FOSSEE Semester Long Internship Report

On

CAD and GUI Development for Osdag

Submitted by

Aryan Gupta

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

Manipal Institute of Technology, Manipal

Manipal, Karnataka, India

Under the Guidance of

Prof. Siddhartha Ghosh

Department of Civil Engineering

Indian Institute of Technology Bombay

Mentors:

Ajmal Babu M S

Parth Karia

Ajinkya Dahale

July 2, 2025

Acknowledgments

I would like to express my sincere gratitude to everyone who supported and guided me throughout the course of this project. This experience has been both enriching and enlightening, and it would not have been possible without the collective efforts and encouragement of many individuals and institutions.

I am deeply thankful to the entire **Osdag team**, especially **Ajmal Babu M. S.**, **Ajinkya Dahale**, and **Parth Karia**, for their constant support, mentorship, and technical guidance throughout the project.

I extend my heartfelt thanks to **Prof. Siddhartha Ghosh**, Principal Investigator of the Osdag Project, Department of Civil Engineering, **IIT Bombay**, for his vision and leadership, which served as the foundation of this work.

I am also grateful to **Prof. Kannan M. Moudgalya**, Principal Investigator of the **FOSSEE** Project, Department of Chemical Engineering, **IIT Bombay**, for providing me with the opportunity to contribute to this impactful open-source initiative.

Special thanks to **Usha Viswanathan**, **Vineeta Parmar**, and the entire **FOSSEE team** for their support, coordination, and assistance during my project tenure.

I would like to acknowledge the support of the **National Mission on Education through ICT (NMEICT)**, **Ministry of Education (MoE)**, **Government of India**, whose initiative and resources made this project possible.

I am also thankful to my peers and colleagues who collaborated with me and contributed to a productive and engaging working environment.

Finally, I sincerely thank my institute, **Manipal Institute of Technology, Manipal**, and the **Department of Computer Science and Engineering** for their academic support and for providing me with the foundation and encouragement to pursue this opportunity.

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.

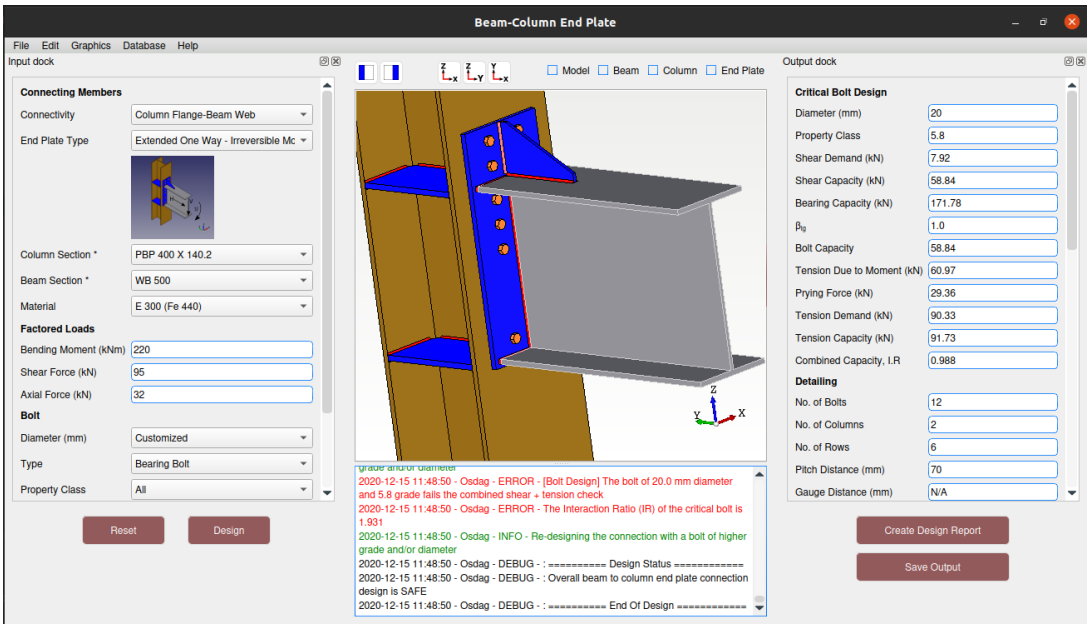


Figure 1.2: Osdag GUI

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.

Chapter 2

Purlin CAD

2.1 Problem Statement

The objective was to develop a 3D modeling capability for C-section purlins. The implementation needed to create accurate geometric representations of steel C-sections based on standard section properties retrieved from a database. The purlin geometry must be parametrically defined to accommodate various standard sizes and lengths as per design requirements.

2.2 Tasks Done

Methodologies and Processes

The implementation utilized the `pythonOCC` library for 3D solid modeling, employing a profile-extrusion approach for C-section generation. The methodology consisted of:

- **Profile Definition:** Creation of a 2D C-section profile in the Y-Z plane using discrete point coordinates.
- **Wire Formation:** Connection of profile points through linear edges to form a closed wire.
- **Face Creation:** Generation of a planar face from the closed wire profile.
- **Extrusion Process:** Linear extrusion of the face along the X-axis to create the 3D beam.

- **Geometric Transformation:** Application of rotation transformation for proper orientation.

Key Implementation Features

- **Parametric Design:** All geometric parameters (depth, flange width, thickness values) are configurable.
- **Database Integration:** Section properties are retrieved from the structural design framework's database.
- **Standard Compliance:** Geometry follows standard C-section proportions and dimensions.
- **3D Visualization:** Integration with OpenCASCADE display system for geometric verification.

Process Flow

Step	Process	Output
1	Database Query	Section properties (depth, flange width, thicknesses)
2	Profile Generation	2D C-section outline
3	Solid Modeling	3D extruded beam geometry
4	Transformation	Properly oriented purlin

2.3 Python Code

This section presents a Python implementation for creating 3D C-section purlin geometry using the Osdag framework. The script generates parametric steel sections based on standard dimensional properties retrieved from a structural database. It automates the geometric modeling process, ensuring accurate representation of purlin cross-sections for structural analysis and visualization.

2.3.1 Description of the Script

The script is structured as follows:

- **Database Integration**: The system retrieves section properties (web thickness, flange thickness, depth, flange width, radii) from the connected structural database using the section designation.
- **Geometric Modeling**: The `create_c_section()` function constructs the 3D geometry through:
  - Definition of 8 key points forming the C-section profile
  - Creation of linear edges connecting consecutive points
  - Formation of a closed wire from the edge sequence
  - Generation of a planar face from the wire
  - Linear extrusion along the longitudinal axis
- **Parametric Design**: All dimensional parameters are configurable:
  - `length`: Purlin span length (converted from meters to millimeters)
  - `depth`: Overall section depth
  - `flange_width`: Width of top and bottom flanges
  - `web_thickness`: Thickness of the vertical web
  - `flange_thickness`: Thickness of horizontal flanges
- **Coordinate System**: The profile is initially created in the Y-Z plane with the web oriented vertically, then transformed through a 90-degree rotation about the Z-axis for standard structural orientation.
- **Output**: The function returns a `TopoDS_Shape` object representing the complete 3D purlin geometry, ready for structural analysis, visualization, or further geometric operations.

2.3.2 Python Code

The Python script is shown below. Each section is commented for clarity.

Listing 2.1: Input parameters to create C-Section Purlin CAD in common_logic.py

```
1 def createPurlin(self):
```

```
    Flex = self.module_class  # Access the module_class instance containing design data

    print(f"This is the module name {Flex}")  # Debug: Print module class reference

    # Retrieve section properties from the database using the section designation
    Flex.section_property = Flex.section_connect_database(Flex, Flex.result_designation)

    # Debug: Print each retrieved section property
    print(f"Flex.section_property.web_thickness : {Flex.section_property.web_thickness}")
    print(f"Flex.section_property.flange_thickness : {Flex.section_property.flange_thickness}")
    print(f"Flex.section_property.depth : {Flex.section_property.depth}")
    print(f"Flex.section_property.flange_width : {Flex.section_property.flange_width}")
    print(f"Flex.section_property.root_radius : {Flex.section_property.root_radius}")
    print(f"Flex.section_property.toe_radius : {Flex.section_property.toe_radius}")
    print(f"Flex.support : {Flex.support}")

    print(dir(Flex.section_property))  # Debug: List all attributes in section_property

    # Create the 3D C-section geometry using retrieved parameters
    purlin = create_c_section(
        length = Flex.length * 1000,  # Convert length from meters to millimeters
        depth = Flex.section_property.depth,
        flange_width = Flex.section_property.flange_width,
        web_thickness = Flex.section_property.web_thickness,
        flange_thickness = Flex.section_property.flange_thickness
    )

```

```
29      return purlin  # Return the 3D purlin shape
```

2.3.3 Explanation of the Code

- **Line 1**: Defines the `createPurlin` method, inside the CommonDesignLogic class.
- **Line 2**: Retrieves the module instance `module_class`, which stores geometry and section information.
- **Line 4**: Prints the module class reference for debugging.
- **Line 7**: Calls the method `section_connect_database()` using the section designation to retrieve section properties such as depth, flange width, and thicknesses from the structural database.
- **Lines 10–16**: Prints individual section properties like web thickness, flange thickness, depth, flange width, root radius, and toe radius for verification.
- **Line 18**: Prints the list of all attributes available in the `section_property` object, useful for debugging and inspection.
- **Lines 21–27**: Calls the `create_c_section()` function with parameters (converted to millimeters where needed) to generate a 3D TopoDS_Shape geometry of the purlin.
- **Line 29**: Returns the generated purlin geometry, ready for display, analysis, or further transformation.

2.3.4 Full code

```
from OCC.Core.BRepBuilderAPI import BRepBuilderAPI_MakeWire,
    BRepBuilderAPI_MakeFace, BRepBuilderAPI_Transform,
    BRepBuilderAPI_MakeEdge
from OCC.Core.BRepPrimAPI import BRepPrimAPI_MakePrism
from OCC.Core.gp import gp_Pnt, gp_Vec, gp_Trsf, gp_Dir, gp_Ax1,
    gp_Pnt
from OCC.Core.TopoDS import TopoDS_Edge
```

```
from OCC.Display.SimpleGui import init_display
import math

def create_c_section(length=1000, depth=200, flange_width=80,
   web_thickness=10, flange_thickness=10):
    # Create points for the C-section profile (in Y-Z plane)
    points = [
        gp_Pnt(0, 0, 0),                          # Bottom-left
           corner
        gp_Pnt(0, 0, depth),                      # Top-left corner
        gp_Pnt(0, -flange_width, depth),          # Top-right of
           upper flange
        gp_Pnt(0, -flange_width, depth-flange_thickness),  #
           Bottom-right of upper flange
        gp_Pnt(0, -web_thickness, depth-flange_thickness), # Top-
           right of web
        gp_Pnt(0, -web_thickness, flange_thickness),       #
           Bottom-right of web
        gp_Pnt(0, -flange_width, flange_thickness),        # Top-
           right of lower flange
        gp_Pnt(0, -flange_width, 0),              # Bottom-right of
            lower flange
    ]

    # Create edges
    edges = []
    for i in range(len(points)-1):
        edge = BRepBuilderAPI_MakeEdge(points[i], points[i+1]).
           Edge()
        edges.append(edge)

    # Close the profile
    edge = BRepBuilderAPI_MakeEdge(points[-1], points[0]).Edge()
    edges.append(edge)
```

```
    # Create wire from edges
    wire_builder = BRepBuilderAPI_MakeWire()
    for edge in edges:
        wire_builder.Add(edge)
    wire = wire_builder.Wire()

    # Create face from wire
    face = BRepBuilderAPI_MakeFace(wire).Face()

    # Extrude along X-axis to create the beam
    vec = gp_Vec(length, 0, 0)
    beam = BRepPrimAPI_MakePrism(face, vec).Shape()

    # Create and apply the rotation transformation
    trsf = gp_Trsf()
    rotation_axis_z = gp_Ax1(gp_Pnt(0, 0, 0), gp_Dir(0, 0, 1))
    trsf.SetRotation(rotation_axis_z, math.pi/2)
    beam_transformed = BRepBuilderAPI_Transform(beam, trsf).Shape
        ()

    return beam_transformed

def main():
    # Initialize display
    display, start_display, add_menu, add_function_to_menu =
        init_display()

    # Create the C-section beam
    beam = create_c_section()

    # Display the beam
    display.DisplayShape(beam, update=True)
```

```
    # Set view
    display.View_Iso()
    display.FitAll()

    # Start the display
    start_display()

if __name__ == "__main__":
    main()
```

2.4 Documentation

Detailed steps to integrate the purlin CAD into the Osdag application were included in the CAD manual, helping new interns speed up and simplify the process.

2.4.1 Directory Structure

```
Osdag
├── osdagMainPage.py
├── Common.py
├── ResourceFiles
│   ├── images
│   ├── last_designs
│   └── ....
├── design_type
├── design_report
├── cad
│   └── items
│       └── purlin.py
└── gui
```

Program Start Calls

The main entry point for the program is `osdagMainPage.py`. To start the program, open the Osdag folder and start the terminal with that path, execute the following command from the project root (src):

```
$ python -m osdag.osdagMainPage
```

Chapter 3

Bolted Lap Joint CAD

3.1 Problem Statement

The objective was to develop a comprehensive 3D modeling capability for bolted lap joint connections within the Osdag framework. The implementation needed to create accurate geometric representations of complete connection assemblies including overlapping plates, bolts, and nuts based on connection design parameters. The joint geometry must be parametrically defined to accommodate various plate dimensions, bolt patterns, and spacing requirements as per structural connection design calculations.

3.2 Tasks Done

Methodologies and Processes

The implementation utilized the `pythonOCC` library for multi-component 3D assembly modeling, employing a component-based approach for connection generation. The methodology consisted of:

- **Component Definition:** Individual creation of plates, bolts, and nuts as separate geometric entities.
- **Spatial Positioning:** Calculation of plate overlap geometry and component placement coordinates.

- **Bolt Pattern Generation:** Systematic positioning of fasteners based on pitch, gauge, edge, and end distances.
- **Assembly Integration:** Boolean operations using `BOPAlgo_Builder` for complete joint assembly.
- **Visualization Setup:** Color-coded component display for enhanced geometric verification.

Key Implementation Features

- **Multi-Component Assembly:** Separate modeling of plates, bolts, and nuts with precise spatial relationships.
- **Connection Design Integration:** Parameters retrieved from structural connection design calculations.
- **Bolt Pattern Automation:** Systematic generation of bolt positions based on standard spacing rules.
- **Material Visualization:** Color-coded components for clear assembly identification.
- **Overlap Geometry:** Accurate plate positioning with specified overlap length.

Process Flow

Step	Process	Output
1	Parameter Extraction	Connection properties (plate dimensions, bolt specifications)
2	Component Creation	Individual plates, bolts, nuts
3	Pattern Generation	Bolt position coordinates
4	Assembly Process	Complete connection assembly

3.3 Python Code

This section presents a Python implementation for creating 3D bolted lap joint assemblies within the Osdag framework. The script generates complete connection geometries

including overlapping plates, bolts, and nuts based on structural connection design parameters retrieved from the design framework. It automates the assembly modeling process, ensuring accurate representation of bolted connections for structural analysis and visualization.

3.3.1 Description of the Script

The script is structured as follows:

- **Connection Integration**: The system retrieves connection design parameters including:
  - Plate dimensions (thickness, width, overlap length)
  - Bolt specifications (diameter, pattern arrangement)
  - Spacing parameters (pitch, gauge, edge distances, end distances)
  - Bolt pattern configuration (rows, columns, total number)
- **Component Modeling**: The `create_bolted_lap_joint()` function constructs the assembly through:
  - Creation of two overlapping plates with specified dimensions and positioning
  - Generation of bolt geometries with heads, shafts, and proper lengths
  - Formation of nuts with appropriate threading and positioning
  - Calculation of component spatial relationships and elevations
- **Geometric Calculations**: Critical positioning computations include:
  - `plate2_offset`: Longitudinal positioning for proper overlap
  - Bolt pattern coordinates based on pitch, gauge, edge, and end distances
  - Component elevation for proper through-thickness assembly
  - Bolt length calculation considering plate thicknesses and nut requirements
- **Assembly Parameters**: All connection parameters are configurable:
  - `plate1_thickness`, `plate2_thickness`: Individual plate thicknesses

  - `plate_width`: Width of connection plates
  - `actual_overlap_length`: Length of plate overlap region
  - `bolt_rows`, `bolt_cols`: Bolt pattern arrangement matrix
  - `pitch`: Longitudinal bolt spacing
  - `gauge`: Transverse bolt spacing
  - `edge`, `end`: Distance from plate edges

- **Component Classes**: The implementation utilizes imported component classes:
  - `Plate`: Rectangular plate geometry with positioning methods
  - `Bolt`: Complete bolt assembly with head, shaft, and threading
  - `Nut`: Hexagonal nut geometry with internal threading

- **Assembly Process**: Boolean operations using `BOPAlgo_Builder` combine all components into a unified assembly while maintaining individual component references for selective visualization and analysis.

- **Output**: The function returns multiple objects:
  - Complete assembly shape for unified visualization
  - Individual component references (plates, bolts, nuts) for selective display
  - Color-coded visualization with material assignments

3.3.2 Python Code

The Python script is shown below. Each section is commented for clarity.

Listing 3.1: Input parameters to create Bolted Lap Joint CAD in common_logic.py

```
1 def createBoltedLapJoint(self):
2
3     Conn = self.module_class  # Access the module class containing all
          input parameters
4
5     print("THIS IS CONN")
6     print(Conn)
7
```

```
    # Print all non-method attributes in the module class for debugging
    for attr in dir(Conn):
        if not callable(getattr(Conn, attr)) and not attr.startswith("
            __"):
             print(f"{attr}: {getattr(Conn, attr)}")

    # Print individual design parameters for verification
    print(f"Plate 1 Thickness: {float(Conn.plate1thk)}")
    print(f"Plate 2 Thickness: {float(Conn.plate2thk)}")
    print(f"Plate Width: {float(Conn.width)}")
    print(f"Bolt Diameter: {Conn.bolt.bolt_diameter_provided}")
    print(f"Actual Overlap Length: {Conn.len_conn}")
    print(f"Bolt Columns: {Conn.cols}")
    print(f"Bolt Rows: {Conn.rows}")
    print(f"Number of Bolts: {Conn.number_bolts}")
    print(f"Pitch: {Conn.final_pitch}")
    print(f"Gauge: {Conn.final_gauge}")
    print(f"Edge Distance: {Conn.final_edge_dist}")
    print(f"End Distance: {Conn.final_end_dist}")

    # Call function to create bolted lap joint with the provided
        parameters
    lap_joint, plate1, plate2, bolts, nuts = create_bolted_lap_joint(
        plate1_thickness = float(Conn.plate1thk),
        plate2_thickness = float(Conn.plate2thk),
        plate_width = float(Conn.width),
        bolt_dia = Conn.bolt.bolt_diameter_provided,
        actual_overlap_length = Conn.len_conn,
        bolt_cols = Conn.cols,
        bolt_rows = Conn.rows,
        number_bolts = Conn.number_bolts,
        pitch = Conn.final_pitch,
        gauge = Conn.final_gauge,
        edge = Conn.final_edge_dist,
        end = Conn.final_end_dist
    )

    return lap_joint, plate1, plate2, bolts, nuts  # Return the full
        assembly and its components
```

3.3.3 Explanation of the Code

- **Line 1**: Defines the `createBoltedLapJoint` method within a class, responsible for creating a bolted lap joint geometry.
- **Line 3**: Retrieves the module instance `module_class`, which holds all connection parameters and component attributes.
- **Lines 9–11**: Prints the module instance and lists all of its non-method attributes for debugging and inspection.
- **Lines 14–25**: Prints individual values such as plate thicknesses, bolt diameter, overlap length, and spacing parameters like pitch, gauge, edge, and end distances to verify correct parameter extraction.
- **Lines 28–41**: Calls the function `create_bolted_lap_joint()` using the extracted parameters to generate the complete assembly including plates, bolts, and nuts.
- **Line 43**: Returns the full lap joint assembly along with references to its individual components for visualization or further processing.

3.3.4 Full code

```
import numpy
from OCC.Display.SimpleGui import init_display
from OCC.Core.BRepAlgoAPI import BRepAlgoAPI_Fuse
from OCC.Core.BOPAlgo import BOPAlgo_Builder
from OCC.Core.Quantity import Quantity_NOC_SADDLEBROWN,
   Quantity_NOC_GRAY,Quantity_NOC_BLUE1,Quantity_NOC_RED
from OCC.Core.Graphic3d import *
from OCC.Core.BRepPrimAPI import BRepPrimAPI_MakeSphere
# Import the component classes
from ...items.bolt import Bolt
from ...items.nut import Nut
from ...items.plate import Plate
```

```
def create_bolted_lap_joint(plate1_thickness = 16,
   plate2_thickness = 8, plate_width = 100, bolt_dia = 16,
   actual_overlap_length=50,
                            bolt_rows=4,bolt_cols=2,pitch=20,
                               gauge=20,edge=12,end=13.6,
                               number_bolts=7):

    plate_length = 3 * actual_overlap_length

    # Calculate the offset of the second plate
    plate2_offset = plate_length - actual_overlap_length

    nut_thickness = 3.0
    # Bolt parameters
    bolt_head_radius = bolt_dia/2
    bolt_head_thickness = 3.0
    bolt_length = (plate1_thickness + plate2_thickness) +
       nut_thickness   # Enough to go through both plates
    bolt_shaft_radius = 1.5

    # Nut parameters
    nut_radius = bolt_head_radius

    nut_height = bolt_head_radius
    nut_inner_radius = bolt_shaft_radius

    # Create the first plate
    # Position it at the origin
    origin1 = numpy.array([0.0, 0.0, 0.0]) # Global origin lies
       at midpoint of plate 1
    uDir1 = numpy.array([0.0, 0.0, 1.0])  # Points along Z axis (
       height)
```

```
wDir1 = numpy.array([1.0, 0.0, 0.0])  # Points along X axis (
    length)

plate1 = Plate(plate_length, plate_width, plate1_thickness)
plate1.place(origin1, uDir1, wDir1)
plate1_model = plate1.create_model()

# Create the second plate
# Position it so that it properly overlaps with the first
    plate
# The second plate is elevated by plate1_thickness and offset
     in Y direction

origin2 = numpy.array([0.0, plate2_offset, 0.5*(
    plate1_thickness+plate2_thickness)])
uDir2 = numpy.array([0.0, 0.0, 1.0])
wDir2 = numpy.array([1.0, 0.0, 0.0])

plate2 = Plate(plate_length, plate_width, plate2_thickness)
plate2.place(origin2, uDir2, wDir2)
plate2_model = plate2.create_model()

bolt_positions=[]

# Calculate bolt positions
count = 0
exit_loops = False  # Flag to break both loops

for col in range(bolt_cols):
    for row in range(bolt_rows):
        if count==number_bolts:
            exit_loops = True
            break  # Break out of the inner loop
```

```
            bolt_positions.append((edge + (row * gauge),
                                   plate_length / 2 -
                                      actual_overlap_length + end +
                                      (col * pitch),
                                   (0.5 * plate1_thickness) +
                                      plate2_thickness))
            count += 1

        if exit_loops:  # Check flag to break outer loop
            break



# Create bolts and nuts at the calculated positions
bolts_models = []
nuts_models = []

bolt_uDir = numpy.array([1.0, 0.0, 0.0])
bolt_shaftDir = numpy.array([0.0, 0.0, -1.0])  # Points
   downward through both plates
for pos in bolt_positions:
    # Start bolts from the top of second plate
    bolt = Bolt(bolt_head_radius, bolt_head_thickness,
       bolt_length, bolt_shaft_radius)
    bolt.place(pos, bolt_uDir, bolt_shaftDir)
    bolt_model = bolt.create_model()
    bolts_models.append(bolt_model)

    # Position nuts at the bottom of the first plate
    nut_origin = numpy.array([pos[0], pos[1], -0.5*
       plate1_thickness])
    nut_uDir = numpy.array([1.0, 0.0, 0.0])
    nut_wDir = numpy.array([0.0, 0.0, -1.0])  # Points
       downward
```

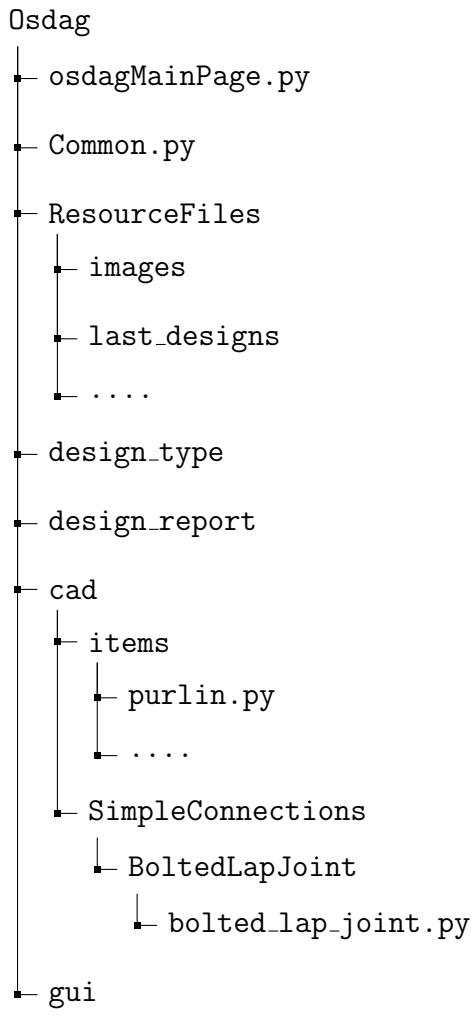
```
        nut = Nut(nut_radius, nut_thickness, nut_height,
            nut_inner_radius)
        nut.place(nut_origin, nut_uDir, nut_wDir)
        nut_model = nut.create_model()
        nuts_models.append(nut_model)

    # Use BOPAlgo_Builder for assembly
    builder = BOPAlgo_Builder()

    # Add all parts to the builder
    builder.AddArgument(plate1_model)
    builder.AddArgument(plate2_model)

    for bolt_model in bolts_models:
        builder.AddArgument(bolt_model)

    for nut_model in nuts_models:
        builder.AddArgument(nut_model)

    # Perform the boolean operation
    builder.Perform()

    # Get the resulting assembly
    assembly = builder.Shape()

    return assembly, plate1_model, plate2_model, bolts_models,
        nuts_models

# Main execution
if __name__ == "__main__":
    # Create the bolted lap joint
    lap_joint, plate1, plate2, bolts, nuts =
        create_bolted_lap_joint()
```

```
# Display the assembly
display, start_display, add_menu, add_function_to_menu =
    init_display()

# Display individual components with different colors for
    better visualization
display.DisplayShape(plate1, material=Graphic3d_NOM_ALUMINIUM
    , update=True)
display.DisplayShape(plate2, update=True)

for bolt in bolts:
    display.DisplayShape(bolt, color=Quantity_NOC_SADDLEBROWN
        , update=True)

for nut in nuts:
    display.DisplayShape(nut, color=Quantity_NOC_SADDLEBROWN,
        update=True)
# Highlight the global origin (0,0,0)
origin_point = BRepPrimAPI_MakeSphere(1).Shape()  # Small
    sphere to mark origin
display.DisplayShape(origin_point, color=Quantity_NOC_RED,
    update=True)

# Alternative: display the full assembly as a single shape
# display.DisplayShape(lap_joint, update=True)
display.set_bg_gradient_color([51, 51, 102], [150, 150, 170])

display.DisableAntiAliasing()
display.FitAll()
start_display()
```

3.4 Documentation

Detailed steps to integrate the Bolted Lap Joint CAD into the Osdag application were included in the CAD manual, helping new interns speed up and simplify the process.

3.4.1 Directory Structure



Chapter 4

Spacing GUI for CADs

4.1 Problem Statement

The objective was to develop a dynamic bolt pattern visualization system within the Osdag application. The implementation needed to create interactive 2D technical drawings that display bolt arrangements with accurate dimensional annotations, including pitch, gauge, edge distances, and end distances. The system must integrate with existing connection design modules to extract spacing parameters.

4.2 Tasks Done

Methodologies and Processes

The implementation utilized the PyQt5 framework for GUI development and technical drawing generation, employing a graphics scene approach for precise dimensional visualization:

- **Parameter Extraction:** Dynamic retrieval of spacing parameters from connection design objects
- **Graphics Scene Creation:** Setup of scalable 2D drawing canvas using `QGraphicsScene`
- **Bolt Pattern Generation:** Systematic positioning of bolt holes based on rows and columns
- **Dimensional Annotation:** Automatic generation of dimension lines with arrows and text labels

- **Interactive Display:** Real-time visualization with zoom and pan capabilities

Key Implementation Features

- **Dynamic Parameter Integration:** Real-time extraction of spacing values from design calculations
- **Professional Drawing Standards:** Technical drawing format with proper dimension lines and annotations
- **Flexible Bolt Patterns:** Support for variable row and column configurations
- **Interactive Visualization:** Scalable graphics with fit-to-view functionality
- **Parameter Display:** Side panel showing all extracted spacing values

Process Overview Table

Step	Process	Output
1	Button Activation	Connection object instantiation
2	Parameter Extraction	Spacing values (pitch, gauge, edge, end)
3	Pattern Generation	Bolt hole positions and plate outline
4	Dimension Creation	Annotated technical drawing

4.3 Python Code

This section presents a Python implementation for creating interactive bolt pattern visualization using PyQt5 framework. The script generates professional technical drawings showing bolt arrangements with comprehensive dimensional annotations based on connection design parameters extracted from the Osdag framework. It provides real-time visualization for design verification and documentation.

Description of the Script

The script for the bolt pattern visualization is modular and follows a clear object-oriented structure. The key components are described below:

GUI Framework Integration

- `BoltPatternGenerator` class inherits from `QMainWindow`, enabling a professional windowing interface
- Resizable layout with split-pane: one for parameter display and one for the drawing canvas
- Utilizes PyQt5's graphics system for rendering technical 2D drawings

Parameter Extraction System

- `get_parameters()` method interfaces with `self.connection.spacing(status=True)` to fetch design data
- Maps internal keys (like 'g1', 'g2', 'pitch') to human-readable labels
- Handles both single-gauge and dual-gauge configurations
- Extracts: pitch, end distance, gauge distances (g1, g2), and edge distances (e1, e2)

Technical Drawing Generation

- `createDrawing()` calculates bolt layout using matrix structure (rows × columns)
- Dynamically computes plate dimensions from spacing parameters
- Renders plate outline, bolt holes, and coordinate-based positioning with proper scaling

Dimensional Annotation System

- `addHorizontalDimension()` and `addVerticalDimension()` draw annotated dimension lines
- Arrowheads rendered using filled polygon graphics
- Dimension text placed clearly above/beside lines, with readable font sizes

Drawing Parameters

- `rows, cols`: Define bolt matrix layout
- `pitch`: Distance between bolts in longitudinal direction

- `gauge1, gauge2`: Lateral distances for one or two gauge lines
- `edge, end`: Margins from edge and end of plate to nearest bolt centers
- `hole_diameter`: Diameter for bolt hole representation

Graphics System Architecture

- Uses `QGraphicsScene` for vector drawing
- `QGraphicsView` enables antialiased rendering for smooth visuals
- Dynamic scene bounding box adjusts based on drawing size + offset margins
- Implements `fitInView()` for automatic zoom-to-fit functionality

Integration Interface

- Integrated with main Osdag system via `run_spacing_script()` function
- Detects connection type and retrieves corresponding bolt arrangement parameters
- Ensures proper window focus and activation during GUI interaction

Professional Drawing Standards

- Follows engineering drafting principles for all visuals
- Uses clear and distinct arrowheads, text labels, and spacing markers
- Color coding applied: Blue for bolt holes, black for dimension lines/text
- All key spacing parameters are clearly labeled in the drawing

Output

- Real-time, scalable visualization window of bolt pattern
- Side panel shows all parameter values dynamically fetched from design object
- Allows immediate visual validation of spacing compliance
- Drawing quality is suitable for professional technical documentation

4.3.1 Python Code

The Python script is shown below. Each section is commented for clarity.

Listing 4.1: Function to create and display the bolt spacing viewer window in ui_template.py

```
def run_spacing_script(self, cols, rows):
    print("Creating spacing window...")

    # Instantiate the BoltPatternGenerator GUI window with required
        parameters
    self.spacing_window = BoltPatternGenerator(self.Obj, cols=cols,
        rows=rows)

    # Set title of the window
    self.spacing_window.setWindowTitle("Spacing Viewer")

    # Raise window to top and activate it
    self.spacing_window.raise()
    self.spacing_window.activateWindow()

    # Show the GUI window
    self.spacing_window.show()
```

4.3.2 Explanation of the Code

- **Purpose:** Launch a window displaying the bolt pattern for a selected CAD joint.
- **Parameters:**
  - `cols`: Number of bolt columns.
  - `rows`: Number of bolts per column.
- **Procedure:**
  1. Instantiates the `BoltPatternGenerator` window using extracted connection object and parameters.
  2. Sets the window title to *Spacing Viewer*.

3. Brings the window to the foreground using `raise()` and `activateWindow()`.

4. Calls `show()` to render the GUI.

4.3.3 Python Code

Listing 4.2: Iterate over button options to determine which operation to execute in the output_button_dialog() function in ui_template.py

```
for op in button_list:
    # Check if current button was clicked
    if op[0] == button.objectName():
        print(op)
        tup = op[3]       # Extract function metadata tuple
        title = tup[0]
        fn = tup[1]

        # Extract class name from qualified function name
        cls = fn.__qualname__.split('.')[0]

        # If the operation is 'spacing', trigger spacing window
        if op[0] == 'spacing':
            module = inspect.getmodule(fn)        # Get module of the
                function
            cls_obj = getattr(module, cls)        # Get class object
            self.Obj = cls_obj()                  # Instantiate class

            # Check if object uses 'spting_leg' structure (likely a
                typo for 'spring_leg')
            if hasattr(self.Obj, 'spting_leg') and \
               hasattr(self.Obj.spting_leg, 'bolt_line') and \
               hasattr(self.Obj.spting_leg, 'bolts_one_line'):

                # Use bolt parameters from spting_leg
                self.run_spacing_script(cols=self.Obj.spting_leg.
                    bolt_line,
                                        rows=self.Obj.spting_leg.
                                            bolts_one_line)
            else:
                # Use bolt parameters from plate object
```

```
28                 self.run_spacing_script(cols=self.Obj.plate.bolt_line,
29                                         rows=self.Obj.plate.
                                               bolts_one_line)
30             break
```

4.3.4 Explanation of the Code

- **Role:** Responds to user interaction with the spacing button in the output interface.
- **Flow:**
  1. Iterates through the `button_list` to identify which button was triggered.
  2. Extracts the function metadata from the operation tuple.
  3. Dynamically determines the class associated with the triggered function using `inspect` and `getattr`.
  4. Instantiates the corresponding connection object.
  5. Based on availability, retrieves bolt layout parameters from either:
     - `spting_leg` structure (if present), or
     - `plate` structure (as fallback).
  6. Passes the retrieved parameters to `run_spacing_script()` to display the viewer.

4.3.5 Full code

```
import sys
from PyQt5.QtWidgets import (QApplication, QMainWindow, QWidget,
   QVBoxLayout,
                             QHBoxLayout, QLabel, QGraphicsView,
                             QGraphicsScene)
from PyQt5.QtGui import QPixmap
from PyQt5.QtCore import Qt, QRectF
from PyQt5.QtGui import QPainter, QPen, QFont
from PyQt5.QtGui import QPolygonF, QBrush
from PyQt5.QtCore import QPointF
```

```
from ..Common import *

class BoltPatternGenerator(QMainWindow):
    def __init__(self, connection_obj, rows=3, cols=2):
        super().__init__()
        self.connection = connection_obj
        self.rows = rows
        self.cols = cols
        self.initUI()

    def initUI(self):
        self.setWindowTitle('Bolt Pattern Generator')
        self.setGeometry(100, 100, 800, 500)

        # Main layout
        main_layout = QHBoxLayout()

        # Left panel for parameter display
        left_panel = QWidget()
        left_layout = QVBoxLayout()

        # Parameter display labels
        params = self.get_parameters()

        # Display the parameter values
        for key, value in params.items():
            param_layout = QHBoxLayout()
            param_label = QLabel(f'{key.title()} Distance (mm):')
            value_label = QLabel(f'{value}')
            param_layout.addWidget(param_label)
            param_layout.addWidget(value_label)
            left_layout.addLayout(param_layout)

        left_layout.addStretch()
```

```
        left_panel.setLayout(left_layout)

        # Right panel for the drawing using QGraphicsView
        self.scene = QGraphicsScene()
        self.view = QGraphicsView(self.scene)
        self.view.setRenderHint(QPainter.Antialiasing)

        # Create and add the drawing to the scene
        self.createDrawing(params)

        # Add panels to main layout
        main_layout.addWidget(left_panel, 1)
        main_layout.addWidget(self.view, 3)

        # Set main widget
        main_widget = QWidget()
        main_widget.setLayout(main_layout)
        self.setCentralWidget(main_widget)

        # Ensure the view shows all content
        self.view.fitInView(self.scene.sceneRect(), Qt.
            KeepAspectRatio)

    def get_parameters(self):
        spacing_data = self.connection.spacing(status=True)  #
            Get actual values
        param_map = {}

        for item in spacing_data:
            key, _, _, value = item

            if key == KEY_OUT_PITCH:
                param_map['pitch'] = float(value)
            elif key == KEY_OUT_END_DIST:
```

```
                param_map['end'] = float(value)
            elif key == KEY_OUT_GAUGE1:
                param_map['gauge1'] = float(value)
            elif key == KEY_OUT_GAUGE2:
                param_map['gauge2'] = float(value)
            elif key == KEY_OUT_GAUGE:
                param_map['gauge'] = float(value)
            elif key == KEY_OUT_EDGE_DIST:
                param_map['edge'] = float(value)

        # Add hardcoded hole diameter
        param_map['hole'] = 10.0

        print("Extracted parameters:", param_map)

        return param_map

    def createDrawing(self, params):
        # Extract parameters
        pitch = params['pitch']
        end = params['end']
        if 'gauge' in params:
            gauge = params['gauge']
        else:
            gauge1 = params['gauge1']
            gauge2 = params['gauge2']
        edge = params['edge']
        hole_diameter = params['hole']

        # Calculate dimensions
        if 'gauge' in params:
            gauge1 = gauge
            gauge2 = 0
        width = gauge1 + gauge2 + edge
```

```
        height = 2 * end + (self.rows - 1) * pitch

        # Set up pens
        outline_pen = QPen(Qt.blue, 2)
        dimension_pen = QPen(Qt.black, 1.5)

        # Dimension offsets
        h_offset = 40
        v_offset = 60

        # Create scene rectangle with extra space for dimensions
        self.scene.setSceneRect(-h_offset, -v_offset,
                                width + 2*v_offset, height + 2*
                                   h_offset)

        # Draw rectangle
        self.scene.addRect(0, 0, width, height, dimension_pen)

        # Draw holes
        for row in range(self.rows):
            for col in range(self.cols):
                x = gauge1 - hole_diameter/2 if col == 0 else
                   gauge1 + gauge2 - hole_diameter/2
                y = end + row * pitch - hole_diameter/2
                self.scene.addEllipse(x, y, hole_diameter,
                   hole_diameter, outline_pen)

        # Add dimensions
        self.addDimensions(params, dimension_pen)

    def addDimensions(self, params, pen):
        # Extract parameters
        pitch = params['pitch']
        end = params['end']
```

```
        if 'gauge' in params:
            gauge = params['gauge']
        else:
            gauge1 = params['gauge1']
            gauge2 = params['gauge2']
        edge = params['edge']

        if 'gauge' in params:
            gauge1 = gauge
            gauge2 = 0

        width = gauge1 + gauge2 + edge
        height = 2 * end + (self.rows - 1) * pitch

        # Offsets for dimension lines
        h_offset = 20
        v_offset = 30

        # Add horizontal dimensions
        self.addHorizontalDimension(0, -h_offset, gauge1, -
            h_offset, str(gauge1), pen)

        if gauge2 > 0:
            self.addHorizontalDimension(gauge1, -h_offset, gauge1
                + gauge2, -h_offset, str(gauge2), pen)

        self.addHorizontalDimension(gauge1 + gauge2, -h_offset,
            width, -h_offset, str(edge), pen)

        # Add bottom horizontal dimension
        self.addHorizontalDimension(0, height + h_offset, width,
            height + h_offset,
                                    str(edge + gauge1 + gauge2),
                                        pen)
```

```
        # Add vertical dimensions
        self.addVerticalDimension(width + v_offset, 0, width +
            v_offset, end, str(end), pen)
        for i in range(self.rows - 1):
            self.addVerticalDimension(width + v_offset, end + i *
                pitch, width + v_offset, end + (i + 1) * pitch,
                str(pitch), pen)

        # Add bottom end distance dimension
        self.addVerticalDimension(width + v_offset, height, width
            + v_offset, height - end, str(end), pen)

        # Add left side dimension
        total_height = 2 * end + (self.rows - 1) * pitch
        self.addVerticalDimension(-v_offset, 0, -v_offset,
            total_height, str(total_height), pen)

    def addHorizontalDimension(self, x1, y1, x2, y2, text, pen):
        self.scene.addLine(x1, y1, x2, y2, pen)
        arrow_size = 5
        ext_length = 10
        self.scene.addLine(x1, y1 - ext_length/2, x1, y1 +
            ext_length/2, pen)
        self.scene.addLine(x2, y2 - ext_length/2, x2, y2 +
            ext_length/2, pen)

        points_left = [
            (x1, y1),
            (x1 + arrow_size, y1 - arrow_size/2),
            (x1 + arrow_size, y1 + arrow_size/2)
        ]
        polygon_left = self.scene.addPolygon(QPolygonF([QPointF(x
            , y) for x, y in points_left]), pen)
```

```
        polygon_left.setBrush(QBrush(Qt.black))

        points_right = [
            (x2, y2),
            (x2 - arrow_size, y2 - arrow_size/2),
            (x2 - arrow_size, y2 + arrow_size/2)
        ]
        polygon_right = self.scene.addPolygon(QPolygonF([QPointF(
           x, y) for x, y in points_right]), pen)
        polygon_right.setBrush(QBrush(Qt.black))

        text_item = self.scene.addText(text)
        font = QFont()
        font.setPointSize(5)
        text_item.setFont(font)

        if y1 < 0:
            text_item.setPos((x1 + x2) / 2 - text_item.
               boundingRect().width() / 2, y1 - 25)
        else:
            text_item.setPos((x1 + x2) / 2 - text_item.
               boundingRect().width() / 2, y1 + 5)

    def addVerticalDimension(self, x1, y1, x2, y2, text, pen):
        self.scene.addLine(x1, y1, x2, y2, pen)
        arrow_size = 5
        ext_length = 10
        self.scene.addLine(x1 - ext_length/2, y1, x1 + ext_length
           /2, y1, pen)
        self.scene.addLine(x2 - ext_length/2, y2, x2 + ext_length
           /2, y2, pen)

        if y2 > y1:
            points_top = [
```

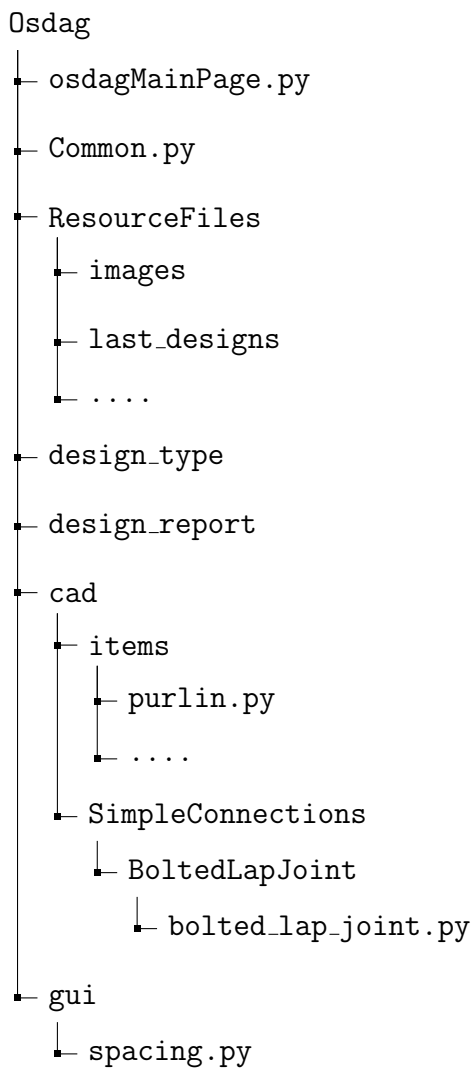
```
                (x1, y1),
                (x1 - arrow_size/2, y1 + arrow_size),
                (x1 + arrow_size/2, y1 + arrow_size)
            ]
            polygon_top = self.scene.addPolygon(QPolygonF([
               QPointF(x, y) for x, y in points_top]), pen)
            polygon_top.setBrush(QBrush(Qt.black))

            points_bottom = [
                (x2, y2),
                (x2 - arrow_size/2, y2 - arrow_size),
                (x2 + arrow_size/2, y2 - arrow_size)
            ]
            polygon_bottom = self.scene.addPolygon(QPolygonF([
               QPointF(x, y) for x, y in points_bottom]), pen)
            polygon_bottom.setBrush(QBrush(Qt.black))
        else:
            points_top = [
                (x2, y2),
                (x2 - arrow_size/2, y2 + arrow_size),
                (x2 + arrow_size/2, y2 + arrow_size)
            ]
            polygon_top = self.scene.addPolygon(QPolygonF([
               QPointF(x, y) for x, y in points_top]), pen)
            polygon_top.setBrush(QBrush(Qt.black))

            points_bottom = [
                (x1, y1),
                (x1 - arrow_size/2, y1 - arrow_size),
                (x1 + arrow_size/2, y1 - arrow_size)
            ]
            polygon_bottom = self.scene.addPolygon(QPolygonF([
               QPointF(x, y) for x, y in points_bottom]), pen)
            polygon_bottom.setBrush(QBrush(Qt.black))
```

```
        text_item = self.scene.addText(text)
        font = QFont()
        font.setPointSize(5)
        text_item.setFont(font)

        if x1 < 0:
            text_item.setPos(x1 - 10 - text_item.boundingRect().
               width(), (y1 + y2) / 2 - text_item.boundingRect().
               height() / 2)
        else:
            text_item.setPos(x1 + 15, (y1 + y2) / 2 - text_item.
               boundingRect().height() / 2)
```

4.4 Documentation

Work is ongoing to improve the GUI window and integrate it with all available CADs.

4.4.1 Directory Structure



Chapter 5

Addition of Tooltips in CADs

5.1 Problem Statement

The **Osdag** application required an interactive feature to enhance user experience within the CAD environment. The primary challenge was to implement a hover-based tooltip system that would meet the following objectives:

- Display visual indicators (e.g., `'+'` signs) at specific points within the 3D CAD window.
- Show contextual information when users hover over these indicators.
- Provide seamless integration with the existing PyQt-based 3D viewer.
- Maintain optimal performance while tracking multiple tooltip locations.
- Ensure tooltips appear dynamically near the user's cursor position.

5.2 Tasks Done

5.2.1 Methodology

The implementation followed a three-tier approach:

- **Viewer Extension:** Extended the existing `qtViewer3d` class to support mouse event handling.

- **Display Function Development:** Created a utility function to manage tooltip placement and registration.
- **Integration:** Incorporated tooltip functionality into the existing 3D model display pipeline.

5.2.2 Processes

Phase 1: Core Infrastructure Development

- Extended the base viewer class to create `HoverableViewer3d`.
- Implemented mouse event detection and object intersection logic.
- Established tooltip storage and retrieval mechanism.

Phase 2: Tooltip Management System

- Developed the `DisplayMsg` function for tooltip creation and positioning.
- Implemented visual indicator generation using `AIS_TextLabel`.
- Created tooltip registration system with coordinate mapping.

Phase 3: Integration and Testing

- Integrated tooltip functionality into the main display pipeline.
- Tested tooltip responsiveness and visual appearance.
- Validated tooltip positioning accuracy.

5.2.3 Key Implementation Features

Enhanced Viewer Class (`HoverableViewer3d`)

- Overridden `mouseMoveEvent` function for real-time mouse tracking.
- Object intersection detection in 3D space.
- Dynamic tooltip display based on cursor position.
- Tooltip storage management through `_hover_tooltips` list.

Tooltip Display System

- Visual '+' indicator creation using `AIS_TextLabel`.
- Customizable tooltip appearance (color, size, position).
- Message association with 3D coordinates.
- Context-sensitive tooltip rendering.

Integration Architecture

- Seamless replacement of default viewer in `init_display` function.
- Centralized tooltip management through utility functions.
- Modular design allowing easy tooltip addition throughout the application.

5.2.4 Implementation Summary Table

Component	File Location	Primary Function	Key Features
`HoverableViewer3d` Class	`ui_template.py`	Mouse event handling	Mouse tracking, object detection, tooltip display
`DisplayMsg` Function	`utilities/_init_.py`	Tooltip creation and registration	Visual indicator creation, message storage, 3D positioning
Integration Layer	`common_logic.py`	Tooltip implementation	Function calls within `display_3DModel`
Tooltip Storage	`ui_template.py`	Data management	`_hover_tooltips` list for tooltip tracking

Table 5.1: Implementation Summary of Hover Tooltip Feature

5.3 Technical Specifications

- **Visual Indicator:** '+' character displayed as `AIS_TextLabel`
- **Tooltip Positioning:** 3D coordinate system using `gp_Pnt(x, y, z)`
- **Color Customization:** RGB color specification (e.g., `(1.0, 1.0, 0.0)` for yellow)
- **Height Configuration:** Adjustable text height parameter
- **Display Context:** Integration with existing `display.Context.Display` system

Python Code

1. Class: `Window(QMainWindow)` — Event Handling and Display Setup in ui_template.py

Listing 5.1: Modifications made to Window class in ui_template.py

```
class Window(QMainWindow):
    closed = QtCore.pyqtSignal()

    def __init__(self):
        super().__init__()
        self.mytabWidget = QTabWidget()
        self.setCentralWidget(self.mytabWidget)

        self.init_display()
        # self.modelTab.installEventFilter(self)
        print("Event filter installed")

    def eventFilter(self, source, event):
        if event.type() == QEvent.MouseMove:
            print("Mouse moved on:", source)
        return super().eventFilter(source, event)

    def check_hover(self, x, y):
        context = self.display.Context
        view = self.display.View

        # Move selection to mouse position
        context.MoveTo(x, y, view)

        owner = context.DetectedOwner()
        if owner:
            obj = owner.Selectable()
            for ais_obj, tooltip in getattr(self.display, "
                _hover_tooltips", []):
                if obj == ais_obj:
```

```
30                      QToolTip.showText(QtGui.QCursor.pos(), tooltip,
                            self.display)
31                      return
32          # Hide tooltip if not hovering
33          QToolTip.hideText()
34
35      def init_display(self, backend_str=None, size=(1024, 768)):
36          from OCC.Display.backend import load_backend, get_qt_modules
37
38          used_backend = load_backend(backend_str)
39
40          if 'qt' in used_backend:
41              QtCore, QtGui, QtWidgets, QtOpenGL = get_qt_modules()
42
43          from OCC.Display.qtDisplay import qtViewer3d
44
45          class HoverableViewer3d(qtViewer3d):
46              def __init__(viewer_self, parent=None):
47                  super(HoverableViewer3d, viewer_self).__init__(parent)
48
49              def mouseMoveEvent(viewer_self, event):
50                  pos = event.pos()
51                  if hasattr(viewer_self, '_display') and viewer_self.
                        _display:
52                      ctx = viewer_self._display.Context
53                      view = viewer_self._display.View
54                      ctx.MoveTo(pos.x(), pos.y(), view, True)
55
56                      owner = ctx.DetectedOwner()
57                      if owner:
58                          obj = owner.Selectable()
59                          for ais_obj, tooltip in getattr(viewer_self.
                                _display, "_hover_tooltips", []):
60                              if obj == ais_obj:
61                                  QtWidgets.QToolTip.showText(QtGui.
                                        QCursor.pos(), tooltip, viewer_self)
62                                  return
63                  QtWidgets.QToolTip.hideText()
64                  super().mouseMoveEvent(event)
```

```
65
66          self.modelTab = HoverableViewer3d(self)
67          self.mytabWidget.addTab(self.modelTab, "")
68          self.modelTab.InitDriver()
69
70          self.display = self.modelTab._display
71          self.display._hover_tooltips = []
72          self.modelTab._display = self.display
73
74          self.display.set_bg_gradient_color([23, 1, 32], [23, 1, 32])
75          self.display.display_triedron()
76          self.display.View.SetProj(1, 1, 1)
```

2. Function: DisplayMsg in utilities/__init__.py

Listing 5.2: DisplayMsg function to add visual indicators

```
1  def DisplayMsg(display, point, text_to_write, height=15, message_color=
       None, update=False):
2      if isinstance(point, gp_Pnt):
3          pnt = point
4      elif isinstance(point, gp_Pnt2d):
5          pnt = gp_Pnt(point.X(), point.Y(), 0.0)
6      else:
7          raise TypeError("point must be gp_Pnt or gp_Pnt2d")
8
9      if hasattr(display, 'Context'):
10         ais_context = display.Context
11     else:
12         raise AttributeError("Display does not have a valid AIS context
               .")
13
14     if message_color is not None:
15         if len(message_color) != 3:
16             raise ValueError("message_color must be a tuple of 3 floats
                   between 0 and 1")
17         r, g, b = message_color
18         if not (0 <= r <= 1 and 0 <= g <= 1 and 0 <= b <= 1):
```

```
            raise ValueError("message_color values must be in range 0 to 1")
        color = Quantity_Color(r, g, b, Quantity_TOC_RGB)
    else:
        color = Quantity_Color(1.0, 1.0, 0.0, Quantity_TOC_RGB)  # Yellow

    plus_label = AIS_TextLabel()
    plus_label.SetText("+")
    plus_label.SetPosition(pnt)
    plus_label.SetColor(color)
    plus_label.SetHeight(height)

    ais_context.Display(plus_label, True)

    if not hasattr(display, "_hover_tooltips"):
        display._hover_tooltips = []
    display._hover_tooltips.append((plus_label, text_to_write))

    if update:
        display.Repaint()

    return plus_label
```

3. Calling DisplayMsg in display_3DModel() in common_logic.py

Listing 5.3: Example of calling DisplayMsg()

```
elif self.connection == KEY_DISP_BCENDPLATE:
    self.Bc = self.module_class
    self.ExtObj = self.createBCEndPlateCAD()

    self.display.View.SetProj(OCC.Core.V3d.V3d_XnegYnegZpos)
    c_length = self.column_length
    Point1 = gp_Pnt(0.0, 0.0, c_length)
    DisplayMsg(self.display, Point1, self.Bc.supporting_section.designation)

```

```
    b_length = self.beam_length + self.Bc.supporting_section.depth/2 +
        100

    Point2 = gp_Pnt(0.0, -b_length, c_length/2)
    DisplayMsg(self.display, Point2, self.Bc.supported_section.
        designation)

    Point3 = gp_Pnt(0.0, -b_length, c_length)
    DisplayMsg(self.display, Point3, f"Bolt Numbers: {self.Bc.
        bolt_numbers}")

    Point4 = gp_Pnt(0.0, -b_length - 100, c_length)
    DisplayMsg(self.display, Point4, f"Bolt Diameter: {self.Bc.
        bolt_diameter_provided}")

    Point5 = gp_Pnt(0.0, -b_length - 200, c_length)
    DisplayMsg(self.display, Point5, f"Bolt Grade: {self.Bc.
        bolt_grade_provided}")

    base_x = 0.0
    base_y = -b_length - 300
    base_z = c_length

    Point6 = gp_Pnt(base_x, base_y, base_z)
    DisplayMsg(self.display, Point6, f"End Plate Height: {self.Bc.
        ep_height_provided}")
```

5.4 Overview of the Three-Script Interaction

The tooltip system operates through a coordinated workflow involving three key scripts that work together to create, register, detect, and display interactive labels in the 3D CAD environment. This section provides a step-by-step breakdown of their interaction.

5.4.1 Phase 1: System Initialization and Setup

Step 1: Enhanced Viewer Creation (`ui_template.py`)

Window Class Initialization Process:

1. **Viewer Replacement**: During application startup, the `Window` class's `init_display` function is called.
2. **HoverableViewer3d Instantiation**: Instead of the standard `qtViewer3d`, a `HoverableViewer3d` instance is created.
3. **Tooltip Registry Initialization**: An empty `_hover_tooltips` list is created to store tooltip data.
4. **Event Handler Setup**: The `mouseMoveEvent` method is overridden for mouse tracking.
5. **Display Context Configuration**: The new viewer is integrated with the display context.

Step 2: Mouse Event Handler Preparation

HoverableViewer3d Class Setup:

1. **Event Override**: `mouseMoveEvent` captures all mouse movements.
2. **Detection Logic Preparation**: Algorithms are ready for 3D intersection testing.
3. **Tooltip Display Mechanism**: Initialized to render tooltips on hover.

5.4.2 Phase 2: Tooltip Creation and Registration

Step 3: Model Display Initiation (`common_logic.py`)

display_3DModel Function Execution:

1. Renders the 3D model.

2. Identifies tooltip locations.

3. Calls `DisplayMsg` with parameters for each tooltip.

Example Function Call:

Listing 5.4: Calling DisplayMsg to show tooltip

```
DisplayMsg(
    display=self.display,
    point=gp_Pnt(x, y, z),
    text_to_write="Bolt dia:5mm",
    height=15,
    message_color=(1.0, 1.0, 0.0),
    update=True
)
```

Step 4: Visual Indicator Creation (`osdag/utilities/__init__.py`)

DisplayMsg Function Workflow:

1. Creates an `AIS_TextLabel` with a '+' character.

2. Positions it using `gp_Pnt(x, y, z)`.

3. Styles the label: color, height, and font.

4. Displays it using `display.Context.Display`.

5. Registers the tooltip in `_hover_tooltips`.

5.4.3 Phase 3: Real-Time Hover Detection and Display

Step 5: Continuous Mouse Monitoring (`ui_template.py`)

mouseMoveEvent Execution:

1. Captures mouse movement.

2. Converts screen coordinates to 3D.

3. Tests intersection with scene objects.

Step 6: Tooltip Detection Process

Detection Algorithm:

Listing 5.5: Detection of tooltip regions on mouse hover

```
for tooltip_object, message in _hover_tooltips:
    if cursor_intersects_with(tooltip_object):
        target_message = message
        show_tooltip = True
        break
```

1. Iterates over `_hover_tooltips`.

2. Computes cursor-to-object proximity.

3. Shows tooltip if hover zone is triggered.

Step 7: Dynamic Tooltip Display

1. Retrieves message.

2. Calculates screen position.

3. Creates popup widget.

4. Displays near cursor.

Step 8: Tooltip Lifecycle Management

1. Tracks cursor in real time.

2. Maintains visibility while hovering.

3. Hides when cursor exits.

4. Frees tooltip resources.

5.4.4 Complete Workflow Summary

Initialization Phase:

- `ui_template.py`: Initializes `HoverableViewer3d`, `_hover_tooltips`

Registration Phase:

- `common_logic.py` → `DisplayMsg`
- `osdag/utilities/__init__.py` registers and displays labels

Runtime Phase:

- `ui_template.py` monitors mouse, checks proximity, shows/hides tooltip

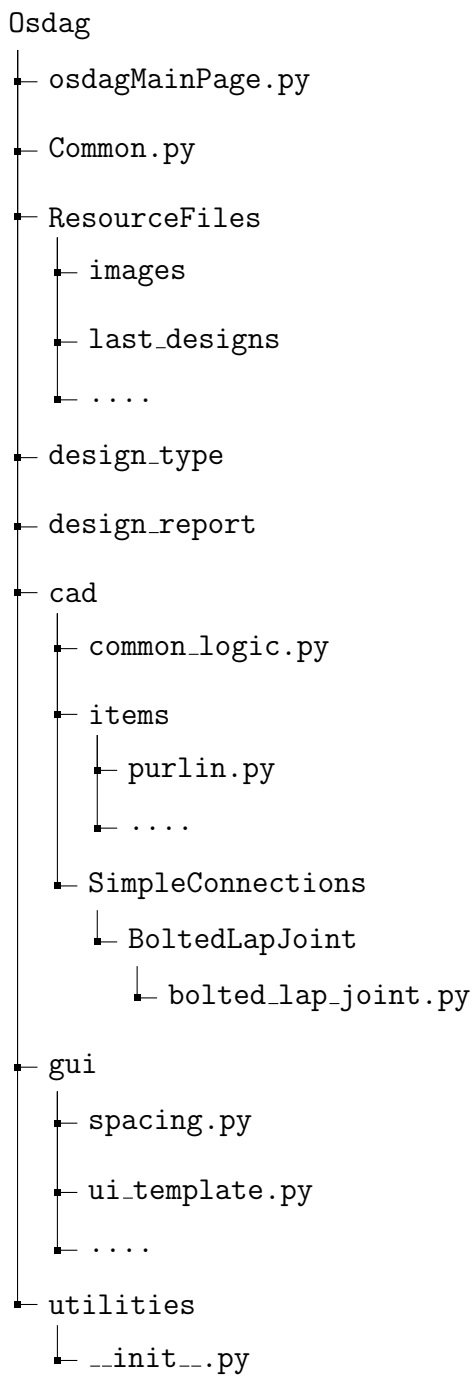
Data Flow Between Scripts:

- `ui_template.py` ↔ `osdag/utilities/__init__.py`: Tooltip registration
- `common_logic.py` → `osdag/utilities/__init__.py`: Tooltip configuration
- `osdag/utilities/__init__.py` → `ui_template.py`: Registration completion
- `ui_template.py` uses registry to show hover tooltips

5.5 Documentation

A separate documentation for other interns to add labels to other CADs in Osdag was created.

5.5.1 Directory Structure



Chapter 6

Osdag Performance Profiling and Optimization

6.1 Problem Statement

The Osdag CAD application was experiencing significant performance bottlenecks, especially during the generation of complex 3D models involving multiple components, such as bolted connections in Column to Column Splice Cover Plate configurations. The main performance concerns observed included:

- **Extended Processing Times:** Model generation became noticeably slower as the number of bolts increased.
- **Laggy User Experience:** The application became unresponsive during CAD rendering and interaction.
- **Inefficient Resource Utilization:** High CPU usage was noted, particularly during geometric shape fusion processes.
- **Profiling Limitations:** Difficulty in pinpointing exact sources of performance degradation without suitable profiling tools.
- **Lack of Optimization:** No clear guidance on which parts of the code required improvement due to missing analysis metrics.

The core challenge was to identify which specific operations consumed the most computational resources and to optimize them accordingly. Special attention was needed for

the geometric fusion operations, which play a central role in CAD model construction and were found to be particularly time-consuming.

6.2 Tasks Done

6.2.1 Methodology

The performance optimization approach followed a systematic profiling and optimization methodology:

- **Profiling Infrastructure Development:** Implemented a real-time profiling system using Python's `cProfile` module.
- **Interactive Profiling Control:** Developed keyboard-controlled profiling triggers for flexible performance monitoring.
- **Performance Analysis:** Analyzed profiling results to identify computational bottlenecks.
- **Targeted Optimization:** Implemented specific optimizations based on profiling findings.
- **Comparative Analysis:** Measured performance improvements before and after optimization.

6.2.2 Processes

Phase 1: Profiling System Implementation

- Integrated `cProfile` module for comprehensive performance monitoring.
- Implemented keyboard-based profiling controls using hotkeys.
- Developed automated profiling output generation and analysis.
- Created both console and file-based profiling report generation.

Phase 2: Performance Analysis

- Conducted systematic profiling of the Osdag application during various CAD operations.
- Identified shape fusion operations as the primary performance bottleneck.
- Analyzed time distribution across different application functions.
- Focused analysis on *Column to Column Splice Cover Plate Bolted* CAD scenarios.

Phase 3: Optimization Implementation

- Replaced multiple sequential `BRepAlgoAPI_Fuse` calls with `BOPAlgo_Builder`.
- Implemented `multi_fusion()` function using `BOPAlgo_Builder` for batch processing.
- Added fallback mechanism to ensure reliability.
- Integrated error handling and performance logging.

Phase 4: Validation and Testing

- Conducted comparative performance testing between old and new fusion methods.
- Validated optimization effectiveness on models with large numbers of bolts.
- Ensured geometric accuracy was maintained with the new implementation.
- Documented performance improvements and optimization impact.

6.2.3 Implementation

Profiling Infrastructure

- Real-time profiling control through keyboard hotkeys (`'p'` to start, `'s'` to stop).
- Threading implementation to prevent blocking of the main application.
- Comprehensive output generation including both statistical analysis and detailed reports.

- File-based profiling output for detailed post-analysis.

Shape Fusion Optimization

- Implementation of `multi_fusion()` function using `BOPAlgo_Builder`.
- Batch processing of multiple shapes in a single operation instead of sequential fusion.
- Error handling with fallback to traditional `BRepAlgoAPI_Fuse` method.
- Performance logging to track optimization effectiveness.

6.2.4 Key Technical Improvements

- Reduced computational complexity from $O(n)$ sequential operations to a single batch operation.
- Minimized intermediate shape creation and memory allocation.
- Improved geometric processing efficiency for multi-component assemblies.
- Enhanced scalability for models with increasing component counts.

6.3 Python Code

6.3.1 Description of Script

The profiler implementation provides comprehensive performance monitoring for the Osdag application through four key components:

1. **Core Profiling Engine:** Utilizes Python's `cProfile` module with a global profiler object to capture detailed execution statistics. This includes:
    - Function call counts
    - Individual function execution times
    - Cumulative time analysis across all calls
2. **Interactive Control System:** Enables flexible real-time control of profiling through keyboard hotkeys:

- Pressing 'p' triggers the `start_profiling()` function.
- Pressing 's' triggers the `stop_profiling()` function.
- Allows developers to begin and end profiling on-demand during execution without modifying the core logic.

3. **Threading Architecture:** Ensures non-blocking operation by:
   - Running the keyboard listener in a separate daemon thread.
   - Ensuring the main Osdag application remains responsive during profiling.
   - Allowing parallel execution of profiling control and CAD model generation.
4. **Output Generation:** Automates result processing upon stopping the profiler:
   - Uses `pstats.Stats` to sort results by total execution time.
   - Displays the top 50 most time-consuming functions in the console.
   - Saves a detailed report in a text file for deeper analysis.
   - Enables developers to focus only on significant performance bottlenecks.

6.3.2 Python Code

The Python script is shown below. Each section is commented for clarity.

Listing 6.1: Profiling code added at the end of osdagMainPage.py

```
import cProfile
import pstats
import threading
import keyboard  # Install with `pip install keyboard`

# Initialize global profiler object for performance monitoring
profiler = cProfile.Profile()

def start_profiling():
    """
    Initiates profiling session
    Enables the cProfile profiler to begin collecting performance data
    """
    print("Profiling started...")
```

```
15      profiler.enable()
16
17  def stop_profiling():
18      """
19      Terminates profiling session and generates comprehensive output
20      Processes collected data and creates both console and file reports
21      """
22      print("Profiling stopped...")
23      profiler.disable()
24
25      # Save raw profiling data to binary file
26      profiler.dump_stats("profile_output")
27
28      # Create stats object for analysis and formatting
29      stats = pstats.Stats("profile_output")
30      stats.sort_stats("time")  # Sort by execution time (most time-
            consuming first)
31
32      # Display top 50 functions in console for immediate feedback
33      stats.print_stats(50)
34
35      # Generate detailed text report for permanent record
36      with open("profile_output.txt", "w") as f:
37          stats.stream = f
38          stats.print_stats(50)  # Save top 50 functions to file
39
40      print("Profile output saved to profile_output.txt")
41
42  def listen_for_keys():
43      """
44      Keyboard listener function for interactive profiling control
45      Runs in separate thread to avoid blocking main application
46      """
47      keyboard.add_hotkey('p', start_profiling)  # Press 'p' to start
            profiling
48      keyboard.add_hotkey('s', stop_profiling)   # Press 's' to stop
            profiling
49      keyboard.wait('esc')  # Keep listening until 'esc' is pressed to
            exit
```

```
50
51 # Start keyboard listener in background daemon thread
52 # Daemon thread ensures it doesn't prevent application shutdown
53 threading.Thread(target=listen_for_keys, daemon=True).start()
54
55 def main():
56     """
57     Main application entry point
58     Placeholder for Osdag application main loop
59     """
60     # Main Osdag application logic would be called here
61     do_stuff()  # Represents main Osdag application execution
62
63 if __name__ == '__main__':
64     main()
```

6.3.3 Explanation of the Code

- **Line 1–4**: Imports necessary modules: `cProfile` and `pstats` for profiling, `threading` to enable parallel execution, and `keyboard` for capturing hotkeys.
- **Line 7**: Initializes a global `cProfile.Profile` object named `profiler` to monitor function calls and performance data.
- **Lines 9–14**: Defines the `start_profiling()` function that enables the profiler and starts collecting performance metrics when called. It prints a confirmation message to the console.
- **Lines 16–34**: Defines the `stop_profiling()` function, which:
  - Disables the profiler (Line 19)
  - Dumps raw data to a file named `profile_output` (Line 22)
  - Creates a `Stats` object from the dumped data and sorts it by execution time (Lines 24–25)
  - Prints the top 50 time-consuming functions to the console (Line 28)
  - Writes the same top 50 functions to a file named `profile_output.txt` (Lines 31–33)

- **Lines 36–41**: Defines the `listen_for_keys()` function that:
  - Registers hotkeys: pressing `'p'` starts profiling and `'s'` stops it (Lines 38–39)
  - Keeps listening for keyboard input until `'esc'` is pressed (Line 40)
- **Line 44–46**: Starts the `listen_for_keys()` function in a background daemon thread. This ensures the listener doesn't block the main application and exits cleanly when the program ends.
- **Lines 48–53**: Defines the `main()` function, a placeholder for Osdag's main execution logic. It currently calls a placeholder function `do_stuff()` that represents the core application.
- **Line 55**: Ensures the `main()` function is only executed if the script is run as the main program.

6.4 Documentation

The code to create the profiler and the results were shared and discussed with the mentors and fellow interns, including a document detailing the advantages of fusing multiple shapes using BOPAlgo_Builder instead of multiple BRepAlgoAPI_Fuse calls

Chapter 7

Conclusions

7.1 Tasks Accomplished

During the internship, five key components were developed and integrated into the Osdag structural design framework, improving its 3D modeling and visualization features.

- **Core 3D Modeling Systems:** Implemented C-Section Purlin modeling using pythonOCC, enabling parametric geometry generation for steel sections with integrated database access for standard properties. Developed Bolted Lap Joint Assembly modeling with accurate positioning of plates, bolts, and nuts, including automated bolt pattern generation based on structural inputs.
- **Interactive Visualization:** Added dynamic bolt pattern visualization with technical drawing standards using PyQt5. Integrated hover-based tooltips into the 3D viewer to display contextual information, enhancing user interaction during design.
- **Performance Optimization:** Used `cProfile` and keyboard-controlled profiling to identify performance bottlenecks. Optimized shape fusion by replacing sequential `BRepAlgoAPI_Fuse` calls with `BOPAlgo_Builder`, improving performance for models with many components.

7.2 Skills Developed

- **Programming and Development:** Gained advanced proficiency in PythonOCC for 3D modeling, PyQt5 for GUI and graphics, and `cProfile` for performance

analysis. Strengthened skills in database integration, threading, and event-driven programming for responsive UI development.

- **CAD and 3D Modeling:** Acquired expertise in parametric modeling, boolean operations, and spatial positioning algorithms. Developed familiarity with engineering drawing standards, dimensional annotations, and interactive 3D visualization techniques.

- **Software Engineering:** Applied systematic profiling and optimization techniques. Built modular architectures, integrated multi-component systems, and implemented fallback mechanisms for reliability. Improved technical writing through documentation and reporting.

- **Problem-Solving and Analysis:** Enhanced ability to identify and resolve performance issues, improve user experience, and translate engineering requirements into functional software solutions.

Chapter A

Appendix

A.1 Work Reports

DATE	DAY	TASK	Hours Worked
Name:	Aryan Gupta		
Project:	Osdag		
Internship:	FOSSEE Semester Long Internship		
15-Feb-2025	Saturday	Set up development environment and installed Osdag source code for testing	2
16-Feb-2025	Sunday	LEAVE	0
17-Feb-2025	Monday	Tested Osdag build and explored 3D modeling architecture	2
18-Feb-2025	Tuesday	Studied PythonOCC structure and its integration into Osdag framework	2
19-Feb-2025	Wednesday	Explored bolt modeling logic in Osdag and began planning tooltip system	4
20-Feb-2025	Thursday	Started implementation of hover-based tooltip functionality in 3D viewer	3
21-Feb-2025	Friday	Integrated tooltip display into custom PyQt viewer; debugged viewer response	5
22-Feb-2025	Saturday	Continued testing and refinement of tooltip logic for various component types	3
23-Feb-2025	Sunday	LEAVE	0
24-Feb-2025	Monday	Mentor meeting: reviewed tooltip progress and discussed viewer enhancements	5
25-Feb-2025	Tuesday	Worked on interactive bolt pattern annotation and dimensional display	2
26-Feb-2025	Wednesday	Improved UI logic for bolt visualization using PyQt5 graphics components	5
27-Feb-2025	Thursday	Linked bolt pattern drawing to backend parameter extraction logic	5
28-Feb-2025	Friday	Mentor feedback and debugging of dynamic drawing inconsistencies	5
01-Mar-2025	Saturday	LEAVE	0
02-Mar-2025	Sunday	LEAVE	0
03-Mar-2025	Monday	LEAVE	0
04-Mar-2025	Tuesday	LEAVE	0
05-Mar-2025	Wednesday	LEAVE	0
06-Mar-2025	Thursday	LEAVE	0
07-Mar-2025	Friday	LEAVE	0
08-Mar-2025	Saturday	LEAVE	0
09-Mar-2025	Sunday	LEAVE	0
10-Mar-2025	Monday	Refactored dimensional annotation system for consistent scaling and positioning	4
11-Mar-2025	Tuesday	Implemented automatic bolt pattern layout logic based on structural rules	5
12-Mar-2025	Wednesday	Continued bolt layout generation, tested on different structural configurations	5
13-Mar-2025	Thursday	Meeting to demonstrate bolt pattern system and collect feedback	2
14-Mar-2025	Friday	Implemented parametric modeling logic for C-section Purlin using PythonOCC	5
15-Mar-2025	Saturday	Tested C-section shape generation and improved geometry definition	2
16-Mar-2025	Sunday	LEAVE	0
17-Mar-2025	Monday	Connected C-section parameters to database values for standard sections	2
18-Mar-2025	Tuesday	Reviewed and debugged auto-retrieval of section properties	2
19-Mar-2025	Wednesday	Mentor discussion on shape modeling consistency and 3D alignment issues	4
20-Mar-2025	Thursday	Implemented boolean union operations for joint assemblies	4
21-Mar-2025	Friday	Faced performance issues in shape fusion; began profiling system setup	2
22-Mar-2025	Saturday	Set up cProfile-based profiling system with keyboard-controlled trigger	4
23-Mar-2025	Sunday	LEAVE	0
24-Mar-2025	Monday	Analyzed profiling output to locate CAD performance bottlenecks	3
25-Mar-2025	Tuesday	Replaced shape fusion algorithm with BOPAlgo_Builder for optimization	3
26-Mar-2025	Wednesday	Benchmarked performance before and after optimization	2
27-Mar-2025	Thursday	Mentor review meeting for performance enhancements and modeling logic	5
28-Mar-2025	Friday	Tested bolt layout generation in complex joint models	5
29-Mar-2025	Saturday	Improved spatial positioning logic for multi-component assemblies	3
30-Mar-2025	Sunday	LEAVE	0
31-Mar-2025	Monday	Prepared internal documentation on tooltip and bolt pattern systems	3
01-Apr-2025	Tuesday	Reviewed and finalized 3D modeling integration for joints and purlins	5
02-Apr-2025	Wednesday	Mentor feedback session on modeling coverage and CAD precision	3
03-Apr-2025	Thursday	Worked on user interface responsiveness using threading/event system	3
04-Apr-2025	Friday	Added fallback mechanisms for missing data scenarios in UI pipeline	5
05-Apr-2025	Saturday	Refactored module communication between CAD and GUI components	2
06-Apr-2025	Sunday	LEAVE	0
07-Apr-2025	Monday	Documented overall implementation and shared codebase walkthrough	2
08-Apr-2025	Tuesday	Drafted system report detailing performance optimization steps	5
09-Apr-2025	Wednesday	Collected user feedback on current visualization tools	4
10-Apr-2025	Thursday	Analyzed feedback and mapped improvements to implementation areas	3
11-Apr-2025	Friday	Prepared updated dimensional annotation tools based on feedback	4
12-Apr-2025	Saturday	Final mentor sync-up for feedback review and project wrap-up	5
13-Apr-2025	Sunday	LEAVE	0
14-Apr-2025	Monday	Worked on internship final report and documentation formatting	2
15-Apr-2025	Tuesday	Set up development environment and installed Osdag source code for testing	3
16-Apr-2025	Wednesday	Tested Osdag build and explored 3D modeling architecture	4
17-Apr-2025	Thursday	Studied PythonOCC structure and its integration into Osdag framework	5
18-Apr-2025	Friday	Explored bolt modeling logic in Osdag and began planning tooltip system	4
19-Apr-2025	Saturday	Started implementation of hover-based tooltip functionality in 3D viewer	4
20-Apr-2025	Sunday	LEAVE	0
21-Apr-2025	Monday	Integrated tooltip display into custom PyQt viewer; debugged viewer response	2
22-Apr-2025	Tuesday	Continued testing and refinement of tooltip logic for various component types	2
23-Apr-2025	Wednesday	Mentor meeting: reviewed tooltip progress and discussed viewer enhancements	4
24-Apr-2025	Thursday	Worked on interactive bolt pattern annotation and dimensional display	5
25-Apr-2025	Friday	Improved UI logic for bolt visualization using PyQt5 graphics components	2
26-Apr-2025	Saturday	Linked bolt pattern drawing to backend parameter extraction logic	4
27-Apr-2025	Sunday	LEAVE	0
28-Apr-2025	Monday	Mentor feedback and debugging of dynamic drawing inconsistencies	5
29-Apr-2025	Tuesday	Refactored dimensional annotation system for consistent scaling and positioning	5
30-Apr-2025	Wednesday	Implemented automatic bolt pattern layout logic based on structural rules	5
01-May-2025	Thursday	Continued bolt layout generation, tested on different structural configurations	4
02-May-2025	Friday	LEAVE	0
03-May-2025	Saturday	LEAVE	0

04-May-2025	Sunday	LEAVE	0
05-May-2025	Monday	LEAVE	0
06-May-2025	Tuesday	LEAVE	0
07-May-2025	Wednesday	LEAVE	0
08-May-2025	Thursday	LEAVE	0
09-May-2025	Friday	LEAVE	0
10-May-2025	Saturday	LEAVE	0
11-May-2025	Sunday	LEAVE	0
12-May-2025	Monday	LEAVE	0
13-May-2025	Tuesday	LEAVE	0
14-May-2025	Wednesday	LEAVE	0
15-May-2025	Thursday	LEAVE	0
16-May-2025	Friday	Meeting to demonstrate bolt pattern system and collect feedback	3
17-May-2025	Saturday	Implemented parametric modeling logic for C-section Purlin using PythonOCC	4
18-May-2025	Sunday	LEAVE	0
19-May-2025	Monday	Tested C-section shape generation and improved geometry definition	3
20-May-2025	Tuesday	Connected C-section parameters to database values for standard sections	3
21-May-2025	Wednesday	Reviewed and debugged auto-retrieval of section properties	4
22-May-2025	Thursday	Mentor discussion on shape modeling consistency and 3D alignment issues	5
23-May-2025	Friday	Implemented boolean union operations for joint assemblies	2
24-May-2025	Saturday	Faced performance issues in shape fusion; began profiling system setup	5
25-May-2025	Sunday	LEAVE	0
26-May-2025	Monday	Set up cProfile-based profiling system with keyboard-controlled trigger	4
27-May-2025	Tuesday	Analyzed profiling output to locate CAD performance bottlenecks	5
28-May-2025	Wednesday	Replaced shape fusion algorithm with BOPAlgo_Builder for optimization	3
29-May-2025	Thursday	Benchmarked performance before and after optimization	5
30-May-2025	Friday	Mentor review meeting for performance enhancements and modeling logic	5
31-May-2025	Saturday	Tested bolt layout generation in complex joint models	3
01-Jun-2025	Sunday	LEAVE	0
02-Jun-2025	Monday	LEAVE	0
03-Jun-2025	Tuesday	LEAVE	0
04-Jun-2025	Wednesday	LEAVE	0
05-Jun-2025	Thursday	LEAVE	0
06-Jun-2025	Friday	LEAVE	0
07-Jun-2025	Saturday	LEAVE	0
08-Jun-2025	Sunday	LEAVE	0
09-Jun-2025	Monday	LEAVE	0
10-Jun-2025	Tuesday	LEAVE	0
11-Jun-2025	Wednesday	LEAVE	0
12-Jun-2025	Thursday	LEAVE	0
13-Jun-2025	Friday	LEAVE	0
14-Jun-2025	Saturday	LEAVE	0
15-Jun-2025	Sunday	LEAVE	0
16-Jun-2025	Monday	LEAVE	0
17-Jun-2025	Tuesday	LEAVE	0
18-Jun-2025	Wednesday	LEAVE	0
19-Jun-2025	Thursday	LEAVE	0
20-Jun-2025	Friday	LEAVE	0
21-Jun-2025	Saturday	LEAVE	0
22-Jun-2025	Sunday	LEAVE	0
23-Jun-2025	Monday	LEAVE	0
24-Jun-2025	Tuesday	LEAVE	0
25-Jun-2025	Wednesday	LEAVE	0
26-Jun-2025	Thursday	LEAVE	0
27-Jun-2025	Friday	LEAVE	0
28-Jun-2025	Saturday	Improved spatial positioning logic for multi-component assemblies	3
29-Jun-2025	Sunday	LEAVE	0
30-Jun-2025	Monday	Prepared internal documentation on tooltip and bolt pattern systems	2

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.