МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра комп'ютерних наук

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА КОЛАБОРАТИВНОГО ВЕБ-РЕДАКТОРА 3D-СЦЕН НА БАЗІ THREE.JS»**

Виконав: студент 2 курсу, групи 8.1222-он
спеціальності 122 Комп'ютерні науки
(шифр і назва спеціальності)
освітньої програми Комп'ютерні науки
(назва освітньої програми)
А.С. Демиденко
(ініціали та прізвище)
Керівник доцент кафедри комп'ютерних наук, доцент, к.т.н. Решевська К.С.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент завідувач кафедри фундаментальної та прикладної математики, професор, д.т.н. Гребенюк С.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра комп'ютерних наук

Рівень вищої освіти магістр

Спеціальність 122 Комп'ютерні науки
(шифр і назва)

Освітня програма Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерних наук,
д.т.н., професор

_____ Шило Г.М.
(підпис)

" ____ " _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Демиденку Артему Сергійовичу
(прізвище, ім'я та по-батькові)

1. Тема роботи Розробка колаборативного веб-редактора 3d-сцен на базі Three.js

керівник роботи Решевська Катерина Сергіївна, к.т.н., доцент
(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 08.06.2026

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

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі.
2. Основні теоретичні відомості.
3. Розробка колаборативного веб-редактора 3D-сцен на базі Three.js

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
презентація

6. Консультанти розділів роботи

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

7. Дата видачі завдання 03.03.2025

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.	01.04.2025	
2.	Збір вихідних даних.	02.06.2025	
3.	Обробка методичних та теоретичних джерел.	04.08.2025	
4.	Розробка першого та другого розділу.	01.12.2025	
5.	Розробка третього розділу.	01.04.2026	
6.	Оформлення та нормоконтроль кваліфікаційної роботи магістра.	08.06.2026	
7.	Захист кваліфікаційної роботи.	20.06.2026	

Студент _____ А.С. Демиденко
(підпис) (ініціали та прізвище)

Керівник роботи _____ К.С. Решевська
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ О.Г. Спиця
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка колаборативного веб-редактора 3D-сцен на базі Three.js»: 120 с., 20 рис., 3 табл., 24 джерел, 1 додаток.

АЛГОРИТМ УЗГОДЖЕННЯ, ВЕБ-ЗАСТОСУНОК, ВЕБСОКЕТ, ЖУРНАЛ ОПЕРАЦІЙ, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА, КОЛАБОРАТИВНЕ РЕДАГУВАННЯ, ПРОТОКОЛ СИНХРОНІЗАЦІЇ, ТРИВИМІРНА ГРАФІКА, ТРИВИМІРНА СЦЕНА, УЗГОДЖЕНІСТЬ ДАНИХ.

Метою роботи є розроблення колаборативного веб-редактора 3D-сцен на базі Three.js, який забезпечує спільне редагування об’єктів сцени в режимі реального часу, централізовану синхронізацію змін між клієнтами, збереження стану сцени та відновлення її актуального представлення після повторного підключення користувачів. Для досягнення мети розв’язано такі задачі: проаналізовано предметну область колаборативних систем реального часу; досліджено моделі подання тривимірних сцен і методи синхронізації стану; побудовано формальну модель сцени та алгоритм узгодження змін; спроектовано клієнт-серверну архітектуру системи; реалізовано програмний засіб і проведено його експериментальне тестування.

Об’єктом дослідження є процеси спільного редагування тривимірних сцен у веб-орієнтованому багатокористувацькому середовищі.

Предмет дослідження становлять моделі подання тривимірної сцени, протоколи та алгоритми синхронізації змін, а також програмні засоби реалізації колаборативного веб-редактора 3D-сцен на базі Three.js, Node.js, WebSocket і MongoDB.

У роботі використано методи системного аналізу, формалізації та математичного моделювання, об’єктно-орієнтованого і клієнт-серверного проектування, програмної інженерії та експериментального тестування.

У результаті дослідження розроблено веб-редактор Collab3D, у якому реалізовано централізовану серіалізацію змін на сервері, синхронізацію через

WebSocket, підтримку операцій create, delete, transform, color, rename, import_model, механізм попереднього перегляду transform_preview, збереження даних у MongoDB та GridFS, а також відновлення актуального стану сцени через snapshot після повторного підключення клієнта. Новизна отриманих результатів полягає в адаптації моделі узгодження змін до веб-орієнтованого багатокористувацького редагування 3D-сцен і в поєднанні журналу операцій із snapshot-підходом до підтримки undo/redo.

Результати роботи взаємопов’язані з дослідженнями у сфері колаборативних систем реального часу, веб-орієнтованої тривимірної графіки та клієнт-серверних застосунків. Вони можуть бути використані в освітньому процесі, під час демонстрації принципів колаборативного редагування, для створення прототипів веб-інструментів спільної роботи з 3D-сценами та як основа для подальшого розвитку системи. Практична значущість роботи полягає в реалізації працездатного програмного засобу, а проведене тестування підтвердило можливість його використання для багатокористувацького редагування тривимірних сцен у режимі реального часу.

SUMMARY

Master's Qualifying Theses "Development of a collaborative web-based 3D scene editor based on Three.js": 120 pages, 20 figures, 3 tables, 24 references, 1 supplement.

KEYWORDS: AGREEMENT ALGORITHM, WEB-APPLICATION, WEBSOCKET, OPERATION LOG, CLIENT-SERVER ARCHITECTURE, COLLABORATIVE EDITING, SYNCHRONIZATION PROTOCOL, THREE-DIMENSIONAL GRAPHICS, THREE-DIMENSIONAL SCENE, DATA CONSISTENCY.

The aim of the study is to develop a collaborative web-based 3D scene editor based on Three.js that provides real-time collaborative editing of scene objects, centralized synchronization of changes between clients, scene state persistence, and restoration of the current scene representation after users reconnect. To achieve this aim, the following tasks were solved: the subject area of real-time collaborative systems was analyzed; models of three-dimensional scene representation and state synchronization methods were studied; a formal scene model and a change agreement algorithm were developed; a client-server architecture was designed; the software tool was implemented and experimentally tested.

The object of the study is the processes of collaborative editing of three-dimensional scenes in a web-oriented multi-user environment.

The subject of the study includes models of three-dimensional scene representation, protocols and algorithms for synchronizing changes, as well as software tools for implementing a collaborative web-based 3D scene editor based on Three.js, Node.js, WebSocket, and MongoDB.

The research methods include system analysis, formalization and mathematical modeling, object-oriented and client-server design, software engineering, and experimental testing.

As a result of the research, the Collab3D web editor was developed. It implements centralized server-side serialization of changes, WebSocket-based synchronization, support for create, delete, transform, color, rename, and import_model operations, the transform_preview live preview mechanism, data storage in MongoDB and GridFS, and restoration of the current scene state through snapshot synchronization after client reconnection. The novelty of the obtained results lies in the adaptation of the change agreement model to web-oriented multi-user 3D scene editing and in the combination of an operation log with a snapshot-based undo/redo mechanism.

The results are related to studies in the fields of real-time collaborative systems, web-based three-dimensional graphics, and client-server applications. They can be used in the educational process, for demonstrating collaborative editing principles, for developing prototypes of shared web tools for working with 3D scenes, and as a basis for further development of the system. The practical significance of the work lies in the implementation of a functioning software tool, and the conducted testing confirmed the possibility of its use for multi-user editing of three-dimensional scenes in real time.

ЗМІСТ

Завдання на кваліфікаційну роботу 2
Реферат 4
Summary 6
Вступ 11
1 Аналіз предметної області та існуючих методів синхронізації 14
1.1 Колаборативні системи реального часу 14
1.1.1 Поняття та характеристики колаборативних систем 14
1.1.2 Архітектурні моделі колаборативних систем 16
1.1.3 Приклади сучасних колаборативних систем 17
1.2 Представлення тривимірних сцен у веб-середовищі 19
1.2.1 Технологічні основи тривимірної графіки в браузері 19
1.2.2 Scene Graph як модель представлення сцени 19
1.2.3 Особливості 3D-сцени як об’єкта синхронізації 20
1.3 Методи синхронізації стану в колаборативних системах 22
1.3.1 Повна синхронізація стану (state broadcasting) 23
1.3.2 Диференційована синхронізація (diff-based synchronization) 23
1.3.3 Operational Transformation (OT) 24
1.3.4 CRDT (Conflict-Free Replicated Data Types) 25
1.3.5 Порівняльний аналіз підходів 27
1.4 Проблеми узгодженості в тривимірних колаборативних системах 28
1.4.1 Конфлікти трансформацій 28
1.4.2 Конфлікти структурних змін 29
1.4.3 Числова нестабільність та накопичення похибок 30
1.4.4 Проблема частоти оновлень 30
1.4.5 Масштабованість системи 31
1.4.6 Відновлення після втрати з’єднання 31
1.5 Висновки до розділу 1 32

2 Формальна модель та алгоритм узгодження змін у тривимірній сцені 34
2.1 Математична модель тривимірної сцени 34
2.2 Формальна модель журналу операцій та механізму синхронізації 36
2.3 Алгоритм узгодження конкурентних змін у тривимірній сцені 38
2.4 Формальний аналіз коректності та ефективності алгоритму 39
2.5 Висновки до розділу 2 41
3 Проектування та реалізація колаборативної системи редагування 3d-сцен 43
3.1 Архітектура системи 43
3.1.1 Загальна концепція архітектури 43
3.1.2 Логічна структура системи 44
3.1.3 Архітектурна модель взаємодії 45
3.1.4 Модель станів клієнта і сервера 45
3.1.5 Формат обміну повідомленнями 47
3.1.6 Механізм відновлення після втрати з’єднання 48
3.1.7 Забезпечення масштабованості та стійкості 48
3.2 Технологічний стек та обґрунтування вибору інструментів 49
3.2.1 Загальні вимоги до технологічної платформи 49
3.2.2 Клієнтська частина 50
3.2.3 Серверна частина 51
3.2.4 Система збереження даних 52
3.2.5 Архітектурна взаємодія компонентів 53
3.2.6 Обґрунтування вибору веб-архітектури 54
3.2.7 Відповідність теоретичній моделі 55
3.3 Протокол взаємодії клієнта і сервера та структура даних 55
3.3.1 Формат представлення об’єктів сцени 56
3.3.2 Формат і семантика WebSocket-повідомлень 59
3.3.3 Сценарій початкової синхронізації та відновлення стану 62
3.3.4 Механізм *undo/redo* і журналювання змін 66
3.3.5 Збереження й відкриття сцени, імпорт зовнішніх моделей 69
3.4. Реалізація серверної підсистеми 71

3.4.1 HTTP-частина сервера та допоміжні маршрути 72
3.4.2 WebSocket-сервер і обробка операцій 73
3.4.3 Організація збереження даних у MongoDB 76
3.4.4 Використання GridFS для збереження 3D-моделей 80
3.5 Реалізація клієнтської підсистеми 82
3.5.1 Побудова та рендеринг сцени у Three.js 82
3.5.2 Механізми вибору об’єктів і трансформацій 84
3.5.3 Реалізація користувацького інтерфейсу редактора 85
3.5.4 Синхронізація змін між клієнтами в режимі реального часу 87
3.6 Висновки до розділу 3 91
4 Експериментальне дослідження та тестування системи 95
4.1 Умови та методика експериментального дослідження 95
4.1.1 Мета та завдання експериментального дослідження 95
4.1.2 Умови проведення дослідження 96
4.1.3 Методика проведення експериментів 98
4.1.4 Критерії оцінювання результатів 99
4.1.5 Межі та припущення експериментальної постановки 101
4.2 Функціональне тестування основних можливостей редактора 102
4.3 Перевірка синхронізації при одночасній роботі кількох клієнтів 105
4.4 Аналіз продуктивності, стійкості та відновлення після розриву з’єднання 108
4.5 Висновки до розділу 4 111
Висновки 114
Перелік посилань 116
Додаток А Використані платформи при розробці колаборативного 3D-редактора 120

ВСТУП

Актуальність дослідження. Розвиток веб-технологій, хмарних сервісів і засобів спільної взаємодії зумовив зростання потреби у програмних системах, що забезпечують одночасну роботу кількох користувачів над єдиним цифровим об’єктом у режимі реального часу. Якщо для текстових документів, електронних таблиць і двовимірної графіки такі рішення вже стали звичними, то для тривимірних сцен задача залишається значно складнішою. Це пов’язано з ієрархічною структурою 3D-сцени, просторовим характером трансформацій, необхідністю синхронізації положення, орієнтації, масштабу, кольору та складу об’єктів, а також з потребою відновлення узгодженого стану після мережевих затримок або розривів з’єднання.

Практична значущість цієї проблеми полягає у можливості використання колаборативних 3D-редакторів у навчанні, інженерному моделюванні, спільному проектуванні цифрових об’єктів, прототипуванні інтерфейсів віртуального середовища та демонстраційних системах. У зв’язку з цим розроблення колаборативного веб-редактора 3D-сцен на базі сучасних браузерних технологій є актуальним науково-практичним завданням, а дослідження моделей подання сцени та алгоритмів узгодження змін має як теоретичне, так і прикладне значення.

Мета роботи. Метою магістерської роботи є розроблення колаборативного веб-редактора 3D-сцен на базі Three.js, який забезпечує спільне редагування об’єктів сцени в режимі реального часу, централізовану синхронізацію змін між клієнтами, підтримку збереження стану сцени та відновлення її актуального представлення після повторного підключення користувачів.

Для досягнення поставленої мети необхідно було розв’язати такі задачі:

- проаналізувати предметну область колаборативних систем реального часу та особливості подання тривимірних сцен у веб-середовищі;

- дослідити існуючі підходи до синхронізації стану в розподілених системах і визначити їх придатність для колаборативного редагування 3D-сцен;
- побудувати формальну модель тривимірної сцени, множини операцій над нею та алгоритму узгодження конкурентних змін;
- спроектувати архітектуру клієнт-серверної системи колаборативного редагування 3D-сцен;
- обґрунтувати вибір технологічного стеку для реалізації клієнтської, серверної та серверно-орієнтованої підсистем збереження даних;
- реалізувати протокол взаємодії клієнта і сервера, структури даних, механізми початкової синхронізації та відновлення стану сцени;
- реалізувати серверну підсистему з підтримкою журналювання операцій, збереження поточного snapshot-стану сцени, undo/redo та збереження зовнішніх 3D-моделей;
- реалізувати клієнтську підсистему з візуалізацією сцени, інструментами трансформації, інспектором властивостей, імпортом моделей і локальним збереженням сцени;
- виконати експериментальне дослідження та функціональне тестування розробленої системи.

Об’єкт дослідження. Процеси спільного редагування тривимірних сцен у веб-орієнтованому багатокористувацькому середовищі.

Предмет дослідження. Моделі подання тривимірної сцени, протоколи та алгоритми синхронізації змін, а також програмні засоби реалізації колаборативного веб-редактора 3D-сцен на базі Three.js, Node.js, WebSocket і MongoDB.

Методи дослідження. У роботі використано методи системного аналізу для дослідження предметної області, методи формалізації та математичного моделювання для опису сцени й алгоритму узгодження змін, методи об’єктно-орієнтованого та клієнт-серверного проектування для побудови архітектури системи, а також методи програмної інженерії та експериментального

тестування для реалізації та перевірки працездатності розробленого програмного засобу.

Структура роботи. Магістерська робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел і додатків. У першому розділі проаналізовано предметну область, представлення тривимірних сцен у веб-середовищі та існуючі підходи до синхронізації стану в колаборативних системах. У другому розділі побудовано формальну модель тривимірної сцени та алгоритму узгодження змін. У третьому розділі розглянуто проектування та реалізацію колаборативної системи редагування 3D-сцен. У четвертому розділі наведено методику експериментального дослідження, результати функціонального тестування, перевірки синхронізації, продуктивності та стійкості системи. У висновках узагальнено основні результати дослідження та оцінено ступінь досягнення поставленої мети.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ МЕТОДІВ СИНХРОНІЗАЦІЇ

1.1 Колаборативні системи реального часу

1.1.1 Поняття та характеристики колаборативних систем

Сучасний етап розвитку інформаційних технологій характеризується активним впровадженням розподілених програмних систем, що забезпечують спільну роботу користувачів над єдиним інформаційним ресурсом. Такі системи отримали назву колаборативних (collaborative systems). Їх основною особливістю є підтримка одночасної взаємодії кількох користувачів із спільним об’єктом або середовищем у режимі реального часу.

Колаборативна система реального часу – це розподілена програмна система, у якій декілька користувачів можуть виконувати операції над спільними даними з мінімальною затримкою, при цьому всі учасники отримують узгоджене представлення поточного стану системи [1; 2]:

Ключовими характеристиками таких систем є:

- спільний стан (shared state) – множина даних, що доступна для редагування кількома користувачами;
- конкурентність операцій (concurrency) – можливість одночасного виконання змін різними учасниками;
- мережева затримка (network latency) – неминуча затримка передачі повідомлень між вузлами;
- узгодженість стану (consistency) – властивість системи підтримувати логічно коректний та синхронізований стан для всіх учасників.

Залежно від способу взаємодії розрізняють:

- асинхронні системи, де користувачі працюють у різний час, а зміни синхронізуються із затримкою;

– синхронні системи, у яких зміни відображаються майже миттєво для всіх учасників.

Для задачі редагування тривимірної сцени актуальною є саме синхронна модель, оскільки користувачі повинні спостерігати зміни об’єктів у режимі реального часу.

Важливою теоретичною основою колаборативних систем є моделі узгодженості (consistency models) [1; 2; 3].

У колаборативних системах важливим є не лише сам факт передавання змін між клієнтами, а й те, наскільки однаковим залишається подання спільного стану для різних користувачів. Для цього застосовуються різні моделі узгодженості, які визначають допустимий рівень розбіжності між копіями даних у розподіленому середовищі.

Сильна узгодженість передбачає, що після виконання операції всі учасники системи мають працювати з однаковим актуальним станом. Така модель є найбільш строгою, однак у веб-системах реального часу її підтримка потребує значних витрат на синхронізацію та може збільшувати затримки взаємодії.

Остаточна узгодженість допускає тимчасові розбіжності між клієнтами, але гарантує, що після завершення обміну повідомленнями всі копії даних поступово набудуть однакового стану. Для колаборативного 3D-редактора така модель є практично важливою, оскільки під час переміщення або обертання об’єктів окремі клієнти можуть отримувати оновлення з невеликою затримкою.

Причинна узгодженість орієнтована на збереження логічного порядку залежних операцій. Наприклад, якщо об’єкт спочатку створюється, а потім трансформується, система повинна гарантувати, що операція зміни положення не буде застосована раніше за операцію створення відповідного об’єкта. Саме така вимога є важливою для підтримки коректного стану тривимірної сцени [1, 2, 3, 4].

Через фізичні обмеження швидкості передачі даних та розподілену природу мережі Інтернет забезпечення сильної узгодженості у глобальних

системах реального часу є складним і ресурсномісткім завданням. Тому більшість сучасних колаборативних платформ використовують моделі остаточної або причинної узгодженості [2, 3, 4].

1.1.2 Архітектурні моделі колаборативних систем

Архітектура колаборативної системи визначає спосіб організації взаємодії між клієнтами та механізм підтримки спільного стану:

а) централізована. У централізованій архітектурі всі клієнтські застосунки взаємодіють із сервером, який виконує роль координатора спільного стану. У контексті колаборативного редагування 3D-сцени такий сервер приймає операції користувачів, визначає порядок їх застосування, оновлює поточний стан сцени та розсилає підтверджені зміни іншим учасникам.

Перевагою такого підходу є те, що система має єдину точку прийняття рішень. Це спрощує реалізацію журналу операцій, механізмів undo/redo, збереження snapshot-стану сцени та відновлення після повторного підключення користувача. Крім того, централізована модель дозволяє легше контролювати цілісність даних, оскільки всі зміни проходять через один серверний вузол.

Водночас така архітектура має і певні обмеження. Сервер стає критичним компонентом системи: у разі його недоступності клієнти втрачають можливість повноцінної синхронізації. Крім того, зі збільшенням кількості користувачів, об’єктів сцени та частоти трансформацій зростає навантаження на сервер і мережевий канал. Тому для масштабування подібних систем у майбутньому можуть знадобитися механізми балансування навантаження, оптимізації частоти повідомлень або розподіленого збереження стану.

Ця модель є найбільш поширеною у веб-застосунках, зокрема у системах спільного редагування документів [1; 2; 5].

б) децентралізована P2P (peer-to-peer) архітектура. Децентралізована, або peer-to-peer, архітектура передбачає, що учасники системи можуть обмінюватися змінами безпосередньо, без постійного посередництва центрального сервера. На

перший погляд такий підхід є привабливим, оскільки зменшує залежність від одного серверного вузла та потенційно розподіляє навантаження між клієнтами.

Однак для задачі колаборативного редагування 3D-сцен P2P-модель створює низку додаткових складнощів. Кожен клієнт повинен не лише передавати власні зміни, а й коректно приймати, впорядковувати та застосовувати операції інших користувачів. Особливо проблемними є ситуації, коли кілька учасників одночасно змінюють один і той самий об’єкт, наприклад переміщують його, змінюють масштаб або видаляють зі сцени.

У такому випадку система повинна самостійно визначати, яка операція має пріоритет, як уникнути дублювання змін і яким чином синхронізувати нового користувача, який підключився пізніше. Для тривимірної сцени ці проблеми ускладнюються тим, що об’єкти мають просторові параметри, ієрархічні зв’язки та можуть містити зовнішні ресурси, зокрема імпортовані GLB/GLTF-моделі. Саме тому в межах цієї роботи доцільнішою є централізована клієнт-серверна архітектура, у якій сервер відповідає за впорядкування операцій і підтримку узгодженого стану сцени.

Для систем редагування складних структур даних, зокрема тривимірних сцен, доцільним є використання централізованої або гібридної моделі, що спрощує контроль над глобальним станом сцени [2].

1.1.3 Приклади сучасних колаборативних систем

Еволюція колаборативних систем простежується у різних предметних областях – від текстових редакторів до графічних та інженерних платформ.

Одним із найбільш відомих прикладів є текстові редактори спільного доступу, зокрема Google Docs, які реалізують синхронізацію змін у реальному часі на основі спеціалізованих алгоритмів трансформації операцій [1].

У сфері графічного дизайну прикладом є платформа Figma, яка забезпечує одночасне редагування векторної графіки багатьма користувачами. Особливістю

таких систем є необхідність узгодження змін у складних структурованих документах [2].

У галузі тривимірної графіки більшість професійних редакторів, таких як Blender, традиційно орієнтовані на локальну роботу одного користувача. Колаборативні можливості реалізуються через додаткові плагіни або системи керування версіями, що не забезпечують повноцінної синхронної взаємодії.

Ігрові рушії, наприклад Unity, підтримують мережеву взаємодію об’єктів сцени, проте їх архітектура орієнтована на синхронізацію ігрового стану, а не на спільне редагування геометрії та матеріалів сцени.

Аналіз існуючих рішень показує, що:

- текстові колаборативні системи працюють із лінійними структурами даних;
- графічні 2D-редактори мають ієрархічну структуру, але оперують переважно плоскими об’єктами;
- 3D-сцена є складною ієрархічною структурою з просторовими трансформаціями, що створює додаткові труднощі при узгодженні стану [1; 2; 8; 9].

Таким чином, задачі синхронізації у тривимірному середовищі є більш складними порівняно з текстовими або двовимірними системами, оскільки:

- зміни мають просторовий характер;
- порядок застосування трансформацій впливає на кінцевий результат;
- об’єкти можуть бути пов’язані ієрархічними залежностями;
- обсяг переданих даних є значно більшим.

1.2 Представлення тривимірних сцен у веб-середовищі

1.2.1 Технологічні основи тривимірної графіки в браузері

Розвиток веб-технологій упродовж останнього десятиліття суттєво розширив можливості браузерного середовища, зокрема у сфері інтерактивної тривимірної графіки. Ключовою технологією, що забезпечує апаратне прискорення 3D-візуалізації у веб-застосунках, є WebGL – програмний інтерфейс, заснований на OpenGL ES, який дозволяє виконувати рендеринг графіки за допомогою графічного процесора (GPU) без використання сторонніх плагінів.

WebGL реалізує програмований графічний конвеєр (graphics pipeline), який включає такі основні етапи:

а) передача геометричних даних до GPU;

б) виконання вершинного шейдера (vertex shader);

в) растеризація;

г) виконання фрагментного шейдера (fragment shader);

д) формування фінального зображення у буфері кадру;

Безпосередня робота з WebGL передбачає низькорівневе керування буферами, шейдерами та матрицями трансформації, що значно ускладнює розробку складних систем [6; 7; 19]. Тому у практиці веб-розробки використовуються бібліотеки високого рівня абстракції, які інкапсулюють складність графічного конвеєра та надають об’єктно-орієнтовану модель сцени.

1.2.2 Scene Graph як модель представлення сцени

Сучасні бібліотеки тривимірної графіки для веб-середовища, зокрема Three.js, використовують підхід scene graph (граф сцени) для організації об’єктів. Scene graph є ієрархічною структурою даних, у якій кожен вузол відповідає окремому об’єкту або групі об’єктів [8, 9].

Узагальнено тривимірну сцену можна представити як орієнтований ациклічний граф:

– вершини (nodes) – геометричні об’єкти, джерела світла, камери;
– ребра – ієрархічні зв’язки між об’єктами.

Ієрархічна організація дозволяє реалізувати наслідування трансформацій: якщо дочірній об’єкт вкладений у батьківський, його положення визначається відносно системи координат батьківського вузла. Таким чином, глобальна трансформація вузла є результатом композиції локальних матриць перетворення вздовж шляху від кореня графа до відповідного вузла.

Типовий вузол сцени містить такі компоненти:

– геометрію (geometry) – множину вершин, нормалей та індексів;
– матеріал (material) – параметри відображення;
– матрицю трансформації (transformation matrix);
– список дочірніх вузлів.

Матрична форма представлення трансформацій базується на використанні однорідних координат та матриць 4×4, що дозволяє описувати переміщення, масштабування та обертання в єдиному математичному форматі. Порядок множення матриць є критично важливим, оскільки операції не є комутативними. Це має принципове значення для задач синхронізації, оскільки різний порядок застосування трансформацій може призводити до різних кінцевих станів сцени.

Таким чином, 3D-сцена не є простою множиною об’єктів, а складною ієрархічною структурою з просторовими залежностями [8; 9].

1.2.3 Особливості 3D-сцени як об’єкта синхронізації

Представлення сцени у вигляді графа з матричними трансформаціями зумовлює низку специфічних властивостей, які суттєво впливають на механізми колаборативного редагування:

а) ієрархічна залежність об’єктів. Зміна трансформації батьківського вузла автоматично змінює глобальні координати всіх дочірніх елементів. Отже,

операції над одним вузлом можуть опосередковано впливати на значну частину сцени. Це ускладнює визначення локальності змін і підвищує складність алгоритмів узгодження;

б) просторовий характер операцій. На відміну від текстових систем, де операції мають лінійну природу (вставка, видалення символів), у 3D-середовищі основними операціями є:

- трансформація (переміщення, обертання, масштабування);
- зміна геометрії;
- зміна матеріалу;
- додавання або видалення об’єктів.

Трансформації описуються неперервними величинами з плаваючою комою. Накопичення похибок округлення може призводити до числової нестабільності та розбіжностей між копіями сцени у різних клієнтів;

в) висока частота оновлень. Під час інтерактивного переміщення об’єкта (drag-and-drop) клієнт генерує десятки оновлень на секунду. Передача кожного такого оновлення через мережу створює значне навантаження та може призвести до зростання затримок. Отже, необхідні механізми агрегації або оптимізації передачі змін;

г) значний обсяг даних. 3D-об’єкти можуть містити тисячі або мільйони вершин. Повна передача геометрії при кожній зміні є недоцільною. Це вимагає диференційованого підходу до синхронізації – передавання лише модифікованих параметрів, а не всього стану;

д) вплив порядку застосування операцій. Оскільки матричні перетворення не є комутативними, результат послідовного застосування двох трансформацій залежить від їх порядку. У розподіленому середовищі, де операції можуть надходити з різною затримкою, виникає необхідність забезпечення узгодженого порядку виконання або застосування алгоритмів, що гарантують інваріантність кінцевого стану.

1.2.4 Вимоги до системи представлення сцени у колаборативному середовищі

З урахуванням наведених особливостей можна сформулювати основні вимоги до моделі представлення тривимірної сцени в умовах багатокористувацького редагування:

- формалізованість структури – можливість математичного опису сцени як графа;
- локальність операцій – чітке визначення області впливу кожної зміни;
- ідентифікованість вузлів – наявність унікальних ідентифікаторів об’єктів;
- можливість версіонування – підтримка історії змін;
- стійкість до конкурентних модифікацій – забезпечення узгодженого кінцевого стану при одночасних операціях.

Реалізація цих вимог потребує поєднання структурованої моделі даних та ефективних алгоритмів синхронізації, що буде розглянуто у наступних підрозділах.

1.3 Методи синхронізації стану в колаборативних системах

Забезпечення узгодженості спільного стану є центральною проблемою будь-якої колаборативної системи реального часу. У розподіленому середовищі декілька клієнтів можуть одночасно виконувати операції над спільними даними, тоді як передача повідомлень відбувається з непередбачуваними затримками. Це створює умови для виникнення конфліктів та розбіжностей у копіях стану.

У сучасних системах застосовуються різні підходи до синхронізації, які відрізняються складністю реалізації, вимогами до мережевих ресурсів та рівнем гарантій узгодженості. У даному підрозділі розглянуто основні методи, що можуть бути застосовані для узгодження стану тривимірної сцени.

1.3.1 Повна синхронізація стану (state broadcasting)

Найпростішим підходом є передача повного стану об’єкта або всієї сцени після кожної зміни [1; 2]. У централізованій архітектурі алгоритм функціонує таким чином:

а) клієнт виконує зміну;

б) новий стан надсилається серверу;

в) сервер розповсюджує оновлений стан іншим клієнтам;

г) клієнти замінюють локальну копію на отриману.

Переваги методу:

– простота реалізації;

– відсутність складних алгоритмів трансформації;

– гарантована узгодженість за умови централізованого контролю.

Недоліки:

– значний обсяг переданих даних;

– високе мережеве навантаження;

– ризик втрати проміжних змін;

– проблема «останній запис перемагає» (last-write-wins).

У випадку 3D-сцени передача повного стану є неефективною через великий розмір геометричних даних та високу частоту оновлень під час інтерактивних трансформацій [2; 8; 9]. Тому даний підхід може використовуватися лише як базова або резервна стратегія.

1.3.2 Диференційована синхронізація (diff-based synchronization)

Більш ефективним є підхід, за якого передаються лише зміни (diff), а не весь стан об’єкта. Операція надсилається у вигляді модифікації певних параметрів:

– зміна координат;

– зміна кута обертання;

– зміна властивостей матеріалу.

Переваги:

– зменшення обсягу переданих даних;

– зниження затримки;

– масштабованість при великій кількості об’єктів.

Однак виникають нові проблеми:

– необхідність збереження версій або часових міток;

– залежність результату від порядку застосування операцій;

– потенційна розсинхронізація при втраті повідомлень.

У разі конкурентних змін одного й того ж об’єкта сервер повинен визначити порядок їх застосування або реалізувати механізм вирішення конфлікту.

Диференційований підхід є більш придатним для 3D-сцен, проте потребує додаткового алгоритмічного забезпечення для гарантування узгодженості [2].

1.3.3 Operational Transformation (OT)

Operational Transformation (OT) – це алгоритмічний підхід, розроблений для підтримки спільного редагування документів у реальному часі. Основна ідея полягає у трансформації операцій таким чином, щоб забезпечити їх коректне застосування незалежно від порядку отримання [1; 2].

Якщо дві операції виконані паралельно, одна з них модифікується з урахуванням іншої перед застосуванням. Формально:

Нехай O_1 і O_2 – дві конкурентні операції. Тоді визначається функція трансформації T така, що:

$$O_1' = T(O_1, O_2) \tag{1.1}$$

Після трансформації обидві операції можуть бути застосовані без порушення узгодженості.

Переваги OT:

– забезпечення високого рівня узгодженості;
– доведена ефективність у текстових системах;
– контрольований порядок застосування змін.

Недоліки:

– складність реалізації;
– необхідність підтримки журналу операцій;
– труднощі при роботі з нелінійними структурами даних.

OT добре працює для лінійних структур (текст), де позиції елементів можна чітко визначити індексами. Однак у випадку графових структур, зокрема 3D-сцен з ієрархічними залежностями, реалізація трансформацій стає значно складнішою [2; 8; 9].

1.3.4 CRDT (Conflict-Free Replicated Data Types)

CRDT (Conflict-Free Replicated Data Types) — це підхід до побудови розподілених структур даних, за якого різні копії одного об’єкта можуть оновлюватися незалежно, але з часом мають сходитися до узгодженого стану. На відміну від підходів, де конфлікти розв’язуються централізовано, CRDT орієнтовані на таку організацію операцій, за якої порядок доставки повідомлень не порушує коректність кінцевого результату [3; 4].

У контексті CRDT важливою є властивість операцій давати однаковий підсумковий результат незалежно від того, в якій послідовності вони були отримані різними репліками системи. Таку властивість називають комутативністю. Для колаборативного середовища це означає, що якщо два клієнти отримали однаковий набір незалежних змін у різному порядку, їхній локальний стан усе одно має зійтися до одного результату. Найпростіші приклади: додавання (a+b=b+a) та множення (a*b=b*a) чисел.

Наприклад, для простих множин операції додавання різних елементів можуть бути застосовані у довільній послідовності без зміни кінцевого

результату. Проте для тривимірних трансформацій ситуація складніша: послідовне застосування обертання, масштабування та переміщення може давати різний результат залежно від порядку операцій. Саме тому пряме перенесення CRDT-підходів на 3D-сцену потребує додаткового аналізу та адаптації.

Ще однією важливою властивістю для розподілених систем є ідемпотентність. Вона означає, що повторне застосування однієї й тієї самої операції не повинно змінювати результат після її першого виконання. У практичному сенсі це важливо тоді, коли повідомлення може бути доставлене повторно через мережеві збої, повторне підключення клієнта або механізм відновлення стану.

Для колаборативного 3D-редактора ідемпотентність є важливою під час обробки операцій створення, видалення або оновлення об’єкта. Наприклад, якщо сервер або клієнт повторно отримає повідомлення про встановлення певної позиції об’єкта, результат не повинен відрізнятися від одноразового застосування цієї операції. Це знижує ризик накопичення помилок і допомагає підтримувати стабільний стан сцени після тимчасових розривів WebSocket-з’єднання.

Водночас не всі операції в 3D-середовищі є ідемпотентними за своєю природою. Наприклад, команда «зсунути об’єкт на 1 одиницю вправо» при багаторазовому застосуванні щоразу змінюватиме координату. Натомість команда «встановити координату X = 5» є більш стабільною, оскільки повторне застосування не змінює кінцевий стан. Саме тому в розробленій системі доцільно передавати не лише інкрементні зміни, а й абсолютні значення позиції, обертання та масштабу.

Існують два основних типи CRDT:

а) Operation-based (CmRDT) – репліки обмінюються операціями;

б) State-based (CvRDT) – репліки періодично обмінюються станами, які зливаються за допомогою операції merge.

Головна перевага CRDT полягає в тому, що незалежно від порядку доставки повідомлень усі репліки сходяться до однакового стану (eventual consistency) [3; 4].

Переваги:

– відсутність необхідності складної трансформації операцій;
– стійкість до мережевих затримок;
– можливість роботи в офлайн-режимі.

Недоліки:

– складність моделювання складних структур;
– збільшення обсягу метаданих;
– необхідність спеціалізованої реалізації для конкретного типу даних.

Для 3D-сцени CRDT може бути застосований до:

– множини об’єктів;
– властивостей об’єктів;
– трансформацій як незалежних атрибутів.

Проте стандартні CRDT не враховують просторових залежностей і некомутативності матричних операцій, що потребує додаткової адаптації [2; 3; 4; 8; 9].

1.3.5 Порівняльний аналіз підходів

Порівняння розглянутих методів дозволяє визначити їх придатність для задачі синхронізації тривимірної сцени (див. табл. 1.1).

Аналіз показує, що прості підходи не забезпечують достатньої масштабованості та гнучкості для 3D-сцен. OT забезпечує високий рівень узгодженості, проте складність реалізації для графових структур є суттєвою. CRDT є перспективним напрямом для колаборативного редагування, але їх застосування до тривимірних сцен потребує спеціальної адаптації. Зокрема, необхідно враховувати некомутативність матричних трансформацій, можливість

одночасної зміни одного об’єкта кількома користувачами та потребу в узгодженому представленні ієрархічної структури сцени.

Таблиця 1.1 – Порівняння методів синхронізації

Метод	Складність реалізації	Мережеве навантаження	Узгодженість	Придатність для 3D
Повна синхронізація	Низька	Високе	Сильна (централізовано)	Обмежена
Diff-based	Середня	Низьке	Залежить від порядку	Помірна
OT	Висока	Помірне	Висока	Ускладнена
CRDT	Висока	Помірне	Не залежить від порядку (eventual consistency)	Перспективна

1.4 Проблеми узгодженості в тривимірних колаборативних системах

Колаборативне редагування тривимірних сцен має низку специфічних особливостей, що суттєво ускладнюють забезпечення узгодженості стану порівняно з текстовими або двовимірними системами. Джерелом цих труднощів є просторовий характер даних, ієрархічна структура сцени та математичні властивості трансформацій [2; 8; 9].

У даному підрозділі проаналізовано ключові проблеми, які виникають під час синхронізації 3D-середовища в умовах конкурентного доступу.

1.4.1 Конфлікти трансформацій

Основними операціями редагування у 3D-сцені є переміщення, обертання та масштабування об’єктів. Ці операції описуються матрицями перетворення або відповідними параметрами (позиція, кут, масштаб).

Нехай два користувачі одночасно виконують операції над одним об’єктом:

– користувач А виконує обертання;

– користувач В виконує переміщення.

У разі асинхронного отримання повідомлень можливі різні порядки застосування операцій:

$$T_1 = R \cdot P, \tag{1.2}$$

$$T_2 = P \cdot R, \tag{1.3}$$

де R – матриця обертання;

P – матриця переміщення.

Оскільки матричне множення не є комутативним, загальна трансформація залежить від порядку виконання операцій. У результаті різні клієнти можуть отримати відмінні фінальні стани об’єкта [2; 8; 9].

Таким чином, навіть за умови однакових вихідних операцій відсутність узгодженого порядку їх застосування призводить до розсинхронізації сцени.

1.4.2 Конфлікти структурних змін

Окрім трансформацій, можливі конфлікти структурного характеру:

- одночасне видалення та редагування одного об’єкта;
- переміщення об’єкта в іншу гілку ієрархії під час його трансформації;
- одночасне додавання об’єктів із однаковими ідентифікаторами.

Наприклад, якщо один користувач видаляє об’єкт, а інший у той самий момент змінює його координати, система повинна визначити, яка операція має пріоритет. Неправильне вирішення такого конфлікту може призвести до «висячих» посилань у графі сцени або помилок рендерингу.

Особливу складність становить синхронізація ієрархічних змін, оскільки переміщення вузла в графі може змінити систему координат для всіх дочірніх елементів.

1.4.3 Числова нестабільність та накопичення похибок

Трансформації у тривимірній графіці базуються на числах з плаваючою комою. Під час послідовного застосування інкрементних змін виникає накопичення похибок округлення.

У розподіленому середовищі проблема посилюється через:

– різні частоти оновлення клієнтів;

– можливу агрегацію або втрату частини операцій;

– відмінності у порядку застосування трансформацій.

Навіть незначні відхилення в координатах можуть з часом призвести до помітного розходження станів сцени між клієнтами. Це явище відоме як drift (дрейф стану).

Для забезпечення стабільності необхідно або періодично виконувати нормалізацію стану, або використовувати абсолютні значення замість накопичення інкрементів.

1.4.4 Проблема частоти оновлень

Під час інтерактивного редагування (наприклад, перетягування об’єкта мишею) клієнт генерує десятки подій на секунду [2]. Якщо кожна подія передається через мережу, виникають:

– перевантаження каналу зв’язку;

– зростання затримки;

– накопичення черги повідомлень.

У результаті інші користувачі можуть спостерігати ривки або застарілий стан об’єкта.

Необхідно реалізовувати механізми:

– обмеження частоти передачі (throttling);

– агрегації змін;

– передавання лише фінального стану після завершення дії.

Однак такі оптимізації можуть впливати на точність синхронізації та створювати додаткові виклики для алгоритмів узгодження.

1.4.5 Масштабованість системи

При збільшенні кількості користувачів та об’єктів сцени зростає:

- кількість конкурентних операцій;
- обсяг мережевого трафіку;
- навантаження на сервер.

У централізованій архітектурі сервер стає вузьким місцем, оскільки повинен обробляти всі зміни та розповсюджувати їх між клієнтами [1; 2]. Зі збільшенням N користувачів кількість повідомлень може зростати пропорційно N^2.

Для тривимірних сцен із великою кількістю об’єктів це створює ризик зниження продуктивності та збільшення затримок, що негативно впливає на сприйняття системи користувачами.

1.4.6 Відновлення після втрати з’єднання

У реальних умовах користувач може тимчасово втратити з’єднання з сервером. Після повторного підключення система повинна:

- синхронізувати актуальний стан сцени;
- коректно інтегрувати локальні зміни, якщо вони виконувалися в офлайн-режимі;
- уникнути дублювання операцій.

1.5 Висновки до розділу 1

Колаборативні системи реального часу є складними розподіленими програмними комплексами, основною проблемою яких є забезпечення узгодженості спільного стану в умовах конкурентного доступу та мережевих затримок. Існуючі рішення демонструють ефективність у текстових та двовимірних середовищах, однак застосування аналогічних підходів до тривимірних сцен вимагає додаткового аналізу та формалізації.

Тривимірна сцена у веб-середовищі реалізується на основі технології WebGL та організовується у вигляді ієрархічного графа об’єктів. Використання scene graph забезпечує зручне структурування об’єктів, однак водночас ускладнює процес синхронізації через ієрархічні залежності, просторовий характер трансформацій та некомутативність операцій.

Специфіка 3D-середовища визначає підвищені вимоги до алгоритмів узгодження змін та потребує формальної моделі сцени, яка дозволить математично описати структуру даних і операції над нею. Саме така модель буде розроблена у наступному розділі роботи.

Розглянуті методи синхронізації мають різний рівень складності та ефективності. Для задачі колаборативного редагування тривимірної сцени необхідний підхід, який:

- підтримує ієрархічну структуру даних;
- мінімізує мережеве навантаження;
- забезпечує узгодженість при конкурентних змінах;
- коректно обробляє просторові трансформації.

Жоден із базових методів у чистому вигляді не повністю задовольняє всі вимоги. Це обґрунтовує необхідність формалізації моделі сцени та розробки або адаптації алгоритму синхронізації, що буде здійснено у наступних розділах роботи.

Аналіз проблем узгодженості в тривимірних колаборативних системах показує, що синхронізація 3D-сцени є значно складнішою порівняно з текстовими або двовимірними середовищами. Основними труднощами є:

- некомутативність матричних трансформацій;
- ієрархічна структура графа сцени;
- числова нестабільність;
- висока частота оновлень;
- масштабування при збільшенні кількості користувачів.

Жоден із стандартних підходів до синхронізації не враховує повною мірою всі ці особливості. Це обґрунтовує необхідність формального опису структури тривимірної сцени та розробки спеціалізованого алгоритму узгодження змін, що буде розглянуто у наступному розділі.

З огляду на структурну складність 3D-сцени та необхідність підтримки просторових трансформацій у реальному часі, виникає потреба у розробці спеціалізованої моделі представлення сцени та алгоритмів узгодження змін, що і становить предмет подальшого дослідження.

2 ФОРМАЛЬНА МОДЕЛЬ ТА АЛГОРИТМ УЗГОДЖЕННЯ ЗМІН У ТРИВИМІРНІЙ СЦЕНІ

2.1 Математична модель тривимірної сцени

Для побудови коректного алгоритму синхронізації змін у колаборативному редакторі тривимірних сцен необхідно формально визначити об’єкт синхронізації, множину допустимих операцій та правила переходу системи між станами. У межах цієї роботи тривимірна сцена розглядається як скінченна множина об’єктів, кожен з яких має унікальний ідентифікатор, тип, геометричні параметри розміщення у просторі, візуальні атрибути та, за потреби, посилання на зовнішній ресурс моделі.

Нехай стан сцени в момент часу *t* задається множиною об’єктів:

$$S_t = \{O_1, O_{2,\dots}, O_n\} \tag{2.1}$$

Кожний об’єкт O_i описується кортежем:

$$O_i = \{id_i, type_i, tr_i, color_i, name_i, res_i\} \tag{2.2}$$

Тут id_i – унікальний ідентифікатор об’єкта;

$type_i = \{cub, sphere, rhombus, triangle, model\}$ – тип об’єкта;

$tr_i = \{p_i, r_i, s_i\}$ – трансформаційний стан,

де p_i– позиція, r_i– орієнтація, s_i– масштаб;

$color_i$– колір;

$name_i$– користувацька назва або псевдонім;

res_i– додатковий ресурс, який для імпортованих моделей містить пару <*fileId, url*>, а для геометричних примітивів має порожнє значення.

У поточній реалізації редактора узгоджуваною одиницею є верхньорівневий колаборативний об’єкт сцени, тобто примітив або кореневий вузол імпортованої моделі. Це відповідає клієнтській логіці, де локальна таблиця *objects* відображає *objId* у кореневий *Object3D*, а для GLB/GLTF-моделей при виборі виконується підйом від дочірнього *Mesh* до кореневого вузла з *userData.id*. Трансформації також застосовуються саме до такого кореневого вузла, а не до довільного внутрішнього дочірнього елемента моделі.

Узагальнено 3D-сцену можна розглядати як ієрархічну структуру з батьківсько-дочірніми відношеннями між вузлами. Для такої моделі природно вводити функцію *parent*, яка задає батьківський вузол для кожного об’єкта, а також операцію *reparent*. Проте в поточному прототипі така операція не реалізована. Серверний формат збереження об’єктів містить поля *id, type, fileId, url, position, rotation, scale, color і customName*, але не містить *parentId*, тому в подальшому викладі *reparent* доцільно трактувати лише як можливе розширення узагальненої моделі, а не як уже реалізовану функцію системи.

$$parent(id_i) = \perp,\ for\ all\ i \tag{2.3}$$

Практичне подання стану сцени на сервері зручно задавати як відображення $Snap_t: I \to D$, де D є множиною записів фіксованої структури. Саме така форма відповідає реалізації *snapshot*: сервер будує асоціативну структуру за ключем *obj.id*, а значенням є набір атрибутів об’єкта – тип, просторовий стан, колір, назва, а для моделей також *fileId* і *url*. Такий підхід є зручним для початкової синхронізації, відновлення після перепідключення та реалізації *undo/redo* через відновлення попередніх станів сцени.

2.2 Формальна модель журналу операцій та механізму синхронізації

Після визначення структури сцени потрібно задати множину операцій, що змінюють її стан. У реалізованій системі до журнальованих операцій належать *create, delete, transform, color, rename* та *import_model*. Крім того, у протоколі використовуються службові команди *undo_request, redo_request, snapshot, client_ping* і *client_pong*, а також окрема множина ефемерних повідомлень *transform_preview*.

$$\Omega_h = \{create, delete, transform, color, rename, import_model\} \quad (2.4)$$

$$\Omega_c = \{undo_request, redo_request, snapshot, client_ping, client_pong\} \quad (2.5)$$

$$\Omega_p = \{transform_preview\} \quad (2.6)$$

Журнальовані операції змінюють логічний стан сцени та мають бути серіалізовані, застосовані до поточного *snapshot* і збережені в журналі. Повідомлення *transform_preview* використовується лише для візуального відображення безперервного переміщення, обертання або масштабування об’єкта під час взаємодії користувача й свідомо не потрапляє до історії *undo/redo*. Це дозволяє зменшити навантаження на сховище та уникнути надмірної деталізації історії змін при drag-операціях.

Кожну журнальовану операцію зручно подати як запис:

$$e_k = \{v_k, op_k, id_k, data_k\} \quad (2.7)$$

Тут v_k – номер версії, що призначається сервером; op_k – тип операції; id_k – ідентифікатор об’єкта; $data_k$ – корисне навантаження. Для різних типів операцій структура $data_k$ є різною:

Для *create*:

$$data_k = \{type, position, rotation, scale, color, customName\} \quad (2.8)$$

для *import_model*:

$$data_k = \{fileId, url, position, rotation, scale, color, customName\} \quad (2.9)$$

для *transform, color і rename* передаються значення {*before, after*}.

Фізично в системі використовується централізований серверний координатор, який виконує роль єдиної точки серіалізації. Після надходження операції сервер нормалізує дані, присвоює нове значення *version*, модифікує колекцію поточного стану *objects*, записує саму операцію до *operations* і розсилає її клієнтам через WebSocket. Для імпортованих моделей додатково використовується GridFS, який забезпечує довготривале збереження GLB- та GLTF-файлів, а в *snapshot* і журналі фіксуються ідентифікатор файлу та URL доступу до нього.

Окремо слід формалізувати механізм *undo/redo*. У поточному прототипі він реалізований не як послідовне застосування інверсних операцій до журналу, а як керування покажчиком на історію *snapshot*-станів. Формально нехай

$$H = \{S^{(0)}, S^{(1)}, \ldots, S^{(m)}\} \quad (2.10)$$

– послідовність збережених станів сцени, а

$$p \in \{0, \ldots, m\} \quad (2.11)$$

– покажчик на активний стан.

Після кожної журнальованої операції сервер будує новий *snapshot* сцени та додає його в *H*; якщо перед цим було виконано *undo* і покажчик не вказував на кінець історії, усі стани праворуч від *p* відкидаються. Операція *undo_request* замінює *p := p - 1*, а *redo_request – p := p + 1*.

2.3 Алгоритм узгодження конкурентних змін у тривимірній сцені

Основна ідея алгоритму полягає в тому, що всі зміни стану сцени проходять через один серверний вузол, який виконує серіалізацію конкурентних дій клієнтів. Такий підхід усуває необхідність у peer-to-peer-механізмах досягнення консенсусу та забезпечує однозначний порядок застосування змін.

Нехай клієнт C_i генерує операцію *op*. Якщо *op* належить множині журнальованих операцій, вона надсилається на сервер у форматі JSON-повідомлення. Якщо користувач виконує безперервну трансформацію об'єкта, під час руху генеруються проміжні *transform_preview*, а наприкінці взаємодії формується одна комітна операція *transform* з полями *before* та *after*. Це означає, що в історію потрапляє лише підсумкова зміна, а не весь ланцюжок мікропереміщень.

Після отримання повідомлення сервер виконує розбір JSON і класифікацію за типом операції. Далі можливі три основні сценарії. Перший сценарій – початкова синхронізація. Після встановлення WebSocket-з'єднання сервер формує поточний *snapshot* сцени та відправляє його клієнту. Новий клієнт не відтворює всю історію операцій, а одразу отримує консистентний знімок актуального стану. Це важливо як для старту нової сесії, так і для відновлення після тимчасового розриву з'єднання.

Другий сценарій – ефемерна операція попереднього перегляду. Якщо *op* = *transform_preview*, сервер не змінює колекції *objects*, не додає запис у *operations* і не оновлює *history*. Повідомлення просто розсилається активним клієнтам. Формально така операція не змінює канонічний стан S_t, а лише породжує короткочасний візуальний стан $\widetilde{S_t}$, який існує до надходження комітної операції *transform* або нового *snapshot*.

Третій сценарій – журнальована операція. Якщо *op* належить Ω_h, сервер призначає нову версію *op.version := nextVersion()*, застосовує зміну до поточного *snapshot.* Для *create* та *import_model* створюється або оновлюється запис об'єкта; для *delete* відповідний об'єкт вилучається; для *transform* оновлюються поля

позиції, орієнтації та масштабу; для *color* – колір; для *rename* – користувацька назва. Після цього операція заноситься в *operations*, розсилається, а оновлений *snapshot* сцени додається до історії *H.* Якщо перед цим виконувався *undo* і покажчик *p* не стояв у кінці історії, *redo*-гілка відкидається.

Операції *undo* та *redo* мають іншу семантику. Вони не інтерпретуються як звичайні предметні дії над окремим об'єктом. Замість цього сервер змінює покажчик *p* у межах історії *snapshot*-станів, відновлює колекцію *objects* відповідно до *snapshot* $S^{(p)}$, зберігає новий стан метаданих у *meta*, а далі розсилає всім клієнтам повний *snapshot*. Отже, *undo/redo* у поточній системі реалізовано як операцію відновлення стану сцени, а не як повторне програвання окремих інверсних команд. Це робить поведінку системи стабільнішою після перезапуску сервера і спрощує синхронізацію всіх клієнтів після відкату історії.

Формально алгоритм переходу можна подати у вигляді:

$$S_{(t+1)} = F(S_t, op), \tag{2.12}$$

де F – серверна функція переходу.

Для *op* з множини Ω_h функція F визначається прямим застосуванням відповідної модифікації, а для *undo_request* і *redo_request* – вибором одного зі збережених *snapshot*-станів. Таким чином, конкурентність у системі вирішується централізовано: усі клієнтські зміни проходять через один упорядковуючий вузол, який перетворює частково впорядкований потік запитів від різних користувачів на єдину лінійну послідовність комітів.

2.4 Формальний аналіз коректності та ефективності алгоритму

У межах цього підрозділу обґрунтовується, що побудований алгоритм забезпечує правильне узгодження змін, а також оцінюються його обчислювальні характеристики.

Теорема 2.1 Для будь-якої множини клієнтів, що взаємодіють із єдиним сервером координування, якщо всі журнальовані операції застосовуються сервером у деякому загальному порядку, а після підключення або відновлення з'єднання клієнт отримує актуальний *snapshot* сцени, то за відсутності нових змін усі клієнти зрештою переходять до однакового стану сцени.

Доведення. У поточній системі всі операції з множини Ω_h застосовуються через сервер. Сервер змінює канонічний *snapshot* сцени, зберігає його та розсилає відповідні повідомлення клієнтам. Для операцій *undo* та *redo* сервер надсилає повний *snapshot*, який однозначно задає актуальний стан сцени. Отже, джерелом істини є централізований серверний стан. Коли передача повідомлень завершується, усі клієнти або вже отримали потрібні інкрементальні повідомлення, або при повторному з'єднанні одержують повний *snapshot*. В обох випадках локальні стани клієнтів стають тотожними одному й тому самому серверному *snapshot*. Теорему доведено.

Теорема 2.2 Для операцій із множини Ω_h алгоритм зберігає причинний порядок: якщо операція a причинно передує операції b, то сервер ніколи не зафіксує b раніше за a.

Доведення. Якщо операції a і b породжені тим самим клієнтом і b створена після a, то обидві операції передаються через одне WebSocket-з'єднання. Оскільки WebSocket працює поверх TCP, порядок повідомлень у межах одного з'єднання зберігається, а отже сервер не може отримати b раніше за a. Якщо ж операція b породжена деяким клієнтом лише після того, як цей клієнт спостерігав ефект операції a, то це означає, що сервер уже прийняв і застосував a, а потім розіслав її результат клієнтам у вигляді інкрементального повідомлення або повного *snapshot*. Лише після включення ефекту a у свій локальний контекст клієнт може породити b. Отже, у серверному порядку a уже належить префіксу журналу до моменту надходження b. У обох випадках серверна серіалізація не руйнує причинної залежності. Теорему доведено.

Слід окремо наголосити, що наведені теореми стосуються лише реалізованої множини операцій. Якщо до узагальненої моделі включати *reparent*,

то коректність для цієї операції треба аналізувати окремо. У поточному прототипі така операція не реалізована, тому включати її до строго доведених властивостей алгоритму було б методично неправильно.

Нехай n – кількість об’єктів у сцені, а m – кількість підключених клієнтів. Для операцій *create, delete, transform, color, rename* та *import_model* сервер виконує обмежену кількість дій над одним логічним об’єктом у *objects*, один запис до *operations* та одну широкомовну розсилку. Саме застосування операції до поточного стану можна оцінити як $O_{(1)}$ відносно числа об’єктів сцени, якщо не враховувати мережеву розсилку. Широкомовне розсилання має складність $O_{(m)}$.

Однак поточна реалізація історії побудована на *snapshot*-підході, а тому після кожної журнальованої операції сервер формує новий *snapshot* усієї сцени й додає його до *history*. Побудова *snapshot* потребує обходу всіх об’єктів сцени, тобто має складність $O_{(n)}$. Тому повна вартість комітної операції в поточному прототипі оцінюється як $O_{(n + m)}$. *Undo* та *redo* також мають складність не нижчу за $O_{(n + m)}$, оскільки потрібно відновити *snapshot*, синхронізувати колекцію *objects* зі станом історії та розіслати повний *snapshot* усім клієнтам.

Попри це, у практичному аспекті обраний підхід є виправданим. По-перше, він значно спрощує реалізацію стабільного *undo/redo* після рестарту сервера. По-друге, він робить відновлення клієнтів після перепідключення прозорим: достатньо надіслати актуальний *snapshot*. По-третє, навантаження при безперервних *drag*-операціях зменшується завдяки виділенню *transform_preview* у окремий клас повідомлень, які не записуються в журнал і не породжують новий *snapshot* історії.

2.5 Висновки до розділу 2

У розділі було побудовано формальну модель тривимірної сцени, що відповідає поточній архітектурі системи Collab3D. Показано, що узгоджуваною

одиницею в реалізованому прототипі є верхньорівневий об’єкт сцени з унікальним ідентифікатором, типом, трансформаційним станом, кольором, назвою та, за потреби, посиланням на зовнішню 3D-модель. На відміну від узагальненої теоретичної моделі ієрархічної сцени, операція *reparent* у поточній реалізації відсутня, тому її коректно розглядати лише як напрям подальшого розширення системи, а не як уже підтримувану функцію.

Також було формалізовано журнал операцій, службові команди та механізм синхронізації. Встановлено, що поточна система використовує централізовану серіалізацію змін на сервері, поєднуючи журнал операцій із *snapshot*-поданням стану сцени. *Undo/redo* реалізовано через історію *snapshot*-станів і покажчик активного стану, що забезпечує стабільне відновлення як під час звичайної роботи, так і після перепідключення клієнтів.

Формальний аналіз показав, що для реалізованої множини операцій алгоритм забезпечує конвергентність і збереження причинного порядку, а також має прийнятні часові характеристики для дослідницького колаборативного веб-редактора 3D-сцен. Отримані результати створюють теоретичну основу для переходу до наступного розділу, у якому буде розглянуто проектування та програмну реалізацію протоколу взаємодії клієнта і сервера, структури даних і підсистем збереження стану сцени.

3 ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ КОЛАБОРАТИВНОЇ СИСТЕМИ РЕДАГУВАННЯ 3D-СЦЕН

3.1 Архітектура системи

3.1.1 Загальна концепція архітектури

Реалізація колаборативного редактора тривимірних сцен Collab3D спирається на централізовану клієнт-серверну архітектуру, у межах якої єдиним джерелом істини щодо поточного стану сцени виступає серверна підсистема. Такий підхід безпосередньо відповідає формальній моделі, розглянутій у розділі 2, оскільки дозволяє впорядкувати конкурентні операції, уніфікувати їх застосування та забезпечити збіжність станів у всіх клієнтів після завершення обміну повідомленнями.

На практичному рівні система складається з браузерного клієнта, серверного застосунку на Node.js та підсистеми збереження на базі MongoDB. Сервер об’єднує HTTP- та WebSocket-функціональність на одному мережевому вузлі. Через HTTP-контур здійснюється видача клієнтських статичних файлів, перевірка працездатності сервісу та завантаження зовнішніх 3D-моделей. Через WebSocket-контур виконується передача операцій редагування в реальному часі, розсилання змін усім підключеним клієнтам, початкова синхронізація нових сесій та підтримка механізму *undo/redo*.

Обрана архітектура є доцільною для дослідницького прототипу 3D-редактора, оскільки вона поєднує відносну простоту реалізації з достатнім рівнем керованості системним станом. На відміну від децентралізованих моделей, у яких конфлікти доводиться розв’язувати на кожному вузлі окремо, у Collab3D інтеграція змін виконується централізовано. Це дозволяє зберегти однозначний порядок застосування операцій типу *create*, *delete*, *transform*, *color*,

rename та *import_model*, а також спростити відновлення актуального стану сцени після повторного підключення клієнта.

3.1.2 Логічна структура системи

Логічно систему можна подати як сукупність трьох взаємопов’язаних компонентів: клієнтської підсистеми, серверної підсистеми синхронізації та підсистеми збереження даних.

Клієнтська підсистема відповідає за побудову та візуалізацію тривимірної сцени у браузері, обробку дій користувача, формування повідомлень про зміни й локальне відображення операцій, отриманих від сервера. Вона включає сцену Three.js, камеру, механізми вибору об’єктів, інструменти трансформації, графічний інтерфейс редактора та модуль синхронізації WebSocket.

Серверна підсистема виконує роль координатора змін. Вона приймає операції від клієнтів, нормалізує структуру даних, застосовує модифікації до поточного стану сцени, журналює завершені операції, підтримує історію *snapshot*-станів для *undo/redo* та розсилає результати іншим учасникам сеансу. У поточній реалізації цей компонент є монолітним сервісом, що одночасно забезпечує HTTP-маршрути й WebSocket-обробку.

Підсистема збереження даних зберігає не лише журнал операцій, а й кілька різних видів структур. Колекція *objects* містить актуальний *snapshot* сцени, тобто поточний канонічний стан усіх об’єктів. Колекція *operations* накопичує журнал застосованих сервером операцій. Колекція *meta* використовується для збереження історії *snapshot*-станів і покажчика активної позиції в історії, що робить механізм *undo/redo* стабільним навіть після перезапуску сервера. Окремо через GridFS зберігаються двійкові файли моделей форматів GLB та GLTF, що імпортуються до сцени.

3.1.3 Архітектурна модель взаємодії

Архітектурна взаємодія компонентів має подієвий характер. На рівні користувача процес виглядає як послідовність «дія – повідомлення – інтеграція – розсилання – локальне застосування». Після виконання певної дії, наприклад додавання примітива, переміщення об’єкта, зміни кольору чи перейменування, клієнт формує структуроване JSON-повідомлення й надсилає його серверу через WebSocket. Сервер, отримавши таке повідомлення, визначає його тип, застосовує відповідну зміну до канонічного стану сцени, за потреби оновлює історію станів і розсилає узгоджений результат усім активним клієнтам, включно з ініціатором.

Взаємодію доцільно розділити на два класи. Перший клас становлять комітні операції, які змінюють логічний стан сцени та мають бути збережені. До них належать *create*, *delete*, *transform*, *color*, *rename* та *import_model*. Другий клас утворюють допоміжні або ефемерні повідомлення, зокрема *transform_preview*, *snapshot*, *client_ping* та *client_pong*. Такі повідомлення або не змінюють канонічний стан безпосередньо, або використовуються для технічної підтримки з’єднання та повторної синхронізації.

Окремий контур взаємодії реалізується через HTTP. Через маршрут *POST /upload-model* клієнт передає файл моделі на сервер, після чого сервер зберігає його в GridFS і повертає *fileId* та URL доступу. Через маршрут *GET /models/:id* збережений файл моделі може бути отриманий клієнтом для фактичного завантаження у Three.js. Через маршрут *GET /health* виконується перевірка доступності HTTP-частини сервісу, що використовується клієнтом у механізмі моніторингу стану з’єднання.

3.1.4 Модель станів клієнта і сервера

Для коректного опису практичної реалізації доцільно окремо виділити серверний і клієнтський стани системи. На клієнті підтримується не журнал у

строгому формальному розумінні, а сукупність локальних структур, необхідних для візуалізації, взаємодії та синхронізації. У спрощеному вигляді клієнтський стан можна подати як

$$ClientState = (LocalScene, Selection, UiMode, SyncState), \quad (3.1)$$

де *LocalScene* – локальне представлення об’єктів сцени у вигляді словника *objects* та відповідних вузлів Three.js; *Selection* – інформація про поточно вибраний об’єкт; *UiMode* – активний режим редагування, зокрема переміщення, обертання або масштабування; *SyncState* – поточний стан мережевої взаємодії, наприклад *connecting*, *online*, *reconnecting* або *offline*.

Дана поточна реалізація не підтримує повноцінну локальну чергу непідтверджених операцій, яка повторно відправляється після *reconnect*. Якщо WebSocket-з’єднання відсутнє, функція надсилання операцій не виконує *socket.send*, а клієнт переходить до стану *reconnecting* або *offline*. Після повторного встановлення з’єднання актуальний стан сцени відновлюється не через *replay* локальної черги, а через отримання нового *snapshot* від сервера.

Серверний стан у спрощеному вигляді можна подати як

$$ServerState = (Snapshot, Log, History, Pointer, Clients), \quad (3.2)$$

де *Snapshot* – поточний канонічний стан сцени, що фактично відповідає колекції *objects*;

Log – журнал завершених операцій у колекції *operations*;

History – послідовність *snapshot*-станів для *undo/redo*;

Pointer – покажчик активного стану в історії;

Clients – множина активних WebSocket-з’єднань.

Саме така модель точно відображає реальну логіку системи й є ближчою до реалізованого коду, ніж попередні абстрактні схеми з локальним журналом на клієнті.

3.1.5 Формат обміну повідомленнями

У системі Collab3D обмін даними між клієнтом і сервером здійснюється у форматі JSON-повідомлень. Ключовими полями повідомлення є *type*, *objId* та *data*. Поле *type* задає семантику операції або службової команди, *objId* ідентифікує об’єкт сцени, а *data* містить конкретні параметри зміни.

Для операції *create* структура *data* включає тип примітива, колір і параметри трансформації: *position*, *rotation* та *scale*. Для *import_model* до цього набору додаються *fileId* і *url*, а також за потреби *customName*. Для *transform* передається пара станів *before* та *after*, що дозволяє коректно фіксувати завершену трансформацію. Для *color* і *rename* передаються попереднє та нове значення відповідної властивості. Для *delete* клієнт також передає достатній набір даних про об’єкт, щоб сервер міг коректно підтримувати історію й подальше відновлення стану через *undo/redo*.

Окрему роль відіграє повідомлення *snapshot*, що містить не окрему операцію, а повний зріз поточного стану сцени. Воно використовується під час початкової синхронізації нового клієнта та після виконання *undo* або *redo*, коли потрібно гарантовано вирівняти стан усіх підключених сеансів. Повідомлення *transform_preview*, навпаки, не призначене для збереження в історії; воно передає поточний проміжний стан об’єкта під час *drag*-взаємодії й використовується лише для плавного відображення змін у реальному часі.

Важливо наголосити, що у фактичному протоколі не використовуються узагальнені типи *add*, *remove* або *reparent*, а також не передаються поля *author* чи *baseVersion*. Натомість набір повідомлень відображає реальні потреби прототипу й безпосередньо збігається з операціями, підтриманими на сервері та клієнті.

3.1.6 Механізм відновлення після втрати з’єднання

Механізм відновлення після втрати з’єднання у поточному прототипі побудовано на повторному підключенні клієнта та повторному отриманні актуального *snapshot* стану сцени. Коли WebSocket-з’єднання закривається або переходить у помилковий стан, клієнт змінює власний *SyncState*, зупиняє цикл технічних *ping*-повідомлень і планує повторне підключення із зростаючою затримкою до встановленої верхньої межі. Після успішного відкриття нового WebSocket-з’єднання сервер, як і для будь-якого нового клієнта, одразу надсилає повний *snapshot* сцени. Це дозволяє відновити консистентний локальний стан без відтворення повної історії операцій.

Для контролю працездатності з’єднання застосовано два взаємодоповнювальні механізми. На сервері працює *heartbeat* на базі *ping/pong* WebSocket-рівня: якщо клієнт не відповідає на кілька послідовних перевірок, з’єднання примусово завершується. На клієнті реалізовано періодичне надсилання *client_ping* і паралельну HTTP-перевірку маршруту */health*, що дає змогу відстежувати не лише стан WebSocket, а й доступність HTTP-частини сервера. Отримана інформація відображається в індикаторі синхронізації та повідомленні про автономний режим.

У реалізованій системі наразі немає повноцінного офлайн-режиму з чергою локальних операцій, повторним надсиланням непідтверджених змін і конфлікт-резолюцією після *reconnect*. Реалізовано саме автоматичне перепідключення, повторне отримання актуального *snapshot* та синхронізацію клієнтського стану із сервером.

3.1.7 Забезпечення масштабованості та стійкості

Питання масштабованості та стійкості в Collab3D розв’язуються насамперед на рівні вибору архітектури та формату обміну даними. Використання платформи Node.js із неблокуючою моделлю вводу-виводу

дозволяє ефективно підтримувати одночасні WebSocket-з’єднання кількох клієнтів без блокування основного потоку сервера. Передавання не повного стану сцени, а окремих предметних операцій також зменшує мережеве навантаження при звичайному редагуванні. Для безперервних трансформацій додатково введено повідомлення *transform_preview*, яке не фіксується у сховищі та не породжує нові *snapshot*-стани історії.

Водночас поточна реалізація є одновузловим прототипом. Сервер одночасно виконує HTTP-обслуговування, WebSocket-координацію та безпосередню роботу з MongoDB. Це є цілком виправданим для магістерської роботи, оскільки дозволяє продемонструвати повний цикл функціонування системи без надмірного архітектурного ускладнення. Проте горизонтальне масштабування в майбутньому вимагатиме зовнішнього механізму координації кількох серверних екземплярів, спільного доступу до стану сцени та узгодженого широкомовного розсилання повідомлень.

З погляду надійності окрему роль відіграє розділення поточного стану, журналу операцій, історії *undo/redo* та сховища двійкових моделей. Такий розподіл відповідальності зменшує ризик логічних помилок, полегшує відновлення системи після збоїв і створює основу для подальшого розвитку редактора.

3.2 Технологічний стек та обґрунтування вибору інструментів

3.2.1 Загальні вимоги до технологічної платформи

Вибір технологічної платформи для реалізації Collab3D визначався сукупністю функціональних і нефункціональних вимог. Система повинна забезпечувати інтерактивну візуалізацію тривимірної сцени у браузері, редагування об’єктів у реальному часі, синхронізацію кількох клієнтів через мережу, збереження поточного стану й історії змін, а також імпорт зовнішніх 3D-

моделей. Крім того, важливими були кросплатформний режим, відсутність потреби у встановленні спеціалізованого клієнтського ПЗ та можливість розгортання у звичайному веб-середовищі.

З огляду на це було обрано веб-орієнтований стек, у якому клієнтська частина реалізується засобами HTML, CSS та JavaScript із використанням бібліотеки Three.js, серверна частина – на платформі Node.js, обмін у реальному часі – через WebSocket, а підсистема збереження – на основі MongoDB та GridFS [5; 8; 13; 14; 16; 17]. Такий набір технологій забезпечує достатній рівень продуктивності для дослідницького редактора і водночас добре узгоджується з математичною моделлю та алгоритмом, описаними в розділі 2.

3.2.2 Клієнтська частина

Клієнтська частина системи побудована як односторінковий браузерний застосунок, що використовує сучасну модульну модель JavaScript. Базову структуру сторінки формує HTML-документ, у якому через *import map* підключається бібліотека Three.js та її допоміжні модулі. Стилізація й організація інтерфейсу виконуються засобами CSS, а вся динамічна логіка реалізується в модулі app.js.

Ключовою бібліотекою клієнта є Three.js. Вона забезпечує високорівневу об’єктну модель сцени поверх WebGL і надає готові абстракції для роботи з об’єктами, матеріалами, камерами, освітленням і трансформаціями [6; 8; 9]. У Collab3D використовуються як базові можливості бібліотеки, так і спеціалізовані допоміжні модулі: *OrbitControls* для керування камерою, *TransformControls* для обертання та масштабування об’єктів, а також *GLTFLoader* для завантаження моделей форматів GLB і GLTF [8; 10; 11].

Важливою перевагою обраного підходу є природне відображення програмних структур на формальну модель сцени. Об’єкти редагування представлені вузлами Object3D, примітиви створюються як *Mesh* із відповідною геометрією, а імпортовані моделі інкапсулюються в кореневий *Group*-вузол.

Локальне відображення стану сцени підтримується через словник *objects*, де кожен *objId* пов’язується з відповідним кореневим вузлом сцени. Це спрощує як вибір об’єктів, так і застосування синхронізованих трансформацій.

Окремо слід відзначити склад і функції користувацького інтерфейсу. У верхній частині сторінки реалізовано меню «Файл», «Додати» та «Редагувати». Через нього користувач може створювати нову сцену, зберігати та відкривати її, додавати примітиви або імпортувати зовнішні моделі, а також виконувати операції *undo*, *redo*, клонування, зміни кольору й видалення вибраного об’єкта. Права панель містить згортні секції «Сцена», «Список об’єктів» і «Об’єкт». У цих секціях розміщено перемикачі інерції камери, сітки, світових осей і *mini-axis*, керування режимом простору трансформацій, інструменти вибору режимів переміщення, обертання та масштабування, а також інспектор властивостей із числовими полями для позиції, обертання, масштабу, назви, UUID та псевдоніма об’єкта.

Інтерфейс містить і додаткові продуктивні механізми: кольорове маркування об’єктів у списку, гарячі клавіші T/R/S для перемикання режимів, окремі чекбокси для керування видимістю *gizmo*, інтерактивний *mini-axis* для швидкого переходу до ортогональних напрямків погляду, а також індикатор синхронізації з повідомленням про автономний режим. Таким чином, клієнтська частина реалізує не лише візуалізацію сцени, а й повноцінне інструментальне середовище редагування.

3.2.3 Серверна частина

Серверна частина реалізована на платформі Node.js і використовує модульну ESM-модель JavaScript [13; 21]. Як HTTP-фреймворк обрано Express, який забезпечує побудову маршрутизації, видачу статичних файлів клієнта та обслуговування допоміжних REST-подібних маршрутів [14; 15]. Для приймання завантажуваних файлів застосовується бібліотека *multer* із конфігурацією *memoryStorage*, що дозволяє приймати GLB/GLTF-моделі безпосередньо в

пам’ять і передавати їх до GridFS. Реальний час взаємодії реалізується через бібліотеку *ws*, яка надає WebSocketServer і механізми роботи з окремими з’єднаннями [5; 20].

Практично сервер виконує кілька функціональних ролей одночасно. По-перше, він є HTTP-сервером і віддає клієнтський застосунок. По-друге, він виступає координатором операцій редагування та реалізує централізовану серіалізацію змін. По-третє, він забезпечує технічний контроль стану з’єднання через *heartbeat*-механізм і сервісний маршрут *health-check*.

Під час отримання WebSocket-повідомлення сервер класифікує його за типом. Для *transform_preview* виконується лише широкомовне розсилання без запису до історії. Для *undo_request* і *redo_request* відбувається переміщення покажчика історії, відновлення відповідного *snapshot* і подальше розсилання повного стану сцени. Для предметних операцій *create*, *delete*, *transform*, *color*, *rename* та *import_model* сервер виконує нормалізацію даних, оновлює поточний *snapshot* сцени, записує операцію до журналу і синхронізує історію *snapshot*-станів. Завдяки цьому Node.js-сервер виступає прямим втіленням алгоритму узгодження, розглянутого в розділі 2.

3.2.4 Система збереження даних

Підсистема збереження даних у Collab3D реалізована на основі MongoDB – документно-орієнтованої СУБД, що природно працює з JSON-подібними структурами й добре підходить для вебсистем із гнучкою схемою даних. Проте на практиці MongoDB у цьому проекті використовується не лише як сховище журналу операцій, а як багатокомпонентне середовище зберігання різних аспектів системного стану [18; 22; 23; 24].

Колекція *objects* виконує роль канонічного *snapshot*-сховища сцени. У кожному її документі зберігаються ідентифікатор об’єкта, тип, позиція, обертання, масштаб, колір, користувацька назва, а для моделей також *fileId* і *url*.

Саме ця колекція є базою для побудови актуального *snapshot*, що надсилається клієнтові при підключенні та після операцій *undo/redo*.

Колекція *operations* містить журнал застосованих сервером предметних операцій. Кожен запис відображає факт зміни канонічного стану сцени та включає тип операції, ідентифікатор об’єкта, корисне навантаження й призначену сервером версію. На відміну від неї, колекція *meta* використовується для службових метаданих системи. У поточній реалізації в ній зберігається документ *undoRedo*, що містить *history*, *pointer* і *timestamp* останнього оновлення. Завдяки цьому стан історії *undo/redo* не втрачається після перезапуску сервера.

Окремим компонентом є GridFS *bucket* з іменем *models*. Через нього зберігаються файли GLB та GLTF, що були імпортовані користувачами до сцени [16; 17]. Такий підхід є суттєво сильнішим за зберігання лише зовнішніх URL, оскільки дозволяє зробити моделі частиною серверно контрольованої інфраструктури застосунку. Після завантаження модель отримує *fileId*, на основі якого генерується маршрут доступу */models/:id*. Клієнт, одержавши повідомлення *import_model* або *snapshot* із даними моделі, може завантажити відповідний ресурс і відтворити його у власній локальній сцені.

Таким чином, MongoDB у Collab3D забезпечує одразу чотири різні функції: збереження поточного стану сцени, журналювання операцій, підтримку історії *undo/redo* та довготривале зберігання двійкових файлів моделей. Саме таке трактування підсистеми збереження є коректним відображенням реальної реалізації.

3.2.5 Архітектурна взаємодія компонентів

Загальна взаємодія компонентів технологічного стеку має вигляд багатоконтурної схеми. Браузерний клієнт, побудований на Three.js, працює з двома мережевими каналами. Через HTTP він завантажує початкову HTML-сторінку, файл стилів, модуль app.js, виконує *health-check*-запити та передає файли моделей на сервер. Через WebSocket він отримує *snapshot* стану сцени,

надсилає операції редагування та приймає широкомовно розіслані зміни інших клієнтів.

На серверному боці ці канали поєднано в межах одного застосунку. Express-частина обслуговує HTTP-маршрути, тоді як *WebSocketServer* використовує той самий HTTP-сервер як транспортну основу. Після інтеграції операції сервер, залежно від її типу, або модифікує MongoDB-стан і розсилає результат, або виконує лише технічні дії з підтримки сесії та синхронізації.

Підсистема збереження не взаємодіє з клієнтом безпосередньо. Усі операції доступу до неї інкапсульовані сервером. Це спрощує безпекову модель, уніфікує правила доступу до даних і зменшує кількість зовнішніх залежностей у браузерному коді.

3.2.6 Обґрунтування вибору веб-архітектури

Веб-орієнтована архітектура була обрана не лише з технічних, а й із прикладних міркувань. По-перше, вона забезпечує кросплатформність: редактор може працювати на настільних комп’ютерах, ноутбуках і мобільних пристроях за наявності сучасного браузера. По-друге, користувачеві не потрібно встановлювати спеціалізоване клієнтське програмне забезпечення, що істотно спрощує демонстрацію системи, тестування й подальше розгортання.

По-третє, веб-архітектура добре поєднується з моделлю централізованої синхронізації. Сервер легко бере на себе функції джерела істини, а клієнти залишаються відносно легкими в реалізації. По-четверте, такий підхід забезпечує прозорий шлях до хмарного розгортання: сервер може бути опублікований як звичайний веб-сервіс, а підсистема збереження – перенесена до віддаленої бази даних без суттєвої зміни клієнтської логіки.

Для магістерської роботи цей вибір є особливо вдалим, оскільки дозволяє продемонструвати повний цикл функціонування колаборативного редактора у середовищі, наближеному до реального використання, і водночас не потребує переходу до важчих десктопних платформ або спеціалізованих CAD-рушіїв.

3.2.7 Відповідність теоретичній моделі

Обраний технологічний стек має пряму відповідність теоретичній моделі, побудованій у розділі 2. По-перше, Three.js надає сцену та вузли Object3D, які слугують програмною основою для представлення множини об’єктів сцени та їх трансформаційного стану. По-друге, WebSocket-канал реалізує механізм передачі операцій у реальному часі, а Node.js-сервер – функцію централізованої серіалізації та інтеграції змін.

По-третє, MongoDB-структури *objects*, *operations* і *meta* є практичним відображенням відповідно *snapshot*-подання, журналу операцій та історії станів із покажчиком. По-четверте, GridFS розширює модель сцени підтримкою зовнішніх модельних ресурсів, що було передбачено у формальному описі об’єкта через атрибут ресурсу. Нарешті, відсутність у коді операції *reparent* також узгоджується з виправленим теоретичним описом: ієрархічна модель сцени розглядається як узагальнення, тоді як поточний прототип синхронізує саме верхньорівневі об’єкти.

Отже, технологічний стек Collab3D не є випадковим набором інструментів, а виступає безпосереднім практичним втіленням формальної моделі та алгоритму узгодження змін, запропонованих у попередньому розділі.

3.3 Протокол взаємодії клієнта і сервера та структура даних

Після визначення архітектури системи та обраного технологічного стека необхідно формалізувати протокол взаємодії між клієнтською і серверною підсистемами, а також структури даних, що використовуються для представлення стану сцени, синхронізації змін, збереження історії та роботи з зовнішніми 3D-моделями. У системі Collab3D взаємодія побудована як поєднання двох каналів обміну: протокол HTTP використовується для завантаження клієнтських ресурсів, перевірки доступності сервера та

передавання файлів моделей, тоді як WebSocket забезпечує низьколатентний обмін операціями редагування в режимі реального часу. Окремо на клієнті реалізовано формат локального файла сцени, що використовується для експорту та імпорту робочого стану редактора.

На відміну від абстрактних схем, у яких усі зміни подаються як універсальні події одного типу, поточний протокол є предметно орієнтованим. Для кожного класу змін виділено окремий тип повідомлення: створення примітива, імпорт зовнішньої моделі, видалення об’єкта, коміт трансформації, попередній перегляд трансформації, зміна кольору, перейменування, службові запити *undo/redo*, а також службові повідомлення початкової синхронізації та контролю доступності з’єднання. Такий підхід зменшує кількість неоднозначних інтерпретацій, спрощує серверну обробку й дозволяє безпосередньо пов’язати формат повідомлення з конкретною дією інтерфейсу.

3.3.1 Формат представлення об’єктів сцени

Базовою одиницею обміну в системі є верхньорівневий об’єкт сцени, який має стабільний колаборативний ідентифікатор `*objId*`. На клієнті цей ідентифікатор записується до `userData.id` кореневого вузла `*Object3D*`, а на сервері використовується як ключ запису в колекції поточного стану `*objects*`. Для примітивів та імпортованих моделей застосовується єдина логіка адресації: усі операції `*create*`, `*delete*`, `*transform*`, `*color*`, `*rename*` і `*import_model*` звертаються саме до такого кореневого вузла, що робить семантику повідомлень інваріантною щодо внутрішньої структури GLTF/GLB-моделі.

У реалізованій системі підтримуються п’ять основних типів об’єктів: `*cube*`, `*sphere*`, `*rhombus*`, `*triangle*` та `*model*`. Для геометричних примітивів достатньо зберігати тип, просторовий стан, колір і користувацьку назву. Для зовнішніх моделей до цього набору додаються `*fileId*` і `*url*`, які вказують на файл, збережений у GridFS, та маршрут доступу до нього. Просторовий стан у всіх випадках описується трьома групами параметрів: `*position*`, `*rotation*` і

`*scale*`, кожна з яких містить координати `x`, `y` та `z`. Така структура є спільною як для серверного *snapshot*-подання, так і для клієнтського JSON-формату файлу сцени.

Серверний *snapshot* поточного стану будується як асоціативне відображення, у якому ключем є `*objId*`, а значенням – нормалізований запис про об’єкт. Саме цей формат використовується під час початкової синхронізації, відновлення після перепідключення та розсилання повного стану сцени після операцій *undo/redo* (див. рис. 3.1, 3.2).

```
      snapshot[objId] = {
        type: "cube" | "sphere" | "rhombus" | "triangle" | "model",
        fileId: string,
        url: string,
        position: { x: number, y: number, z: number },
        rotation: { x: number, y: number, z: number },
        scale:    { x: number, y: number, z: number },
        color: string,
        customName: string
      }
```

Рисунок 3.1 – Узагальнений вигляд одного елемента *snapshot*

```
/**
 * buildSnapshot()
 * Збирає поточний стан сцени з objects.
 */
async function buildSnapshot() {
  const docs = await objectsCollection.find({}).toArray();
  const snapshot = {};

  for (const obj of docs) {
    snapshot[obj.id] = {
      type: obj.type,
      fileId: obj.fileId || "",
      position: obj.position || { x: 0, y: 0, z: 0 },
      rotation: obj.rotation || { x: 0, y: 0, z: 0 },
      scale: obj.scale || { x: 1, y: 1, z: 1 },
      color: obj.color || "#ffffff",
      customName: normalizeCustomName(obj.customName),
      url: obj.url || (obj.fileId ? `/models/${obj.fileId}` : "")
    };
  }
  return snapshot;
}
```

Рисунок 3.2 – Реалізація функції *buildSnapshot* в коді проекту

Клієнтська підсистема використовує споріднене, але не тотожне представлення. У пам’яті браузера таблиця `*objects*` відображає `*objId*` у кореневий `Object3D`; для моделей додатково зберігаються службові поля `*isModel*`, `*fileId*`, `*url*` та `*tintColor*`. При отриманні `*snapshot*` або окремих операцій клієнт або створює новий примітив, або забезпечує наявність плейсхолдера `*Group*` для моделі й далі підвантажує реальний вміст через `*GLTFLoader*`. Завдяки цьому транспортний рівень не залежить від геометричної складності моделі, а мережеві повідомлення завжди мають компактний і сталий формат.

Окремо в системі реалізовано формат локального файла сцени, який використовується командами «Зберегти сцену» та «Відкрити сцену». На відміну від серверного *snapshot*, цей формат призначений не для безпосередньої WebSocket-синхронізації, а для експорту й повторного завантаження робочої сцени через браузер. Файл містить службові метадані (`*format*`, `*version*`, `*savedAt*`, `*app*`), параметри подання сцени (`*sceneSettings*`) та список об’єктів (`*objects*`) у формі масиву (див. рис. 3.3, 3.4).

```
{
  format: "collab3d-scene",
  version: 1,
  savedAt: "2026-03-31T...Z",
  app: { name: "Collab3D", schema: 1 },
  sceneSettings: {
    inertia: boolean,
    showGrid: boolean,
    showAxes: boolean,
    showMiniAxis: boolean,
    transformSpace: "world" | "local"
  },
  objects: [ ... ]
}
```

Рисунок 3.3 – Узагальнений вигляд реалізації формату локального файла сцени

Таким чином, у системі фактично співіснують два взаємопов’язані формати представлення: асоціативний *snapshot* для мережевої синхронізації та

масив об’єктів для локального файлу сцени. Перший є оптимальним для адресного доступу за `*objId*` і швидкого відновлення стану, тоді як другий є зручним для серіалізації у переносимий JSON-документ.

```
function buildSceneFilePayload()
{
  const ids = Object.keys(objects).sort();
  return
  {
    format: "collab3d-scene",
    version: 1,
    savedAt: new Date().toISOString(),
    app: { name: "Collab3D", schema: 1 },
    sceneSettings: cloneSceneSettingsForFile(),
    objects: ids.map((id) =>
getSerializableObjectData(objects[id])).filter(Boolean)
  };
}
```

Рисунок 3.4 – Реалізація функції *buildSceneFilePayload* в коді проекту

3.3.2 Формат і семантика WebSocket-повідомлень

WebSocket-канал є основним механізмом синхронізації змін між клієнтами. Клієнт формує адресу `WS_URL` на основі поточного джерела документа: при роботі через HTTPS використовується `*wss://*`, а в локальному середовищі – `*ws://*`. Кожне повідомлення передається як окремий JSON-об’єкт, серіалізований функцією `*JSON.stringify*`. Це дає змогу підтримувати єдину точку формування вихідних операцій і спрощує трасування протоколу під час налагодження.

Узагальнений формат більшості предметних повідомлень можна подати як трійку `⟨*type, objId, data*⟩`, де `type` задає клас операції, `*objId*` ідентифікує об’єкт сцени, а `data` містить параметри зміни. Для службових запитів, що не прив’язані до конкретного об’єкта, наприклад `*undo_request*`, `*redo_request*` або `*client_ping*`, поле `*objId*` не використовується. Сервер, отримавши журнальовану операцію, додає до неї поле `*version*`, після чого зберігає операцію в журналі та широкомовно розсилає її всім клієнтам, включно з автором.

Предметні WebSocket-повідомлення в поточній реалізації мають таку семантику. Повідомлення `*create*` створює примітив заданого типу і передає його початкові параметри. Повідомлення `*import_model*` оголошує появу нового кореневого вузла зовнішньої моделі та передає `*fileId*`, `*url*`, колір і початковий трансформаційний стан. Повідомлення `*delete*` видаляє об'єкт зі сцени. Повідомлення `*transform*` фіксує завершену зміну положення, обертання або масштабу й містить структуру `*before/after*`, що дозволяє зберігати завершений коміт трансформації у зрозумілій формі. Повідомлення `*color*` і `*rename*` передають відповідно попереднє і нове значення кольору або назви.

Окремий клас становить повідомлення `*transform_preview*`. Воно надсилається під час безперервного переміщення, обертання або масштабування об'єкта, коли користувач ще не завершив жест редагування. На відміну від `*transform*`, таке повідомлення не записується до журналу і не потрапляє до історії *undo/redo*. Його призначення полягає у забезпеченні візуально плавного оновлення віддалених клієнтів під час інтерактивної дії, без засмічення історії десятками проміжних станів.

Крім предметних повідомлень, протокол містить службові об'єкти. Повідомлення `*snapshot*` передає повний актуальний стан сцени. Воно використовується під час початкового підключення клієнта, а також після *undo/redo*, коли сервер навмисно розсилає повний знімок стану для вирівнювання всіх клієнтських копій. Повідомлення `*client_ping*` і `*client_pong*` використовуються для прикладного контролю доступності каналу: клієнт періодично надсилає час у полі `*ts*`, а сервер відповідає `*client_pong*`, додаючи власну часову мітку `*serverTs*`. Паралельно із цим сервер використовує також стандартний механізм `*ws.ping/ws.pong*` для мережевого *heartbeat*-контролю.

Створення примітива передається у вигляді повідомлення (див. рис. 3.5, 3.6), а збереження всіх трансформацій над об'єктом можна представити як на рис. 3.7 – 3.8.

```
      {
        type: "create",
        objId: "obj_1743410000000",
        data: {
          type: "cube",
          position: { x: 0, y: 0.5, z: 0 },
          rotation: { x: 0, y: 0, z: 0 },
          scale: { x: 1, y: 1, z: 1 },
          color: "#ffffff"
        }
      }
```

Рисунок 3.5 – Узагальнений вигляд створення примітива

```
function sendCreatePrimitive(type) {
  const objId = `obj_${Date.now()}`;
  sendOperation({
    type: "create",
    objId,
    data: {
      type,
      position: { x: 0, y: 0.5, z: 0 },
      rotation: { x: 0, y: 0, z: 0 },
      scale: { x: 1, y: 1, z: 1 },
      color: colorPicker.value
    }
  });
}
```

Рисунок 3.6 – Реалізація функції *sendCreatePrimitive* в коді проекту

```
      {
        type: "transform",
        objId: "obj_1743410000000",
        data: {
          before: { position: {...}, rotation: {...}, scale: {...} },
          after:  { position: {...}, rotation: {...}, scale: {...} }
        }
      }
```

Рисунок 3.7 – Узагальнений вигляд коміту завершеної трансформації

```
/** applyTransform() */
function applyTransform(obj, st) {
    / * Застосовує position/rotation/scale до об'єкта. */
if (!obj || !st) return;
  if (st.position) obj.position.set(st.position.x, st.position.y,
st.position.z);
  if (st.rotation) obj.rotation.set(st.rotation.x, st.rotation.y,
st.rotation.z);
  if (st.scale) obj.scale.set(st.scale.x, st.scale.y, st.scale.z);
}
```

Рисунок 3.8 – Реалізація функції *applyTransform* в коді проекту

А початкова синхронізація використовує повідомлення такого загального вигляду (див. рис. 3.9).

```
{  type: "snapshot",
  data: {
    "obj_...": { ... },
    "model_...": { ... }  }   }
```

Рисунок 3.9 – Узагальнений вигляд початкової синхронізації

Отже, WebSocket-протокол у системі Collab3D поєднує інкрементальні повідомлення для щоденних операцій редагування, ефемерні повідомлення попереднього перегляду та повні *snapshot*-повідомлення для вирівнювання глобального стану. Саме це поєднання забезпечує баланс між оперативністю взаємодії та керованістю історії змін.

3.3.3 Сценарій початкової синхронізації та відновлення стану

Початкова синхронізація у системі навмисно побудована не через повторне відтворення всього журналу операцій, а через безпосередню передачу повного актуального *snapshot*-стану. Після встановлення WebSocket-з'єднання сервер одразу викликає процедуру `*sendSnapshot(ws)*`, яка будує поточний стан сцени на основі колекції `*objects*` і відправляє клієнту повідомлення виду `*{type: "snapshot", data: snapshot}*`. Завдяки цьому новий учасник сеансу або клієнт після перепідключення не залежить від довжини журналу та відразу отримує узгоджений стан сцени (див. рис. 3.10).

```
/**
 * sendSnapshot(ws)
 * Відправляє snapshot одному клієнту.
 */
async function sendSnapshot(ws) {
  const snapshot = await buildSnapshot();
  ws.send(JSON.stringify({ type: "snapshot", data: snapshot }));
}
```

Рисунок 3.10 – Реалізація функції *sendSnapshot* в коді проекту

На клієнті обробка такого повідомлення виконується функцією *applySnapshot(snapshot)*. Алгоритм складається з трьох етапів. По-перше, видаляються всі локальні об’єкти, яких немає в отриманому *snapshot*, щоб усунути розсинхронізацію (див. рис. 3.11).

```
function applySnapshot(snapshot = {}) {
  /**
   * Обробляє snapshot від сервера (повний стан сцени).
   * Кроки:
   * 1) Видалити локальні об'єкти, яких немає в snapshot (щоб не було
розсинхрону).
   * 2) Для кожного елемента snapshot:
   *    - якщо модель: ensure placeholder, застосувати трансформацію,
завантажити URL, застосувати tint
   *    - якщо примітив: створити або оновити mesh + колір + трансформацію
   * 3) Оновити меню і gizmo (updateTransformControlsForMode).
   */
for (const id of Object.keys(objects)) {
    if (!snapshot[id]) removeObjectFromScene(id);
  }

  for (const [id, data] of Object.entries(snapshot)) {
    const existing = objects[id];

    if (data.type === "model") {
      const root = existing || ensureModelPlaceholder(id, data);
      root.userData.tintColor = data.color || root.userData.tintColor ||
"#ffffff";
      setObjectCustomName(root, data.customName || "");

      applyTransform(root, data);
      loadModelIntoPlaceholder(data.url, id);

      // якщо модель вже завантажена – tint одразу
      applyTintToObject(root, root.userData.tintColor);

      continue;
    }

    if (!existing) {
      const mesh = createPrimitiveMesh(data.type, data.color);
      addObjectToScene(id, mesh, data);
    } else {
      setObjectCustomName(existing, data.customName || "");
      applyTransform(existing, data);
      if (existing.isMesh && existing.material?.color) {
        existing.material.color.set(normalizeColor(data.color));
        existing.material.needsUpdate = true;
      }
    }
  }
```

Рисунок 3.11 – Реалізація функції *applySnapshot* в коді проекту

По-друге, для кожного запису зі *snapshot* виконується створення або оновлення відповідного об’єкта: для примітивів створюється `*Mesh*` потрібного типу або оновлюються його параметри; для моделей забезпечується наявність кореневого `*Group*`, застосовуються трансформації та колір, а далі через `*GLTFLoader*` підвантажується вміст за маршрутом `*url*`. По-третє, після завершення оновлення сцени синхронізуються список об’єктів, інспектор властивостей і стан `*TransformControls*`.

Відновлення після втрати з’єднання також базується на повторному отриманні серверного *snapshot*. Клієнт підтримує змінні стану `*syncState*`, `*lastSyncAt*` і механізми перевірки доступності: HTTP-маршрут `*/health*` періодично опитується функцією `*runHealthCheck()*`, а при відкритому WebSocket-з’єднанні кожні 25 секунд надсилається прикладний `*client_ping*`. У разі помилки або закриття сокета спрацьовує механізм `*scheduleSocketReconnect()*`, який запускає спробу перепідключення з наростаючою затримкою, обмеженою десятьма секундами.

Важливо підкреслити, що в поточному прототипі не реалізовано повноцінний офлайн-режим із чергою локальних непідтверджених операцій. Якщо в момент відсутності з’єднання користувач спробує сформувати команду, функція `*sendOperation()*` не поміщає її до локальної черги, а просто завершується без відправлення й переводить індикатор синхронізації в стан `*offline*` або `*reconnecting*`. Отже, коректніше говорити не про повторне надсилання накопичених змін, а про повторне встановлення з’єднання, повторне отримання актуального *snapshot* і повторне вирівнювання локальної сцени з серверним станом.

Серверна частина, у свою чергу, додатково контролює стан WebSocket-клієнтів за допомогою *heartbeat*-циклу. Через кожні 30 секунд сервер перевіряє, чи відповідає клієнт на `*ws.ping*`; якщо відсутні дві послідовні відповіді `pong`, з’єднання примусово завершується. Така комбінація серверного *heartbeat* і клієнтського `*client_ping*` дозволяє вчасно виявляти мережеві збої та

підтримувати для користувача прозорий сценарій відновлення стану (див. рис. 3.12).

```
const WS_HEARTBEAT_MS = 30000;

function wsHeartbeat() {
  this.isAlive = true;
  this.missedPongs = 0;
}
const wsHeartbeatInterval = setInterval(() => {
  for (const ws of wss.clients) {
    if (ws.readyState !== WebSocket.OPEN) continue;

    if (ws.isAlive === false) {
      ws.missedPongs = (ws.missedPongs || 0) + 1;
      if (ws.missedPongs >= WS_MAX_MISSED_PONGS) {
        console.warn(`WS heartbeat тайм-аут -> завершення
${formatWsClientLabel(ws)} missedPongs=${ws.missedPongs}`);
        try {
          ws.terminate();
        } catch (err) {
          console.error("помилка завершення WS:", err?.stack || err);
        }
        continue;
      }
    }
    ws.isAlive = false;

    try {
      ws.ping();
    } catch (err) {
      console.error(`WS помилка ping ${formatWsClientLabel(ws)}:`, err?.stack ||
err);
      try {
        ws.terminate();
      } catch (terminateErr) {
        console.error("WS помилка завершення:", terminateErr?.stack ||
terminateErr);
      }
    }
  }
}, WS_HEARTBEAT_MS);

wss.on("close", () => {
  clearInterval(wsHeartbeatInterval);
});

ws.on("pong", wsHeartbeat);
```

Рисунок 3.12 – Реалізація функції *wsHeartbeat* в коді проекту

Таким чином, механізм початкової синхронізації та відновлення стану в Collab3D є *snapshot*-орієнтованим. Він добре узгоджується з централізованою моделлю сервера як єдиного джерела істини і дає змогу уникнути складних процедур добору відсутніх операцій або інтеграції локально накопичених змін.

3.3.4 Механізм *undo/redo* і журналювання змін

У реалізованій системі механізм скасування та повторення дій побудований на поєднанні журналу операцій і історії *snapshot*-станів. На сервері для цього використовуються три окремі структури збереження. Колекція `*objects*` містить поточний канонічний стан сцени й фактично є серверним *snapshot*-поданням. Колекція `*operations*` накопичує журнал застосованих журнальованих операцій. Колекція `*meta*` зберігає метадані механізму *undo/redo*, а саме масив `*history*` і покажчик `*pointer*`, що визначає активний стан історії. Такий поділ ролей є принципово важливим: журнал дає хронологію дій, `*objects*` відбиває актуальний стан, а `*meta*` забезпечує стабільне відновлення попередніх станів навіть після перезапуску сервера.

До історично значущих у поточній реалізації належать лише шість типів операцій: `*create*`, `*delete*`, `*transform*`, `*color*`, `*rename*` та `*import_model*`. Після надходження такої операції сервер виконує процедуру `*applyOperation(op)*`. Вона присвоює полю `*version*` наступне значення, отримане з `*nextVersion()*`, нормалізує дані операції, оновлює запис у `*objects*`, додає саму операцію до `*operations*`, а потім широкомовно розсилає повідомлення всім клієнтам. Відразу після цього сервер будує новий *snapshot* сцени через `*buildSnapshot()*` і додає його до історії `*history*` процедурою `*pushCurrentSnapshotToHistory()*`. Якщо перед цим було виконано *undo* і покажчик історії не стояв у кінці масиву, права *redo*-гілка відкидається, що відповідає стандартній семантиці редакторів.

Операція `*transform_preview*` принципово не потрапляє ні до журналу `*operations*`, ні до `*history*`. Сервер лише ретранслює таке повідомлення іншим клієнтам без модифікації `*objects*`. Це рішення дозволяє істотно зменшити розмір історії та не перетворювати один жест перетягування на десятки дрібних кроків

undo/redo. Фактично в історії фіксується тільки завершений коміт `*transform*`, який містить поля `*before*` та `*after*` і є логічно завершеною зміною стану об’єкта.

Запити `*undo_request*` і `*redo_request*` не застосовуються як звичайні предметні операції. Після отримання `*undo_request*` сервер перевіряє, чи можливий відкат, зменшує значення `*pointer*`, виконує `*restoreSnapshot(history[pointer])*`, зберігає метадані в `*meta*` і розсилає всім клієнтам повний `*snapshot*`. Аналогічно, `*redo_request*` збільшує `*pointer*`, відновлює відповідний *snapshot* і знову надсилає повний знімок стану. Отже, реальна логіка *undo/redo* у прототипі є *snapshot*-орієнтованою, а не побудованою на послідовному програванні інверсних команд. Хоч у коді присутня допоміжна функція `*inverseOf(op)*`, у поточній версії вона не визначає головний механізм відновлення стану.

Історія скасування обмежена сталою `*HISTORY_LIMIT = 100*`, що запобігає неконтрольованому зростанню обсягу метаданих. Під час ініціалізації сервера функція `*loadUndoRedoMeta()*` читає документ `*undoRedo*` з колекції `*meta*`, а якщо історії ще немає, створює стартовий *snapshot* поточної сцени й ініціалізує `*pointer*` нулем. Завдяки цьому після рестарту сервера система може продовжувати роботу з уже наявною історією, а редагування не починається з повного скидання стану *undo/redo* (див. рис. 3.13).

```
const HISTORY_LIMIT = 100;  /* обмежуємо довжину історії, (можна оптимізувати,
зберігаючи тільки останні N або використовуючи інші структури даних)*/
/**
 * loadUndoRedoMeta()
 * Витягує history/pointer з MongoDB (meta).
 */
async function loadUndoRedoMeta() {
  try {
    const doc = await metaCollection.findOne({ _id: "undoRedo" });
    if (doc && Array.isArray(doc.history) && typeof doc.pointer === "number") {
      history = doc.history;
      pointer = doc.pointer;
    } else {
      history = [];
      pointer = -1;
    }
```

Рисунок 3.13 – Реалізація функції *loadUndoRedoMeta* в коді проекту

```
    // обрізаємо історію до HISTORY_LIMIT, якщо вона вже розрослася.
    let historyWasTrimmed = false;
    if (history.length > HISTORY_LIMIT) {
      const overflow = history.length - HISTORY_LIMIT;
      history = history.slice(-HISTORY_LIMIT);
      pointer = Math.max(0, pointer - overflow);
      historyWasTrimmed = true;
    }
    // якщо історія порожня – стартовий snapshot поточної сцени
    if (!history.length) {
      const snapshot = await buildSnapshot();
      history = [deepClone(snapshot)];
      pointer = 0;      await saveUndoRedoMeta();
      console.log("Undo/Redo meta ініціалізується початковим знімком");
      return;
    }

    // захист від виходу pointer за межі
    if (pointer < 0) pointer = 0;
    if (pointer >= history.length) pointer = history.length - 1;

    if (historyWasTrimmed) {
      await saveUndoRedoMeta();
      console.log(`Undo/Redo історія скорочена до ${history.length}`);
    }

    console.log("Undo/Redo завантажено meta знімок:", { history: history.length,
pointer });
  } catch (err) {
    console.error("loadUndoRedoMeta помилка:", err);
    const snapshot = await buildSnapshot();
    history = [deepClone(snapshot)];
    pointer = 0;
  }
}
/**
 * saveUndoRedoMeta()
 * Зберігає history/pointer у MongoDB (meta).
 */
async function saveUndoRedoMeta() {
  try {
    const trimmedHistory = history.length > HISTORY_LIMIT
      ? history.slice(-HISTORY_LIMIT)
      : history;

    if (trimmedHistory !== history) {
      const overflow = history.length - trimmedHistory.length;
      history = trimmedHistory;
      pointer = Math.max(0, pointer - overflow);
      if (pointer >= history.length) pointer = history.length - 1;
    }
```

Рисунок 3.13, аркуш 2

```
    await metaCollection.updateOne(
      { _id: "undoRedo" },
      { $set: { history, pointer, updatedAt: new Date() } },
      { upsert: true }
    );
  } catch (err) {
    console.error("saveUndoRedoMeta error:", err);
  }
}
```

Рисунок 3.13, аркуш 3

З практичного погляду описаний механізм має дві важливі переваги. По-перше, він забезпечує стійку й передбачувану поведінку редактора при складних сценаріях видалення, імпорту та відновлення об’єктів. По-друге, він спрощує синхронізацію кількох клієнтів, оскільки після *undo* або *redo* всі учасники гарантовано отримують один і той самий повний *snapshot*. Саме тому така реалізація є доцільною для дослідницького прототипу колаборативного редактора 3D-сцен, де пріоритетом є коректність і прозорість поведінки, а не мінімізація обсягу історії даних.

3.3.5 Збереження й відкриття сцени, імпорт зовнішніх моделей

Окрім мережевої синхронізації, у системі реалізовано два додаткові сценарії роботи зі станом: локальне збереження та відкриття сцени у власному JSON-форматі, а також імпорт зовнішніх GLB/GLTF-моделей через сервер. Ці механізми не дублюють один одного, а розв’язують різні задачі. Локальний файл сцени призначений для експорту й повторного відкриття користувацького проекту, тоді як імпорт моделі забезпечує довготривале збереження окремого 3D-ресурсу на сервері та його подальше колаборативне використання в усіх клієнтів.

При виконанні команди збереження клієнт формує JSON-об’єкт функцією `*buildSceneFilePayload()*`. До нього входять службові поля формату, версії та часу збереження, параметри поточного подання сцени (`*sceneSettings*`) і масив

серіалізованих об’єктів. Для кожного об’єкта до файлу записуються його `*id*`, `*type*`, `*customName*`, `*color*`, `*position*`, `*rotation*`, `*scale*`, а для моделей також `*fileId*` і `*url*`. Після цього вміст перетворюється на текст із форматуванням і завантажується у браузер як файл з розширенням `*.collab3d.json*`. Таким чином, користувач отримує переносимий опис сцени, не прив’язаний до конкретної сесії редактора.

Під час відкриття сцени файл спочатку проходить валідацію функцією `*normalizeLoadedSceneData(raw)*`. Перевіряється службове поле `*format*`, нормалізуються параметри `*sceneSettings*`, допустимі типи об’єктів і числові координати, а за відсутності окремих полів задаються безпечні стандартні значення. Після цього клієнт очищає поточну сцену через серію операцій `*delete*`, застосовує параметри візуалізації (`*showGrid*`, `*showAxes*`, `*showMiniAxis*`, `*transformSpace*`, `*inertia*`) і для кожного елемента з файла знову надсилає або `*create*`, або `*import_model*`. Важливо, що відтворення сцени після відкриття відбувається не локальним «тихим» оновленням, а через той самий протокол операцій, що і звичайне редагування. Це означає, що відкриття сцени на одному клієнті автоматично синхронізується з іншими учасниками поточного сеансу.

Імпорт зовнішньої моделі починається на клієнті з вибору файла користувачем. Обраний файл додається до `*FormData*` під ключем `*model*` і надсилається на HTTP-маршрут `*POST /upload-model*`. Сервер перевіряє наявність файла, контролює допустимі розширення `*.glb*` і `*.gltf*`, обмежує максимальний розмір до 200 МБ і далі записує бінарний вміст у GridFS *bucket* `*models*`. Після успішного запису клієнту повертається JSON-відповідь, що містить `*ok*`, `*fileId*`, `*url*`, службову назву файла, його початкове ім’я та розмір.

Отримавши результат HTTP-завантаження, клієнт не вставляє модель у сцену безпосередньо, а формує звичайну колаборативну операцію `*import_model*`. У ній передаються `*objId*`, `*fileId*`, `*url*`, початкові просторові параметри й колір. Сервер записує ці дані в `*objects*` і `*operations*`, а далі розсилає повідомлення іншим клієнтам. Кожен клієнт, у свою чергу, створює для такої моделі плейсхолдер `*Group*`, застосовує до нього трансформації, а потім

асинхронно підвантажує фактичний вміст файлу через `*GLTFLoader*` за адресою `*GET /models/:id*`. Після завершення завантаження до моделі додатково застосовується `*tintColor*`, щоб її колір був узгодженим у всіх учасників.

Слід враховувати, що найнадійнішим форматом для такого сценарію є самодостатній `*.glb*`, оскільки він містить усі ресурси в одному бінарному контейнері. Формат `*.gltf*` також приймається сервером, однак якщо модель залежить від зовнішніх `*.bin*`-файлів або текстур, ці ресурси мають бути доступні окремо. У поточному прототипі саме `*.glb*` є базовим практичним форматом для колаборативного імпорту моделей.

Отже, у системі реалізовано два комплементарні механізми роботи зі станом сцени. Локальний JSON-файл забезпечує переносимість повного користувацького проекту разом з параметрами подання, а серверний імпорт через GridFS дає змогу довготривало зберігати зовнішні 3D-моделі й робити їх доступними для всіх клієнтів одного колаборативного сеансу.

3.4 Реалізація серверної підсистеми

Серверна підсистема Collab3D реалізує центральну логіку координації спільного редагування сцени, забезпечує доставку змін між клієнтами, підтримує початкову синхронізацію та відповідає за довготривале збереження стану. На практичному рівні ця частина системи побудована на базі Node.js і містить два основні модулі: файл index.js, у якому зосереджено HTTP- і WebSocket-логіку, та файл db.js, що інкапсулює підключення до MongoDB і ініціалізацію GridFS-сховища моделей.

На відміну від децентралізованих підходів, у Collab3D усі комітні операції проходять через сервер, який виконує роль єдиного координуючого вузла. Саме сервер визначає остаточний порядок застосування змін, призначає версії журнальованим операціям, оновлює канонічний *snapshot* сцени та розсилає результати всім підключеним клієнтам. Така організація безпосередньо

відповідає формальній моделі, розглянутій у попередніх розділах, і дає змогу уникнути розходження локальних станів при одночасному редагуванні.

3.4.1 HTTP-частина сервера та допоміжні маршрути

Точкою входу серверної підсистеми є модуль index.js. Після запуску програми спочатку виконується асинхронне підключення до бази даних за допомогою функції *connectDB()*, після чого ініціалізуються посилання на базу *collab3d*, *GridFS-bucket models* та колекції *objects*, *operations* і *meta*. Лише після цього створюється HTTP-сервер, що є важливим для коректної роботи системи, оскільки значна частина серверної логіки залежить від уже доступного сховища даних.

Для реалізації HTTP-частини використано фреймворк Express. Сервер віддає статичні файли клієнтської підсистеми з каталогу *client*, а маршрут кореня «/» повертає головну HTML-сторінку редактора. Отже, сервер виконує не лише функцію синхронізації, а й роль точки публікації веб-застосунку, що спрощує розгортання системи на локальному сервері або в хмарному середовищі.

Окремий middleware-шар забезпечує базову підтримку CORS для сценаріїв локальної розробки. У коді передбачено дозвіл на запити з довільного джерела, а також обробку методів GET, POST, PUT, PATCH, DELETE та OPTIONS. Така конфігурація насамперед потрібна для коректного надсилання файлів моделей із клієнта в ситуаціях, коли редактор відкрито не безпосередньо з того ж самого джерела, а, наприклад, через локальний засіб попереднього перегляду сторінок.

Допоміжний маршрут *GET /health* використовується як легка перевірка доступності HTTP-сервера. Він повертає простий JSON-об’єкт із полем ok і застосовується як для зовнішнього health-check, так і для внутрішньої клієнтської перевірки стану з’єднання. Такий підхід дозволяє інтерфейсу редактора відображати окремо стан WebSocket-з’єднання і стан HTTP-сервера, що є корисним при діагностиці мережевих збоїв.

Ключовим HTTP-маршрутом серверної підсистеми є *POST /upload-model*. Для приймання файлів використано пакет *multer* у режимі *memoryStorage*, а в конфігурації задано обмеження на максимальний розмір файлу 200 МБ. Після отримання запиту сервер перевіряє наявність файлу в полі *model*, додатково валідуючи розширення і дозволяючи лише формати *.glb* та *.gltf*. Якщо перевірка завершується успішно, файл передається до GridFS як потік завантаження, а клієнтові повертається JSON-відповідь із ознакою успіху, ідентифікатором файлу, службовою URL-адресою завантаження, згенерованим ім’ям і метаданими початкового файлу.

Маршрут *GET /models/:id* реалізує зворотну операцію – видачу раніше збереженої моделі за її ідентифікатором. Сервер спочатку перевіряє коректність *ObjectId*, потім звертається до системної колекції *models.files* для пошуку опису файлу і, якщо такий запис існує, відкриває потік читання з GridFS. Перед передачею встановлюються заголовки *Content-Type* і *Cache-Control*, завдяки чому модель може бути ефективно кешована на боці браузера. Фактична доставка виконується у потоковому режимі без попереднього повного завантаження файлу в пам’ять, що є важливим для великих 3D-ресурсів.

HTTP-сервер створюється засобами модуля *http*, а не запускається окремо всередині Express. Це зроблено для того, щоб до одного й того самого мережевого порту можна було підв’язати і HTTP-обробники, і WebSocket-сервер. У підсумку застосунок використовує єдину точку входу для завантаження клієнта, завантаження моделей, перевірки працездатності та двостороннього обміну подіями синхронізації.

3.4.2 WebSocket-сервер і обробка операцій

Реальний режим спільної роботи в Collab3D забезпечується WebSocket-сервером, створеним на базі бібліотеки *ws*. Об’єкт *WebSocketServer* ініціалізується поверх уже створеного HTTP-сервера, що дозволяє об’єднати обидва типи взаємодії в межах одного мережевого вузла. Така архітектура

спрощує конфігурацію системи та не вимагає окремого порту для каналу реального часу.

Після встановлення нового WebSocket-з’єднання для клієнта формуються технічні атрибути *clientId*, *clientIp* і *clientUserAgent*. Крім того, підтримуються службові поля isAlive і missedPongs, які використовуються механізмом *heartbeat* для контролю життєздатності з’єднань (див. рис. 3.14).

```
wss.on("connection", async (ws, req) => {
  ws.clientId = `ws_${++wsClientSeq}`;
  ws.clientIp = parseClientIp(req);
  ws.clientUserAgent = parseClientUserAgent(req);
  ws.isAlive = true;
  ws.missedPongs = 0;

  console.log(`Підключений новий клієнт WS ${formatWsClientLabel(ws)}
ua=${ws.clientUserAgent}`);

  ws.on("pong", wsHeartbeat);

  // 1) даємо повний стан
  await sendSnapshot(ws);
```

Рисунок 3.14 – Формування технічніх атрибутів

Відразу після підключення сервер викликає функцію *sendSnapshot()* і надсилає клієнтові повний поточний стан сцени у вигляді повідомлення типу *snapshot*. Завдяки цьому новий учасник сеансу не відтворює всю історію змін, а одразу отримує актуальне консистентне подання сцени (див. рис. 3.15).

```
/**
 * sendSnapshot(ws)
 * Відправляє snapshot одному клієнту.
 */
async function sendSnapshot(ws) {
  const snapshot = await buildSnapshot();
  ws.send(JSON.stringify({ type: "snapshot", data: snapshot }));
}
```

Рисунок 3.15 – Реалізація функції *sendSnapshot* в коді проекту

Для підвищення стійкості з’єднань у сервері реалізовано періодичний *heartbeat*-контроль, який було частково описано в п. 3.3.3 (див. рис. 3.12). Через

заданий інтервал часу сервер надсилає *ping*-пакети всім відкритим WebSocket-з'єднанням. Якщо клієнт не відповідає на них і кількість пропущених *pong*-відповідей перевищує поріг, таке з'єднання примусово завершується. Це дозволяє очищати неактуальні сесії і запобігає накопиченню «завислих» клієнтів на сервері. На прикладному рівні додатково підтримуються повідомлення *client_ping* і *client_pong*, що використовуються клієнтською частиною для індикатора синхронізації.

Усередині обробника *ws.on('message')* сервер виконує розбір JSON-повідомлення і класифікує його за полем *type*. Найпростішим є сценарій службового *ping*-повідомлення: у відповідь сервер повертає *client_pong* із часовими позначками. Окремий сценарій стосується *transform_preview* – проміжних повідомлень попереднього перегляду трансформацій. Такі події лише широкомовно розсилаються іншим клієнтам і свідомо не записуються до журналу та історії *undo/redo*.

Обробка команд *undo_request* і *redo_request* побудована на відновленні попередньо збережених *snapshot*-станів. Для *undo* сервер зменшує покажчик *pointer*, для *redo* – збільшує його в межах дозволеного діапазону. Далі виконується *restoreSnapshot()*, після чого новий стан технічної історії зберігається в колекції meta, а всім клієнтам розсилається повний *snapshot* через *broadcastSnapshot()*. Така схема гарантує, що після відкату або повторного застосування змін усі учасники сеансу одразу отримують однакове подання сцени.

Журнальованими в поточній реалізації вважаються операції *create*, *delete*, *transform*, *color*, *rename* та *import_model*. Для них сервер застосовує єдиний алгоритм: спочатку перевіряється належність типу до множини *history*-операцій, далі викликається *applyOperation()*, а після цього – *pushCurrentSnapshotToHistory()*. Усередині *applyOperation()* кожній операції призначається новий номер *version*, що формується функцією *nextVersion()*. Потім залежно від типу змінюється колекція *objects*, операція додається до *operations* і широкомовно розсилається всім клієнтам, включно з автором події.

Розсилання автору є свідомим рішенням, оскільки підтримує єдину модель синхронізації для всіх учасників сесії.

Особливість реалізації *transform* полягає в тому, що на сервер передається структура з полями *before* і *after*, а фактично в канонічний стан сцени записується саме кінцевий стан *after*. Це добре узгоджується з клієнтською логікою: під час безперервного переміщення, обертання або масштабування користувачі бачать *transform_preview*, а після завершення дії надсилається один комітний запис *transform*, придатний для коректного включення до історії *undo/redo*.

Для стабільності серверної підсистеми в коді також передбачено глобальні обробники *uncaughtException* і *unhandledRejection*, а для конкретних WebSocket-сесій – обробники *close* та *error*. Завдяки цьому навіть при некоректних вхідних даних, мережевих збоях або внутрішніх винятках сервер зберігає можливість журналювання проблем і не переходить у повністю некерований стан.

3.4.3 Організація збереження даних у MongoDB

Сховище даних серверної підсистеми організовано на базі MongoDB. У модулі db.js створюється клієнт MongoClient, а рядок підключення задається через змінну середовища MONGO_URI з резервним локальним значенням mongodb://localhost:27017. Після успішного встановлення з’єднання відкривається база даних collab3d, а допоміжні функції *getCollection()*, *getDB()* і *getModelsBucket()* надають іншим частинам серверного коду типізований доступ до колекцій і *GridFS-bucket*.

На відміну від спрощених схем, де база даних використовується лише як журнал операцій, у Collab3D MongoDB виконує кілька різних ролей. По-перше, колекція *objects* містить поточний канонічний *snapshot* сцени. Кожен документ у цій колекції відповідає одному об’єкту й містить поля *id, type, position, rotation*, *scale, color, customName*, а для імпортованих моделей – також *fileId* і *url*. Саме ця колекція є практичним джерелом істини для відновлення стану нових клієнтів і побудови повного *snapshot*.

По-друге, колекція operations виконує функцію журналу застосованих комітних операцій. Після виклику applyOperation() до неї додається повний запис операції разом із полем *version*, призначеним сервером. Такий журнал може використовуватися для аналізу послідовності змін, діагностики роботи системи та дослідження поведінки алгоритму синхронізації на реальних даних.

По-третє, колекція *meta* містить технічні метадані, необхідні для підтримки історії *undo/redo*. У поточній реалізації використовується документ із фіксованим ідентифікатором *undoRedo*, у якому зберігаються масив *history*, покажчик *pointer* та часовий штамп *updatedAt*. Саме цей документ дає змогу не втрачати стан історії при перезапуску сервера і забезпечує стабільну роботу повторного відновлення змін після рестарту.

Важливою деталлю є ініціалізація змінної *version*. Її початкове значення формується через *countDocuments()* для колекції *operations*, тобто нумерація комітних операцій продовжується від уже наявного журналу, а не починається щоразу з нуля. Це спрощує інтерпретацію журналу і робить послідовність версій природною навіть у випадку перезапуску сервера.

Механізм завантаження стану *undo/redo* реалізує функція *loadUndoRedoMeta()*. Якщо документ *undoRedo* вже існує й має коректну структуру, сервер відновлює *history* та *pointer* із бази даних. Якщо ж історія відсутня, сервер будує стартовий *snapshot* поточної сцени через *buildSnapshot()*, поміщає його до *history* і фіксує як початковий стан. Така поведінка запобігає появі невизначеного стану історії після першого запуску системи або після очищення метаданих.

При формуванні та оновленні історії використовується обмеження HISTORY_LIMIT. Якщо масив *history* перевищує встановлену межу, з нього відкидаються найстаріші елементи, а покажчик *pointer* коригується відповідно до нового індексного простору. Таким чином, сервер підтримує прогнозований обсяг технічних метаданих і запобігає неконтрольованому зростанню документа історії.

Операція *restoreSnapshot()* виконує відновлення колекції *objects* зі *snapshot*-структури за допомогою *bulkWrite()*. Для кожного об’єкта, що має бути присутнім у сцені, формується *replaceOne* з *upsert=true*, а для об’єктів, яких уже немає в цільовому *snapshot*, генеруються *deleteOne*-операції. Такий підхід є ефективнішим і концептуально чистішим, ніж повне очищення колекції з подальшим повторним заповненням, оскільки сервер виконує лише ті модифікації, які необхідні для приведення поточного стану до потрібного.

У підсумку організація збереження даних у MongoDB має багаторівневий характер: *objects* відповідає за актуальний стан сцени, *operations* – за журнал змін, *meta* – за технічну історію *undo/redo*, а GridFS – за двійкові ресурси моделей. Саме така структура даних найбільш адекватно відображає реальну архітектуру прототипу й дає змогу уникнути методичної невідповідності між теоретичним описом і програмною реалізацією (див. рис. 3.16).

```
/* ========================================================
   ІМПОРТИ
   ======================================================== */
// Цей файл відповідає за підключення до MongoDB,
// надання колекцій та GridFS bucket для зберігання GLB/GLTF.
import { MongoClient, GridFSBucket } from "mongodb";
/* ========================================================
   1. НАЛАШТУВАННЯ ПІДКЛЮЧЕННЯ
   ======================================================== */
// URI локальної MongoDB
const uri = process.env.MONGO_URI || "mongodb://localhost:27017";

// ПЕРЕВІРКА ВХОДУ
// Створюємо клієнт
const client = new MongoClient(uri);
// Тут зберігатиметься активна база після підключення
let db;
// GridFS bucket для моделей
let modelsBucket;
/* ========================================================
   2. ПІДКЛЮЧЕННЯ ДО БАЗИ
   ======================================================== */
/**
 * connectDB() * Підключається до MongoDB і відкриває базу "collab3d".
 * Також ініціалізує GridFS bucket "models".
 */
```

Рисунок 3.16 – Файл db.js. Програмна реалізація

```
export async function connectDB() {
  try {
    await client.connect();

    db = client.db("collab3d");
    modelsBucket = new GridFSBucket(db, { bucketName: "models" });


    console.log("MongoDB підключено");
  } catch (err) {
    console.error("Помилка підключення MongoDB:", err);
    throw err;
  }
}
/* ============================================================
   3. ОТРИМАННЯ КОЛЕКЦІЙ / DB / BUCKET
   ============================================================ */
export function getCollection(name) {
  if (!db) {
    throw new Error("База даних не підключена");
  }
  return db.collection(name);
}
export function getDB() {
  if (!db) {
    throw new Error("База даних не підключена");
  }
  return db;
}

export function getModelsBucket() {
  if (!modelsBucket) {
    throw new Error("GridFS bucket не ініціалізовано");
  }
  return modelsBucket;
}
/* ============================================================
   4. КОРЕКТНЕ ЗАКРИТТЯ ПІДКЛЮЧЕННЯ
   ============================================================ */
export async function closeDB() {
  try {
    await client.close();
    console.log("З'єднання MongoDB закрито");
  } catch (err) {
    console.error("Помилка закриття MongoDB:", err);
  }
}
```

Рисунок 3.16, аркуш 2

3.4.4 Використання GridFS для збереження 3D-моделей

Особливе місце в серверній підсистемі займає збереження зовнішніх 3D-моделей. На відміну від параметрів примітивних об’єктів, двійкові файли GLB і GLTF нераціонально зберігати безпосередньо в документах колекції *objects* або operations. Тому в системі застосовано GridFS – стандартний механізм MongoDB для зберігання великих файлів у вигляді потоку блоків.

Ініціалізація GridFS виконується в модулі db.js (див. рис. 3.15) через створення об’єкта *GridFSBucket* із назвою *bucketName = 'models'*. Це означає, що фізично службові записи про моделі розміщуються в системних колекціях *models.files* і *models.chunks*. Для інших частин серверного коду GridFS представлено через функцію getModelsBucket(), що приховує технічні деталі реалізації і дозволяє працювати з моделями як із поточним логічним сховищем.

Під час завантаження моделі маршрут *POST /upload-model* створює в GridFS новий запис із технічним ім’ям, що генерується на основі поточного часу і випадкового суфікса. Разом із бінарним вмістом у *metadata* зберігаються *originalName*, *size* та *uploadedAt*. Після успішного завершення операції клієнтові повертається *fileId*, який надалі використовується як стабільне посилання на модель у документах *objects*, у журналі operations і в локальних файлах сцен формату collab3d-scene.

При імпорті моделі сервер не вбудовує її двійковий вміст у *snapshot* сцени. Натомість у даних об’єкта зберігаються лише *fileId* і *url*. Така схема робить *snapshot* компактним, спрощує широкомовну розсилку між клієнтами й дозволяє повторно використовувати один і той самий ресурс без дублювання бінарних даних. Клієнтська підсистема, отримавши інформацію про модель, створює *placeholder*-об’єкт і самостійно завантажує реальний ресурс за наданим URL.

Маршрут *GET /*models/:id реалізує потокове читання моделі з GridFS та її повернення клієнтові. Використання *openDownloadStream()* важливе з погляду економії пам’яті, оскільки сервер не формує повну копію файлу в оперативній пам’яті перед відправленням. Додаткове встановлення довготривалого

кешування через *Cache-Control* є виправданим, оскільки після завантаження файлу з унікальним ідентифікатором його вміст фактично не змінюється.

Практична цінність GridFS у Collab3D полягає ще й у тому, що стан сцени зберігає працездатність після перезавантаження сторінки або рестарту сервера. Якщо об’єкт типу *model* вже присутній у колекції *objects*, сервер при побудові *snapshot* знову передає клієнту *fileId* і *url*, а клієнт може повторно витягти модель зі сховища. Це усуває проблему тимчасових локальних посилань і робить імпортовані 3D-ресурси повноцінною частиною довготривалого стану системи.

Разом із тим поточна реалізація має і цілком визначене обмеження: формат *.glb* підтримується безпосередньо, тоді як для *.gltf*-файлів із зовнішніми *.bin*-файлами або текстурами необхідна окрема схема збереження й видачі пов’язаних ресурсів. Отже, найстабільнішим форматом для поточного прототипу є саме GLB. Фіксація цього обмеження в тексті роботи є важливою, оскільки вона відображає реальний стан системи, а не бажану, але ще не реалізовану функціональність.

Таким чином, використання GridFS дозволило відокремити параметричний стан сцени від двійкових ресурсів моделей, забезпечити повторне використання імпортованих об’єктів, зробити серверне сховище стійким до перезапусків і підготувати основу для подальшого розширення системи в напрямі підтримки складніших форматів зовнішніх 3D-активів.

Отже, серверна підсистема Collab3D реалізує завершений цикл обробки спільних змін: від приймання HTTP- і WebSocket-запитів до оновлення канонічного *snapshot* сцени, журналювання подій, підтримки історії *undo/redo* та збереження зовнішніх 3D-моделей. Саме така організація сервера створює надійну технологічну основу для функціонування клієнтської частини, опис якої буде наведено в наступному підрозділі.

3.5 Реалізація клієнтської підсистеми

Клієнтська підсистема редактора Collab3D реалізує побудову тривимірної сцени, відображення користувацького інтерфейсу, обробку локальних дій користувача та синхронізацію стану із сервером у режимі реального часу. На відміну від серверної частини, яка відповідає за серіалізацію та збереження змін, клієнтська підсистема поєднує графічний рендеринг, інтерактивну взаємодію з об’єктами сцени та прикладну логіку мережевого обміну. Така побудова дає змогу розділити відповідальність між компонентами системи: візуальне представлення та первинна обробка подій залишаються на клієнті, тоді як канонічний стан сцени формується на сервері.

У поточній реалізації клієнтська частина завантажується як статичний вебзастосунок і використовує модульний підхід. Бібліотека Three.js та її допоміжні модулі підключаються через *import map*, а прикладна логіка зосереджена у модулі app.js. Структурно клієнтська підсистема складається з чотирьох логічно пов’язаних шарів: сцени візуалізації, механізмів вибору та трансформації, користувацького інтерфейсу та модуля синхронізації з сервером. Саме така структура узгоджується з підрозділами даного розділу й дозволяє послідовно описати програмну реалізацію редактора.

3.5.1 Побудова та рендеринг сцени у Three.js

Побудова клієнтської сцени виконується засобами Three.js. Після ініціалізації створюється об’єкт *Scene*, для якого задається темний фон, перспективна камера та WebGL-рендерер з увімкненим згладжуванням. Розмір полотна рендерера автоматично пристосовується до поточних розмірів вікна браузера, а коефіцієнт *pixel ratio* обмежується зверху для зменшення надмірного навантаження на пристрої з високою щільністю пікселів. Такий підхід забезпечує достатню якість візуалізації без непотрібного зростання обчислювальної вартості рендерингу.

Просторова орієнтація сцени підтримується за допомогою сітки координат та глобального вказівника осей. Для освітлення використовуються джерело типу *HemisphereLight* та спрямоване світло *DirectionalLight*, чого достатньо для наочного відображення геометричних примітивів і завантажених моделей. Крім основного полотна рендерингу, у правому верхньому куті створюється окремий накладений *viewport mini-axis*. Він має власну сцену, камеру та рендерер і відображає орієнтацію глобальних осей відносно поточного положення камери. Кінці осей у цьому віджеті є інтерактивними: після клацання виконується анімований перехід камери до відповідного ортогонального виду, що полегшує навігацію у просторі сцени.

Локальна модель сцени на клієнті подається асоціативною структурою *objects*, у якій кожному колаборативному ідентифікатору об’єкта ставиться у відповідність кореневий Object3D. Для геометричних примітивів таким кореневим вузлом є окремий *Mesh*, а для зовнішніх GLB/GLTF-моделей – група *Group*, що виконує роль контейнера. Важливо, що саме кореневий вузол використовується як об’єкт трансформації та синхронізації, тоді як внутрішня структура імпортованої моделі лишається локальною для клієнта. Такий підхід повністю узгоджується з формальною моделлю, у якій синхронізованою одиницею є верхньорівневий об’єкт сцени.

Для імпортованих моделей застосовано механізм *placeholder*-завантаження. Після надходження повідомлення про імпорт на сцену одразу додається порожній кореневий контейнер із потрібним ідентифікатором, трансформацією та службовими атрибутами. Після цього модуль *GLTFLoader* асинхронно завантажує вміст моделі за переданим URL і вставляє її до вже існуючого контейнера. Таке рішення дозволяє не блокувати логіку редактора під час мережевого завантаження та відображати уніфікований стан об’єкта навіть до повного завершення імпорту ресурсу.

Основний цикл візуалізації реалізовано через *requestAnimationFrame*. У кожній ітерації оновлюється стан камери орбіти, анімований перехід до ортогональних видів, значення інспектора, орієнтація *mini-axis* та безпосередньо

виконується рендеринг основної сцени і допоміжного осьового *viewport*. Оскільки більшість прикладної логіки редагування виконуються в обробниках подій, цикл анімації зосереджений переважно на плавному візуальному оновленні та не містить важких операцій синхронізації.

3.5.2 Механізми вибору об’єктів і трансформацій

Вибір об’єктів у сцені базується на використанні *Raycaster*. Координати вказівника миші або сенсорної події перетворюються до нормалізованих координат області рендерингу, після чого виконується перетин променя з вмістом сцени. Особливість реалізації полягає в тому, що після попадання в окремий дочірній *Mesh* відбувається підйом по ієрархії до найближчого вузла, який має *userData.id*. Таким чином, навіть коли користувач клацає по окремій частині GLTF-моделі, вибирається саме той кореневий об’єкт, що бере участь у колаборативній синхронізації.

Для трансформацій використовується комбінована схема. Режими обертання і масштабування реалізовано через *TransformControls*, який приєднується до вибраного об’єкта та працює спільно з окремим helper-представленням *gizmo*. Під час початку маніпуляції зберігається стан *before*, а після завершення – стан *after*. Саме пара значень *before/after* передається на сервер як комітна операція transform, що спрощує відновлення стану, журналювання та *undo/redo*. Під час самого переміщення маркер маніпулятора породжує лише короткі повідомлення *transform_preview*, які дають іншим клієнтам уявлення про поточний рух, але не засмічують історію змін.

Режим переміщення реалізовано окремим власним механізмом *pointer-drag* без використання *DragControls*. Після початку перетягування фіксується *pointerId*, вимикається орбітальна камера та встановлюється захоплення вказівника через *setPointerCapture*. Переміщення об’єкта обчислюється як перетин променя курсора з допоміжною площиною, що проходить через поточну позицію об’єкта й орієнтована перпендикулярно напряму погляду камери. Такий

підхід усуває характерні проблеми з нестабільністю перетягування та дає більш природну поведінку на настільних і мобільних пристроях.

Окрему увагу в реалізації приділено конфлікту між орбітальною камерою та *gizmo*. Для коректної роботи використовуються додаткові перевірки попадання курсора у *helper TransformControls* з підвищеним порогом чутливості до тонких ліній осей. Якщо система визначає, що користувач взаємодіє саме з *gizmo*, обертання сцени тимчасово вимикається. Це усуває ситуації, коли при натисканні на вісь маніпулятора браузерна сцена починає обертатися замість зміни параметрів вибраного об’єкта.

Редагування параметрів трансформації доступне також через числовий інспектор. Для позиції, обертання та масштабу передбачено поля X, Y, Z та однорідне значення Δ, крок яких користувач задає окремими степерами. Зміни можуть вноситися вручну, колесом миші або перетягуванням маркерів осей безпосередньо в інспекторі. Після завершення редагування формується одна журнальована операція *transform*, що забезпечує єдність мережевої та локальної моделі редагування.

3.5.3 Реалізація користувацького інтерфейсу редактора

Користувацький інтерфейс редактора побудовано як поєднання верхньої панелі команд, правої бічної панелі інспектора та допоміжних оверлеїв. У верхньому меню розміщено три основні групи дій: «Файл», «Додати» та «Редагувати». Меню «Файл» забезпечує створення нової сцени, збереження сцени до локального файла і відкриття сцени з раніше збереженого JSON-подання. Меню «Додати» дає змогу створювати геометричні примітиви та імпортувати GLB/GLTF-моделі, а меню «Редагувати» об’єднує дії *undo*, *redo*, клонування, зміни кольору та видалення вибраного об’єкта.

Права панель реалізує функції інспектора сцени та може згортатися для збільшення робочої області. На верхньому рівні панелі розміщено швидкі кнопки *undo/redo*, далі – згортані секції «Сцена», «Список об’єктів» і «Об’єкт». У секції

«Сцена» користувач керує інерцією камери, видимістю сітки, глобальних осей та *mini-axis*, а також обирає світовий або локальний простір обертання. Тут же реалізовано перемикання режимів переміщення, обертання та масштабування та окремі прапорці вмикання або вимикання *gizmo* для кожного режиму (див. рис. 3.17).

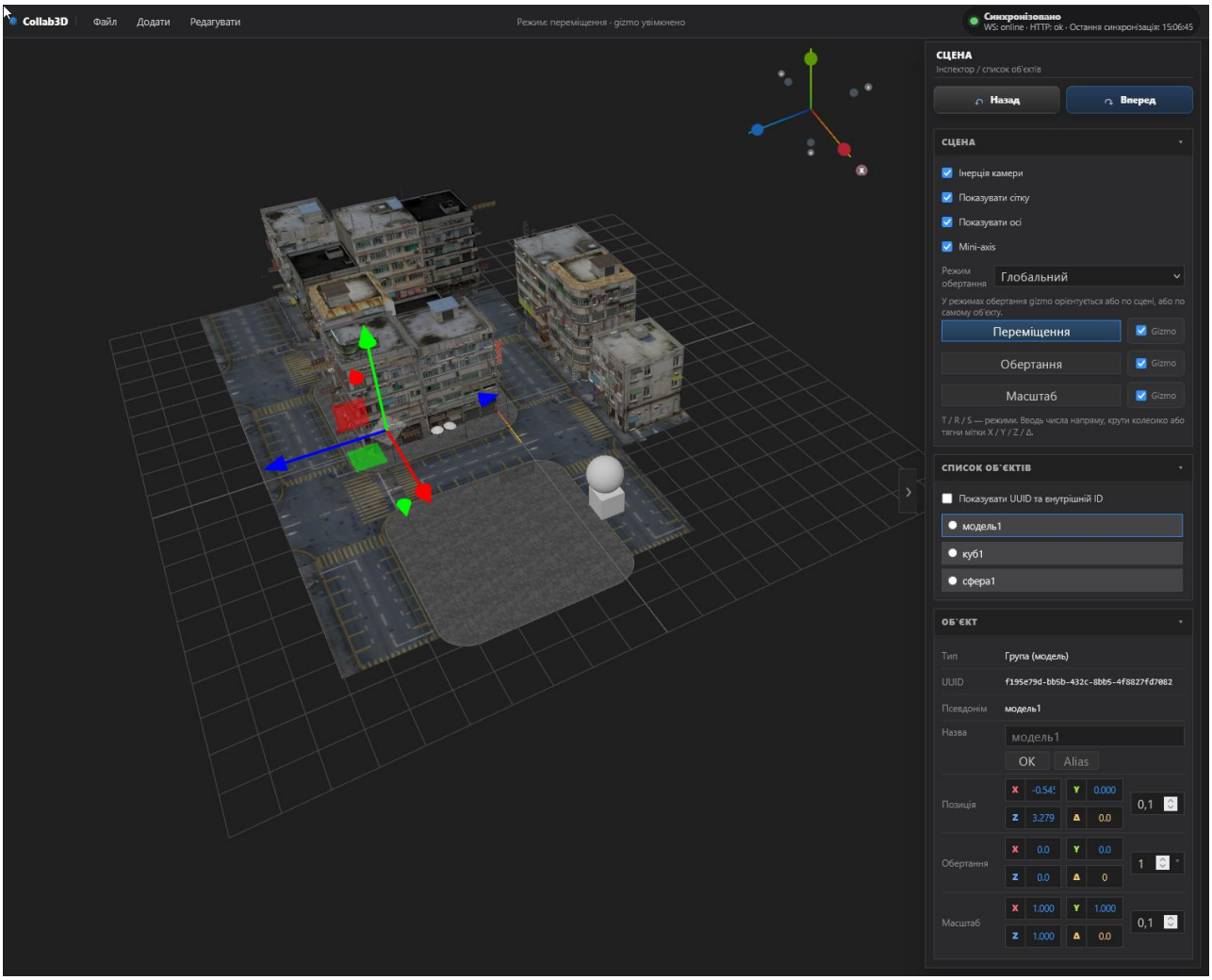


Рисунок 3.17 – Інтерфейс 3D-редактора

Список об’єктів формується динамічно на основі локальної таблиці *objects*. Для кожного елемента відображається псевдонім, сформований автоматично за типом і порядковим номером, а також колірний маркер, який відповідає поточному матеріалу примітива або *tint*-кольору моделі. За потреби користувач може ввімкнути відображення внутрішнього UUID та колаборативного

ідентифікатора. Такий спосіб подання полегшує орієнтацію в сцені та напряму пов’язує графічні об’єкти з їх логічним поданням.

Секція властивостей вибраного об’єкта містить тип, UUID, автоматичний псевдонім, користувацьку назву та редаговані параметри трансформації. Механізм іменування розділяє автоматично згенеровані псевдоніми та довільні імена користувача, що зберігаються в *customName*. Завдяки цьому інтерфейс поєднує стабільну службову ідентифікацію з більш зручним для людини способом найменування елементів сцени.

До складу інтерфейсу входить також інформаційний блок синхронізації. Він складається з компактного індикатора стану з’єднання та окремого повідомлення про автономний режим. Індикатор розрізняє стани *connecting*, *online*, *reconnecting* і *offline*, відображає результат перевірки WebSocket та HTTP-каналу, а також показує час останньої успішної синхронізації. Таке рішення є важливим з прикладної точки зору, оскільки дозволяє користувачеві швидко зрозуміти, чи його дії вже синхронізуються із сервером.

3.5.4 Синхронізація змін між клієнтами в режимі реального часу

Синхронізація клієнтської частини із сервером виконується через постійне WebSocket-з’єднання. Формування й передавання повідомлень зосереджено у функції *sendOperation*, яка є єдиною точкою серіалізації клієнтських операцій у формат JSON. Якщо канал ще не відкрито, операція не надсилається, а інтерфейсний стан синхронізації переходить до *reconnecting* або *offline*. Така поведінка принципово відрізняється від повноцінного офлайн-режиму: у поточному прототипі локальна черга непідтверджених операцій не ведеться, а після відновлення з’єднання клієнт повторно синхронізується із серверним станом (див. рис. 3.18).

Після встановлення з’єднання клієнт отримує від сервера повний *snapshot* сцени. Обробка цього повідомлення реалізована у функції *applySnapshot*, яка видаляє локальні об’єкти, відсутні у знімку, створює або оновлює примітиви,

ініціалізує *placeholder*-контейнери для моделей, застосовує трансформації та кольори і, за потреби, запускає асинхронне підвантаження GLTF-вмісту. Після цього оновлюються список об’єктів, інспектор та стан *gizmo*. Отже, *snapshot* виконує роль механізму початкової синхронізації, відновлення після reconnect і глобального вирівнювання стану клієнта.

```
/**
 * sendOperation()
 */
function sendOperation(op) {

  /**
   * Відправляє одну "операцію" на сервер по WebSocket.
   * Це єдина точка, де ми серіалізуємо дані в JSON і робимо socket.send().
   * Якщо socket ще не OPEN, функція нічого не робить - це захист від помилок
при старті/перепідключенні.
   */
  if (!socket || socket.readyState !== WebSocket.OPEN) {
    setSyncState(healthOk ? "reconnecting" : "offline");
    return false;
  }

  socket.send(JSON.stringify(op));
  return true;
}
```

Рисунок 3.18 – Реалізація функції *sendOperation* в коді проекту

Поточні інкрементальні зміни обробляються в *handleSocketMessage*. Для повідомлень типу *create* клієнт створює відповідний примітив і додає його на сцену; для *import_model* – ініціалізує контейнер моделі й запускає її асинхронне завантаження; для *delete* – видаляє елемент зі сцени; для *transform* та *transform_preview* – застосовує новий трансформаційний стан; для *rename* і *color* – відповідно оновлює назву та візуальні характеристики об’єкта. Таким чином, одна й та сама клієнтська логіка дозволяє відтворювати локальні та віддалені зміни у єдиному поданні сцени (див. рис. 3.19).

```
function handleSocketMessage(event) {
  let msg;
  try { msg = JSON.parse(event.data); } catch { return; }

  lastSyncAt = Date.now();
  if (healthOk) setSyncState("online");

  if (msg.type === "client_pong") {
    return;
  }

  if (msg.type === "snapshot") {
    applySnapshot(msg.data);
    return;
  }

  if (msg.type === "transform_preview") {
    const obj = objects[msg.objId];
    if (!obj) return;
    applyTransform(obj, msg.data?.after);
    return;
  }

  if (msg.type === "create") {
    const data = msg.data || {};
    const mesh = createPrimitiveMesh(data.type, data.color);
    addObjectToScene(msg.objId, mesh, data);
    return;
  }

  if (msg.type === "import_model") {
    const data = msg.data || {};
    if (!data.url) return;

    const root = ensureModelPlaceholder(msg.objId, data);
    root.userData.fileId = data.fileId || root.userData.fileId || "";
    root.userData.tintColor = data.color || root.userData.tintColor ||
"#ffffff";

    applyTransform(root, data);
    loadModelIntoPlaceholder(data.url, msg.objId);
    return;
  }

  if (msg.type === "delete") {
    removeObjectFromScene(msg.objId);
    return;
  }

  if (msg.type === "transform") {
    const obj = objects[msg.objId];
    if (!obj) return;
```

Рисунок 3.19 – Реалізація функції *handleSocketMessage* в коді проекту

```
    applyTransform(obj, msg.data?.after || msg.data);
    return;
  }
  if (msg.type === "rename") {
    const obj = objects[msg.objId];
    if (!obj) return;
    setObjectCustomName(obj, msg.data?.after || "");
    refreshObjectList();
    updateInspector();
    return;
  }

  if (msg.type === "color") {
    const obj = objects[msg.objId];
    if (!obj) return;

    const after = normalizeColor(msg.data?.after || msg.data || "#ffffff");

    if (obj.isMesh && obj.material?.color) {
      obj.material.color.set(after);
      obj.material.needsUpdate = true;
    } else {
      obj.userData.tintColor = after;
      applyTintToObject(obj, after);
    }

    if (selectedObject === obj) colorPicker.value = after;
    refreshObjectList();
    updateInspector();
  }
}
```

Рисунок 3.19, аркуш 2

Для імпортованих моделей синхронізація поєднує HTTP- і WebSocket-взаємодію. Спочатку клієнт надсилає файл моделі на сервер методом *POST /upload-model*, після чого отримує ідентифікатор GridFS-файлу та URL доступу. Далі в систему передається стандартна операція *import_model*, яка вже синхронізується між усіма клієнтами так само, як і інші журнальовані зміни. Для кольорового тонування імпортованих моделей використано окрему локальну процедуру *applyTintToObject*, яка рекурсивно проходить по всіх *Mesh* усередині контейнера та змінює значення *material.color*.

Окремий рівень підтримки стійкості забезпечується періодичними перевірками каналу зв’язку. Клієнт регулярно виконує HTTP-запит до маршруту */health* і надсилає службові повідомлення *client_ping*, а сервер відповідає

client_pong і підтримує власний *heartbeat*-механізм. При розриві з'єднання використовується автоматичне перепідключення з наростаючою затримкою. Після успішного відновлення каналу клієнт не намагається інтегрувати локально накопичені операції, а повторно отримує актуальний стан сцени, що відповідає прийнятому в роботі принципу централізованого джерела істини.

Крім мережевої синхронізації між клієнтами, у клієнтській підсистемі реалізовано і локальні механізми збереження та відкриття сцени. Сцена може бути експортована до файла формату *collab3d-scene JSON*, який містить список об'єктів і поточні параметри відображення сцени. Під час відкриття такого файла виконується нормалізація даних, локальне очищення поточної сцени через послідовність *delete*-операцій та повторне створення всіх елементів засобами стандартних мережевих команд *create* і *import_model*. Це дозволяє не вводити окремий спеціалізований механізм завантаження у протокол і повторно використовувати вже реалізовані канали синхронізації.

3.6 Висновки до розділу 3

У розділі 3 було послідовно розглянуто проектування та програмну реалізацію колаборативної системи редагування 3D-сцен Collab3D на основі централізованої клієнт-серверної архітектури. Показано, що практична побудова системи узгоджується з формальною моделлю, сформульованою в попередньому розділі: клієнтська частина відповідає за візуалізацію сцени, обробку дій користувача та формування повідомлень протоколу, а серверна частина виконує прийом, серіалізацію, фіксацію та розповсюдження змін між усіма підключеними клієнтами.

У результаті аналізу архітектури та технологічного стеку встановлено, що використання Three.js, WebSocket, Node.js та MongoDB є обґрунтованим для задачі спільного редагування тривимірних сцен у браузері. Такий стек забезпечує апаратно прискорений рендеринг, подієву модель взаємодії в режимі

реального часу, неблокуючу обробку запитів на сервері та гнучке документно-орієнтоване збереження стану сцени, історії змін і зовнішніх 3D-моделей.

У підрозділі 3.1 було уточнено архітектуру реалізованої колаборативної системи редагування 3D-сцен відповідно до фактичного програмного прототипу. Показано, що система має централізовану клієнт-серверну структуру, у якій сервер виступає координатором змін і джерелом істини для актуального стану сцени. Окремо конкретизовано моделі станів клієнта і сервера, уточнено реальний формат JSON-повідомлень та скориговано опис механізму відновлення після втрати з'єднання.

Поточний опис підрозділу не приписує системі не реалізованих властивостей, таких як *replay* локальної черги після *reconnect* або використання операції *reparent*. Натомість архітектурний опис побудовано безпосередньо на реальних компонентах прототипу: HTTP-маршрутах, WebSocket-координації, *snapshot*-синхронізації, *heartbeat*-механізмах і MongoDB-підсистемі. Саме така постановка створює коректну основу для подальшого опису протоколу взаємодії, структури даних і практичної реалізації серверної та клієнтської частин.

У підрозділі 3.2 було обґрунтовано вибір технологічного стеку, використаного для реалізації Collab3D. Показано, що клієнтська частина на базі HTML, CSS, JavaScript і Three.js забезпечує інтерактивне редагування та візуалізацію сцени, серверна частина на Node.js, Express і ws реалізує координацію та синхронізацію в реальному часі, а MongoDB і GridFS формують багаторівневу підсистему збереження стану, журналу, історії й модельних ресурсів.

Окрему увагу приділено тому, що опис технологічної платформи повинен відображати саме фактичну реалізацію. Тому підсистему збереження уточнено не як абстрактний журнал операцій, а як сукупність колекцій *objects*, *operations*, *meta* та сховища GridFS. Також у підрозділі зафіксовано реальні інтерфейсні можливості клієнта – меню, праву панель, список об'єктів, інспектор властивостей, режими трансформації, *mini-axis* і механізми збереження та відкриття сцени. Усе це створює коректний перехід до подальшого детального

опису протоколу взаємодії клієнта і сервера, структур даних та програмної реалізації окремих підсистем.

У підрозділі 3.3 було деталізовано протокол взаємодії клієнта і сервера та структури даних, що лежать в основі системи. Встановлено, що синхронізація стану сцени реалізується через сукупність WebSocket-повідомлень для створення, видалення, трансформації, перейменування, зміни кольору, імпорту моделі, попереднього перегляду трансформацій, а також службових команд початкової синхронізації, перевірки з'єднання та відновлення стану після перепідключення. Окремо показано, що механізм *undo/redo* у поточному прототипі спирається на історію *snapshot*-станів сцени, а не на повторне програвання незалежних інверсних команд.

У підрозділі 3.4 було охарактеризовано реалізацію серверної підсистеми, яка поєднує HTTP-частину з WebSocket-сервером на одному мережевому вузлі. Визначено призначення основних маршрутів, механізм обробки операцій, модель збереження поточного стану сцени в колекції *objects*, журналу операцій у *collections operations*, метаданих історії в *meta*, а також використання GridFS для довготривалого зберігання імпортованих GLB/GLTF-моделей. Це забезпечує не лише поточну синхронізацію між клієнтами, а й коректне відновлення сцени після перезапуску системи або повторного підключення.

У підрозділі 3.5 було розкрито реалізацію клієнтської підсистеми редактора. Показано, що інтерфейс і функціональні засоби системи охоплюють побудову та рендеринг сцени, вибір об'єктів, режими переміщення, обертання та масштабування, використання *gizmo*, *mini-axis*, список об'єктів, інспектор властивостей, зміну кольору, імпорт моделей, а також збереження і відкриття сцени у локальному файловому форматі. Окремо підкреслено, що клієнтська частина підтримує автоматичне повторне підключення до сервера та повторне отримання актуального *snapshot* сцени, але не реалізує повноцінний офлайн-режим із чергою локальних операцій, що методично узгоджує опис системи з фактичним кодом прототипу.

Таким чином, у розділі 3 було сформовано цілісне уявлення про архітектуру, протокол взаємодії, серверну логіку, підсистему збереження та клієнтський редактор Collab3D. Отримані результати підтверджують працездатність запропонованих проектних рішень і створюють необхідну прикладну основу для переходу до наступного розділу, присвяченого експериментальному дослідженню, функціональному тестуванню, перевірці синхронізації, оцінюванню продуктивності та аналізу стійкості системи до мережевих збоїв.

4 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА ТЕСТУВАННЯ СИСТЕМИ

4.1 Умови та методика експериментального дослідження

Експериментальне дослідження колаборативної системи редагування 3D-сцен Collab3D спрямоване на практичну перевірку того, наскільки реалізований програмний прототип відповідає сформульованій у попередніх розділах моделі синхронізації, архітектурним рішенням і функціональним вимогам. На відміну від теоретичного аналізу, який доводить коректність алгоритму на рівні формальних властивостей, експериментальна частина має показати працездатність системи в реальних умовах використання – при одночасному підключенні кількох клієнтів, виконанні типових операцій редагування, імпорті зовнішніх моделей, використанні механізму *undo/redo*, а також у сценаріях повторного підключення після тимчасової втрати зв’язку.

Побудова методики експерименту виконувалася з урахуванням специфіки реалізованого прототипу. Система складається з клієнтської частини на базі браузерного застосунку Three.js, серверної підсистеми на Node.js з використанням HTTP та WebSocket, а також документно-орієнтованого сховища MongoDB, у якому окремо підтримуються поточний *snapshot* сцени, журнал операцій, метадані історії *undo/redo* та сховище зовнішніх 3D-моделей на основі GridFS. Відповідно, експериментальна постановка повинна охоплювати не лише зовнішню функціональність редактора, а й взаємодію між цими підсистемами.

4.1.1 Мета та завдання експериментального дослідження

Основною метою експериментального дослідження є комплексна оцінка працездатності реалізованої системи колаборативного редагування 3D-сцен у типових і граничних режимах використання. У межах цієї мети необхідно

встановити, чи забезпечує система коректне створення, зміну, видалення та відновлення об’єктів сцени, чи підтримує однаковий стан сцени на всіх підключених клієнтах, чи відновлює актуальний стан після перепідключення, а також чи зберігає прийнятну інтерактивність під час виконання безперервних трансформацій.

Для досягнення поставленої мети було сформульовано такі завдання експериментального дослідження. По-перше, перевірити коректність базових операцій редагування над об’єктами сцени: додавання геометричних примітивів, зміни кольору, перейменування, просторового переміщення, обертання, масштабування та видалення. По-друге, оцінити узгодженість стану при одночасній роботі кількох клієнтів із єдиною сценою. По-третє, перевірити працездатність імпорту зовнішніх GLB/GLTF-моделей, їх збереження на сервері та повторне відтворення на інших клієнтах. По-четверте, дослідити механізм *undo/redo*, що реалізований через історію *snapshot*-станів і покажчик активного стану. По-п’яте, перевірити поведінку системи в умовах тимчасового розриву WebSocket-з’єднання та наступної синхронізації клієнта із сервером через повторне отримання *snapshot* сцени.

Таким чином, експериментальна частина орієнтована не на ізольовану перевірку окремих функцій, а на комплексну валідацію всієї системи як завершеного колаборативного веб-редактора, у якому коректність окремої операції має оцінюватися разом із її впливом на синхронізацію, історію змін, підсистему збереження даних і користувацький інтерфейс.

4.1.2 Умови проведення дослідження

Експериментальне дослідження проводилося у двох основних режимах: локальному та мережевому. У локальному режимі серверна частина запускалася як Node.js-застосунок, що одночасно обслуговує HTTP-маршрути та WebSocket-з’єднання, тоді як MongoDB використовувалася як локальне документне сховище стану сцени, журналу операцій, метаданих *undo/redo* та GridFS-файлів

моделей. Такий режим дозволяє ізолювати функціональні властивості системи від зовнішніх мережевих факторів і перевірити правильність логіки інтеграції операцій, формування *snapshot*-станів та відновлення сцени з бази даних.

У мережевому режимі дослідження виконувалося з доступом до застосунку через веб-клієнт за реальним мережевим з'єднанням. Це дозволяло оцінити роботу механізмів WebSocket-синхронізації в умовах природних затримок, перевірити повторне підключення клієнтів, реакцію індикаторів стану синхронізації та загальну стійкість редактора при доступі з різних пристроїв. Для клієнтської перевірки використовувалися сучасні браузери настільного класу; окремо враховувався сценарій відкриття редактора на мобільному пристрої, оскільки поточна реалізація містить спеціальні доробки, пов'язані з взаємодією з *gizmo*, камерою та сенсорним введенням.

З погляду логічної постановки експерименту використовувалися такі режими роботи: одноклієнтський режим для первинної перевірки функціональності, двоклієнтський режим для перевірки синхронності та багатоклієнтський режим для спостереження за стабільністю розсилки змін. У кожному з цих режимів стан сцени оцінювався одночасно на всіх активних клієнтах, а також за станом серверних колекцій. Особлива увага приділялася відповідності між візуальним станом сцени в браузері, поточним *snapshot* у колекції *objects*, журналом *operations* та метаданими *history/pointer* у колекції *meta*.

Важливою умовою експерименту було використання саме тієї реалізації, що описана в розділі 3, без введення спрощених або спеціально модифікованих тестових механізмів. Це означає, що перевірка виконувалася на реальному прототипі з фактичними маршрутами */health, /upload-model, /models/:id*, реальною WebSocket-обробкою повідомлень і чинним клієнтським інтерфейсом, який включає верхнє меню, праву панель, список об'єктів, інспектор властивостей, режими трансформацій, *mini-axis* та індикатор стану синхронізації.

4.1.3 Методика проведення експериментів

Методика дослідження будувалася на послідовному виконанні серій тестових сценаріїв, кожен із яких був спрямований на перевірку окремої групи властивостей системи. На першому етапі виконувалася первинна ініціалізація сцени: запуск сервера, підключення одного або кількох клієнтів і перевірка надходження початкового *snapshot*. Тут контролювалося, чи отримує новий клієнт повний актуальний стан сцени, а не порожню копію або фрагмент попередніх змін.

На другому етапі перевірялися базові операції над об’єктами. Користувачі послідовно створювали примітиви різних типів, змінювали їхні назви, кольори та трансформаційні параметри, після чого фіксувалося, чи відображаються зміни на інших клієнтах без логічних розходжень. Для режиму переміщення оцінювалася безперервна синхронізація через *transform_preview* та фінальна фіксація комітної операції *transform*; для режимів обертання і масштабування аналогічно контролювалися попередній перегляд і кінцевий *commit* після завершення дії користувача.

На третьому етапі досліджувався імпорт зовнішніх моделей. Вибраний GLB або GLTF-файл завантажувався з клієнта на сервер через HTTP-маршрут */upload-model*, після чого в сцені створювався об’єкт типу *model* з посиланням на */models/:id*. Методика перевірки вимагала не лише появи моделі на клієнті-ініціаторі, а й її коректного відтворення на інших клієнтах, а також перевірки того, що запис про модель зберігається у вигляді метаданих об’єкта, тоді як сам файл фізично міститься у GridFS.

На четвертому етапі тестувався механізм *undo/redo*. Для цього виконувалася наперед визначена послідовність операцій над сценою: створення одного або кількох об’єктів, модифікація їхніх параметрів, видалення, імпорт моделі, повторна зміна властивостей. Після цього послідовно ініціювалися команди *undo_request* і *redo_request*. Контролювалося, чи повертається сцена до попередніх *snapshot*-станів у правильному порядку, чи узгоджено оновлюється

стан на всіх клієнтах і чи не виникає логічного розходження між візуальною сценою, колекцією *objects* та покажчиком *history/pointer*.

На п’ятому етапі перевірялася робота механізмів повторного підключення та відновлення стану. Для цього один із клієнтів штучно від’єднувався від сервера або потрапляв у ситуацію тимчасової недоступності WebSocket-з’єднання, після чого виконував повторне підключення. Методика перевірки передбачала спостереження за зміною локального індикатора синхронізації, надходженням нового *snapshot* після *reconnect* та відновленням актуального стану сцени без ручного втручання. При цьому окремо враховувалося, що поточна реалізація не підтримує повноцінну офлайн-чергу локальних операцій, а відновлює клієнтський стан саме шляхом повторної синхронізації із серверним *snapshot*.

На шостому етапі контролювалися функції локального збереження та відкриття сцени у форматі collab3d-scene. Перевірка полягала у формуванні сцени з кількох примітивів і моделей, збереженні її в локальний файл, очищенні поточного стану та подальшому повторному відкритті. Методично важливо було встановити, чи зберігаються типи об’єктів, ідентифікатори, кольори, назви, просторові параметри та налаштування сцени, а також чи коректно відтворюються моделі через наявні *fileId* та *url*.

4.1.4 Критерії оцінювання результатів

Для оцінювання результатів експериментального дослідження використовувався набір функціональних і поведінкових критеріїв. Першим критерієм була функціональна коректність, тобто відповідність фактичної поведінки системи очікуваному результату для кожної окремої операції. Функціонально коректною вважалася така операція, після якої об’єкт сцени набуває належного стану на клієнті-ініціаторі, а серверне представлення сцени також переходить до відповідного *snapshot*-стану.

Другим критерієм була узгодженість між клієнтами. Цей критерій оцінювався шляхом одночасного порівняння сцен на кількох активних клієнтах після виконання одних і тих самих тестових сценаріїв. Ознакою успішної роботи вважалася відсутність тривалих розбіжностей у складі сцени, властивостях об’єктів, їх положенні, кольорі, назві та факті присутності імпортованих моделей. Для *undo/redo* додатково перевірялося, чи повертаються всі клієнти до одного й того самого стану після розсилки *snapshot*-відновлення.

Третім критерієм була стійкість до мережевих порушень і повторного підключення. Тут оцінювалося, чи не переходить клієнт у неконсистентний стан після тимчасової втрати зв’язку, чи коректно змінюється індикатор синхронізації, чи відновлюється актуальний *snapshot* після *reconnect* та чи відображаються подальші зміни без необхідності перезавантаження сторінки. Окремим аспектом цього критерію була перевірка *heartbeat*-механізму та наявності службових повідомлень *client_ping* і *client_pong*.

Четвертим критерієм була інтерактивність системи. Оскільки редактор орієнтований на безпосередню просторову взаємодію з об’єктами, важливою ознакою якості вважалася відсутність помітних зависань під час переміщення, обертання, масштабування, відкриття панелей інспектора, роботи з *mini-axis* і перемикання режимів *gizmo*. У межах роботи цей критерій оцінювався прикладним способом – за плавністю реакції інтерфейсу та відсутністю деструктивних затримок, які б робили нормальне редагування сцени незручним або неможливим.

П’ятим критерієм була цілісність даних у серверному сховищі. Для цього перевірялася відповідність між візуальною сценою та записами у колекціях *objects*, *operations* і *meta*, а для зовнішніх моделей – наявність відповідних файлів у GridFS. Такий критерій є особливо важливим для прототипу, де *undo/redo* базується не на абстрактному журналі трансформацій, а на реальних *snapshot*-станах, що зберігаються і відновлюються сервером.

4.1.5 Межі та припущення експериментальної постановки

Під час постановки експерименту враховувалися й обмеження поточної реалізації. По-перше, система досліджується як програмний прототип колаборативного редактора, а не як промислова платформа з кластеризованою серверною інфраструктурою та складним балансуванням навантаження. Тому головний акцент робиться на коректності архітектури, узгодженості стану, практичній працездатності механізмів синхронізації та стабільності в типовому багатокористувацькому режимі, а не на екстремальних навантажувальних тестах рівня промислової експлуатації.

По-друге, в експериментальній частині враховується реальна поведінка поточного клієнта: після втрати з’єднання він не накопичує повноцінну офлайн-чергу локальних операцій, а повторно синхронізується із сервером шляхом отримання актуального *snapshot* сцени. Саме тому оцінка відновлення після розриву з’єднання здійснюється в категоріях повторної синхронізації та відновлення консистентного стану, а не в категоріях автономного офлайн-редагування.

По-третє, імпорт GLTF/GLB-моделей оцінюється в межах підтримуваного протоколу завантаження файлів і їх подальшого відтворення. Якщо GLTF-модель залежить від зовнішніх бінарних або текстурних ресурсів, така конфігурація потребує додаткового супроводу на рівні серверного зберігання. Тому в базових експериментальних сценаріях пріоритет віддається самодостатнім GLB-файлам і тим GLTF-ресурсам, що можуть бути коректно відтворені в наявній реалізації.

Отже, підготовлена експериментальна постановка охоплює як функціональні, так і системні аспекти роботи редактора Collab3D. Вона дозволяє послідовно перевірити базові операції редагування, узгодженість багатокористувацької взаємодії, механізми збереження й відновлення стану, а також особливості роботи з імпортованими моделями та повторним підключенням клієнтів. Саме на основі цієї постановки в наступних підрозділах

буде здійснено функціональне тестування, перевірку синхронізації, а також аналіз продуктивності та стійкості системи.

4.2 Функціональне тестування основних можливостей редактора

Після завершення програмної реалізації ключових компонентів системи Collab3D було проведено функціональне тестування основних можливостей редактора. Метою цього етапу є перевірка коректності виконання базових користувацьких сценаріїв, відповідності фактичної поведінки редактора заявленій архітектурі та підтвердження працездатності основних інструментів редагування сцени. На відміну від подальших підрозділів, у яких окремо досліджуються багатокористувацька синхронізація та стійкість до розривів з’єднання, у цьому підрозділі акцент зроблено саме на локальній функціональній повноті редактора.

Функціональне тестування виконувалося за сценарним підходом. Для кожної групи можливостей формувався набір дій користувача, очікуваний результат та критерій успішності. Перевірка здійснювалася у браузерному середовищі з підключенням до серверної частини системи, яка забезпечує обслуговування HTTP-запитів, WebSocket-взаємодію та збереження стану сцени. Такий підхід дозволяє оцінити не лише інтерфейсну реакцію клієнта, а і коректність обміну даними з сервером у тих випадках, де функціональність безпосередньо пов’язана з мережевою взаємодією або серверним збереженням.

Першу групу перевірок становили функції створення та базового керування об’єктами сцени. У ході випробувань перевірялося додавання геометричних примітивів через верхнє меню «Додати», зокрема куба, сфери, ромба та трикутника. Для кожного створеного об’єкта контролювалися коректне відображення у вікні рендерингу, поява запису у списку об’єктів правої панелі, присвоєння внутрішнього ідентифікатора, а також можливість подальшого вибору. Окремо було протестовано видалення вибраного об’єкта, клонування та

зміну кольору через відповідні команди меню «Редагувати». Результати показали, що всі перелічені операції виконуються коректно, а стан інтерфейсу оновлюється узгоджено зі станом сцени.

Другу групу тестів було присвячено механізмам вибору та трансформації об’єктів. Для режиму переміщення перевірявся власний механізм *pointer-drag*, у якому перетягування об’єкта супроводжується відправленням проміжних повідомлень попереднього перегляду, а після завершення дії формується одна комітна операція зміни трансформації. Для режимів обертання та масштабування перевірялося використання *TransformControls*, відображення та приховування *gizmo*, коректне перемикання між режимами за допомогою кнопок і гарячих клавіш T, R, S, а також поведінка системи у випадку взаємодії безпосередньо з лініями gizmo. Додатково тестувалися перемикання між глобальним і локальним простором трансформацій, а також робота чекбоксів, що дозволяють окремо керувати видимістю *gizmo* для кожного режиму. У всіх перевірених сценаріях редактор демонстрував очікувану поведінку без втрати вибору та без небажаного перехоплення керування камерою під час роботи з *gizmo*.

Окремий блок становило тестування правої панелі редактора, яка виконує роль інспектора властивостей та навігаційного елемента. Перевірялося відображення списку об’єктів, підсвічування вибраного елемента, показ типу, UUID, псевдоніма й користувацької назви об’єкта, а також редагування трансформацій через числові поля інспектора. Для полів позиції, обертання та масштабу тестувалося пряме введення значень, зміна параметрів колесом миші, використання перетягування за мітки X, Y, Z, Δ, а також налаштування кроку зміни параметрів. Додатково було підтверджено працездатність кнопок згортання правої панелі, чекбоксів відображення сітки, світових осей, *mini-axis* та режиму інерції камери. Результати тестування засвідчили, що інтерфейсні засоби редагування властивостей працюють коректно та забезпечують користувачеві декілька взаємодоповнюючих способів керування станом об’єкта.

Наступний етап стосувався імпорту зовнішніх 3D-моделей і операцій збереження та відкриття сцени. Під час тестування імпорту перевірялося

завантаження файлів форматів GLB і GLTF через форму вибору файлу, передавання моделі на сервер, збереження її у GridFS, отримання URL доступу та подальше створення у сцені кореневого об’єкта типу *model*. Після імпорту контролювалися коректність візуалізації моделі, можливість її вибору, трансформації та зміни кольору шляхом тонування всіх вкладених *Mesh*-компонентів. Для функцій «Зберегти сцену» та «Відкрити сцену» перевірявся локальний файловий формат collab3d-scene, збереження параметрів сцени та масиву об’єктів, коректна валідація структури файлу під час відкриття і відновлення конфігурації редактора після імпорту збереженої сцени. У результаті підтверджено, що редактор забезпечує не лише інтерактивне редагування, а і відтворюваний цикл збереження та повторного відкриття проекту.

Завершальною групою функціональних випробувань стали перевірки механізму *undo/redo* та загальної цілісності сценаріїв редагування. Послідовно виконувалися операції створення, видалення, трансформації, зміни кольору, перейменування та імпорту моделі з подальшим скасуванням і повторним відтворенням дій. Особлива увага приділялася випадкам, коли після *undo* виконується нова дія, що має призводити до відсікання попередньої *redo*-гілки. Перевірка показала, що відновлення стану працює відповідно до прийнятої *snapshot*-моделі історії, а після *undo/redo* клієнтський інтерфейс, список об’єктів та вміст сцени залишаються узгодженими.

Для узагальнення результатів функціонального тестування наведено основні групи перевірок та підтверджені результати (див. табл. 4.1). Отримані результати свідчать, що на момент проведення дослідження система реалізує повний мінімально необхідний набір функцій колаборативного 3D-редактора для побудови, модифікації, локального збереження та подальшого відновлення сцени.

Отже, функціональне тестування підтвердило працездатність основних можливостей редактора Collab3D і показало, що реалізована система забезпечує коректне виконання базових операцій редагування тривимірної сцени. Це

створює підґрунтя для переходу до наступного етапу дослідження, у якому оцінюватиметься поведінка системи в умовах одночасної роботи кількох клієнтів і мережевих затримок.

Таблиця 4.1 – Основні групи функціональних випробувань редактора

Група перевірок	**Тестовані можливості**	**Підсумок**
Створення та базові дії	Додавання примітивів, вибір, видалення, клонування, зміна кольору	Підтверджено
Трансформації	Переміщення, обертання, масштабування, *gizmo*, гарячі клавіші, глобальний і локальний простір	Підтверджено
Інспектор і панель	Список об’єктів, редагування назви, числові поля, кроки змін, згортання панелі	Підтверджено
Візуальні інструменти	Сітка, світові осі, *mini-axis*, інерція камери, індикація стану	Підтверджено
Імпорт і файлові операції	Завантаження GLB/GLTF, тонування моделей, збереження та відкриття сцени	Підтверджено
Історія дій	*Undo/redo* для основних операцій, обрізання *redo*-гілки після нової дії	Підтверджено

4.3 Перевірка синхронізації при одночасній роботі кількох клієнтів

Після завершення функціонального тестування окремих можливостей редактора було проведено окремий етап експериментальної перевірки синхронізації при одночасній роботі кількох клієнтів. Метою цього етапу було підтвердження того, що система Collab3D забезпечує узгоджене відображення стану сцени в різних клієнтських сеансах, коректну передачу операцій у реальному часі та відновлення актуального стану після повторного підключення.

Перевірка виконувалася у сценарному режимі. До сервера одночасно підключалися два або більше клієнтів, відкритих у різних вікнах браузера або на різних пристроях, після чого в одному з них послідовно виконувалися дії редагування сцени. В інших клієнтах фіксувалося, чи відображаються ці зміни без ручного оновлення сторінки, чи зберігається однакова структура списку об’єктів, а також чи не виникають розбіжності в положенні, кольорі, назві та типі об’єктів.

Перший сценарій стосувався початкової синхронізації нового клієнта. Після створення кількох примітивів, зміни їх положення та кольору до системи підключався додатковий клієнт. Сервер надсилав йому повний *snapshot* поточного стану сцени, унаслідок чого новий учасник одразу отримував узгоджене представлення редактора: об’єкти відображалися у правильних координатах, список правої панелі формувався коректно, а імпортовані моделі завантажувалися за збереженим URL.

Другий блок перевірок був присвячений поширенню базових операцій редагування між клієнтами. У ході випробувань перевірялося додавання примітивів, видалення об’єктів, зміна кольору, перейменування та імпорт GLB/GLTF-моделей. Результати показали, що кожна журнальована операція після прийому сервером відображалася в інших клієнтах без необхідності ручного втручання. Це підтверджує коректність вибраної централізованої схеми, за якої сервер виступає єдиним джерелом істини для всіх підключених учасників.

Окрему увагу було приділено перевірці безперервних трансформацій об’єктів. Під час переміщення, обертання та масштабування один клієнт генерував проміжні повідомлення *transform_preview*, що дозволяло іншим клієнтам спостерігати рух об’єкта майже в реальному часі. Після завершення взаємодії сервер отримував комітну операцію *transform* із параметрами *before* та *after*, а підсумковий стан фіксувався в журналі та історії. Такий підхід показав свою ефективність: візуальна плавність синхронізації поєднувалася зі збереженням компактної історії змін для *undo/redo*.

Наступний сценарій стосувався конкурентних дій кількох учасників. Якщо різні клієнти одночасно працювали з різними об’єктами, усі зміни успішно інтегрувалися в єдиний стан сцени без взаємного блокування. У випадках, коли кілька операцій майже одночасно стосувалися одного й того самого об’єкта, остаточний стан визначався серверним порядком прийому повідомлень. Практично це означає, що всі клієнти після завершення обміну бачили однаковий фінальний результат, навіть якщо локально дії були ініційовані майже синхронно.

Окремим напрямом перевірки стало тестування глобальних операцій *undo* та *redo*. Якщо один клієнт ініціював *undo_request* або *redo_request*, сервер змінював покажчик історії, відновлював відповідний *snapshot* та розсилав повний стан сцени всім підключеним учасникам. Унаслідок цього всі клієнти одночасно переходили до однакового стану сцени. Це особливо важливо для колаборативного режиму, оскільки механізм *undo/redo* в даній архітектурі діє не локально, а на рівні спільного серверного стану.

Завершальною частиною випробувань була перевірка поведінки системи після тимчасового розриву з’єднання одного з клієнтів. Після втрати WebSocket-з’єднання клієнт переходив у режим перепідключення, а після відновлення каналу повторно отримував актуальний *snapshot* сцени та синхронізувався з сервером. Проведені перевірки показали, що така модель коректно відновлює актуальний стан редактора без повноцінної локальної офлайн-черги операцій. Отже, система забезпечує саме повторну синхронізацію після *reconnect*, а не автономне накопичення змін із подальшим їх злиттям.

Для систематизації результатів перевірки синхронізації наведено основні тестові сценарії, очікувані ефекти та фактичні підсумки випробувань (див. табл. 4.2).

Таким чином, проведена перевірка підтвердила, що реалізований у Collab3D механізм синхронізації забезпечує узгоджену спільну роботу кількох користувачів над однією сценою. Система коректно поширює журнальовані операції, відображає безперервні трансформації в режимі реального часу,

підтримує глобальний *undo/redo* та забезпечує повторну синхронізацію після відновлення з’єднання. Отримані результати дають підстави вважати архітектуру та протокол взаємодії придатними для подальшого практичного використання й розширення.

Таблиця 4.2 – Результати перевірки синхронізації при одночасній роботі кількох клієнтів

Сценарій	**Очікуваний результат**	**Підсумок**
Підключення нового клієнта	Після з’єднання клієнт отримує повний *snapshot* і бачить актуальну сцену.	Підтверджено
Додавання та видалення об’єктів	Створення або видалення об’єктів в одному клієнті автоматично відображається в інших.	Підтверджено
Зміна кольору та перейменування	Операції color і rename поширюються на всі клієнти без ручного оновлення.	Підтверджено
Безперервні трансформації	Проміжні transform_preview показують рух у реальному часі, а transform фіксує підсумок.	Підтверджено
Конкурентні дії над різними об’єктами	Операції інтегруються без конфліктів, а кінцевий стан сцени є однаковим.	Підтверджено
Глобальні *undo/redo*	Після *undo_request* або *redo_request* усі клієнти отримують однаковий *snapshot*.	Підтверджено
Повторне підключення	Після reconnect клієнт повторно отримує актуальний *snapshot* і відновлює синхронізацію.	Підтверджено

4.4 Аналіз продуктивності, стійкості та відновлення після розриву з’єднання

Після перевірки базової функціональності редактора та коректності синхронізації між кількома клієнтами наступним етапом експериментального дослідження став аналіз продуктивності, стійкості роботи системи та механізмів

відновлення після порушення з’єднання. Метою цього етапу було оцінити, наскільки реалізована архітектура Collab3D зберігає працездатність за умов тривалої взаємодії, багаторазового обміну повідомленнями, імпорту моделей та тимчасових збоїв мережевого каналу.

Оцінювання продуктивності виконувалося на рівні основних сценаріїв роботи користувача: створення та видалення об’єктів, зміна кольору, перейменування, переміщення, обертання, масштабування, імпорт GLB/GLTF-моделей, а також збереження і відновлення сцени. Окремо враховувалися особливості поточної реалізації: поділ повідомлень на журналюванні операції та проміжні transform_preview, збереження канонічного стану сцени в колекції *objects*, журналу подій в *operations*, історії *undo/redo* в meta та файлів моделей у GridFS. Така організація дозволяє рознести навантаження між різними підсистемами та зменшити кількість дорогих операцій під час безперервних трансформацій.

У ході експериментів було встановлено, що найбільш типовими для користувача є короткі операції редагування окремого об’єкта, для яких система демонструє стабільну реакцію інтерфейсу. Додавання примітивів, видалення, зміна кольору та перейменування не потребують складних локальних обчислень на клієнті й після прийому сервером майже одразу стають видимими для інших учасників сеансу. Це пояснюється тим, що для таких сценаріїв передається компактне JSON-повідомлення, а сервер виконує обмежену кількість змін у поточному *snapshot* сцени та одне широкомовне розсилання.

Більш показовим для оцінювання продуктивності є режим безперервних трансформацій. У реалізації Collab3D для переміщення, обертання та масштабування використано дворівневу схему передавання стану. Під час активної взаємодії клієнт періодично надсилає *transform_preview*, що дозволяє іншим підключеним клієнтам спостерігати поточний рух об’єкта майже в реальному часі. Після завершення дії передається одна комітна операція *transform* із полями *before* та *after*. Практичний результат такої схеми полягає в тому, що мережевий канал і сервер не перевантажуються надмірною кількістю

журнальованих записів, тоді як візуальна плавність взаємодії зберігається на прийнятному рівні.

Серверна підсистема також виявила достатній рівень стійкості для дослідницького прототипу. Node.js забезпечує неблокуючу модель обробки запитів, а WebSocket-сервер працює поверх того самого HTTP-сервера, що й маршрути видачі клієнтських файлів, health-check і модулів імпорту. Для контролю активності з'єднань на сервері реалізовано *heartbeat*-механізм: через фіксований інтервал виконуються ping-запити, а клієнти, що не відповіли після допустимої кількості пропусків, примусово відключаються. Це запобігає накопиченню «завислих» WebSocket-сеансів та позитивно впливає на загальну стабільність роботи сервера під час тривалих сесій.

З боку клієнта стійкість системи посилюється за рахунок окремого механізму моніторингу доступності сервера. Застосунок періодично виконує HTTP-перевірку маршруту */health*, підтримує цикл *client_ping*/*client_pong* і відображає поточний стан синхронізації через індикатор *online*, *reconnecting* або *offline*. У практичному використанні це дає змогу не лише автоматично виявляти проблеми зі зв'язком, а й робити поведінку системи зрозумілішою для користувача: у разі втрати з'єднання редактор не «зависає» без пояснення, а явно показує перехід у стан перепідключення або автономного режиму.

Окремий блок випробувань був присвячений відновленню після короткочасного розриву з'єднання. Після імітації відключення мережі або примусового завершення WebSocket-сеансу клієнт переходив у режим *reconnecting*, після чого намагався повторно встановити з'єднання з наростаючою затримкою між спробами. Після успішного перепідключення сервер одразу надсилав повний *snapshot* актуального стану сцени, і локальний стан клієнта перебудовувався відповідно до канонічних даних із сервера. Це дозволяло коректно відновити склад сцени, просторові параметри об'єктів, кольори, назви, а також посилання на імпортовані моделі.

Суттєвою перевагою поточної реалізації є те, що відновлення після розриву з'єднання не вимагає повторного програвання всього журналу операцій

від початку. Джерелом істини слугує поточний *snapshot*, що зберігається в колекції *objects*, а глобальна історія *undo/redo* підтримується окремо через *meta*. Завдяки цьому повторне підключення клієнта зводиться до отримання одного узгодженого знімка стану сцени. Такий підхід значно спрощує відновлення та знижує ризик накопичення похибок, характерних для довгих ланцюжків інкрементних оновлень.

Разом із тим експериментальне дослідження виявило й обмеження поточного прототипу. Система не реалізує повноцінний офлайн-режим із локальною чергою непідтверджених операцій. Якщо в момент втрати з'єднання користувач ініціює дію, яка не може бути доставлена на сервер, така дія не інтегрується автоматично після *reconnect*. Отже, механізм відновлення в поточній версії слід трактувати як повторну синхронізацію клієнта з актуальним серверним станом, а не як прозоре злиття автономно накопичених локальних змін. Для дослідницького редактора цей підхід є виправданим, однак у перспективі може бути розширений шляхом введення локальної черги операцій і додаткового протоколу підтверджень.

Таким чином, результати дослідження показали, що реалізована система забезпечує прийнятний рівень продуктивності для колаборативного редагування 3D-сцен малого та середнього обсягу, демонструє стійку роботу під час тривалих сеансів та коректно відновлює актуальний стан після розриву з'єднання. Використання *snapshot*-підходу, розмежування між *transform_preview* і комітними операціями, *heartbeat*-контролю та клієнтського механізму перепідключення дозволяє підтримувати практичну працездатність системи навіть за наявності короткочасних мережевих збоїв.

4.5 Висновки до розділу 4

У четвертому розділі було проведено експериментальне дослідження та тестування розробленої системи Collab3D з метою перевірки її працездатності,

функціональної повноти, коректності синхронізації та стійкості до типових експлуатаційних ситуацій. Запропонована методика дала змогу оцінити систему не лише як програмний прототип, а і як цілісне колаборативне веб-застосування, у якому поєднуються клієнтська візуалізація 3D-сцени, серверна інтеграція операцій, збереження стану сцени та відновлення після розриву з'єднання.

Результати функціонального тестування показали, що реалізований редактор коректно підтримує основні сценарії роботи користувача: створення примітивів, імпорт зовнішніх GLB/GLTF-моделей, вибір об'єктів, зміну положення, обертання і масштабу, редагування кольору та назви, використання правої панелі, інспектора властивостей, *mini-axis*, а також локальне збереження і відкриття сцени. Окремо було підтверджено працездатність механізму *undo/redo*, який у поточній реалізації спирається на історію *snapshot*-станів сцени, а не на повторне програвання інверсних команд.

Перевірка багатокористувацьких сценаріїв засвідчила, що централізована серверна серіалізація операцій забезпечує узгоджене відображення стану сцени для всіх підключених клієнтів. Передавання комітних змін через WebSocket-повідомлення та використання окремого класу ефемерних повідомлень *transform_preview* дали змогу поєднати коректність фінального стану з достатньо плавним візуальним відображенням безперервних трансформацій у режимі реального часу. Це підтверджує практичну придатність обраної архітектурної схеми для колаборативного редагування 3D-сцен.

Оцінювання продуктивності і стійкості показало, що для дослідницького прототипу система демонструє прийнятний рівень швидкодії та надійності. Використання неблокуючої серверної моделі Node.js, WebSocket-з'єднання, *heartbeat*-контролю, клієнтських *ping/pong*-повідомлень, періодичної перевірки маршруту */health* і автоматичного перепідключення створює достатній запас стійкості для типових сценаріїв використання. У разі короткочасного розриву зв'язку клієнт після повторного підключення отримує актуальний *snapshot* сцени і синхронізує локальний стан із сервером без потреби в ручному втручанні користувача.

Разом із тим результати розділу 4 дозволили виявити і наявні обмеження поточної версії системи. До них належать відсутність повноцінного офлайн-режиму з локальною чергою непідтверджених операцій, *snapshot*-орієнтований характер *undo/redo*, що збільшує вартість комітної операції зі зростанням сцени, а також залежність стабільності від параметрів мережевого середовища і серверного розгортання. Крім того, у поточному прототипі не реалізовано ієрархічні структурні операції на кшталт reparent, тому система орієнтована насамперед на синхронізацію верхньорівневих об’єктів сцени.

Отже, проведене експериментальне дослідження підтвердило, що розроблена система Collab3D загалом виконує поставлені функції та забезпечує коректне колаборативне редагування тривимірних сцен у веб-середовищі. Отримані результати дають підстави вважати запропоновану архітектуру, структуру даних і механізми синхронізації обґрунтованими для навчально-дослідницького прототипу та придатними для подальшого розвитку в напрямках оптимізації продуктивності, розширення набору підтримуваних операцій і поглиблення механізмів відмовостійкості.

ВИСНОВКИ

У магістерській роботі вирішено актуальне науково-практичне завдання, що полягало у розробленні колаборативного веб-редактора тривимірних сцен на базі сучасних веб-технологій із підтримкою синхронізації змін у режимі реального часу. Актуальність виконаного дослідження зумовлена зростанням потреби у засобах спільного редагування просторових даних, які поєднують наочність 3D-подання, доступність через браузер та можливість одночасної роботи кількох користувачів над єдиною сценою.

У ході роботи було проаналізовано особливості колаборативних систем реального часу, підходи до представлення тривимірних сцен у веб-середовищі та сучасні методи синхронізації стану в розподілених застосунках. Показано, що для задачі багатокористувацького редагування 3D-сцен особливо важливими є проблеми узгодженості конкурентних змін, некомутативності просторових трансформацій, високої частоти оновлень під час інтерактивної взаємодії та необхідності відновлення актуального стану після тимчасової втрати з’єднання.

На основі проведеного аналізу було побудовано формальну модель тривимірної сцени, журналу операцій і серверної функції переходу між станами. У межах реалізованого прототипу сцену подано як множину верхньорівневих колаборативних об’єктів із фіксованим набором атрибутів, а множину підтримуваних операцій становлять створення, видалення, трансформація, зміна кольору, перейменування та імпорт зовнішніх моделей. Окремо обґрунтовано доцільність використання ефемерних повідомлень попереднього перегляду трансформацій, які забезпечують плавність сприйняття змін без перевантаження журналу операцій та історії undo/redo.

У роботі спроектовано та реалізовано клієнт-серверну архітектуру системи. Клієнтську підсистему побудовано на базі Three.js з використанням WebGL для візуалізації сцени, механізмів вибору об’єктів, режимів переміщення, обертання і масштабування, а також розвиненого користувацького

інтерфейсу з верхнім меню, правою панеллю, списком об’єктів, інспектором властивостей, mini-axis та засобами локального збереження й відкриття сцени. Серверну підсистему реалізовано на платформі Node.js із використанням HTTP та WebSocket, що дало змогу поєднати завантаження ресурсів, обробку операцій і розповсюдження змін між клієнтами в межах єдиного застосунку.

Важливим результатом роботи є реалізація механізму збереження даних, у якому MongoDB використовується не лише як сховище журналу операцій, а як комплексна підсистема підтримки стану сцени. Колекція objects зберігає поточний snapshot сцени, operations – журнал застосованих операцій, meta – історію станів і покажчик для стабільної роботи undo/redo, а GridFS забезпечує довготривале збереження імпортованих GLB- та GLTF-моделей. Такий підхід підвищив надійність системи, спростив повторну синхронізацію після reconnect і зробив можливим відновлення коректного стану навіть після перезапуску серверної частини.

В процесі розробки колаборативного веб-редактора 3D-сцен були використані хмарні ресурси, на яких було розміщено: сам програмний код застосунка, зовнішній сервер з HTTP, WebSocket сервер, Node.js, Express, ws, сервер бази даних з Mongo DB, GridFS. А також арендований домен *collab3D* на платформі *render.com* (див. додаток А).

Експериментальне дослідження та функціональне тестування підтвердили працездатність розробленої системи. Установлено, що редактор коректно підтримує створення та редагування примітивів, імпорт зовнішніх моделей, зміну параметрів трансформації, кольору й назви об’єктів, збереження та відкриття сцен, а також синхронізацію змін між кількома клієнтами в режимі реального часу. Окремо підтверджено коректність механізмів undo/redo та працездатність сценарію повторного підключення, за якого клієнт повторно отримує актуальний snapshot сцени і вирівнює локальний стан із серверним.

Практичне значення отриманих результатів полягає у створенні працездатного веб-орієнтованого прототипу колаборативного 3D-редактора, який може бути використаний як основа для подальшого розвитку систем

дистанційної спільної роботи з просторовими даними, навчальних середовищ, засобів демонстрації тривимірних проектів та спеціалізованих редакторів предметної області. Запропоновані рішення можуть бути адаптовані до ширшого класу багатокористувацьких інтерактивних застосунків, де необхідно поєднати візуальну складність сцени, роботу в браузері та узгодження змін у реальному часі.

Перспективи подальшого розвитку роботи пов’язані з розширенням функціональних можливостей системи. До таких напрямів належать підтримка ієрархічних структур редагування з окремими операціями на кшталт reparent, удосконалення механізмів вирішення конфліктів для складніших сценаріїв конкурентної взаємодії, реалізація повноцінної офлайн-черги локальних операцій, розширення набору підтримуваних форматів моделей, оптимізація масштабованості серверної частини та поглиблення інструментів колективної взаємодії. Отже, поставлену мету роботи досягнуто, а сформульовані завдання виконано у повному обсязі.

ПЕРЕЛІК ПОСИЛАНЬ

1. Ellis C. A., Gibbs S. J. Concurrency control in groupware systems. *Proceedings of the ACM SIGMOD Conference on Management of Data*. 1989. P. 399–407. DOI: 10.1145/67544.66963.

2. Sun C., Chen D. Consistency maintenance in real-time collaborative graphics editing systems. *ACM Transactions on Computer-Human Interaction*. 2002. Vol. 9, no. 1. P. 1–41. DOI: 10.1145/505151.505152.

3. Shapiro M., Preguiça N., Baquero C., Zawirski M. Conflict-Free Replicated Data Types. *Stabilization, Safety, and Security of Distributed Systems : 13th International Symposium, SSS 2011. Berlin ; Heidelberg : Springer, 2011*. P. 386–400. DOI: 10.1007/978-3-642-24550-3_29.

4. Preguiça N. Conflict-free Replicated Data Types (CRDTs) . 2018. URL : https://arxiv.org/abs/1805.06358 (дата звернення : 05.06.2025).

5. Fette I., Melnikov A. The WebSocket Protocol. RFC 6455. *Internet Engineering Task Force, 2011*. URL : https://datatracker.ietf.org/doc/html/rfc6455 (дата звернення : 05.06.2025).

6. MDN Web Docs. WebGL API. URL : https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API (дата звернення : 07.06.2025).

7. MDN Web Docs. WebGLRenderingContext. URL : https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext (дата звернення : 10.06.2025).

8. Three.js Documentation. URL : https://threejs.org/docs/ (дата звернення : 11.06.2025).

9. Three.js. Scene. URL : https://threejs.org/docs/pages/Scene.html (дата звернення : 11.06.2025)

10. Three.js. GLTFLoader. URL : https://threejs.org/docs/pages/GLTFLoader.html (дата звернення : 11.06.2025).

11. Khronos Group. glTF™ 2.0 Specification. URL : https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html (дата звернення : 15.04.2025).

12. ISO. ISO/IEC 12113:2022 Information technology – Runtime 3D asset delivery format. URL : https://www.iso.org/standard/83990.html (дата звернення : 12.04.2025)

13. Node.js Documentation. URL : https://nodejs.org/api/index.html (дата звернення : 15.06.2025).

14. Express.js. API Reference. URL : https://expressjs.com/en/api.html (дата звернення : 15.06.2025).

15. Express.js. Routing Guide. URL : https://expressjs.com/en/guide/routing.html (дата звернення : 15.06.2025).

16. MongoDB. GridFS – Database Manual. URL : https://www.mongodb.com/docs/manual/core/gridfs/ (дата звернення : 17.06.2025).

17. MongoDB. Store Large Files with GridFS – Node.js Driver. URL : https://www.mongodb.com/docs/drivers/node/current/crud/gridfs/ (дата звернення : 17.06.2025).

18. MongoDB. Limits and Thresholds. URL : https://www.mongodb.com/docs/manual/reference/limits/ (дата звернення : 17.06.2025).

19. Основи WebGL – україномовний матеріал WebGL Fundamentals. URL : https://webglfundamentals.org/webgl/lessons/uk/ (дата звернення : 21.06.2025).

20. WebSocket – розділ з українського javascript.info. URL : https://uk.javascript.info/websocket (дата звернення : 25.06.2025).

21. Node.js Підручник на W3Schools. URL : https://w3schoolsua.github.io/nodejs/index.html#gsc.tab=0 (дата звернення : 01.07.2025).

22. MongoDB. Підручник та API запитів MongoDB на W3Schools. URL : https://w3schoolsua.github.io/mongodb/index.html#gsc.tab=0 (дата звернення : 01.07.2025).

23. Інтеграція MongoDB та Node.js – фахова публікація. URL : https://dspace.khadi.kharkov.ua/items/b6c88767-eeef-4fa5-96d5-e0e22394f004 (дата звернення : 05.07.2025).

24. Деякі аспекти використання Node.js та MongoDB під час створення освітніх веб-застосунків –- наукова стаття. URL : https://infotech-soccult.knukim.edu.ua/article/view/283958 (дата звернення : 10.07.2025)

ДОДАТОК А

Використані платформи при розробці колаборативного 3D-редактора

1. Платформа розміщення програмного коду 3D-редактора : https://github.com/cgushenka762/collab3D.git.

2. Платформа розміщення зовнішнього сервера з HTTP + WebSocket сервер + Node.js + Express + ws : https://dashboard.render.com/web/srv-d70jlc4hg0os73adb29g.

3. Платформа розміщення сервера БД : https://cloud.mongodb.com/v2/69c158a0525475092d441ba3#/explorer/69c15c5355fdaf0d623fa5e3/collab3d/models.files/find.

4. Платформа розміщення робочої версії 3D-редактора : https://collab3d.onrender.com/ (див. рис. А.1).

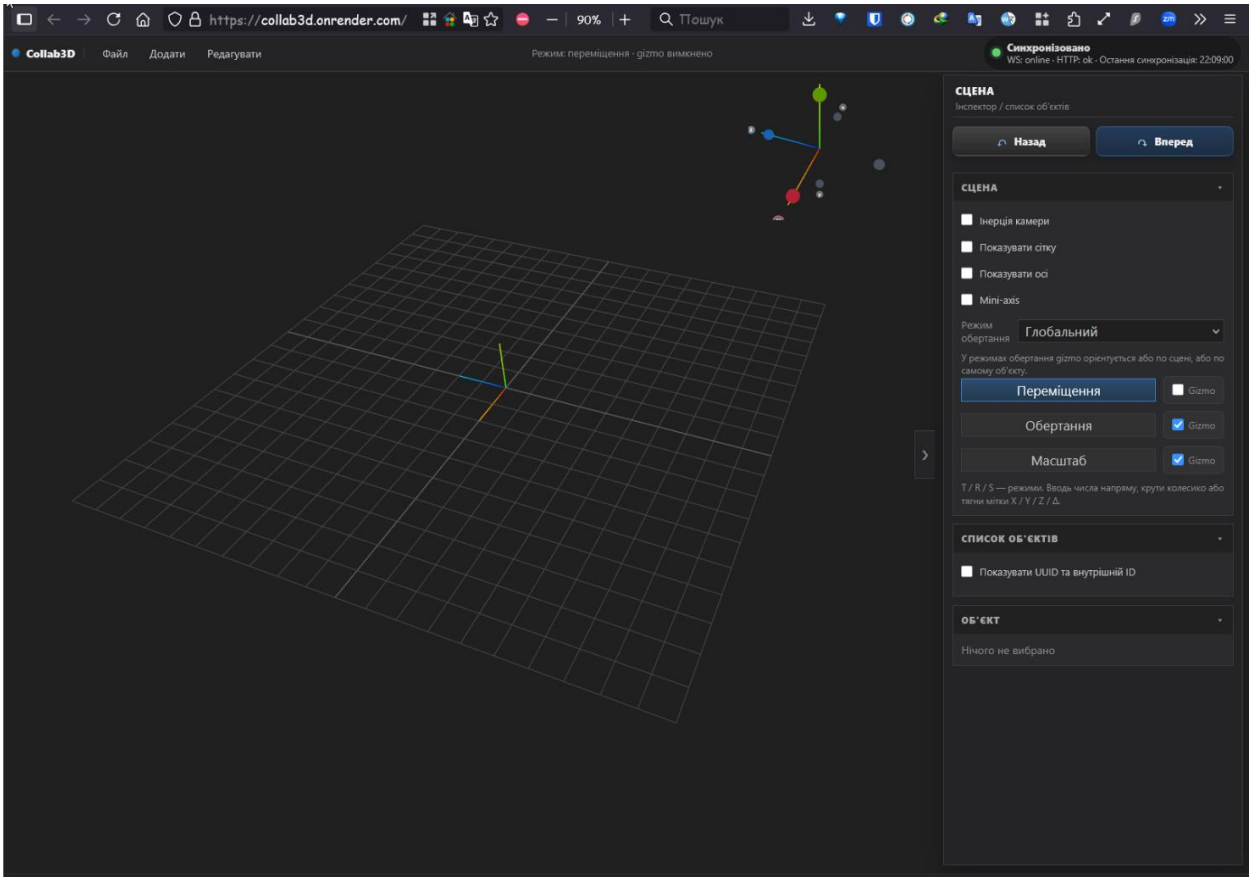


Рисунок А.1 – Скріншот головного вікна 3D-редактора