МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

перший (бакалаврський)
(рівень вищої освіти)

на тему **Розробка веб-застосунку для імпорту, візуалізації та експорту 3D-об'єктів різних форматів на базі WebGL**

Виконав: студент 4 курсу, групи 6.1211-ПЗС
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Програмне забезпечення систем
(код і назва освітньої програми)
С.О. Скрипак
(ініціали та прізвище)
Керівник доцент, А.І. Безверхий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ «Дискус»
Р.О. Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ім. Ю.М. Потебні ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра електроніки, інформаційних систем та програмного забезпечення
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 Інженерія програмного забезпечення
(код та назва)
Освітня програма Програмне забезпечення систем
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри Тетяна КРИТСЬКА

“ 01 ” березня 2025 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Скрипаку Степанові Олеговичу
(прізвище, ім’я, по батькові)

1. Тема роботи Розробка веб-застосунку для імпорту, візуалізації та експорту 3D-об'єктів різних форматів на базі WebGL
керівник роботи Безверхий Анатолій Ігорович, доцент,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ЗНУ від 27.12.2024 р. № 2029-с
2. Строк подання студентом кваліфікаційної роботи 02.06.2025
3. Вихідні дані бакалаврської роботи
   - комплект нормативних документів ;
   - технічне завдання до роботи.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
   - огляд та збір літератури стосовно теми кваліфікаційної роботи;
   - огляд та аналіз існуючих рішень та аналогів;
   - дослідження проблеми обміну 3D-даними між різними форматами;
   - створення програмного продукту та його опис;
   - перелік вимог для роботи програми;
   - розробка висновків та пропозицій.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень) слайдів презентації

6. Консультанти розділів бакалаврської роботи

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

7. Дата видачі завдання 01.03.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів магістерської роботи	Примітка
1.	Аналіз предметної області.	20.03-24.03	виконано
2.	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником.	25.03	виконано
3.	Аналіз програм-аналогів.	26.03-29.04	виконано
4.	Вибір технологій для реалізації.	01.04-02.04	виконано
5.	Аналіз існуючих форматів 3D-об'єктів.	02.04-25.04	виконано
6.	Проектування внутрішнього формату-посередника.	02.04-05.04	виконано
7.	Побудова архітектури застосунку.	06.04-08.04	виконано
8.	Розробка модулів імпорту для оглянутих форматів.	08.04-16.05	виконано
9.	Реалізація системи візуалізації 3D-об'єктів за допомогою Three.js.	10.04-11.04	виконано
10.	Розробка користувацького інтерфейсу.	12.04-13.04	виконано
11.	Розробка модулів експорту для оглянутих форматів.	08.04-16.05	виконано
12.	Розширення функціоналу застосунку переглядачем мета-інформації.	14.05-16.05	виконано
13.	Рефакторинг коду.	16.05-22.05	виконано
14.	Системне тестування імпорту/візуалізації/експорту	23.05-28.05	виконано
15.	Оформлення звіту.	29.05-06.05	виконано
16.	Оформлення презентації і отримання рецензії.	07.05-13.06	в процессі

Студент _____ _____С.О. Скрипак

Керівник роботи _____ _____А. І. Безверхий

Нормоконтроль пройдено

Нормоконтролер _____ _____І.А. Скрипник

АНОТАЦІЯ

Сторінок — 86,

Рисунків — 21,

Джерел — 19.

Скрипак С.О. Розробка веб-застосунку для імпорту, візуалізації та експорту 3D-об'єктів різних форматів на базі WebGL кваліфікаційна робота бакалавра спеціальності 121 «Програмне забезпечення систем» / наук. керівник А.І. Безверхий Запоріжжя : ЗНУ, 2025. 86 с.

У сучасному світі, завдяки поширенню 3D-графіки, актуальним є питання конвертації 3D-моделей між різними форматами файлів. Більшість існуючих рішень орієнтовані на досвідчених користувачів 3D-графіки, а альтернативи не можуть забезпечити швидкодію чи безпеку даних, на яку здатна система користувача.

Метою роботи є створення швидкого та зручного веб-застосунку, розрахованого на користувачів без глибоких знань у 3D-графіці, який дозволяє імпортувати, візуалізувати та конвертувати популярні формати 3D-файлів. Основне завдання — спрощення та оптимізація процесу конвертації між різними 3D-форматами.

Для цього проведено аналіз найпоширеніших форматів збереження 3D-файлів і розроблено власний формат-посередник для оптимізації процесу конвертації. Реалізовано веб-застосунок із клієнтською архітектурою, який використовує обчислювальні потужності браузера клієнта для виконання всіх операцій конвертації. Створено алгоритми десеріалізації найпоширеніших форматів у формат посередник, та їх серіалізацію у вихідні формати.

Ключові слова: *3D-моделі, WebGL, Three.js, конвертація форматів, формат-посередник, огляд форматів 3D-моделей*

ABSTRACT

Pages — 86,

Drawings — 21

Sources — 19.

Skrypak S.O. Development of a web application for importing, visualizing and exporting 3D objects of various formats based on WebGL qualification thesis of the bachelor of specialty 121 "System software" / Science. Manager A.I. Bezverkhyi Zaporizhzhia: ZNU, 2025. 86 p

In the modern world, with the widespread use of 3D graphics, the issue of converting 3D models between different file formats has become increasingly relevant. Most existing solutions are designed for experienced 3D graphics users, while others offer complex or insecure services that require uploading models to a server.

The aim of this project is to develop a fast and user-friendly web application targeted at users with limited knowledge of 3D graphics, enabling them to import, visualize, and convert popular 3D file formats. The primary objective is to simplify and optimize the process of converting between various 3D formats.

To achieve this, an analysis of the most common 3D file formats was conducted, and an intermediary format was developed to streamline the conversion process. A web application with a client-side architecture was implemented, utilizing the computational capabilities of the client’s browser for all conversion operations. Algorithms were created to deserialize the most common formats into the system’s internal representation and to serialize them into target formats.

Keywords: *3D models, WebGL, Three.js, format conversion, intermediary format, analysis of 3D model formats.*

ЗМІСТ

ВСТУП 8
1 АНАЛІЗ СУЧАСНИХ 3-D ТЕХНОЛОГІЙ 12
1.1 Історія та огляд сучасних 3D-технологій 12
1.2 Структура 3D-об'єктів 15
1.3 Матеріали та текстури у 3D-графіці реального часу 16
1.4 Структура та особливості форматів 3D-файлів 18
1.4.1 Формат OBJ 19
1.4.2 Формат STL 21
1.5.3 Формат PLY 23
1.4.4 Формат GLTF / GLB 25
1.4.5 Формат COLLADA 30
2 АНАЛІЗ ТЕХНОЛОГІЙ ТА ІНСТРУМЕНТІВ 35
2.1 Огляд існуючих рішень 35
2.3 Особливості 3D у веб середовищі 41
2.4 Особливості роботи з Three.js 42
3 РОЗРОБКА ЗАСТОСУНКУ ДЛЯ ІМПОРТУ ТА ЕКСПОРТУ 3-D ФАЙЛІВ 44
3.1 Опис проблеми 44
3.2 Вимоги до програмної системи 45
3.3. Архітектура програмної системи 46
3.2 Розбір компоненту Model 48
Компонент модель не тільки надає доступ до даних, але й виступає контейнером їх збереження. Розберемо модулі, що існують в цьому компоненті. Їх повний перелік надано на рис. 7. 48
3.2.1 Внутрішнє представлення даних 49
3.2.2 Імпорт 3D-моделей 55
3.2.3 Експорт 3D-моделей 62
3.3 Розбір компоненту View 65

3.4 Розбір компоненту Controller 73
3.5 Тестування застосунку 77
3.6 Висновки з розділу 3 83
ВИСНОВКИ 84
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 85

ВСТУП

Актуальність теми

Швидкий розвиток 3D-технологій, оптимізації 3D-рендерингу, покращення користувацьких комп'ютерів створюють ідеальне середовище до інтеграції інтерактивних трьох вимірних моделей у веб-застосунки. Сьогодні 3D графіка активно застосовується майже усюди: розвагах, інтернет магазинах, дизайну, науковій візуалізації, архітектурі. Завдяки цьому 3D-моделі стають дедалі доступнішими до розробників, любителів і користувачів.

Однак з поширенням 3D об'єктів виникає проблема їх не універсальності. Проблема відсутності глобальної стандартизації щодо формату збереження трьох вимірних об'єктів. Сучасний ринок містить велику кількість форматів 3D-файлів, внутрішня структура яких відрізняється. Через те, що кожний зберігає трьох-вимірну інформацію по різному – часто виникають проблеми імпорту файлів певного формату в певні застосунки. Цю проблему частково вирішують популярні формати накштал OBJ, FBX, GLTF/GBL, що мають майже універсальну підтримку у багатьох застосунках для роботи з 3D графікою. Саме тому більшість файлів онлайн розповсюджується саме в таких форматах. Але виникають ситуації, коли моделі розповсюджуються в інших форматах, чи програма просто не підтримує багато форматів.

Це створює нову проблему конвертації між 3D-файлами. Це можна вирішити використавши універсальні застосунки для роботи з 3D-графікою, такі як, наприклад, Blender чи 3DS MAX – але не всі користувачі мають можливість установити та запустити їх, чи є недостатньо обізнаними для праці у застосунків такого формату.

Актуальним стає створення універсального інструменту, який дозволить користувачам легко завантажувати, переглядати, на експортувати 3D-моделі різних форматів без потреби в додатковому програмному забезпечені.

Мета дослідження

Створення простого веб-застосунку, що дозволить користувачам завантажувати, переглядати, та експортувати 3D-моделі в декількох форматах.

Завдання дослідження

Аналіз сучасних форматів 3D-об’єктів, виявлення їх особливостей і проблем з конвертацією. Аналіз WebGL та доступних фреймворків для візуалізації 3D-об’єктів. Розробка архітектури веб-застосунку для ефективної обробки та візуалізації 3D-об’єктів. Реалізувати функціонал імпорту, обробки, і експорту 3D-об’єктів з підтримкою проаналізованих форматів. Забезпечити безпечну конвертацію між форматами.

Об’єкт дослідження

Об’єктом дослідження є внутрішня структура файлів 3D-об’єктів різних форматів.

Предмет дослідження

Предметом дослідження є методи та інструменти для імпорту, обробки, візуалізації та експорту 3D-об’єктів у різних форматах у середовищі веб-браузера.

Методи дослідження

Аналіз наукової та технічної літератури з внутрішньої структури файлів 3D-об’єктів. Аналіз можливостей трьох вимірної графіки в веб середі, порівняння бібліотек для роботи з 3D-графікою. Проектування архітектури веб-застосунку та реалізація конвертації між форматами 3D-об’єктів.

Практичне значення одержаних результатів

Практичне значення одержаних результатів полягає у створенні універсального веб-застосунку, який дозволить користувачам легко імпортувати, переглядати, і перетворювати файли 3D-об’єктів на інші формати. Результати дослідження можуть бути використані для створення подібних веб-застосунків, що забезпечують підтримку 3D-графіки у сферах науокової візуалізації, архітектури, дизайну, розваг і т.д.

Глосарій

Серіалізація — процес перетворення даних об’єкта у формат, придатний до зберігання або передачі.

Десеріалізація — процес відновлення даних об’єкту з серіалізованого формату.

3D-графіка — галузь комп’ютерної графіки, що охоплює методи да технології для створення, обробки та візуалізації тривимірних (3D) зображень.

3D-сцена — віртуальне середовище, що містить 3D-об’єкти, їх взаємне розташування, і всі потрібні налаштування необхідні для їх візуалізації.

3D-об’єкт — сукупність даних, що повністю описує об’єкт у 3D середовищі, його меш, матеріали, трансформації, та інші атрибути потрібні для його відтворення.

Геометрична меш сітка (меш) — структурований набір даних що описує поверхню 3D-об’єкту, складається з вершин (точок), ребер (ліній), і граней (плоских багатокутників), що разом формують його топологію.

3D-пакет — комплексне програмне забезпечення, призначене для створення, редагування, анімації, текстурування, освітлення, рендерингу та інших видів маніпуляцій 3D-об’єктів та сцен.

CPU — Central Processing Unit, або центральний процессор.

GPU — Graphical Processing Unit, або графічний процессор.

API — інтерфейс прикладного програмування, що забезпечує взаємодію між різними програмами чи компонентами.

Рендерінг — процес створення двовимірного зображення або анімації з 3D-сцени шляхом обчислення освітлення, текстур і ефектів.

САП — система автоматизованого проектування, програмне забезпечення для створення технічних креслень і 3D-моделей.

Шейдер — програма, що виконується на GPU, і визначає, як поверхня 3D-об’єкта взаємодіє зі світлом, кольором і текстурами під час рендерингу.

Простір кліпу — область 3D-сцени, що визначається після застосування матриці проекції, і в межах якої об'єкти чи їх частини видимі, та будуть відображені на екрані. Все поза цією областю відсікається

Растеризація — етап у рендерингу, під час якого векторні графічні примітиви перетворюються на пікселі на екрані.

Парсинг — процес синтаксичного аналізу вхідних файлів з метою вилучення значущих компонентів для подальшого використання.

Триангуляція — процес розбиття складної геометричної поверхні на набір трикутників для спрощення обробки, рендерингу або аналізу.

1 АНАЛІЗ СУЧАСНИХ 3-D ТЕХНОЛОГІЙ

1.1 Історія та огляд сучасних 3D-технологій

Історія 3D-графіки випереджає винахід комп’ютерів. Зокрема, система декартових координат, широко вживана для опису позиції об’єктів в тривимірному середовищі, була описана Рене Декартом ще у 1600-х а у середині 1800-х років англійський математик Джеймс Джозеф Сильвестр описав матричну математику, яка широко застосовується в сучасній 3D графіці. [1]

Ключовою датою для комп’ютерної графіки стала розробка першого САПР застосунку Іваном Сазерлендом у 1962 році. Його Sketchpad, за допомогою світлового пера, надавав користувачеві можливість малювати лінії прямо на екрані комп’ютера. Як описував сам Сазерленд, інформація про сцену в цьому застосунку зберігалась у N-компонентних об’єктах. Це стандартизовані структури у пам’яті комп’ютера, які описували інформацію, потрібну для відтворення графічної інформації — так об’єкт лінії містить посилання на точки початку і кінця, а об’єкт точки — свої координати [2]. Фактично це можна вважати першим в світі форматом для збереження графічних сцен — але збережені дані не можна було серіалізувати чи легко перенести.

Пізніше у 1964 році, Тімоті Джонсон розробив Sketchpad III. Ця версія застосунку оперувала у тривимірних координатах. Вперше було застосовано однорідні координати (x:y:z:s), що, крім позиції довільної точки, також зберігали “розмірність”, на яку потрібно було поділити інші показники щоб нормувати їх, і привести в систему декартових координат. Крім цього були описані та використані матриці перетворень, такі як матриці повороту, збільшення, перекладу та перспективи. Графічне представлення моделей у вигляді точок і відрізків було названо wireframe, тобто каркас [3]. На той

момент втілений Тімоті Джонсоном функціонал був проривом у 3D-графіці, і більшість розроблених методів використовується і сьогодні.

Першим фільмом, що використовував 3D графіку, став Tron (1982), а першою повнометражною кіно-картиною, що була повністю анімована у 3D, став Toy Story(1995). Перші повноцінні пакети для 3D моделювання, орієнтовані на користувача, з'явились у 1984 році. Це були системи Pro/Engineer (нині Creo Parametrics), що була першою, революційною, системою САП моделювання, і неназваний прототип 3D анімації від Wavefront. Якщо подивитись на сучасні пакети для 3D моделювання, то більшість з них також з’явилась у 1990-х роках.

Завдяки пів-сторіччю розробки, оптимізації, і інтеграції — в сучасному світі комп’ютерна графіка, і 3D-графіка зокрема, використовуються майже в усіх галузях діяльності, які пов’язані з комп’ютерами. 3D-анімація є дешевшою за практичні ефекти чи ручне малювання, кіностудії і розробники ігор віддають перевагу саме їй. Віртуальні 3D-моделі допомагають дизайнерам, інженерам і архітекторам швидко створювати прототипи, а дослідники і аналітики використовують 3D середовище для візуалізації даних. Але це все було б неможливо, якщо б користувацьке обладнання не було здатним до обробки 3D-графікі.

На відміну від задачі обробки тексту чи аудіо файлів, де система працює з одним безперервним потоком даних, обробка 3D-інформації є ресурсоємною задачею. Рендерінг — це процес перетворення 3D-моделей, що визначаються вершинами і полігонами, розташованими у тривимірному просторі сцени, на 2D зображення, враховуючи властивості матеріалів, налаштування середовища, присутні джерела світла, тощо. В сучасному світі існують вимоги до пришвидшення цього довгого процесу. Зокрема, для комфортного користувацького досвіду, відеоігри мають показувати нову картинку від 24 раз на секунду. Оптимальний ігровий процес потребує 60, і навіть 140 кадрів на секунду. При популярній роздільній здатності екрану 1920 на 1080 пікселів — це обробка від 5, до 300 мільйонів пікселів в секунду. Звісно це досягається

значною мірою завдяки програмній оптимізації і мінімізації 3D-об’єктів — наприклад обробка заміна функцій трасування променів апроксиматорами, чи чи видалення надмірної деталізації 3D-моделі. Втім, найбільш ефективною оптимізацією є апаратна заміна центрального процесора на графічний.

Центральний процесор (далі в тексті CPU) не здатен достатньо швидко обробляти задачі лінійної алгебри, щоб ефективно виконувати задачу рендерингу в реальному часі. Обчислення 3d графіки має специфічні вимоги: великий кеш оперативної пам'яті для текстур, сіток, шейдерів і т. д., можливість виконання великої кількості паралельних обчислень на задачах лінійної алгебри. Графічні процесори (далі в тексті GPU) вирішують ці проблеми. Їх архітектура обмінює швидкість обчислень на паралелізм і реалізує відео пам’ять з довільним доступом (VRAM), яка повільніша за кеш процесора, але достатньо швидка для виконання свого призначення [4].

Історія GPU почалася з появою 3D-прискорювачів, таких як Voodoo Graphics від 3dfx, випущених у 1996 році для обробки 3D-графіки. У 1999 році Nvidia представила термін GPU з випуском GeForce 256 — одно кристального процесора, здатного обробляти трансформації, освітлення та рендеринг із продуктивністю щонайменше 10 мільйонів полігонів на секунду.

Новітня архітектура GPU також потребувала нових API для взаємодії існуючих систем з новим апаратним забезпеченням. Перша версія OpenGL (Open graphics library) вийшла у 1992 році як альтернатива закритому Iris GL. Вона описувала себе як “набір команд, що дозволяють специфікувати геометричні об'єкти у двох або трьох вимірах разом з командами, що керують їхнім відображенням”, або “машину станів, яка керує набором операцій малювання”[5]. В 1996 році Microsoft випустила свою альтернативу — Direct3D, що увійшов до їх колекції API DirectX.

На поточний момент найпопулярнішими графічними API є Direct3D 12, (2020). Vulkan 1.3 2022 року — кросплатформенний наступник OpenGL. Metal — для екосистеми Apple. І OpenGL 4.6 (2017), що є застарілою в порівнянні з іншими API, але через велику кількість документації, написаних на ньому

бібліотек, і загальної популярності — вона все ще підтримується всіма постачальниками відеокарт.

1.2 Структура 3D-об'єктів

Під терміном “структура 3D об’єкта” мається на увазі інформація, потрібна для відтворення його поверхні у тривимірному середовищі. Умовно, 3D-графіку можна поділити на чотири основні типи: аналітична, фрактальна, полігональна і сплайнова [6]. В кожній з цих моделей структура 3D-об’єкту зберігається специфічно.

В аналітичній і фрактальній графіці об’єкти задаються формулами, які застосунки інтерпретують, генеруючи відповідні 3D-сціни. Сплайнова графіка описує контури 3D-об’єктів кривими. Це корисно для опису обтічних поверхонь в САП застосунках, але перед практичним застосування моделі, зазвичай, конвертуються в полігоналний вигляд.

Полігональна графіка описує гометричну сітку поверхні об’єкту, і реалізується через три основні підходи подання: просте, граничне та об’ємне [6]. У прості моделі входять точкова та каркасна. У першій об’єкт задається множиною точок, що належать його поверхні. Кожен об’єкт описується множиною $V = \{V_1, V_2, V_3 \ldots, V_n\}$. Каркасна модель розширює це представлення множиною ребер $E = \{v_i, v_j, v_k, \ldots v_n\}$, де кожне ребро є прямою, проведеною через 2 точки.

Об’ємна модель представляє об’єкти як сукупність примітивів. Вокселі є 3D аналогом пікселю, і описуються як кубічні елементи. Альтернативою вокселів є метасфери, що є сферичними тілами, здатними визначеним чином впливати одне на одного.

Гранична модель передбачає розширення простої каркасної моделі гранями. Грані є сукупністю точок чи ребер, що описують безперервну, плоску поверхонь. Незважаючи на те, що для опису будь-якої площини достатньо трьох точок — полігони можуть складатись з довільної кількості вершин, що

належать одній площині. Кожен полігон можна описати як: $P = \{V_1,V_2,V_3 \ldots, V_n\}$. Тобто як множину точок, але, на відміну від представлення простої точкової моделі — ці вершини повинні належати одній площині p, $\{V_1,V_2,V_3 \ldots, V_n\} \in p$. Ця модель також має декілька модифікацій. До неї можна додати ребра — відрізки між двома вершинами, вигляду $E = \{V_a,V_b\}$. Тоді полігон (грань) буде виглядати як: $P\{E_1,E_2,E_n \ldots ,E_n\}$.

Крім стандартизованої геометричної структури, 3D-об’єкти часто містять додаткові атрибути, що впливають на поведінку чи зовнішній вигляд об’єкту під час рендерингу. Одним з таких важливих атрибутів є нормалі — вектори, що вказують напрям, перпендикулярний до поверхні об’єкта в довільній точці. У 3D-графіці, нормалі відіграють ключову роль в обчислені освітлення і реалізації ефектів згладжування, також вони використовуються в процесі оптимізації. Поширеним підходом до зберігання нормалей є представлення їх у вигляді масиву векторів, де кожній вершині відповідає нормаль: $N = \{N_1,N_2,N_3 \ldots, N_n\}$

Також зберігаються дані про UV-розгортання. Це підхід до текстурування, що передбачає створення двовимірної мапи, і розташування на ній кожної вершини мешу. Таким чином текстура накладається на поверхню 3D-об’єкту. Для кожної точки потрібно зберігати координати в осях U та V, що є аналогами простої 2D декартової системи, $V_n = \{u,v\}$. [7]

1.3 Матеріали та текстури у 3D-графіці реального часу

Невід'ємною частиною сучасної 3D-графіки є матеріали та текстури. Вони відповідають за візуальну якість та реалістичність об’єктів у 3D-середовищі, додаючи деталізації до порожньої поверхні без значних затрат ресурсів. Їх використовують для імітації чи апроксимації фізичних властивостей поверхонь: кольору, блиску, шорсткості, прозорості, дисперсії світла та інших властивостей — це допомагає користувачам набагато легше

розрізняти, розбивати на складові та загалом ідентифікувати 3D-об'єкти, використовуючи візуальний досвід з реального світу.

Але матеріали — лише набори даних, що повинні оброблятися певними функціями. Шейдер (shader, від англ. shading, тобто затінення) — це програма, написана на спеціалізованих мовах графічного програмування, що виконується на графічному процесорі, і описує, як повинна виглядати 3D-сцена в кожний момент часу. Шейдери, що використовуються при рендерингу в реальному часі, діляться на кілька підвидів.

Вершинні шейдери відповідають за перетворення координатів вершин. Вони, за допомогою матриць перетворень, трансформують вершини з простіру об’єкта, до простіру кліпу, відсікаючи геометрію, яка не буде обраховуватись. Це вимагає використання однорідних координат.

Фрагментні (чи піксельні) шейдери використовуються після вершинних, і проводять растеризацію сцени. Для кожного пікселя (фрагмента) результуючого зображення, вони беруть перелік інформації: значення координат, нормалей, і UV; дані про матеріал поверхні; перелік і властивості джерел світла; текстурну інформацію — і, відповідно до моделі освітлення, повертають фінальний колір, яким піксель (фрагмент) має бути зафарбовано. Потрібно зазначити, що ці шейдери обробляються всі пікселі об’єктів, до яких застосовуються, в паралелі. [8]

Є велика кількість різних шейдерів, що використовуються для рендерингу в реальному часі. Всі вони використовують різні набори даних, і використовуються для різних задач. Модель Ламберта (також відома як дифузійний шейдер) використовується для матових поверхонь, і описує дифузійне розсіювання світла. Матеріал, що описує цю модель, буде мати лише одну змінну — кольору поверхні у форматі RGB, albedo. Модель Фонга розширює це, і використовується для простих блискучих поверхонь. Вона потребує додачі двох нових змінних: ступені блиску shininess, і коефіцієнту блиску specular. Але в реальній середі майже завжди використовується модель PBR (Physically Based Rendering, фізично обґрунтований рендеринг). Вона

намагається апроксимувати реальну поверхонь, використовуючи для обчислення її фізичні властивості. Матеріал, який її описує, має містити: albedo — колір поверхні; metallic — коефіцієнт металевості; roughness — шорсткість поверхні; AO (ambient occlusion) — попередньо розрахована мапа затінення.

Текстури використовуються, коли до поверхні потрібно надати деталізацію, без ускладнення мешу. Але крім текстур зображень, в графіці використовуються текстурні мапи — спеціальні текстури, що призначені для управління конкретного параметру матеріалу. Цими мапами можна замінити всі фізичні параметри, наявні в описаних моделях освітлення. Мапа кольору (diffusion map, albedo map, map) передає колір у форматі rbg на кожній точці поверхні, це звичайна кольорова картинка. Мапи фізичних властивостей (roughness map, metalic map, sheen map і т.д.) передають значення відповідних властивостей у форматі значень від 0 до 1, це монохромне зображення. Також існує спеціальна мапа нормалей (normal map), допомагає уточнювати напрям нормалей на плоскій поверхні. Ця мапа, хоча і представлена кольоровим зображенням, сприймає кольори в кожній точці не як rbg, а як вектори xyz, що описують напрям нормалі. Для коректного використання текстур об’єкт має мати функціональну UV розгортку. Вона трансформується шейдером вершин до простору кліпу, і шейдер пікселів у кожному пікселі має доступ до відповідного значення кожної зазначеної мапи.

1.4 Структура та особливості форматів 3D-файлів

Файли 3D-об’єктів можна умовно поділити за призначеннями, вмістом, сферою застосувань, та форматом збереження На практиці, розглядаються формати авторського середовища і формати посередники.

Формати авторського середовища (.blend, .max, .mb, .c4d) зберігають унікальні дані, такі як історія редагування чи модифікатори, але через закритість і складність парсингу вони обробляються лише відповідними 3D-

пакетами. Вони зручні для редагування 3D-сцін, але, як правило, неоптимізовані для передачі.

Формати посередники призначені для сумісності між застосунками. Вони розроблені таким чином, щоб бути універсальними, відкритими — щоб розробники могли реалізовувати власні парсери — і простими для розуміння. Підходи до їх розробки є різними: деякі формати посередники зберігають тільки основну, специфічну інформацію (obj та ply), деякі призначені для передачі сцен між 3D-пакетами (collada та fbx), деякі працюються лише у своїй сфері 3D (stl у сфері 3D-принтерів), а деякі оптимізовано для скорості читання і обробки (gltf).

1.4.1 Формат OBJ

Формат .obj був розроблений Wavefront Technologies наприкінці 1980-х. Це був один із перших популярних форматів збереження 3D-файлів. Це простий у читанні текстовий формат для опису мешу 3D-моделі, який легко імпортувати на експортувати між різними застосунками. Формат є часто вживаним, підтримується майже усіма застосунками, і фактично є найпростішим у реалізації стандартом обміну інформації про 3D-об’єкти.

Важливим є факт, що .obj — текстовий. Тобто для роботи з ним достатньо відкрити текстовий потік і зчитувати рядками. Інформацію подано у вигляді інструкцій, за якими слідують значення. Макет найпростішого варіанту формату .obj надано у лістингу 1.

Лістинг 1— Схема формату OBJ

```
mtlib x.mtl
o object
g meshgroup
usemtl material
v x y z
vt u v
vn x y z
f v1/vt1/vn1 v2/vt2/vn2 vn/vtn/vnn
```

Розбір структури формату подано в Табл. 1 [9].

Таблиця 1 – Поля формату OBJ

Інструкція	**Опис**
`mtlib [.mtl]`	Файл бібліотеки матеріалів (.mtl)
`o [object name]`	Назва поточного об’єкту
`g [group name]`	Назва групи геометрії.
`usemtl [mat name]`	Матеріал поточної геометричної групи.
`s [on/off]`	Наявність згладжування.
`v x y z (w)`	Координати вершини у декартовому просторі.
`vt u v (w)`	Текстурні UV координати для вершини.
`vn x y z (w)`	Вектор нормалі.
`f v1/vt1/vn1…`	Грань, індекси вершин, текстурних координат, і нормалей.
`l v1/vt1…`	Лінія, індекси вершин і текстурних координат.

Формат .mtl (Material Template Library) був розроблений для використання разом із .obj. Це текстовий формат, що описує бібліотеку матеріалів, кожен з яких може використовувати .obj файл. Цей формат не має стандартизації так само, як її не має і .obj. Подивимось на макет матеріалу, який може описувати .mtl файл, його надано у лістингу 2.

Лістинг 2 — Схема формату MTL

```
newmtl material
Ka r g b
Kd r g b
Ks r g b
Ns value
Ni value
d value
illum model
```

```
map_Ka ambient.jpg
```

Розбір структури формату подано в Табл. 2 [10].

Таблиця 2 – Поля формату MTL

Інструкція	**Опис**
`newmtl material`	Назва поточного матеріалу матеріалу.
`Ka r g b`	Колір навколишнього світла.
`Kd r g b`	Основний колір матеріалу.
`Ks r g b`	Колір блиску.
`Ke r g b`	Колір випромінювання.
`Ns value`	Жорсткість відбиття
`Ni value`	Індекс заломлення.
`d value`	Значення прозорості.
`illum model`	Поточна модель ілюмінації, приймає фіксовані значення. 0: статичний колір. 1: модель Ламберта (прості тінні). 2: модель Блінна-Фонга(просте відбиття). 3+: будь які ускладнені моделі освітлення (PBR, extended BRDF, etc)
`map_[name] [image filename]`	Будь яке з полів, що використовують формат даних r g b, можна зазначити текстурою.

Це тільки основні, більш підтримані інструкції, які можна знайти в форматах .obj та .mtl.

1.4.2 Формат STL

STL (STereo Lithography) формат було розроблено у 1987 році на замовлення компанії 3D Systems. Цей формат використовується для зберігання 3D-об’єктів у сфері 3D-друку. Формат має текстову (ASCII) версію, і бінарну. Незалежно від версії, STL формат може зберігати тільки дані про меш об’єкту. Приклад текстової версії формату, наведено в Лістингу 3.

Лістинг 3 — Схема формату .STL

```
solid
     facet normal ni nj nk
     outer loop
          vertex v1x v1y v1z
          vertex v2x v2y v2z
          vertex v3x v3y v3z
     endloop
     endfacet
endsolid
```

Розбір структури формату подано в Табл. 3 [11].

Таблиця 3 – Поля формату STL

Структура	**Опис**

`solid`
…
`endsolid`
	Структура, що містить фігуру

`facet normal ni nj nk`	Нормаль полігону.

`outer loop`
` vertex v1x v1y v1z`
` vertex v2x v2y v2z`
` vertex v3x v3y v3z`
`endloop`
	Полігон завжди є трикутником. Порядок вершин має відповідати обходу за годинниковою стрілкою.

Важливо зазначити, що всі числові значення представлені у форматі з плаваючою комою. Подібну структуру має бінарний STL. Його розбір надано в Табл. 4.

Таблиця 4 – Бінарний потік даних STL

Частина	**Величина**	**Опис**
1.	80 байт	Перші 80 байт займає метадані, часто представлена у форматі строки тексту
2.	4 байт	Наступні 4 байт це число 32-бітне ціле число, що вказує кількість полігонів.
3…n	50 байт	Кожен полігон займає 50 байт. Кожна координата закодована в 4 байтах як число з плаваючою комою. Першими в потоці йдуть 12 байт нормалі, потім 36 байт для вершин, і 2 порожні щоб відокремити полігони один від одного

1.5.3 Формат PLY

PLY (Polygon File Format) чи “Стенфордський формат трикутників”, було оприлюднено у 1994 році, розробником Грегом Тюрк із Стенфордського університета. PLY було розроблено як основний формат файлу для використання в новітніх системах 3D-сканування.

PLY — це текстовий (ASCII) формат який структурою нагадує OBJ, але з низкою поліпшень. По-перше, числова інформація, доступна до бінарної серіалізації, була відокремлена від текстової інформації. PLY файл можна поділити на дві умовні частини: заголовок, що описує кількість і типи даних, та тіло, яке містить перелік числових значень. Також було вирішено проблему парсингу, використовуючи інструкцій element та property. Така структура дозволяє зберігати будь-які дані, в будь-якому форматі, без жодних обмежень. Стандартизація дозволяє парсерам ефективно обробляти поля, які ними не підтримуються. Структуру заголовку формату розібрано в Табл. 5.

Таблиця 5 – Поля формату PLY

Інструкція	**Опис**
`element <name> <count>`	описує новий тип елементу назвою name, що зустрічається count разів.
`property <type> <name>`	описує скалярний атрибут поточного елемента за назвою name, і типом даних type.
`property list <count_type> <type> <name>`	описує рядок параметрів змінної довжини.
`ply`	Інструкція, з якої починається кожен PLY файл.
`format <type> <version>`	Формат кодування файлу (ASCII або бінарний)
`comment <text>`	Коментар до файлу.

Тіло файлу PLY — це перелік значень, зазначених у заголовку. Спосіб представлення значень залежить від обраного типу. В ASCII, їх представлено текстовими символами через пробіли. В бінарному представленні — потоками даних, де кожний тип має свою довжину. Схему стандартного заголовку PLY, що використовується для поширення геометрії у стандартному представлені, подано у Лістингу 4.

Лістинг 4 —Схема стандартного заголовку PLY

```
ply
format ascii 1.0
element vertex 3
property float x
property float y
property float z
element face 1
property list uchar int vertex_indices
end_header
```

1.4.4 Формат GLTF / GLB

GLTF/GLB це формат для збереження 3D-сцен, розроблений Khronos Group і оприлюднений у 2015 року. GLTF задумувався як універсальний формат-посередник, але його також спеціально розробляли для компактності, і оптимізованості при роботі з рендерингом реального часу. Формат доступний у текстовий (json) формі glTF (Graphics Library Transmission Format), і у ще більш оптимізованій бінарній формі GLB (GL Transmission Format Binary File). Слід зазначити, що “текстовий” glTF посилається на бінарний файл bin, де міститься інформація про вершини, полігони, геометричні групи, та інш. Також в сучасних варіантах, файл glTF може вбудовувати в себе двійковий потік.

Для розуміння структури формату glTF потрібно розуміти поняття “нод”. В контексті gltf, нода — це точка входу для опису кожного об’єкта.. Крім стандартних представлень 3D даних, кожен файл цього формату містить об’єкти Buffer, BufferView і Acessor.

Buffer — це буфер бінарної інформації. Він може посилатись на зовнішній .bin файл, чи може сам містити бінарні дані. Шлях до файлів визначено полем “uri”, розмір потоку зазначено у полі “byteLength”. Слід зазначити, що glTF не має обмежень на кількість потоків. Приклад Buffer надано у лістингу 5.

Лістинг 5 — Об’єкт buffers формату GLTF

```
"buffers":[
     {
          "byteLength":36,
          "uri":"binary.bin"
     }
]
```

Об’єкти типу BufferView описують вміст буферу. Кожен містить посилання на буфер у полі “buffer”, довжину фрагменту “byteLength”, зсув

фрагменту з початку потоку “byteOffset”, і тип даних “target” що описує тип даних, які знаходяться у фрагменті. Приклад надано у Лістингу 6.

Лістинг 6 — Об’єкт bufferViews формату GLTF

```
"bufferViews":[
     {
          "buffer":0,
          "byteLength":36,
          "byteOffset":0,
          "target":34962
     },
]
```

Об’єкти типу Accessor займаються інтерпретацією BufferView. Вони описують ті дані, які закодовано в виділенні у bufferView фрагменти, і містять атрибуті, потрібні для їх відтворення і контексттуалізації. Поле “bufferView” посилається на той фрагмент, який описує Accessor. “ComponentType” визначає тип описуваного компонента. “count” містить кількість елементів у фрагменті. “max” і “min” містить максимальні і мінімальні значення розташування об’єктів, а “type” визначає тип даних: VECN — вектори довжини N, MATN — матриці розмірності N, SCALAR — скалярні величини, string — символьні потоки. Приклад надано у лістингу 7

Лістинг 7 — Об’єкт accessors формату GLTF

```
"accessors": [
  {
    "bufferView": 0,
    "componentType": 5126,
    "count": 3,
    "max": [1, 0, 1],
    "min": [-1, 0, -1],
    "type": "VEC3"
  }
]
```

Тепер повернемось до традиційних даних, що описують 3D-об’єкт. Розберемо конструкцію meshes, що містить всі геометричні дані файлу. Всі

числові дані, наприклад координат, векторів нормалей і так далі, передаються посиланням на послідовний номер об’єкту типу Accessor, який їх описує. Приклад надано у Лістингу 8

Лістинг 8 — Об’єкт meshes формату GLTF

```
"meshes":[
     {
          "name":"Plane",
          "primitives":[
               {
                    "attributes":{
                         "POSITION":0,
                         "NORMAL":1,
                         "TEXCOORD_0":2
                    },
                    "indices":3
               }
          ]
     }
],
```

Аналіз можливих полів конструкції meshes було проведено в Табл. 6.

Таблиця 6 – Поля meshes формату GLTF

Поле	**Опис**
`name`	Назва 3D-об’єкту
`primitives`	Масив геометричних одиниць. (полігонів, відрізків)
`primitives .attributes`	Словник, що містить дані про геометричні одиниці.
`attributes .POSITION`	Посилання на координати кожної вершини у декартовій системі.
`attributes .NORMAL`	Посилання на вектори нормалі кожної вершини.
`attributes .TEXCOORD_n`	Посилання на текстурні координати кожної вершини.
`attributes .COLOR_n`	Посилання на кольори кожної вершини.
`primitives .indices`	Посилання на послідовність вершин.
`primitives .material`	Посилання на матеріал, застосований до об’єкту.
`primitives .mode`	Метод рендерингу поточного примітиву. 0 — точки, 1 — лінії, 4 — трикутники (стандарт за замовчуванням)

Матеріали у glTF, за замовчуванням, представлені у варіації моделі PBR, що містить значення металевості та шорсткісті. Слід додати, що кожна текстура визначається полями index, що визначає порядковий номер текстури в переліку текстур, та texCoord, що визначає порядковий номер текстурних координат. Текстурні координати визначають перетворення картинки текстури у UV просторі, які проходять перед накладанням її на 3D-об’єкт. Приклад описано у Лістингу 9

Лістинг 9 — Об’єкт materials формату GLTF

```
"materials":[
          {
          "doubleSided":true,
          "name":"Material.001",
          "pbrMetallicRoughness":{
               "baseColorTexture":{
                    "index":0
               },
               "metallicFactor":0.5,
               "roughnessFactor":0.5
          }
     }
],
```

Аналіз можливих полів конструкції materials було проведено в табл. 7.

Таблиця 7 – Поля materials формату GLTF

Поле	**Опис**
`name`	Назва матеріалу.
`doublesided`	Чи відбувається видалення повернутих задом полігонів.
`pbrMetallicRoughness`	Контейнер для фізичних властивостей, потрібних PBR
`baseColorFactor`	Базовий колір матеріалу у форматі rbg alpha.
`baseColorTexture`	Посилання на текстуру, яка містить колір матеріалу.
`metallicFactor`	Значення “металевості” матеріалу.
`metallicFactorTexture`	Посилання на текстуру яка визначає металевість матеріалу.
`roughnessFactor`	Шорсткість матеріалу.
`roughnessFactorTexture`	Посилання на текстуру яка визначає шорсткість матеріалу.
`normalTexture`	Посилання на мапу нормалей.
`normalScale`	Множник інтенсивності мапи нормалей.
`occlusionTexture`	Посилання на текстуру, що визначає затінення близькості.
`occlusionStrength`	Множник інтенсивності мапи затінення близькості.
`emissiveFactor`	Колір випромінювання.
`emissiveTexture`	Посилання на текстуру, що визначає колір випромінювання.
`emissiveStrength`	Множник інтенсивності випромінювання.
`alphaMode`	Визначає метод обрахунку прозорості. “OPAQUE” — непрозорий матеріал, “MASK” — маска, “BLEND” — прозорість Alpha Blend.

В результаті glTF є зручним і добре документованим форматом посередником, а також форматом, який найбільше використовують саме для браузерної 3D-графіки. Найбільшими проблемою є факт збереження геометрії у трикутниках, а також відсутність підтримки формату старими системами.

1.4.5 Формат COLLADA

COLLADA (Collaborative Design Activity) є відкритим форматом для збереження 3D-сцен, розробленим Sony Computer Entertainment у 2004 році. З самого початку COLLADA був відкритим форматом, спробою зробити універсальний формат-посередник між популярними застосунками того часу. У в 2006, Sony передала права на підтримку COLLADA Khronos Group. Остання, на момент написання цієї роботи, ревізія COLLADA 1.5 була оприлюднена у 2008 році. Формат базується на XML.
Кореневі структури формату наведено у табл. 8.

Таблиця 8 – Деякі кореневі структури формату COLLADA

Елемент	**Пояснення**
<COLLADA>	Інформація про версію формата COLLADA.
<asset>	Метаданих про застосунки-автори, дату створення і останньої зміни, одиниці виміру, і систему координат.
<library_effects>	Інформація про шейдери.
<library_images>	Інформація про картинки та текстури, посилання до них.
<library_materials>	Інформація про матеріали.
<library_geometries>	Інформація про меш об’єктів.
<library_visual_scenes>	Інформація про вміст сцен, і ієрархію об’єктів.
<scene>	Інформація про основну сцену для візуалізації.

Детальніше розберемо структуру кожного сегмента, на прикладі <library_geometries> містить всю інформацію про меш об’єктів, які є в збереженій сцені. Він зберігає кожний у окремому елементі <geometry>, що містить параметри id з унікальним символьно-числовим ідентифікатором, і name з неунікальною назвою предмету. Всередині елементу geometry може знаходитись як <mesh>, тобто меш, так і <spline>, тобто сплайн-крива задана вершинами. Приклад наведено у Лістингу 10.

Лістинг 10 — Приклад збереження геометричних даних у COLLADA

```
<source id="C-mesh">
     <float_array id="C-mesh-array" count="192">0 1 -1
…</float_array>
     <technique_common>
     <accessor source="#C-mesh-array" count="64" stride="3">
          <param name="X" type="float"/>
          <param name="Y" type="float"/>
          <param name="Z" type="float"/>
     </accessor>
     </technique_common>
</source>
```

Сама інформація про геометричний об’єкт міститься у тегах <source>, що описує джерела. На першому місці завжди йдуть самі дані, зазвичай це масив значень в параметрах якого вказана кількість. В цьому випадку це <float_array>, тобто масив числових значень типу float. Після цього зазначається об’єкт <accessor>, що описує структуру збережених даних і порядок отримання доступу до них. Назва і формат полів зазначається у тегах <param/>, і кількість структур у параметрі count. Після зазначення всіх джерел, в файлі визначаються ті примітиви, що їх використовують, і в якому вигляді. Приклад у Лістингу 11.

Лістинг 11 — Приклад збереження даних про полігони у COLLADA

```
<triangles count="124">
     <input semantic="VERTEX" source="#C-vertices"
offset="0"/>
```

```
     <input semantic="NORMAL" source="#C-normals"
offset="1"/>
     <input semantic="TEXCOORD" source="#C-map-0" offset="2"
set="0"/>
     <p>1 0 0 2 …</p>
</triangles>
```

В цьому прикладі визначають об’єкти полігональний трикутників <triangles>, але можуть описуватись інші види інформації: <polygons> — полігони довільної довжини, <vertices> — вершини, <lines> — лінії, і так далі. Кожен примітив має певні дані, зазначені тегами <input>. В параметрі semantic міститься тип даних (вершина, нормаль, текстурна UV координата), в параметрі source — асоційоване джерело, а в offset — зсув даних у масиві індексів. Сам масив індексів міститься у тегу <p>, і вказує порядкові номери семантичних даних вказаних у тегах input, циклічно, для кожної вершини кожного трикутника.

На відміну від геометричних даних, дані про матеріали об’єктів представлені трьома групами, <library_effects>, <library_materials>, <library_effects>. Сегмент <library_materials> визначає існування матеріалів, їх назву, і посилається на об’єкт типу “effect”. Приклад подану у лістингу 12.

Лістинг 12 Приклад збереження даних про матеріали у COLLADA

```
<material id="Material_004-material" name="Material.004">
      <instance_effect url="#Material_004-effect"/>
</material>
```

Опис фізичних властивостей матеріалу зберігається у <library_effects>, в об’єктах ефектів. Якщо матеріал використовує текстури — створюється тег типу <sampler>. Далі, в technique, зазначається використаний шейдер, COLLADA підтримує 4 різні шейдери: <constant> — що описує поверхню статичного кольору незалежну від джерел світла; <lambert> — що описує матову поверхню без відблиску; <phong> — описує поверхню з дзеркальними відблисками, відповідно до моделі Фонга; <blinn> — описує поверхню з

дзеркальними відблисками, за моделлю Блінна-Торренса-Спероу. Кожна модель має перелік (не обов’язкових) полів, які можливо задавати числовими значеннями, чи змінними текстур. Загальний приклад наведено у Лістингу 13.

Лістинг 13 — Приклад збереження даних про шейдери у COLLADA

```
<effect>
<profile_COMMON>
<newparam sid="_texture_jpg-surface">
     <surface type="2D">
          <init_from>_texture_jpg</init_from>
     </surface>
</newparam>
<newparam sid="_texture_jpg-sampler">
     <sampler2D>
          <source>_texture_jpg-surface</source>
     </sampler2D>
</newparam>
<technique sid="common">
<lambert>
     <diffuse>
     <texture texture="_texture_jpg-sampler"
              texcoord="Cube_002-mesh-map-0"/>
     </diffuse>
</lambert>
</technique>
</profile_COMMON>
</effect>
```

Змінні картинок створюються в сегменті <library_images>, і посилаються безпосередньо на файл картинки. Приклад у лістингу 14.

Лістинг 14 — Приклад збереження даних про текстури у COLLADA

```
<image id="_texture_jpg" name="_texture_jpg">
      <init_from>texture.jpg</init_from>
</image>
```

Всі ці компоненти поєднуються разом у компоненті <library_visual_scenes>, що поєднує перелік сцен. Кожен компонент сцени зберігається у так званій “ноді”, що містить матрицю трансформацій, посилання на геометрію поточного об’єкту, посилання на використані

матеріали. Також в сцена реалізована ієрархія — якщо будь-який об’єкт є дочірнім об’єктом іншого, то його нода буде безпосередньо знаходиться в ноді батьківського об’єкта. Крім геометрії, нода може містити джерело світла, камеру, скелет контролер, тощо. Нода також може бути порожньою, і використатись як порожній організаційний елемент. Приклад у Лістингу 15.

Лістинг 15 — Приклад збереження даних про сцену у COLLADA

```
<visual_scene id="Scene" name="Scene">
<node id="Cube" name="Cube1" type="NODE">
     <matrix sid="transform">1 0 0 …</matrix>
     <instance_geometry url="#Cube_1-mesh" name="Cube1">
     <bind_material>
     <technique_common>
          <instance_material symbol="Material_1-material"
          target="#Material_1-material">
          <bind_vertex_input semantic="UVMap"
          input_semantic="TEXCOORD" input_set="0"/>
     </instance_material>
     </technique_common>
     </bind_material>
     </instance_geometry>
     <node id="Cube_001" name="Cube.001" type="NODE">
          <matrix sid="transform">0 0 0 …</matrix>
          <instance_geometry url="#Cube_2-mesh"
          name="Cube2"/>
     </node>
</node>
</visual_scene>
```

В результаті формат є універсальним, екстенсивним у кількості збережених даних, добре документованим, і підтримуваним в багатьох сучасних застосунках.

2 АНАЛІЗ ТЕХНОЛОГІЙ ТА ІНСТРУМЕНТІВ

2.1 Огляд існуючих рішень

Оскільки на ринку програмного забезпечення існує багато спеціалізованих рішень для загальнох 3D-маніпуляції — проблема конвертації між форматами залишається актуальною і популярною. В ході роботи було розібрано декілька програм-аналогів.

Blender — повноцінний пакет для 3D-моделювання з відкритим кодом. Це автономний кросплатформений застосунок, що працює на всіх популярних операційних системах: Linux, Windows, macOS. (див. рис.1)

Переваги:

- Повна підтримка всіх аспектів 3D-моделювання.
- Доступність — безкоштовний застосунок, за відкритим кодом, оптимізований під старі системи, доступний на всіх популярних платформах.
- Підтримує майже всі формати. Завдяки системі користувацьких плагінів, можна знайти майже будь-яке розширення файлів.

Недоліки:

- Перевантаження зайвими (у контексті конвертації форматів) інструментами, інтерфейс із високою кривою навчання.
- Проблеми з оптимізацією при імпорті великих сцен чи моделей.
- Проблеми з деякими модулями імпорту та експорту (нестабільність FBX, порушення ієрархії у COLLADA)

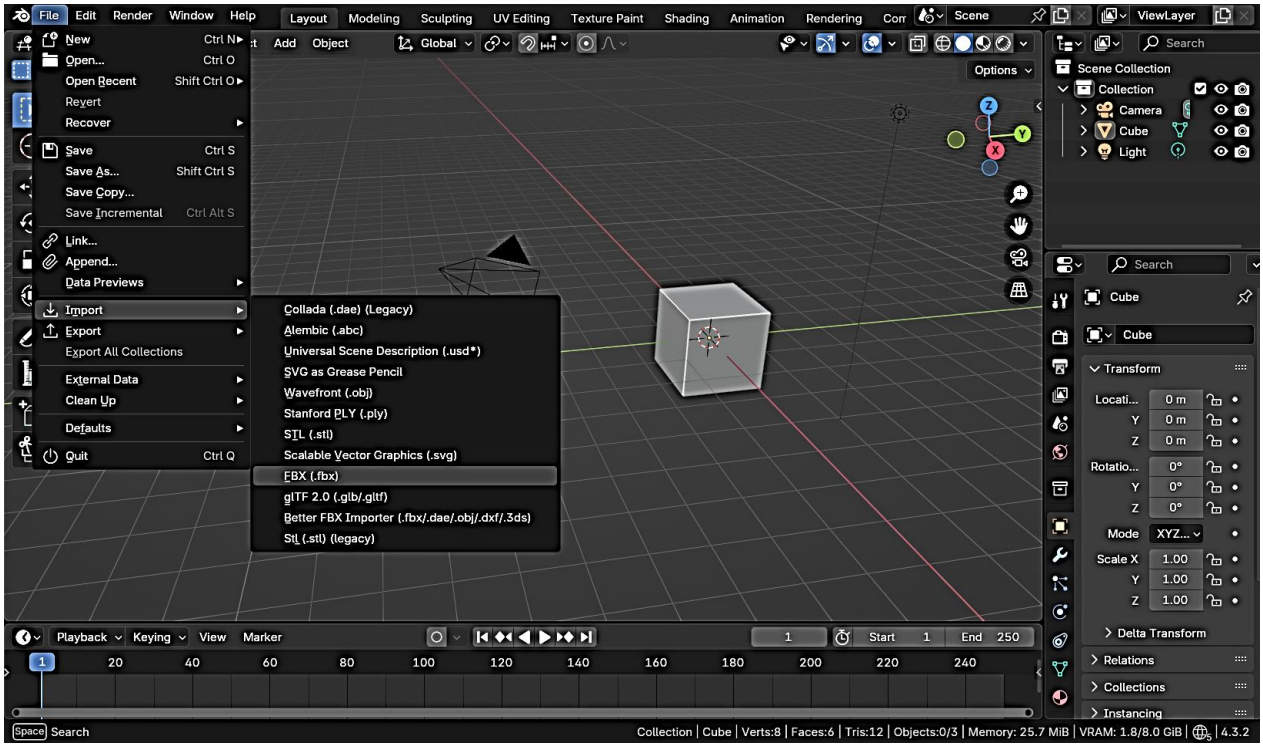


Рисунок 1 — Інтерфейс Blender з демонстрацією меню експорту

Online 3D Viewer — онлайн застосунок/переглядач 3D-файлів з відкритим кодом. Він працює безпосередньо у браузері, тому не вимагає встановлення додаткового програмного забезпечення. Основна функція застосунку — візуалізація 3D-моделей, завантажених користувачем, і огляд її метаданих. Крім індивідуальних об’єктів, Online 3D Viewer підтримує сцени з ієрархією. (див. рис.2)

Переваги:

- Застосунок є кросплатформним — він працює на будь-якій системі, що підтримує OpenGl.
- Користувацькі моделі обробляються локально, у системі. Дані не завантажуються на сервер, що допомагає із швидкістю сервісу і приватністю.
- Підтримує багато форматів імпорту.

Недоліки:

- Хоча застосунок підтримує імпорт багатьох форматів, список опцій для експорту є малим.
- Застосунок використовує внутрішнє представлення сцени three.js, через це деякі дані про 3D об’єкти видаляються.
- Швидкість роботи застосунку залежить від потужності системи користувача, що може призводити до технічних проблем.

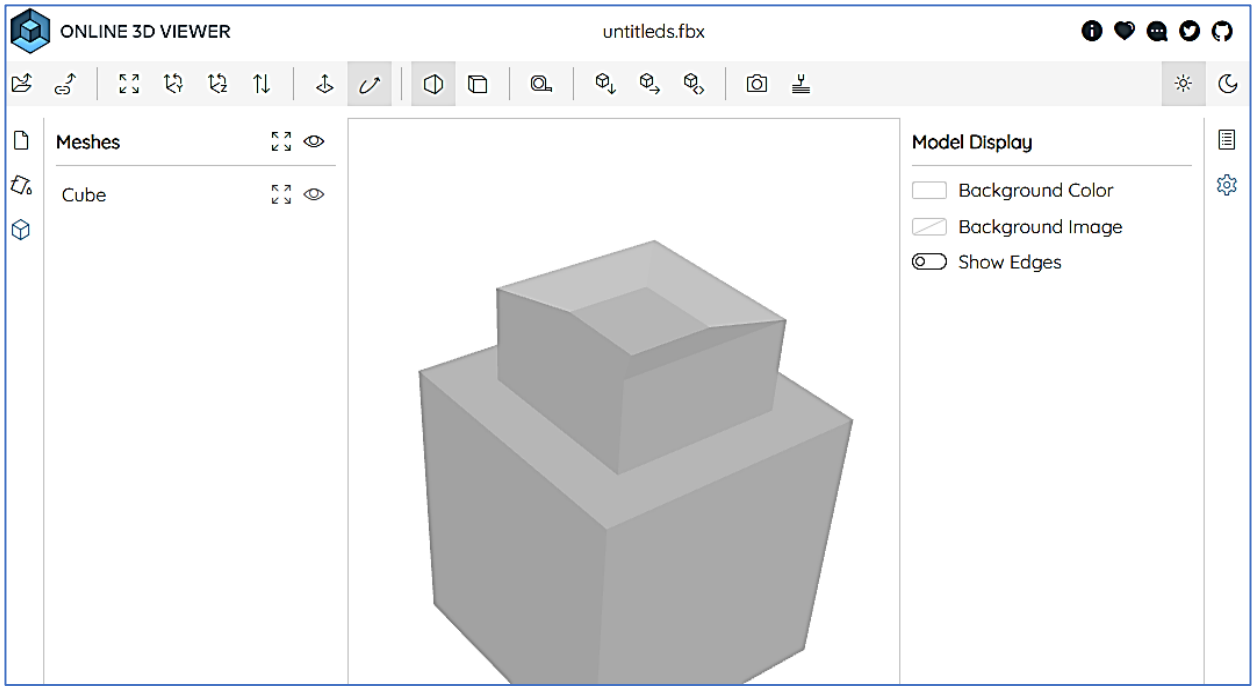


Рисунок 2 — Інтерфейс Online 3D Viewer з тестовим 3D-об’єктом

3D-CONVERT.com — безкоштовний онлайн застосунок для конвертації 3D-моделей різних форматів. Підтримує багато форматів як для імпорту, так і для експорту, але вимагає завантаження моделей на сервер. (див Рис. 3)

Переваги:

- Підтримка великої кількості форматів 3D-файлів.

- Серверна архітектура надає можливість обробки пропрієтарних форматів.
- Швидкість сервісу не залежить від комп'ютеру користувача.

Недоліки:

- Всі моделі завантажуються на сервер застосунку.

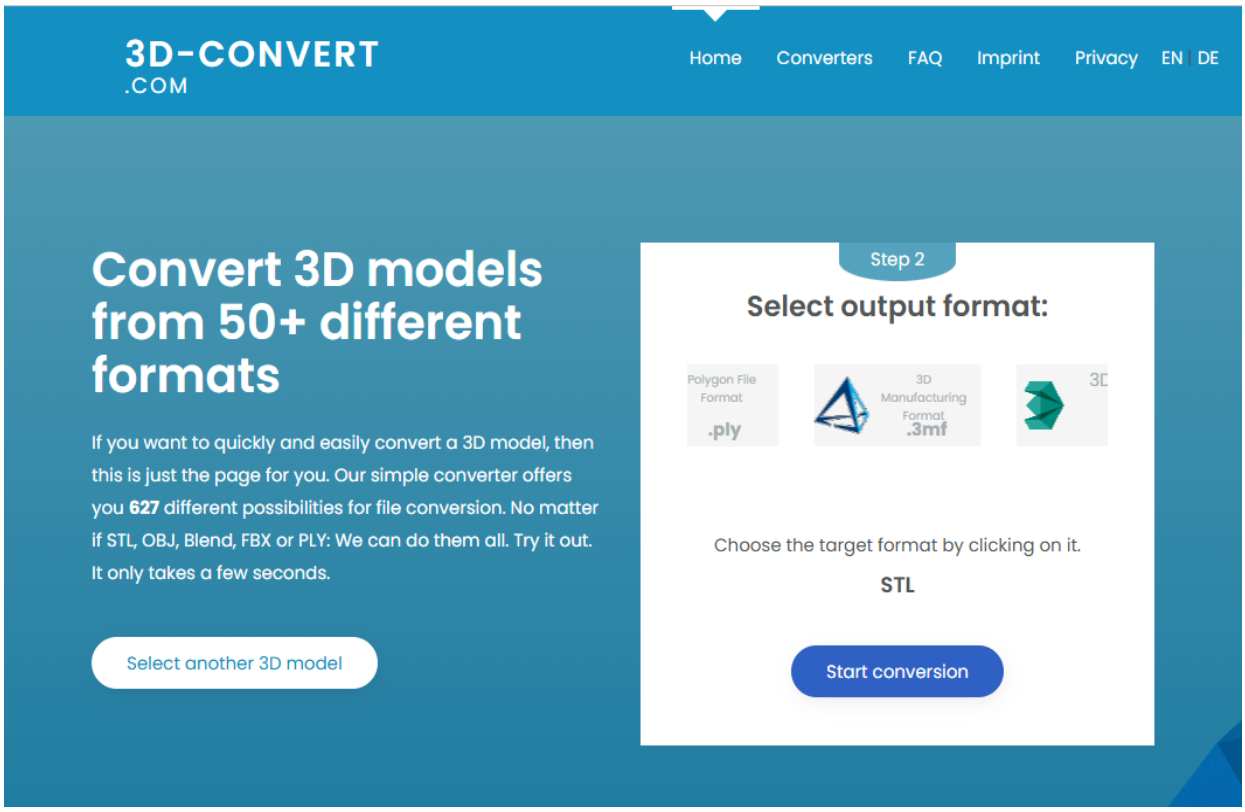


Рисунок 3 — Головна сторінка 3D-convert.com

Sketchfab — це публічний магазин 3D-об'єктів, що надає сервіси для візуалізації і базової конвертації файлів. Це кросплатформений браузерний сервіс, що базується на WebGL, доступний на будь-якому пристрої з підтримкою сучасних браузерів. Основна функція — розміщення 3D-моделей для перегляду, обміну чи продажу, але платформа також підтримує імпорт і експорт різних форматів із можливістю автоматичної конвертації. (див. рис. 4)

Переваги:

- Підтримка великої кількості форматів.

- Кросплатформеність.
- Швидкість сервісу не залежить від потужності системи користувача.

Недоліки:

- Моделі зберігаються на сервері, і доступні для перегляду всім користувачам платформи.
- Обмеження на розмір завантажуваних файлів.
- Конвертація моделей обмежена, без платної підписки.
- Потребує реєстрації користувача.

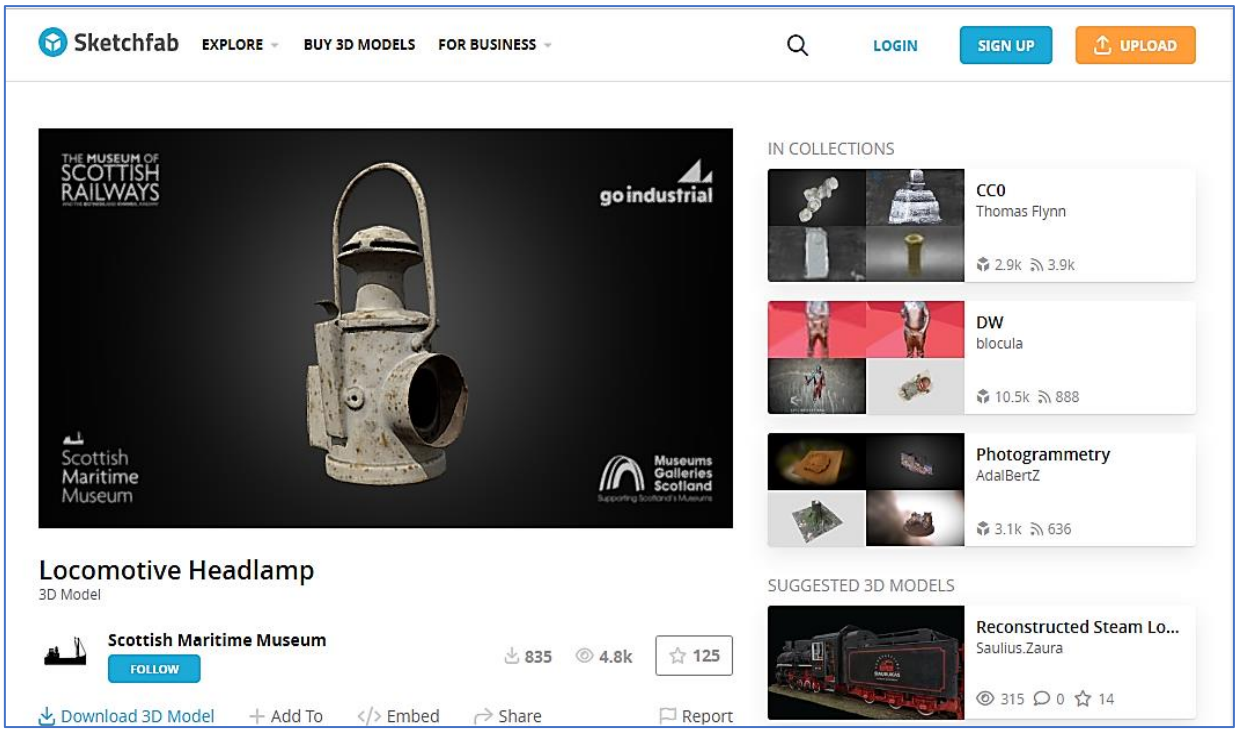


Рисунок 4 — Приклад перегляду моделі на sketchfab

MeshLab — відкритий кросплатформений застосунок для обробки та конвертації 3D-сіток. Доступний на всіх основних комп’ютерних платформах. Основна мета застосунку це очищення, аналіз, та редагування трикутних сіток — з їх подальшою конвертацією в інші формати. (див. Рис.5)

Переваги:

- Підтримка великої кількості форматів як для імпорту, так і для експорту.
- Всі дані обробляються локально, на системі користувача.
- Має прості інструменти очищення геометричних сіток.

Недоліки:

- Працює тільки з трикутними сітками. Полігони більшого розміру розбиваються на малі.
- Обмежена підтримка текстур та матеріалів.

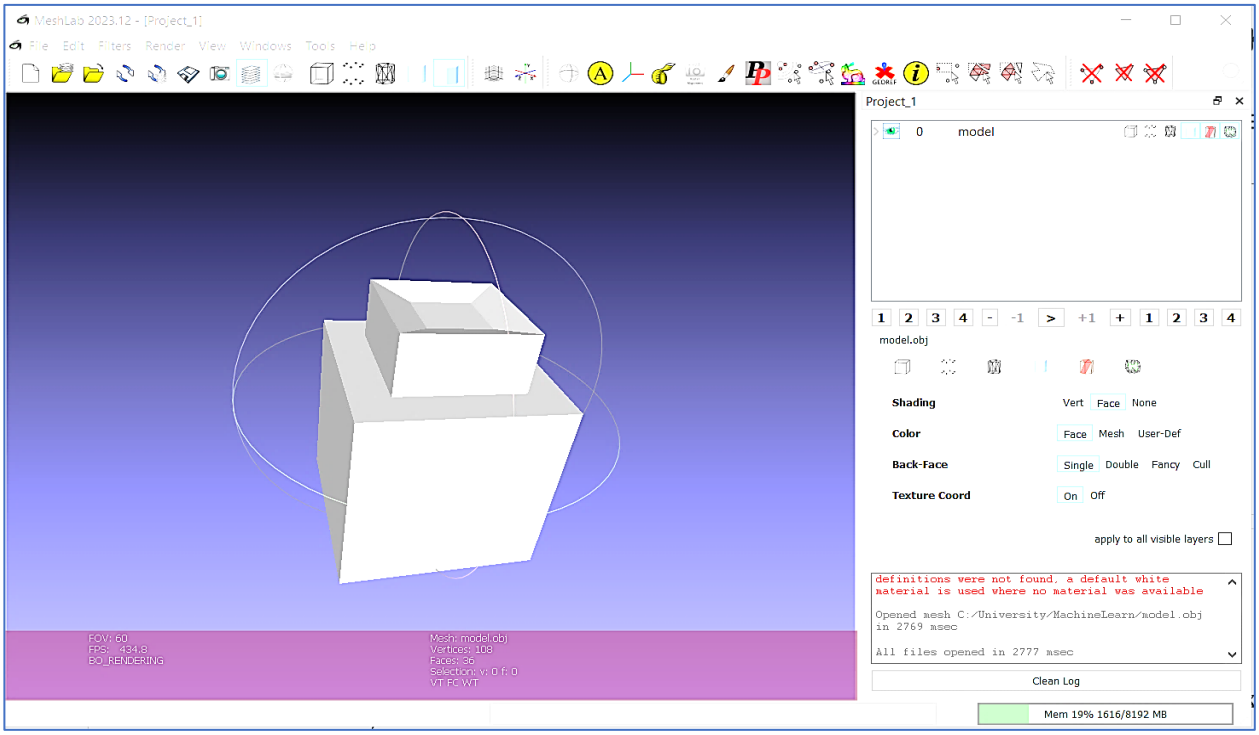


Рисунок 5 — Інтерфейс MeshLab

Аналіз розкрив велику кількість різних програмних рішень, що повністю, чи частково, втілюють функціонал конвертації файлів 3D-об’єктів. Однак для більшості знайдених рішень конвертація не є основним функціоналом, це може негативно вплинути на користувацький досвід. Інші продукти вимагають завантаження файлів 3D-об’єктів до сервера, що залишає

дані користувача без захисту. Отже можна зробити висновок, що розробка системи конвертації файлів, що виконує конвертацію як основну функцію, і не потребує компрометації конфіденційності даних користувача, є доцільною і перспективною задачею. Розробка подібного застосунку може зробити задачу конвертації файлів 3D-об’єктів набагато зручнішою, інтуїтивною, і безпечнішою.

2.3 Особливості 3D у веб середовищі

Стандарт HTML5, оприлюднений у 2008 році, став основою для інтеграції 3D-графіки у веб-середовище. Ключовим моментом стала розробка WebGL консорціумом Khronos group у 2011 році [15], API для javascript яке, використовуючи OpenGl, забезпечує доступ до апаратного 3D-прискорення прямо в браузері. На сьогодні, найпопулярнішим рішенням для веб 3D-графіки є WebGL 2.0. Альтернативою є, на момент написання цієї роботи ще нестабільний, WebGPU, що надає розробникам низькорівневий доступ до графічного процесора.

3D-графіка у веб-середі стикається з деякими проблемами. Найбільша з них — це обмеження щодо доступних ресурсів. На відміну від застосунків специфічних до платформ, веб-сторінки повинні працювати на всіх сучасних системах — через це розробники повинні йти на компроміси, що негативно впливають на продуктивність веб-застосунків загалом.

Найскладніший спосіб роботи з 3D у веб середовищі — безпосередньо через WebGL. Такий підхід дозволяє розробнику найбільшу ступінь свободи і найменше обмежень при реалізації 3D-графіки, але потребує багато теоретичних і прикладних знань від загальних алгоритмів 3D-рендерингу, до специфічної документації WebGL. Швидше, простіше, і популярніше є використання бібліотеки чи фреймворка. Це дозволяє не тільки прискорити процес розробки, але й уникнути ситуації “створення велосипеду”.

Популярними бібліотеками для webGL є Three.js, Babylon.js і playCanvas. Вони дозволяють спростити роботу з WebGL, створюючи абстраговані поняття 3D-об’єкту, сцени, і функції для керування ними. PlayCanvas є рішеннями з низьким рівнем коду. Це повноцінний двигун для розробки ігор на базі HTML5, Javascript та webGL. Він надає користувачу деталізований інтерфейс для маніпулювання 3D-сцінами, але більша частина коду не підтримує редагування. Babylon.js є потужним фреймворком WebGL, який теж використовується для створення 3D-ігор і застосунків. На відміну від PlayCanvas, він не має інтерфейсу, і надає розробнику велику кількість функціоналу, обробляючи ієрархічну структуру сцени, фізичні симуляції, анімації, тощо. Хоча потужний, цей фреймворк є дуже спеціалізованим, і надає перевагу важким застосункам з високою інтерактивністю. Three.js — найпоширеніший варіант. Незважаючи на свою високорівневість, Three.js підтримує інтеграцію елементів з “сирого” WebGL. Крім того Three.js є модульним, дозволяючи користувачам прибирати зайві, і додавати потрібні компоненти. Він є найпростішим для модифікації, і, потенційно, найлегшим з розібраних.

WebGPU має потенціал обійти WebGL у популярності і ефективності. Це API для графічних процесорів наступного покоління, яке відкриває веб-середі доступ до низькорівневих функцій GPU. Це забезпечить кращу оптимізацію, та додатковий функціонал, недоступний у WebGL 2.0. Однак на момент написання цієї роботи стандарт webGPU є нестабільним, не підтримується великою кількістю систем, і не є добре документованим.

2.4 Особливості роботи з Three.js

Three.js — це бібліотека JavaScript з відкритим вихідним кодом, створена для спрощення роботи з 3D-графікою, шляхом абстрагування WebGL [16]. Вона дозволяє користувачам працювати з об’єктами, матеріалами, сценами, світлом, абстрагуючи складний інструментарій WebGL. На момент

написання цієї роботи, Three.js є найпопулярнішою бібліотекою для розробки інтерактивних 3D-сцен.

Three.js використовує об’єктно-орієнтований і ієрархічний підхід до організації 3D-сцени. Сама сцена Scene є центральним кореневим об’єктом, що містить всі інші абстракції. Базовим є тип Object3D. Він визначає основні властивості об’єкту в 3D середовищі, такі як базові трансформації (position, rotation, scale).

Об’єкт Mesh є більш спеціалізованим і описує вже повноцінний 3D-об’єкт. Він поєднує екземпляр геометричної сітки (формат Geometry, чи BufferGeometry), і матеріалу (Material).

Geometry і BufferGeometry визначають геометричну сітку об’єкту. Geometry зберігає дані у форматі json, а BufferGeometry використовує типізовані масиви.

BufferGeometry містить набір атрибутів для опису геометричної інформації. Position — містить координати вершин (x, y, z). Normal — вектори нормалей для розрахунку освітлення. uv — координати UV розгортки для накладання зображень. index — індекси, що визначають, як вершини формують трикутники. Ці дані зберігаються у вигляді одновимірних масивів.

Material описує матеріали, застосовані до поверхні 3D-об’єкту. Вони, відповідно до використаного шейдеру, діляться на підвиди: MeshBasicMaterial для шейдеру типу constant, MeshLambertMaterial для Lambert, MeshPhongMaterial для Phong. Three.js також підтримує PBR шейдери через об’єкти MeshStandardMaterial та MeshPhysicalMaterial. Крім цього об’єкт ShaderMaterial дозволяє користувачу створити свій особливий шейдер використовуючи WebGL код.

Матеріали, крім числових значень, можуть отримувати дані з текстур. У Three.js текстури завантажуються через об’єкт TextureLoader, який завантажує текстури методом load().

Three.js є ідеальним вибором для використання в цьому застосунку через його відкритість.

3 РОЗРОБКА ЗАСТОСУНКУ ДЛЯ ІМПОРТУ ТА ЕКСПОРТУ 3-D ФАЙЛІВ

3.1 Опис проблеми

Предметна область цієї роботи торкається загальної сфери комп’ютерної, а точніше тривимірної графіки, і веб-розробки. Вона поєднує знання полігональної 3D-графіки та аналіз форматів для збереження 3D-об’єктів і 3D-сцен, і з методами браузерної візуалізації 3D-моделей. У сучасному середовищі існує велика кількість форматів збереження 3D-інформації, і проблема конвертації між ними залишається актуальною.

Метою роботи є створення легкого і інтуїтивного застосунку для полегшення задачі конвертації між 3D-форматами, через створення веб-застосунку, який дозволить навіть початківцям конвертувати файли. Застосунок повинен бути завжди доступним, тому найбільшу увагу буде приділено клієнтській частині, на якій і будуть проходити всі перетворення.

Робота зосереджується на аналізі популярних форматів для збереження 3D-файлів, дослідженню їх внутрішньої структури, сфери застосування, і алгоритми конвертації. Це дозволить розробити ідеальний формат-посередник для збереження мінімального обсягу інформації про 3D-сцени, потрібного для їх відтворення.

Крім цього, в роботі досліджені базові механізми візуалізації і редагування 3D-сцен. Для роботи з 3D у браузері, використано бібліотеку Three.js замість WebGL. Всі інші алгоритми буде розроблено вручну, з урахування внутрішнього формату представлення 3D-інформації і специфікою кожного формату окремо.

Завдяки модульній архітектурі MVC (модель-представлення-контролер), з орієнтацією на події, а також використання патерну “стратегія” для вибору функції імпорту експорту, система є легко розширюваною і відкритою до подальших модифікації.

3.2 Вимоги до програмної системи

Вимоги до системи повинні чітко відображати її головну ціль, в цьому випадку це надання користувачу будь-якого рівня обізнаності в сфері 3D простого і інтуїтивного інструменту для конвертації між поширеними форматами 3D-файлів.

З цього можна сформувати низку функціональних, і нефункціональних вимог.

Функціональні вимоги:

- Підтримка імпорту 3D-моделей у форматах: .obj, .gltf, .glb, .stl, .ply, .dae.
- Підтримка експорту 3D-моделей у форматах: .obj, .gltf, .stl, .ply, .dae.
- Можливість експорту окремих елементів сцени.
- Відображення технічної інформації про поточну сцену: кількість об’єктів, полігонів, вершин, об’єм пам’яті, тощо
- Інтерактивний перегляд 3D-об’єктів із налаштуванням сцени.
- Підтримка та редагування матеріалів і текстур.

Нефункціональні вимоги:

- Використання розробленого внутрішнього формату-посередника.
- Обробка помилок і винятків, для некоректних чи пошкоджених файлів.
- Можливість розширювати застосунок новими форматами та функціями.
- Клієнтська обробка даних для швидкодії, безпеки, та мінімального навантаження серверу.
- Низький поріг входження.
- Підтримка популярних браузерів і кросплатформність.

Ці вимоги визначають необхідні характеристики і функціонал, який повинен бути реалізован в фінальній версії розробленого продукту. Проектування та розробка буде проведена з урахуванням цих вимог.

3.3. Архітектура програмної системи

Важливою частиною задоволення вимог є вибір правильної архітектури системи. Для забезпечення безпеки користувацьких даних, низького навантаження хостингових серверів, і вимоги щодо доступності і швидкої системи було вирішено розробляти застосунок клієнтської архітектури. Такий підхід дозволяє перенести відповідальність за обробку даних на пристрій користувача, що майже повністю усуває потребу в робочому сервері, надає найбільшу доступність, надійність і швидкодію.

Модульність і масштабованість досягається використання модифікованої архітектури MVC (Model View Controller). Це дозволяє розбити систему на 3 основні частини. Model (модель) — компонент, що керує зберіганням, отриманням, оновленням і видаленням даних. Цей компонент буде описувати клас і основні методи проміжного формату (Internal representation), містити поточний стан системи (state), і описувати алгоритми імпорту/експорту підтримуваних форматів (import/export). View (представлення) — це візуальний компонент системи, з яким працює користувач. Він містить функції-конструктори для елементів користувацького інтерфейсу (UI Elements), без реалізації функціоналу. Controller (Контроллер) — це посередник між представленням і моделями. Він містить логіку, яка керує представленнями (UI controllers): створює і наповнює їх інформацією, обробляє викликані ними події, приймає і обробляє форми. Крім цього, цей елемент відповідає на ініціалізацію користувацького інтерфейсу, і відображення 3D-сцен. Всі основні компоненти детально розібрано в розділах 3.2-3.4. Схематичне представлення архітектури можна подивитись на UML діаграмі моделі (Рис. 6).

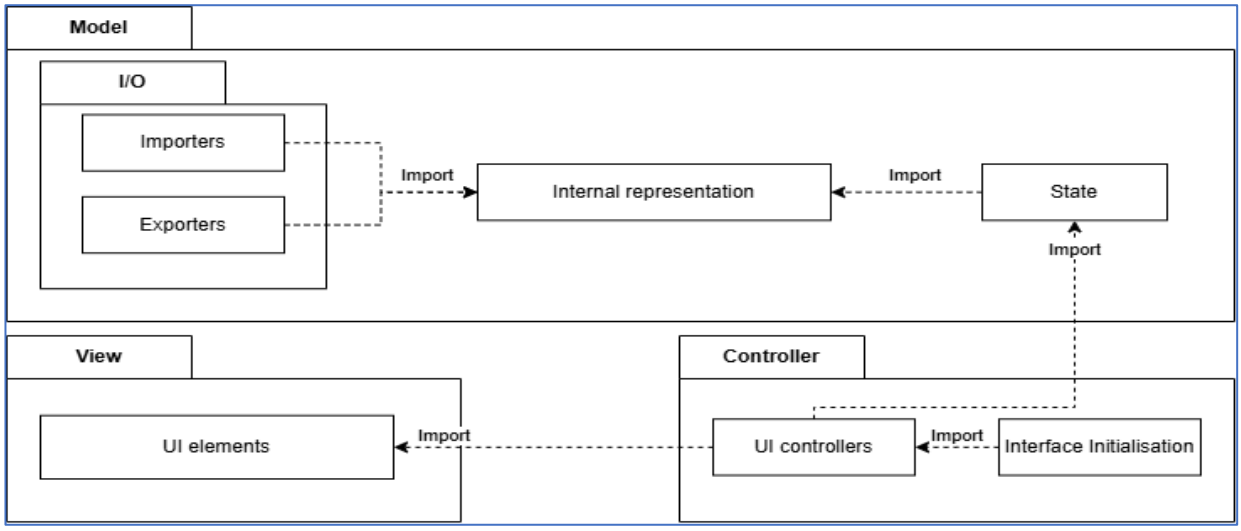


Рисунок 6 — UML діаграма пакетів

Використання такого архітектурного шаблону дозволяє зробити систему модульною, і знизити зчеплення між елементами. Кожен контролер є автономним класом з власним асоційованим представленням, що дозволяє додавати чи видаляти компоненти без рефакторину. Модульність підтримуваних файлів забезпечується завдяки реалізації патерну програмування “Стратегія”. Допоміжний файл FormatMap містить мапу підтримуваних форматів і відповідні їм функції імпорту/експорту. Це дозволяє значно пришвидшити додавання чи видалення нових форматів.

На початковому етапі проєктування архітектури також було розглянуто шаблон MVVM (Model View ViewModel), і MVP (Model View Presenter). MVVM добре підходить для двостороннього зв’язку між інтерфейсом і внутрішніми даними, але його було відкинуто через високу складність прив’язки між собою елементів View і ViewModel. В середовищі браузеру, це негативно вплинуло б на продуктивність. MVP було відкинуто через більш пасивну роль представлень у ньому — через зменшення зв’язування між View та Presenter, динамічне оновлення даних у користувацькому інтерфейсі було б складніше реалізувати.

3.2 Розбір компоненту Model

Компонент модель не тільки надає доступ до даних, але й виступає контейнером їх збереження. Розберемо модулі, що існують в цьому компоненті. Їх повний перелік надано на рис. 7.

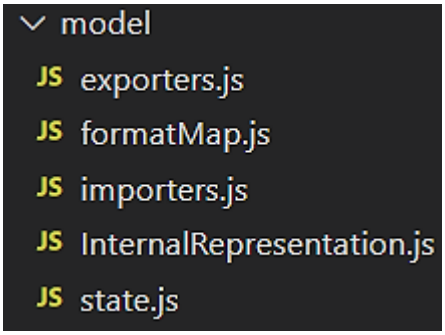


Рисунок 7 — Модулі компоненту Model

Модулі імпортерів і експортерів представлено файлами exporters.js і importers.js. Вони містять список функцій імпорту та експорту для кожного формату, тобто функцій парсингу, з перетворенням інформації до внутрішнього формату-посередника, і функції побудови вихідного формату.

Агрегація та доступ до функцій імпорту та експорту реалізується через мапу форматів, описану у файлі formatMap.js. Для кожного підтримуваного формату зберігається принаймні один запис, що містить посилання на функцію його імпорту, функцію його експорту, і список розширень файлів, що співпадають з ним. Загальний приклад цього запису можна побачити на лістингу 16.

Лістинг 16 Загальний вигляд запису у formatMap.js

```
export const formatMap = {
    'Format Name':{
    importer:<Importer Function>,
    exporter:<exporterFunction>,
    extensions:['.extension'],
```

```
  }
};
```

Така архітектура дозволяє легко реалізовувати паттерн “стратегія” у контролерах імпорту та експорту, які буде оглянуто детальніше у частині 3.3.

Поточний стан 3D-сцени знаходиться у файлі state.js. Крім зберігання поточного стану системи, цей модуль також надає доступ до функцій її зміни, і є центром виклику подій. Для збереження низки інформації про поточний стан системи він містить внутрішній об’єкт state.

Поле InternalData зберігає екземпляр внутрішнього представлення, визначеного у файлі InternalRepresentation.js. LightPreset посилається на поточні налаштування світла (описано у частині 3.3), exportMeshIndex описує індекс поточного обраного об’єкту в 3D-сцені, а textures містить мапу текстур.

Цей модуль використовує патерн “спостерігач” для керування оновленням візуальних даних застосунку. В цьому випадку сам state.js є об’єктом спостереження, а контролери, підписані на нього — спостерігачами. Це реалізовано через механізм подій, що використовує вбудований в браузер API EventTarget.

Для доступу до даних використовуються функції-аксесори, по типу setCurrent(data) та getCurrent(data). При оновлені даних через функції типу set, викликаються події використовуючи функцію dispatchMeshUpdate(type).

3.2.1 Внутрішнє представлення даних

Процес конвертації файлів 3Д моделей між собою є складним процесом, який потребує розробки алгоритмів для кожної пари форматів. Якщо додавати до застосунку n форматів, і дозволяти перетворення кожної пари форматів то кількість потрібних алгоритмів становитиме n*(n-1), тобто зростатиме квадратично. Цю проблему можна вирішити використанням формату-посередника для збереження інформації. Це знизить кількість розроблених алгоритмів до 2*n, тобто до двух алгоритмів за формат. Схему розробленого

формату описано у лістингу 17. Формат можна розбити на 2 головні частини: meshes та materials.

Лістинг 17 — Схема внутрішнього формату

```
"metadata":{
  "name": "string",
  "source": "string"
},
"geometry": {
  "name": "string",
  "positions": [[float, float, float], ...],
  "normals": [[float, float, float], ...],
  "uvs": [[float, float], ...],
  "faces": [
    {
      "vertices": [int, int, int, ...],
      "normals": [int, int, int, ...],
      "uvs": [int, int, int, ...],
      "materialIndex": int
    },
    ...
  ],
  "location": [float, float, float],
  "rotation": [float, float, float, float],
  "scale": [float, float, float],
  "children": [int, ...]
},
"materials": [
  {
    "name": "string",
    "shaderType": "string",
    "doubleSided": boolean,
    "color": [float, float, float],
    "alpha": float,
    "emissive": [float, float, float],
    "roughness": float,
    "specular": [float, float, float],
    "ambient": [float, float, float],
    "metallic": float,
    "colorTexture": "string" | null,
    "emissiveTexture": "string" | null,
    "ambientTexture": "string" | null,
    "specularTexture": "string" | null,
    "metallicTexture": "string" | null,
    "roughnessTexture": "string" | null,
    "normalTexture": "string" | null,
```

```
      "alphaMode": "string",
      "alphaCutoff": float
    },
    ...
  ]
```

Структура geometry містить перелік всіх об’єктів, які є в сцені. Розбір полів описано у Таб. 9 і Таб. 10.

Таблиця 9 – Структура класу Mesh

Поле	**Формат**	**Призначення**
`name`	`“name”`	Ім’я поточного об’єкту
`positions`	`[x,y,z]…`	Масив вершин, представлений масивами декартових координат.
`normals`	`[x,y,z]…`	Масив векторів нормалей.
`uvs`	`[u,v]…`	Масив координат для розгортки.
`faces`	`{face}…`	Масив полігонів (детальніше у табл. 15)
`location`	`[x,y,z]`	Декартові координати зсуву об’єкту
`rotation`	`[x,y,z,w]`	Вектор обертання об’єкту формату кватерніону
`scale`	`[x,y,z]`	Вектор розміров об’єкту по осях x, y, z.
`children`	`[int,int…]`	Масив посилань на дочірні об’єкти.

Таблиця 10 – Структура об’єкту face

Поле	**Формат**	**Призначення**
`vertices`	`[int,int…]`	Індекси, що посилаються на масив positions.
`normals`	`[int,int…]`	Індекси, що посилаються на масив normals.
`uvs`	`[int,int…]`	Індекси, що посилаються на масив uvs.
`material Index`	`int`	Індекс матеріалу на визначеному полігоні.

Поля Positions, normals, uvs містять перелік координат і векторів, тоді як полігони посилаються на них (посилання дублюються для кожної вершини

полігону). Масив Children дозволяє додати просте представлення ієрархії збереженням індексів посилань на дочірні елементи. Location, rotation, scale існують для підтримки стандартних не-деструктивних трансформації, а розташування посилань на матеріали всередині кожного індивідуального полігону відкривають підтримку для 3D-об’єктів з кількома різними матеріалами.

Структура materials розроблена з підтримкою 4 основних типів шейдерів, зазначених у змінній shaderType. Це constant, phong, lambert, і pbr. Для їх комбінації було обрано мінімальну кількість полів, потрібну для їх відтворення. Опис кожного поля, і їх призначення описано у Таб. 11.

Таблиця 11 – Структура класу Material

Поле	**Призначення**	**Опис**
`name`	загальний	Назва матеріалу.
`shaderType`	загальний	Тип шейдеру.
`DoubleSided`	загальний	Чи відображати протилежну сторону об’єкту.
`color`	загальний	Колір об’єкту [r,g,b] .
`alpha`	загальний	Значення прозорості.
`emmisive`	lambert, phong, pbr	Колір сяйва об’єкту [r,g,b].
`roughness`	pbr	Шорсткість об’єкту.
`specular`	phong	Коефіцієнт відбиття.
`ambient`	phong	Колір навколишнього світла.
`metallic`	pbr	Значення металевості об’єкту.
`_Texture`	варіативно	Текстурне значення для відповідного поля.
`normal Texture`	pbr	Мапа нормалей.
`AlphaMode`	загальний	Тип прозорості.

`AlphaCutoff`	загальний	Значення активація для прозорості типу “маска”

Розглянувши схему формату, можна описати внутрішню структуру файлу InternalRepresentation.js. Діаграма класів цього файлу подана на Рис. 8

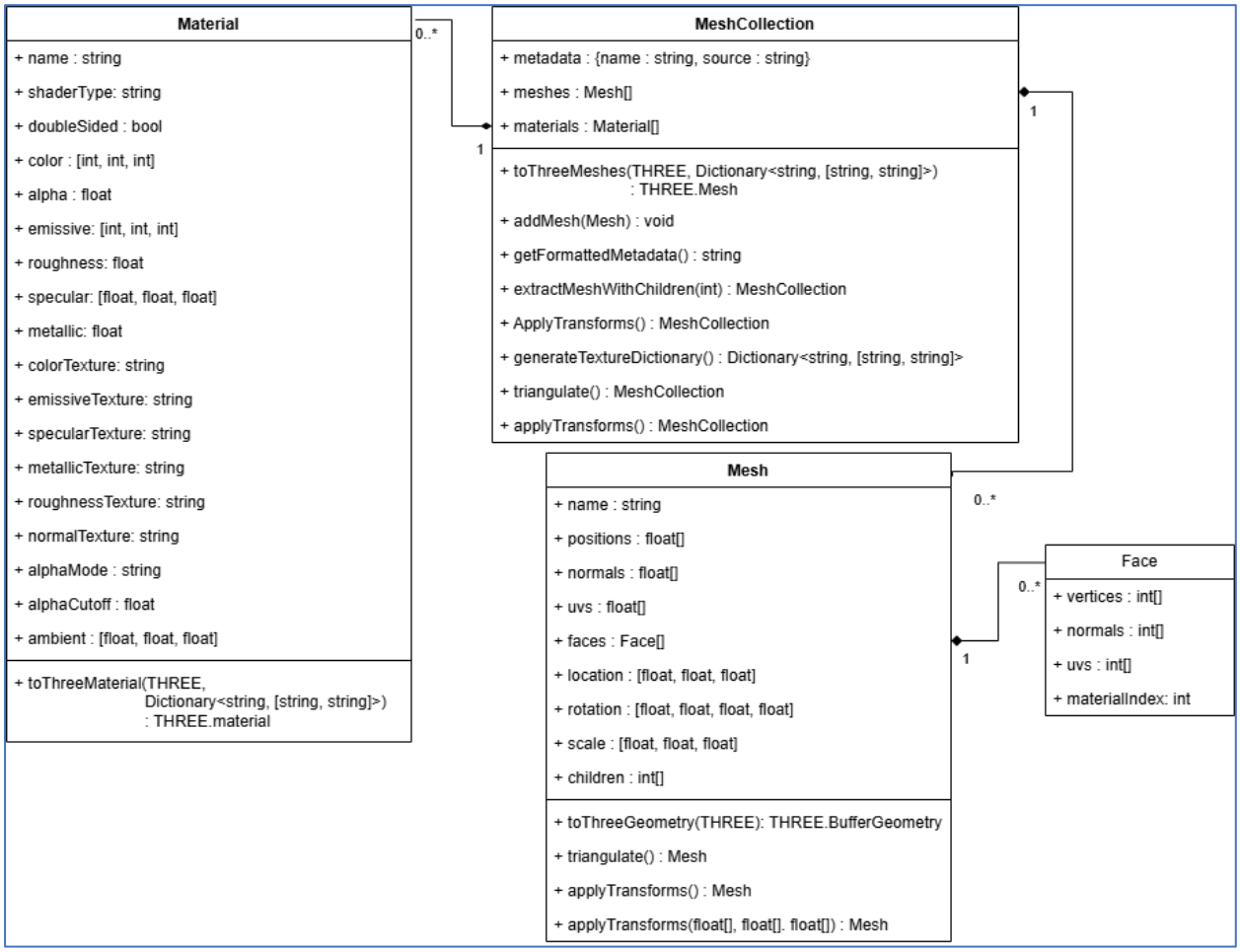


Рисунок 8 — Діаграма класів модулю InternalRepresentation

Ця схема описує 3 основні класи: Mesh, що відповідає елементам geometry в схемі формату з лістинга 17, Material, що відповідає вмісту materials, і MeshCollection, що є контейнером з допоміжними функціями. Слід зазначити, що ці класи є нащадками Serializeable, з визначеними функціями toJSON і fromJSON що дозволяє їм серіалізуватись у формат json.

Крім загачених класів, цей файл також неявно описує структуру TextureDictionary, що створюється у класі MeshCollection функцією generateTextureDictionary(). Це — асоціативний масив, де ключами є назви текстур зазначених в матеріалах, і пари типу сирого blob файлу картинки (зберігається для подальшого експорту), і THREE.TextureLoader(), для подальшого відображення.

Кожен клас містить функцію, для перетворення у формат, сумісний з Three.js. Так, функція ToThreeMaterial класу Material створює і повертає об’єкт шейдеру, відповідного зазначеному типу в полі shaderType. Constant відповідає MeshBasicMaterial, lambert — MeshLambertMaterial, phong — MeshPhongMaterial, і pbr — MeshStandardMaterial. Крім цього цей елемент завантажує текстури, якщо йому передано об’єкт textureDictionary.

ToThreeMesh зтикається з проблемою несумісності даних, оскільки Three.BufferGeometry() підтримує тільки трикутники. Тому метод toThreeGeometry реалізує алгоритм тріангуляції для перетворення полігонів у трикутники, сумісні з Three.js, а також забезпечує коректне перенесення даних про вершини, нормалі, UV-координати та матеріали. Реалізовано алгоритм віялової тріангуляції, при якому n-кутник розбивається на n-2 трикутники. Для цього обирається основна точка (яка буде у всіх трикутниках), і до неї додаються по дві послідовні точки. Крім цього будуються геометричні групи трикутників для кожного матеріалу.

ToThreeMeshes класу MeshCollection створює масив об’єктів формату Three.Mesh, що поєднує геометрію і використані в ній матеріали.

Крім цього, клас MeshCollection містить низку інших методів:

- addMesh(meshData) : void — додає до поля meshes новий екземпляр класу Mesh.
- getFormattedMetadata() : String — повертає текстовий об’єкт, що містить мета-опис поточних збережених даних: кількість точок, полігонів, займаного місця, і т.д.

- extractMeshWithChildren(index) : MeshCollection — створює новий екземпляр MeshCollection що містить тільки об’єкт, що знаходиться по заданому індексу. Якщо об’єкт має дочірні об’єкти — вони теж потрапляють в цей об’єкт.
- generateTextureDictionary() : Dictionary<string:[blob, Three.TextureLoader]> — створює порожній екземпляр словника текстур, в якому вже містяться незаповнені записи кожної текстури, яка фігурує в матеріалах.
- triangulate() : MeshCollection — повертаю копію MeshCollection, в якому всі екземпляри Mesh состоять з трикутників.
- applyTransforms() : MeshCollection — повертаю копію MeshCollection, в якому до всіх екземплярів Mesh було застосовано трансформацій location, rotation і scale.

В результати, розроблений формат дозволяє зберігати дані про геометричну сітку об’єктів, матеріали, і текстури. Формат не підтримує скінінг, спеціальні вершинні дані, анімації, чи морфінг.

3.2.2 Імпорт 3D-моделей

Для конвертації інших форматів файлів у формат внутрішнього представлення використовується модуль importers.js. Він описує стандартизовані функції парсингу файлів для підтримуваних форматів, що, в ході роботи, створюються екземпляри класів Mesh, Material і MeshCollection з модулю InternalRepresentation.js.

Для використання патерну Стратегія, ці функції повинні бути чітко стандартизовані, мати однакові сигнатури — приймати однакові параметри, і повертати однакові результати. Розберемо їх загальну структуру на прикладі importJSON на лістингу 18.

Лістинг 18 — Код функції importJSON

```
    export async function importJSON(files) {
return new Promise((resolve, reject) => {
```

```
  const jsonFile = files.find(f =>
f.name.toLowerCase().endsWith('.json'));
  if (!jsonFile) {
    reject(new Error('No .json file found in the selected
files'));
    return;
  }
  const reader = new FileReader();
  reader.onload = (event) => {
  try {
    const jsonContents = JSON.parse(event.target.result);
    const collection = new MeshCollection(jsonContents);
    resolve(collection);
  } catch (error) {
    reject(new Error(`Failed to parse JSON:
${error.message}`));
  }
 };
 reader.onerror = () => reject(new Error('Error reading
JSON file'));
 reader.readAsText(jsonFile);
});
}
```

Функції імпорту мають уніфіковану сигнатуру: async Function<File[], MeshCollection>(input: File[]): meshCollection. Кожна функція отримує довільну кількість файлів, з яких вона повинна відокремити потрібні. Це зроблено з огляду на формати що потребують декілька файлів, наприклад obj чи gltf. Такий підхід підтримує одночасне завантаження текстур, разом з імпортом 3D-файлів. Функції імпорту є асінхронними, це дозволяє обробляти їх без блокування інтерфейсу користувача. Кожна функція здатна обробляти помилки. Нарешті, всі функції повертають єдиний екземпляр формату MeshCollection.

Далі в цьому розділі буде розібрано кожну функцію імпорту окремо. Через довжину реалізації, кожна функція буде описуватись як алгоритм, а лістинги надаватись тільки в частинах, які потребують додаткового пояснення.

Функція **importJSON**, доступна для перегляду в лістингу 18, використовує здатність внутрішнього формату для серіалізації з json. Вона

знаходить файл, будує json об’єкт, і використовує конструктор класу MeshCollection.

Функція **importOBJCustom** створена для парсингу файлів у форматі .obj. Першим кроком серед завантажених файлів проходить пошук самого файлу obj, і файлу mtl. Їх структуру було описано в таблицях 1 і 2 відповідно.

Спочатку проходить парсинг формату mtl. За його відсутністю, створюється базовий матеріал, і подалі полю materialIndex кожного полігону присвоюється індекс 0. Коли парсер виявляє на newmtl, створюється новий матерал із зазначеною назвою. Значення Ka, Kd, Ks, Ke напряму переносяться, відповідно, до полів ambient, color, specular, emmisive класу Material. Поле Ns відображає шорсткість цілим числом в діапазоні між 0 і 1000, тому для перенесення його до roughness необхідно поділити значення на 1000. Значення illum було умовно поставлено в відповідність до певних shadertype: 0 — constant, 1 — lambert, 2 — phong, 3+ — pbr. Значення d відповідає прозорості, воно переноситься у alpha. Текстури Map_Kd, map_Ks, map_Ke, map_Ka відповідно переносяться у colorTexture, specularTexture, emissiveTexture, ambientTexture, як імена файлів.

Файл формату OBJ є обов’язковим для імпорту 3D-об’єкту. Поля v, vn і vt відповідають полям positions, normals, uvs класу Mesh, і переносяться без змін. Полігони у форматі obj мають глобально інкрементовані індекси, тому при виявлені нової геометричної групи, що починається з o чи g, потрібно зберігати змінні типу offset, з поточною довжиною positions, normals, uvs, і, перед збереження індексів у конструкцію faces, корегувати їх.

Функція **importSTLCustom** створена для парсингу бінарних і ASCII файлів у форматі .stl. Цей формат не підтримує текстур, і не розбивається на декілька файлів — тому парсер генерує виключення, якщо користувач обрав більше одного файлу. Парсер зчитує перші 80 байт файлу і перевіряє їх на відповідність ASCII чи бінарному формату. Відповідно до результату, він передає файл функціям ImportSTLASCII, або importSTLBinary.

ImportSTLASCII аналізує файл STL у випадку, коли його передано текстом. При знаходженні рядку facet normal, парсер зберігає нормаль як поточну, і починає обробляти три рядки vertex, зі значеннями вершин полігону. Коли парсер потрапляє на endloop, то дані поточної нормалі і вершин додаються до normals і positions, також формується полігон.

importSTLBinary аналізує STL у випадку, коли його передано у бінарному вигляді. Пропустивши перші 80 байт метаданих, алгоритм зчитує кількість трикутників, і, для кожного трикутнику, читає по 50 байт. Перші 12 байт — вектор нормалі, наступні 36 — координати вершин. Ці дані обробляються згідно з текстовими аналогами.

Оскільки .stl має одну нормаль на трикутник, а внутрішній формат вимагає нормаль на кожну вершину, нормаль трикутника зберігається один раз, а в faces додаються три однакові індекси нормалей. STL також не підтримує UV розгортку, для сумісності замість UV величин до uvs додається єдинє значення [0,0], і всім точка присвоюється UV за індексом 0. Крім того у форматі STL є проблема дуплікації вершин, кожен трикутник, який ділить вершину з іншими трикутниками — дублює її у файлі. Для усунення повторення вершин по декілька разів, було використано об’єкт vertexmap, словник, що зберігає індекс вершини за текстовим ключем “x y z”. Перевіряючи цей ключ, і уникаючи повторів вершин.. Результатом парсингу є MeshCollection з матеріалом за замовчуванням, і єдиний Mesh.

Функція **importPLYCustom** призначена для імпорту бінарних і текстових (ASCII) файлів формату PLY. Першим кроком алгоритм зчитує заголовок файлу, розібраний у табл. 5. Алгоритм будує внутрішню структуру metadata, з повною інформацією про вміст файлу.

Відповідно до формату, подальший парсинг делегується функції **parsePLYASCII** чи **parsePLYBinary**, яким передається буфер і metadata. Обидві функції, для кожної вершини, зчитують 3, або (при наявності нормалей і UV розгортки) 6 чи 8 значень. Перші три додаються як координати до positions, наступні 3 — до normals, останні 2 — до UVs. У бінарному форматі

зчитування вершин і граней базується на зміщеннях властивостей (propertyOffsets) і кроці (stride), визначених за vertexProperties структури metadata. Для кожного полігону, першим в строці, чи блоку даних, є число його вершин, що зберігаються у вигляді індексів посилань. Ці індекси копіюються у vertices, normals і uvs. У разі відсутності у файлі даних про нормалі, вони обчислюються як перпендикуляр по трьом довільним точкам полігону, додаються до масиву об'єкту normals, а їх індекс зберігається у полі структури faces, normals — по одному на вершину. Якщо відсутня UV розгортка — для сумісності, UV представляється одним значенням [0,0]. Результатом парсингу є екземпляр MeshCollection з одним екземпляром Mesh, і матеріалом за замовчуванням.

Функція **importGLTFCustom** призначена для імпорту файлів у форматі .gltf, які можуть супроводжуватися бінарним файлом .bin або містити вбудовані бінарні дані. Спочатку алгоритм перевіряє наявність .gltf файлу серед переданих. Якщо його немає, генерується виключення. Далі зчитується вміст .gltf файлу як JSON, а також запускається пошук .bin. Якщо .bin відсутній, парсер намагається витягнути вбудована бінарну інформацію з gltf.buffers. Парсинг даних делегується функції **processGLTF**, яка отримує вміст текстової частини gltf, бінарний поток даних, і назву файлу.

processGLTF першим кроком проходить по gltf.nodes, де визначає відповідність між нодами і геометрією, і зберігає перетворення. Для кожного мешу, алгоритм зчитує доступні примітиви (POSITION, NORMAL, TEXCOORD_0), і індекси INDICES для побудови полігонів. Для вилучення цієї інформації з бінарного буферу, створюється допоміжна функція **readAcessor(AccessorIndex, componentCount, type, normalize)**, яка знаходить accessor по індексу AcessorIndex, через нього отримує доступ до bufferView, і використовує інформацію з обох елементів щоб прочитати дані. Координати додаються до positions, нормалі — до normals, UV-координати (з інверсією v) — до uvs. Формат gltf зберігає всі поверхні трикутниками, тому полігони будуються з трьох послідовних індексів indices, чи (за відсутність indices) з

трьох послідовних точок. На цьому кроці, за відсутності нормалей, вони розраховуються по трьох точках, а за відсутністю даних UV розгортки вони заміщаються одним значенням [0,0]. Останнім кроком алгоритм формує ієрархію, додаючи до поля children кожної сітки індекси дочірніх сіток, отриманих із gltf.nodes.

За наявності матеріалів, парсер проходить по всіх членах gltf.materials. Він копіює значення у поля класу Material, згідно з відповідністю: color — baseColorFactor, alpha — baseColorFactor, roughness — roughnressFactor, metallic — metallicFactor, emissive — emmisiveFactor, alphaMode — alphaMode, alphaCutoff — alphaCutoff, doubleSided — doubleSided, colorTexture — baseColorTexture, roughnessTexture — metallicRoughnessTexture, metallicTexture — metallicRoughnessTexture, normalTexture — normalTexture, emmisiveTexture — emmisiveTexture. shaderType, тобто тип шейдеру, обирається як pbr.

Результатом парсингу є MeshCollection з масивами meshes і materials.

Функція **importGLBCustom** призначена для імпорту файлів у форматі .glb, які є бінарною версією .gltf і містять JSON-структуру та бінарні дані в одному файлі. Спочатку алгоритм перевіряє наявність .glb файлу серед переданих. Якщо його немає, генерується виключення. Далі файл зчитується як бінарний буфер через DataView.

Алгоритм перевіряє правильність структури файлу а також версію gltf. Якщо файл не викликає помилок, і відповідає підтриманій версії, алгоритм декодує JSON сегмент і зберігає бінарний поток даних, які він функції **processGLTF**.

Функція **importDAECustom** призначена для імпорту файлів формату COLLADA (.dae). Цей формат описується у структурі XML, тому для його парсингу була розроблена допоміжна функція **parseXML**, яка перетворює XML на зручну в роботі деревоподібну структуру. Основою цієї структури є елементи/ноди, загальний вигляд яких надано в лістингу 19.

Лістинг 19 Приклад конвертованого XML елементу

```
"name": "назва_тегу",
"attributes": {
  "attributen": "значення",
},
"children": [
  { },
],
"text": "текст всередині поточного тегу"
```

Першим кроком алгоритм знаходить “кореневу” ноду COLLADA, відсутність якої вказує на некоректність файлу, і викликає виключення. Якщо файл бракує головних елементів, описаних у табл. 8 (libraryVisualScenes), генерується виключення.

У Структурі effectMap зіставляються матеріали і їх ідентифікатори. Значення нод LibraryEffects переносяться у поля класу Materials відповідно:shaderType — <blinn>, <phong>, <lambers>, <constant>; color, alpha, alphaMode, colorTexture — <diffuse>; specular, specularTexture — <specular>; emissive, emissiveTexture — <emission>; ambient, ambientTexture — <ambient>; roughness — <shininess>; alpha — <transparent>; doubleSided — <double_sided>. Додатково, з ноди LibraryMaterials, отримується назва матеріалу.

Далі, рекурсивно, обробляються всі ноди visualScene, і ті, в яких є геометрична інформація — перетворюються на екземпляри класу Mesh за допомогою функції **ProcessNode.** В першу чергу ця функція знаходить ноду Mesh, відповідну поточній ноді. Джерела source надають дані для positions, normals, uvs. Полігони можуть бути трикутниками (нода triangles), чи довільними полігонами (polylist). З тегу p вилучаються індекси вершин, з яких будуються полігони. На цьому етапі, за відсутності нормалей, вони генеруються по трьох точках, а за відсутності UV розгортки — до uvs

додається елемент [0,0], і всі полігони посилаються на нього. Також з ноди matrix вилучається матриця трансформацій, яка за допомогою функції **decomposeMatrix** розбивається на location, rotation і scale. Нарешті, матеріал поточної геометрії, через materialMap, зіставляється з індексом вже обробленого матеріалу. Індекси дочірніх нод додаються до ноди батька в ході рекурсивного обходу дерева нод в глибину. Результатом функції є екземпляр MeshCollection з довільною кількістю об'єктів Mesh і Material.

3.2.3 Експорт 3D-моделей

Для експорту 3D-моделей із внутрішнього формату представлення у підтримуваний формат використовується модуль exporters.js. Він містить стандартизовані функції експорту, що конвертують екземпляр MeshCollection у цільовий формат (json, stl, ply, obj, fltf, dae). Усі функції є асинхронними, що дозволяє обробляти складні операції без блокування інтерфейсу для користувача. Для форматів, які зберігають дані в декількох окремих файлах, чи підтримуються текстури, використовується бібліотека JSzip [18], яка дозволяє, з мінімальним навантаженням на систему, створити .zip архів з потрібними файлами.

Функції експорту мають уніфіковану сигнатуру: *async function<MeshCollection, String | Blob>(input: MeshCollection, filename?: String): void*. Кожна функція приймає екземпляр MeshCollection, який містить дані про геометрію, матеріали та метаінформацію, а також необов'язковий параметр filename для назви вихідного файлу. Результатом роботи функції є створення файлу (або архіву), який завантажується через браузер.

Допоміжними функціями є **downloadFile(name, content)** та **downloadArchive(archivename, files[])**. Перший завантажує на комп'ютер користувача один файл content з назвою name, другий — створює і завантажує архів.

Далі в цьому розділі буде розібрано кожну функцію експорту окремо. Через довжину реалізації, кожна функція буде описуватись як алгоритм, а лістинги надаватись тільки в частинах, які потребують додаткового пояснення.

Функція **exportJSON** використовує властивість MeshCollection для серіалізації щоб отримати JSON структуру, яку вона завантажує на комп'ютер користувача за допомогою downloadFile.

Функція **exportSTLText** призначена для конвертації екземпляру MeshCollection у текстовий (ASCII) stl файл. Першим кроком до екземпляру MeshCollection застосовуються його методи triangulate і applyTransforms, що, відповідно, триангулюють кожний об'єкт Mesh, і застосовують до нього трансформації. Це зроблено з урахуванням того, що STL не підтримує трансформації чи варіативні полігони. Дані про нормалі та позиції точок обробляються наступними. Формат STL не підтримує UVs, а нормалі в ньому існують не для освітлення, а для орієнтації полігонів — тому в ході формування структури файлу, нормалі обчислюються по трьох точках трикутних полігонів. Нарешті, генерується структура STL зазначена у лістингу 3, для кожного полігону кожного об'єкту. В результаті текстовий stl файл завантажується через браузер.

Функція **exportSTLBinary** призначення для конвертації екземпляру MeshCollection у бінарний stl файл. Вона конвертує дані аналогічно з ExportSTLText, і запаковує їх у бінарний буфер описаний у табл. 4. В результаті бінарний stl файл завантажується через браузер.

Функція **exportPLYAscii** конвертує екземпляр MeshCollection у текстовий (ASCII) PLY файл. Спочатку застосовується метод applyTransforms для врахування трансформацій об'єктів (location, rotation, scale). Далі створюється карта вершин (vertexMap) для уникнення дублювання даних, де унікальний ключ формується з індексів позицій, нормалей та UV-координат. Алгоритм проходить по всім полігонам всіх об'єктів, і зберігає сигнатури кожної вершини у vertexMap. Якщо вершина унікальна — вона буде записана

у файл, як нова. Якщо вона повторюється — то всі подальші повторення будуть посилатись на вже існуючий екземпляр рядка.

Після ремапінгу, до порожнього файлу додається заголовок. Цей заголовок відрізняється від заголовку, зазначеного у лістингу 4 повнотою даних. При експорті, серіалізується найповніша підтримана версія PLY. Після цього до файлу додаються всі рядки вершин, і всі рядки полігонів. В результаті файл завантажується через браузер.

Функція **exportPLYBinary** конвертує екземпляр MeshCollection у бінарний PLY файл. Трансформація даних в ній проходить аналогічно з **exportPLYAscii**, але до кінця файлу додаються не текстові рядки, а буфери значень. 32 байт на вершину (8 чисел формату float), і 1 + 4*n байт на кожен полігон (де n це кількість точок). В результаті файл завантажується через браузер.

Функція **exportOBJ** конвертує екземпляр MeshCollection у формат OBJ, який складається з двох основних файлів: .obj для геометрії та .mtl для матеріалів, а також пов’язаних текстур.

Першим кроком до об’єкту MeshCollection застосовується ApplyTransforms, через відсутність в obj підтримки трансформацій.

Далі генерується вміст файлу .mtl за допомогою допоміжної функції **generateMTL**, яка формує описи матеріалів. В ньому, у відповідність полям mtl, що описано у табл. 3, ставляться поля Material, у наступній відповідності: newmtl — name, Ka — ambient, Kd — color, Ks — specular, Ke — emissive, Ns —roughness, d — alpha, doubleSided — doubleSided, map_Kd — colorTexture, map_Ka — ambientTexture, map_Ks — specularTexture, map_Bump — normalTexture, map_Pm — metallicTexture, map_Pr — roughnessTexture. Значення illum було умовно зіставлено з типами ShaderType: 0 — Constant, 1 — lambert, 2 — phong, 3+ — pbr.

Вміст файлу .obj створюється функцією **generateOBJ**. Вона записує заголовок із назвою сцени та посиланням на .mtl файл, якщо матеріали є. Для кожного об’єкта (Mesh) додаються вершини (v), UV-координати (vt, з

інверсією осі Y для забезпечення сумісності з OBJ), нормалі (vn) та полігони (f). Полігони групуються за матеріалами через usemtl, а їх індекси формуються у форматі v/vt/vn, з урахуванням зсувів для вершин, UV-координат і нормалей.

За наявності, функції також збирає текстури з модулю state.js, за допомогою функції getCurrentTextures. Файл obj, mtl і текстури групуються у zip архів функцією downloadArchive, і завантажуються через браузер.

Функція **exportGLTF** конвертує MeshCollection у формат GLTF. Через JSON-подібну структуру формату, достатньо створити внутрішній об’єкт, що описує всі головні компоненти, притаманні типовому GLTF, і серілізувати його вбудованою функцією JSON.stringify. Складною частиною є пакування всієї інформації до бінарного потоку, і створення коректних об’єктів bufferView і Accessor для доступа до неї. Всі матеріали зберігаються у вигляді PBR через брак підтримки форматом інших типів шейдерів. В результаті, файли .gltf та .bin пакуються у архів за допомогою downloadArchive, і завантажуються через браузер.

Функція **exportDAE** конвертує MeshCollection у формат COLLADA. Створення коректної XML структури ускладнено через відсутність вбудованих парсерів, або сторонніх пакетів. Тому текстова структура XML створюється вручну. При створені матеріалів — алгоритм будує поле у тегу effect для кожного поля, незалежно від значення. Геометрія, перед серіалізацією, перетворюється на трикутники, це зроблено через простоту і стабільність елементу triangles. Полігону групуються у об'єкти за матеріалами, а трансформації об'єднуються у матрицю перетворень. Результатом функції є текстовий об’єкт у форматі XML, що завантажується через браузер за допомогою downloadFile.

3.3 Розбір компоненту View

Компонент “Представлення” відповідає за візуальне відображення інформації, а також за UI, з яким взаємодіє користувач. Згідно з принципом

єдиної відповідальності, компоненти інтерфейсу розбиті на файл представлення, що описує створення об’єкту в DOM-середовищі, контроллер, якому делегується обробка подій.

В цій частині буде розібрано кожен елемент view, як він виглядає у інтерфейсу користувача, і в чому, з точки зору користувача, полягає його функція. Всі модулі, присутні в компоненті представлення, зображено на Рис. 9.

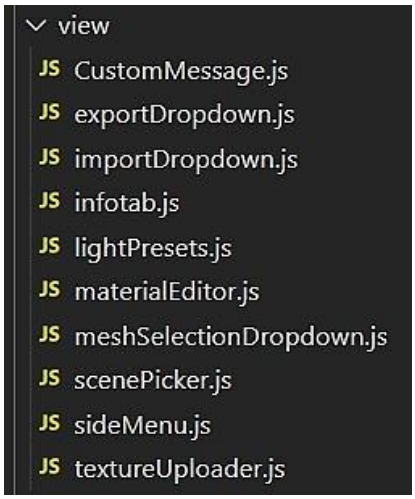


Рисунок 9 — Модулі компоненту View.

Для створення візуального стилю застосунку було обрано css. Такий підхід мінімізує залежність від зовнішніх бібліотек. Створення елементів інтерфейсу проходить через функції javascript.

Модуль **CustomMessage.js** описує текстове повідомлення, що з’являється посеред користувацького екрану, і відображає коротке повідомлення, перед тим, як зникнути. Цей елемент створюється функцією showCustomMessage(message, duration, color), де message — це текстове повідомлення, duration — тривалість відображення в мілісекундах, а color — колір елементу. Цей елемент немає свого контролеру, натомість його викликають інші контролери для повідомлень про помилку, чи про успішне виконання операції. Елемент з класом message-float динамічно додається

до <body> за допомогою document.createElement і видаляється через setTimeout. Приклад UI елементу зображено на Рис. 10.

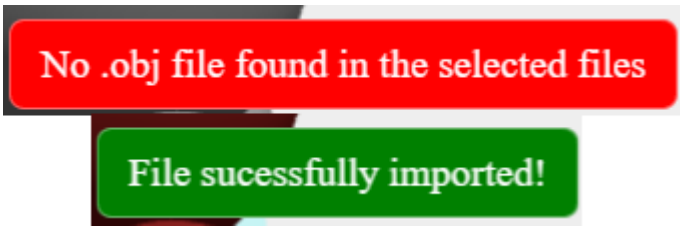


Рисунок 10 — UI елемент CustomMessage

Модуль **ExportDropdown.js** описує інтерактивний UI елемент типу select, який використовується щоб обрати формат експорту для 3D-сцени. Він створюється функцією createExportDropdown(onFormatChange, formatMap), де onFormatChange це callback функція для обробки зміни обраного елементу, описана у відповідному контролері, а formatMap — повна мапа доступних форматів, як описана у лістингу 16. Для кожного запису formatMap, у елементі <select> створюється елемент <option>, зі значенням value — вмістом запису формату. Приклад UI елементу зображено на рис. 11.

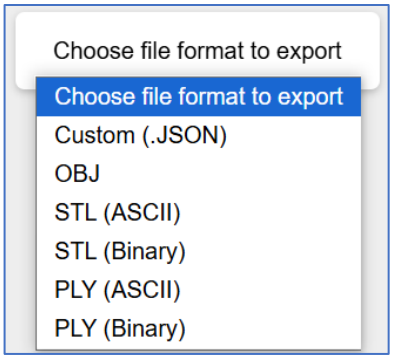


Рисунок 11 — UI елемент ExportDropdown

Модуль importDropdown.js описує інтерактивний UI елемент типу select, який дозволяє обрати формат для імпорту. Крім випадаючого меню, елемент також створює кнопку “завантажити”, що може використовуватись користувачем для завантаження іншого файлу обраного формату. Цей елемент створюється за допомогою функції createImportDropdown(onFormatChange, onUploadClick, onFilesSelected,formatMap), аналогічно до createExportDropdown, але додається callback функція onUploadClick, яка обробляє натиснення на елемент кнопки. Приклад елементу надано на Рис. 12.

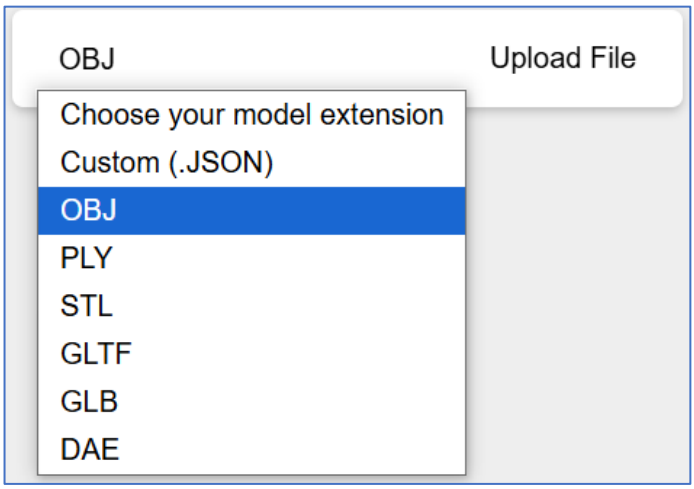


Рисунок 12 — UI елемент ImportDropdown

Модуль **infotab.js** описує плаваюче вікно, що існує для донесення інформації для користувача. Infotab створюється функцією createInfotab(headerTitle, text, onClose), де headerTitle — це текстова назва заголовку, text — текстове наповнення елементу, а onClose — callback функція, яка викликається при натисканні на кнопку закриття. Сам елемент створено з елементу div, в якому, використовуючи flex, в одному рядку відображається div з назвою і кнопка закриття вікна, а на наступній — текстове наповнення елементу. Приклад елементу надано на Рис. 13.

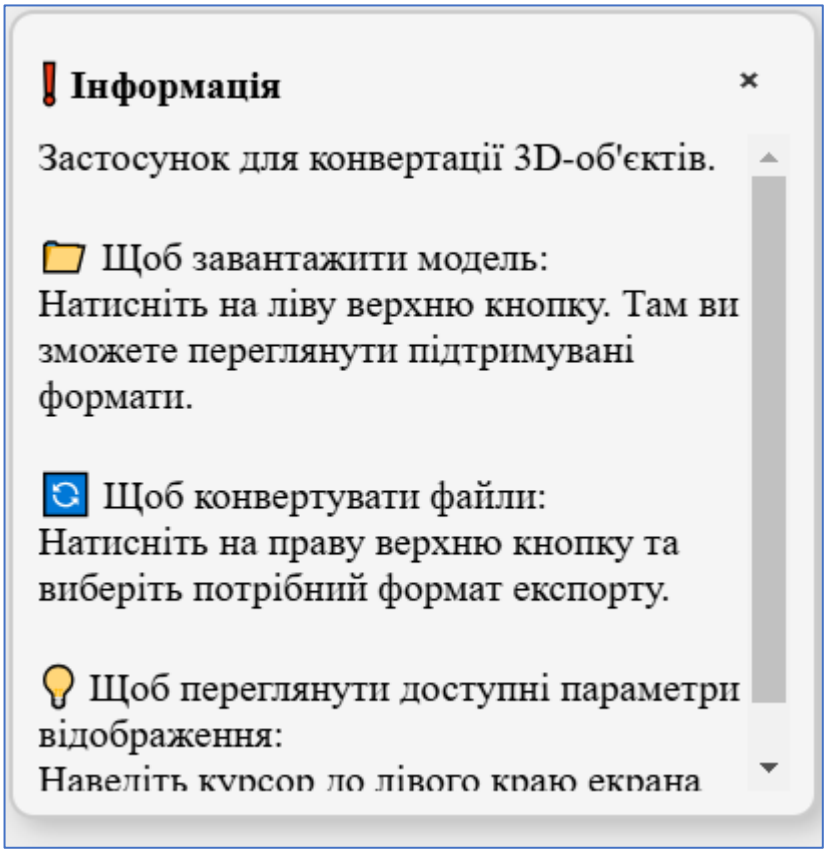
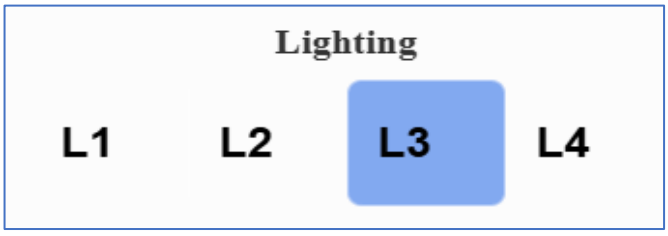


Рисунок 13 — UI елемент Infotab

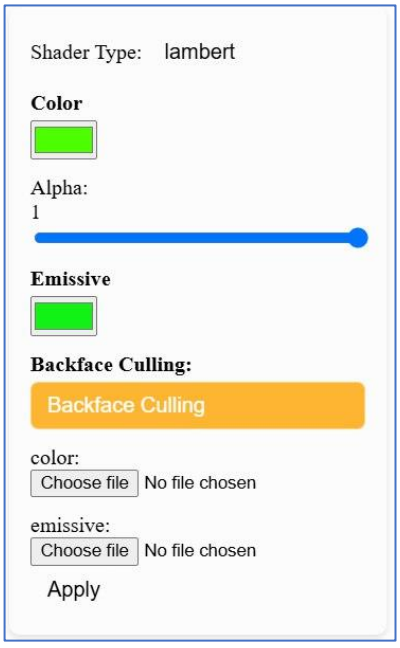
Модуль **lightPresets.js** описує кнопки переключення між режимами освітлення. Він створюється за допомогою функції createLightPresets(onPresetClick, lightPresets) : container. Де onPresetClick це



callback функція при натисканні на кнопку, а lightPresets — список режимів освітлення, описаний у контролері visualisation.js, який буде детально описану у частині 3.4. Ця функція повертає елемент з інтерактивними кнопками <button>. Приклад елементу надано на рис. 14.

Рисунок 14 —UI елемент LightPresets

Модуль materialEditor.js описує динамічну і інтерактивну форму для редагування матеріалів. Він створюється функцією createMaterialEditor(materials, shaderFields, onShaderChange, onColorChange, onSliderChange, onToggleChange, onTextureChange, onApply) : container, де параметри OnShaderChange, onColorChange, onSliderChange, onToggleChange, onTextureChange, onApply — це все callback функцій які дозволяють контролеру керувати елементами форми, а materials — це поточні матеріали. Також цей модуль описує додаткові елементи CreateShaderDropdown, createColorInput, createSlider, createToggle, createTextureInput, createApplyButton, які створюють випадаюче меню для поля ShaderType, елемент вибору кольору для полів color, ambient, emissive, specular, елемент слайдеру для полів alpha, roughness, metallic, елемент перемикання для поля doubleSided, елементи завантаження текстур для полів colorTexture, eissiveTexture, ambientTexture, specularTexture, metallicTexture,



roughnessTexture, normalTexture. Динамічність елементу полягає в його можливості перебудовувати відображені поля в залежності від обраного типу шейдеру. Приклад елементу надано на Рис. 15.

Рисунок 15 —UI елемент materialEditor

Модуль **MeshSelectionDropdown.js** описує випадаюче меню, яке дозволяє користувачу вибрати поточний Mesh елемент для відображення і експорту. Він створюється за допомогою функції createMeshSelectionDropdown(onMeshChange), що отримує callback метод onMeshChange, який виконується при вибору елементу з меню. Приклад елементу наведено на Рис. 16

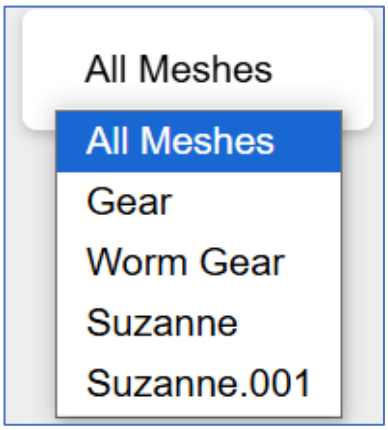
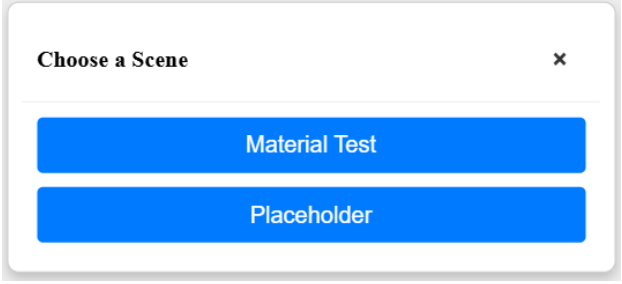


Рисунок 16 — UI елемент sceneSelector

Модуль **scenePicker.js** описує меню, що з’являється при завантаженні сторінки яке пропонує завантажити готову сцену для тестування застосунку. Елемент створюється функцією createScenePicker(title, scenes, onButtonClick,



onClose), де onButtonClick і onClose це callback функції контролерів, title це текстова назва вікна, а scenes — перелік сцін з їх назвою, шляхом до файла в public. Елемент видаляється при завантаженні сцени через імпортер, чи при натисканні на кнопку. Приклад наведено на Рис. 17.

Рисунок 17 — UI елемент ScenePicker.

Модуль sideMenu.js описує інтерактивне меню, що з'являється при наближенні курсору миші до краю екрану. Воно повинно містити елементи materialEditor та lightPresets. Створюється воно функцією createSideMenu(textureSection, lightContainer, materialContent), де textureSection — це елемент TextureUploader, lightSection — lightPreset, а materialContent — materialEditor. Відображення елементу реалізовано подією "mouseMove". Приклад наведено на рис. 18

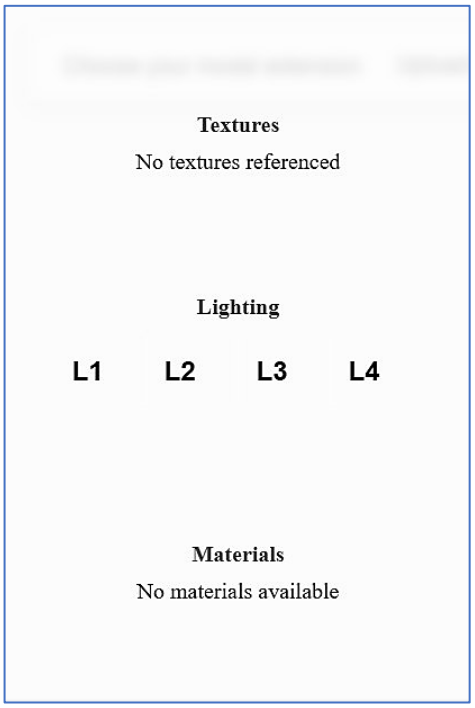
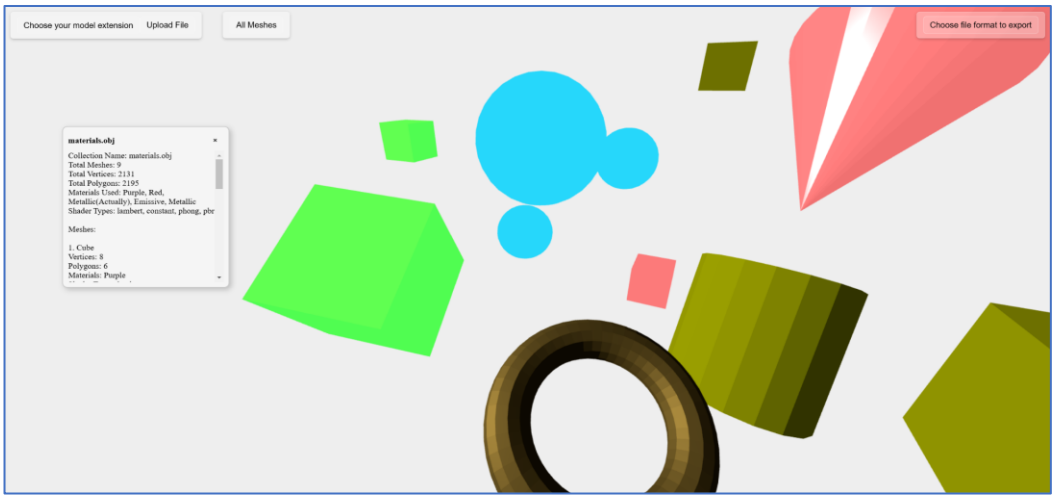


Рисунок 18 — UI елемент sideMenu

Інтерфейс проектувався для користувачів з обмеженим досвідом роботи з 3D-графікою. Дотримуючись закону Фітса, всі інтерактивні елементи і кнопки були максимально збільшені і чітко виділені на фоні, доступ до будь-

якої функції потребує не більше двох дії, а головний елемент системи —3D сцена — займає центральну позицію. Інтерфейс виконано з урахуванням



естетики: елементи мають закруглені краї, напівпрозорість і тіні для кращого візуального відокремлення. Елементи інтерфейсу, непотрібні в поточний момент, прибираються з екраную. Повний вигляд інтерфейсу надано на рис. 19

Рисунок 19 — Демонстрація повного інтерфейсу застосунку

3.4 Розбір компоненту Controller

Компонент “Контроллер” відповідає за основний функціонал застосунку. Він забезпечує зв’язок між представленням і моделлю,



дозволяючи відображення та редагування інформації користувачем. Модулі компоненту Controller зображено на Рис. 20.

Рисунок 20 — модулі компонента Controller.

Точкою входу в програму є контролер **appInit.js**. Це — центральна точка застосунку, що викликається при завантаженні сторінки, і ініціалізує всі інші контролери. Цей модуль містить одну функцію, initApp(), яка ініціалізує всі інші елементи інтерфейсу.

Контролер **visualisation.js** відповідає за рендеринг 3D-сцін в застосунку. Функція його ініціалізації, initVisualisation(), створює і налаштовує нову Three.js сцену, поле рендерингу, і ініціалізує orbitControls для інтерактивного огляду сцени з різних ракурсів. Цей елемент пов’язаний із моделлю, а саме з модулем state.js, через патерн “спостерігач”, де visualisation.js є спостерігачем подій. При виникненні подій, тобто при зміні сцени, контролер видаляє стару сцену, і додає до нової всі елементи масиву, використовуючи функцію handleDataChange(). Крім цього, контролер містить пресети освітлення, які використовує контролер lightPresetController.js. Вони зберігаються як анонімні функції, що додають до сцени певні елементи освітлення. Приклад світлового пресету надано у лістингу 20.

Лістинг 20 Приклад світлового пресету

```
// 0. Basic light (ambient + directional)
() => {
    clearLights(scene);
    const directionalLight = new
THREE.DirectionalLight(0xffffff, 1);
    directionalLight.position.set(1, 1, 2);
    scene.add(directionalLight);
    const ambientLight = new THREE.AmbientLight(0x404040,
2);
    scene.add(ambientLight);
},
```

Контролер **exportDropdownController.js** забезпечує зв’язок з UI елементом exportDropdown і моделлю (formatMap.js). Він передає конструктору exportDropdown дані з formatMap.js, та реалізує callback функцію handleFormatChange(format), яка викликає відповідний формату експортер з модулю exporters.js.

Контролер **importDropdownController.js** забезпечує зв’язок з UI елементом importDropdown і моделлю (formatMap.js). Він передає конструктору importDropdown дані formatMap.js, і реалізує callback функції HandleFormatChange(format, extensions), handleUploadClick(), handleFileSelect(). Він організує завантаження довільної кількості файлів, і передачу їх до коректної функції-імпортеру, з подальшим завантаженням їх у state.js. Крім цього, importDropdownController також обробляє текстурні дані через допоміжну функцію parseAndLoadTextures(textDictionary, files). Ця функція отримує завантажені файли, і екземпляр textDictionary. Якщо в ньому є порожні записи(ключ, з значенням [null, null], то він шукає в переданих файлах ці текстури.

Контролер **infoTabController.js** забезпечує зв’язок з представленням infoTab і моделлю (state.js). Він виконує ініціалізацію цього об’єкту, і забезпечує, щоб плаваюче вікно з інформацією було в одному екземплярі. Крім цього, цей контролер є спостерігачем state.js, і при зміні сцени, воно заповнює infoTab метаданими, отриманими з методу getFormattedMetadata класу MeshCollection.

Контролер **LightPresetsController.js** забезпечує зв’язок між модулем lightPresets.js і моделлю (state.js), керуючи вибором режимів освітлення 3D-сцени. Метод init() ініціалізує контейнер кнопок через функцію createLightPresets, передаючи callback handlePresetClick(index) і список пресетів із visualization.js. Метод handlePresetClick(index) оновлює активну кнопку (додаючи клас activeLight) та викликає setCurrentLightPreset(index) для оновлення моделі.

Контролер **MeshSelectionDropdownController.js** забезпечує зв’язок між модулем meshSelectionDropdown.js і моделлю (state.js), керуючи випадаючим меню для вибору мешів 3D-сцени. Метод init() створює меню через createMeshSelectionDropdown, передаючи callback handleMeshChange(meshIndex), і налаштовує обробник події dataChanged. Метод handleMeshChange(meshIndex) оновлює модель, викликаючи setExportMeshIndex(meshIndex) для вибору активного меша. Метод handleDataChanged(event) оновлює опції меню на основі списку мешів (meshes) із моделі та керує видимістю меню залежно від наявності мешів.

Контролер **ScenePickerController.js** забезпечує зв’язок між модулем scenePicker.js і моделлю (state.js), керуючи меню вибору тестових сцен. Метод init() створює меню через createScenePicker, передаючи заголовок, список сцен, callback handleSceneSelect(index) для вибору сцени та handleClose() для закриття. Метод handleSceneSelect(index) завантажує JSON-файл сцени через fetch, імпортує його за допомогою importJSON і оновлює модель через setCurrentInternalData. У разі помилки відображається повідомлення через showCustomMessage. Метод handleClose() видаляє меню з DOM після вибору сцени або події dataChanged від моделі.

Контролер **MaterialEditorController.js** забезпечує зв’язок між модулем materialEditor.js і моделлю (state.js), керуючи інтерактивною формою для редагування матеріалів 3D-сцени. Метод init(materials) ініціалізує форму, передаючи копію матеріалів, конфігурацію shaderFields і callback-функції для обробки змін шейдера, кольору, слайдерів, перемикачів, текстур і підтвердження змін. Методи handleShaderChange, handleColorChange, handleSliderChange, handleToggleChange, handleTextureChange і handleApply оновлюють локальний стан матеріалів, завантажують текстури через THREE.TextureLoader, оновлюють модель через setCurrentMaterials і перебудовують форму за потреби, забезпечуючи динамічне відображення параметрів залежно від типу шейдера.

Контролер **TextureUploaderController.js** забезпечує зв’язок між модулем textureUploader.js і моделлю (state.js), керуючи інтерфейсом завантаження текстур для 3D-сцени. Метод init() створює елемент TextureUploader через createTextureUploader, передаючи callback handleFileSelected(filename, file) для обробки вибраних файлів, і викликає updateUI для ініціалізації інтерфейсу. Методи handleFileSelected і updateUI завантажують текстури через THREE.TextureLoader, оновлюють словник текстур у моделі за допомогою setCurrentTextures і відображають статус завантаження, показуючи повідомлення про помилки через showCustomMessage у разі невдачі.

Контролер **SideMenuController** забезпечує зв’язок між модулем sideMenu.js і моделлю (state.js), об’єднуючи функціонал контролерів TextureUploaderController, LightPresetsController і materialEditorController для керування бічним меню 3D-сцени. Метод init() створює меню через createSideMenu, ініціалізуючи секції для завантаження текстур, вибору освітлення та редагування матеріалів, і налаштовує обробники подій dataChanged і textureChanged. Методи rebuildMaterialEditor і rebuildTextureUploader динамічно оновлюють секції матеріалів і текстур у відповідь на зміни в моделі, відображаючи актуальні дані або повідомлення про їх відсутність.

3.5 Тестування застосунку

Фізичні характеристики поточної версії системи:

Об’єм коду по мовах програмування:

1. Javascript — 28 модулів, ~185 кілобайт, ~4000 рядків коду, ~600 коментарів.
2. CSS — 1 модуль, ~7 кілобайт, ~300 рядків коду.

Об’єм коду по компонентах:

1. Model — 5 модулів, ~118 кілобайт, ~2700 рядків коду, ~200 коментарів.

2. View — 10 модулів, ~25 кілобайт, ~450 рядків коду, ~150 коментарів.
3. Controller — 12 модулів, ~42 кілобайт, ~800 рядків коду, ~250 коментарів.

Загальний розмір зібраного застосунку: 1330 кілобайт (з public), 260 кілобайт (без public).

Середній час завантаження: ~10 мс (на сучасному ПК)

Застосунок було протестовано вручну, на чотирьох типах моделей: легкій (300 трикутників), середній (5 000 трикутників), складній (30 000 трикутників), і надскладний (1 000 000 трикутників).

- Легке навантаження: CPU: 10%; GPU: 15%; RAM: 80 МБ; FPS:120;
- Середнє навантаження: CPU:12% GPU: 17% RAM: 95 МБ FPS: 120
- Високе навантаження: CPU: 14% GPU: 18% RAM: 95 МБ FPS: 120
- Найвище навантаження: CPU: 13% GPU: 17% RAM:517 МБ FPS: 120

Під час тестування функцій імпорту\експорту вимірювались швидкість конвертації та розмір результуючого внутрішнього формату, які порівнювалися з розміром вихідного формату. Результати тестування наведено у таблицях 12–16.

Таблиця 12 – Продуктивність алгоритмів на моделі низької складності

Формат	**Вага (КБ) до конвертації**	**Час конвертації (мс)**	**Вага (Кб) після конвертації**	**Час експорту (мс)**
STL	14	11	76	5
PLY	5	22	32	4
OBJ	13	21	19	3
DAE	9	22	5	3
GLTF	4	21	5	5
JSON	72	20	19	2

Таблиця 13 – Продуктивність алгоритмів на моделі середньої складності

Формат	**Вага (КБ) до конвертації**	**Час конвертації (мс)**	**Вага (Кб) після конвертації**	**Час експорту (мс)**
STL	262	51	1234	25
PLY	101	25	626	31
OBJ	235	42	350	16
DAE	352	47	511	15
GLTF	76	23	502	13
JSON	1184	24	350	12

Таблиця 14 – Продуктивність алгоритмів на моделі високої складності

Формат	**Вага (КБ) до конвертації**	**Час конвертації (мс)**	**Вага (Кб) після конвертації**	**Час експорту (мс)**
STL	1573	290	9677	117
PLY	606	158	4380	140
OBJ	1533	173	2165	60
DAE	2109	185	3068	57
GLTF	380	92	3015	53
JSON	7164	108	2165	61

Таблиця 15 – Продуктивність алгоритмів на моделі середньої складності з матеріалами

Формат	**Вага (КБ) до конвертації**	**Час конвертації (мс)**	**Вага (Кб) після конвертації**	**Час експорту (мс)**
OBJ	236	86	350	16
DAE	353	101	511	12
GLTF	76	46	502	12
JSON	1184	74	350	12

Таблиця 16 – Продуктивність алгоритмів на моделі найвищої складності

Формат	**Вага (КБ) до конвертації**	**Час конвертації (мс)**	**Вага (Кб) після конвертації**	**Час експорту (мс)**
OBJ	66 287	6 009	100 020	2 618
DAE	154 000	21 017	220 134	3 060
GLTF	64 000	14 849	43 513	3 761
JSON	100 020	8 931	100 020	3 326
PLY	18 000	7 948	147 662	2 885
STL	48 000	14 277	316 203	5 149

Також вручну було проведено функціональне тестування для всіх реалізованих імпортерів та експортерів. Для кожного формату принаймні одна модель була успішно імпортована до застосунку, відображена в середовищі візуалізації, а також експортована та відкривалась без помилок у сторонніх 3D-застосунках (Blender 3D, MeshLab). Тестування проводилось за

методикою чорної скриньки, тобто розглядались лише результати, без аналізу швидкодії і продуктивності.

Застосунок коректно працює на браузерах: Chrome, Firefox, Edge. При завантаженні поламаного файлу, застосунок, за допомогою UI елементу CustomMessage повідомляє користувачу про проблему (див рис. 21)



Рисунок 21 — помилка імпорту пошкодженого файлу

Розробка застосунку проводилась в середі Visual Studio Code. Для збирання метрик використовувались Chrome Developer Tools. Для оцінки розміру коду використовувався плагін VS Code Counter.
Для розробки застосунку використовувались сторонні бібліотеки: node.js, three.js, JSzip.

Проведене тестування підтвердило стабільну роботу функцій імпорту та експорту 3D-моделей у всіх досліджених форматах (STL, PLY, OBJ, DAE, GLB, JSON). Результати показали, що час конвертації та розмір результуючого файлу зростають пропорційно до складності моделі, при цьому внутрішній формат-посередник займає більше місця ніж формати-аналоги. Для моделей із текстурами та матеріалами спостерігається незначне збільшення часу обробки, що свідчить про коректну інтеграцію додаткових даних. Застосунок демонструє високу продуктивність на клієнтській стороні, не потребуючи серверних ресурсів, що відповідає поставленим цілям розробки.

3.6 Висновки з розділу 3

1. Для розробки застосунку була застосована модифікована трьох-компонентна структура MVC, Компонентами якої є модель, представлення, та контролери.
2. Були визначені та досягнуті функціональні та нефункціональні вимоги до застосунку.
3. Був розроблений клієнтський веб-застосунок для імпорту, візуалізації, та експорту 3D-об’єктів через формат посередник.
4. Було розроблено формат-посередник для спрощення процесу конвертації
5. Детально описано реалізацію всіх компонентів, модулів, і функцій, а також тестування системи.

ВИСНОВКИ

1. В процесі роботи було проаналізовано найпопулярніші формати для збереження 3D-інформації, досліджено їх структуру, сферу використання, та типи інформації, яку вони зберігають
2. Розроблено власний формат-посередник, призначений для максимального збереження даних про геометрію та матеріали 3D-об’єктів з метою мінімізації втрат під час конвертації.
3. Створено набір алгоритмів для перетворення популярних 3D-форматів у формат-посередник, а також алгоритмів для зворотної конвертації.
4. Реалізовано веб-застосунок із фронтенд-архітектурою, що забезпечує обробку даних виключно на стороні клієнта.
5. Розроблено інтуїтивно зрозумілий інтерфейс, орієнтований на користувачів без спеціальних знань у галузі 3D-графіки.
6. Результати тестування підтвердили коректність роботи функцій конвертації та достатню швидкодію застосунку без залучення серверних ресурсів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Деркач А. С. Історія розвитку та сучасний стан 3D моделювання / А. С. Деркач // Науковий вісник Південноукраїнського національного педагогічного університету імені К. Д. Ушинського : наук. журнал. – Одеса : ПНПУ ім. К. Д. Ушинського, 2023. – № 4(145). С. 713.

2. Sutherland I. E. Sketchpad. the May 21-23, 1963, spring joint computer conference, Detroit, Michigan, 21–23 May 1963. New York, New York, USA, 1963. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-574.pdf (date of access: 24.03.2025).

3. Johnson T. E. Sketchpad III. the May 21-23, 1963, spring joint computer conference, Detroit, Michigan, 21–23 May 1963. New York, New York, USA, 1963. URL: https://doi.org/10.1145/1461551.1461592 (date of access: 06.05.2025).

4. Difference between CPU and GPU. geeksforgeeks. URL: https://www.geeksforgeeks.org/difference-between-cpu-and-gpu/ (date of access: 25.04.2025).

5. History of OpenGL. Khronos. URL: https://www.khronos.org/opengl/wiki/History_of_OpenGL (date of access: 25.04.2025).

6. 3D-графіка : навчальний посібник / В. П. Гаврилов. – Харків : ХНЕУ ім. С. Кузнеця, 2018. – 127 с. ISBN 978-966-676-735-9

7. Розуміння UV-розгортання. adobe. URL: https://www.adobe.com/ua/products/substance3d/discover/uv-mapping.html (дата звернення: 25.04.2025).

8. Patricio Gonzalez Vivo, Jen Lowe. The book of shaders. URL: https://thebookofshaders.com/. (дата звернення: 25.04.2025).

9. Wavefront OBJ file format summary. FileFormat.Info. URL: https://www.fileformat.info/format/wavefrontobj/egff.htm (date of access: 25.04.2025).

10. Alias/Waverfront MTL file format. Clemson University. URL: https://people.computing.clemson.edu/~dhouse/courses/405/docs/brief-mtl-file-format.html (date of access: 25.04.2025).
11. STL format. Paul Bourke. URL: https://paulbourke.net/dataformats/stl/. (дата звернення: 25.04.2025).
12. PLY - polygon file format. Paul Bourke. URL: https://paulbourke.net/dataformats/ply/. (дата звернення: 25.04.2025).
13. GlTF™ 2.0 specification. khronos.org. URL: https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html (date of access: 25.04.2025).
14. COLLADA – Digital Asset Schema Release 1.5.0. khronos.org. URL: https://www.khronos.org/files/collada_spec_1_5.pdf (date of access: 25.04.2025).
15. WebGL Specification. *registry.khronos.org*. URL: https://registry.khronos.org/webgl/specs/latest/1.0/?utm_source=chatgpt.com (дата звернення: 25.04.2025).
16. three.js docs. threejs. https://threejs.org/docs/index.html (дата звернення: 25.04.2025).
17. Difference Between MVC, MVP and MVVM Architecture Pattern in Android. geeksforgeeks. URL: https://www.geeksforgeeks.org/difference-between-mvc-mvp-and-mvvm-architecture-pattern-in-android/ (date of access: 25.04.2025).
18. JSZip. github. URL: https://stuk.github.io/jszip/ (date of access: 25.04.2025).
19. Скрипак С.О., бакалавр, Полякова Н.П., доцент — науковий керівник. РОЗРОБКА ВЕБ-ЗАСТОСУНКУ ДЛЯ ІМПОРТУ, ВІЗУАЛІЗАЦІЇ ТА ЕКСПОРТУ 3D-ОБ'ЄКТІВ РІЗНИХ ФОРМАТІВ НА БАЗІ WEBGL. Молода наука- 2025 : зб. наук. праць студентів, аспірантів і молодих вчених. Запоріжжя : ЗНУ, 2025. Т. 5. С. 413–414.

Декларація
академічної доброчесності
здобувача ступеня вищої освіти ЗНУ

Я, Скрипак Степан Олегович, студент 4 курсу, форми навчання денної, Інженерного навчально-наукового інституту ім. Ю.М. Потебні ЗНУ, спеціальність 121 Інженерія програмного забезпечення освітньої програми “Програмне забезпечення систем”, адреса електронної пошти skripstepana@gmail.com, — підтверджую, що написана мною кваліфікаційна робота на тему «Розробка веб-застосунку для імпорту, візуалізації та експорту 3D-об'єктів різних форматів на базі WebGL» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

-згоден на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 12.06.2025 _____ Скрипак Степан Олегович
(студент)

Дата 12.06.2025 _____ Безверхий Анатолій Ігорович
(керівник)