FOSSEE Semester Internship Report

On

Osdag on Web

Submitted by

Sabeena Viklar

1st Year B.Tech Student, Department of Electronics and Communication

Usha mittal institute of technology,

Mumbai

Under the Guidance of

Prof. Siddhartha Ghosh

Department of Civil Engineering

Indian Institute of Technology Bombay

Mentors:

Parth Karia

June 4, 2026

Acknowledgments

- Start with a general statement of thanks. Express your overall gratitude to everyone who supported you during your project or research.
- Project staff at the Osdag team, Ajmal Babu M. S., Ajinkya Dahale, and Parth Karia,
- Osdag Principal Investigator (PI) Prof. Siddhartha Ghosh, Department of Civil Engineering at IIT Bombay
- FOSSEE PI Prof. Kannan M. Moudgalya, FOSSEE Project Investigator, Department of Chemical Engineering, IIT Bombay
- FOSSEE Managers Usha Viswanathan and Vineeta Parmar and their entire team
- Acknowledge the support from the National Mission on Education through Information and Communication Technology (ICT), Ministry of Education (MoE), Government of India, for their role in facilitating this project
- Acknowledge your colleagues who worked with you during your internship or project.
- If appropriate, thank your college, department, head, and principal for their support during your studies.

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.

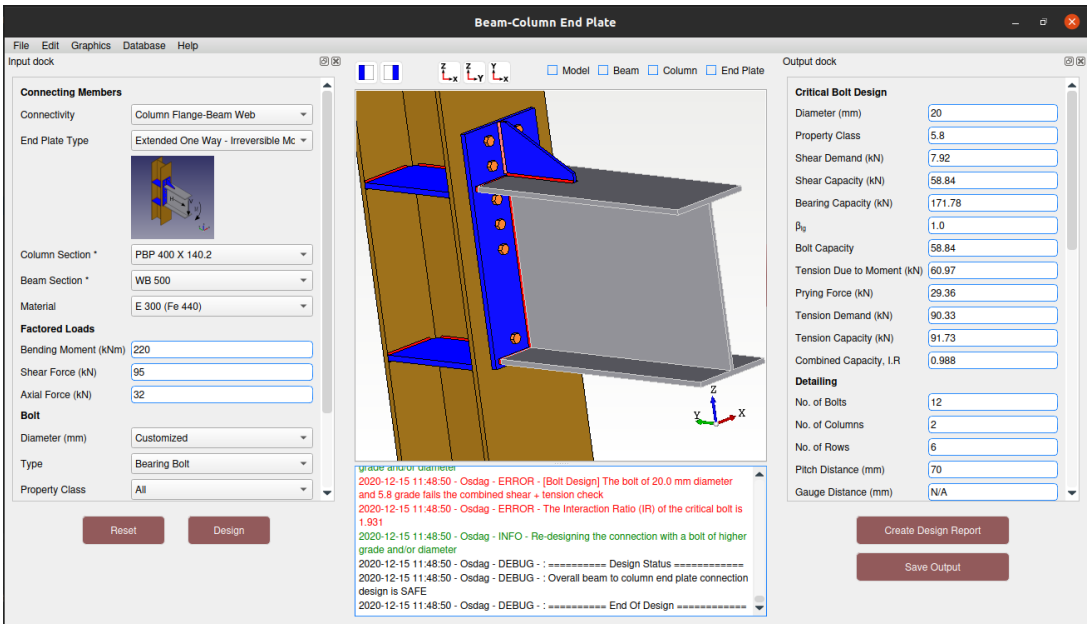

Figure 1.2: Osdag GUI

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

Development of Location-Based Wind and Seismic Zone Wizard for Osdag-Bridge

2.1 Problem Statement

The manual design of bridge structures requires engineers to constantly cross-reference multiple Indian Standard codes to determine location-specific environmental parameters, such as wind speeds, seismic zone factors, and temperature variations. Furthermore, determining the initial geometric layout of the bridge deck (e.g., span, carriageway width, girder spacing) involves managing complex, interdependent constraints.

The lack of a centralized, digital "Group Design" wizard leads to workflow inefficiencies, manual calculation errors, and a high risk of violating critical design constraints (such as IRC 24 compliance for skew angles). There is a critical need for a full-stack web application module that can automate environmental parameter retrieval, enforce real-time engineering constraints, and dynamically calculate interdependent geometric values through an intuitive user interface.

2.1.1 Tasks Accomplished

To solve this problem, a decoupled full-stack web application was engineered using a React frontend (with Material-UI) and a Django REST Framework backend. The following key

technical tasks were successfully completed:

- **Dual-Mode Location Wizard:** Developed a dynamic project location module capable of operating in two modes. Mode 1 utilizes Django REST API endpoints (`/api/states/`, `/api/districts/`) to query a custom SQLite database and automatically populate wind speed and seismic data. Mode 2 provides a spreadsheet-style React modal for manual, custom parameter entry.

- **Real-Time Engineering Validation:** Implemented rigorous validation logic at both the frontend UI level and the backend Django Serializer level. This includes strict range constraints for Span (5m – 100m) and Carriageway Width (3m – 50m), alongside automated compliance warnings for Skew Angles exceeding 20° as per IRC 24 (2010) standards.

- **Full-Stack API Architecture:** Architected a clean, RESTful backend featuring 5+ distinct endpoints. The system handles complex JSON data structures, utilizes Axios for asynchronous HTTP requests, and provides immediate visual feedback (such as green color-coding for validated parameters) through robust React state management (`useState`, `useEffect`).

2.1.2 Backend Django Architecture

The backend is built with Python and the Django REST Framework (DRF), following a strictly decoupled API architecture. It enforces business logic and engineering validations at the serializer level before any data reaches the database.

Geometry Calculation Engine (`serializers.py`)

The most complex constraint management occurs within the `GeometryCalculationSerializer`. It dynamically recalculates missing geometric parameters based on the specific field the user modified on the frontend.

Code Implementation:

```
class GeometryCalculationSerializer(serializers.Serializer):
    """Serializer for geometry calculation"""
    carriageway_width = serializers.FloatField(required=True)
```

```
    girder_spacing = serializers.FloatField(required=False, allow_null=True)
    num_girders = serializers.IntegerField(required=False, allow_null=True)
    deck_overhang = serializers.FloatField(required=False, allow_null=True)
    changed_field = serializers.CharField(required=True)

    def validate(self, data):
        carriageway_width = data['carriageway_width']
        overall_width = carriageway_width + 5
        changed_field = data['changed_field']

        # ... variable extraction ...

        # Calculate based on which field changed
        if changed_field == 'girder_spacing' and girder_spacing is not None:
            if num_girders is not None:
                deck_overhang = overall_width - (girder_spacing * num_girders)
            elif deck_overhang is not None:
                num_girders = int((overall_width - deck_overhang) / girder_spacing)

        # Validate constraints
        errors = {}
        if girder_spacing and girder_spacing >= overall_width:
            errors['girder_spacing'] = 'Must be less than overall bridge width'
        if deck_overhang and deck_overhang >= overall_width:
            errors['deck_overhang'] = 'Must be less than overall bridge width'

        if errors:
            raise serializers.ValidationError(errors)

        return data
```

Component Structure and Explanation: The serializer processes the `POST` payload and executes the following logic:

- **Dynamic Deduction:** By identifying the `changed_field`, the serializer determines which of the interdependent variables (Spacing, Number of Girders, or Deck Overhang) needs to be deduced using the structural equation.
- **Constraint Validation:** It inherently validates the results against the physical limits of the bridge (e.g., ensuring girder spacing never exceeds the overall bridge width). If limits are exceeded, a structured `ValidationError` is thrown and parsed by the React frontend.

Location Data API (`views.py`)

The `DistrictViewSet` manages the retrieval of location-specific environmental parameters required for the Mode 1 Location Wizard.

Code Implementation:

```
class DistrictViewSet(viewsets.ReadOnlyModelViewSet):
    """ViewSet for listing districts"""
    queryset = District.objects.all()
    serializer_class = DistrictSerializer
    permission_classes = [AllowAny]

    def get_queryset(self):
        queryset = District.objects.all()
        state_id = self.request.query_params.get('state_id', None)
        if state_id is not None:
            queryset = queryset.filter(state_id=state_id)
        return queryset

    @action(detail=True, methods=['get'])
    def location_data(self, request, pk=None):
        """Get complete location data for a district"""
        district = self.get_object()
        data = {
            'state': district.state.name,
```

```
            'district': district.name,
            'basic_wind_speed': district.basic_wind_speed,
            'seismic_zone': district.seismic_zone,
            'zone_factor': district.zone_factor,
            'max_shade_air_temp': district.max_shade_air_temp,
            'min_shade_air_temp': district.min_shade_air_temp,
        }
        return Response(data)
```

Component Structure and Explanation:

- **Dynamic Filtering (`get_queryset`):** Overrides the default queryset to allow the frontend to filter districts dynamically based on the `state_id` parameter, populating the secondary dropdown.
- **Custom Action Endpoint (`@action`):** Generates a highly specific sub-route (`/api/districts/{id}/location_data/`) that serializes only the relevant environmental parameters (Wind Speed, Seismic Zone, Zone Factor, Temperatures) for immediate injection into the React UI's "green-highlighted" validation fields.

2.1.3 Frontend React Architecture

The frontend is built with React 18, following a component-based architecture with centralized state management to handle the complex, interdependent data flows required by the bridge module.

Interactive Geometry Component (`GeometryPopup.js`)

The `GeometryPopup` component is responsible for collecting the user's geometric inputs and managing the asynchronous communication with the Django backend to auto-calculate the missing constraint values.

Code Implementation:

```
const calculateGeometry = async (changedField) => {
  try {
    const response = await axios.post(`${API_BASE_URL}/calculate-geometry/`, {
```

```
      carriageway_width: carriageway,
      girder_spacing: girderSpacing !== '' ? parseFloat(girderSpacing) : null,
      num_girders: numGirders !== '' ? parseInt(numGirders) : null,
      deck_overhang: deckOverhang !== '' ? parseFloat(deckOverhang) : null,
      changed_field: changedField
    });

    if (response.data.success) {
      const data = response.data.data;
      setGirderSpacing(data.girder_spacing || '');
      setNumGirders(data.num_girders || '');
      setDeckOverhang(data.deck_overhang || '');
      setErrors({}); // Clear errors on success
    }
  } catch (error) {
    if (error.response && error.response.data && error.response.data.errors) {
      setErrors(error.response.data.errors);
    }
  }
};

// Example of the Input Field mapping
<input
  type="number"
  value={girderSpacing}
  onChange={(e) => setGirderSpacing(e.target.value)}
  onBlur={() => girderSpacing !== '' &&
              (numGirders !== '' || deckOverhang !== '') &&
              calculateGeometry('girder_spacing')}
  className={`form-control ${errors.girder_spacing ? 'error' : ''}`}
/>
```

Component Structure and Explanation: The React component is engineered to

provide a seamless, non-blocking user experience:

- **Asynchronous API Integration (`calculateGeometry`):** Utilizes Axios to send a `POST` request to the Django geometry engine. It carefully parses the local React state into nullable floats/integers to ensure the backend serializer receives clean data.

- **On-Blur Validation UX:** Instead of firing the heavy calculation logic on every single keystroke (which would cause UI stuttering and rapid, unnecessary API calls), the `calculateGeometry` function is strictly bound to the `onBlur` event. It only triggers when the user finishes typing and clicks away from the input field, provided at least two fields are populated.

- **Dynamic Error Handling:** The `catch` block intercepts HTTP 400 Bad Request responses from the Django serializer. It dynamically extracts the specific field errors (e.g., *"Must be less than overall bridge width"*) and updates the `errors` state object, which conditionally applies a red `.error` CSS class to the specific offending input field.

Chapter 3

Real-Time Particle Swarm Optimization (PSO) Visualization System

3.1 3.1 Introduction and Problem Definition

The Osdag Web interface needed some updates to match the functionality and overall user experience of the Desktop version. During team discussions, we identified that while the standalone desktop application provided great visual feedback during the lengthy optimization calculations, the web version was lacking in this area.

To bridge this gap and improve usability, we decided to implement a real-time visualization dashboard specifically for the Plate Girder module. The primary goal of this task was to build a dynamic 3D graph that behaves and looks like the one found in the desktop version of Osdag. By doing this, we ensured that engineers using the web platform have the exact same ability to monitor the Particle Swarm Optimization (PSO) process in real-time, making the web tool much more engaging and transparent.

3.2 Asynchronous Background Processing

Integrated Celery workers backed by a Redis message broker. This ensures the backend Django API remains highly responsive, immediately returning a taskid while the optimization engine evaluates thousands of potential cross-sections asynchronously.

3.2 Proposed System Architecture

To achieve real-time, non-blocking visualization of a computationally heavy algorithm, a decoupled, event-driven architecture was designed. The architecture consists of three distinct layers:

1. **Computation Layer (Celery and Redis):** Offloads the intensive CPU-bound PSO calculations from the main web server to dedicated background worker processes.

2. **Transport Layer (Django Channels and WebSockets):** Establishes a persistent, full-duplex communication channel between the background workers and the client's browser, enabling continuous high-frequency data streaming.

3. **Presentation Layer (React.js and Plotly.js):** Provides a high-performance, GPU-accelerated 3D rendering environment capable of plotting thousands of dynamic data points without degrading the user experience.

3.2.1 Frontend 3D Visualization and User Interface

The script is structured as follows:

- **Input Parameters**: The user specifies the beam and column sections, applied shear load, and bolt properties.

- **Design Calculations**: The program computes the number of bolts required, their arrangement, and verifies the adequacy of the end plate and bolt group.

- **Output**: Results include the number of bolts, their layout, and adequacy checks for the connection components.

3.2.2 Python Code

Listing 3.1: Celery Task Integration (tasks.py)

```
%--------------------begin code-------------

import os
```

```python
from celery import shared_task
from channels.layers import get_channel_layer
from asgiref.sync import async_to_sync
import time

@shared_task(bind=True, max_retries=3)
def run_pso_optimization(self, channel_name: str, input_data: dict):
    channel_layer = get_channel_layer()
    task_id = self.request.id if self and hasattr(self, 'request') else
        "SYNC_FALLBACK"

    # Callback passed to the core optimization engine
    def viz_callback(depth, ur, weight_kg, iteration, particle_idx,
        position, variable_list, lb, ub):
        # Format and serialize particle data
        payload = {
            "iteration": int(iteration),
            "particle_index": int(particle_idx),
            "ur": float(ur),
            "weight_kg": float(weight_kg),
            "depth": float(depth),
            "variables": [float(v) for v in position],
            "variable_names": list(variable_list)
        }
        # Broadcast real-time update to the frontend WebSocket
        async_to_sync(channel_layer.send)(
            channel_name,
            {"type": "pso_update", "data": payload}
        )

    # Initialize and run the Osdag module with the callback attached
    module = create_module()
    result = module.optimized_method(
        input_data,
        viz_callback=viz_callback
    )

    # Send completion signal and final results
    async_to_sync(channel_layer.send)(
```

```
41          channel_name,
42          {"type": "pso_complete", "data": {"result": result}}
43      )
44
45  %-------------------- end code ---------------
```

3.2.3 Explanation of the Code

- **Line 1-4**: Imports necessary libraries, including Celery for background task processing and Django Channels components for asynchronous WebSocket communication.
- **Line 6-9**: Defines the asynchronous `run_pso_optimization` background task, initializes the WebSocket channel layer, and retrieves the unique task ID to manage the execution lifecycle.
- **Line 11-27**: Defines the `viz_callback` function, which acts as a hook inside the optimization engine. It captures raw particle data, formats it into a JSON-serializable payload, and broadcasts the real-time update to the frontend WebSocket channel.
- **Line 29-34**: Initializes the core Osdag Plate Girder module and executes the heavy `optimized_method` calculation, passing in the user inputs and attaching the real-time visualization callback.
- **Line 36-40**: Sends a final completion signal (`pso_complete`) via the WebSocket channel once the optimization concludes, transmitting the final selected design results back to the user interface.

3.2.4 Full code

```
import React, { useState, useRef } from 'react';
import Plot from 'react-plotly.js';

function OptimizationGraph({ data }) {
    // Separate feasible (safe) and infeasible (unsafe) particles
    const feasible = { x: [], y: [], z: [] };
```

```
    const infeasible = { x: [], y: [], z: [] };

    data.history.forEach(particle => {
        if (particle.ur <= 1.0) {
            feasible.x.push(particle.ur);
            feasible.y.push(particle.depth);
            feasible.z.push(particle.weight_kg);
        } else {
            infeasible.x.push(particle.ur);
            infeasible.y.push(particle.depth);
            infeasible.z.push(particle.weight_kg);
        }
    });

    return (
        <Plot
            data={[
                // 1. Feasibility Boundary Plane (UR = 1.0)
                {
                    type: 'mesh3d',
                    x: [1, 1, 1, 1],
                    y: [0, 3000, 3000, 0],
                    z: [0, 0, 8000, 8000],
                    opacity: 0.1,
                    color: '#F87171',
                },
                // 2. Safe Designs (Green)
                {
                    x: feasible.x, y: feasible.y, z: feasible.z,
                    type: 'scatter3d', mode: 'markers',
                    marker: { color: '#6b7d20', size: 4 }
                },
                // 3. Unsafe Designs (Red)
                {
```

```
                    x: infeasible.x, y: infeasible.y, z:
                        infeasible.z,
                    type: 'scatter3d', mode: 'markers',
                    marker: { color: '#ef4444', size: 4 }
                },
                // 4. Global Best Design (Gold Star)
                {
                    x: [data.globalBest.ur], y: [data.globalBest.
                        depth], z: [data.globalBest.weight_kg],
                    type: 'scatter3d', mode: 'markers',
                    marker: { symbol: 'star', color: '#FFD700',
                        size: 14 }
                }
            ]}
            layout={{
                scene: {
                    xaxis: { title: 'Utilization Ratio' },
                    yaxis: { title: 'Depth (mm)' },
                    zaxis: { title: 'Weight (kg)' }
                }
            }}
        />
    );
}
```

3.3 Task 1: Documentation

3.3.1 Testing and Validation

To ensure the robustness of the visualization engine without constantly running the computationally expensive backend solver, a standalone "Demo Mode" was integrated into the frontend.

- **Simulated Data Generation:** A synthetic data generator was built directly into

the React component to mimic the behavior of the backend PSO algorithm, generating 300 randomized particles across 30 iterations.

- **UI Stress Testing:** This allowed for rapid UI iteration and stress testing of the Plotly.js rendering engine, ensuring it could handle high volumes of data without crashing.
- **Cross-Validation:** The final web visualization outputs were cross-referenced with the legacy desktop application to ensure mathematical and visual parity.

3.3.2 User Experience (UX) Enhancements

Beyond simply plotting the data, several UX enhancements were implemented to make the dashboard a good engineering tool:

- **Live Metrics Dashboard:** A footer summary bar was developed to display real-time physical dimensions (Depth, Flange Width, Web Thickness) of the current "Global Best" design.

Chapter 4

Global UI/UX Refinements

4.1 Problem Statement

The Osdag Web interface required specific updates to match the functionality and overall user experience of the standalone Desktop version. During team discussions, several key changes were identified across the global application to bridge this gap:

1. **Inefficient Screen Real Estate:** The global navigation bar occupied excessive vertical space, unnecessarily constraining the viewport available for the core 3D engineering canvas.

2. **Component Layout Issues:** The Output Dock and graphics control buttons suffered from styling inconsistencies, wrapping text on smaller screens, and outdated labels that did not match the premium aesthetic of the application.

3. **Asynchronous Rendering Delays:** The heavy 3D WebGL models lacked proper loading states, causing the canvas to appear frozen while assets were being fetched and parsed by the browser.

4.2 Tasks Accomplished

To implement the changes identified during team discussions and elevate the professional feel of the web application, the following UI/UX tasks were successfully completed:

- **Navbar and Layout Optimization :** Reduced the global navbar height to maximize the engineering workspace and resolved alignment and state-management bugs

within the 3D graphics control buttons.

- **Output Dock Overhaul:** Replaced outdated labels and completely overhauled the `BaseOutputDock` component's styling to ensure responsive, non-wrapping text and a better visual hierarchy.
- **3D Canvas Upgrades:** Integrated React `<Suspense>` boundaries for graceful loading states, added raycasting hover events for structural members, and implemented a programmatic screenshot capture utility for the WebGL canvas.

4.3 Code Implementation and Documentation

4.3.1 Output Dock Layout Refactor (`BaseOutputDock.jsx`)

To ensure the Output Dock components rendered correctly on smaller screens without text wrapping or layout breaks, targeted Tailwind CSS utility classes were updated.

Code Implementation:

```
<button
    onClick={handleCreateDesignReport}
    className="flex flex-1 items-center justify-center gap-x-2
               bg-osdag-green text-white text-sm font-semibold
               px-3 py-2.5 rounded-lg shadow-md duration-200
               hover:bg-osdag-dark-green whitespace-nowrap"
    type="button"
>
    <svg height="18px" viewBox="0 -960 960 960" width="18px"
         fill="#FFFFFF" className="flex-shrink-0">
        {/* SVG Path */}
    </svg>
    Generate Report
</button>
```

Documentation & Key Points:

- **Typography & Spacing:** The button text was reduced using `text-sm`, and

padding was tightened from `px-4 py-3` to `px-3 py-2.5`. The SVG icon was correspondingly scaled down from 24px to 18px.

- **Preventing Layout Shifts:** The `whitespace-nowrap` class was added to the button text, and `flex-shrink-0` was added to the SVG icon. This guarantees the label "Generate Report" will never break onto two lines, maintaining a clean aesthetic regardless of the dock's width.

4.4 3D Canvas Loading State Management (`EngineeringModule`

The React Three Fiber rendering pipeline was updated to handle asynchronous CAD file loading gracefully.

Code Implementation:

```
<Suspense
    fallback={
        <Html>
            <p>Loading 3D Model...</p>
        </Html>
    }
>
    {/* Asynchronous 3D component rendering */}
</Suspense>
```

Documentation & Key Points:

- **React Suspense:** A `<Suspense>` boundary was introduced to "catch" the asynchronous rendering of heavy 3D assets.
- **HTML Overlay (`@react-three/drei`):** By utilizing the `<Html>` helper, a standard DOM paragraph element ("Loading 3D Model...") is projected directly over the WebGL canvas, providing immediate user feedback and preventing the appearance of a frozen application.

4.4.1 Advanced 3D Model Interactivity (`EngineeringModule.jsx`)

The core `<Model>` component was expanded with new props to support structural interactivity, dynamic view resets, and screenshot capabilities.

Code Implementation:

```
<Model
    modelPaths={normalizedCadModelPaths}
    selectedView={Array.isArray(selectedSection) ?
       selectedSection[0] : selectedSection}
    selectedViews={selectedSection}
    isMobile={isMobile}
    cameraSettings={{
        ...cameraSettings,
        connectivity: getConnectivity(),
    }}
    hoverDict={hoverDict}
    onHoverLabel={handleHoverLabel}
    onHoverEnd={handleHoverEnd}
    moduleCadConfig={moduleConfig}
    key={`${modelKey}-${selectedSection}`}
/>
<ScreenshotCapture
    screenshotTrigger={screenshotTrigger}
    setScreenshotTrigger={setScreenshotTrigger}
    selectedView={Array.isArray(selectedSection) ?
       selectedSection[0] : selectedSection}
/>
```

Documentation & Key Points:

- **Contextual Camera Control:** The `cameraSettings` object now dynamically computes `getConnectivity()`. This ensures the camera automatically frames and focuses on the most critical structural joints when a new view is selected.
- **Component Remounting:** A composite React key (`key=${modelKey}-${selectedSection}``) was added. This is a critical React pattern that forces the `<Model>` to unmount and

remount entirely on view changes, ensuring the GPU memory and WebGL context are cleanly reset.

4.5 Fixing the Global UI/UX Refinements and Graphics Controls

1. **Broken Graphics Controls:** The 3D canvas control buttons (Zoom, Pan, Rotate, and Orthographic views) were unresponsive or failed to register consecutive clicks, severely hindering the user's ability to inspect 3D structural models.
2. **Component Layout Issues:** The Output Dock suffered from styling inconsistencies, wrapping text on smaller screens, and outdated labels that did not match the premium aesthetic of the application.
3. **Inefficient Screen Real Estate:** The global navigation bar occupied excessive vertical space, unnecessarily constraining the viewport available for the core engineering workspace.

4.5.1 Tasks Accomplished

To implement the changes identified during team discussions and elevate the professional feel of the web application, the following targeted UI/UX tasks were successfully completed:

- **Graphics Controls State Management (PR #32):** Completely refactored the event handling for the 3D canvas controls. Implemented custom DOM events for camera actions and built a robust React `useEffect` hook to handle orthographic view switching and state cleanup.
- **Output Dock Overhaul (PR #30 & #33):** Replaced outdated labels and completely overhauled the `BaseOutputDock` component's Tailwind styling to ensure responsive, non-wrapping text.
- **Navbar Optimization (PR #32):** Reduced the global navbar height to maximize the available engineering workspace.

4.5.2 Code Implementation: Graphics Controls Fixes

The most significant technical challenge was resolving the broken 3D graphics buttons. This required architectural changes across multiple React components.

Custom Event Dispatching (`UnifiedDropdownMenu.jsx`)

To allow UI buttons located outside the 3D canvas to control the WebGL camera, a custom DOM event dispatcher was implemented.

Code Implementation:

```
case "Zoom In":
    document.dispatchEvent(new CustomEvent('cad-camera-action
        ', { detail: 'zoom-in' }));
    break;
case "Pan":
    document.dispatchEvent(new CustomEvent('cad-camera-action
        ', { detail: 'pan-left' }));
    break;
case "Rotate 3D Model":
    document.dispatchEvent(new CustomEvent('cad-camera-action
        ', { detail: 'auto-rotate' }));
    break;
```

Documentation & Key Points: Instead of passing complex React refs down a deeply nested component tree, custom DOM events (`cad-camera-action`) were utilized. This safely decouples the UI layout from the 3D canvas, allowing the WebGL engine to independently listen for and react to camera manipulation commands (zoom, pan, rotate).

Graphics State Routing and Cleanup (`EngineeringModule.jsx`)

A comprehensive `useEffect` hook was developed to intercept graphics options from the user input state and route them to the appropriate 3D component states.

Code Implementation:

```
useEffect(() => {
    if (inputs?.graphicsOption) {
        const opt = inputs.graphicsOption;

        // Handle orthographic view switching
        if (opt === "Show front view") {
            setSelectedCameraView("XY");
        } else if (opt === "Change Background") {
            if (colorPickerRef.current) colorPickerRef.
               current.click();
        }
        // ... (additional view routing logic)

        // Critical Cleanup: Allow consecutive clicks of the
           same option
        setInputs(prev => {
            const next = { ...prev };
            delete next.graphicsOption;
            return next;
        });
    }
}, [inputs?.graphicsOption, selectedSection]);
```

Documentation & Key Points:

- **State Routing:** The hook listens for changes to `inputs.graphicsOption` and dynamically updates the specific camera view state (e.g., mapping "Show front view" to the "XY" coordinate plane).
- **Consecutive Click Bug Fix:** Previously, clicking the same UI button twice failed because the React state did not mutate. The addition of the cleanup logic (`delete next.graphicsOption`) immediately strips the input state after processing. This forces React to recognize subsequent clicks of the exact same button, resolving a major user experience bug.

Documentation & Key Points: The `whitespace-nowrap` class was added to the button text, and `flex-shrink-0` was added to the downsized 18px SVG icon. This guarantees the label will never break onto two lines, maintaining a clean aesthetic regardless of the browser window's width.

Chapter 5

Conclusions

5.1 Project Summary and Conclusion

5.1.1 Overall Summary

The overarching objective of this project was to elevate the Osdag Web interface, bringing its functionality, performance, and user experience up to parity with the established standalone Desktop application. This was achieved through a full-stack engineering effort divided into two primary domains: First, eliminating the "black-box" nature of the Plate Girder module by engineering a real-time, WebSocket-driven 3D visualization of the Particle Swarm Optimization (PSO) algorithm. Second, resolving critical frontend usability bugs by optimizing the global application layout, stabilizing the 3D graphics controls, and refining the Output Dock interface.

5.1.2 Summary of Tasks Accomplished

Over the course of the project, the following major technical tasks were successfully completed:

- **Real-Time Optimization Dashboard:** Designed and implemented a full-stack data pipeline (using Celery, Redis, and Django Channels) to stream live PSO calculation data from the Python backend to the React frontend.
- **3D Data Visualization:** Built a high-performance Plotly.js React component to render the PSO swarm in 3D space, featuring dynamic color-coding for structural

feasibility and reactive cross-section SVG diagrams.

- **Graphics Controls Fixes:** Refactored the broken 3D canvas UI controls by implementing a decoupled Custom DOM Event system and robust React `useEffect` cleanup hooks, ensuring flawless zooming, panning, and orthographic view switching.

- **Global UI/UX Refinement:** Modernized the application layout by reducing the Navbar height to maximize the 3D engineering workspace and rewriting the Output Dock component with responsive, non-wrapping Tailwind CSS utility classes.

5.1.3 Technical Skills Developed

This project provided extensive hands-on experience across a modern, full-stack web development environment. Key technical and professional skills developed include:

- **Backend Architecture:** Proficiency in asynchronous task queues (Celery/Redis) and full-duplex WebSocket communication (Django Channels) within a Python ecosystem.

- **Advanced Frontend State Management:** Deepened understanding of React.js hooks (`useState`, `useEffect`, `useRef`) and React's reconciliation process (using the `key` prop to force component remounting).

- **3D Graphics & WebGL:** Gained practical experience manipulating 3D data visualizations using Plotly.js and managing 3D canvas states within the React Three Fiber ecosystem.

- **Modern CSS Frameworks:** Applied advanced Tailwind CSS layout strategies (`flex-shrink`, `whitespace-nowrap`) to ensure bulletproof UI responsiveness.

5.1.4 Impact on the Osdag Ecosystem

The enhancements implemented during this project represent a significant leap forward for Osdag Web. By visualizing the PSO algorithm, structural engineers now receive immediate, intuitive feedback during lengthy calculations, fostering trust in the software's mathematical outputs. Furthermore, the UI/UX overhauls have removed frustrating

layout bugs and unresponsive controls. The result is a highly polished, professional-grade web application that provides a seamless, uninterrupted workflow mirroring the premium experience of the native desktop software.

5.1.5 Conclusion

Transitioning a computationally heavy, desktop-first engineering tool to the web requires more than just porting code; it requires careful consideration of asynchronous data flow, browser rendering limits, and user experience. Through the successful implementation of real-time WebSockets, 3D Plotly visualizations, and strict React state-management techniques, this project successfully bridged the gap between Osdag's Web and Desktop platforms. The updated web application now stands as a interactive structural engineering tool ready for deployment.

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.