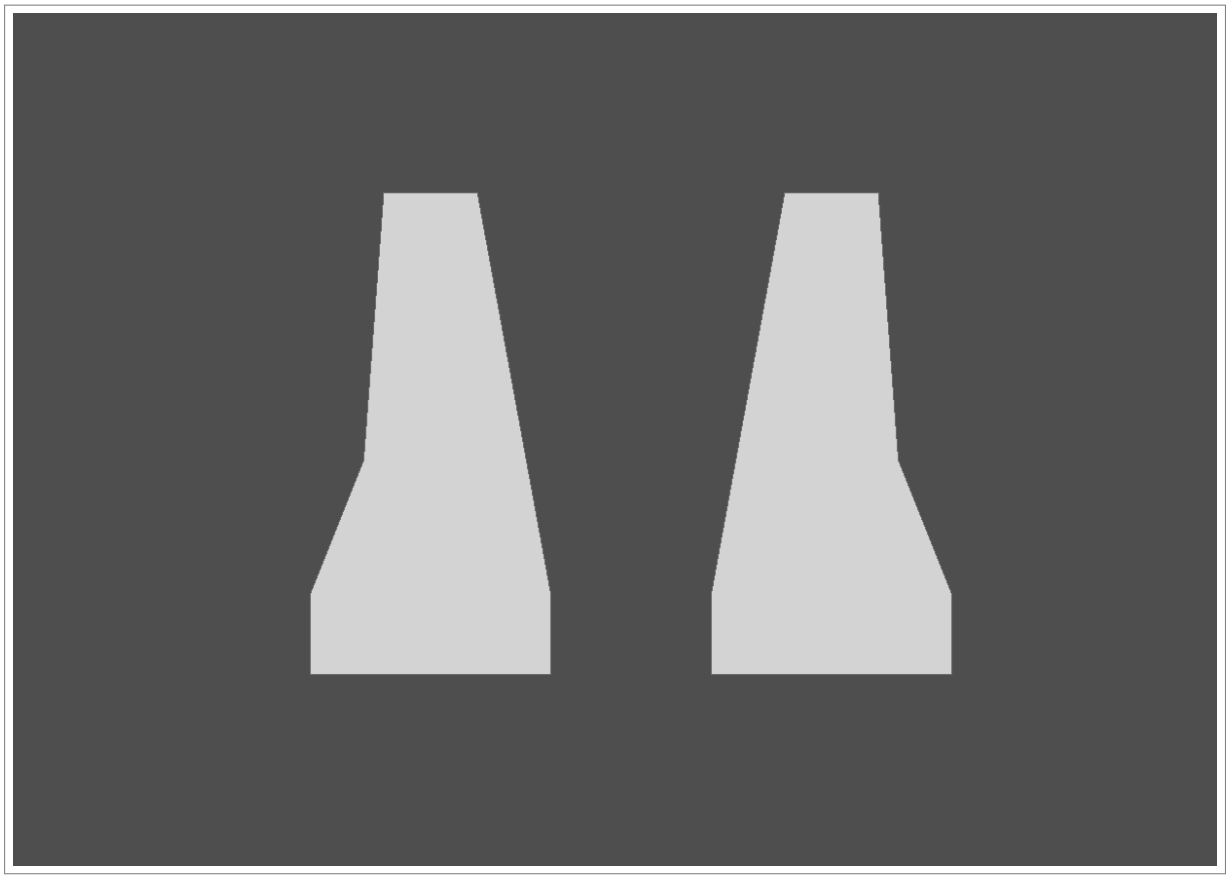


Figure 3.21: RCC crash barrier median

#### 3.3.5.3 Metallic Crash Barrier Median

Central kerb with metallic barrier assemblies on both sides:

Listing 3.20: Metallic Barrier Median Assembly

```
def create_metallic_barrier_system(length, design_dict,
                                   kerb_top_width, kerb_height,
                                      skew_angle=0):
    # Post placement with skew consideration
    safe_skew_shift = (kerb_top_width / 2.0 + 25.0) * math.tan(
       math.radians(abs(skew_angle)))
    start_x = safe_skew_shift + end_offset + post_web_thk / 2.0

    # Generate posts at uniform spacing
    num_posts = int(length / post_spacing) + 1
    actual_spacing = (end_x - start_x) / (num_posts - 1)

    for i in range(num_posts):
        x_pos = start_x + (i * actual_spacing)
        post_solid = create_vertical_channel(...)
        post_solid = _translate(post_solid, x=x_pos, y=
           post_y_center, z=kerb_height)
        posts_combined = BRepAlgoAPI_Fuse(posts_combined,
           post_solid).Shape()

    # Generate W-beams and spacers
    return {"posts": posts_combined, "spacers": spacers_combined,
            "w_beams": beams_combined}
```

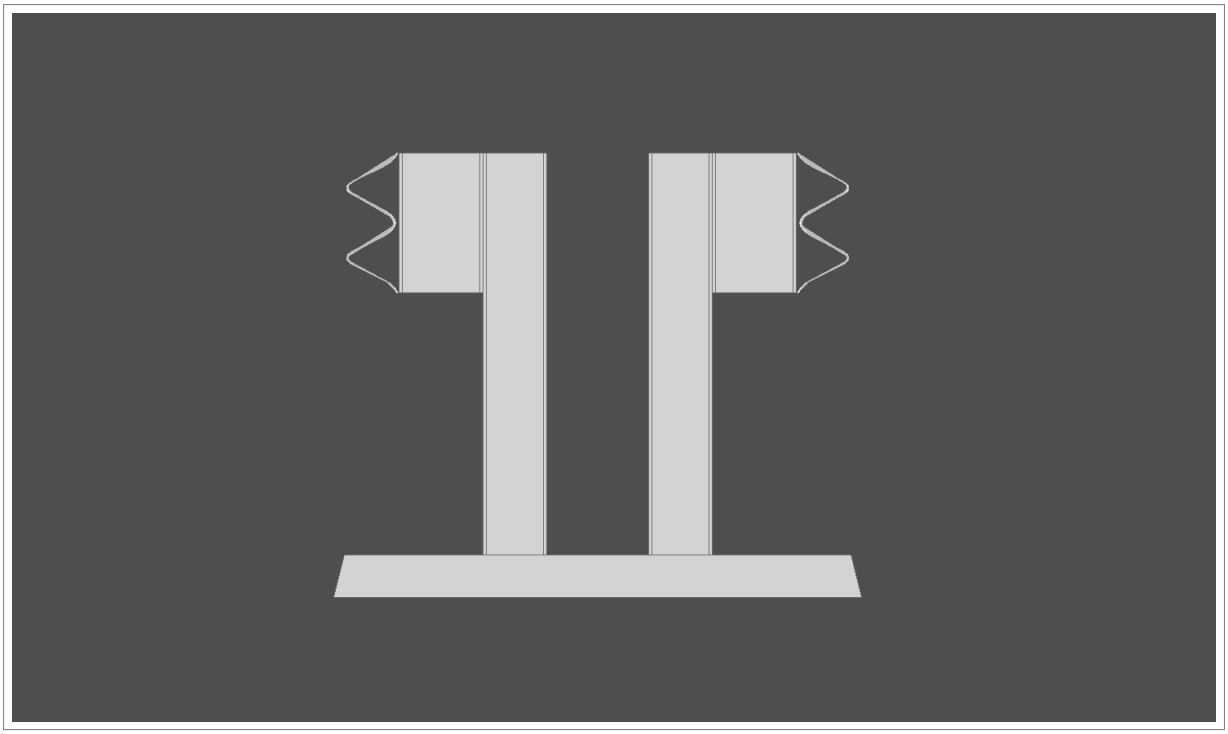


Figure 3.22: Metallic crash barrier median system

### 3.3.6 Railing Component

Railings protect pedestrians along footpaths. Two types are supported: solid RCC railings with aesthetic openings and steel railings with posts and horizontal rails.

#### 3.3.6.1 RCC Railing

Concrete base supporting vertical body with decorative rectangular openings:

Listing 3.21: RCC Railing with Decorative Openings

```
# Create base and body
base = create_rectangular_prism(length, railing_width,
   BASE_HEIGHT, skew_angle)
body = create_rectangular_prism(length, railing_width,
   body_height, skew_angle)

# Create and subtract openings
spacing = body_height / (rail_count + 1)
for i in range(rail_count):
    z_center = BASE_HEIGHT + (i + 1) * spacing
```

```
    hole = create_rectangular_prism(hole_length, hole_width,
        hole_height, skew_angle)
    body_with_holes = BRepAlgoAPI_Cut(body_with_holes, hole).
        Shape()
```

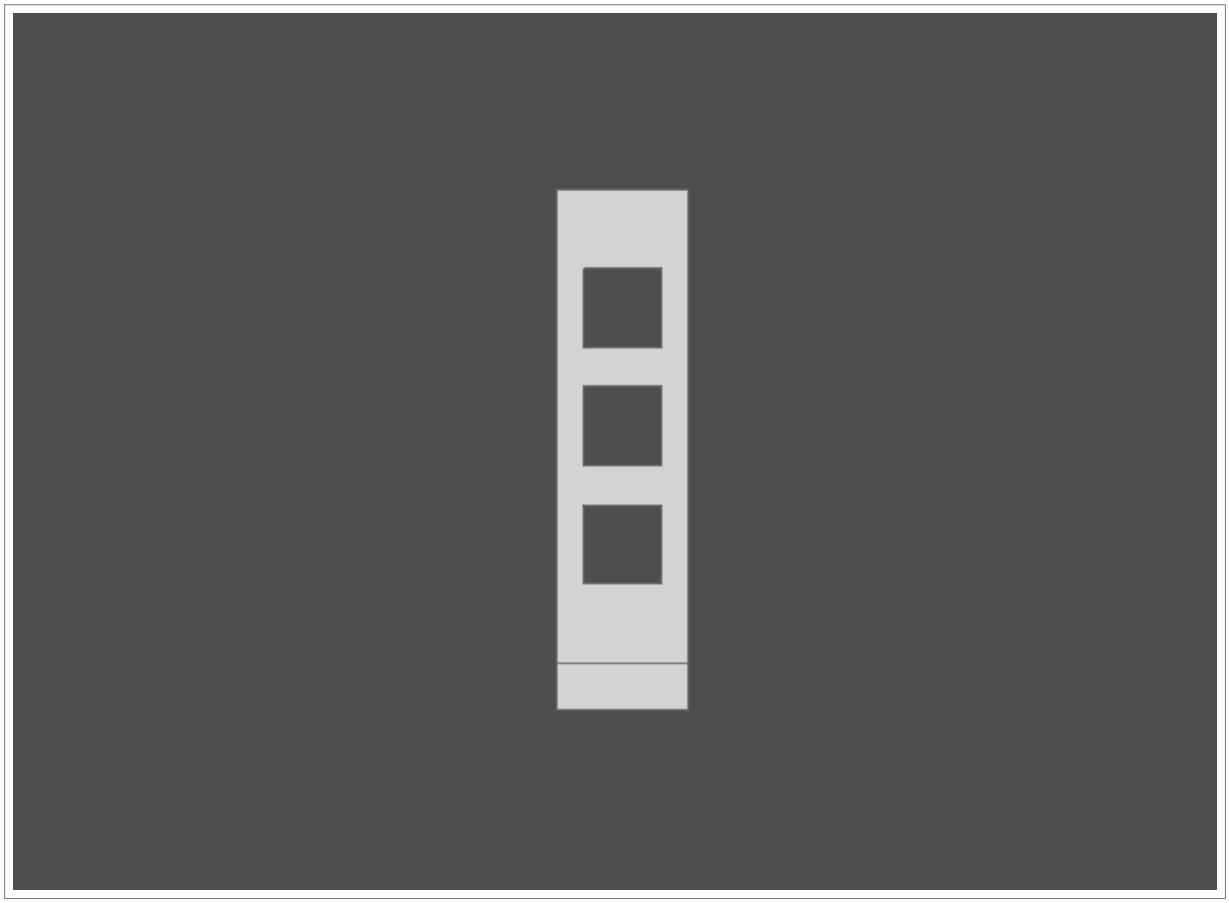


Figure 3.23: RCC railing Front View

Figure 3.24: RCC Railing Side View

#### 3.3.6.2 Steel Railing

A semi-rigid steel railing system is provided along the bridge edges for user safety. The assembly consists of a concrete base supporting uniformly spaced rectangular steel posts and horizontal rails. The model is generated parametrically by controlling post spacing, rail positioning, and overall railing height.

Listing 3.22: Parametric Steel Railing Generation

```
# Create concrete base
base = create_rectangular_prism(length, BASE_WIDTH, BASE_HEIGHT)

# Generate equally spaced vertical posts
for i in range(num_spaces + 1):
    post = create_rectangular_prism(POST_LENGTH, POST_BREADTH,
       post_height)
    post = translate(post, x=i * actual_spacing, z=BASE_HEIGHT)

# Create top and mid horizontal rails
top_rail = create_rectangular_prism(length, RAIL_SIZE, RAIL_SIZE)
mid_rail = create_rectangular_prism(length, RAIL_SIZE, RAIL_SIZE)
```

```
# Fuse base, posts, and rails into final assembly
final_shape = BRepAlgoAPI_Fuse(base, steel_structure).Shape()
```

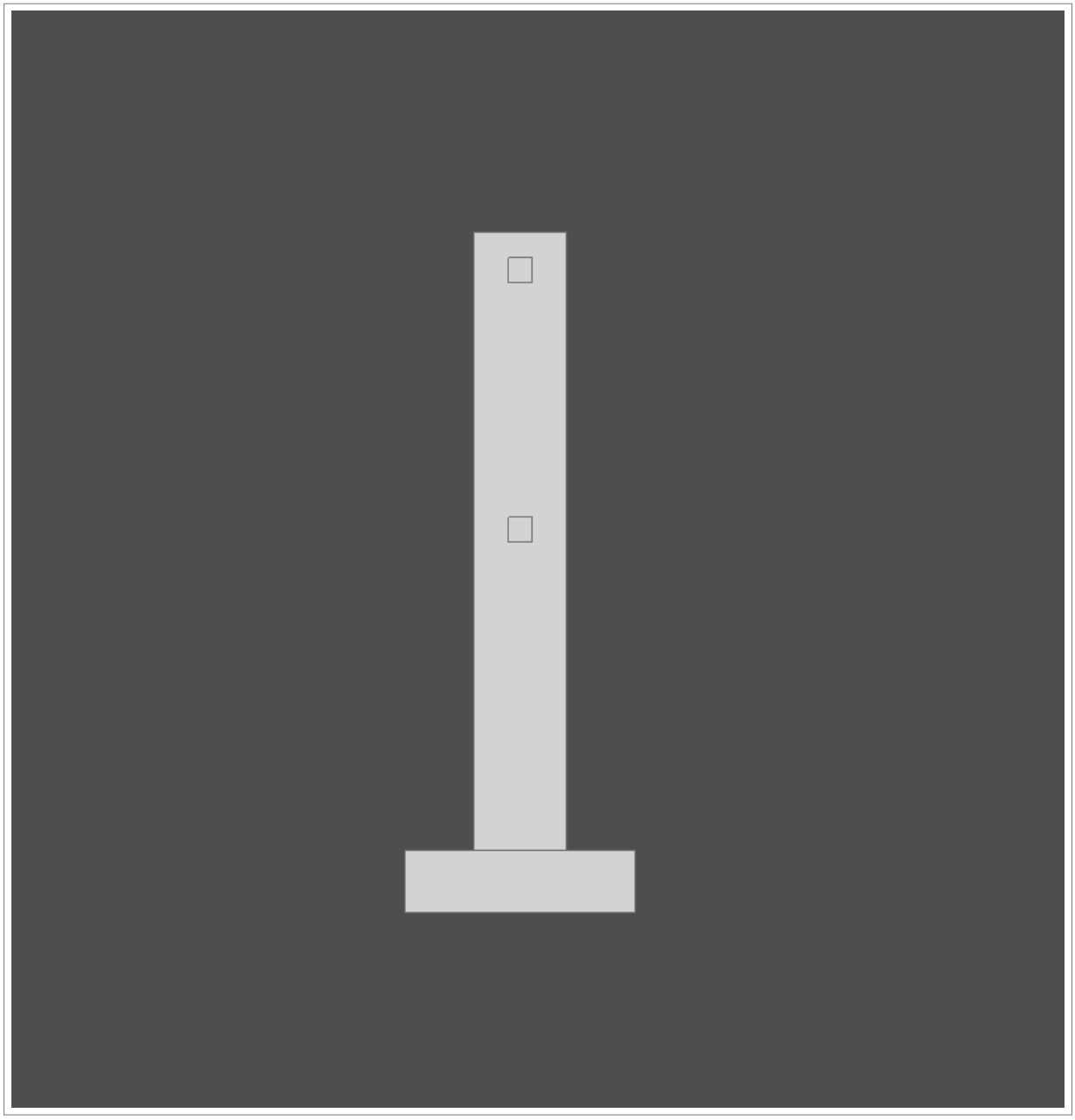


Figure 3.25: Steel Railing - Front View

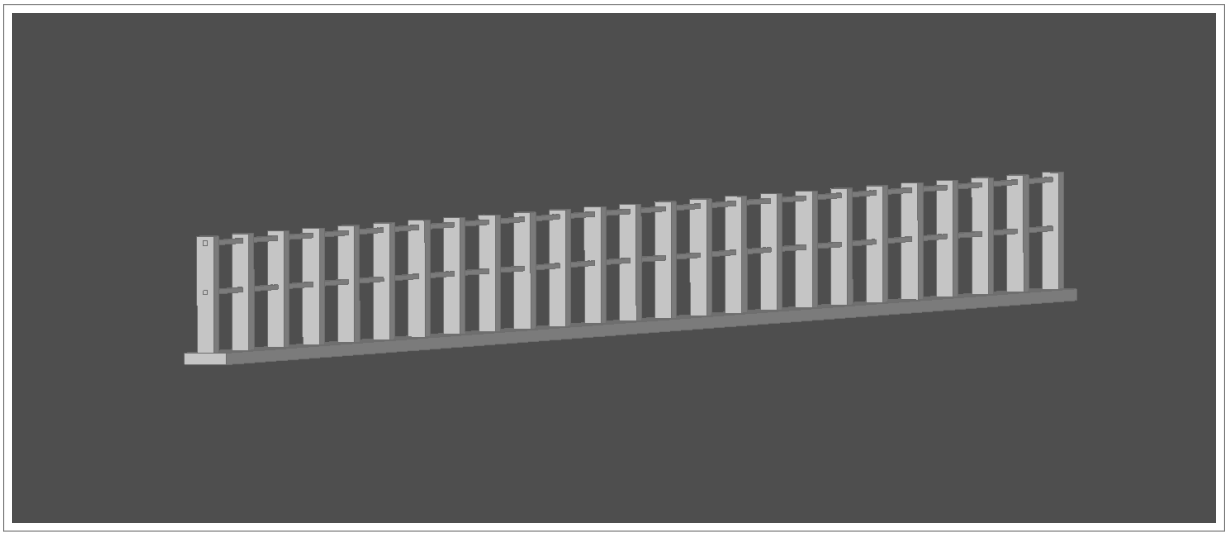


Figure 3.26: Steel Railing - Side View

## 3.4 System Architecture Overview

The plate girder bridge generation system consists of two primary modules that work in tandem to produce complete 3D bridge models:

1. **CAD Generator** (`cad_generator.py`): Core parametric modeling engine that generates all bridge component geometries based on IRC:5-2015 specifications

2. **3D Viewer** (`cad_3d.py`): Interactive visualization interface that displays, manipulates, and exports the generated bridge models

### 3.4.1 Module Interaction Workflow

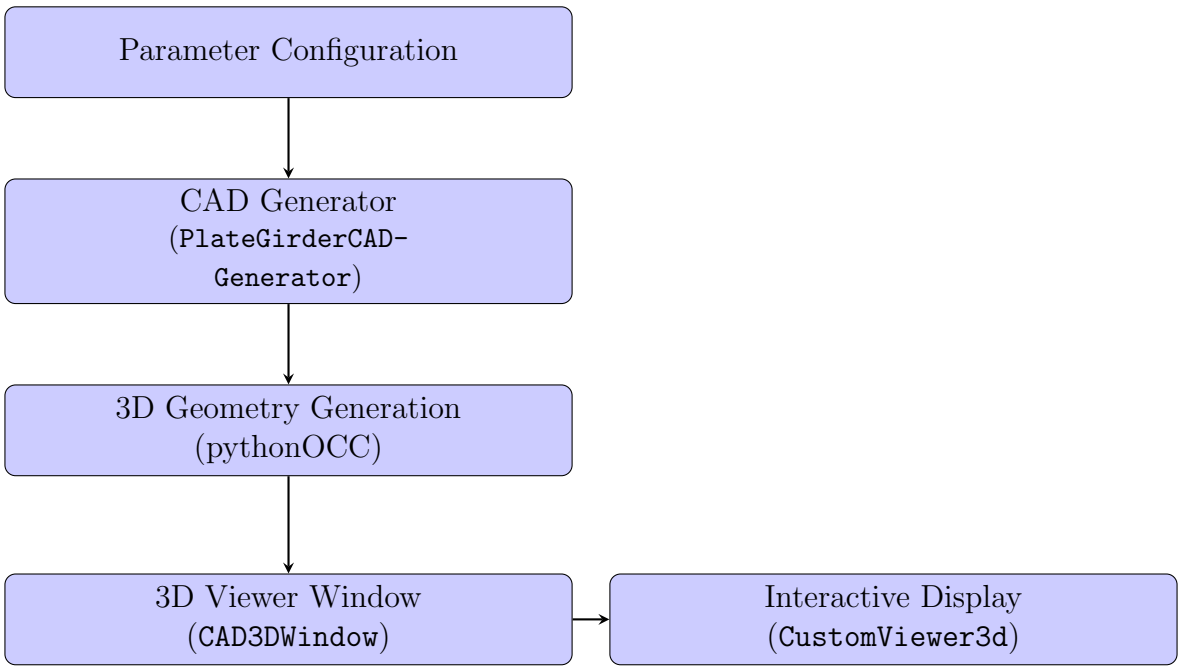



Figure 3.27: System architecture and data flow between CAD generator and 3D viewer modules

## 3.5 CAD Generator Module

The `PlateGirderCADGenerator` class serves as the central orchestrator for bridge geometry creation. It manages all design parameters and coordinates the generation of individual bridge components through specialized builder functions.

### 3.5.1 Main Assembly Workflow

Listing 3.23: Bridge Assembly Process (Simplified)

```
class PlateGirderCADGenerator:
    def generate(self):
        # STEP 1: Build single plate girder geometry
        pg = build_plate_girder_geometry(...)

        # STEP 2: Place multiple girders with spacing and skew
           offset
```

```
        for i in range(self.num_girders):
            y_offset = (i * self.girder_spacing) - (total_width /
                2)
            x_offset = calculate_skew_offset(y_offset)
            girders.append(translate(pg["web"], x_offset,
               y_offset))
            # [Place flanges and stiffeners]

        # STEP 3: Place support structures
        # [Triangular and cylindrical supports]

        # STEP 4: Build cross bracing system
        cross_bracings = build_cross_bracings(...)

        # STEP 5: Apply IRC:5-2015 specifications
        design_dict = IRC5_2015.cl_109_6_3_shapes(...)

        # STEP 6: Build deck system
        deck_out = build_deck(...)

        # STEP 7: Build crash barriers
        crash_barriers = build_crash_barriers(...)

        # STEP 8: Build median barriers (if enabled)
        if self.enable_median:
            median_barriers = build_median(...)

        # STEP 9: Build railings
        railings = build_railings(...)

        # STEP 10: Return organized component dictionary
        return {"girders": girders, "deck_slab": deck_slab, ...}
```

## 3.6 Parameter Configuration

The bridge model is highly configurable through comprehensive parameters organized by component type. All parameters follow IRC:5-2015 specifications and can be adjusted to meet project requirements.

### 3.6.1 Girder Parameters

| Parameter | Default | Description |
|---|---|---|
| span_length_L | 25000 mm | Total span length of the bridge |
| girder_section_d | 900 mm | Clear web depth (height between flanges) |
| girder_section_bf | 500 mm | Top flange width |
| girder_section_bf_b | 500 mm | Bottom flange width |
| girder_section_tf | 260 mm | Top flange thickness |
| girder_section_tf_b | 260 mm | Bottom flange thickness |
| girder_section_tw | 100 mm | Web thickness |
| num_girders | 5 | Number of parallel girders |
| girder_spacing | 2750 mm | Center-to-center spacing between girders |

Table 3.1: Plate Girder Configuration Parameters

### 3.6.2 Geometry Parameters

| Parameter | Default | Description |
|---|---|---|
| skew_angle | 0° | Bridge skew angle in degrees (0 = no skew). Affects longitudinal positioning of components based on transverse location. |

Table 3.2: Geometry Configuration Parameters

### 3.6.3 Deck Parameters

| Parameter | Default | Description |
|---|---|---|
| carriageway_width | 12000 mm | Width of the vehicular carriageway |
| deck_thickness | 400 mm | Thickness of the concrete deck slab |
| footpath_config | "BOTH" | Footpath configuration: NONE, LEFT, RIGHT, BOTH |
| footpath_width | 1500 mm | Width of each footpath (when present) |
| railing_width | 300 mm | Width of railing base |

Table 3.3: Deck and Footpath Configuration Parameters

### 3.6.4 Stiffener Parameters

| Parameter | Default | Description |
|---|---|---|
| stiffener_width | 200 mm | Width of intermediate stiffeners |
| stiffener_length | 10 mm | Projection length of stiffeners from web |
| include_end_stiffeners | True | Enable/disable bearing stiffeners at supports |
| end_stiffener_thickness | 25 mm | Thickness of bearing stiffeners |

Table 3.4: Stiffener Configuration Parameters

### 3.6.5 Crash Barrier Parameters

| Parameter | Default | Description |
|---|---|---|
| barrier_type | "Semi-Rigid" | Barrier type: Flexible, Semi-Rigid, or Rigid |
| crash_barrier_subtype | "Double W-beam" | For Rigid: IRC-5R or High Containment. For Semi-Rigid/Metallic: Single W-beam or Double W-beam |

Table 3.5: Crash Barrier Configuration Parameters

### 3.6.6 Median Barrier Parameters

| Parameter | Default | Description |
|---|---|---|
| enable_median | True | Enable or disable median barrier |
| median_type | "Metallic Crash Barrier" | Type: Raised Kerb, RCC Crash Barrier, or Metallic Crash Barrier |

Table 3.6: Median Barrier Configuration Parameters

### 3.6.7 Railing Parameters

| Parameter | Default | Description |
|---|---|---|
| railing_type | "rcc" | Railing material type: rcc or steel |
| rail_count | 3 | Number of horizontal rail openings (for RCC railings) |

Table 3.7: Railing Configuration Parameters

### 3.6.8 Cross Bracing Parameters

| Parameter | Default | Description |
|---|---|---|
| cross_bracing_spacing | 4000 mm | Longitudinal spacing between bracing panels |
| cross_bracing_thickness | 5 mm | Thickness of bracing members |
| bracing_type | "K" | Bracing configuration: X or K |
| x_bracket_option | "BOTH" | X-bracing bracket location: NONE, UPPER, LOWER, BOTH |
| k_top_bracket | True | Include top bracket for K-bracing |
| cross_bracing_section_type | "DOUBLE_ANGLE" | Section type: CHANNEL, ANGLE, DOUBLE_ANGLE, DOUBLE_CHANNEL, I_SECTION |

Table 3.8: Cross Bracing Configuration Parameters

### 3.6.9 Cross Bracing Section Dimensions

The cross_bracing_section_dims parameter is a dictionary that varies based on the selected section type:

| Section Type | Required Keys | Description |
|---|---|---|
| ANGLE | `leg_h` | Vertical leg height (mm) |
| | `leg_w` | Horizontal leg width (mm) |
| DOUBLE_ANGLE | `leg_h` | Vertical leg height (longer leg, mm) |
| | `leg_w` | Horizontal leg width (shorter leg, mm) |
| | `connection_type` | Connection configuration: LONGER_LEG or SHORTER_LEG (determines which legs are connected back-to-back) |
| CHANNEL, DOUBLE_CHANNEL, I_SECTION | `depth` | Overall section depth (mm) |
| | `flange_width` | Flange width (mm) |
| | `web_thickness` | Web thickness (mm) |
| | `flange_thickness` | Flange thickness (mm) |

Table 3.9: Cross Bracing Section Dimension Parameters

**Example configurations:**

Listing 3.24: Cross Bracing Section Configuration Examples

```
# OPTION 1: Double Angle (default)
self.cross_bracing_section_type = "DOUBLE_ANGLE"
self.cross_bracing_section_dims = {
    "leg_h": 100,                    # Vertical leg
    "leg_w": 50,                     # Horizontal leg
    "connection_type": "LONGER_LEG"  # Back-to-back connection
}

# OPTION 2: Channel Section
self.cross_bracing_section_type = "CHANNEL"
self.cross_bracing_section_dims = {
    "depth": 100,
    "flange_width": 50,
    "web_thickness": 5,
    "flange_thickness": 7
}

# OPTION 3: Single Angle
self.cross_bracing_section_type = "ANGLE"
```

```
self.cross_bracing_section_dims = {
    "leg_h": 100,   # Vertical leg
    "leg_w": 50     # Horizontal leg
}
```

**Note:** When `end_diaphragm_type` is set to "Cross Bracing", the end diaphragms use the same section configuration as `cross_bracing_section_type` and `cross_bracing_section_dims`. For "Rolled Beam" or "Welded Beam" types, the `end_diaphragm_dims` dictionary follows the same structure as cross bracing section dimensions (Table 3.9).

## 3.7 3D Visualization Module

The `CAD3DWindow` class provides an interactive 3D visualization environment built on PySide6 and pythonOCC. It enables real-time viewing, component selection, and model manipulation.

### 3.7.1 Core Components

#### 3.7.1.1 CAD3DWindow

Main application window that manages the complete visualization pipeline:

- **Initialization**: Creates CAD generator instance and generates initial model data
- **Display Management**: Controls component visibility, colors, and transparency
- **User Interaction**: Handles component selection, zoom controls, and view manipulation

#### 3.7.1.2 CustomViewer3d

Custom 3D viewer widget that extends pythonOCC's display capabilities:

- **Interactive Navigation**: Pan, rotate, zoom with mouse controls
- **View Cube**: Provides quick access to standard views (front, top, isometric, etc.)
- **Component Highlighting**: Dynamic hover effects and selection feedback

- **Context Management**: Manages OpenCascade rendering context and AIS objects

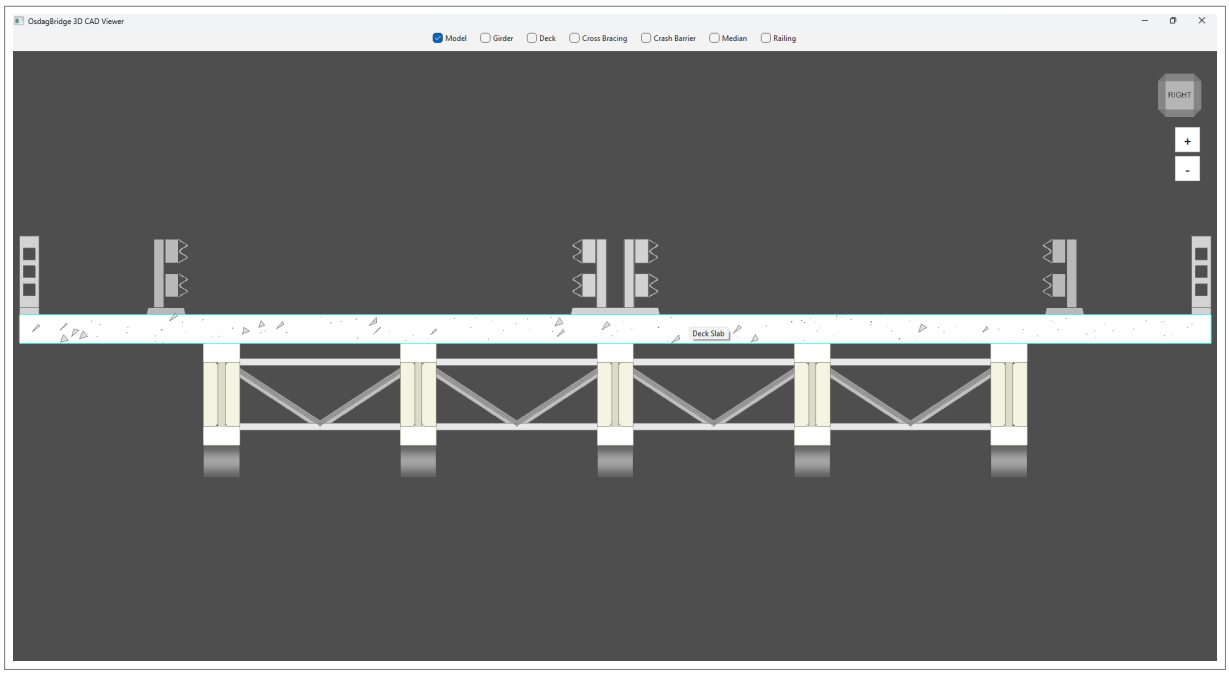



Figure 3.28: Interactive 3D viewer interface showing complete bridge assembly with navigation controls: view cube (top-right) for quick orientation changes, zoom controls (below view cube), and component hover highlighting system

## 3.7.2 Visualization Features

### 3.7.2.1 Component Color Scheme

The viewer applies a standardized color scheme for visual differentiation of components:

| Component | RGB Color | Visual Purpose |
|---|---|---|
| Girder Web | (47, 47, 35) | Dark steel appearance |
| Girder Flange | (134, 134, 100) | Medium steel tone |
| Stiffeners | (72, 72, 54) | Darker steel variation |
| Deck Slab | (100, 100, 100) | Concrete gray |
| Crash Barriers | (40, 40, 40) | Dark protective barrier |
| Cross Bracing | (60, 60, 60) | Medium-dark structural steel |
| W-Beams | (128, 128, 128) | Metallic gray |
| Support Structures | (20, 20, 20) | Very dark foundation color |

Table 3.10: Component Color Mapping in 3D Viewer

#### 3.7.2.2 Multi-Select Component Visibility

The `BridgeComponentCheckbox` widget provides flexible component visibility control:

- **Model View**: Displays complete bridge assembly with all components
- **Girder View**: Shows plate girders, flanges, stiffeners, and supports
- **Deck View**: Isolates deck slab and surface textures
- **Cross Bracing View**: Displays bracing system and diaphragms
- **Crash Barrier View**: Shows barriers, W-beams, posts, and spacers
- **Median View**: Isolates median barrier components
- **Railing View**: Displays pedestrian railings

Multiple components can be selected simultaneously to view specific combinations (e.g., Girder + Deck + Cross Bracing).

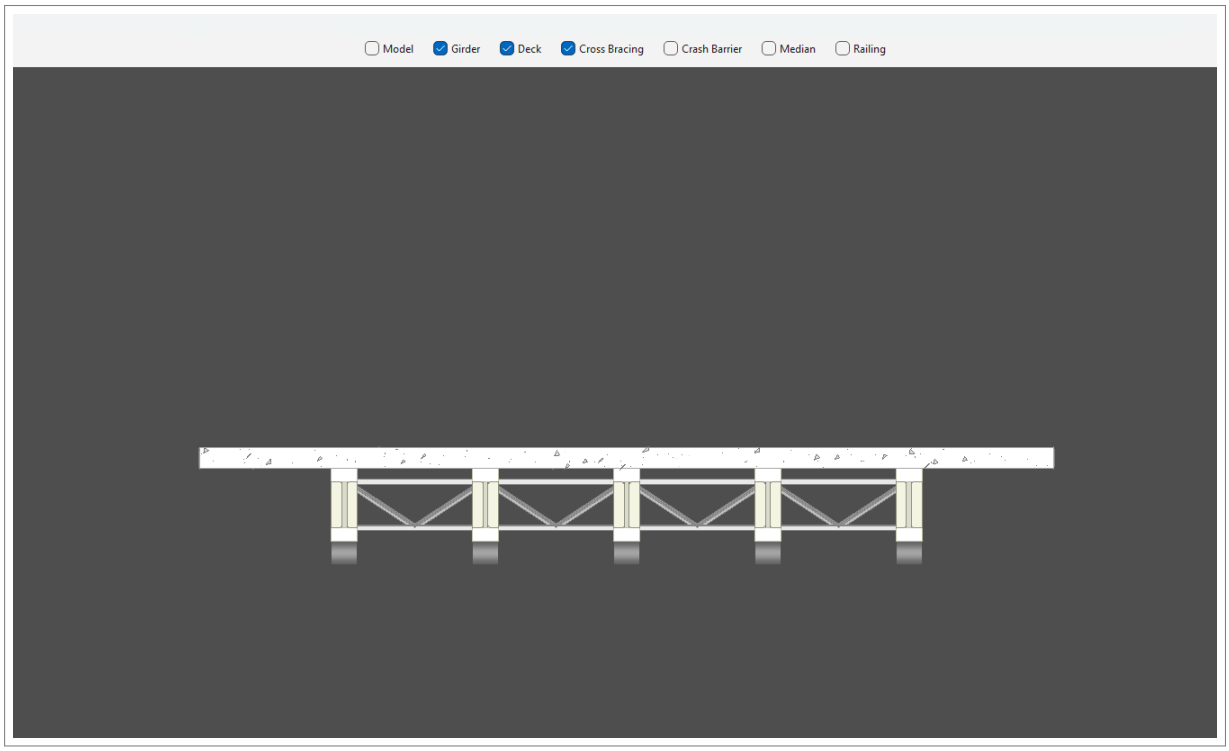



Figure 3.29: Multi-select visualization showing girders, deck, and cross bracing components

### 3.7.3 Viewer Initialization Workflow

Listing 3.25: 3D Viewer Initialization Process

```
class CAD3DWindow:
    def __init__(self):
        # 1. Create CAD generator and generate model
        self.generator = PlateGirderCADGenerator()
        self.generator.model_data = self.generator.generate()

        # 2. Setup UI components
        self.setup_ui()

        # 3. Initialize display (deferred for stability)
        self.init_display()

    def init_display(self):
        # Create CustomViewer3d widget
        self.viewer = CustomViewer3d(self)

        # Defer InitDriver to avoid Qt initialization conflicts
        QTimer.singleShot(0, self._deferred_init_driver)

    def _deferred_init_driver(self):
        # Initialize OpenCascade display context
        self.viewer.InitDriver()
        self.display = self.viewer._display

        # Setup interactive features
        self.viewer.display_view_cube()
        self.create_cad_view_controls()

        # Load and display bridge model
        self.load_bridge()
```

### 3.7.4 Component Registration and Display

The viewer maintains organized dictionaries for component management:

Listing 3.26: Component Display Registration

```
def display_and_register(shapes, key, label, color, transparency=
    None):
    """
    Display shapes and register them for selection/hover.

    Args:
        shapes: List of TopoDS_Shape objects
        key: Component category key
        label: Display label for hover tooltip
        color: Quantity_Color for component
        transparency: Optional transparency value (0-1)
    """
    ais_list = []

    for shp in shapes:
        # Display shape with specified color
        ais = display.DisplayShape(shp, color=color,
                                   transparency=transparency,
                                   update=False)

        # Activate for interaction
        context.Activate(ais, 0)
        ais_list.append(ais)

    # Register for component management
    self.viewer.model_ais_objects[key] = ais_list
    self.viewer.model_hover_labels[key] = label
```

### 3.7.5 Interactive Controls

#### 3.7.5.1 Zoom Controls

Dedicated zoom buttons positioned below the view cube:

- **Zoom In (+)**: Increases view magnification by factor of 1.1
- **Zoom Out (-)**: Decreases view magnification by factor of 1/1.1
- **Fit All**: Automatically adjusts view to fit entire model

#### 3.7.5.2 View Manipulation

Standard 3D navigation controls:

- **Rotate**: Left mouse drag
- **Pan**: Middle mouse drag or Shift + Left mouse drag
- **Zoom**: Mouse wheel scroll
- **Standard Views**: Click view cube faces for predefined orientations

### 3.7.6 Data Structure Organization

The CAD generator returns a comprehensive dictionary containing all bridge components:

| Dictionary Key | Contents |
|---|---|
| girders | Combined list of web and flange shapes |
| girder_web | Web shapes only |
| girder_flanges | Top and bottom flange shapes |
| stiffeners | Intermediate and bearing stiffeners |
| supports | All support structures (triangular + cylindrical) |
| supports_tri | Triangular support shapes |
| supports_cyl | Cylindrical support shapes |
| cross_bracings | Cross bracing members and diaphragms |
| deck_slab | Main deck slab geometry |
| deck_textures | Surface texture elements |
| deck_top_z | Z-coordinate of deck top surface |
| total_deck_width | Total width of deck including all features |
| crash_barriers | Crash barrier kerbs, posts, and spacers |
| crash_barrier_w_beams | W-beam rail components |
| median_barriers | Median barrier structures |
| median_w_beams | Median W-beam components |
| railings | Pedestrian railing elements |

Table 3.11: CAD Generator Output Dictionary Structure

## 3.8 Challenges and Solutions

### 3.8.1 Challenge 1: Coordinate System Consistency

**Problem:** Ensuring all components use a consistent coordinate system and reference points, particularly when components are generated by different modules with varying local coordinate conventions.

**Solution:** Established a global coordinate system with the bridge centerline at Y=0, longitudinal axis along X-direction, and vertical axis along Z-direction. All component builders were standardized to use this convention. Reference Z-levels (such as `girder_top_z` and `deck_top_z`) were calculated explicitly and passed between modules to ensure accurate vertical positioning.

### 3.8.2 Challenge 2: Skew Angle Implementation

**Problem:** Implementing bridge skew geometry where components at different transverse positions require different longitudinal offsets while maintaining structural integrity and visual correctness.

**Solution:** Developed a geometric offset calculation function that computes longitudinal shifts based on lateral position and skew angle using the formula: $x_{offset} = (y_{position} - y_{reference}) \times \tan(\theta_{skew})$. This ensures that all components (girders, bracings, barriers) follow the skew geometry consistently.

### 3.8.3 Challenge 3: Complex Geometry Generation

**Problem:** Creating accurate 3D models of corrugated W-beam profiles and complex New Jersey barrier profiles using parametric equations while maintaining manufacturing feasibility.

**Solution:** Implemented B-spline curve generation using Gaussian wave functions for W-beam corrugations, providing smooth, realistic profiles. For crash barriers, decomposed the complex profile into discrete geometric sections (base, transition, top) defined by precise IRC specifications, then assembled them using boolean operations.

### 3.8.4 Challenge 4: Dynamic Component Positioning

**Problem:** Automatically positioning crash barriers, railings, and median barriers based on variable deck width, footpath configuration, and carriageway layout while maintaining IRC clearance requirements.

**Solution:** Developed a hierarchical positioning system that first calculates total deck width from component dimensions, then determines carriageway offset based on footpath configuration, and finally computes individual component positions relative to these reference points. This ensures components automatically adjust their positions when bridge parameters change.

### 3.8.5 Challenge 5: IRC:5-2015 Specification Integration

**Problem:** Ensuring all safety barrier dimensions, profiles, and configurations strictly comply with IRC:5-2015 specifications across multiple barrier types and subtypes.

**Solution:** Implemented a dedicated IRC:5-2015 specification module (`IRC5_2015.cl_109_6_3_shapes`) that encodes all dimensional requirements and returns compliant design dictionaries. The main assembly queries this module before generating each component, ensuring automatic compliance regardless of user selections.

### 3.8.6 Challenge 6: Multi-Component Assembly Management

**Problem:** Managing and organizing dozens of individual component shapes (girders, stiffeners, bracings, barriers) for efficient rendering and material assignment in the visualization system.

**Solution:** Structured the assembly output as a categorized dictionary grouping related components (e.g., all girder webs, all W-beams, all railings). This organization enables efficient batch processing for material assignment, color coding, and selective visibility control in the CAD viewer.

### 3.8.7 Challenge 7: 3D Viewer Initialization Timing

**Problem:** Qt-pythonOCC initialization conflicts causing display failures when InitDriver is called too early in the widget lifecycle.

**Solution:** Implemented deferred initialization using QTimer.singleShot(0) to ensure the Qt event loop is fully established before initializing the OpenCascade display context. This eliminates race conditions and ensures stable rendering across different platforms.

### 3.8.8 Challenge 8: Interactive Component Selection

**Problem:** Enabling selective component visibility while maintaining proper AIS (Application Interactive Services) object management and avoiding memory leaks.

**Solution:** Created a component registration system that maps logical component groups to lists of AIS objects. The visibility update system uses context.Display() and context.Erase() with deferred updates (update=False) followed by a single Repaint() call to minimize rendering overhead.

## 3.9 Results

The parametric bridge generation system successfully produces complete IRC:5-2015 compliant 3D models featuring:

- Plate girders with intermediate and bearing stiffeners
- Reinforced concrete deck slabs with realistic surface texture

- Crash barriers (rigid RCC and semi-rigid metallic with W-beams)
- Pedestrian railings (RCC and steel tubular)
- Median barriers (raised kerb, RCC, metallic)
- Cross bracing systems (X-type and K-type with multiple section options)
- End diaphragms (cross bracing, rolled beam, welded beam)
- Support structures (triangular and cylindrical bearings)

Generated models feature precise dimensional accuracy with components positioned according to engineering standards. The system handles variable configurations including:

- Different girder numbers (3-7 typical)
- Adjustable span lengths (8m-30m range)
- Multiple footpath configurations (none, left, right, both)
- Various barrier combinations (rigid, semi-rigid, flexible)
- Skewed bridge geometries (0-45° typical)
- Multiple cross bracing section types (angle, channel, I-section)

The integrated 3D viewer provides:

- Real-time interactive visualization with smooth navigation
- Component-level visibility control with multi-select capability
- Standardized color coding for component identification
- View cube for quick orientation changes
- Zoom controls and automatic fit-to-view functionality
- Hover tooltips for component identification

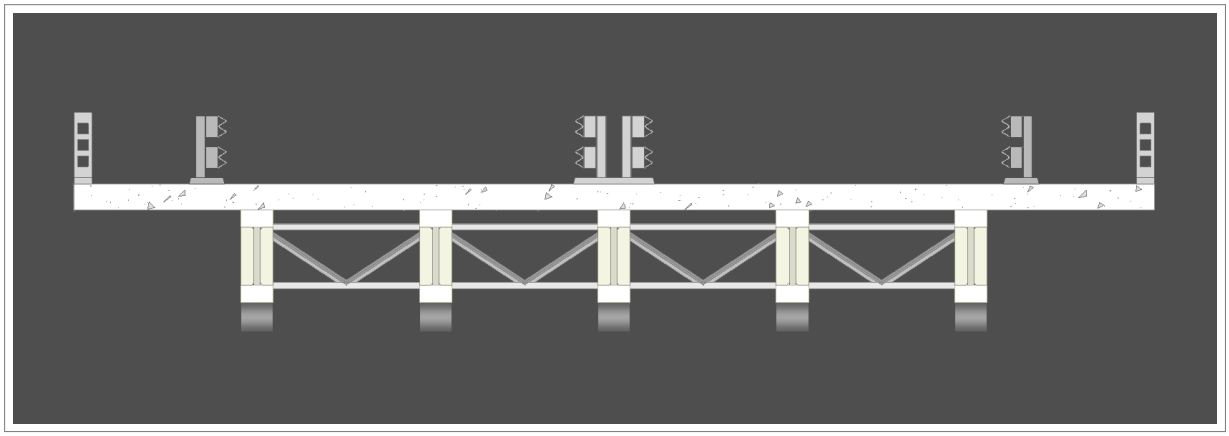


Figure 3.30: Complete 3D bridge model - Front-View

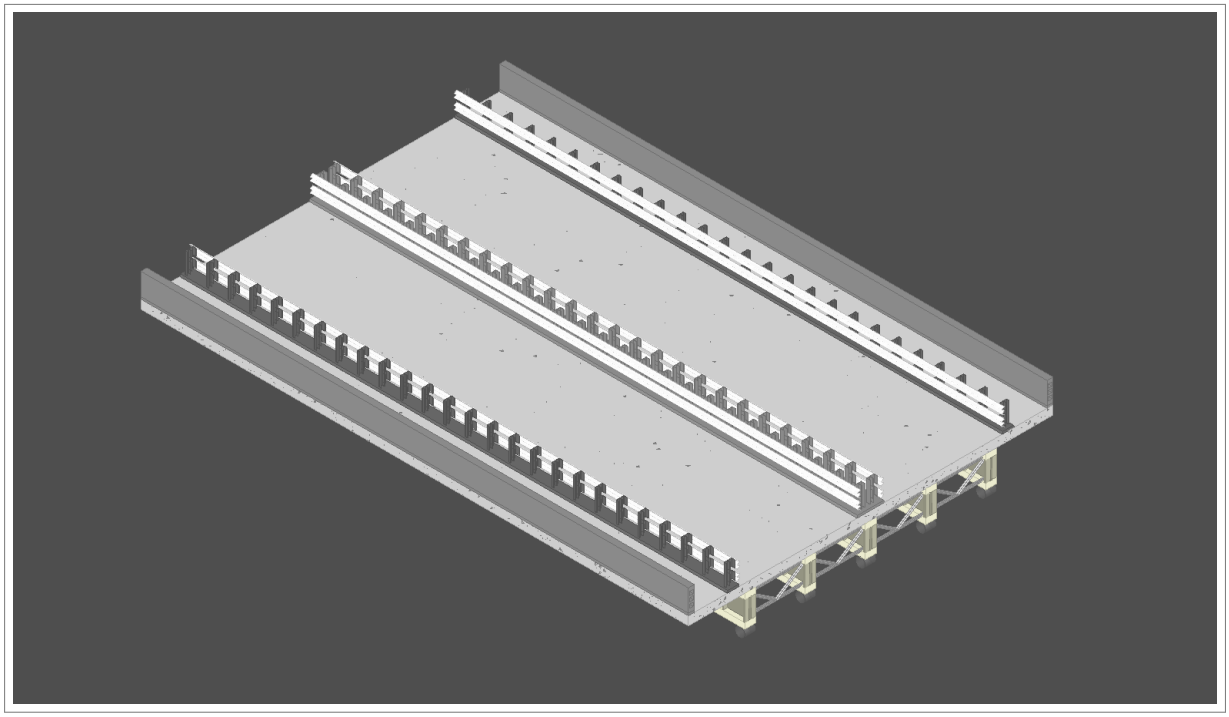


Figure 3.31: Complete 3D bridge model - Iso-View

## 3.10 Conclusion

The developed parametric bridge modeling system demonstrates successful integration of pythonOCC capabilities with IRC:5-2015 specifications to generate production-ready 3D models. The dual-module architecture—comprising the CAD generator (`cad_generator.py`) and interactive 3D viewer (`cad_3d.py`)—enables rapid design iterations through parameter adjustments while maintaining code compliance and geometric accuracy.

The modular design allows independent development and testing of individual bridge components, while the comprehensive parameter set provides flexibility for diverse project requirements. The interactive visualization module enhances the user experience by providing intuitive component exploration and export capabilities.

Future enhancements could include:

- Real-time parameter editing within the 3D viewer
- Structural analysis integration (stress, deflection)
- Automated bill of materials generation
- Multi-span bridge support
- Curved alignment capabilities
- Integration with BIM (Building Information Modeling) workflows

This system establishes a robust foundation for automated bridge design workflows in civil engineering practice.

# Chapter 4

# Internship Task -2: CAD Image Generation and Bug Fixing in Osdag

## 4.1 Task 2: Problem Statement

The Osdag software required automated CAD image generation for various structural steel connection modules to provide visual representations of designed connections. Additionally, several bugs in the existing codebase needed identification and resolution to improve software stability and user experience.

## 4.2 Task 2: Tasks Done

- Developed and implemented CAD image generation functionality for 11 Osdag modules
- Created automated scripts to generate 2D/3D representations of structural steel connections including:
  - Beam-to-Beam End Plate connections
  - Cover Plate Bolted connections
  - Fin Plate connections
  - And 8 other connection types

- Standardized CAD generation workflow across all modules
- Identified and documented bugs in the existing codebase through testing
- Fixed critical bugs affecting module functionality and user interface
- Conducted testing and validation of generated CAD images for accuracy
- Collaborated with the development team to integrate CAD generation features

## 4.3 Task 2: Documentation

### 4.3.1 CAD Image Generation Approach

The CAD generation process follows a systematic approach across all 11 modules:

1. Create mock configuration objects with module-specific parameters (dimensions, bolt layouts, spacing)
2. Initialize structural components (beams, columns, plates, stiffeners)
3. Generate connection elements (bolts, nuts, welds)
4. Assemble complete 3D model using module's specific CAD class
5. Apply material-specific colors and edge styling
6. Set standardized camera views for consistency
7. Export high-quality PNG images

### 4.3.2 Implementation Structure

#### 4.3.2.1 Mock Configuration Setup

Each module requires specific geometric and connection parameters. The configuration varies based on connection type:

Listing 4.1: Configuration Structure - Beam-to-Beam End Plate

```
class MockBBE:
    # Connection parameters
```

```
    endplate_type = "Extended Both Ways - Reversible Moment"
    beam_D = 400.0           # Beam depth
    beam_bf = 165.0          # Beam flange width
    ep_width_provided = 190.0    # End plate width
    ep_height_provided = 700.0   # End plate height

    # Bolt configuration
    bolt_diameter_provided = 20.0
    bolt_numbers = 16
    bolt_row = 8
    bolt_column = 2

    # Spacing parameters
    pitch_distance_provided = 70.0
    gauge_distance_provided = 104.0
    edge_distance_provided = 40.0
```

Listing 4.2: Configuration Structure - Fin Plate Connection

```
class FinPlateConfig:
    def __init__(self):
        self.gauge_provided = PITCH       # 105.0
        self.pitch_provided = GAUGE       # 0.0
        self.end_dist_provided = 40.0
        self.edge_dist_provided = 40.0
        self.gap = 10.0
        self.bolts_one_line = 2
        self.bolt_line = 1
```

#### 4.3.2.2 Component Initialization

Structural components are instantiated using Osdag's CAD library classes:

Listing 4.3: Creating Structural Components

```
# I-Section members (Beams/Columns)
beam = ISection(
```

```
    B=beam_bf,       # Flange width
    T=beam_tf,       # Flange thickness
    D=beam_D,        # Depth
    t=beam_tw,       # Web thickness
    R1=beam_R1,      # Root radius
    R2=beam_R2,      # Toe radius
    alpha=beam_alpha,
    length=beam_length,
    notchObj=None
)

# Connection plates
plate = Plate(
    L=plate_height,      # Length/Height
    W=plate_width,       # Width
    T=plate_thickness    # Thickness
)

# Bolt and nut assemblies
bolt = Bolt(
    R=bolt_diameter * 1.5 / 2,  # Head radius
    T=bolt_diameter * 0.6,       # Head thickness
    H=50,                        # Bolt length
    r=bolt_diameter / 2          # Shank radius
)

nut = Nut(
    R=bolt_diameter * 1.5 / 2,
    T=bolt_diameter,
    H=bolt_diameter,
    innerR1=bolt_diameter / 2
)

# Welds
```

```
weld = FilletWeld(
    b=weld_size,
    h=weld_size,
    L=weld_length
)
```

#### 4.3.2.3 Nut-Bolt Array Generation

Bolt patterns are created using module-specific array classes:

Listing 4.4: Bolt Array Placement

```
# Calculate grip length for proper nut placement
nut_space = plate_thickness + beam_web_thickness + nut.T

# Create bolt array with configuration
nut_bolt_array = NutBoltArray(
    config,
    nut,
    bolt,
    total_bolts,
    nut_space
)
```

#### 4.3.2.4 3D Model Assembly and Visualization

Listing 4.5: CAD Assembly and Rendering

```
# Assemble complete connection model
connectivity = ColFlangeBeamWeb(
    column, beam, weld, plate,
    nut_bolt_array, gap
)
connectivity.create_3dmodel()

# Define material colors
beam_color = Quantity_Color(100/255.0, 100/255.0, 70/255.0,
```

```
                                Quantity_TOC_RGB)
plate_color = Quantity_Color(47/255.0, 47/255.0, 35/255.0,
                                 Quantity_TOC_RGB)
bolt_color = Quantity_Color(255/255.0, 0/255.0, 0/255.0,
                                Quantity_TOC_RGB)
weld_color = Quantity_Color(Quantity_NOC_SADDLEBROWN)
edge_color = Quantity_Color(Quantity_NOC_BLACK)

# Display with edge coloring
models = connectivity.get_models()
for shape, color in zip(models, [column_color, beam_color,
                                    plate_color, weld_color]):
    display.DisplayShape(shape, color=color, update=False)
    color_the_edges(shape, display, edge_color, 0.5)

# Set camera view and export
display.View.SetProj(-1, 1, 1)
display.View.SetUp(-1, 1, -1)
display.View.FitAll()
display.View.SetEye(1000, 0, -500)
display.View.Dump("module_cad.png")
```

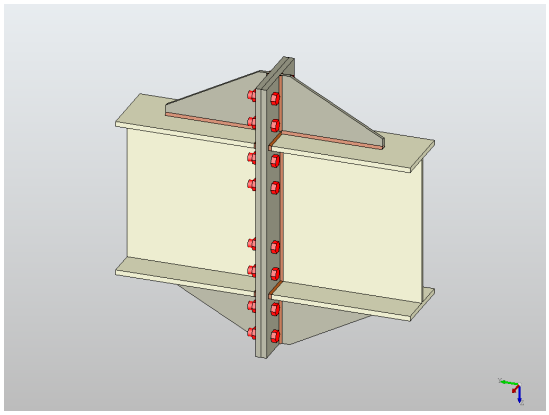
### 4.3.3 Key Implementation Features

- **Parametric Design**: All dimensions are configurable through mock objects, allowing easy modification for different connection scenarios
- **Modular Architecture**: Each connection type uses specialized CAD classes (BBEndplate_cadFile, BBCoverPlateBoltedCAD, ColFlangeBeamWeb, etc.)
- **Standardized Coloring**: Consistent color scheme across modules (beams in brown, plates in dark gray, bolts in red, welds in saddle brown)
- **Edge Enhancement**: Black edge coloring with 0.5 width for improved visualization

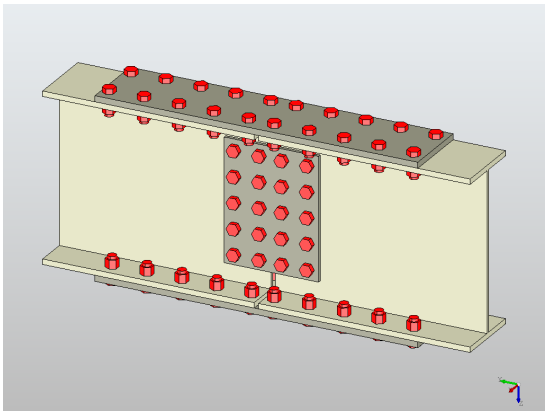
- **Optimized Views**: Standardized camera positioning for professional presentation

### 4.3.4 Sample Generated CAD Images

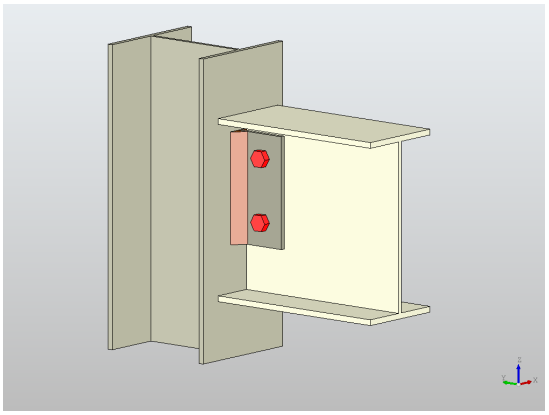
Representative CAD visualizations from three different connection types are shown below to demonstrate the rendering quality and detail achieved across all 11 modules:



(a) Beam-to-Beam End Plate Connection



(b) Cover Plate Bolted Connection



(c) Fin Plate Connection

Figure 4.1: Sample CAD images generated for different Osdag connection modules, demonstrating consistent rendering quality and detail across all 11 implemented modules

### 4.3.5 Bug Fixes

#### 4.3.5.1 Bug 1: Save 3D CAD Functionality Not Working for Multiple Modules

**Affected Modules:**

- Lap Joint Bolted

- Lap Joint Welded
- Butt Joint Bolted
- Butt Joint Welded
- Column to Column Cover Plate Welded
- Column to Column Cover Plate Bolted
- Strut Bolted to End Gusset
- Strut Welded to End Gusset
- Axially Loaded Column
- Simply Supported Beam
- Cantilever Beam
- Plate Girder
- Purlin

**Problem Description:** The `create2Dcad()` method in `common_logic.py` was failing to properly handle CAD model generation and fusion for several modules. The primary issues were:

1. Missing or incomplete conditional logic for newer connection types
2. Improper handling of compound shapes and nested geometries
3. Inconsistent return types (TopoDS_Shape vs dict vs list) across different modules
4. No generic shape normalization before fusion operations

**Root Cause:** The method lacked a unified approach to handle different CAD object structures returned by various modules. Some modules returned single shapes, others returned lists, dictionaries, or nested combinations, leading to fusion failures.

**Solution Implemented:**

**1. Added Helper Functions for Generic Shape Handling:**

Listing 4.6: Shape Normalization Helper Function

```
def _flatten(obj):
    """
    Normalize CAD output:
    - TopoDS_Shape      -> [shape]
    - list / tuple      -> flat [shapes]
    - dict              -> flatten dict values
    - None              -> []
    """
    if obj is None:
        return []

    if isinstance(obj, dict):
        out = []
        for v in obj.values():
            out.extend(_flatten(v))
        return out

    if isinstance(obj, (list, tuple)):
        out = []
        for i in obj:
            out.extend(_flatten(i))
        return out

    return [obj]
```

Listing 4.7: Compound Shape Explosion Function

```
def _explode_compound(shape):
    """
    If shape is a compound, extract all solids inside it.
    Otherwise return the shape as-is.
    """
    from OCC.Core.TopExp import TopExp_Explorer
    from OCC.Core.TopAbs import TopAbs_SOLID
```

```
    solids = []
    exp = TopExp_Explorer(shape, TopAbs_SOLID)
    while exp.More():
        solids.append(exp.Current())
        exp.Next()

    # If no solids found, return original shape
    return solids if solids else [shape]
```

**2. Added Module-Specific Logic:**

Listing 4.8: Lap Joint Bolted Connection Logic

```
elif self.mainmodule == 'Lap Joint Bolted Connection':
    if self.component == "Plate1":
        final_model = self.plate1_model
    elif self.component == "Plate2":
        final_model = self.plate2_model
    elif self.component == "Connector":
        cadlist = self.bolt_models + self.nuts_models
    else:
        final_model = self.assembly
```

Listing 4.9: Butt Joint Welded Connection Logic

```
elif self.mainmodule == 'Butt Joint Welded Connection':
    if self.component == "Plate1":
        final_model = self.plate1_model
    elif self.component == "Plate2":
        final_model = self.plate2_model
    elif self.component == "Cover Plate":
        cadlist = []
        # Top cover plate (always present)
        if self.platec_model:
            cadlist.append(self.platec_model)
        # Bottom cover plate (Double-Cover case)
```

```
        if hasattr(self, 'platec2_model') and self.platec2_model:
            cadlist.append(self.platec2_model)
        # Packing plates (optional)
        if hasattr(self, 'packing_plate1_model'):
            cadlist.append(self.packing_plate1_model)
        if hasattr(self, 'packing_plate2_model'):
            cadlist.append(self.packing_plate2_model)
    elif self.component == "Weld":
        cadlist = self.welds_models
    else:
        final_model = self.assembly
```

**3. Unified Shape Collection and Fusion:**

Listing 4.10: Generic Shape Normalization and Fusion

```
# Collect all shapes from final_model and cadlist
shapes = []
shapes.extend(_flatten(final_model))
shapes.extend(_flatten(cadlist))

# Explode compounds to individual solids
normalized = []
for s in shapes:
    if isinstance(s, TopoDS_Shape):
        normalized.extend(_explode_compound(s))

shapes = normalized

if not shapes:
    return None

# Fuse all shapes into single solid
result = shapes[0]
for shp in shapes[1:]:
    result = BRepAlgoAPI_Fuse(result, shp).Shape()
```

```
# Register with memory manager to prevent garbage collection
self._register_shapes(result)

return result
```

**Impact:**

- Successfully enabled 3D CAD export for 13 previously non-functional modules
- Improved code maintainability through generic helper functions
- Eliminated shape fusion errors caused by type inconsistencies
- Prevented memory leaks through proper shape registration

**Testing:** All affected modules were tested by:

1. Generating CAD models with various component configurations
2. Exporting individual components and full assemblies
3. Verifying proper shape fusion without geometry errors
4. Confirming successful .step/.iges file generation

# Chapter 5

# Conclusions

## 5.1 Tasks Accomplished

During the fellowship at FOSSEE, IIT Bombay, the following tasks were successfully completed:

### 5.1.1 Task 1: 3D CAD Modeling of Steel Girder Bridge

#### 5.1.1.1 Parametric Bridge Component Development

- Developed a comprehensive parametric 3D CAD modeling system for short-span steel girder bridge superstructures using pythonOCC library
- Created modular component factories for all major bridge elements including:
  - Plate girders with web, flanges, and intermediate/bearing stiffeners
  - Reinforced concrete deck slabs with realistic surface textures
  - Cross bracing systems (X-type and K-type configurations)
  - IRC:5-2015 compliant crash barriers (rigid RCC and semi-rigid metallic with W-beams)
  - Median barriers (raised kerb, RCC crash barrier, and metallic variants)
  - Pedestrian railings (RCC and steel tubular designs)
  - Support structures (triangular roller and cylindrical pin bearings)
- Implemented advanced geometric features:

  - Skew bridge geometry with automatic component positioning
  - Dynamic deck width calculation based on footpath configuration
  - Parametric stiffener placement with chamfered hexagonal profiles
  - Corrugated W-beam profile generation using Gaussian wave functions

#### 5.1.1.2 Interactive 3D Visualization System

- Developed an integrated 3D viewer using PySide6 and pythonOCC for real-time visualization
- Implemented interactive features including:
  - Multi-select component visibility control with standardized color schemes
  - View cube navigation for quick orientation changes
  - Zoom controls and automatic fit-to-view functionality
  - Component hover highlighting and tooltip identification
- Created organized data structures for efficient component management and rendering
- Ensured stable initialization through deferred display context setup

#### 5.1.1.3 IRC:5-2015 Compliance Implementation

- Integrated IRC:5-2015 specifications for all safety barrier components
- Developed automated specification module that returns compliant design parameters
- Implemented multiple barrier configurations (Flexible, Semi-Rigid, Rigid) with proper dimensional accuracy
- Ensured all geometric profiles match IRC standards for crash barriers, median barriers, and railings

#### 5.1.1.4 System Architecture and Integration

- Designed dual-module architecture separating CAD generation (`cad_generator.py`) from visualization (`cad_3d.py`)
- Implemented comprehensive parameter configuration system covering 50+ adjustable parameters
- Created hierarchical positioning system for automatic component placement
- Developed coordinate system management for consistent geometric assembly

### 5.1.2 Task 2: CAD Image Generation and Bug Fixing in Osdag

#### 5.1.2.1 Automated CAD Image Generation

- Developed and implemented automated CAD image generation scripts for 11 Osdag structural steel connection modules
- Standardized CAD visualization workflow across all modules including:
  - Mock configuration object creation with module-specific parameters
  - Structural component initialization (beams, columns, plates, stiffeners)
  - Connection element generation (bolts, nuts, welds)
  - Material-specific color application and edge styling
  - Standardized camera view positioning for professional presentation
- Implemented parametric design approach allowing easy dimension modification
- Created consistent color schemes across modules (beams, plates, bolts, welds)
- Generated high-quality PNG exports for documentation and user guidance

#### 5.1.2.2 Critical Bug Identification and Resolution

- Identified critical bug in `create2Dcad()` method affecting 13 Osdag modules where 3D CAD export functionality was completely non-functional
- Analyzed root causes including:

  - Inconsistent return types (TopoDS_Shape vs dict vs list) across modules
  - Improper handling of compound shapes and nested geometries
  - Missing conditional logic for newer connection types
  - Lack of generic shape normalization before fusion operations
- Implemented comprehensive solution including:
  - Generic shape flattening function (`_flatten()`) to normalize all CAD output types
  - Compound shape explosion function (`_explode_compound()`) to extract solids from complex geometries
  - Module-specific conditional logic for 13 affected connection types
  - Unified shape collection and fusion pipeline with proper memory management
- Successfully restored 3D CAD export (.step/.iges) functionality for all 13 modules
- Conducted comprehensive testing across various component configurations to validate fixes

### 5.1.3 Overall Impact

- **Bridge Design Automation**: Created a production-ready parametric bridge modeling system that enables rapid design iterations while maintaining IRC:5-2015 compliance, significantly reducing manual modeling time
- **Osdag Software Reliability**: Restored critical CAD export functionality affecting 13 modules, improving software stability and user experience for structural engineers
- **Code Quality Improvement**: Enhanced Osdag codebase maintainability through generic helper functions and consistent shape handling patterns
- **Open-Source Contribution**: Contributed reusable, well-documented code that benefits the broader civil and structural engineering community

- **Visualization Enhancement**: Developed intuitive 3D visualization systems for both bridge models and steel connections, improving design understanding and communication

## 5.2 Skills Developed

### 5.2.1 Programming and Software Development

- **Python Development**: Advanced proficiency in object-oriented programming, working with complex class hierarchies, parametric systems, and modular architecture design
- **CAD Library Mastery**: Extensive hands-on experience with pythonOCC (OpenCascade) for:
  - 3D geometric modeling and shape manipulation
  - Boolean operations (fusion, cutting, intersection)
  - B-spline curve generation and surface modeling
  - Coordinate transformations and geometric positioning
  - AIS (Application Interactive Services) object management
- **Debugging and Problem Analysis**: Developed systematic debugging methodology:
  - Root cause analysis of shape fusion failures and type inconsistencies
  - Memory management debugging in CAD applications
  - Trace analysis through complex geometric operations
  - Identification of missing conditional logic through code flow analysis
- **GUI Development**: Experience with PySide6 for creating interactive desktop applications with custom 3D viewer widgets
- **Software Testing**: Implemented comprehensive validation workflows including:
  - Parameter variation testing across multiple configurations

  - Component assembly verification
  - Export format validation (.step, .iges, .png)
  - Geometric accuracy confirmation

### 5.2.2 Technical Documentation

- **LaTeX**: Advanced proficiency in creating professional technical documentation with:
  - Complex multi-chapter structure with proper cross-referencing
  - Code listings with syntax highlighting
  - Mathematical equations and geometric formulas
  - Tables, figures, and multi-panel subfigures
  - Custom styling and formatting
- **Technical Writing**: Developed ability to document:
  - Complex software architecture and system workflows
  - Algorithm implementation details with code examples
  - Problem-solution narratives for bug fixes
  - Parametric configuration specifications
- **Visual Documentation**: Created comprehensive visual aids including:
  - System architecture diagrams using TikZ
  - Annotated CAD screenshots demonstrating features
  - Code flow diagrams and component relationship visualizations

### 5.2.3 Software Engineering Practices

- **Version Control**: Experience with Git for code management, branch management, and collaborative development workflows
- **Code Refactoring**: Improved existing codebase through:

  - Generic helper function extraction
  - Elimination of code duplication
  - Type consistency standardization
  - Memory management improvements

- **Modular Architecture Design**: Created separation of concerns between CAD generation logic and visualization interface
- **API Design**: Experience in designing parametric configuration interfaces and component factory patterns
- **Code Documentation**: Developed well-commented code with clear parameter descriptions and usage examples

### 5.2.4 Domain-Specific Knowledge

- **Engineering Standards Application**: Working knowledge of IRC:5-2015 specifications for bridge safety components, enabling compliant parametric design implementation
- **Structural Connection Types**: Familiarity with common steel connection configurations (end plate, cover plate, fin plate, lap and butt joints) through CAD modeling work
- **CAD Automation Principles**: Understanding of:
  - Parametric design methodologies
  - Geometric constraint systems
  - Automated visualization pipelines
  - Shape fusion and boolean operation optimization
- **Bridge Component Systems**: Basic understanding of bridge superstructure components (plate girders, cross bracing, supports, deck systems) for accurate CAD model assembly

### 5.2.5 Problem-Solving and Analytical Skills

- **Complex Problem Decomposition**: Ability to break down large-scale challenges into manageable components:
  - Bridge system decomposed into 8+ independent component factories
  - Bug fixes approached through systematic root cause analysis
  - Geometric positioning solved through hierarchical calculation systems
- **Geometric Problem Solving**: Developed solutions for:
  - Skew angle implementation with coordinate offset calculations
  - Dynamic component positioning based on variable configurations
  - Compound shape normalization and fusion logic
  - Coordinate system consistency across multiple modules
- **Performance Optimization**: Implemented efficiency improvements:
  - Deferred display initialization to avoid race conditions
  - Batch rendering with single repaint calls
  - Memory-conscious shape registration

### 5.2.6 Professional Skills

- **Self-Directed Learning**: Independently mastered:
  - pythonOCC library through documentation and experimentation
  - IRC:5-2015 specifications through standard document analysis
  - PySide6 GUI framework
  - Advanced LaTeX document preparation
- **Project Management**: Successfully managed:
  - Multiple concurrent tasks (bridge modeling + Osdag bug fixing)
  - Deliverable prioritization and timeline management

  - Quality assurance and testing workflows

- **Open-Source Contribution**: Gained understanding of:
  - Community-driven development practices
  - Code quality standards for collaborative projects
  - Documentation requirements for user-facing software
  - Bug tracking and resolution workflows

- **Interdisciplinary Collaboration**: Applied computer science principles to civil engineering domain problems, bridging theoretical knowledge with practical engineering requirements

## 5.3 Conclusion

This fellowship at FOSSEE, IIT Bombay provided comprehensive experience in parametric CAD automation, software debugging, and technical documentation within the civil and structural engineering domain. The dual-task structure allowed development of both greenfield implementation skills (bridge modeling system from scratch) and legacy code maintenance skills (Osdag bug fixing and enhancement).

The bridge modeling project demonstrated the ability to design and implement complex, multi-component parametric systems with extensive configuration options while maintaining compliance with engineering standards. Creating an integrated 3D visualization system enhanced understanding of GUI development and user experience design.

The Osdag bug fixing work developed critical debugging and code analysis skills, particularly in identifying and resolving shape handling inconsistencies across a large codebase. Successfully restoring functionality to 13 non-functional modules demonstrated the value of systematic problem analysis and generic solution design.

Working on actively-used open-source software provided valuable insight into:

- The importance of code maintainability and documentation
- User-focused design considerations for engineering software
- The impact of software quality on professional engineering workflows

- Collaborative development practices and community contribution

The combination of parametric design implementation, 3D geometric modeling, GUI development, bug fixing, and comprehensive technical documentation has built a strong foundation for future work in:

- CAD automation and engineering software development
- Computational geometry and 3D modeling applications
- Software maintenance and legacy code improvement
- Interdisciplinary applications of computer science to engineering domains

The skills acquired during this fellowship—particularly in pythonOCC mastery, parametric system design, systematic debugging, and professional documentation—will be directly applicable to advanced software development projects in both academic research and industry applications. The experience of contributing meaningful improvements to production software used by practicing engineers has provided valuable perspective on the real-world impact of quality software development.

# Chapter A

# Appendix

## A.1 Technical Reference

The work described in this document is referenced to the following repository and specific commit:

- **Repository:** `https://github.com/Faizan9077/OsdagBridge/commits/builder-dev/`
- **Commit Hash:** `0c5482853ce5da081b308c26d46ca21dd14df532`

## A.2 Work Reports

# Internship Work Report

| | |
|---|---|
| Name: | Faizan Khan |
| Project: | Osdag Bridge 3D CAD |
| Internship: | FOSSEE Winter Internship 2025 |

| DATE | DAY | TASK | Hours Worked |
|---|---|---|---|
| 01-Dec-2025 | Monday | Attended onboarding meeting and received overview of internship tasks and OSDAG project | 3 |
| 02-Dec-2025 | Tuesday | Started environment setup for OSDAG Web, including dependency installation | 5 |
| 03-Dec-2025 | Wednesday | Continued OSDAG Web setup and verified application build and execution | 6 |
| 04-Dec-2025 | Thursday | Began environment setup for OSDAG Desktop application | 5 |
| 05-Dec-2025 | Friday | Exam Break | 0 |
| 06-Dec-2025 | Saturday | Completed OSDAG Desktop setup and validated basic functionality | 6 |
| 07-Dec-2025 | Sunday | Reviewed documentation and previous intern's work related to testing tasks | 4 |
| 08-Dec-2025 | Monday | Prepared for testing task; understood testing workflow and issue reporting process | 5 |
| 09-Dec-2025 | Tuesday | Tested OSDAG modules and raised issues on GitHub before deadline | 6 |
| 10-Dec-2025 | Wednesday | Continued testing OSDAG modules and documented test observations | 6 |
| 11-Dec-2025 | Thursday | Exam Break | 0 |
| 12-Dec-2025 | Friday | Reviewed previous intern's CAD work and studied implementation approach | 5 |
| 13-Dec-2025 | Saturday | Studied 3D CAD modeling techniques and bridge component design | 5 |
| 14-Dec-2025 | Sunday | Initialized bridge CAD project structure and setup development environment | 6 |
| 15-Dec-2025 | Monday | Implemented generic plate creation and I-section assembly for plate girders | 6 |
| 16-Dec-2025 | Tuesday | Added initial X-type cross bracing between girders and crash barrier geometry | 6 |
| 17-Dec-2025 | Wednesday | Exam Break | 0 |
| 18-Dec-2025 | Thursday | Exam Break | 0 |
| 19-Dec-2025 | Friday | Exam Break | 0 |

| | | | |
|---|---|---|---|
| 20-Dec-2025 | Saturday | Exam Break | 0 |
| 21-Dec-2025 | Sunday | Exam Break | 0 |
| 22-Dec-2025 | Monday | Implemented rectangular railing component using reusable prism factory | 5 |
| 23-Dec-2025 | Tuesday | Added trapezoidal crash barrier geometry and alignment fixes | 5 |
| 24-Dec-2025 | Wednesday | Added K-bracing type and top/bottom horizontal members in X-bracing | 6 |
| 25-Dec-2025 | Thursday | Refactored cross bracing into reusable X and K components with bracket options | 6 |
| 26-Dec-2025 | Friday | Added footpath configuration options and crash barrier positioning logic | 6 |
| 27-Dec-2025 | Saturday | Implemented correct crash barrier placement for different footpath configurations | 5 |
| 28-Dec-2025 | Sunday | Integrated girders, deck, bracing, crash barriers in unified assembly | 5 |
| 29-Dec-2025 | Monday | Code cleanup and verification of complete 3D bridge assembly | 6 |
| 30-Dec-2025 | Tuesday | Personal Leave | 0 |
| 31-Dec-2025 | Wednesday | Personal Leave | 0 |
| 01-Jan-2026 | Thursday | Holiday | 0 |
| 02-Jan-2026 | Friday | Modularized bridge CAD code and aligned with project architecture | 6 |
| 03-Jan-2026 | Saturday | Implemented angle and channel section geometry for cross bracing members | 6 |
| 04-Jan-2026 | Sunday | Updated diagonal and horizontal members to support angle and channel sections | 5 |
| 05-Jan-2026 | Monday | Added deck texture module and applied texture to deck side faces | 6 |
| 06-Jan-2026 | Tuesday | Fixed railing geometry, hole placement, and crash barrier shape refinements | 6 |
| 07-Jan-2026 | Wednesday | Minor geometry fixes and component alignment improvements | 5 |
| 08-Jan-2026 | Thursday | Implemented parametric stiffener plate with chamfered hexagonal geometry | 6 |
| 09-Jan-2026 | Friday | Added median barrier component (raised kerb, RCC, metallic) and integrated in bridge | 6 |
| 10-Jan-2026 | Saturday | Implemented double angle and double channel section geometry for bracing | 5 |
| 11-Jan-2026 | Sunday | Registered DOUBLE_CHANNEL in section database and roll rules | 4 |
| 12-Jan-2026 | Monday | Enabled DOUBLE_CHANNEL support in section factory and tested integration | 6 |
| 13-Jan-2026 | Tuesday | Applied parametric double channel sections in bridge bracing system | 6 |
| 14-Jan-2026 | Wednesday | Implemented skew bridge geometry and skew-adjusted component positioning | 6 |
| 15-Jan-2026 | Thursday | Integrated IRC:5-2015 specifications module for barrier compliance | 5 |

| | | | |
|---|---|---|---|
| 16-Jan-2026 | Friday | Implemented semi-rigid metallic crash barriers with W-beam profile generation | 5 |
| 17-Jan-2026 | Saturday | Developed corrugated W-beam profile using Gaussian wave functions | 5 |
| 18-Jan-2026 | Sunday | Implemented single and double W-beam configurations for metallic barriers | 4 |
| 19-Jan-2026 | Monday | Generated 3D CAD images for documentation and reporting | 5 |
| 20-Jan-2026 | Tuesday | Continued CAD image generation and visual refinement | 5 |
| 21-Jan-2026 | Wednesday | Finalized CAD visuals with proper colors and visibility | 5 |
| 22-Jan-2026 | Thursday | Prepared CAD images for submission | 4 |
| 23-Jan-2026 | Friday | Implemented Save 3D CAD functionality | 6 |
| 24-Jan-2026 | Saturday | Tested saved CAD outputs and validated geometry | 5 |
| 25-Jan-2026 | Sunday | Reviewed saved 3D CAD structure and validated exported model integrity | 4 |
| 26-Jan-2026 | Monday | Developed interactive 3D viewer with PySide6 and pythonOCC integration | 6 |
| 27-Jan-2026 | Tuesday | Implemented component visibility controls and multi-select functionality | 6 |
| 28-Jan-2026 | Wednesday | Added view cube navigation and zoom controls to 3D viewer | 5 |
| 29-Jan-2026 | Thursday | Implemented component hover highlighting and tooltip identification system | 6 |
| 30-Jan-2026 | Friday | Debugged viewer initialization timing issues with Qt-pythonOCC integration | 5 |
| 31-Jan-2026 | Saturday | Final integration testing and stability verification of complete bridge CAD system | 5 |

# Bibliography

[1] Siddhartha Ghosh, Danish Ansari, Ajmal Babu Mahasrankintakam, Dharma Teja Nuli, Reshma Konjari, M. Swathi, and Subhrajit Dutta. Osdag: A Software for Structural Steel Design Using IS 800:2007. In Sondipon Adhikari, Anjan Dutta, and Satyabrata Choudhury, editors, *Advances in Structural Technologies*, volume 81 of *Lecture Notes in Civil Engineering*, pages 219–231, Singapore, 2021. Springer Singapore.

[2] FOSSEE Project. FOSSEE News - January 2018, vol 1 issue 3. Accessed: 2024-12-05.

[3] FOSSEE Project. Osdag website. Accessed: 2024-12-05.