МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА ВЕБЗАСТОСУНКУ ПОСТПРОЦЕСОРА ІЗ ВИКОРИСТАННЯМ БІБЛІОТЕКИ REACT»**

Виконала: студентка 2 курсу, групи 8.1214-іпз-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
А.Р. Омельяненко
(ініціали та прізвище)

Керівник декан математичного факультету, професор, д.т.н. Гоменюк С.І.
(вчене звання, науковий ступінь, посада, прізвище та ініціали)

Рецензент в.о. завідувача кафедри комп'ютерних наук, доцент, к.пед.н. Пшенична О.С.
(вчене звання, науковий ступінь, посада, прізвище та ініціали)

Запоріжжя – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТЦІ

Омельяненко Альоні Русланівні
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка вебзастосунку постпроцесора із використанням бібліотеки React

керівник роботи Гоменюк Сергій Іванович, д.т.н., професор
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 28 » лютого 2025 року № 326-с

2. Строк подання студентом роботи 05.12.2025 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Технічне завдання.
2. Основні теоретичні відомості.
3. Програмна реалізація вебзастосунку та тестування.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 07.03.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	14.03.2025	
2.	Збір вихідних даних.	18.04.2025	
3.	Обробка методичних та теоретичних	28.05.2025	
	джерел.		
4.	Розробка першого та другого розділу.	05.09.2025	
5.	Розробка третього розділу.	21.11.2025	
6.	Оформлення та нормоконтроль	01.12.2025	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	19.12.2025	

Студент _____ А.Р. Омельяненко
(підпис) (ініціали та прізвище)

Керівник роботи _____ С.І. Гоменюк
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка вебзастосунку постпроцесора із використанням бібліотеки React»: 44 с., 7 рис., 5 табл., 14 джерел, 2 додатка.

3D-ВІЗУАЛІЗАЦІЯ, МЕТОД СКІНЧЕННИХ ЕЛЕМЕНТІВ, ПОСТПРОЦЕСОР, ШЕЙДЕРНИЙ РЕНДЕРИНГ, FRONT-END, GLSL, REACT, WEBGL.

Об’єкт дослідження – процес інтерактивної візуалізації багатовимірних масивів результатів методу скінченних елементів у вебсередовищі.

Мета роботи: розробка вебзастосунку постпроцесора на базі React та WebGL для ефективної обробки та інтерактивної візуалізації складних 3D-сіток та розподілу функціональних значень методу скінченних елементів у веббраузері.

Метод дослідження – методи системного аналізу та моделювання, методи об’єктно-орієнтованого проєктування, методи комп’ютерної графіки, зокрема шейдерне програмування, для реалізації графічного конвеєра.

У роботі проаналізовано методи рендерингу тривимірних інженерних моделей у вебсередовищі. Спроєктовано компонентну архітектуру додатка, що забезпечує взаємодію інтерфейсу користувача з графічним ядром. Реалізовано алгоритми візуалізації полів результатів та механізм фрагментного відсікання на базі GLSL-шейдерів.

В результаті створено швидкодіючий вебпостпроцесор, який дозволяє виконувати детальний аналіз результатів розрахунку без необхідності встановлення десктопного ПЗ.

SUMMARY

Master’s qualifying paper “Development of a Post-processor Web Application Using the React Library”: 44 pages, 7 figures, 5 tables, 14 references, 2 supplements.

3D VISUALIZATION, FINITE ELEMENT METHOD, FRONT-END, GLSL, POST-PROCESSOR, REACT, SHADER RENDERING, WEBGL.

The object of the study is the process of interactive visualization of multidimensional arrays of the finite element method results in a web environment.

The aim of the study is the development of a post-processor web application based on React and WebGL for efficient processing and interactive visualization of complex 3D meshes and the distribution of finite element method functional values in a web browser.

The methods of research are methods of system analysis and modeling, object-oriented design methods, and computer graphics methods, specifically shader programming, for implementing the graphics pipeline.

The thesis analyzes methods for rendering three-dimensional engineering models in a web environment. The component architecture of the application, which ensures the interaction of the user interface with the graphics core, is designed. Algorithms for visualizing result fields and a fragment clipping mechanism based on GLSL shaders are implemented.

As a result, a high-performance web post-processor has been created, which allows performing a detailed analysis of calculation results without the need to install desktop software.

ЗМІСТ

Завдання на кваліфікаційну роботу 2
Реферат 4
Summary 5
Вступ 8
1 Дослідження предметної області та аналіз технологій 11
1.1 Аналіз проблеми візуалізації даних методу скінченних елементів .. 11
1.2 Огляд існуючих систем постпроцесорної обробки 12
1.2.1 Десктопні постпроцесори 12
1.2.2 Веборієнтовані рішення 13
1.3 Особливості використання технології WebGL для 3D-графіки 14
1.4 Висновки до першого розділу 15
2 Архітектура та алгоритми візуалізації 17
2.1 Архітектура вебзастосунку: взаємодія React та WebGL 17
2.2 Проєктування структур даних та парсинг вхідних файлів МСЕ 18
2.2.1 Структура вхідних даних 18
2.2.2 Підготовка буферів WebGL 19
2.3 Розробка алгоритмів перетворення інженерних даних для GPU 20
2.3.1 Алгоритми тріангуляції та розрахунку нормалей 21
2.3.2 Метод колірного відображення функціональних значень 21
2.4 Розробка та інтеграція функції вдосконаленої візуалізації 22
2.4.1 Математична модель відсікання 22
2.4.2 Реалізація алгоритму у фрагментному шейдері 23
2.5 Висновки до другого розділу 23
3 Програмна реалізація та тестування застосунку 25
3.1 Структура та компоненти за стосунку 25
3.2 Програмна реалізація графічного ядра 28
3.2.1 Ініціалізація контексту та компіляція шейдерів 28

3.2.2 Обробка буферів даних та цикл рендерингу 29
3.3 Реалізація користувацького інтерфейсу 29
3.3.1 Головний компонент керування станом 30
3.3.2 Компоненти керування трансформаціями та візуалізацією 31
3.4 Тестування продуктивності та верифікація результатів 32
3.5 Розгортання вебзастосунку 35
Висновки 36
Перелік посилань 38
Додаток А Лістинг програмного коду шейдерів (GLSL) 40
А.1 Вершинний шейдер (Vertex Shader) 40
А.2 Фрагментний шейдер (Fragment Shader) 41
Додаток Б Фрагменти коду реалізації логіки клієнтської частин (JavaScript).42
Б.1 Алгоритм розрахунку габаритів моделі (Bounding Box) 42
Б.2 Метод формування площини зрізу у класі RenderMesh 43

ВСТУП

Сучасне інженерне проєктування значною мірою залежить від чисельного моделювання, зокрема методу скінченних елементів (МСЕ) [1, 2]. Результати таких розрахунків – це не просто набір чисел, а складні багатовимірні масиви даних, що описують напружено-деформований стан, температурні поля та інші фізичні процеси у 3D-геометрії.

Для прийняття обґрунтованих інженерних рішень критично важливим етапом є ефективна візуалізація та аналіз цих даних. Традиційно це завдання вирішується за допомогою десктопного програмного забезпечення (постпроцесорів). Однак, такі рішення часто є ресурсоємними, вимагають дороговартісних ліцензій, потужних робочих станцій та прив’язані до конкретного робочого місця. Це створює суттєві бар’єри в умовах сучасної тенденції до віддаленої роботи, хмарних обчислень та необхідності оперативної колаборації інженерних команд.

Враховуючи зазначені фактори, виникає об’єктивна потреба у трансформації інструментів інженерного аналізу в бік вебтехнологій. Актуальність теми магістерської роботи зумовлена необхідністю створення доступного, кросплатформенного та високопродуктивного інструменту для аналізу результатів МСЕ, який не поступається за функціоналом десктопним аналогам, але забезпечує мобільність та простоту доступу.

Поєднання бібліотеки React для побудови реактивних користувацьких інтерфейсів та технології WebGL для апаратного прискорення тривимірної графіки відкриває нові можливості для вирішення цього завдання. Такий підхід дозволяє перенести складні обчислення візуалізації безпосередньо у браузер клієнта, забезпечуючи високу швидкість рендерингу великих інженерних моделей [4, 14].

Об’єкт дослідження: процес візуалізації багатовимірних даних, отриманих методом скінченних елементів.

Предмет дослідження: методи та засоби розробки високопродуктивних вебзастосунків для інтерактивної 3D-візуалізації інженерних даних.

Мета дослідження: розробка ефективного та інтерактивного вебзастосунку постпроцесора на базі бібліотеки React з використанням технології WebGL для візуалізації та аналізу результатів розрахунків, отриманих методом скінченних елементів.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

– проаналізувати та обґрунтувати вибір технологічного стеку (React, WebGL, JavaScript) для створення високопродуктивного односторінкового застосунку (SPA);
– розробити архітектуру клієнтського застосунку, що забезпечує надійну взаємодію між декларативним інтерфейсом React та імперативним графічним ядром WebGL;
– створити інтерактивний користувацький інтерфейс для керування сценою та параметрами візуалізації (вибір функції, колірна шкала, відображення сітки/поверхні);
– провести тестування розробленого застосунку та оцінити його продуктивність при роботі з типовими інженерними моделями.

Наукова новизна одержаних результатів:

– удосконалено метод візуалізації полів результатів МСЕ у вебсередовищі шляхом розробки комбінованого алгоритму, що поєднує реактивне керування станом сцени (React) з низькорівневим шейдерним рендерингом (WebGL), що дозволило забезпечити високу інтерактивність для моделей великої розмірності;
– набув подальшого розвитку метод динамічного відсікання (clipping) для 3D-сіток, який реалізовано на рівні фрагментного шейдера з підтримкою довільної орієнтації площини перерізу (вздовж осей X, Y, Z), що дозволяє виконувати детальний аналіз внутрішньої структури об’єкта в реальному часі.

Практичне значення одержаних результатів: розроблений вебзастосунок є кросплатформенним інструментом, що не потребує встановлення та ліцензування, і може бути використаний інженерами-конструкторами для швидкого аналізу результатів розрахунків, а також у навчальному процесі для демонстрації розподілу фізичних полів.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА АНАЛІЗ ТЕХНОЛОГІЙ

1.1 Аналіз проблеми візуалізації даних методу скінченних елементів

Метод скінченних елементів (МСЕ) є стандартом у сучасному інженерному проєктуванні [6], забезпечуючи високоточні результати розрахунків напружено-деформованого стану, теплових полів та гідродинаміки. Однак, ефективна робота з результатами МСЕ пов’язана з низкою викликів, розглянемо їх детальніше.

Обсяг даних. Типові 3D-моделі складаються з десятків, а іноді й сотень тисяч скінченних елементів та вузлів. Кожен вузол містить декілька функціональних значень (наприклад, три компоненти переміщення U_x, U_y, U_z та шість компонентів напруги σ_{ij}). Це генерує багатовимірні масиви даних, що потребують високої обчислювальної потужності для обробки.

Складність візуалізації. Для якісного аналізу інженерам необхідно інтерактивно відображати три основні типи інформації:

– геометрія та сітка (Mesh);
– розподіл функції (Color Mapping) по поверхні та всередині об’єкта;
– керування сценою (обертання, масштабування), яке має бути плавним (мінімум 30 FPS) навіть при великих навантаженнях.

Доступність інструментів. Традиційні постпроцесори (наприклад, вбудовані в ANSYS або Abaqus) є дорогими та вимагають встановлення на потужний локальний комп’ютер, що обмежує спільну роботу та віддалений доступ.

1.2 Огляд існуючих систем постпроцесорної обробки

1.2.1 Десктопні постпроцесори

Традиційні системи (зокрема, модулі постпроцесингу у комерційних FEA-пакетах) пропонують найповніший функціонал та найвищу продуктивність за рахунок прямого доступу до ресурсів GPU. Серед найбільш поширених комерційних рішень можна виділити програмні комплекси ANSYS Mechanical, Abaqus CAE та Siemens Simcenter. Ці системи надають користувачеві вичерпний набір інструментів для візуалізації: від побудови ізополів (contour plots) до створення анімації деформаційних процесів у часі.

Проте, їх архітектура, як правило, базується на «товстому клієнті» (thick client), що накладає жорсткі вимоги до апаратного забезпечення робочої станції. Для комфортної роботи з моделями, що містять понад 1 мільйон скінченних елементів, необхідно використовувати професійні відеокарти (наприклад, серії NVIDIA Quadro) та значний обсяг оперативної пам’яті (від 32 ГБ). Крім того, ліцензійна політика таких вендорів часто прив’язує використання ПЗ до конкретної машини або локальної мережі підприємства через апаратні ключі захисту, що унеможливлює мобільну роботу інженера.

Альтернативою комерційним пакетам є рішення з відкритим вихідним кодом (Open Source), серед яких стандартом де-факто став пакет ParaView, побудований на базі бібліотеки VTK (Visualization Toolkit), та постпроцесор Gmsh. Ці інструменти дозволяють обробляти надвеликі масиви даних та підтримують скриптову автоматизацію (переважно мовою Python). Однак, суттєвим недоліком таких систем є високий поріг входження: інтерфейс користувача часто є неінтуїтивним, а налаштування специфічних сценаріїв візуалізації вимагає глибоких знань архітектури бібліотек візуалізації. Крім того, вони так само залежні від локальних апаратних ресурсів, що не вирішує проблему мобільності.

1.2.2 Веборієнтовані рішення

Розвиток технологій WebGL (Web Graphics Library) та WebAssembly створив передумови для перенесення інженерних розрахунків та візуалізації у середовище веббраузера. Цей підхід базується на архітектурі «тонкого клієнта» (thin client), де браузер виступає універсальною платформою для відображення результатів, незалежно від операційної системи користувача (Windows, Linux, macOS) чи типу пристрою.

Існують два основні підходи:

а) серверний рендеринг (streaming): графіка рендериться на потужному віддаленому сервері, а до клієнта передається лише потік зображень – це усуває проблему обчислювальної потужності клієнта, але вимагає високої пропускної здатності мережі та створює затримку при інтерактивному керуванні;

б) клієнтський рендеринг: всі дані (координати, результати) завантажуються в браузер, а рендеринг здійснюється локально на GPU клієнта – це забезпечує нульову затримку при обертанні та масштабуванні, але вимагає використання технологій, здатних ефективно працювати з GPU.

Для визначення оптимального підходу до реалізації розроблюваної системи доцільно співставити характеристики традиційних локальних програм та сучасних веборієнтованих рішень. Результати порівняльного аналізу архітектурних підходів за критеріями продуктивності, вартості та доступності наведено в таблиці 1.1.

Таблиця 1.1 – Порівняльний аналіз підходів до візуалізації

Характеристика	**Десктопні системи (ANSYS, Abaqus)**	**Серверний рендеринг (Cloud/Pixel Streaming)**	**Клієнтський рендеринг (WebGL/WebGPU)**
Місце виконання обчислень	Локальний CPU/GPU	Віддалений сервер	Локальний GPU (у браузері)

Продовження таблиці 1.1

Вимоги до мережі	Не потребує мережі	Високі (передача відеопотоку)	Середні (завантаження моделі один раз)
Інтерактивність (Latency)	Миттєва	Висока затримка (Input lag)	Миттєва (після завантаження)
Вартість розгортання	Висока (ліцензії на кожне місце)	Висока (оренда GPU-серверів)	Низька (будь-який ПК з браузером)
Доступність	Прив’язка до робочої станції	Доступ з будь-якого пристрою	Доступ з будь-якого пристрою

Висновок: для створення доступного та інтерактивного постпроцесора, критично важливим є використання клієнтського рендерингу з апаратним прискоренням.

1.3 Особливості використання технології WebGL для 3D-графіки

Вибір бібліотеки React зумовлений її компонентною природою та ефективним механізмом узгодження (reconciliation) [12]. Оскільки WebGL працює з єдиним HTML-елементом <canvas>, безпосереднє маніпулювання DOM-деревом для рендерингу інтерфейсу поверх 3D-сцени було б неефективним. React дозволяє створити шар абстракції, де стан UI (наприклад, положення слайдера зрізу) синхронізується зі станом WebGL-контексту через життєві цикли компонентів (componentDidUpdate, useEffect). Це забезпечує розділення логіки відображення даних та логіки рендерингу 3D-графіки.

Для реалізації високопродуктивного 3D-постпроцесора у вебсередовищі було обрано технологію WebGL. Вибір цього інструменту зумовлений його архітектурними особливостями, що забезпечують необхідну обчислювальну потужність та гнучкість відображення.

Прямий доступ до GPU. WebGL є JavaScript API, що дозволяє веббраузерам взаємодіяти з апаратним графічним прискорювачем (GPU)

через стандарти OpenGL ES 2.0/3.0 [10]. Це гарантує максимальну продуктивність для рендерингу великих 3D-сіток.

Графічний конвеєр. WebGL використовує концепцію шейдерів (програм, що виконуються безпосередньо на GPU) [14]:

– вершинний шейдер (vertex shader): відповідає за трансформацію координат моделі, освітлення та передачу даних (тут відбувається перетворення інженерних координат на координати екрана);
– фрагментний шейдер (fragment shader): визначає фінальний колір кожного пікселя (тут реалізується колірне відображення та застосовується логіка відсікання площини).

Буферна система. Дані МСЕ (координати, нормалі, кольори) завантажуються на GPU у вигляді буферів атрибутів. Це забезпечує швидку передачу великих масивів даних.

1.4 Висновки до першого розділу

Проведений у першому розділі аналіз предметної області показав, що існуючі десктопні системи постпроцесингу результатів МСЕ є потужними, але мають суттєві обмеження щодо мобільності, вартості та можливостей колективної роботи. Водночас існуючі вебрішення часто страждають від затримок передачі даних або обмеженого функціоналу.

Враховуючи особливості технології WebGL та архітектурні можливості бібліотеки React, було сформульовано задачу розробки спеціалізованого вебзастосунку. Основні вимоги до розроблюваної системи:

а) архітектура: SPA (Single Page Application) з клієнтським рендерингом для забезпечення миттєвого відгуку інтерфейсу;
б) продуктивність: здатність візуалізувати моделі обсягом до 100 000 елементів зі стабільним FPS (не менше 30 кадрів/с);
в) інтерфейс: забезпечення зручного керування параметрами сцени без перезавантаження сторінки;

г) функціональність: підтримка базових операцій постпроцесингу (відображення деформованої схеми, колірне картування полів результатів, виконання динамічних зрізів).

Виходячи з цього, подальша робота буде присвячена проєктуванню архітектури застосунку, розробці алгоритмів обробки даних на GPU та реалізації програмного комплексу.

2 АРХІТЕКТУРА ТА АЛГОРИТМИ ВІЗУАЛІЗАЦІЇ

2.1 Архітектура вебзастосунку: взаємодія React та WebGL

Для забезпечення як високої продуктивності 3D-графіки, так і ефективної розробки користувацького інтерфейсу, у роботі спроєктовано та реалізовано дворівневу архітектуру клієнтського рендерингу. Ця архітектура базується на чіткому розділенні відповідальності між декларативним рівнем інтерфейсу та імперативним графічним ядром. Деталізуємо структуру архітектури за рівнями функціонування.

Декларативний рівень (UI/UX): реалізований на базі бібліотеки React. Цей рівень відповідає за взаємодію з користувачем, керування станом застосунку (state management) та відображення елементів керування (панелей, меню, легенди). Стан застосунку зберігає всі параметри візуалізації: кути обертання, коефіцієнт масштабування, індекс обраної функції, параметри площини перерізу.

Імперативний графічний рівень (Graphics Core): реалізований на базі WebGL (клас RenderMesh). Цей рівень відповідає за безпосередню взаємодію з графічним процесором (GPU), керування буферами даних, компіляцію шейдерів та виконання циклу рендерингу.

Шар взаємодії (Bridge): забезпечує однонаправлений потік даних від React до WebGL. При зміні стану інтерфейсу (наприклад, переміщення повзунка), React викликає відповідний метод публічного API класу RenderMesh, передаючи нові параметри.

Такий підхід дозволяє використовувати реактивність React для побудови зручного інтерфейсу, зберігаючи при цьому максимальну продуктивність WebGL [3, 7], оскільки графічний цикл працює незалежно від циклу оновлення DOM-дерева. Структуру класів розробленого застосунку та зв’язки між ними наведено на рисунку 2.1.

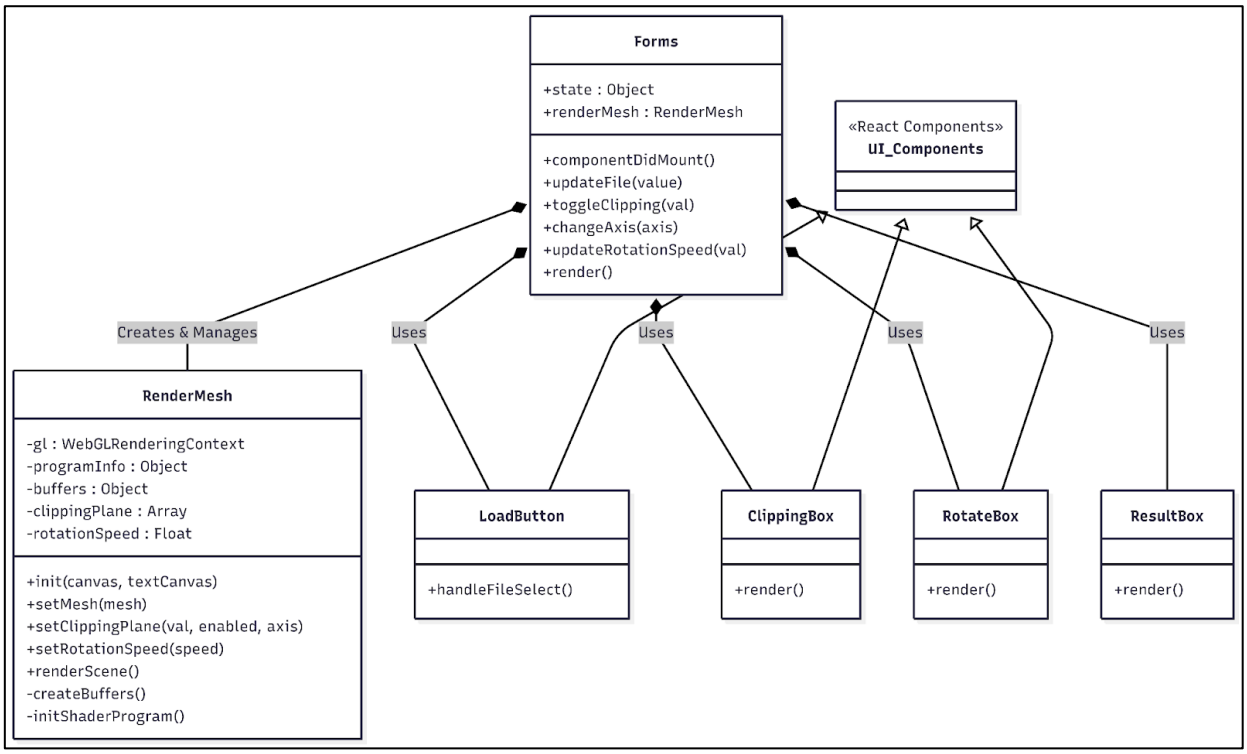


Рисунок 2.1 – Діаграма класів архітектури вебзастосунку

2.2 Проєктування структур даних та парсинг вхідних файлів МСЕ

Ефективність постпроцесора значною мірою залежить від оптимізованого зберігання даних. Вхідні файли результатів розрахунку містять неструктуровані бінарні або текстові дані, які необхідно адаптувати для графічного конвеєра.

2.2.1 Структура вхідних даних

Модель даних МСЕ, що використовується в роботі, базується на трьох основних масивах:

а) масив координат вузлів (x): одновимірний масив, що містить координати всіх вузлів сітки у форматі $[x_1, y_1, z_1, x_2, y_2, z_2, \dots, x_N, y_N, z_N]$, де N – кількість вузлів сітки;

б) масив топології елементів (fe): масив цілих чисел, що визначає зв'язність вузлів, а кожна група індексів відповідає одному скінченному елементу;

в) масив функціональних значень ($func$): містить результати розрахунків (наприклад, компоненти вектора переміщень U або тензора напружень σ) для кожного вузла.

2.2.2 Підготовка буферів WebGL

Для ефективної передачі геометричних даних на графічний процесор було спроєктовано структуру буферів вершин (Vertex Buffer Objects – VBO). Оскільки WebGL оперує виключно трикутниками, на цьому етапі спершу виконується перетворення топології скінченних елементів у набір незалежних вершин, придатних для рендерингу [11].

Для кожної вершини графічної моделі формується набір атрибутів, необхідних для роботи шейдерного конвеєра:

– a_position (позиція) – просторові координати вузла;
– a_normal (нормаль) – вектор для розрахунку освітлення;
– a_color (колір) – значення для візуалізації розподілу фізичного поля.

Важливим аспектом програмної реалізації є оптимізація передачі цих даних. Вхідні файли MCE часто містять надлишкову інформацію, тому на етапі парсингу реалізовано конвертацію даних у типізовані масиви JavaScript (Typed Arrays) [9], зокрема Float32Array для координат та Uint16Array для індексів. На відміну від стандартних масивів Array, типізовані масиви представляють собою бінарні буфери фіксованої довжини, структура яких ідентична масивам у мовах C/C++.

Такий підхід дозволяє уникнути накладних витрат на серіалізацію даних при їх передачі у графічний контекст. Браузер отримує можливість копіювати вміст масиву безпосередньо у відеопам'ять GPU через шину PCI

Express, що забезпечує максимальну пропускну здатність. Для моделей великої розмірності (сотні тисяч елементів) це дозволило зменшити час ініціалізації сцени з кількох секунд до мілісекунд.

Загальну динаміку взаємодії компонентів системи під час процесу завантаження, парсингу та підготовки даних зображено на діаграмі послідовності (див. рис. 2.2).

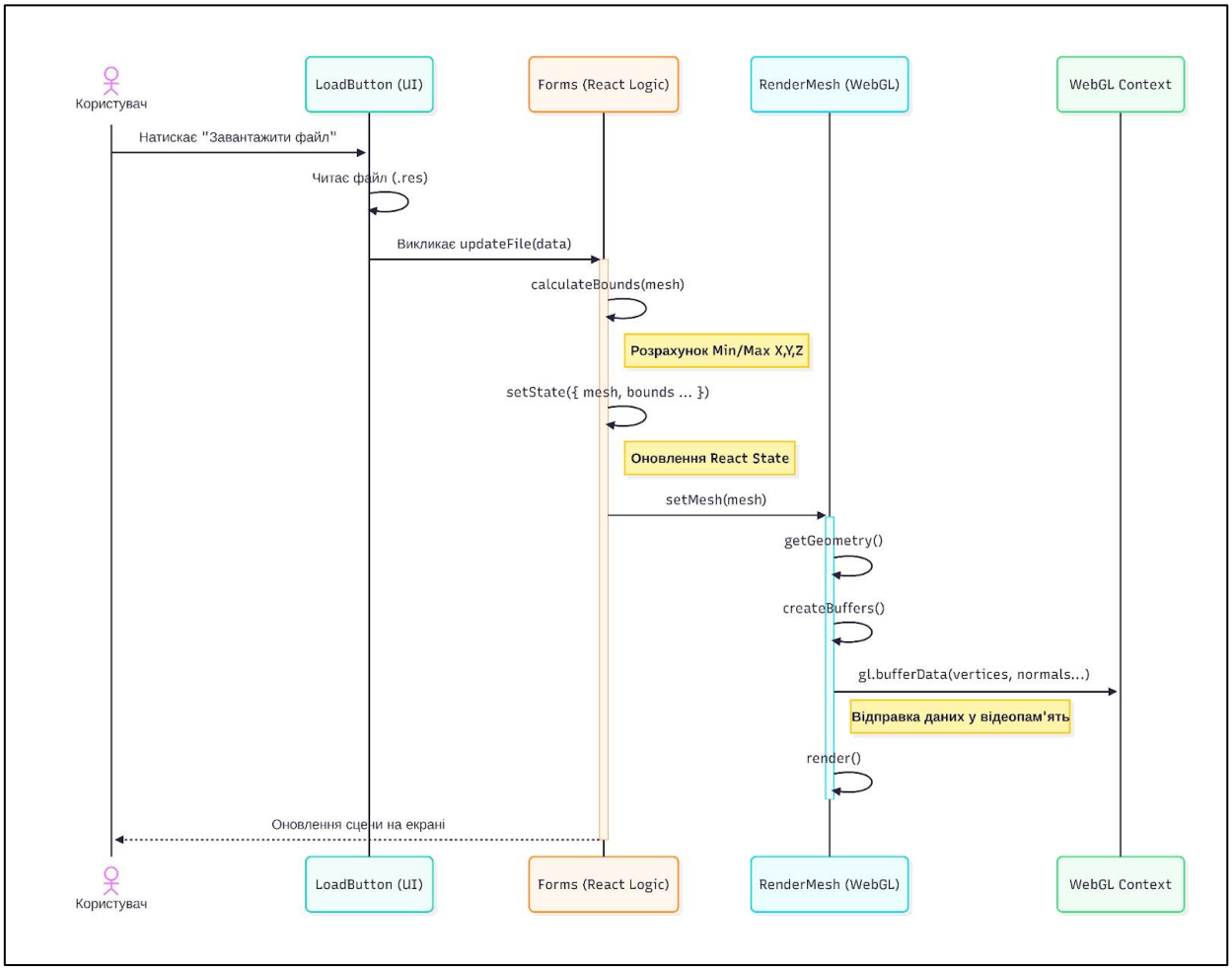


Рисунок 2.2 – Діаграма послідовності процесу завантаження та візуалізації моделі

2.3 Розробка алгоритмів перетворення інженерних даних для GPU

Оскільки WebGL є низькорівневим API, більшість високорівневих операцій візуалізації необхідно реалізовувати алгоритмічно.

2.3.1 Алгоритми тріангуляції та розрахунку нормалей

Для коректного відображення 3D-моделі та розрахунку освітлення необхідно обчислити вектори нормалей до поверхні [13]. У роботі реалізовано алгоритм розрахунку усереднених нормалей у вершинах для забезпечення ефекту гладкого затінення (smooth shading).

Вектор нормалі N_i для і-го вузла розраховується як нормалізована сума нормалей усіх граней, що містять цей вузол:

$$N_i = \frac{\sum_{j \in S_i} n_j}{\left\| \sum_{j \in S_i} n_j \right\|},$$

де n_j – вектор нормалі j-ї грані, S_i – множина граней (трикутників), що примикають до і-го вузла (спільна вершина), ‖...‖ – операція обчислення довжини (модуля) вектора.

2.3.2 Метод колірного відображення функціональних значень

Для візуалізації скалярних полів (напруг, температур) розроблено алгоритм колірного картування (color mapping). Алгоритм виконує лінійну інтерполяцію значення функції *F* у колірний простір RGB.

Процес складається з трьох етапів:

а) визначення діапазону: знаходження мінімального F_{min} та максимального F_{max} значень функції у масиві;

б) нормалізація: приведення значення *F* у вузлі до діапазону [0, 1];

в) генерація кольору: отримання кольору з градієнтної текстури або розрахунок за формулою на основі нормалізованого параметра t ($0 \leq t \leq 1$):

$$t = \frac{F_i - F_{min}}{F_{max} - F_{min}},$$

де F_i – значення фізичної величини у поточному вузлі, F_{min}, F_{max} – мінімальне та максимальне значення функції у всьому масиві результатів.

Отримані кольори записуються у буфер a_color.

2.4 Розробка та інтеграція функції вдосконаленої візуалізації

Одним із ключових завдань роботи є вдосконалення візуалізаційних можливостей шляхом реалізації функції динамічної площини перерізу (clipping plane). Це дозволяє аналізувати внутрішній розподіл полів без зміни геометрії моделі.

2.4.1 Математична модель відсікання

Площина перерізу задається у просторі рівнянням:

$$Ax + By + Cz + D = 0,$$

де (A, B, C) – вектор нормалі до площини, що визначає її орієнтацію, а D – відстань від початку координат.

Для точки $P(x, y, z)$ відстань до площини $(dist)$ визначається як результат скалярного добутку:

$$dist = A \cdot x + B \cdot y + C \cdot z + D.$$

Якщо $dist < 0$, точка знаходиться по один бік площини (у зоні відсікання), якщо $dist > 0$ – по інший (у зоні видимості).

2.4.2 Реалізація алгоритму у фрагментному шейдері

Для забезпечення інтерактивності (обробка у реальному часі) алгоритм реалізовано на рівні фрагментного шейдера WebGL [5]. Це дозволяє виконувати перевірку видимості для кожного пікселя екрана паралельно.

Алгоритм включає наступні кроки:

а) параметри площини передаються у шейдерну програму через uniform-змінну u_clippingPlane;

б) у вершинному шейдері розраховується світова координата вершини, яка передається інтерполятором (varying) у фрагментний рейдер;

в) у фрагментному шейдері для кожного фрагмента обчислюється відстань до площини;

г) застосовується інструкція discard: якщо відстань від'ємна, фрагмент вилучається з конвеєра рендерингу до запису у буфер кадру.

Така реалізація забезпечує високу продуктивність і дозволяє динамічно змінювати положення площини перерізу через інтерфейс користувача без необхідності перебудови геометричних буферів на CPU.

Обчислювальна складність алгоритму візуалізації залежить від кількості вершин N у моделі. Парсинг файлу на CPU має лінійну складність $O(N)$. Рендеринг кадру на GPU завдяки паралельній архітектурі виконується за час, що наближається до $O(1)$ для сучасних відеокарт при $N < 10^6$.

Алгоритм відсікання (clipping) виконується для кожного пікселя екрана, тому його складність залежить від роздільної здатності екрана ($W \times H$), тобто $O(W \cdot H)$, що не залежить від складності геометрії.

2.5 Висновки до другого розділу

У другому розділі виконано проєктування архітектури та розробку ключових алгоритмів вебзастосунку постпроцесора:

- обґрунтовано та спроєктовано компонентну архітектуру системи, що базується на розділенні декларативного інтерфейсу (React) та імперативного графічного ядра (WebGL);
- реалізовано шар взаємодії (Bridge), який забезпечує реактивне оновлення 3D-сцени при зміні параметрів користувачем;
- розроблено структуру даних та алгоритми парсингу вхідних файлів MCE (використання типізованих масивів JavaScript дозволило оптимізувати процес передачі геометричних даних у відеопам’ять GPU);
- реалізовано математичні моделі та шейдерні алгоритми для візуалізації інженерних полів (зокрема, впроваджено метод усереднення нормалей для гладкого затінення та алгоритм колірного картування (color mapping));
- розроблено та інтегровано механізм динамічного відсікання на рівні фрагментного шейдера, що дозволяє виконувати аналіз внутрішньої структури моделі в реальному часі без необхідності перебудови геометрії на CPU.

Запропоновані архітектурні та алгоритмічні рішення створюють фундамент для програмної реалізації високопродуктивного постпроцесора, опис якої наведено у наступному розділі.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ЗАСТОСУНКУ

3.1 Структура та компоненти за стосунку

Аналіз архітектури, проведений у попередніх розділах, дозволяє визначити основні структурні одиниці інтерфейсу, що забезпечують реалізацію функціонала постпроцесора. Застосунок побудовано за модульним принципом, де кожен елемент керування винесено в окремий компонент. Перелік компонентів та їх призначення наведено в таблиці 3.1.

Таблиця 3.1 – Компоненти застосунку

Компонент	**Призначення компонента**
Forms	Головний компонент-контейнер. Керує станом застосунку (state), зберігає посилання на графічне ядро (RenderMesh), обробляє події завантаження файлів та зміни параметрів візуалізації. Відповідає за ініціалізацію WebGL-контексту у методі componentDidMount.
Canvas	Обгортка для HTML-елемента <canvas>. Використовується двічі: для основного шару 3D-графіки (id="gl") та для шару текстових міток (id="text").
LoadButton	Компонент завантаження файлів. Реалізує інтерфейс вибору файлу (input type="file"), зчитує вміст через FileReader та передає дані у батьківський компонент для парсингу.
RenderMesh	Клас графічного ядра (не є React-компонентом, але є ключовим модулем). Інкапсулює логіку WebGL: компіляцію шейдерів, буферизацію даних та цикл рендерингу.
LegendBox	Візуалізація колірної легенди. Динамічно генерує градієнтну шкалу та числові мітки (мінімум/максимум) відповідно до обраної фізичної величини.
ClippingBox	Блок керування динамічним зрізом. Містить перемикач активації зрізу, радіо-кнопки вибору осі (X, Y, Z) та слайдер глибини зрізу. Викликає методи оновлення uniform-змінних шейдера.

Продовження таблиці 3.1

Компонент	**Призначення компонента**
RotateBox	Панель керування орієнтацією моделі. Дозволяє змінювати кути обертання по трьох осях, вмикати/вимикати автообертання та регулювати його швидкість.
ResultBox	Блок вибору результатів. Використовує компонент ListBox для перемикання між фізичними величинами та налаштовує кількість кольорів градієнта.
VisualizationBox	Перемикач режимів відображення (сітка, поверхня, каркас) та керування видимістю координатних осей.
ListBox	Кастомний елемент інтерфейсу для випадаючих списків. Розроблений для коректного відображення поверх WebGL-контексту (вирішує проблему z-index стандартних елементів).
Slider / CheckBox	Базові UI-примітиви, що забезпечують уніфікований вигляд елементів керування у всьому застосунку.

Структура файлів і папок React-проєкту (див. табл. 3.2) спроєктована таким чином, щоб розділити логіку інтерфейсу, математичні обчислення та роботу з файлами. Це дозволяє легко масштабувати застосунок та підтримувати код.

Таблиця 3.2 – Структура файлів і папок проєкту

Папка/файл	**Призначення/вміст**
node_modules/	Папка містить встановлені залежності проєкту.
public/	Папка для статичних файлів. Містить index.html – точку входу, куди монтується React-застосунок, та базові метадані сторінки.
src/	Основна папка вихідного коду за стосунку.
src/draw/draw.js	Містить клас RenderMesh та логіку WebGL.
src/ui/forms.js	Основний файл інтерфейсу. Містить компонент Forms та спеціалізовані панелі керування (RotateBox, ClippingBox тощо).
src/ui/primitives.js	Бібліотека базових UI-компонентів (Slider, Button, CheckBox), що використовуються для побудови інтерфейсу.
src/utils/utils.js	Допоміжні математичні функції (конвертація градусів у радіани, робота з матрицями).

Продовження таблиці 3.2

Папка/файл	**Призначення/вміст**
src/file/file.js	Модуль для читання та парсингу вхідних файлів доступних форматів. Відповідає за перетворення бінарних/текстових даних у масиви JS.
src/index.css	Глобальні стилі за стосунку.
src/index.js	Точка входу для React-застосунку. Відповідає за рендеринг кореневого компонента Forms у DOM-дерево.
package.json	Файл конфігурації проєкту, містить метадані застосунку та перелік залежностей.

Для розробки у роботі були використані пакети, що забезпечують ефективність створення інтерфейсу, тестування та збірки проєкту. Оскільки застосунок створено на базі середовища Create React App, більшість інструментарію інкапсульовано в пакеті react-scripts. Перелік основних залежностей наведено в таблиці 3.3.

Таблиця 3.3 – Залежності проєкту (згідно з package.json)

Пакет	**Версія**	**Призначення пакету**
React	18.2.0	Основна бібліотека для створення інтерфейсів користувача. Відповідає за компонентну архітектуру та керування станом застосунку.
react-dom	18.2.0	Пакет для інтеграції React із DOM-деревом браузера. Забезпечує рендеринг компонентів у HTML-документ.
react-scripts	5.0.1	Набір скриптів та конфігурацій. Інкапсулює в собі Webpack, Babel та ESLint, забезпечуючи готове середовище для розробки, тестування та збірки проєкту без необхідності ручного налаштування конфігураційних файлів.
web-vitals	2.1.4	Бібліотека для вимірювання метрик продуктивності (Core Web Vitals). Дозволяє оцінювати швидкість завантаження та відгуку інтерфейсу на пристрої користувача.
testing-library	13.x	Набір інструментів (react, jest-dom, user-event) для тестування компонентів. Дозволяє імітувати дії користувача та перевіряти коректність відображення елементів інтерфейсу.

3.2 Програмна реалізація графічного ядра

Основним модулем, що відповідає за низькорівневу роботу з графікою, є клас RenderMesh, реалізований мовою JavaScript [8]. Він інкапсулює логіку взаємодії з WebGL API, приховуючи складність конфігурації графічного конвеєра від решти застосунку.

3.2.1 Ініціалізація контексту та компіляція шейдерів

Метод init(glCanvas, textCanvas) є точкою входу для ініціалізації графічного ядра. Він викликається одноразово після монтування DOM-дерева сторінки.

У цьому методі виконуються наступні операції:

– отримання контексту WebGL: здійснюється запит до HTML-елемента <canvas> на отримання контексту webgl або experimental-webgl;
– компіляція шейдерних програм: вихідний код вершинного (vertexShaderSource) та фрагментного (fragmentShaderSource) шейдерів передається компілятору відео карти і, у разі успішної компіляції, створюється програмний об’єкт WebGL (shaderProgram);
– кешування локацій: для оптимізації продуктивності адреси (локації) всіх атрибутів (a_position, a_normal, a_color) та uniform-змінних (u_worldViewProjection, u_clippingPlane) зберігаються у внутрішньому об’єкті this.programInfo, що дозволяє уникнути повільних запитів до GPU під час циклу рендерингу.

Завершивши налаштування статичних параметрів графічного контексту, необхідно організувати передачу динамічних даних моделі (координат, нормалей, кольорів) у відеопам’ять та налаштувати механізм безперервного оновлення сцени.

3.2.2 Обробка буферів даних та цикл рендерингу

Завантаження нової моделі реалізовано у методі setMesh(mesh). Процес підготовки даних включає:

– генерація геометрії: на основі масивів вузлів та елементів формуються типізовані масиви Float32Array для вершин та нормалей;
– завантаження у VBO: створені масиви передаються у пам’ять GPU через методи gl.createBuffer, gl.bindBuffer та gl.bufferData;
– налаштування стану: скидаються параметри трансформації (обертання, масштаб) та ініціюється цикл малювання.

Рендеринг сцени здійснюється методом renderScene(), який викликається рекурсивно через механізм requestAnimationFrame. Це забезпечує синхронізацію з частотою оновлення екрана (зазвичай 60 Гц). У кожному кадрі виконується:

– очищення буферів кольору та глибини (gl.CLEAR_BUFFER_BIT, gl.DEPTH_BUFFER_BIT);
– розрахунок матриць трансформації на основі поточних кутів обертання та масштабу;
– оновлення значень uniform-змінних, зокрема вектора площини перерізу (u_clippingPlane);
– виклик команди малювання gl.drawArrays.

3.3 Реалізація користувацького інтерфейсу

Інтерфейс користувача побудовано за компонентним принципом з використанням бібліотеки React [4]. Головним компонентом, що керує логікою застосунку, є клас Forms.

3.3.1 Головний компонент керування станом

Компонент Forms виконує роль контролера, що пов’язує UI та графічне ядро. У його стані (this.state) зберігаються всі налаштування візуалізації:

а) mesh – посилання на завантажені дані моделі;

б) rotation, scale, translate – параметри афінних перетворень;

в) clippingValue – поточна координата площини перерізу вздовж обраної осі;

г) clippingAxis – обрана вісь відсікання ('x', 'y' або 'z');

д) isClipping – прапорець активації режиму зрізу;

е) funIndex – індекс обраної фізичної величини для візуалізації.

Життєвий цикл компонента використовується для керування графічним ядром: метод componentDidMount ініціалізує екземпляр RenderMesh, а методи-обробники подій (updateRotation, updateFile) передають зміни стану у графічне ядро.

Реалізацію головного вікна вебзастосунку з панеллю керування та областю візуалізації наведено на рисунку 3.1.

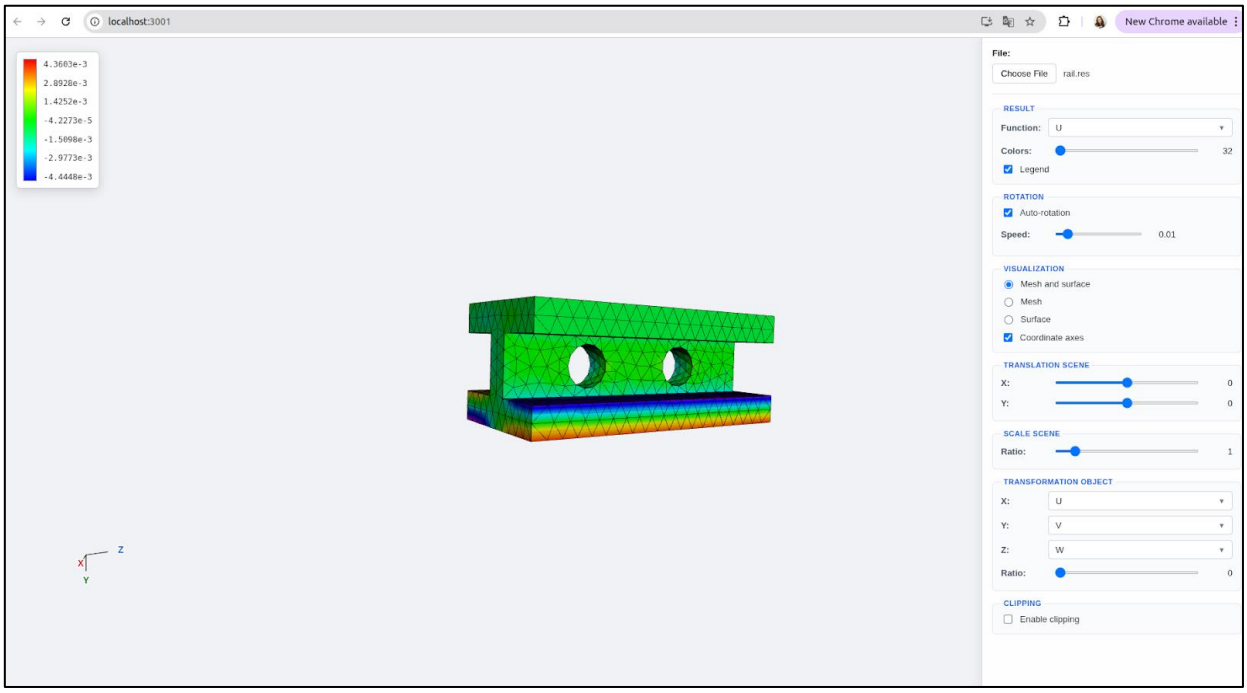


Рисунок 3.1 – Інтерфейс користувача розробленого вебзастосунку

3.3.2 Компоненти керування трансформаціями та візуалізацією

Для забезпечення інтерактивності реалізовано набір спеціалізованих компонентів інтерфейсу: LoadButton – компонент для завантаження та парсингу файлів результатів; RotationBox – група слайдерів для керування кутами Ейлера по осях X, Y, Z; ScaleSceneBox – слайдер для зміни коефіцієнта масштабування; ClippingControl – спеціалізований компонент для керування положенням площини перерізу. Зміна значення слайдера викликає метод setClippingPlane графічного ядра, що миттєво оновлює uniform-змінну у шейдері. Така структура забезпечує реактивність інтерфейсу: зміни параметрів відображаються на 3D-сцені без помітної затримки.

Окрему увагу приділено розробці кастомного компонента ListBox. Стандартні HTML-елементи вибору (<select>) мають обмеження при відображенні поверх активного WebGL-контексту (проблеми з перекриттям та стилізацією). Розроблений компонент забезпечує коректний рендеринг списків поверх 3D-сцени та підтримує віртуалізацію для великої кількості функцій результатів (див. рис. 3.2).

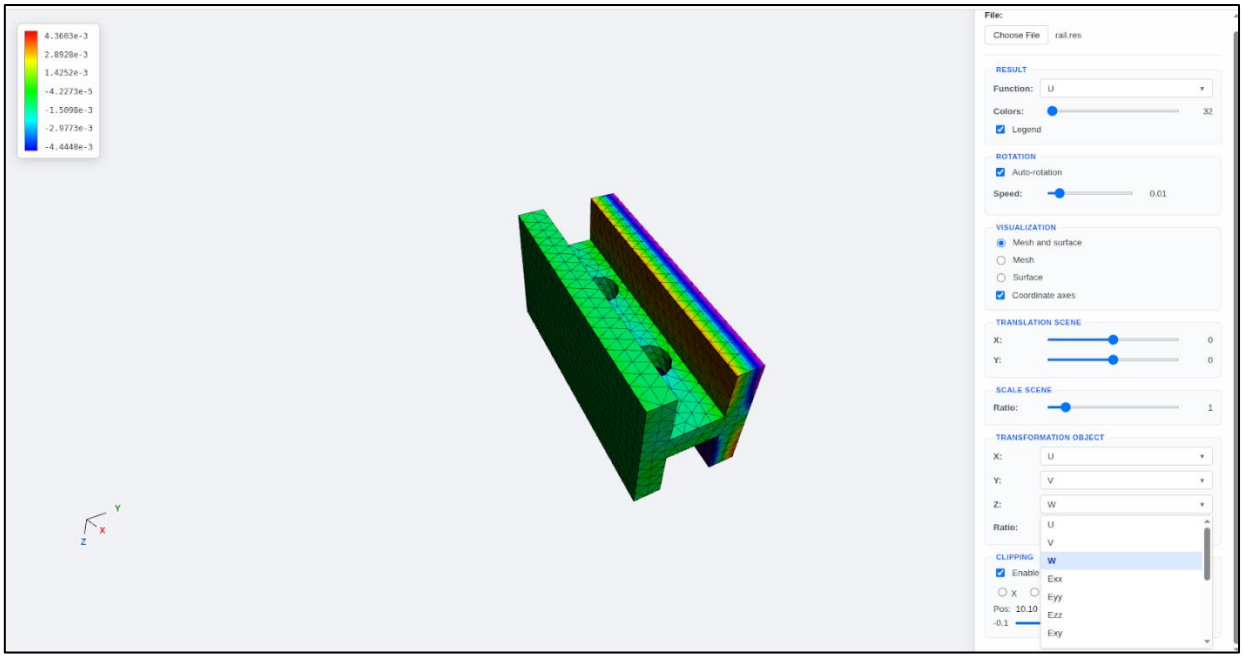


Рисунок 3.2 – Відображення компоненту ListBox

3.4 Тестування продуктивності та верифікація результатів

Для підтвердження ефективності розробленого рішення було проведено серію експериментальних досліджень.

Верифікація функціональності. Перевірка коректності візуалізації проводилася шляхом порівняння зображень, отриманих у вебзастосунку, з результатами еталонного десктопного постпроцесора.

Критерії перевірки:

– геометрична точність: візуальний контроль збігу форми об’єкта та коректності побудови сітки;
– колірне відображення: перевірка відповідності кольорів числовим значенням фізичних величин (напружень, переміщень) згідно з легендою;
– робота зрізу: перевірка коректності відсікання частини моделі площинами.

Тести показали, що внутрішня структура відображається коректно, а межа зрізу чітко відповідає заданій координаті.

Додатково було проведено оцінку якості відтворення градієнтних полів. Візуальний аналіз підтвердив, що переходи кольорів між сусідніми скінченними елементами залишаються плавними, а ізолінії – неперервними, що свідчить про коректну інтерполяцію значень у фрагментному шейдері. Відсутність візуальних артефактів, таких як мерехтіння поверхонь (z-fighting) або розриви сітки при сильному наближенні, гарантує однозначність інтерпретації напружено-деформованого стану конструкції інженером.

Для детального аналізу розподілу фізичних величин всередині об’ємних тіл було протестовано функцію динамічного відсікання (clipping). Як видно з рисунків 3.3-3.5, розроблений шейдерний алгоритм коректно обробляє геометрію та інтерполює кольори на нових гранях при зміні орієнтації площини вздовж осей X, Y та Z, що підтверджує коректність роботи математичної моделі зрізу.

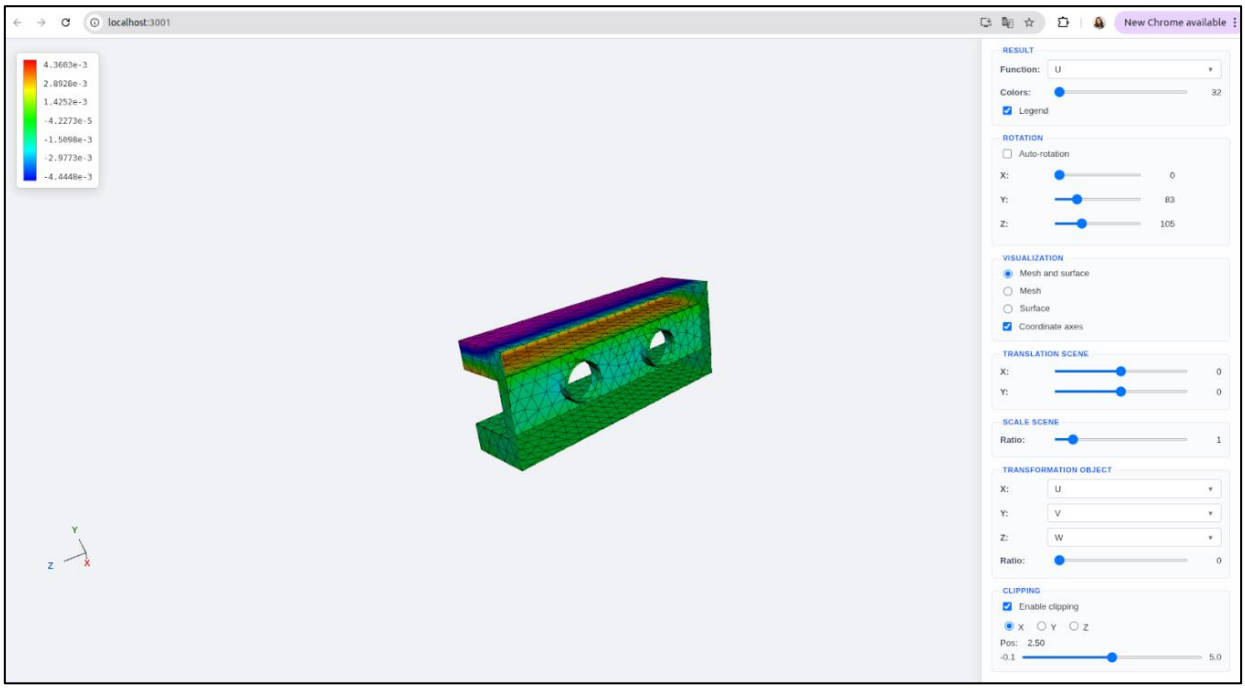


Рисунок 3.3 – Візуалізація розподілу результатів у перерізі моделі площиною по осі X

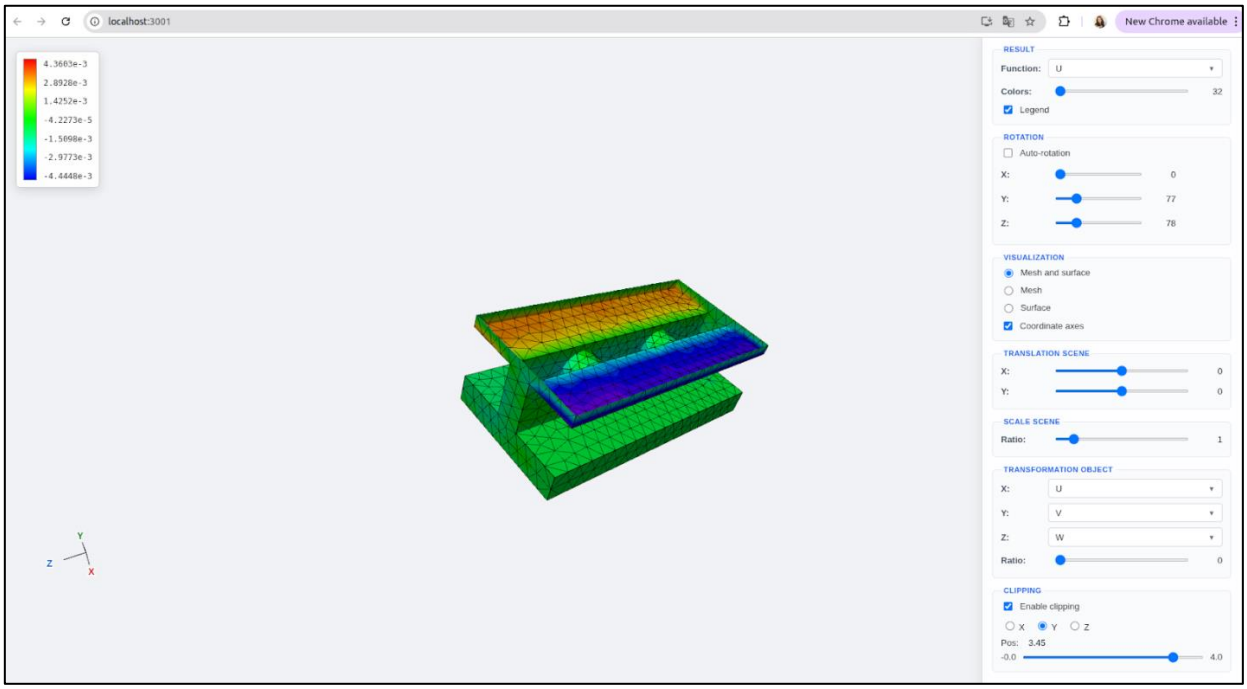


Рисунок 3.4 – Дослідження внутрішніх полів напружень за допомогою вертикального зрізу (вісь Y)

Тестування продуктивності (FPS). Оцінка швидкодії проводилася за допомогою інструментів розробника браузера Google Chrome (Performance Monitor). Тестування виконувалося на моделі обсягом 15 000 елементів.

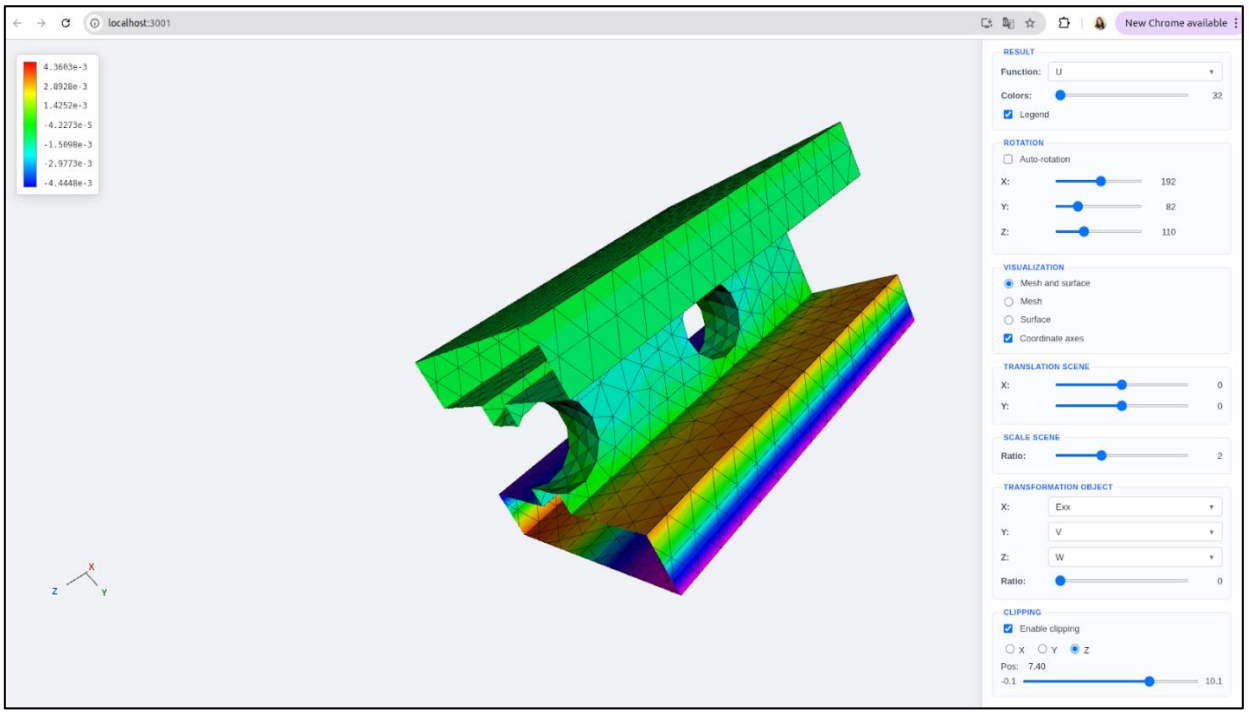


Рисунок 3.5 – Горизонтальний переріз моделі (вісь Z) з відображенням градієнта функціональних значень

Для забезпечення об’єктивності отриманих показників швидкодії, експериментальні дослідження проводилися на апаратному забезпеченні середнього цінового сегмента, що відповідає типовому робочому місцю інженера або студента.

Конфігурація тестового стенда:

– центральний процесор (CPU): Intel Core i5 (або аналог AMD Ryzen 5), 6 ядер, частота 3.2 ГГц;
– графічний адаптер (GPU): інтегрована графіка Intel UHD Graphics або дискретна карта рівня NVIDIA GeForce GTX 1050 (4 ГБ відеопам’яті);
– оперативна пам’ять (RAM): 16 ГБ DDR4;
– операційна система: Windows 10 / 11 (64-bit);
– веббраузер: Google Chrome (версія 120+) з активованою підтримкою апаратного прискорення WebGL 2.0.

Вимірювання частоти кадрів (FPS) здійснювалося за допомогою вбудованих засобів розробника Chrome DevTools (вкладка Rendering > Frame

Rendering Stats), що дозволяє фіксувати падіння продуктивності в реальному часі. Результати вимірювань середньої частоти кадрів (FPS) при різних режимах роботи наведено в таблиці 3.4.

Таблиця 3.4 – Показники продуктивності вебзастосунку на тестовій моделі

Сценарій використання	**Середній FPS (кадрів/с)**	**Час генерації кадру (мс)**	**Навантаження на GPU**
Статична сцена	60	16.6	~120 МБ
Обертання/Масштабування	59-60	16.8	~125 МБ
Увімкнення зрізу (Clipping)	58-60	17.2	~125 МБ
Зміна колірної шкали	45-50	22.0	~140 МБ
Завантаження/Парсинг файлу	–	1200 (сумарно)	Пікове навантаження GPU

Як видно з результатів, застосунок забезпечує стабільну частоту кадрів, близьку до максимальної (60 FPS), навіть при використанні ресурсоємного алгоритму відсікання у фрагментному шейдері. Зниження FPS спостерігається лише при інтенсивних перерахунках колірних масивів на CPU, що не впливає на загальну плавність роботи при перегляді 3D-сцени.

3.5 Розгортання вебзастосунку

Оскільки розроблений застосунок використовує клієнтський рендеринг, його розгортання не потребує спеціалізованих серверів застосунків. Проєкт збирається за допомогою інструменту Webpack (команда npm run build) у набір статичних файлів (HTML, CSS, JS). Ці файли можуть бути розміщені на будь-якому вебсервері (Nginx, Apache) або платформах хостингу статичних сайтів (GitHub Pages, Vercel), що забезпечує високу доступність та мінімальні витрати на інфраструктуру.

ВИСНОВКИ

У результаті виконаної дипломної роботи було успішно розроблено вебзастосунок постпроцесора для інтерактивної 3D-візуалізації результатів розрахунків методом скінченних елементів (МСЕ). Поставлену мету досягнуто шляхом реалізації високопродуктивного клієнтського рішення на базі сучасного технологічного стеку. Були отримані наступні ключові результати, що відповідають завданням, сформульованим у Вступі.

Проаналізовано та обґрунтовано вибір технологічного стеку на основі React та WebGL. Доведено, що це архітектурне рішення є оптимальним для створення високопродуктивного односторінкового застосунку (SPA), здатного ефективно використовувати апаратне прискорення GPU для складних графічних обчислень.

Розроблено архітектуру клієнтського застосунку, що забезпечує чітке розділення відповідальності між декларативним UI (React) та імперативним графічним ядром (WebGL-клас RenderMesh). Встановлено ефективний шар взаємодії, який забезпечує передачу параметрів від елементів керування React до WebGL-ядра.

Реалізовано ядро візуалізації на чистому WebGL (клас RenderMesh). Це ядро успішно виконує:

- перетворення вхідних даних МСЕ (масиви координат, елементів та функціональних значень) у буфери, зрозумілі для GPU;
- розрахунок нормалей та матриць трансформації для 3D-обертання та освітлення;
- ефективний колірний мапінг (color mapping) для відображення інженерних величин на поверхні моделі.

Створено інтерактивний користувацький інтерфейс за допомогою React. Інтерфейс включає функціонал для завантаження даних, керування трансформаціями (обертання, масштабування), вибору функції для візуалізації та керування режимами відображення (сітка/поверхня).

Проведено тестування функціональності розробленого застосунку. Верифікація підтвердила коректність геометричної побудови, точність колірного відображення та коректність роботи алгоритму динамічного відсікання (clipping) для всіх координатних площин (X, Y, Z).

Таким чином, розроблений вебзастосунок постпроцесора являє собою повноцінний інструмент, який може бути використаний для доступного та швидкого аналізу результатів скінченно-елементного моделювання в середовищі веббраузера.

ПЕРЕЛІК ПОСИЛАНЬ

1. Гоменюк С. І., Козуб В. Ю. Паралельна реалізація методу скінченних елементів для задачі термопружності. *Вісник Східноукраїнського національного університету імені Володимира Даля*. 2019. № 5 (275). С. 5–9. URL: https://doi.org/10.33216/1998-7927-2019-256-8-5-9 (дата звернення: 03.09.2025).

2. Гребенюк С. М., Гоменюк С. І. Чисельні методи розв’язання механічних задач : навч. посіб. для здобувачів третього освіт.-наук. рівня спец. «Прикладна математика». Запоріжжя : ЗНУ, 2022. 80 с.

3. Abbas H., Flayyih W. Postgraduate Student Management System: Web Application Made with React.js and Express.js. *Journal of Advanced Web Applications*. 2023. Vol. 10, no. 5. P. 215–221. DOI: 10.1155/2023.5671234.

4. Bhatt D., Parekh K., Minat M., Patel B. ReactJS: A Comprehensive Analysis of its Features, Performance, and Suitability for Modern Web Development. *International Journal of Scientific Research in Engineering and Management (IJSREM)*. 2023. Vol. 07, Iss. 04. URL: https://ijsrem.com/download/reactjs-a-comprehensive-analysis-of-its-features-performance-and-suitability-for-modern-web-development/ (дата звернення: 17.09.2025).

5. Design and Implementation of WebGL Model Rendering Engine / X. Liu et al. *Scientific Journal of Intelligent Systems Research*. 2024. Vol. 6, no. 7. P. 18–23. URL: https://doi.org/10.54691/778tvh90 (дата звернення: 26.09.2025).

6. Homeniuk S., Kozub V. Features of the use of parallel calculations in the package of application programs «MIRELA+». *Herald of Khmelnytskyi National University. Technical sciences.* 2022. Vol. 315, no. 6. P. 60–64. DOI: https://doi.org/10.31891/2307-5732-2022-315-6(2)-60-64 (дата звернення: 27.09.2025).

7. Jonathan R., Suprihadi. Development of Front-End Web Applications Utilizing Single Page Application Framework and React.js Library. *International*

Journal of Software Engineering and Computer Science (IJSECS). 2023. Vol. 3, no. 3. P. 529–536. DOI: 10.35870/ijsecs.v3i3.1943.

8. JR Development. WebGL React component – EP1. *YouTube*. 2018. URL: https://www.youtube.com/watch?v=wbI55mc7ksE (дата звернення: 12.10.2025).

9. MDN. Data in WebGL – Web APIs. URL: https://developer.mozilla.org/enUS/docs/Web/API/WebGL_API/Data (дата звернення: 26.10.2025).

10. OpenGL Programming Guide : The Official Guide to Learning OpenGL, Version 4.3 / D. Shreiner et al. Addison-Wesley, 2013. 986 p.

11. Vertices and Indices. WebGL Beginner's Guide. *Packt*. URL: https://subscription.packtpub.com/book/gamedevelopment/9781849691727/2/ch02lvl1sec21/vertices-andindices (дата звернення: 21.10.2025).

12. Srivastava A. K., Laxmi V., Singh P., Pratima K., Kirti V. React JS (Open Source JavaScript Library). *International Journal of Innovative Research in Technology (IJIRT).* 2021. Vol. 8, no. 9. P. 342–346. URL: https://ijirt.org/publishedpaper/IJIRT153854_PAPER.pdf (дата звернення: 27.10.2025).

13. A Beginner's Guide: What Are Normals in 3D Modeling? *The Tech Edvocate*. URL: https://www.thetechedvocate.org/a-beginners-guide-what-are-normalsin-3d-modeling/ (дата звернення: 29.10.2025).

14. WebGL Shaders and GLSL. *WebGL Fundamentals*. URL: https://webglfundamentals.org/webgl/lessons/webgl-shaders-and-glsl.html (дата звернення: 30.10.2025).

ДОДАТОК А

Лістинг програмного коду шейдерів (GLSL)

А.1 Вершинний шейдер (Vertex Shader)

Відповідає за трансформацію координат та розрахунок відстані до площини зрізу.

```
attribute vec4 a_position;
attribute vec3 a_normal;
attribute vec4 a_color;
uniform mat4 u_worldViewProjection;
uniform mat4 u_worldInverseTranspose;
uniform vec4 u_translation_center;
uniform vec4 u_translation;
uniform vec4 u_scale;
uniform vec4 u_clippingPlane;
varying vec3 v_normal;
varying vec4 v_color;
varying float v_dist;

void main() {
  vec4 worldPosition = u_scale * a_position - u_scale * u_translation_center;
  gl_Position = u_worldViewProjection * worldPosition - u_translation;
  // orient the normals and pass to the fragment shader
  v_normal = mat3(u_worldInverseTranspose) * a_normal;
  v_color = a_color;
  v_dist = dot(a_position, u_clippingPlane);
}
```

А.2 Фрагментний шейдер (Fragment Shader)

Реалізує логіку відсікання (clipping) та фінальне зафарбовування пікселя.

```
precision mediump float;
// Passed in from the vertex shader.
varying vec3 v_normal;
varying vec4 v_color;
varying float v_dist;
uniform vec3 u_reverseLightDirection;
uniform vec4 u_color;
uniform int u_is_mesh;
uniform bool u_clippingEnabled;

void main() {
   if (u_clippingEnabled && v_dist < 0.0) {
      discard;
   }
   vec3 normal = normalize(v_normal);
   float light = dot(normal, u_reverseLightDirection);

   if (u_is_mesh == 0)
      gl_FragColor = v_color;
   else
      gl_FragColor = u_color;

   gl_FragColor.rgb *= abs(light);
}
```

ДОДАТОК Б

Фрагменти коду реалізації логіки клієнтської частини (JavaScript)

Б.1 Алгоритм розрахунку габаритів моделі (Bounding Box)

Використовується для автоматичного центрування камери та налаштування меж слайдерів зрізу.

```
const calculateBounds = (mesh) => {
    const bounds = {
        minX: -10, maxX: 10,
        minY: -10, maxY: 10,
        minZ: -10, maxZ: 10
    };
    if (!mesh || !mesh.x || mesh.x.length === 0) return bounds;
    const nodes = mesh.x;
    let minX = Infinity, maxX = -Infinity;
    let minY = Infinity, maxY = -Infinity;
    let minZ = Infinity, maxZ = -Infinity;

    for (let i = 0; i < nodes.length; i++) {
        const x = nodes[i][0];
        const y = nodes[i][1];
        const z = (nodes[i].length > 2) ? nodes[i][2] : 0;

        if (x < minX) minX = x; if (x > maxX) maxX = x;
        if (y < minY) minY = y; if (y > maxY) maxY = y;
        if (z < minZ) minZ = z; if (z > maxZ) maxZ = z;
    }
```

```
        if (minX === Infinity) return bounds;

        const addBuffer = (min, max) => {
            const buf = (max - min) * 0.01;
            return buf === 0 ? { min: min - 1, max: max + 1 } : { min: min - buf, max: max + buf };
        };

        const xB = addBuffer(minX, maxX);
        const yB = addBuffer(minY, maxY);
        const zB = addBuffer(minZ, maxZ);

        return {
            minX: xB.min, maxX: xB.max,
            minY: yB.min, maxY: yB.max,
            minZ: zB.min, maxZ: zB.max
        };
    };
```

Б.2 Метод формування площини зрізу у класі RenderMesh

Формує вектор нормалі площини залежно від обраної осі та передає його у шейдер.

```
    // value - координата зрізу
    // enabled - стан (увімкнено/вимкнено)
    // axis - обрана вісь ('x', 'y', 'z')
      setClippingPlane(value, enabled, axis = 'z') {
        let plane;
        if (axis === 'x') {
```

```
            plane = [-1, 0, 0, value];
        } else if (axis === 'y') {
            plane = [0, -1, 0, value];
        } else {
            plane = [0, 0, -1, value];
        }

        this.clippingPlane = plane;
        this.clippingEnabled = enabled;

        if (!this.isAutoRotation) {
            this.renderScene();
        }
    }
```