МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ЕЛЕКТРОНІКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

Кваліфікаційна робота бакалавра

ІНФОРМАЦІЙНЕ ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ СТВОРЕННЯ ОТВОРІВ У 3D МОДЕЛЯХ

Здобувач освіти гр. ІН – 82 Вероніка КОТЕЛЕВСЬКА

Науковий керівник,
кандидат фізико-математичних наук,
асистент кафедри комп'ютерних наук Борис КУЗІКОВ

Завідувач кафедри
доктор технічних наук, професор Анатолій ДОВБИШ

СУМИ 2022

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ЕЛЕКТРОНІКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

Затверджую_____
Зав. кафедрою Довбиш А.С.
"_____" _____2022 р.

ЗАВДАННЯ

до кваліфікаційної роботи

здобувача вищої освіти четвертого курсу, групи ІН-82 спеціальності «122 – Комп'ютерні науки» денної форми навчання Котелевської Вероніка Андріївни.

Тема: «ІНФОРМАЦІЙНЕ ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ СТВОРЕННЯ ОТВОРІВ У 3D МОДЕЛЯХ»

Затверджена наказом по СумДУ
№_____ від _____2022 р.

Зміст пояснювальної записки: 1) інформаційний огляд способів отримання моделей з отворами; 2) постановка завдання для розробки; 3) вибір інструментів для розробки графічного додатку; 4) програмна реалізація.

Дата видачі завдання «_____»_____2022 р.
Керівник роботи_____Б. КУЗІКОВ
Завдання прийняла до виконання_____В. КОТЕЛЕВСЬКА

РЕФЕРАТ

Записка: 44 стор., 18 рис., 1 формула, 19 джерел.

Об'єкт дослідження — програмне та інформаціне забезпечення створення отворів в 3D моделях.

Мета роботи — розробка інформаційного та програмного забезпечення для створення отворів у 3D моделях які мають структуру трикутної сітки завдяки видаленню трикутників з поверхні моделі або розрізання моделі площиною на дві половини.

Інструменти розробки — мова програмування C++, графічний API OpenGL специфікації 2.0 та суміжні з OpenGL бібліотеки GLFW, GLAD, GLM а також C++ парсер XML – бібліотека TinyXML.метод функціонально-статистичних випробувань.

Результати — розроблено інформаційного та програмного забезпечення для створення отворів у 3D моделях, яке дозволяє завантажити файли формату STL або COLLADA, видалити обрані трикутники, розрізати площиною та зберегти зміни. Інструмент володіє простим та зручним інтерфейсом для комфортного та швидкого створення отворів, сконцентрованого на інтерактивній взаємодії з користувачем.

ГРАФІЧНИЙ РЕДАКТОР, 3D ГРАФІКА, 3D МОДЕЛЬ,
ТРИКУТНА СІТКА, ОТВІР В ПОЛІГОНАЛЬНІЙ СІТЦІ,
МАНІПУЛЯЦІЇ З ПОЛІГОНАЛЬНОЮ СІТКОЮ,
РОЗРІЗАННЯ 3D МОДЕЛІ ПЛОЩИНОЮ.

ЗМІСТ

ВСТУП

Програми для 3D моделювання стають все більш популярними. Ці системи не тільки у багато разів спрощують процес конструювання та дизайну, а також мають безліч додаткових можливостей, недоступних іншим шляхом. Зі стандартного списку користувачів ситем 3D моделювання – архітектори, інженери, дизайнери. Але насправді коло користувачів набагато ширше, а отже, і потреби різні: самих тільки дизайнерів багато видів (дизайн інтер'єру, одягу, UI/UX, game- дизайн тощо). Очевидно, що однією програмою не покрити потреби кожної галузі.

Зростаючу потребу в 3D-редакторах зауважують і IT-компанії. Вони вдосконалюють вже існуючі рішення та створюють нові, розширюючи можливості. Але прогрес відбувається не самостійно, а зусиллями вчених-дослідників та розробників в галузі комп'ютерної графіки та математики. Вони займаються створенням нових алгоритмів, вдосконаленнями підходів у вирішенні конкретних задач, покращенням перфомансу та якості. Для візуалізацій складних процесів, результатів досліджень, симуляції фізичних явищ потрібні програми для 3D моделювання.

Часто необхідно аналізувати вже готові моделі, та потім використовувати отримані дані для того, щоб щось змінити чи створити нове. Для таких завдань необхідно мати безліч готових моделей для тестування та дослідження. Необхідні характеристики моделі (тип, властивості, особливості, формат даних) залежать від конкретних ситуацій. Це можуть бути простий кубик у форматі STL або дрібна деталь двигуна на 300 фасетів у форматі COLLADA або дуже деталізована будівля у форматі RVT.

Іноді є необхідність в 3D моделях, які складаються з трикутної сітки та мають отвори, наприклад, для таких алгоритмів як знаходження отворів в

моделях, або тріангуляція отворів [1], або відновлення відсутньої форми апроксимуючи граничні частини [2] тощо.

Готові моделі можна спробувати знайти в інтернеті, або створити самостійно за допомогою програм 3D моделювання. Можна створити потрібну модель «з нуля», і для цього існує багато інструментів, але це вимагає часу та зусиль, певних навичок роботи з цим інструментом.

Кращим підходом є модифікація вже готових моделей. Все що потрібно – завантажити модель, створити в ній отвори, і зберегти зміни.

Тож, дана робота присвячена створенню простого, не ресурсомісткого 3D редактора, який би вмів завантажувати 3D модель, створювати отвори і зберігати змінений файл.

Виходячи з мети, можна поставити такі задачі:

- Проаналізувати формати зберігання 3D моделей. Визначити які формати є найбільш розповсюдженими.
- Проаналізувати наявні інтернет-ресурси для пошуку та завантаження 3D моделей, наявність моделей з отворами.
- Проаналізувати наявні 3D редактори, їх можливості, переваги та недоліки, зручність у використанні.
- Спроєктувати та реалізувати програмне рішення для створення отворів в 3D моделях.
- Реалізувати завантаження найпопулярніших форматів файлів з 3D моделями, рендеринг моделі в 3D сцені, маніпуляції з камерою, створення отворів у вказаних місцях та збереження файлу зі змінами.

1 ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1 Формати файлів з 3D моделями

Існують різні формати для зберігання інформації про 3D моделі. Найпопулярніші з них: STL, OBJ, FBX, COLLADA і т.п. Вони широко використовуються в 3D друкуванні, відеоіграх, кіно, архітектурі, медицині, конструюванні та в процесі навчання. При цьому в кожній із перерахованих сфер є свої найбільш популярні формати, які сформувалися з різних історичних та практичних причин [4].

Основне призначення 3D файлу - зберігати інформацію про 3D моделі як у звичайному текстовому чи бінарному вигляді. Вони кодують інформацію про геометрію, зовнішній вигляд, сцену та анімацію 3D моделі. Геометрія моделі визначає її форму. Зовнішній вигляд включає кольори, текстури, матеріал і т.п. Під сценою мається на увазі розташування джерел освітлення, камер та периферійних об'єктів. А анімація характеризує переміщення 3D моделі. Однак не всі формати 3D-файлів зберігають всю цю інформацію, наприклад, COLLADA зберігає все, а STL формат зберігає лише інформацію про геометрію та ігнорує інші данні [5].

Насправді існує так само багато різних методів зберігання 3D моделей, як багато їх існує для зберігання фото і відео. Але існують універсальні формати, які, хоч і з деякими обмеженнями, можна відкривати майже в будь-якій програмі. Два найпопулярніші універсальні формати - це STL (розширення .stl) і COLLADA (розширення .dae). Вони дуже широко використовуються для обміну даними про 3D моделі між CAD програмами [5]. Отже, будемо вважати можливість роботи з STL та COLLADA – необхідною умовою для бажаного 3D-редактору.

1.2 Представлення 3D моделі у вигляді полігональної сітки

STL і COLLADA формати зберігають інформацію про 3D об'єкт як перелік трикутних граней, що описують його поверхню. Трикутні грані сполучаються так, що створюють цілісну поверхню, кожному ребру кожної грані відповідає інше ребро. Ребра з’єднують вершини, утворюючи сітку з трикутними отворами (див. рис. 1.1). Така структура називається полігональною (трикутною) сіткою (triangle mesh) [6].

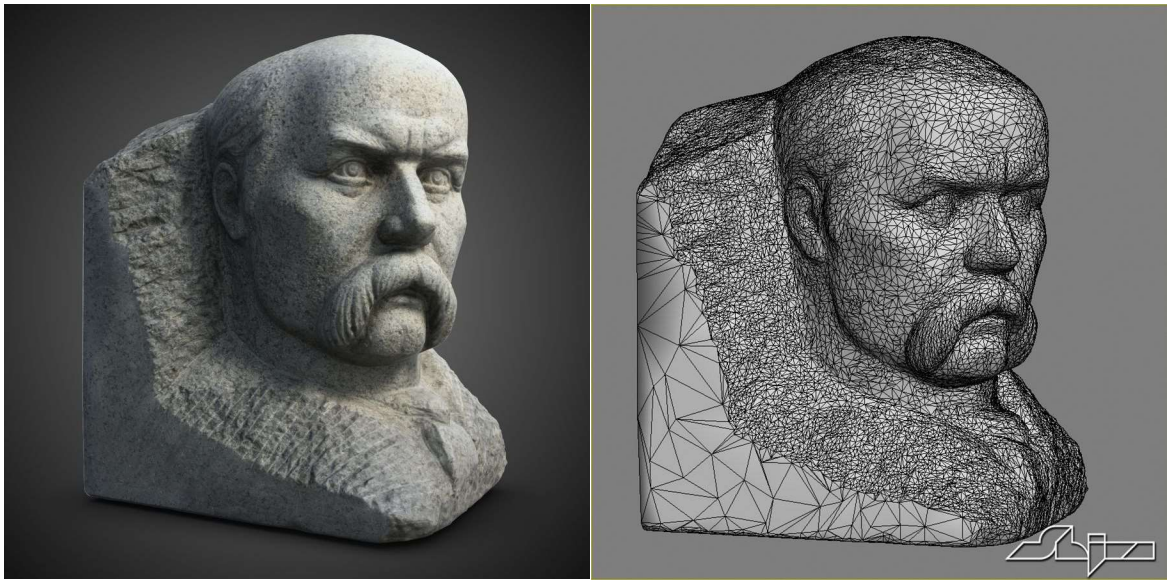

Рисунок 1.1 – 3D модель Т.Г.Шевченка у вигляді трикутної сітки [7].

Багато застосунків комп'ютерної графіки використовують саме таку структуру для представлення об’єктів. На те є ряд причин [8]:

1) Немає проблем однозначно інтерпретувати форму поверхні.
2) Ігнорування об’єму і характеристик внутрішнього простору.
3) Можливість представлення будь-яких форм.
4) Різний степінь деталізації.
5) Перелік трикутників використовується для рендерингу.
6) Можливість застосування візуальних ефектів, матеріалів.

Щоб модель мала фізичний зміст, потрібно чітке розділення тривимірного простору на внутрішній та зовнішній для моделі. Наприклад, якщо модель складається з 1 трикутнику, то для неї неможливо розділити простір на зовнішній та внутрішній. Так само якщо взяти фізично реальну модель, як сферу, але видалити з неї 1 трикутник, то розділення простору знову неоднозначне (див. рис. 1.2).

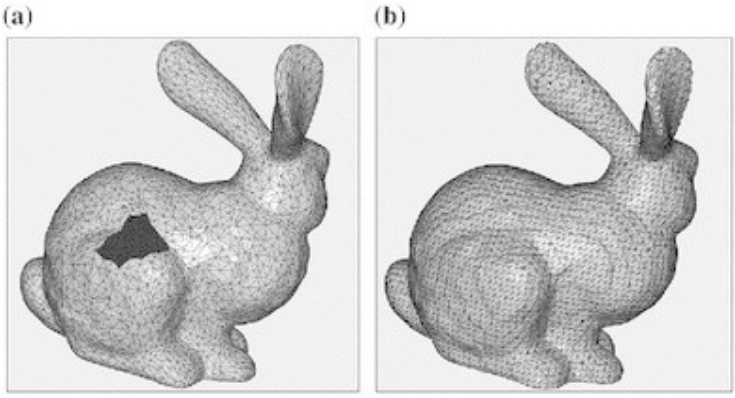

Рисунок 1.2 – 3D сітка з отворами (а) і з цілісною поверхнею (б) [9].

Такі моделі є не валідними для використання в комп'ютерній графіці або в технологічних процесах спрямованих в кінцевому етапі на друк цієї моделі на 3D принтері. Очевидно що моделі з отворами не є популярними, але саме такі моделі іноді бувають необхідні для інших задач, наприклад для тестування алгоритмів триангуляції (див рис. 1.3) тощо.

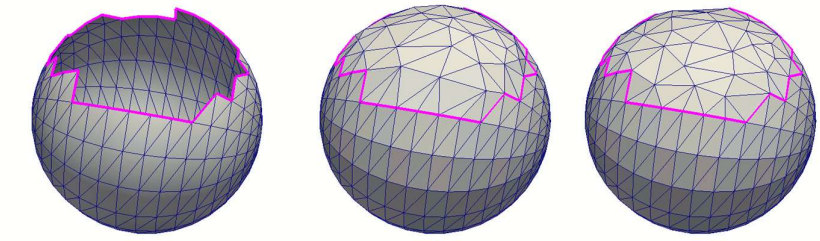

Рисунок 1.3 – Використання моделі з отвором для триангуляції.

Отже, в даній роботі під терміном «модель з отвором» мається на увазі така 3D модель, яка має структуру трикутної сітки (triangle mesh), а її поверхня не є цілісною, тобто з відсутньою трикутною гранню або гранями.

1.3. Інтернет-ресурси для завантаження 3D моделей

Завантажити безкоштовно 3D моделі зараз стало набагато простіше. З'явилося безліч сайтів та ресурсів з 3D моделями. Найпопулярніші інтернет-ресурси де можна вільно завантажити 3D моделі це [10]:

1) GrabCAD - https://grabcad.com
2) Sketchfab - https://sketchfab.com
3) TurboSquid - https://www.turbosquid.com
4) Free3D - https://free3d.com
5) 3dsky - https://3dsky.org
6) Open3dModel - https://open3dmodel.com

Спроба знайти полігональні моделі з отворами завершилась невдало: жодного знайденого результату за цілу годину пошуку. Причини наступні:

1) Непопулярність моделей з отворами, тому, що вони не валідні для використання в графіці або для 3D друку.
2) Відсутність потрібних фільтрів для сортування моделей за структурою – сортування переважно за змістом (як предмет інтер'єру або модель людини) та призначенням (як автоконструювання чи будівництво).
3) Відсутність фільтрів за способом представлення геометричної інформації: mesh, solid, voxel, wireframe тощо, проте є фільтри за форматом даних та програмним забезпеченням, в якому модель згенерована, але це вимагає додаткових знань про всі формати та програми.
4) Пошук за ключовими словами holes, sheet bodies, mesh with hole, mesh with slit не дав результатів, бо ці слова трактуються інакше (отвір не як розрив поверхні, а як отвір в гудзику, листкове (плоске) тіло не як модель без об'єму, а як аркуш паперу, який, технічно, має об'єм).

Висновки: Знайти моделі з отворами дуже складно, тому використання програми для 3D моделювання є необхідним для створення моделі з отвором.

1.4 Програмні рішення для проєктування 3D моделей

1.4.1 Системи автоматизованого проєктування (САПР)

Замість «програми для 3D моделювання» частіше використовуються термін САПР або CAD software. CAD - це скорочення від Computer-Aided Design, тобто система автоматизованого проєктування (САПР) – тип програмного забезпечення для проєктування 2D або 3D моделей, що використовується в різних галузях промисловості: дрібних деталей, архітектури, двигунів, конструкцій усіх видів, транспортних засобів, схем, персонажів для фільмів тощо. Він також може використовуватися для симуляції певних процесів або анімації [11].

Різноманітних 3D редакторів існує багато, від простих до складних, для дизайнерів або інженерів, з різними наборами можливостей. Більшість з них націлена на створення моделі «з нуля», але це не ефективно і вимагає певних навичок як дизайнера чи інженера. Кращим рішенням було б обрати з безкоштовних ресурсів модель з цілісною поверхнею, яка має структуру (або може бути конвертована) в полігональну (трикутну) сітку, а потім створити в її поверхні отвори. Цей процес теоретично може займати дуже мало часу, особливо в порівнянні з конструюванням «з нуля». Від редактора не вимагається багато функцій, тільки завантаження моделі, створення отворів і збереження змін, що також може свідчити про простоту використання та лаконічність інтерфейсу, а отже, швидке освоєння інструменту.

Можемо сформулювати вимоги до САПР, яке може бути рекомендованим для використання з метою створення отворів в полігональних моделях. Вимагається можливість:

- ✓ створювати отвори в трикутній сітці в заданому положенні;
- ✓ завантаження та збереження файлів формату STL та COLLADA;
- ✓ використання без спеціальної підготовки.

1.4.2 Огляд існуючих САПР для створення отворів

САПР які можуть працювати з трикутними сітками та створювати отвори це: Blender, 3ds Max, PowerShape та Meshlab. Проаналізуємо ці програмні рішення на відповідність критеріям та продемонструємо створення отворів в них.

1.4.2.1 Blender

Blender — це публічний проєкт, розміщений на blender.org, ліцензований як GNU GPL, що належить його учасникам. З цієї причини Blender є безкоштовним програмним забезпеченням з відкритим вихідним кодом, назавжди [12].

Підтримка формату STL та COLLADA ✓

Використання без спеціальної підготовки ✗

Перший спосіб створення отвору це видалення грані. Порядок дій: в режимі «Edit Mode» обрати режим вибору граней, обрати потрібну грань і натиснути клавішу «X», обрати режим видалення «Faces» (див. рис. 1.4).

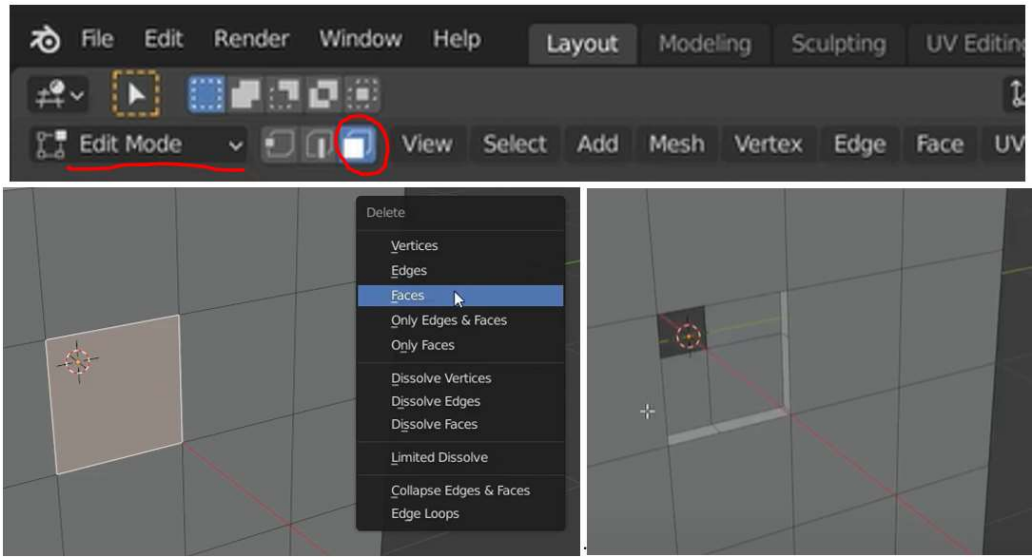

Рисунок 1.4 – Blender. Видалення граней [13].

Другий спосіб створення отвору це утворення щілини. Порядок дій: в режимі «Edit Mode» обрати режим вибору вершин, обрати вершину, меню «Vertex», опція «Rip Vertices» і зрушити вершину за допомогою переміщення мишки з натиснутою лівою кнопкою (див. рис. 1.5). Функція «Rip» створює копію обраної вершини і розміщує в новому положенні. Частина граней, що включала ту вершину, тепер змінилася так, що має вершину в іншому місці.

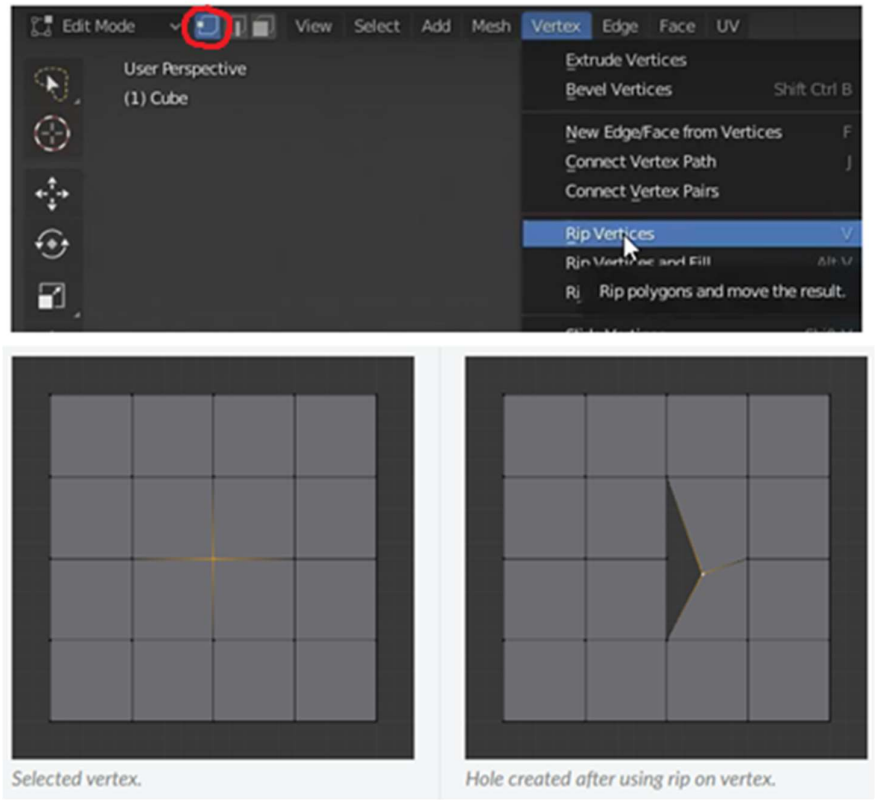

Рисунок 1.5 – Blender. Створення щілини [13].

Інтерфейс редактора перенавантажений, бо Blender має дуже багато можливостей, для комфортного використання бажано запам’ятати гарячі клавіші для перемикання між режимами перегляду, вибору, зміни та інше.

Висновок: Blender відповідає вимогам, але складний в освоєнні. Це потужний інструмент для модифікації полігональної сітки, створення отворів можливе двома шляхами, функції дуже гнучкі, користування повністю безкоштовне. Головний недолік в перенавантаженості інтерфейсу, широкий спектр можливостей, способів відображення, режимів модифікації ускладнює

розуміння програми, для вільного користування даним САПР рекомендовано пройти етап навчання (інтернет-курси, документація або туторіали).

1.4.2.2 3ds Max

3ds Max — тривимірний графічний редактор, повнофункціональний професійний застосунок, система для створення і редагування об'єктів та створення візуалізацій, розроблена компанією Autodesk. Містить найсучасніші засоби для архітекторів, дизайнерів, художників і фахівців в області мультимедіа [14]. Програма має пробний період на 1 місяць та безкоштовне користування для студентів в період навчання, в іншому випадку це коштує $225 на місяць на момент написання роботи [15].

Підтримка формату STL та COLLADA ✓

Використання без спеціальної підготовки ✘

Для створення отворів в 3ds Max треба мати навички роботи з редактором, бо інтерфейс пропонує безліч можливостей, серед яких початківцю важко знайти потрібний (див. рис. 1.6 та 1.7).

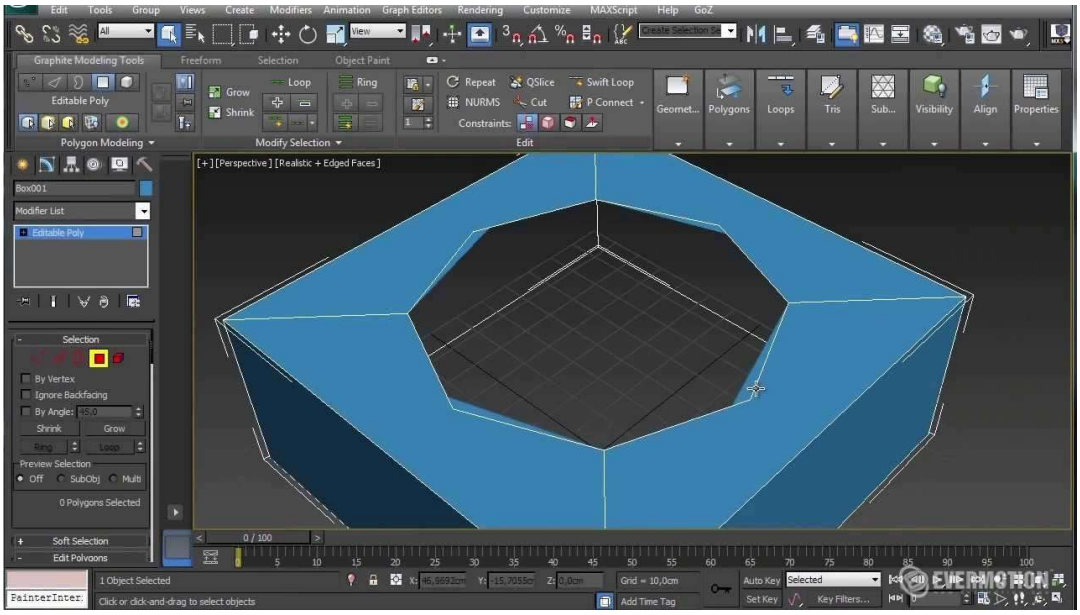

Рисунок 1.6 – Інтерфейс 3ds Max. Створення та редагування отворів в режимі модифікації граней [16].

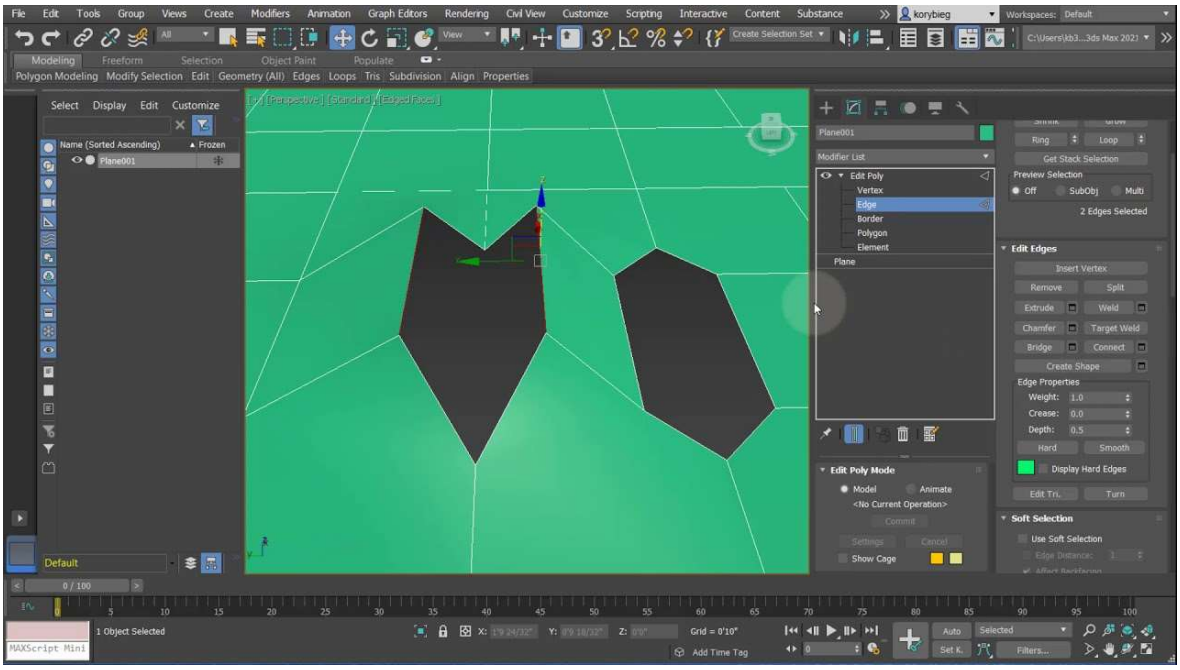

Рисунок 1.7 – Інтерфейс 3ds Max. Створення та редагування отворів в режимі модифікації ребер [16].

Висновок: Способів створення отворів дуже багато: видалення граней або окремих трикутників, утворення щілин за допомогою переміщення вершин або ребер, є можливість редагування країв отвору тощо. Тому, підсумовуючи, 3ds Max – дуже потужний інструмент, але з дуже складний в освоєнні (суб'єктивно складніший за Blender), для освоєння необхідно пройти підготовку, а також, доведеться купувати ліцензію при завершенні пробного періоду.

1.4.2.3 PowerShape

PowerShape – програмне забезпечення САПР для складних деталей, проєктування електродів та моделювання для виробництва [17], розроблене компанією Autodesk, не безкоштовне, ціна залежить від різних факторів.

Підтримка формату STL	✓	COLLADA	✘
Використання без спеціальної підготовки	✘		

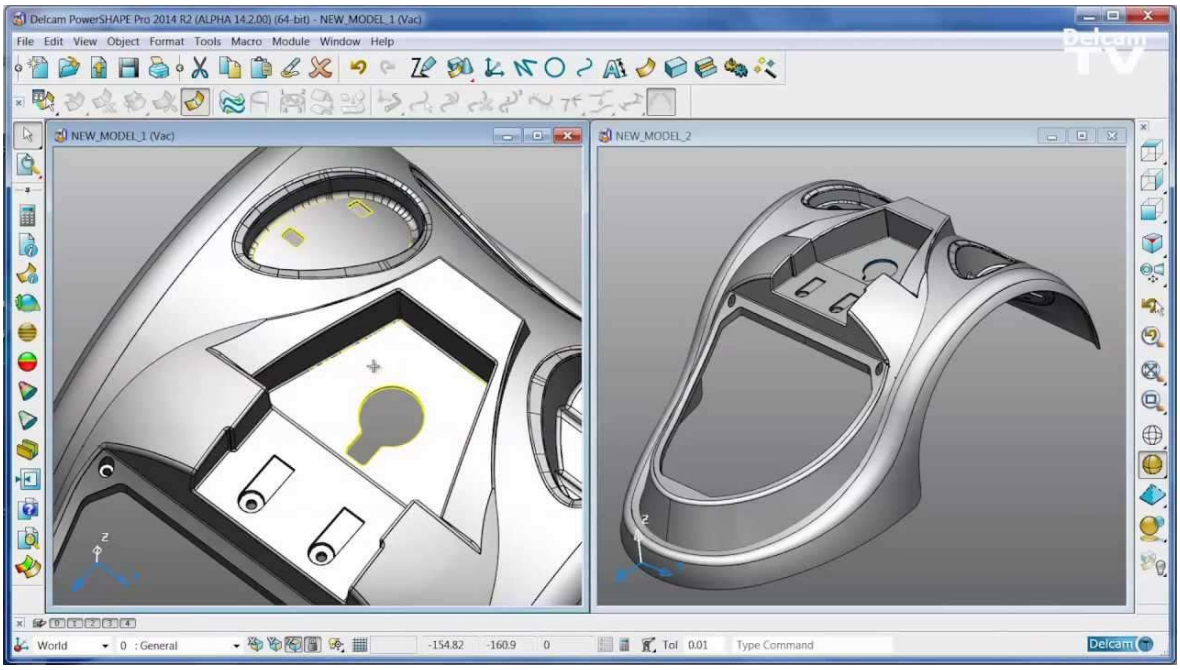

Рисунок 1.8 – Інтерфейс PowerShape [17].

Висновок: PowerShape призначений для кваліфікованих інженерів та не розповсюджується у відкритому доступі. Також він менш популярний ніж 3ds Max та Blender, тому навіть маючи бажання пройти етап навчання, зробити це буде складніше, бо кількість курсів та уроків набагато менша, ймовірно, вибір буде зроблено в користь платних курсів. Також немає можливості імпортування моделей формату COLLADA.

1.4.2.4 Meshlab

MeshLab — система з відкритим кодом для обробки та редагування трикутних сіток в 3D. Він надає набір інструментів для редагування, очищення, відновлення, перевірки, візуалізації, текстурування та перетворення сіток. Він пропонує функції для обробки необроблених даних, створених інструментами/пристроями 3D оцифровки, і для підготовки моделей до 3D-друку [18].

Підтримка формату STL та COLLADA	✓
Використання без спеціальної підготовки	✓

Перед використанням бажано переглянути туторіали, але інтерфейс не такий складний як в 3ds Max та Blender. Для створення отворів в MeshLab треба виконати такі дії (див. рис. 1.9): обрати в верхньому меню режим вибору трикутників (рис. 1.9.a), обрати трикутник кліком або виділити область прямокутної форми (рис. 1.9.b), в верхньому меню обрати режим видалення трикутників (рис. 1.9.c) і обрати зафарбований трикутник з вершинами або без.

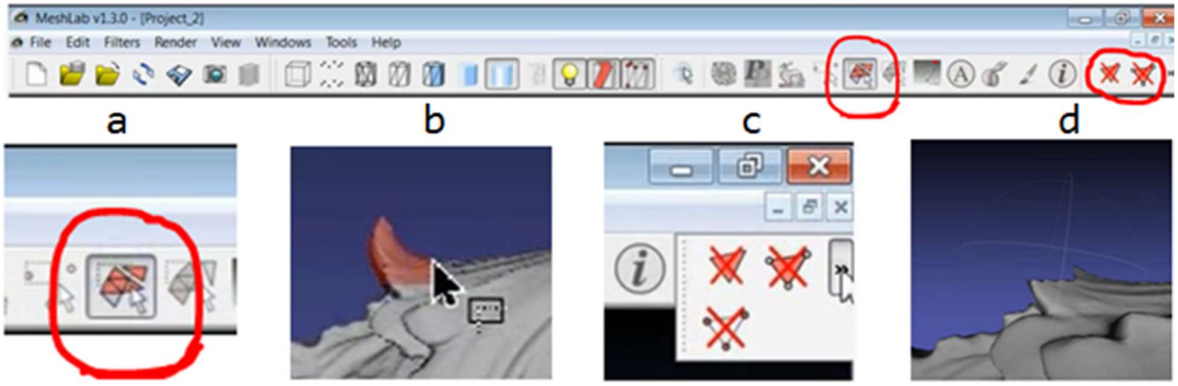

Рисунок 1.9 – створення отворів в MeshLab.

Висновок: Meshlab найбільше підходить для поставленої задачі серед переглянутих САПР: Meshlab безкоштовний, підтримує велику кількість форматів файлів, має можливість видаляти та додавати трикутники та дуже багато інших можливостей, з акцентом на роботі з трикутною сіткою та 3D скануванням.

Серед негативних моментів – небагато ресурсів для навчання роботи з даним інструментом, через що можуть виникати складнощі.

1.5 Постановка задачі

Інформаційний огряд показав, що найпопулярніші формати файлів з 3D моделями – це STL і COLLADA, вони збергігають інформацію про геометрію моделі у вигляді полігональої сітки, що описує поверхню моделі як набір координат вершин та трикутників, що їх поєднує. Зазвичай такі моделі мають цілісну поверхню, тому моделі з отворами дуже складно знайти в інтернеті, що підтверджує потребу в інструменті для створення отворів в полігональних моделях.

Проведений огляд показав. що жоден з наявних продуктів не забесзпечує досягнення мети у повному обсязі. Відтак задачею роботи буде створення додатку, який має віміти:

- читати та записувати файли форматів STL і COLLADA;
- стоворювати отовори в полігональній сітці;
- візуально інформувати користувача про вже наявні отвори;

Також, важливими критеріями є:

- простота інтерфейсу;
- зручне керування;
- можливість використання без спеціальних навичків роботи в САПР.

2 ВИБІР МЕТОДІВ РІШЕННЯ ЗАДАЧІ

2.1 Вибір способу взаємодії з користувачем

Програмне рішення має вміти завантажувати 3D моделі формату COLLADA та STL, створювати отвори в трикутній сітці, зберігати модель у форматі COLLADA та STL. Всі САПР – це десктопні застосунки, тому наш додаток також буде десктопним графічним застосунком, з основною зоною – тривимірною сценою, на якій знаходиться імпортована модель та відбуваються маніпуляції. Видалення трикутників буде відбуватись по кліку миші, якщо ввімкнено режим видалення трикутників. Інші маніпуляції з мишкою, такі як перетягування з затиснутими клавішами, будуть трактовані як маніпуляції з камерою (зміна кута зору, обертання камери або переміщення). В режимі видалення трикутників, при наведенні на трикутник відбувається його підсвічування. Після натискання в зоні трикутника відбувається його видалення, створюється отвір. Після створення отвору (або якщо в моделі вже були отвори) границі отворів підсвічуються червоними лініями. Для ввімкнення та вимкнення режиму видалення трикутників буде створена кнопка в верхній панелі інструментів та призначена гаряча клавіша Delete (Escape для виходу з режиму).

Для завантаження та збереження змін будуть спеціальні кнопки в верхній панелі інструментів, а також стандартні гарячі клавіші – Ctrl+L та Ctrl+S відповідно. Вибір файлу для завантаження та шляху для збереження буде відбуватись у випливаючому вікні з файловим провідником.

Для маніпуляцій з камерою будуть застосовані стандартні для всіх САПР механізми: користувач може вільно керувати камерою, переміщуючись по своїй осі рухом мишки з затиснутою правою кнопкою та здійснюючи оберти навколо центра сцени рухом мишки з затиснутою лівою кнопкою. Віддалятись та наближатись до моделі за допомогою скролінгу. Кнопка «Z» здійснює віддалення або наближення камери відносно центру моделі так, щоб

було видно всю сцену у вікні проєкції. Кнопки «F1» – «F7» перемикають між стандартними положеннями камери (front, back, up, bottom, right, left, isoview відповідно). Кнопка «F8» здійснює перемикання між перспективною та паралельною проєкцією.

Додатково створена можливість розрізання моделі площиною. Для цього буде створена кнопка, пісня натискання з’явиться площина, яку можна переміщувати та обертати так само як і інших режимах відбувається маніпуляція з камерою. Після повторного натискання на кнопку модель розділяється на 2 половини, що автоматично створює 2 (або більше) отворів.

2.2 Вибір мови програмування та графічного API

Для розробки редактору обрано мову програмування C++. Основна частина коду буде присвячена рендерингу, роботі з камерою, та геометричним розрахункам, тому C++ ідеально підходить в даній ситуації. Для доказу можна згадати, що майже всі ігрові рушії були написані саме на C++ (Dagor Engine 5.0 (C/C++), CryEngine V (C++/C#/Lua), id Tech 6, RenderWare, Source 2, Unity 2017 (C/C++/C#), Unreal Engine 4).

Серед різних графічних API (OpenGL, Vulkan, DirectX, Allegro) для застосунку було обрано OpenGL, бо це найкраще середовище для розробки портативних інтерактивних 2D та 3D-графічних додатків. З моменту своєї появи в 1992 році OpenGL став широко використовуваним і підтримуваним програмним інтерфейсом 2D і 3D-графіки (API), який забезпечує тисячі додатків на різних комп'ютерних платформах. OpenGL сприяє інноваціям та прискорює розробку додатків за рахунок включення широкого набору рендерингу, складання текстур, спеціальних ефектів та інших потужних функцій візуалізації. Розробники можуть використовувати можливості OpenGL для всіх популярних десктопних та робочих платформ, забезпечуючи широке розгортання програм.

Оскільки основою редактора буде OpenGL, то найкращим вибором бібліотеки для математичних розрахунків буде бібліотека OpenGL Mathematics, скорочено GLM. Це бібліотека для OpenGL, що надає програмісту на C++ структури та функції, що дозволяють використовувати дані для OpenGL та проводити математичні розрахунки. Одна з особливостей GLM полягає в тому, що його реалізація ґрунтується на специфікації GLSL (OpenGL Shading Language). Це хороший кандидат для програмної візуалізації (трасування променів / растеризації), обробки зображень, фізичного моделювання та будь-якого контексту розробки, який вимагає простої та зручної математичної бібліотеки.

Для створення вікна використано бібліотеку GLFW. GLFW – це легка бібліотека-утиліта для використання з OpenGL. GLFW означає Graphics Library Framework. Він надає програмістам можливість створювати вікна та контексти OpenGL і керувати ними, а також керувати реакцію на події з маніпуляторами: джойстиком, клавіатурою та мишею.

Для коректного підключення та взаємодії різних бібліотек OpenGL використовується Glad – багатомовний генератор-завантажувач GL/GLES/EGL/GLX/WGL на основі офіційних специфікацій.

Для читання та записування 3D моделі в файл формату COLLADA необхідно підключити бібліотеку TinyXML, оскільки формат COLLADA базується на XML. Extensible Markup Language (XML) — це мова розмітки та формат файлів для зберігання, передачі та відновлення довільних даних. Він визначає набір правил для кодування документів у форматі, який зрозумілий як людині, так і машині. TinyXML – це невеликий, простий, залежний від операційної системи XML-парсер для мови C ++.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Інтерфейс та приклади використання

Інтерфейс застосунку містить область для рендерингу сцени та панель інструментів яка містить кнопки. Створення вінка та рендеринг, реакції на події імплементовано самостійно за допомогою OpenGL і описано в розділі 3.2.

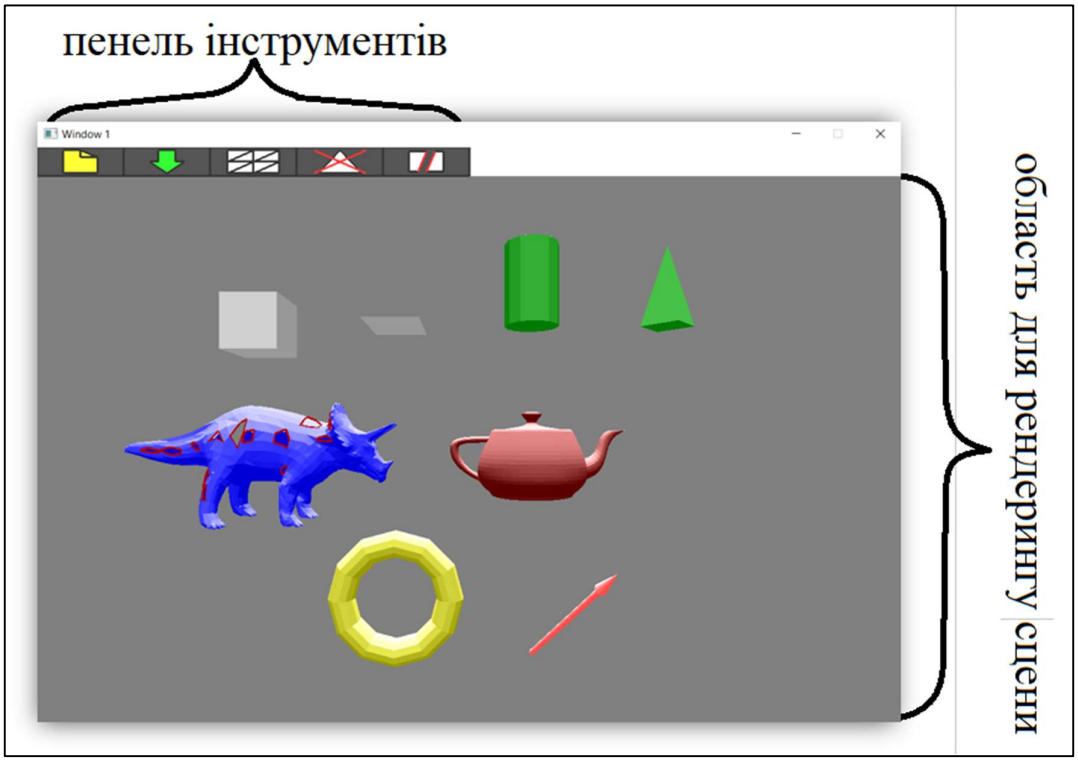

Рисунок 3.1 – Інтерфейс застосунку.

На рисунку 3.1 сцена має декілька 3D моделей, серед них носоріг – модель завантажена з файлу STL, чайник – з файлу COLLADA, а всі інші – створені алгоритмічно з можливістю зміни розмірів та ступеню теселяції. За допомогою застосування матеріалів до об’єктів та налаштуванню джерел світла, моделі можуть мати різні кольори та характеристики відбиття світла, що сприяє кращому візуальному розумінню форми моделі.

Розглянемо кнопки панелі інструментів на рисунку 3.2. Опис імплементації візуалізації кнопок та логіки реагування на їх натискання описано в розділі 3.3, а зараз розглянемо функціонал за який вони відповідають та наведемо приклади використання.

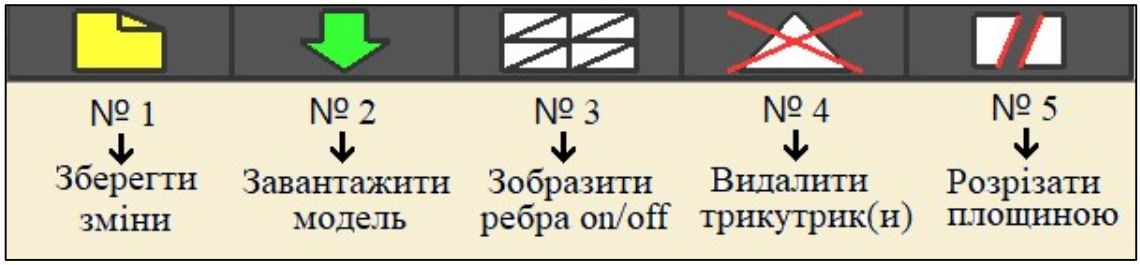

Рисунок 3.2 – Кнопки панелі інструментів.

Кнопка «Зберегти зміни»

Кнопка №1 з зображенням файлу – це кнопка «Зберегти зміни». Вона застосовується до поточного вікна. Відредагована модель, що знаходиться на сцені в даному вікні зберігається в файл формату STL або COLLADA в залежності від того в якому форматі модель була завантажена. Шлях та назва відредагованої копії визначається за допомогою файлового провідника (рис 3.3).

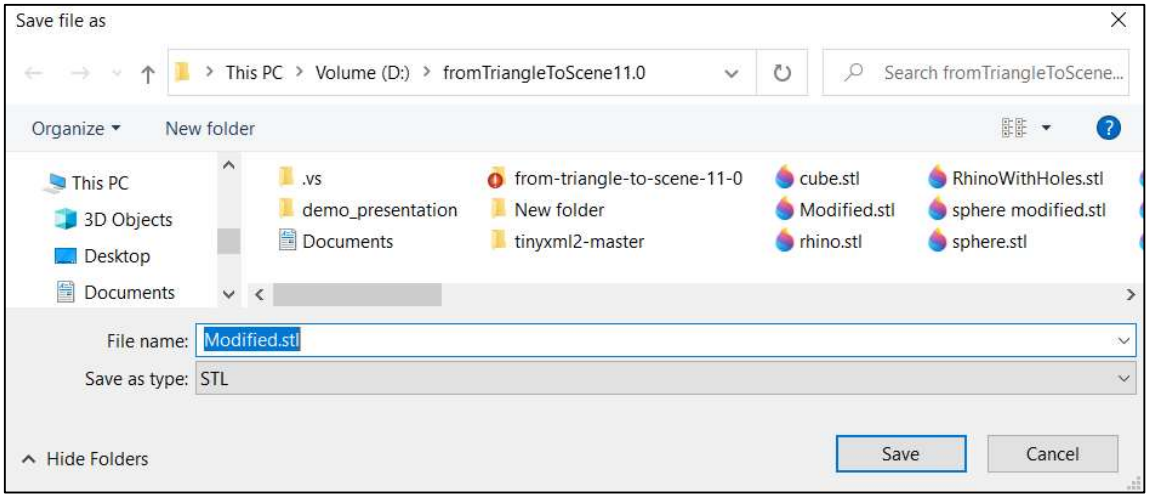

Рисунок 3.3 – Вікно файлового провідника для збереження файлу зі змінами.

Кнопка «Завантажити модель»

Кнопка №2 з зображенням зеленої стрілки – це кнопка «Завантажити модель». Відкривається вікно з файлом провідником для вибору моделі, доступні формати: STL та COLLADA. Модель з’являється в новому вікні.

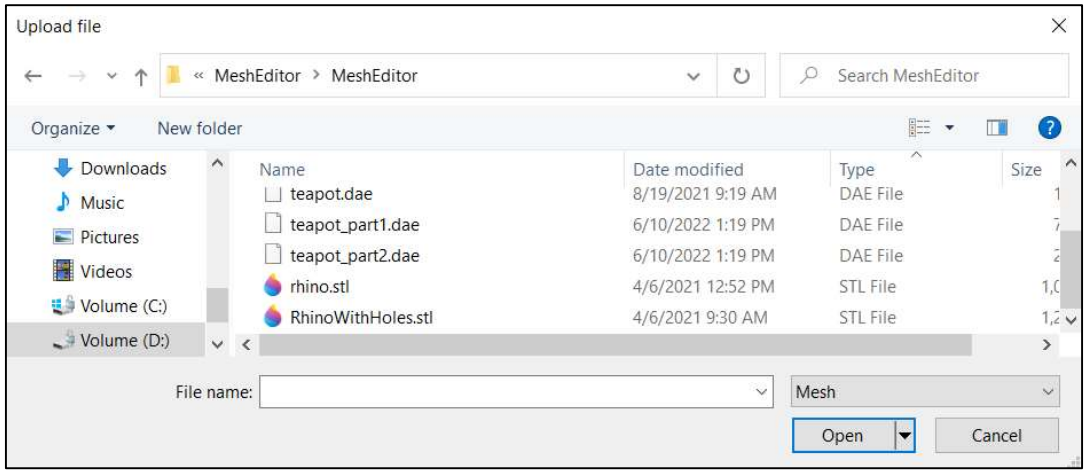

Рисунок 3.4 – Вікно файлового провідника для завантаження моделі.

Кнопка «Зобразити ребра»

Кнопка №3 з зображенням трикутної сітки – це кнопка «Зобразити ребра». Після її натискання з’являються лінії чорного кольору, які показують межі трикутників, з яких складається модель. Кнопка працює як перемикач, тобто при повторному натисканні лінії зникнуть.

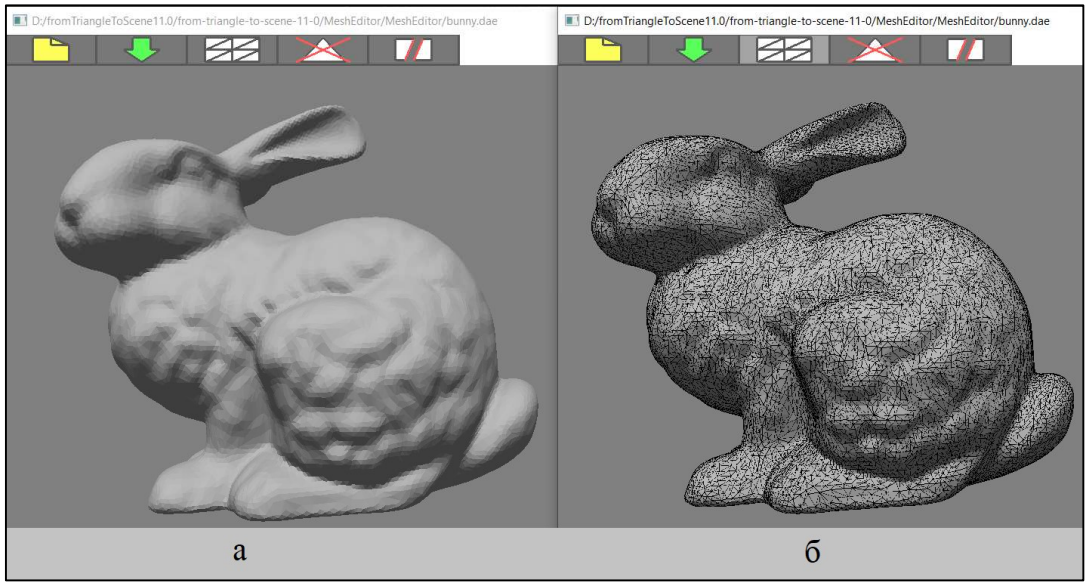

Рисунок 3.5 – Відображення ребер: а – вимкнено, б – увімкнено.

Кнопка «Видалити трикутник»

Кнопка №4 з зображенням закресленого трикутника – це кнопка «Видалити трикутник». Це перший з двох спосіб створення отворів. Вмикається режим, в якому натискання на деякий трикутник моделі призводить до його видалення. Для прикладу створимо отвір в військовій техніці окупанта. На рисунку 3.6 завантажена модель в форматі коллада. В деяких місцях є червоні лінії: вони показують місця де є порушення цілісності поверхні моделі, тобто танк уже дірявий.

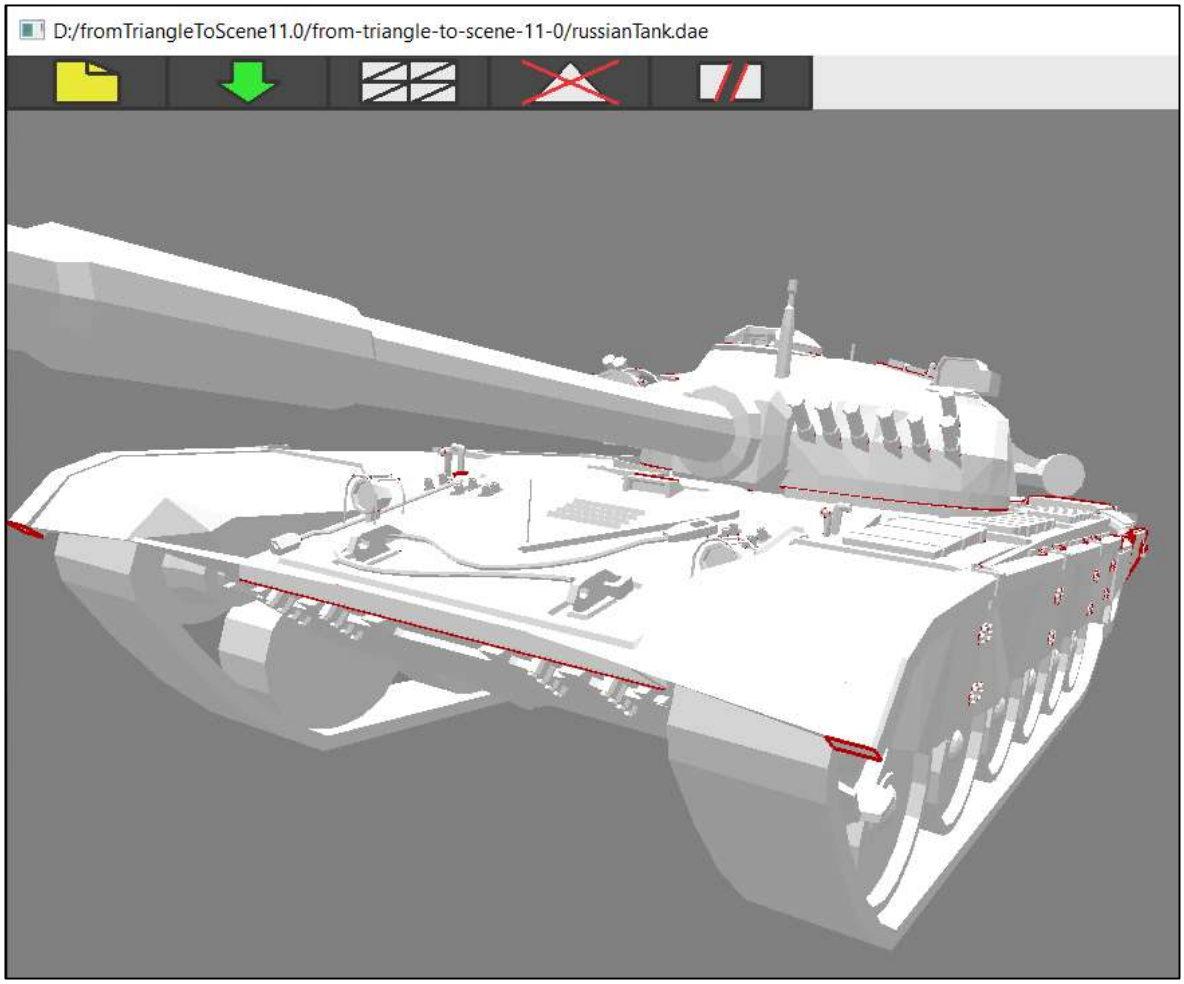

Рисунок 3.5 – Дірявий танк окупанта.

Після натискання кнопки «Видалити трикутник» до моделі автоматично вмикається відображення ребер, а при наведенні курсору на модель, трикутник на якому знаходиться курсор підсвічується червоним (див. рис. 3.6).

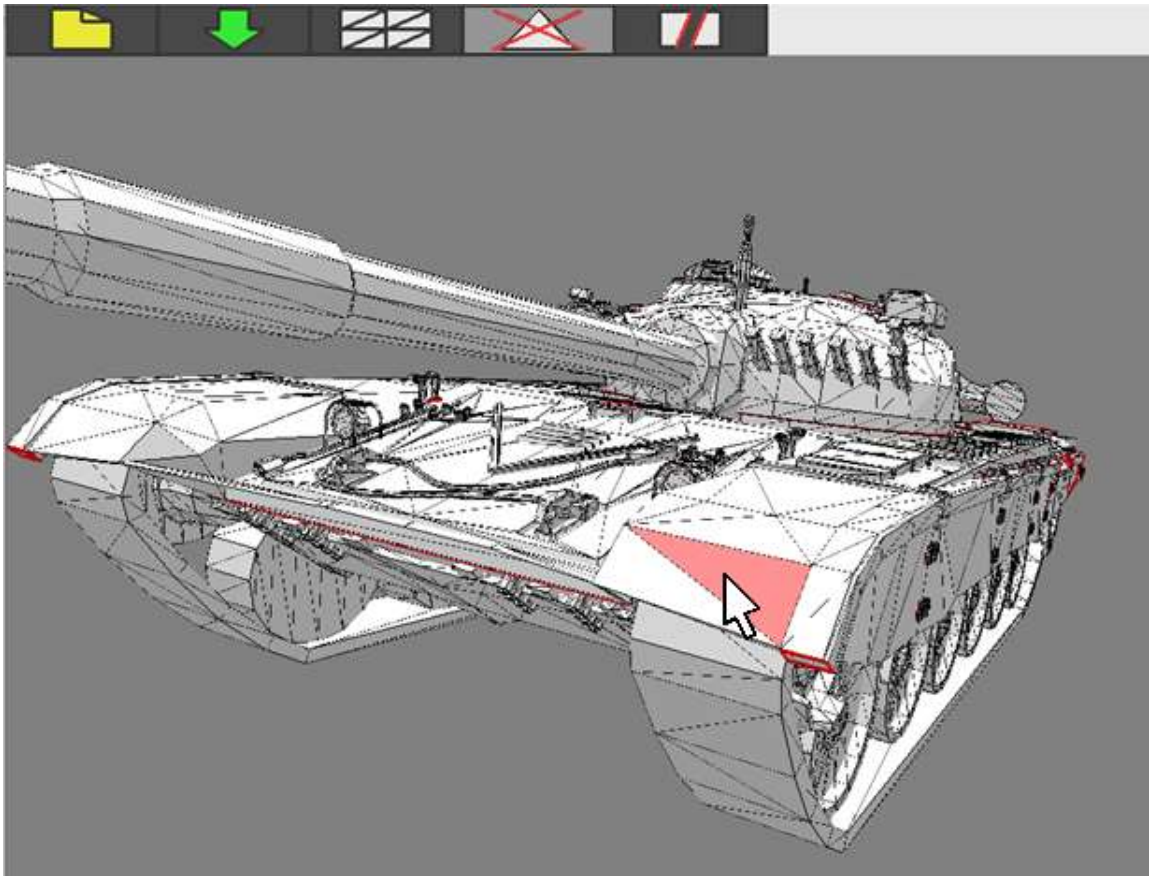

Рисунок 3.6 – Виділення трикутника на який вказує курсор.

Можемо видалити не один, а декілька трикутників, а за бажанням можна знищити танк повністю. Створені отвори помічаються лініями червоного кольору (рис 3.7). Як видно з малюнку, отвори можуть бути як трикутної форми, так і багатокутної, але межі отворів завжди виділяються коректно, незалежно від того в якому порядку видалялись трикутники. Це досягається шляхом використання структури даних HalfEdge, яка зберігає дані про зв’язки між сусідніми трикутниками і про те яким саме двом трикутникам належить кожне ребро. При завантаженні моделі відбувається побудова структури HalfEdge, а після видалення трикутника відбувається корекція зв’язків між трикутниками, ребрами та вершинами в зоні навколо видаленого трикутника. Для виходу з режиму видалення трикутників потрібно повторно натиснути на кнопку.

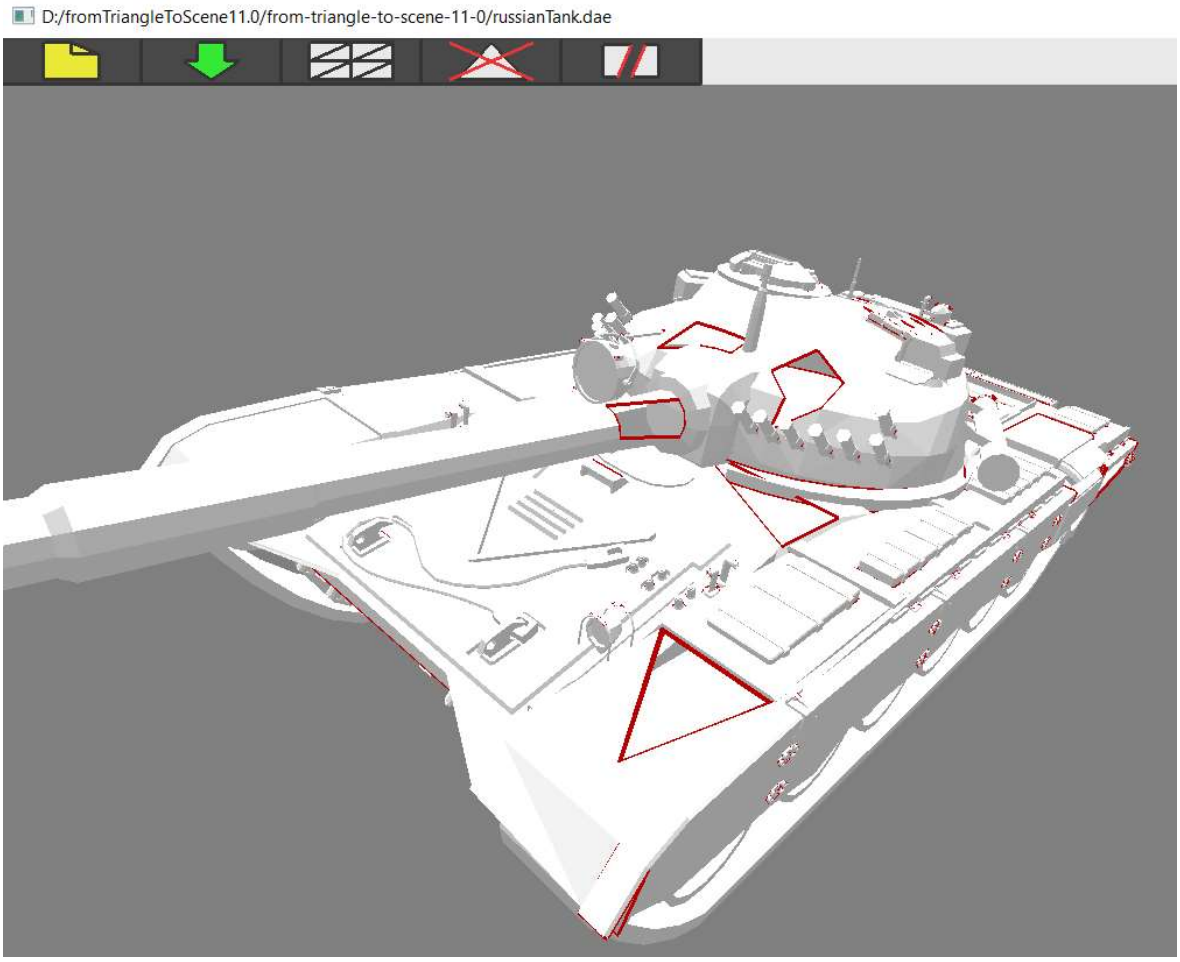

Рисунок 3.7 – Результат видалення трикутників – порушення цілісності поверхні танку. Червоні лінії демонструють межі отворів.

Кнопка «Розрізання площиною».

Кнопка №5 з зображенням розрізаного прямокутника – це кнопка «Розрізання площиною». При її натисканні з’являється площина (див. рис 3.8), яку можна повертати та переміщувати, так само як в інших режимах здійснюється переміщення камери: перетягування мишки з затиснутою лівою клавішею – обертання, з правою клавішею – переміщення. Тоді коли площина набула бажаного положення і користувач бажає здійснити переріз, потрібно повторно натиснути кнопку і розрізання відбудеться. Модель буде розділена на 2 частини, кожна з яких буде збережена в окремому файлі, а на екрані з’явиться 2 нових вікна для візуалізації результату.

Як видно з рисунку 3.8, у кожній із двох частин моделі з’явилися отвори, що помічені лініями червоного кольору. Якщо трикутник перетинався площиною, то він буде розділений на 2 частини по лінії перетину і кожна з частин опиниться у відповідному вікні результату.

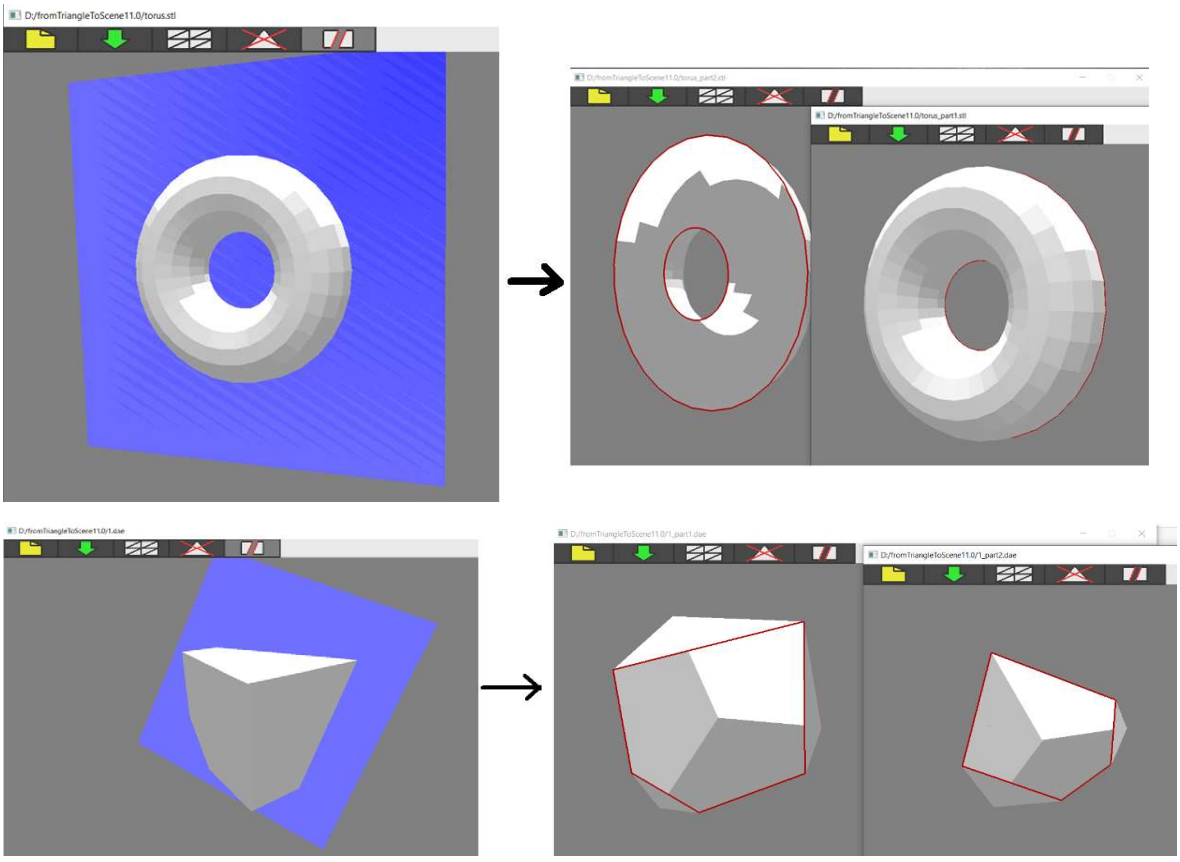

Рисунок 3.8 – Розрізання моделі площиною. Процес та результат.

На рисунку 3.8 показано розрізання торуса площиною в початковому положенні – вертикальному, і розрізання куба площиною, що була певним чином нахилена та переміщена. Після завершення процесу площина зникне з екрана, а модель в початковому вікні залишиться незміненою.

Змінені файли будуть збережені в тому ж місці, що і початкова модель, з тим самим ім’ям, але з додаванням «_part1» та «_part2» до імені першої та другої частини відповідно.

3.2 Імплементація створення вікна та рендерингу 3D сцени

Створення вікна та рендерингу імплементовано з використанням функцій OpenGl та GLWF. Для спрощення API взаємодії з вікном, створено класи-обгортки GLWindow та GLRenderSystem. Створенням вікна займається клас GLWindow, використовуючи функції бібліотеки GLWF. Конструктор вікна приймає параметри назви, ширину та висоту в пікселях.

```
GLWindow::GLWindow(const std::string& title, uint32_t width, uint32_t height) :
IWindow(), _width(width), _height(height) {
	static bool initGLFW = false;
	if (!initGLFW) {
		initGLFW = true;
		glfwInit();
	}
	glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);
	_handle = glfwCreateWindow(width, height, title.data(), nullptr, nullptr);
	glfwMakeContextCurrent(_handle);
```

Ініціалізується бібліотека GLFW, яка займається роботою з вікном OpenGL, її функція glfwCreateWindow створює вікно і повертає покажчик на нього.

```
	static bool initGLAD = false;
	if (!initGLAD) {
		initGLAD = true;
		gladLoadGLLoader((GLADloadproc)glfwGetProcAddress);
	}
	glfwSetWindowUserPointer(_handle, this);
```

Далі ініціалізується бібліотека GLAD яка керує імпортом потрібних версій OpenGL бібліотек. Функція glfwSetWindowUserPointer(handle, this) прив’язує GLFW вікно до нашої структури GLWindow.

```
	glShadeModel(GL_SMOOTH);
	glEnable(GL_DEPTH_TEST);
	glDepthFunc(GL_LEQUAL);
	glEnable(GL_NORMALIZE);
	glEnable(GL_COLOR_MATERIAL);
	glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT);
}
```

В кінці відбувається налаштування шейдеру OpenGL: ввімкнення згладжування, тесту глибини, нормалізації векторів нормалі та використання матеріалів.

Рендеринг 3D сцени в 2D зображення відбувається методами OpenGL специфікації 2.0. Основний цикл рендерингу представлений нижче.

```
while (!_shouldStop) {
	for (auto it = _views.begin(); it != _views.end(); ++it) {
		auto& view = *(it->get());
		makeContextCurrent(view.getWindow());
		_renderSystem->clearDisplay(0.5f, 0.5f, 0.5f);
		view.update(*_renderSystem);
		swapDisplayBuffers(view.getWindow());
		waitEvents();
		if (windowShouldClose(view.getWindow()))
			_views.erase(it);
	}
	if (_views.empty()) {
		_renderSystem.reset();
		terminateWindowsResources();
		_shouldStop = true;
	}
}
```

Тут кожному створеному вікну відповідає свій об’єкт view. Функція makeContextCurrent налаштовує OpenGL так щоб подальші дії відбувались з саме вказаним вікном, clearDisplay робить очищення буферу пікселів і створює однотоннне полотно, view.update(*_renderSystem) запускає рендеринг, а swapDisplayBuffers – функція яка потрібна для такого прийому покращення візуалізації що називається подвійною буферизацією [19]. Якщо вікно було закрите, windowShouldClose повертає true і поточне вікно видаляється. Коли всі вікна будуть закриті, ресурси вікна будуть звільнені і цикл завершиться.

Розглянемо функцію update з попереднього прикладу:

```
void View::update(GLRenderSystem& rs) {
	rs.setViewport(0, 0, _viewport.getWidth(), _viewport.getHeight());
	auto viewMatrix = _viewport.getCamera().calcViewMatrix();
	auto projMatrix = _viewport.calcProjectionMatrix();
	auto globalToScreen = projMatrix * viewMatrix;
	for (Mesh& mesh : _model->_meshes) {
		rs.setupMaterial(mesh.getMaterial());
		auto absoluteTransform = globalToScreen * mesh.GetWorldMatrix();
		auto triangles = mesh.GetTriangles();
		auto lines = mesh.GetLines();
		for (auto& tr : triangles)
			tr.Transform(absoluteTransform);
		for (auto& ln : lines)
			ln.Transform(absoluteTransform);
		rs.renderTriangles(triangles);
		rs.renderLines(lines);
	}
}
```

Тут трикутники та лінії всіх об’єктів на сцені зазнають трансформації (результуюча матриця трансформації створена шляхом перемноження матриць таким чином, що в результаті до об’єкта спочатку застосовується його особиста трансформація (mesh.GetWorldMatrix()), потім трансформація положення камери (viewMatrix), а потім трансформація, яка створює перспективну проекцію на екран (projMatrix). Далі трикутники та лінії передаються на рендеринг екземпляру класу GLRenderSystem в методи rs.renderTriangleSoup(triangles) та rs.renderLines(lines).

Клас GLRenderSystem займається рендерингом зображення на екрані, а також налаштуванням джерел світла та матеріалів.

```
void GLRenderSystem::renderTriangles(const std::vector<Triangle>& triangles) const {
	glBegin(GL_TRIANGLES);
	for (auto& tr : triangles)
	{
		glColor4d(tr.color.r, tr.color.g, tr.color.b, 0.0);
		glNormal3d(tr.normal.x, tr.normal.y, tr.normal.z);
		glVertex4d(tr.a.x, tr.a.y, tr.a.z, tr.a.w);
		glVertex4d(tr.b.x, tr.b.y, tr.b.z, tr.b.w);
		glVertex4d(tr.c.x, tr.c.y, tr.c.z, tr.c.w);
	}
	glEnd();
}
void GLRenderSystem::renderLines(const std::vector<renderElements::Line>& lines) const {
	glLineWidth(lines[0].width);
	glBegin(GL_LINES);
	for (auto& l : lines)
	{
		glColor4d(l.color.r, l.color.g, l.color.b, 0.0);
		glVertex4d(l.a.x, l.a.y, l.a.z, l.a.w);
		glVertex4d(l.b.x, l.b.y, l.b.z, l.b.w);
	}
	glEnd();
}
```

Рендеринг трикутників починається викликом функції glBegin з параметром GL_TRIANGLES, а ліній – GL_LINES. Функція glVertex4d задає координати вершини трикутника або кінця відрізку, glColor4d налаштовує колір для кожної точки, але оскільки вона викликається 1 раз для 1 елемента, то весь елемент буде мати цей колір, а функція glNormal3d застосовується, щоб задати значення нормалі, щоб розуміти яке положення щодо джерел світла має поверхня трикутника і яким чином це впливає на колір трикутника.

3.3 Імплементація керування реакціями на події маніпуляторів

Для того, щоб аналізувати події вікна такі як натискання мишки або її переміщення, натискання кнопок клавіатури або кнопок панелі інструментів створено клас OperatorDispatcher. Він зберігає всі можливі реакції і коли відбувається будь-яка подія, OperatorDispatcher обирає яку реакцію потрібно застосувати в конкретно даній ситуації. Реакції зберігаються в нащадках класу Operator. Почнемо з опису інтерфейсу класу OperatorDispatcher:

```
class OperatorDispatcher
{
private:
	std::map<KeyCode, std::unique_ptr<Operator>> _keyOperators;
	std::unique_ptr<Operator> _scrollingOperator;
	std::unique_ptr<Operator> _rightButtonMoveOperator;
	std::unique_ptr<Operator> _leftButtonMoveOperator;
	std::unique_ptr<Operator> _activeModeOperators;
```

Поля класу OperatorDispatcher зберігають набори операторів: keyOperators – реакції на кнопки клавіатури, зберігає відповідність між ключем-кнопкою та оператором-реакцією, scrollingOperator – реакція на прокручування колеса мишки, rightButtonMoveOperator – оператор який відповідає переміщення мишки з затиснутою правою клавішею, leftButtonMoveOperator – з затиснутою лівою клавішею. Деякі реакції не можуть бути логічно завершеними використовуючи тільки 1 подію, тому вони розбиваються на етапи. Тоді оператор виконує не одну дію, а декілька, а період між початком виконання «місії» та завершенням – це період в який даний оператор є активним. Поле activeModeOperators зберігає оператор який цієї миті знаходиться в активному стані.

```
	std::vector<Button> windowButtons;
```

Поле windowButtons зберігає всі кнопки панелі інструментів, так що при натисканні в зоні деякої кнопки, OperatorDispatcher викликає функцію, яка призначена даній кнопці.

```
        void addButtonOperator(ButtonCode button, std::unique_ptr<Operator> op);
        void addKeyOperator(KeyCode key, std::unique_ptr<Operator> op);
        void addMouseMoveOperator(ButtonCode button, std::unique_ptr<Operator> op);
        void addScrollingOperator(std::unique_ptr<Operator> op);
```

Тут об’явлені add-методи які призначають оператори до подій.

```
        void processMouseInput(View& view, ButtonCode button, Action action, Modifier
mods, double x, double y);
        void processKeyboardInput(View& view, KeyCode key, Action action, Modifier mods);
        void processScrolling(View& view, double x, double y);
        void processMouseMove(View& view, Action leftButtonCondition, Action
rightButtonCondition, double x, double y);
};
```

Тут об’явлені process-методи які аналізують реакції з врахуванням активних режимів та налаштувань конкретних операторів.

Перейдемо до опису класу Operator. Це віртуальний клас, тобто створюючи нового нащадка оператора, ми маємо перевизначити деякі його методи для того що отримати бажану поведінку.

```
class Operator
{
public:
        virtual ~Operator() {}
        virtual void onEnter(View&) {}
        virtual void onExit(View&) {}
        virtual void onMouseMove(View& view, double x, double y) {}
        virtual void onMouseInput(View& view, ButtonCode button, Action action, Modifier
mods, double x, double y) {}
        virtual void onKeyboardInput(View& view, KeyCode key, Action action, Modifier
mods) {}
        virtual void onScrolling(View& view, double x, double y) {}

        Action leftButtonCondition = Action::Release;
        Action rightButtonCondition = Action::Release;

        bool processOtherOperators = true;
};
```

Operator має методи onMouseMove, onMouseInput, onKeyboardInput, onScrolling які займаються опрацюванням подій руху мишки, кліком мишки, клавіатури та скролінгу відповідно. Методи onEnter та onExit призначені для таких операторів які опрацьовують не одну подію, щоб виконати своє завдання, тому мають періоди активності та неактивності. Коли такий оператор стає активним, для нього викликається метод onEnter, а при деактивації – метод onExit.

Далі в роботі ми будемо описувати створення окремих операторів, таких як оператор видалення трикутника або оператор завантаження моделі з файлу.

В огляді структури класу OperatorDispatcher ми говорили про поле std::vector<Button> windowButtons. Опишемо структуру класу Button.

```
struct Button {
	Operator* op;

	bool onOfType = true;
	bool active = false;
	bool setActiveOperatorMode = true;
```

Поле "op" зберігає оператор, для якого створена кнопка, поле onOfType показує чи є оператор таким, що має режими активності та неактивності або це простий оператор на 1 дію. Якщо це режимний оператор, то поле active вказує чи є кнопка активною цієї миті, а setActiveOperatorMode чи потрібно віддавати пріоритет в обробці подій даному оператору якщо він активний.

```
	glm::vec2 minCorner;
	glm::vec2 maxCorner;
```

В полях minCorner та maxCorner зберігаються координати кнопки на екрані в екранній системі координат (від -1 до 1 по x та y). Тому ми можемо ідентифікувати натискання на кнопку якщо координати курсора в процесі натискання знаходяться всередині зони кнопки.

```
	std::vector<renderElements::Triangle> triangles;
	std::vector<renderElements::Line> lines;
};
```

Для того, щоб зобразити кнопки на екрані ми повинні задати їм певного виду за допомогою трикутників та ліній. Наприклад кнопка видалення трикутників складається з двох трикутників які формують фон, 1 білого трикутника, 3 чорних ліній по периметру та 2 червоних ліній для хрестика (див рис 3.9).

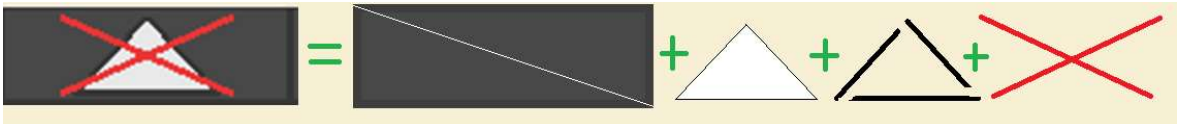

Рисунок 3.9 - Імплементація візуалізації кнопок.

А зараз повернемось до класу OperatorDispatcher. Продемонструємо роботу класу на прикладі реалізації метода processMouseInput.

```
void OperatorDispatcher::processMouseInput(View& view, ButtonCode button, Action action,
Modifier mods, double x, double y)
{
	for (auto& btn : WindowButtons::buttons)
	{
		if (btn.minCorner.x <= x && x <= btn.maxCorner.x &&
			btn.minCorner.y <= y && y <= btn.maxCorner.y &&
			action == Action::Press)
		{
			if (!btn.onOfType)
			{
				btn.op->onMouseInput(view, button, action, mods, x, y);
				return;
			}

			btn.active = !btn.active;
			btn.HighLight(btn.active);
			if (btn.setActiveOperatorMode)
				_activeModeOperators = btn.active ? btn.op : nullptr;

			if (btn.active)
				btn.op->onEnter(view);
			else
				btn.op->onExit(view);
			return;
		}
	}
```

В першу чергу перевіряється чи не знаходиться курсор в одній з кнопок і якщо така кнопка знайдена – викликається метод onMouseInput якщо не це є режимна кнопка (як, наприклад завантаження моделі), або onEnter для режимної кнопки якщо ми тільки що активували її, або onExit для режимної кнопки при деактивації.

```
	if (_activeModeOperators)
		_activeModeOperators->onMouseInput(view, button, action, mods, x, y);
}
```

Якщо жодна кнопка не була натиснута, то перевіряється чи є зараз деякий активний оператор, якому буде доручено відповідати на дану подію таким чином, як цей оператор перевизначив метод onMouseInput.

3.4 Імплементація видалення трикутників

Для реалізації основної задачі додатка – створення отворів, розроблено оператор DeleteFaceOperator. Це режимний оператор, тобто він має момент активації, що відбувається при натисканні на кнопку, і деактивації – при повторному натисканні. Весь час поки оператор активний, він приймає на себе відповідальність за обробку подій від мишки та клавіатури.

Продемонструємо як він перевизначає методи батьківського класу.

```
void DeleteFaceOperator::onEnter(View& view)
{
	ShowEdgesOperator seo;
	seo.onEnter(view);
}

void DeleteFaceOperator::onExit(View& view)
{
	ShowEdgesOperator seo;
	seo.onExit(view);
}
```

Бачимо що при активації вмикається відображення ребер трикутників, а при деактивації вимикається. Оскільки ми вже маємо такий функціонал, тож щоб не переписувати код ми просто використовуємо той самий оператор, що і кнопка «Відображенні ребер».

Ми маємо реагувати на переміщення курсору, щоб підсвічувати трикутник на якому цієї миті знаходиться курсор.

```
void DeleteFaceOperator::onMouseMove(View& view, double x, double y)
{
	if (hovered.node && hovered.face.index != -1)
		hovered.node->getMesh()->DehighLightTriangle(hovered.face.index);
```

По-перше, ми скасовуємо підсвічування трикутника на якому раніше знаходився курсор.

```
	std::vector<Contact> contacts = view.raycast(x, y, false);
```

Потім ми запускаємо промінь з позиції курсора вглиб сцени і запам’ятовуємо з ким контактує цей промінь. Це відбувається в методі raycast, це великий та складний метод, тому тут лише відмітимо що для покращення швидкості цього методу я застосувала в ньому структуру Octree,

яка дозволяє пришвидшити пошук контактів з моделлю шляхом її сегментації на кубічні зони. Отже, contacts це певні трикутники певних моделей через які проходить промінь курсора.

```
        if (!contacts.empty() && contacts.front().node && contacts.front().face.index!=-1)
        {
                hovered = contacts.front();
                hovered.node->getMesh()->HighLightTriangle(hovered.face.index);
        }
}
```

Ми обираємо найближчий та записуємо його в поле hovered, а також підсвічуємо знайдений трикутник.

Але поле hovered потрібно не тільки для підсвічування. Коли користувач обере трикутник який бажає видалити, ми видалимо трикутник, який був в той момент записаний в поле hovered. При кліку в режимі видалення трикутників відбувається наступні дії:

```
void DeleteFaceOperator::onMouseInput(View& view, ButtonCode button, Action action,
Modifier mods, double x, double y)
{
        if (action == Action::Press)
        {
                pressX = x;
                pressY = y;
        }
```

Запам’ятовуємо я якій позиції було зроблено натискання.

```
        if (action == Action::Release)
        {
                if (fabs(pressX - x) > 0.001 || fabs(pressY - y) > 0.001)
                        return;
```

Якщо в той момент як користувач відпускає клавішу миші курсор вже знаходиться в іншому положенні, то така дія не вважається цілеспрямованим натисканням на трикутник, це вважається переміщенням положення камери.

```
                if (hovered.node && && hovered.node->getMesh())
                        hovered.node->getMesh()->deleteFace(hovered.face);
        }
}
```

Якщо позиція при натисканні та відпусканні співпали, то ми звертаємось до моделі якій належить трикутник та просимо її видалити трикутник за заданим індексом.

3.5 Імплементація розрізання моделі площиною

Розрізання моделі площиною також імплементовано за допомогою оператора. Створено клас SplitOperator та перевизначено його методи.

```
void SplitOperator::onEnter(View& view)
{
	float planeSize = view.getViewport().getCamera().distanceFromEyeToTarget();
	auto meshptr = std::make_unique<Mesh>(createBox(planeSize, planeSize, 0.001));
	Material mat;
	mat.amb = glm::vec3(0.2, 0.2, 1);
	meshptr->setMaterial(mat);
	std::unique_ptr<Node> planeNode = std::make_unique<Node>();
	planeNode->attachMesh(std::move(meshptr));
	planeNode->setName(planeName);
	mPlaneNode = planeNode.get();
	view.getModel()->attachNode(std::move(planeNode));
}
```

При активації створюється новий об’єкт на сцені – площина. Вона служить для візуалізації перерізу. Початкове положення співпадає з площиною xOy. Далі задається синій колір матеріалу площині, створюється вузол (Node) – структура яка зберігає трансформацію моделі, і площина додається до сцени.

Розглянемо реалізацію методу onMouseMove:

```
void SplitOperator::onMouseMove(View& view, double x, double y)
{
	double w = viewport.getWidth();
	double h = viewport.getHeight();
	float distanceFromEyeToTarget = camera.distanceFromEyeToTarget();
	auto viewMatrix = camera.calcViewMatrix();

	if (rightButtonCondition == Action::Press)
	{
		double u = ((x - x0) / w) * distanceFromEyeToTarget;
		double v = ((y0 - y) / h) * distanceFromEyeToTarget;
		mPlaneNode->applyRelativeTransform(
			glm::translate(glm::mat4(1.0f), glm::vec3(u, v, 0)));
		x0 = x;
		y0 = y;
	}
```

Якщо затиснута права клавіша, ми рахуємо як змістився курсор в порівнянні з останніми спостереженнями. Зміщення по x записується в змінну u, а по y – в змінну v. До площини застосовується трансформація переміщення по x на значення u, а по y – на значення v.

```
else if (leftButtonCondition == Action::Press)
{
	glm::vec3 pos1{ (2.0 * x0 - w) / h, -(2.0 * y0 - h) / h, 0.0 };
	glm::vec3 pos2{ (2.0 * x - w) / h, -(2.0 * y - h) / h, 0.0 };
	glm::mat4 initialTransform = mPlaneNode->getRelativeTransform();
	glm::vec3 translatePart = initialTransform[3];
	glm::vec3 origin = viewMatrix * glm::vec4(translatePart, 1.f);
	glm::vec3 a = pos1 - origin;
	glm::vec3 b = pos2 - origin;
```

Якщо затиснута ліва клавіша мишки, то відбувається обертання площини. Для здійснення обертання нам потрібно знати кут обертання та вектор навколо якого здійснюється обертання. Для цього ми маємо якось описати положення курсора в момент попереднього спостереження і теперішнього, щоб потім знайти яке обертання було здійснено, щоб попереднє положення курсора (pos1) опинилось в теперішньому (pos2). Обертання здійснюється навколо центрі площини. Якщо не було застосовано переміщення, то центр залишається в позиції (0, 0, 0), а інакше треба подивитись на матрицю трансформації і взяти звідти ту частину яка відповідає за переміщення – це і є актуальний центр. Запишемо в змінні a та b вектори з центра площини до позицій курсора pos1 і pos2 відповідно.

```
if (length(a) >= 1.0)	a = normalize(a);
else		a.z = pow(1.0f - pow(a.x, 2.0f) - pow(a.y, 2.0f), 0.5f);
if (length(b) >= 1.0)	b = normalize(b);
else		b.z = pow(1.0f - pow(b.x, 2.0f) - pow(b.y, 2.0f), 0.5f);
```

Якщо довжина векторів a та b >= 1, відбувається їх нормалізація, а інакше – координати x та y не змінюються, а координата z набуває такого значення, щоб довжина вектора була рівною 1 за формулою 3.1.

$$\sqrt{x^2 + y^2 + z^2} = 1 \quad \rightarrow \quad z = \sqrt{1 - x^2 + y^2} \qquad (3.1)$$

```
float angle = acos(dot(a, b));
glm::vec3 axis =cross(a, b);
```

Маючи 2 одиничні вектори a та b, можемо знайти кут між ними, що дорівнює арккосинусу значенню скалярного добутку – dot(a, b). Векторний добуток векторів – cross(a, b) – це вектор, який направлений перпендикулярно спільній площині векторів a та b, тобто площині в якій здійснено обертання, отже axis – це вісь обертання.

```
            glm::mat3 toWorldCameraSpace = glm::inverse(glm::mat3(viewMatrix));
            glm::mat4 rotationMatrix = glm::rotate(angle, axis);
            plane->applyRelativeTransform(glm::translate(-translatePart));
            plane->applyRelativeTransform(rotationMatrix);
            plane->applyRelativeTransform(glm::translate(translatePart));
            x0 = x;
            y0 = y;
        }
}
```

Переводимо координати вісі обертання з екранного простору в координатний простір сцени. Щоб обертання було навколо центру площини, а не центру сцени, спочатку переносимо площину в центр сцени, застосовуємо обертання навколо осі axis на кут angle, а потім повертаємо на попереднє положення.

Залишилося розглянути що відбувається після того, як користувач обрав бажане положення площини і повторно натиснув кнопку для завершення операції. На завершальному етапі викликається метод onExit:

```
void SplitOperator::onExit(View& view)
{
        glm::mat4 transformation = mPlaneNode->getRelativeTransform();
        view.getModel()->deleteNode(mPlaneNode);
        mPlaneNode = nullptr;
```

Запам’ятовуємо трансформацію яка була застосована до площини і видаляємо площину зі сцени.

```
        glm::vec3 origin = transformation * glm::vec4(0,0,0,1);
        glm::vec3 normal = transformation * glm::vec4(0,0,1,0);
```

Щоб параметрично описати площину нам достатньо знати її нормаль та точку на цій площині. Оскільки спочатку площина проходила через точку (0,0,0) і мала нормаль рівну (0,0,1), то помноживши ці координати на матрицю трансформації, дізнаємось де знаходиться точка на площині та актуальний напрямок нормалі.

```
        Splitter splitter(origin, normal);
```

Створюємо екземпляр класу Splitter, задаючи йому параметри площини.

```
        for (auto& mesh : view.getModel()->GetMeshes())
            for (size_t faceIndex = 0; faceIndex < mesh.facesCount; faceIndex++)
                splitter.processTriangle(mesh, faceIndex);
```

Для кожного трикутнику кожного об’єкту на сцені викликаємо метод processTriangle. Цей метод аналізує з якої сторони від площини знаходиться трикутник та зберігає його в відповідному масиві об’єкта splitter. Якщо трикутник перетинається площиною, то він буре розрізаний, а його частини будуть додані у відповідні масиви.

```
	auto& group1 = splitter.getTrianglesGrop1();
	auto& group2 = splitter.getTrianglesGrop2();
```

Після завершення аналізу всіх трикутників ми забираємо результат. В змінних group1 записані трикутники що знаходяться з тієї сторони площини куди вказує нормаль, а в group2 – з іншої.

```
	auto outFilePath1 = AddToName(model->getName(), "_part1");
	auto outFilePath2 = AddToName(model->getName(), "_part2");
	CreateSplitWindow(group1, outFilePath1, windowWidth, windowHeight);
	CreateSplitWindow(group2, outFilePath2, windowWidth, windowHeight);
}
```

Розрізання відбулось, залишилось тільки записати результат в 2 файли та створити 2 нових вікна для відображення результату. Це відбувається в функції CreateSplitWindow.

```
void CreateSplitWindow(const std::vector<std::array<glm::vec3, 3>>& triangles, const
std::string& fileName, uint32_t width, uint32_t height)
{
	HalfEdge halfEdgeDS;
	for (auto& triangle : triangles) {
		std::vector<halfEdgeElements::VertexHandle> faceVertices;
		faceVertices.push_back(halfEdgeDS.addUniqueVertex(triangle[0]));
		faceVertices.push_back(halfEdgeDS.addUniqueVertex(triangle[1]));
		faceVertices.push_back(halfEdgeDS.addUniqueVertex(triangle[2]));
		halfEdgeDS.addFace(faceVertices);
	}
	halfEdgeDS.connectTwins();
```

Створюється нова структура HalfEdge та заповнюється трикутниками. Метод connectTwins завершає етап створення структури, знаходить та запам'ятовує зв'язки між сусідніми трикутниками.

```
	View* newView = Application::singleton->createView(width, height, halfEdgeDS);
	SaveFileOperator sfop;
	sfop.onKeyboardInput(newView, KeyCode::S, Action::Release, Modifier::Control);
```

Створюється нове вікно, а модель зберігається в файл.

ВИСНОВКИ

В результаті роботи було створено інформаційне та програмне забезпечення, що може створювати отвори в3d моделях двома шляхами: завдяки видаленню трикутників з поверхні моделі або розрізання моделі площиною на дві половини.

Застосунок взміє завантажувати та зберігати файли найпопулярніших форматів з 3D моделями – STL і COLLADA. Інформація про геометрію моделі представлено у вигляді полігональої сітки, що описує поверхню моделі як набір координат вершин та трикутників, що їх поєднує. Зазвичай такі моделі мають цілісну поверхню, тому моделі з отворами дуже складно знайти в інтернеті, що підтверджує потребу в інструменті для створення отворів в полігональних моделях. Проведений огляд показав. що жоден з наявних продуктів не забесзпечує досягнення мети у повному обсязі. САПР які можуть створювати отвори в трикутній сітці - це Blender, 3ds Max, PowerShape та Meshlab, але користування даними продуктами вимагає певних знань та навичок роботи в САПР.

Інформаційне та програмне забезпечення, що було розробленов ході роботи, має простий та зручний інтерфейс для комфортного та швидкого створення отворів, сконцентрованого на інтерактивній взаємодії з користувачем. Було розроблено алгоритми видалення трикутників з полігональної сітки та знаходження порушення цілісності поверхні моделі з використанням структури HalfEdge. Розроблено алгоритм розрізання моделі площиною з інтерактивною маніпуляцією положенням та нахилом площини розрізання, керування реакцією на події маніпуляторів.

СПИСОК ЛІТЕРАТУРИ

1. Jean-Philippe Pernot, Georges Moraru, Philippe Veron. «Filling holes in meshes using a mechanical model to simulate the curvature variation minimization.»
2. Emiliano Perez, Santiago Salamanca, Pilar Merchan, Antonio Adan. «A comparison of hole–filling methods in 3d.»
3. 3D моделирование. URL: https://hmong.ru/wiki/3D_modelling (дата звернення: 11.04.2022).
4. Strutynska, Oksana. (2018). Modern state and perspectives of the development of the 3D modeling and 3D printing technologies.
5. 8 самых распространенных форматов 3D файлов. URL: https://3dprintstory.org/8-samih-rasprostranennih-formatov-3d-failov (дата звернення: 11.04.2022).
6. Triangle mesh. URL: https://en.wikipedia.org/wiki/Triangle_mesh (дата звернення: 11.04.2022).
7. Скульптура Тараса Шевченко. 3D модель. URL: https://free3d.com/ru/3d-model/sculpture-taras-shevchenko-2199.html (дата звернення: 8.06.2022).
8. Schroeder, William & Zarge, Jonathan & Lorensen, William. (1997). Decimation of triangle meshes. SIGGRAPH Comput. Graph.. 26. 65-70. 10.1145/133994.134010.
9. Guo, Xiaoyuan & Xiao, Jun & Wang, Ying. (2018). A survey on algorithms of hole filling in 3D surface reconstruction. The Visual Computer. 34. 10.1007/s00371-016-1316-y.
10. Где скачать 3D модели БЕСПЛАТНО? ТОП 15 сайтов. URL: https://youtu.be/tM5heD86rG0 (дата звернення: 01.06.2022).
11. CAD: all about computer-aided design software. URL: https://www.hwlibre.com/en/cad-software/ (дата звернення: 02.06.2022).

12. Blender official web-site. URL: https://www.blender.org/ (дата звернення: 03.06.2022).
13. Deleting, Dissolving And Collapsing In edit Mode. Blender Tutorial. URL: https://youtu.be/qcWY9NgMoxM (дата звернення: 03.06.2022).
14. Autodesk 3ds MAX. URL:https://uk.wikipedia.org/wiki/Autodesk_3ds_MAX (дата звернення: 03.06.2022).
15. Buy 3ds Max. URL: https://www.autodesk.com/products/3ds-max/overview?term=1-MONTH&tab=subscription (дата звернення: 03.06.2022).
16. 3ds Max Tutorial: Edit Poly, Edges. URL: https://youtu.be/DTId0GxseG8 (дата звернення: 03.06.2022).
17. PowerShape. URL: https://www.autodesk.eu/products/powershape/overview (дата звернення: 03.06.2022).
18. MeshLab. URL: https://www.meshlab.net/ (дата звернення: 03.06.2022).
19. Multiple buffering. URL: https://en.wikipedia.org/wiki/Multiple_buffering (дата звернення: 03.06.2022).