МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА ТЕХНОЛОГІЙ
Кафедра комп'ютерних систем та мереж

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач випускової кафедри
_____Ігор ЖУКОВ
«___»_____2023 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ

«МАГІСТР»

ЗА СПЕЦІАЛЬНІСТЮ 123 «КОМП'ЮТЕРНА ІНЖЕНЕРІЯ»

Тема: «Кросплатформний програмний додаток для роботи із *3D*-даними»

Виконавець: Ілля БУДАРОВ

Керівник: Сергій ГІЛЬГУРТ

Нормоконтролер: Василь МАЛЯРЧУК

Засвідчую, що у дипломній роботі
немає запозичень праць інших авторів
без відповідних посилань
студент _____Ілля БУДАРОВ

Київ 2023

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет комп'ютерних наук та технологій

Кафедра комп'ютерних систем та мереж

Спеціальність 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Комп'ютерні системи та мережі»

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерних систем
_____Ігор ЖУКОВ
«___»_____2023 р.

ЗАВДАННЯ

на виконання кваліфікаційної роботи

Бударова Іллі Ігоровича

1. Тема роботи: «Кросплатформний програмний додаток для роботи із *3D*-даними». затверджена наказом ректора від «29» серпня 2023 року № 1521/ст.

2. Термін виконання роботи: з 2 жовтня 2023 р. по 31 грудня 2023 р.

3. Вхідні дані до роботи: технічна документація для створення кросплатформних графічних додатків, інформація про специфіку *3D*-моделювання, додатки для візуалізації *3D*-форматів файлів.

4. Зміст пояснювальної записки: вступ, огляд сучасних методів та інструментів для представлення та роботи із *3D*-моделями, опис стеку технологій для розробки додатку, архітектура програмного додатку, візуалізація та обробка *3D*-моделей, аналіз отриманих результатів, висновки.

5. Перелік обов'язкового графічного (ілюстрованого) матеріалу:

– діаграма компонентів;

– діаграма прецедентів;

– блок-схема;

– мануальне тестування.

6. Календарний план-графік

№ пор.	Завдання	Термін виконання	Відмітка про виконання
1	Аналіз сучасних відомостей про *3D*-моделювання.	02.10.2023 – 05.10.2023	
2	Аналіз методів *3D*-моделювання та обрання найефективнішого.	05.10.2023 – 10.10.2023	
3	Аналіз стеку технологій для програмного додатку.	10.11.2023 – 15.11.2023	
4	Розробка архітектури та інтерфейсу програмного додатку.	16.11.2023 – 25.11.2023	
5	Розробка методів обробки та візуалізації *3D*-об’єктів.	25.11.2023 – 04.12.2023	
6	Проведення мануального тестування.	04.12.2023 – 05.12.2023	
7	Оформлення пояснювальної записки.	05.12.2023 – 19.12.2023	
8	Розробка презентації для захисту роботи та підготовка до захисту.	19.12.2023 – 24.12.2023	

7. Дата видачі завдання: « 02 » жовтня 2023 р.

Керівник дипломного дослідження _____ Сергій ГІЛЬГУРТ

Завдання прийняв до виконання _____ Ілля БУДАРОВ

РЕФЕРАТ

Пояснювальна записка до дипломного дослідження «Кросплатформний програмний додаток для роботи із 3*D*-даними»: 96 сторінок, 40 рисунків, 1 таблиця, 26 використаних джерел.

КРОСПЛАТФОРМНИЙ ПРОГРАМНИЙ ДОДАТОК, *OBJ, C++, QT, CGAL, OPENGL, CMAKE, 3D*-МОДЕЛЮВАННЯ.

Мета дослідження – покращення та розширення можливостей кросплатформного програмного додатку для роботи із *3D*-даними, з метою сприяння розвитку і вдосконаленню технологій обробки та візуалізації цих даних у різних галузях; забезпечення широкого спектру функціональних можливостей для роботи з 3*D*-моделями (імпорт, маніпуляції, аналіз та відображення) на різних операційних системах, таких як *Windows, MacOS, Linux*.

Об'єкт дослідження – процеси завантаження, аналізу, обробки та візуалізації *3D*-даних.

Предмет дослідження – програмні засоби завантаження, аналізу, обробки та візуалізації *3D*-даних.

Методи дослідження – літературний аналіз, аналіз технічних вимог, проєктування архітектури, оптимізація, тестування, інноваційні покращення.

Теоретична і практична значимість – результати можуть служити підґрунтям для подальшого розвитку наукової галузі, обміну досвідом у науковій спільноті та покращення процесів обробки та візуалізації *3D*-даних у промисловості. Крім того, вони мають потенціал вплинути на вищу освіту та сприяти створенню відкритого програмного забезпечення для доступу до ефективних інструментів без ліцензійних обмежень.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

CGAL (The Computational Geometry Algorithms Library) – бібліотека алгоритмів обчислювальної геометрії

MVC (*Model View Controller*) – архітектурний шаблон, що використовується для досягнення ефективної та розширюваної архітектури

OBJ – формат даних, що представляє *3D*-геометрію

API (Application Programming Interface) – набір визначень підпрограм, протоколів взаємодії та засобів для створення програмного забезпечення

OpenGL – специфікація, що визначає кросплатформний *API* для написання застосунків, що використовують *2D* та *3D*-комп'ютерну графіку

UV-mapping – процес створення відповідності між точками на поверхні *3D*-моделі та їхніми текстурними координатами

NURBS (*Non-Uniform Rational B-Spline*) – математичне представлення тривимірних об'єктів

3D-mesh – сукупність вершин, ребер і граней, які разом утворюють тривимірний об'єкт

UML (*Unified Modeling Language*) – мова моделювання, яка використовується для візуалізації та опису архітектури системи

ВСТУП

У зв'язку з стрімким розвитком сучасних технологій та нестримним наростанням обсягів інформації, усвідомлення просторових відносин та взаємодій об'єктів стає важливим завданням для багатьох галузей науки та промисловості. У цьому контексті *3D*-дані та *3D*-моделювання виступають не лише як інструменти візуалізації, але й як ключові компоненти для аналізу складних структур і процесів у тривимірному просторі.

Моделювання у тривимірному просторі пройшло великий шлях з часів свого скромного виникнення понад 40 років тому, при цьому його можливості постійно розвивалися та ставали все кращими поруч із технологіями, які його використовують. Завдяки цьому відбувається багато цікавих подій. Людство на порозі революції в області моделювання, де сучасні технології та ті, що знаходяться в розробці, здається, стануть кардинальними змінниками у світі тривимірного моделювання. Наприклад, поява тривимірного друку ще більше розширила можливості та потенційні застосування тривимірного моделювання. Тим часом віртуальна реальність, хоч і дуже нова у своїй поточній версії, здається, змінить спосіб, яким ми взаємодіємо з тривимірними моделями назавжди.

За останні роки спостерігається зростання інтересу до роботи з тривимірними даними в різних галузях, включаючи архітектуру, інженерію, медицину, виробництво та багато інших. Завдяки розвитку комп'ютерної графіки та обробки даних, технології 3*D*-моделювання стають все більш потужними та доступними. Однак, робота з тривимірними даними вимагає спеціалізованих програмних рішень, які дозволяють зручно та ефективно працювати з цими об'єктами. Крім того, зростає потреба у кросплатформних рішеннях, що працюють на різних операційних системах і надають користувачам зручний та інтуїтивно зрозумілий інтерфейс.

Зростаючий обсяг тривимірних даних в сучасному світі породжує нові виклики та можливості для вивчення та використання цих інформаційних ресурсів. Широкий спектр застосувань тривимірних даних в різних галузях суспільства вказує на

необхідність подальших досліджень та розвитку технологій для оптимального використання цих ресурсів.

Однією з ключових проблем у роботі з тривимірними даними є необхідність уніфікації та стандартизації процесів обробки цих даних. Різноманітність форматів та структур тривимірних об'єктів ускладнює роботу фахівців та унеможливлює обмін даними між різними платформами та програмними продуктами. У зв'язку з цим, однією з ключових мет цього дослідження є розробка універсальних рішень, які сприятимуть інтеграції та обробці тривимірних даних незалежно від їхнього формату.

Крім того, інтенсивний розвиток технологій штучного інтелекту та машинного навчання відкриває нові перспективи для розвитку інноваційних методів обробки тривимірних даних. Використання алгоритмів глибокого навчання може значно полегшити завдання аналізу та інтерпретації тривимірних об'єктів, що стає актуальним у контексті зростаючої складності цих даних.

Більшість людей мають виражену тенденцію асоціювати тривимірне моделювання лише з архітектурним дизайном. Це не зовсім невірно, але тривимірне моделювання може бути використано в практично будь-якій галузі, включаючи виробництво товарів та навіть навчання, наприклад, у школах. Отримання візуального представлення об'єкта – це лише одна з переваг. Окрім отримання точного зображення об'єкта, тривимірне моделювання (принаймні з інженерного погляду) надає велику кількість технічних деталей з майже нульовою кількістю помилок, якщо взагалі є які-небудь.

Одним із ключових напрямів є також розширення застосувань тривимірного моделювання в медичній сфері. Завдяки високій точності та деталізації тривимірних моделей, їх використання в медичних дослідженнях, діагностиці та плануванні лікування стає надзвичайно важливим. Це може об'єднати передові методи візуалізації з високоточними медичними даними для досягнення оптимальних результатів. Наприклад, у сучасній хірургії тривимірні моделі можуть використовуватися для точного планування операцій, а також для виготовлення реалістичних моделей органів пацієнтів. Це дозволяє лікарям отримати більше

інформації перед самою операцією та забезпечує вищу точність виконання медичних втручань.

3D-моделювання стало не лише важливим інструментом в ряді промислових галузей, а й відкрило нові можливості для використання у сферах, які раніше не знаходилися під його прямим впливом. Однією з таких є використання *3D*-моделей у виробництві, де цей інструмент допомагає покращити виробничі процеси та зменшити витрати завдяки можливості виробляти деталі та прототипи безпосередньо з використанням *3D*-друку.

Однією з найактуальніших проблем у галузі *3D*-моделювання є пошук ефективних та універсальних рішень для управління та обробки різноманітних форматів тривимірних даних. Стандартизація процесів обробки даних стає кроком вперед у полегшенні спільної роботи фахівців та підвищенні ефективності використання *3D*-моделей.

Мета дослідження – покращення та розширення можливостей кросплатформного програмного додатку для роботи із *3D*-даними, з метою сприяння розвитку і вдосконаленню технологій обробки та візуалізації цих даних у різних галузях; забезпечення широкого спектру функціональних можливостей для роботи з 3*D*-моделями (імпорт, маніпуляції, аналіз та відображення) на різних операційних системах, таких як *Windows, MacOS, Linux*.

Об’єкт дослідження – процеси завантаження, аналізу, обробки та візуалізації *3D*-даних.

Предмет дослідження – програмні засоби завантаження, аналізу, обробки та візуалізації *3D*-даних.

Методи дослідження – літературний аналіз, аналіз технічних вимог, проєктування архітектури, оптимізація, тестування, інноваційні покращення.

Наукова новизна отриманих результатів – розроблено унікальних механізм оптимізованого завантаження, аналізу, обробки та візуалізації *3D*-даних на різних операційних системах; впроваджено ефективний інструментарій для роботи із форматом *OBJ,* враховуючи його специфіку та динамічні вимоги сучасних застосувань; внесено новаторські покращення в сфері візуалізації *3D*-даних, що

передбачає використання технології *OpenGL* для створення реалістичних зображень; реалізовано інтерактивний інтерфейс з можливістю маніпуляцій та взаємодії з об’єктами в *3D*-просторі, що дозволяє користувачам отримати більше інформації та контролю над представленими даними.

Наукова новизна розробленого додатку не обмежується лише поточним етапом дослідження, але також визначає напрямки подальших досліджень та розвитку програмного забезпечення. В майбутньому планується:

– розробка механізмів для інтеграції з іншими популярними форматами *3D*-даних, що розширить функціональні можливості додатку та забезпечить більший спектр сумісності з різноманітними джерелами даних;

– подальша оптимізація алгоритмів для обробки та аналізу великих обсягів *3D*-даних, зокрема застосування паралельних обчислень та технік штучного інтелекту для автоматизації певних завдань;

– запровадження інтерактивних інструментів для більш ефективної маніпуляції та аналізу *3D*-моделей, таких як системи динамічного підсвічування ключових елементів, анімація та покращені інструменти для взаємодії з об'єктами;

– використання методів машинного навчання для автоматичного розпізнавання та класифікації об'єктів у *3D*-просторі, що розширить можливості автоматизації обробки даних;

– впровадження опцій для виконання булевих операцій, таких як об'єднання (*Union*), перетин (*Intersection*), різниця (*Difference*) між об'єктами;

– розробка механізму, який автоматично визначатиме оптимальну структуру топології для кожного об'єкта на сцені, враховуючи його форму, розмір та деталі.

Практичне значення отриманих результатів – результати можуть бути підґрунтям для подальшого розвитку наукової галузі, обміну досвідом у науковій спільноті та покращення процесів обробки та візуалізації *3D*-даних у промисловості. Крім того, вони мають потенціал вплинути на вищу освіту, архітектуру для віртуального проєктування будівель, медицині для аналізу медичних зображень чи в інженерії для створення та модифікації складних об'єктів. Також на основі отриманих знань можна безкоштовно спроєктувати архітектуру, розробити користувацький

інтерфейс, а також побудувати програмну реалізацію додатку для роботи із *3D*-даними з відкритим кодом та без ліцензійних обмежень, що дозволить ефективно модифікувати та оптимізувати алгоритми обробки та візуалізації *3D*-даних. Отримані результати можуть знайти застосування в різних галузях, де важлива робота з *3D*-даними.

Обрано *C++* та *Qt* як основний стек розробки через їхню широку популярність, високу ефективність та кросплатформність. *C++* є потужною мовою програмування, яка надає високий рівень контролю над ресурсами та є ефективною для великих та складних проєктів. *Qt*, у свою чергу, є потужною бібліотекою, яка дозволяє розробляти кросплатформні застосунки з графічним інтерфейсом та включає інструменти для маніпулювання тривимірними об'єктами.

CGAL вибрано для обробки та аналізу геометричних об'єктів у тривимірному просторі. *CGAL* надає великий набір алгоритмів для обробки геометричних структур, що робить його ідеальним інструментом для роботи з тривимірними даними.

CMake вибрано для налаштування та збірки проєкту, оскільки це потужний інструмент для автоматизації процесу збірки, тестування та розгортання кросплатформних додатків.

OpenGL обрано як бібліотеку для візуалізації тривимірних даних. Вона надає високопродуктивний доступ до графічних можливостей комп'ютера та дозволяє створювати реалістичні зображення.

Vcpkg вибрано для управління залежностями проєкту та для зручного встановлення та оновлення бібліотек.

Всі ці компоненти утворюють добре зіграну команду технологій, яка забезпечить оптимальну продуктивність та гнучкість для розробки програмного додатку для роботи із *3D*-даними.

РОЗДІЛ 1
ОГЛЯД СУЧАСНИХ МЕТОДІВ ТА ІНСТРУМЕНТІВ ДЛЯ ПРЕДСТАВЛЕННЯ ТА РОБОТИ ІЗ 3*D*-МОДЕЛЯМИ

У тривимірній комп'ютерній графіці *3D*-моделювання – це процес розробки математичного координатного представлення будь-якої поверхні об'єкта в трьох вимірах за допомогою спеціалізованого програмного забезпечення шляхом маніпулювання ребрами, вершинами та полігонами в модельованому тривимірному просторі (рис. 1.1). *3D*-моделі представляють фізичне тіло за допомогою набору точок в тривимірному просторі, які з'єднані різними геометричними об'єктами, такими як трикутники, лінії, криві поверхні і так далі. Можна уявляти процес *3D*-моделювання як реалістичне малювання, але з значною складністю, оскільки можна включати в модель реальний час і взаємодіяти з об'єктом за допомогою комп'ютера. Так само, як фізична модель, об'єкт може обертатися, відбиватися, розриватися чи маніпулюватися різними способами на екрані.

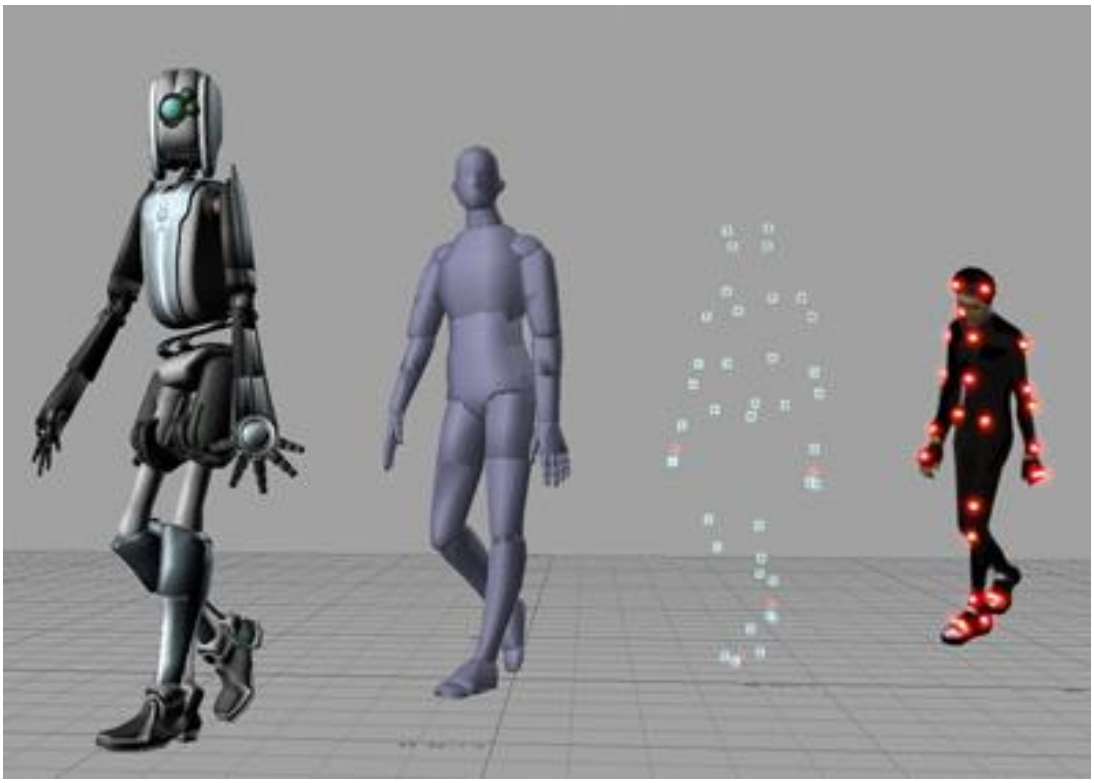


Рис. 1.1. Тривимірне моделювання

Залежно від використаного програмного забезпечення та технічних характеристик комп'ютера, тривимірні моделі можна створити за допомогою кількох різних процедур, таких як моделювання на основі алгоритмів, ручне малювання (на комп'ютері) чи сканування. Мапування текстур допоможе створити більш деталізовані деталі на поверхнях моделей. Щоб зробити кожну модель якомога більш реалістичною та взаємодійною, вона повинна базуватися на наборі точок даних. Майже всі *3D*-моделі можна розділити на дві категорії:

– тверді, що визначають об'єм об'єкта, який вони представляють (наприклад, камінь). Тверді моделі використовуються в основному для інженерних та медичних симуляцій і зазвичай створюються з допомогою конструктивної геометрії об'ємів;

– оболонка або границя, що представляють поверхню об'єкта, а не його об'єм, (наприклад, безкінечно тонка шкаралупа яйця) та використовуються в іграх та кіно.

Тверді та оболонкові моделі можуть створювати функціонально ідентичні об'єкти. Різниця між ними в основному полягає у способі їх створення та редагування, у конвенціях використання в різних галузях та різниці в типах наближень між моделлю та реальністю.

Оболонкові моделі повинні бути замкнутими (без отворів або тріщин у оболонці), щоб мати сенс як реальний об'єкт. У оболонковій моделі куба нижня і верхня поверхні куба повинні мати однакову товщину без отворів або тріщин у першому і останньому надрукованому шарі. Багатокутні мережі (*polygonal meshes*) є найпоширенішим представленням зображення. Набори рівнів є корисним представленням для змінюваних поверхонь, які зазнають багато топологічних змін, таких як рухи рідини [1].

Процес перетворення представлень об'єктів, таких як координата середньої точки сфери та точка на її обхваті, у полігональне представлення сфери називається тесселяцією (*tessellation*). Цей крок використовується в основі заснованому на полігональному рендерингу, де об'єкти розбиваються на абстрактні представлення («примітиви»), такі як сфери, конуси, на так звані сітки, які є мережами взаємопов'язаних трикутників. Сітки з трикутників є популярними, оскільки вони виявилися легкими для растеризації (поверхня, описана кожним трикутником, є

плоскою, тому проекція завжди є опуклою). Полігональне представлення не використовуються в усіх техніках рендерингу, і в таких випадках крок тесселяції не включається у перехід від абстрактного представлення до відтвореної сцени.

Стадія моделювання полягає у формуванні окремих об'єктів, які потім використовуються у сцені. Існує кілька технік моделювання, включаючи:

– конструктивна геометрія тіл (*constructive solid geometry*) використовує логічні операції (об'єднання, віднімання, перетин) для створення складних об'єктів шляхом комбінування простіших геометричних форм;

– неявні поверхні (*implicit surfaces*) використовують математичні рівняння для визначення поверхні об'єкта. Це дозволяє створювати плавні і складні форми, які може бути важко створити з використанням інших методів;

– поверхні поділу (*subdivision surfaces*) використовуються для створення деталізованих поверхонь, розділяючи кожен полігон на більшу кількість менших полігонів. Це дозволяє отримати гладку поверхню з дрібних деталей.

1.1. Огляд найпопулярніших способів представлення *3D*-моделей

У зв'язку з постійним розвитком технологій та комп'ютерної графіки в сучасному інформаційному суспільстві виникає об'єктивна потреба в систематичному аналізі та огляді різноманітних методів представлення тривимірних об'єктів. Спричинені цифровою революцією у сфері дизайну, інженерії, візуальних мистецтв та інших сферах, *3D*-моделі стають важливим інструментом для концептуалізації, аналізу та візуалізації об'єктів та просторових структур. Далі буде розглянуто та проаналізовано ключові стратегії представлення *3D*-моделей.

1.1.1. Полігональне моделювання

У комп'ютерній *3D*-графіці полігональне моделювання (*polygonal modeling*) є підходом до створення моделей об'єктів шляхом представлення або апроксимації їх поверхонь за допомогою полігонів. Полігональне моделювання добре підходить для відображення за допомогою скануючих растрових алгоритмів і, отже, є методом вибору для комп'ютерної графіки в реальному часі.

Основним об'єктом, який використовується в полігональному моделюванні, є вершина – точка в тривимірному просторі. Дві з'єднані прямою лінією вершини утворюють ребро. Три вершини, з'єднані між собою трьома ребрами, визначають трикутник, який є найпростішим полігоном у Евклідовому просторі. Більш складні полігони можуть бути створені з кількох трикутників або як один об'єкт з більш ніж 3 вершинами. Чотирикутники (зазвичай називаються квадратами) та трикутники є найпоширенішими формами, які використовуються в полігональному моделюванні. Група полігонів, які з'єднані між собою спільними вершинами, загалом називається елементом. Кожен полігон, що складає елемент, називається гранню. Далі можна побачити приклад полігонального моделювання (рис. 1.2).

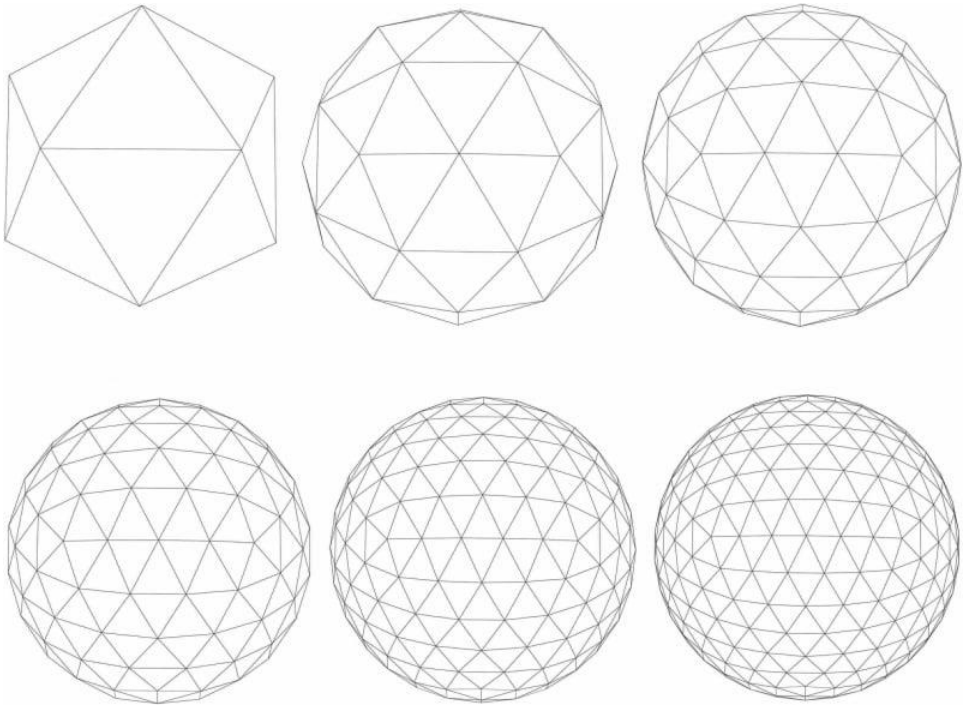


Рис. 1.2. Полігональне моделювання

У Евклідовій геометрії будь-які три неколінеарні точки визначають площину. З цієї причини трикутники завжди знаходяться в одній площині. Однак це не обов'язково вірно для більш складних полігонів. Плоскість трикутників дозволяє легко визначити їх нормалі, яка є тривимірним вектором, перпендикулярним до поверхні трикутника. Нормалі поверхні корисні для визначення передачі світла при трасуванні променів і є ключовим компонентом популярної моделі затінення Фонга. Деякі системи рендерингу використовують нормалі вершин замість нормалей

поверхонь, щоб створити кращу систему освітлення за рахунок більшої обробки. Слід зауважити, що кожний трикутник має дві нормалі поверхні, які вказують в протилежних напрямках одна відносно однієї. У багатьох системах вважається валідною лише одна з цих нормалей – інша сторона полігона називається зворотною гранню і може бути видимою або невидимою в залежності від бажань програміста.

Багато програм моделювання не строго дотримуються геометричної теорії; наприклад, можливо, що дві вершини мають два різних з'єднання між ними, які знаходяться в точно одній просторовій позиції. Також можливо, що дві вершини знаходяться в одних і тих самих координатах або дві грані знаходяться в одній позиції. Такі ситуації зазвичай не бажані, і багато програм підтримують функцію автоматичного очищення. Однак, якщо функція автоматичного очищення відсутня, їх потрібно видаляти вручну.

Група полігонів, які з'єднані спільними вершинами, називається сіткою (*mesh*). Для того, щоб сітка виглядала привабливо при рендерингу, бажано, щоб вона не перетиналася сама собою, тобто жодне ребро не проходило через полігон. Іншою властивістю є те, що сітка не повинна містити помилок, таких як подвоєні вершини, ребра або грані. Для деяких цілей важливо, щоб сітка була бездефектною – тобто вона не містила отворів або особливостей (місць, де дві різні частини сітки з'єднані однією вершиною).

Оскільки сучасні комп'ютери оптимізовані для обробки полігонів, полігональні моделі найлегше відображати і візуалізувати. Полігональне моделювання дозволяє дизайнерам створювати більш унікальні/органічні дизайни (люди, тварини і т. д.). Оскільки полігональні моделі створюються з менших компонентів (полігонів і трикутників), їх можна більш природно деформувати і анімувати.

Існує багато недоліків для представлення об'єкта з використанням полігонів. Полігони нездатні точно представляти вигнуті поверхні, тому велика їх кількість має використовуватися для апроксимації кривих візуально привабливим чином. Використання складних моделей вимагає зниження швидкості. При перетворенні в розгортку (*scanline)* кожен полігон повинен бути перетворений і відображений незалежно від розміру, і на екрані часто з'являється велика кількість моделей в будь-

який момент часу. Часто програмісти повинні використовувати кілька моделей з різними рівнями деталізації, щоб представляти один і той же об'єкт, щоб скоротити кількість полігонів, які відображаються [2].

1.1.2. Криволінійне моделювання

Криволінійне моделювання (*curve modeling*) – це метод в області комп'ютерної графіки і тривимірного моделювання, який використовує криві для створення і формування поверхонь об'єктів. Криві контролюються ваговими точками. Крива проходить через ці точки (але не обов'язково інтерполює їх). Збільшення ваги для точки зміщує криву ближче до цієї точки.

Криволінійне моделювання відсилається до певного методу або техніки, яка використовується для створення геометричних форм шляхом «вирізання» або «видавлювання» матеріалу з базового об'єкта. Цей підхід часто використовується для створення складних органічних форм або деталей, де потрібно видалити частину матеріалу, щоб створити бажану форму.

Процес криволінійного моделювання може бути використаний для створення деталей, таких як рельєфи на поверхні об'єкта, витончених кривих або складних текстур. Він дозволяє художникам і дизайнерам використовувати свою творчість і досліджувати різноманітні форми та структури, виходячи з базового об'єкта. В основі моделювання кривих лежать математичні криві, такі як Без'є, *B*-сплайни, *NURBS* (нелінійні раціональні базисні функції) і інші. Ці криві визначають форму об'єкта, і їх можна контролювати, змінюючи положення і параметри точок керування, які визначають криву. Наприклад, за допомогою моделювання кривих можна створити плавні поверхні для автомобілів, літаків, меблів, живопису та багатьох інших об'єктів [2]. Результатом застосування цього методу є можливість досягнення вражаючої якості та деталізації у створенні графічних об'єктів. Далі можна побачити приклад криволінійного моделювання (рис. 1.3).

Однією з характерних особливостей криволінійного моделювання є можливість створення плавних переходів між поверхнями, що робить його ідеальним для створення реалістичних об'єктів.

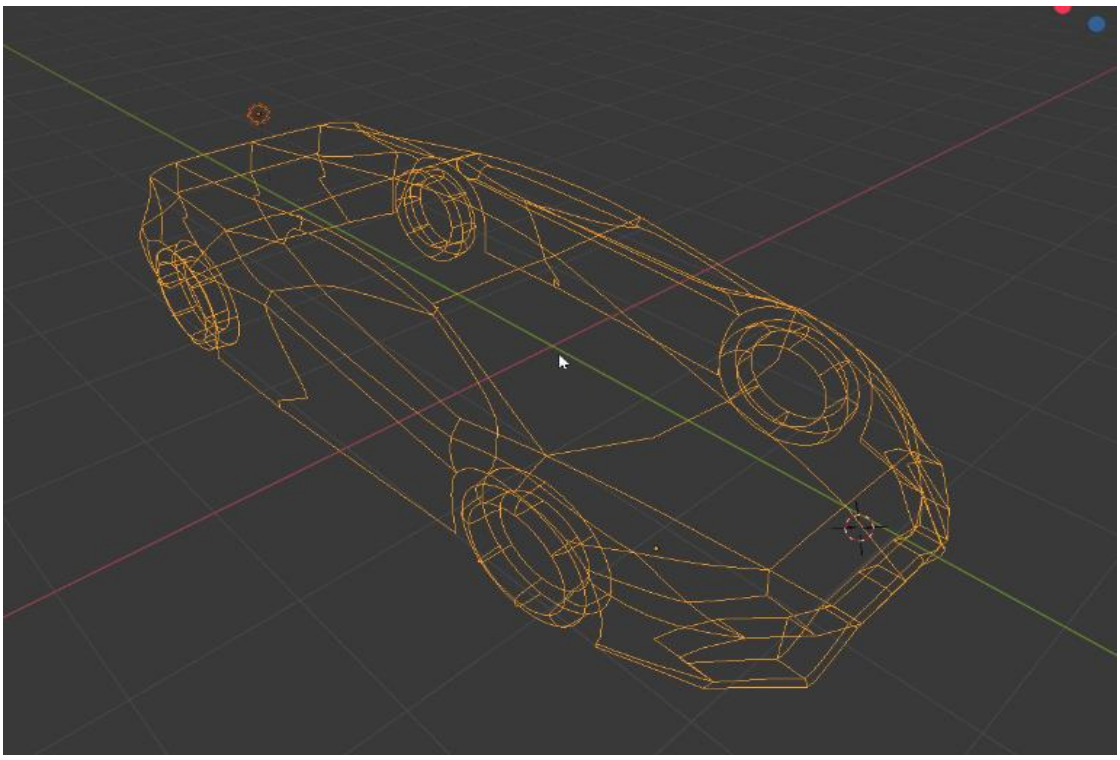


Рис. 1.3. Криволінійне моделювання

1.1.3. Цифрове ліплення

Цифрове ліплення (*digital sculpting*) – це досить новий метод моделювання, який за останні кілька років став дуже популярним. Геометрія, використовувана в програмах цифрового ліплення для представлення моделі, може варіюватися, кожна з них має свої переваги і обмеження. Більшість цифрових інструментів для ліплення на ринку використовують сітчасту геометрію, в якій об'єкт представлений взаємопов'язаною поверхневою мережею полігонів, які можна маніпулювати. Це в деякій мірі схоже на фізичний процес вибивання рельєфної сцени на мідних пластинах. Інші цифрові інструменти для ліплення використовують об'ємну геометрію на основі вокселів (*voxels*), де об'єм об'єкта є основним елементом [3]. Матеріал можна додавати та видаляти, подібно до ліплення глиною. Деякі інструменти використовують більше одного виду геометрії. Далі можна побачити приклад цифрового ліплення (рис. 1.4).

Цифрове ліплення є важливим інструментом для ігрової індустрії та кінематографії. Моделі, створені за допомогою цього методу, використовуються для розробки персонажів, архітектурних об'єктів, предметів інтер'єру та багато іншого.

Рис. 1.4. Цифрове ліплення

Перевагою сіткових (*mesh-based*) програм є можливість ліплення моделі на кількох рівнях деталізації. Області моделі, які мають дрібні деталі, можуть мати дуже малі полігональні елементи, тоді як інші області можуть мати більші полігони. У багатьох програмах на основі сітки, мережу можна редагувати на різних рівнях деталізації, і зміни на одному рівні вплинуть на вищі та нижчі рівні деталізації моделі. Обмеження цифрового ліплення полягає у фіксованій топології мережі; конкретна організація полігонів може обмежувати способи додавання або маніпуляції деталями. Перевагою є те, що вокселі дозволяють повну свободу форми. Топологію моделі можна змінювати безперервно під час процесу ліплення шляхом додавання та видалення матеріалу, що звільняє скульптора від необхідності враховувати розташування полігонів на поверхні моделі. Вокселі, однак, мають обмеження в роботі з різними рівнями деталізації. У відмінності від полігонального моделювання, значні зміни, зроблені вокселями на низькому рівні деталізації, можуть повністю знищити дрібні деталі. Існують три типи цифрового скульптурування:

– переміщення (d*isplacement*), який зараз найбільш поширений серед програм, використовує щільну модель (часто генеровану поверхнями підрозділів багатокутної

керуючої сітки) і зберігає нові положення вершин за допомогою карти зображення, яка містить відкориговані положення;

– об'ємне скульптурування (*volumetric*), що використовує концепцію вокселів (*voxels*), має подібні можливості до переміщення, але не має проблеми розтягування полігонів, коли в деякій області недостатньо полігонів для досягнення деформації;

– динамічна тесселяція (*dynamic tessellation*), схожа на воксельну, розбиває поверхню на трикутники для збереження гладкої поверхні і можливості відтворити дрібні деталі, що дозволяє виконувати художнє дослідження, оскільки над моделлю буде створено нову топологію після того, як буде сформована сама модель і, можливо, будуть виконані деталі.

1.2. Етапи розробки *3D*-моделей

Процес створення *3D*-моделі поділяється на кілька етапів, кожен із яких відіграє ключову роль у реалізації проєкту [8].

Перший крок у створенні *3D*-моделі – це концептуалізація. Він включає в себе розробку плану і створення основної концепції для моделі. Для чого модель створюється? Які її розміри? Які матеріали використовуватимуться? Для створення успішної *3D*-моделі маркетологам, дизайнерам та виробникам необхідно спочатку визначити цілі. Що вони намагаються показати або пояснити за допомогою *3D*-моделі? Якщо зрозуміла концепція, то легше створити більш конкретний план.

Наступним кроком у створенні *3D*-моделі є розробка основної геометрії продукту за допомогою програм для *3D*-моделювання. Це включає в себе створення фундаменту для моделі за допомогою базових фігур і ліній. Цей етап може бути трохи важким, оскільки потрібно бути точними, щоб створити добре сформовану *3D*-структуру. *3D*-моделі створюються багатьма різними способами, але найпопулярніші це моделювання твердих і поверхневих типів. Можна створити все, від простого куба до складної машини за допомогою цього типу моделювання.

Наступним кроком у створенні *3D*-моделі є модифікація полігонів та топології. Полігони це будівельні блоки *3D*-моделей, і їх потрібно створювати та модифікувати

правильно, щоб побудувати якісну модель. Це критично для *3D*-моделей, які є занадто великими для використання в додатках, віртуальній реальності, розширеній реальності або комп'ютерних іграх. Топологія – це структура полігонів і їх зв’язки в межах моделі. Гарна топологія важлива для оптимізації та зручності редагування моделі. Модифікація топології може включати в себе операції покращення розподілу полігонів, створення петель для підтримки анімації, а також додавання підсвічувань для створення реалістичних освітлення та тіней. Модифікація топології *3D*-моделі може бути складним завданням, але вона є важливою для створення високоякісного продукту. Це включає в себе переконання, що полігони розташовані в правильних місцях і що модель вірно збалансована.

Четвертим кроком у процесі розробки моделі є додавання деталей до *3D*-моделі. Це включає в себе додавання текстур (розгортання *UV*-координат), кольорів, нормалей, дисплейних карт, висотних карт, освітлення, затінення та інших елементів, щоб зробити модель реалістичною. Деталі є важливими, оскільки вони можуть підсилити або погіршити загальний вигляд моделі. Текстура є однією з найважливіших деталей, які можна додати до *3D*-моделі. За допомогою відповідного програмного забезпечення для текстурування *3D*, модель буде виглядати так, ніби вона вийшла прямо з реального світу.

Рендеринг – це наступний крок процесу створення зображення з тривимірної сцени або моделі [6]. Цей процес забезпечує перетворення тривимірних об’єктів і оточення в двовимірне зображення з врахуванням освітлення, тіней, кольорів та текстур. Цей етап є надзвичайно важливим, оскільки він дозволяє побачити модель в її остаточному вигляді. Рендеринг *3D*-моделі може бути часом витратним процесом, але результати варті зусиль. З відповідними програмами для *3D*-рендерингу можна створити вражаючі зображення і анімації. Рендеринг також полягає в деталізації налаштувань відображення *3D*-моделі: розрахунку глибини (рендерер обчислює глибину об’єктів, щоб визначити, який об’єкт перебуває попереду іншого), згладжування (техніка допомагає створити більш гладкий вигляд), додаванні графічних спецефектів, таких як відблиски, туман, сяйво і таке інше.

Останнім кроком у процесі *3D*-моделювання є пост-процесинг. Це фаза в графічному процесі, яка відбувається після рендерингу сцени або об’єкта і перед виведенням зображення на екран або в медіаформат. Пост-процесинг використовується для покращення, зміни або стилізації зображення з метою досягнення певних художніх або технічних ефектів. Пост-процесинг включає в себе такі процеси: колірна корекція (дозволяє змінювати кольорову палітру, насиченість, яскравість та інші параметри кольору для досягнення бажаного візуального ефекту), розмиття, вийняткового кольорового виокремлення, реалістичних світлових променів, об’ємних тіней, фільтри та шумопониження, можливість редагування кадру (змінити розмір, обрізати, змінити співвідношення сторін). Пост-процесинг також може використовуватися для покращення загальної якості. Завершення *3D*-моделі є важливим етапом у процесі створення об’єкту, і важливо впевнитися, що зображення чи анімація мають найвищу якість.

1.3. Перелік галузей застосування *3D*-моделей

3D-моделі знаходять широке застосування у різних галузях, завдяки своїй здатності точно відтворювати об'єкти та їх властивості у тривимірному просторі. Деякі з найбільш поширених галузей, де використовуються *3D*-моделі, включають:

– архітектура та будівництво: *3D*-моделі дозволяють архітекторам та інженерам створювати віртуальні прототипи будівель, проєктувати планування, аналізувати конструкцію та взаємодію з оточенням перед фізичною реалізацією проєктів;

– комп'ютерні ігри та розваги: *3D*-графіка використовується для створення віртуальних світів, персонажів, об'єктів, ландшафтів, анімацій та спеціальних ефектів. Анімаційні студії створюють *3D*-анімацію для фільмів, рекламних відео і відеоігор;

– медицина та біологія*: 3D*-моделювання використовується для створення тривимірних моделей органів, тканин та клітин, що дозволяє лікарям і науковцям досліджувати, діагностувати та лікувати хвороби, а також планувати хірургічні втручання, складні операції та навчання медичних студентів;

– промисловість та виробництво: *3D*-моделі використовуються для проєктування та виготовлення компонентів, виробів та машин, а також для віртуального тестування їх функціональності та ефективності;

– наука та дослідження: *3D*-дані використовуються для моделювання природних явищ, кліматичних змін, галактик, молекулярних структур та інших об'єктів, що допомагає науковцям розуміти складні процеси та розробляти нові технології;

– географія та картографія: *3D*-дані дозволяють створювати точні тривимірні моделі місцевості та географічних об'єктів, що сприяє удосконаленню карт, навігаційних систем та плануванню маршрутів;

– реклама та маркетинг: у сучасних презентаціях продукції майбутнього важливим елементом є тривимірна графіка. Щоб розпочати виробництво, необхідно спочатку створити малюнок та на його основі створити тривимірну модель об'єкту. Після цього, використовуючи різні технології швидкого прототипування, такі як *3D*-друк, фрезерування, лиття силіконових форм та інші, можна створити реалістичний прототип майбутнього виробу.

Таким чином, *3D*-дані грають важливу роль у багатьох галузях, де вони забезпечують візуалізацію, аналіз, моделювання та вирішення різноманітних задач, що робить розробку кросплатформного програмного додатку для роботи з цими даними актуальною та значущою.

1.4. Аналіз існуючих програмних рішень для роботи із 3*D*-моделями

Аналіз програмних рішень для роботи із *3D*-моделями допомагає визначити їх функціональні можливості, переваги та недоліки, а також сприяє вибору найбільш підходящих інструментів для реалізації мети дослідження. У даному розділі будуть вивчені різноманітні програмні рішення, призначені для роботи з 3*D*-моделями.

SolidWorks (SolidWorks Corporation) застосовується для дизайну, деталізації та візуалізації продуктів, систем, машин та оснащення. Всі версії включають

моделювання, збірки, малювання, зварювані деталі та функціональність поверхні вільної форми. Він також підтримує *Visual Basic* та *C*.

ProEngineering – система автоматизованого проєктування, інженерного аналізу та підготовки виготовлення виробів будь-якої складності і призначення. *ProEngineering* є ядром інтегрованого комплексу автоматизації підприємства, за допомогою якого здійснюється підтримка життєвого циклу виробу відповідно до концепції *CALS*-технологій (*Continuous Acquisition and Life cycle Support*), включаючи двонаправлений обмін даними з іншими *Windows*-додатками і створення інтерактивної документації. *ProEngineering Enterprise SE (Standard Edition*) – повний інструментальний пакет, що забезпечує комплексне рішення задач розробки виробу і точно відповідає сучасним вимогам глобально-розподілених виробничо-конструкторських груп.

3DMAX у своєму розпорядженні має засоби для створення різноманітних за формою і складністю тривимірних комп'ютерних моделей, реальних чи фантастичних об'єктів навколишнього світу, з використанням різноманітних технік і механізмів, включаючи полігональне моделювання, в яке входять *editable mesh* (редагована поверхня) *editable poly* (редагований полігон). Це поширений метод моделювання, який використовується для створення складних моделей і низькополігональних моделей для ігор.

SketchUp Pro (Trimble) – програма для моделювання, що підтримує *2D* та *3D*-моделі. Безкоштовна версія також доступна та інтегрована в *Google Earth. SketchUp* – програма для моделювання відносно простих трьох-вимірних об'єктів – будівель, меблів, інтер'єру. Основною особливістю цієї програми є майже повна відсутність вікон попередніх налаштувань. Всі геометричні характеристики під час або зразу після закінчення дії інструменту задаються з клавіатури в поле *value control box* (поле контролю параметрів), яке знаходиться в правому нижньому кутку робочої області, справа від напису *measurements* (панель вимірів). Ще одною ключовою особливістю є інструмент «*Push/Pul*» («Тягни/Штовхай»), завдяки якому будь-яку площину можна «витягнути» в сторону, створивши по мірі її руху нові бокові стінки.

Рухати площину можна в притик до наперед заданої кривої, для цього служить спеціальний інструмент «*Follow Me*» («Ведення»).

AutoCAD – двовимірна і тривимірна система автоматизованого проєктування і креслення, що включає в себе повний набір інструментів для комплексного тривимірного моделювання (підтримується твердотіле, поверхневе і полігональне моделювання). *AutoCAD* дозволяє отримати високоякісну візуалізацію моделей за допомогою рендеринга «*mental ray*». Також в програмі реалізовано управління тривимірним друком (результат моделювання можна відправити на *3D*-принтер) і підтримка хмарних точок (дозволяє працювати з результатами *3D*-сканування). Тим не менш, слід зазначити, що відсутність тривимірної параметризації не дозволяє *AutoCAD* безпосередньо конкурувати з машинобудівними САПР, такими як *Inventor, SolidWorks* та іншими.

Inventor (*Autodesk*) *3D*-САПР для створення і вивчення поведінки цифрових прототипів виробів і деталей. Розробник компанія *Autodesk*. Створений для *3D*-дизайну механіки, емуляції продукту, створення інструментаріїв.

КОМПАС-*3D* – інтерактивний графічний редактор з сучасним інтерфейсом, оснащений інструментальними засобами, які дозволяють створювати твердотілі об'єкти з використанням набору елементарних параметричних тіл (паралелепіпед, циліндр та ін.). Основні компонентні базові можливості системи передбачають функціонал, який дозволяє спроєктувати виріб будь-якого ступеня складності в *3D*, а потім оформити на цей виріб комплект документації, необхідний для його виготовлення відповідно до чинних стандартів.

SolveSpace – безкоштовне програмне забезпечення з відкритим кодом для параметричного автоматизованого проєктування САПР, що підтримує базове *2D/3D*-моделювання. Загалом застосовується для креслення окремих деталей або збірних механізмів. Його ще називають параметричним модельєром з простими можливостями механічного моделювання. Версія програми розроблена під всі широковідомі операційні системи: *Windows, Linux* та *MacOS*. Із 2022 року програмне забезпечення підтримується спільнотою волонтерів. Розробка *SolveSpace* розпочалася

у 2008 році. Попередній програмний пакет під назвою *SketchFlat*, був також замінений на *SolveSpace.*

Висновки за розділом

Тривимірне моделювання – це процес розробки математичного координатного представлення будь-якої поверхні об'єкта в трьох вимірах за допомогою спеціалізованого програмного забезпечення шляхом маніпулювання ребрами, вершинами та полігонами в модельованому тривимірному просторі.

3D-моделі представляють фізичне тіло за допомогою набору точок в тривимірному просторі, які з'єднані різними геометричними об'єктами, такими як трикутники, лінії, криві поверхні і т.д.

Залежно від використаного програмного забезпечення та технічних характеристик комп'ютера, тривимірні моделі можна створити за допомогою кількох різних процедур, таких як моделювання на основі алгоритмів, ручне малювання (на комп'ютері) чи сканування.

Існує три способи представлення *3D*-моделей: полігональне моделювання (*polygonal modeling*), криволінійне моделювання (*curve modeling*) та цифрове ліплення (*digital sculpting*).

Етапами розробки *3D*-моделей є: концептуалізація, розробка основної геометрії, модифікація полігонів та топології, додавання текстур, кольорів та інших деталей, рендеринг та пост-процесинг.

3D-моделювання застосовується в: архітектурі та будівництві, комп’ютерних іграх, медицині та біології, промисловості та виробництві, науці та дослідженнях, географії та картографії, рекламі та маркетингу.

До списку найпоширеніших програм, що працюють із *3D*-моделями входять: *SolidWorks (SolidWorks Corporation), ProEngineering, 3DMAX, SketchUp Pro (Trimble), AutoCAD, Inventor* (*Autodesk*), КОМПАС-*3D, SolveSpace.*

РОЗДІЛ 2
ОПИС СТЕКУ ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ ДОДАТКУ

2.1. Мова програмування

Мова програмування – це штучна мова, яку програмісти використовують для розробки програм, скриптів або інших наборів інструкцій для виконання комп'ютерами; будь-який набір правил, який перетворює рядки або графічні елементи програми у випадку мов візуального програмування в різні види виводу машинного коду. Хоча багато мов мають схожість, кожна з них має свій власний синтаксис. Як тільки програміст вивчає правила мови, синтаксис і структуру, він пише вихідний код у текстовому редакторі або *IDE*. Потім програміст компілює код на машинну мову, яку може зрозуміти комп'ютер. Мови сценаріїв, які не потребують компілятора, використовують інтерпретатор для виконання сценарію. Створено тисячі різних мов програмування, і щороку їх створюється більше. Мова програмування забезпечує структурований механізм для визначення фрагментів даних, а також операцій або перетворень, які можуть виконуватися автоматично над цими даними. Програміст використовує абстракції, наявні в мові, щоб представити концепції, залучені в обчислення. Ці концепції представлені як сукупність найпростіших доступних елементів. Програмування – це процес, за допомогою якого програмісти об'єднують ці примітиви для створення нових програм або адаптації існуючих до нових цілей чи середовища, що змінюється. *C++* – це мова програмування високого рівня, розроблена Б'ярном Страуструпом. Метою Страуструпа було створення такої мови, яка компілюється в економний, ефективний код, але також забезпечує високорівневі абстракції для кращого керування великими проєктами розробки. Синтаксис *C++* значною мірою успадкований від мови *C*. Він додає до свого попередника функції об'єктно-орієнтованого програмування, такі як класи, абстракція, інкапсуляція, успадкування та поліморфізм. Він також забезпечує

функціональність для перевантаження функцій і операторів, загальні засоби програмування (наприклад, можливість створювати шаблони) та обробку винятків.

C++ також має надійну *STL* (стандартну бібліотеку) корисних структур даних, алгоритмів та засобів введення/виводу. C++ працює на більшості платформ (на всіх основних), таких як *Windows, macOS* і різні версії *UNIX*, наприклад *Linux*. *C++* – це мова загального призначення, що означає, що її можна використовувати для створення різноманітних додатків. *C++20* – це остання версія стандарту, яка, наприклад, у грудні 2020 року нарешті додала підтримку модулів (в той час як багато мов підтримували модулі за десятиліття, якщо не два чи більше роки). У великих компіляторах вже майже повна підтримка *C++20*, але майже всі за замовчуванням використовують старий стандарт *C++17*, тому потрібно перемикати компілятор для активації підтримки *C++20*. *C++* залишається однією з найпопулярніших мов програмування завдяки своїй потужності та гнучкості. З кожним новим стандартом, мова доповнюється сучасними можливостями, що відображає актуальні тенденції в розробці програмного забезпечення [11].

2.2. Фреймворк для розробки користувацького інтерфейсу

QT – кросплатформний інструментарій розробки програмного забезпечення (ПЗ) мовою програмування *C++* [4]. Дозволяє запускати написане за його допомогою ПЗ на більшості сучасних операційних систем (ОС), просто компілюючи текст програми для кожної операційної системи без зміни початкового коду. Містить всі основні класи, які можуть бути потрібні для розробки прикладного програмного забезпечення, починаючи з елементів графічного інтерфейсу й закінчуючи класами для роботи з мережею, базами даних, *OpenGL*, *SVG* і *XML*. Бібліотека дозволяє керувати потоками, працювати з мережею та забезпечує кросплатформний доступ до файлів. З часу своєї появи в 1996 році комерційна версія бібліотеки *Qt* лягла в основу тисяч успішних проєктів у всьому світі. Крім того, *Qt* є фундаментом популярного робочого середовища *KDE*, що входить до складу багатьох дистрибутивів *GNU/Linux*. Серед відомих проєктів особливо треба відзначити: програма для *IP*-телефонії *Skype*,

програма для обробки зображень *Adobe Photoshop Album*, мережева карта світу *Google Earth*.

Відмінна особливість *Qt* від інших бібліотек – використання *Meta Object Compiller (MOC)* — попередньої системи обробки початкового коду (загалом, *Qt* – це бібліотека не для чистого *C++*, а для його особливого діалекту, з якого й «перекладає» *MOC* для подальшої компіляції будь-яким стандартним *C++* компілятором). *MOC* дозволяє в багато разів збільшити потужність бібліотек, вводячи такі поняття, як слоти (*slots*) і сигнали (*signals*). *Qt* комплектується графічним середовищем розробки графічного інтерфейсу *QTDesigner*, що дозволяє створювати діалоги і форми «мишею». Ідеологія створення форм у *Qt* базується на використанні менеджерів розташування, котрі надають «гумовий» дизайн, при якому розмір і розташування елементів форм визначаються автоматично, що значно прискорює розробку графічного інтерфейсу. В поставці *Qt* є «*Qt Linguist*» – могутня графічна утиліта, що дозволяє спростити локалізацію й переклад вашої програми багатьма мовами, та «*Qt Assistant*» – довідкова система *Qt*, що спрощує роботу з документацією для бібліотек і дозволяє створювати кросплатформну довідку для ПЗ, розробленого на основі *Qt*.

Qt використовується для створення графічних інтерфейсів користувача (*GUI*) та кросплатформних додатків, які працюють на різних настільних платформах та мобільних або вбудованих системах. Більшість програм із *GUI*, створених з використанням *Qt*, мають інтерфейс, який виглядає нативно, тому *Qt* вважається набором інструментів для віджетів. Також можна створювати програми без *GUI*, такі як інструменти командного рядка та консольні застосунки для серверів. Прикладом такого програмного продукту без *GUI*, який використовує *Qt*, є веб-фреймворк *Cutelyst*.

Qt підтримує різні компілятори, включаючи *GCC C++*, набір *Visual Studio, PHP* за допомогою розширення для *PHP5*, і має обширну підтримку інтернаціоналізації. Крім того, *Qt* надає *Qt Quick*, який включає декларативну мову сценаріїв під назвою *QML*, що дозволяє використовувати *JavaScript* для реалізації логіки. З використанням *Qt Quick* стало можливим швидке розроблення додатків для мобільних пристроїв, при

цьому логіку все ще можна програмувати мовою нативного коду для досягнення максимальної продуктивності.

Qt базується на ключових концепціях, що описані нижче.

– Повна абстракція графічного інтерфейсу користувача (*GUI*). Початково *Qt* використовував власний двигун малювання та елементи керування, емулюючи вигляд різних платформ, на яких він працює, при відображенні своїх віджетів. Це спрощувало роботу з перенесенням, оскільки дуже небагато класів у *Qt* дійсно залежали від цільової платформи; однак це іноді призводило до невеличких розходжень там, де ця емуляція була неповною. У новіших версіях *Qt* використовуються *API* нативного стилю різних платформ, на платформах з власним набором віджетів, для отримання метрик та малювання більшості елементів керування, і такі проблеми виникають рідше. На деяких платформах (таких як *MeeGo* та *KDE*) *Qt* є нативним *API*. Деякі інші переносні графічні інструментарії вибрали інші концепції дизайну; наприклад, *wxWidgets* використовує інструментарії цільової платформи для своїх реалізацій.

– Сигнали та слоти. Мовна конструкція, введена в *Qt* для забезпечення зв'язку між об'єктами, яка спрощує реалізацію патерна спостерігача, уникати надмірного коду. Ідея полягає в тому, що віджети *GUI* можуть відправляти сигнали з інформацією події, які можуть бути прийняті іншими елементами керування за допомогою спеціальних функцій, відомих як слоти.

– Метаоб'єктний компілятор. Метаоб'єктний компілятор, позначений як *moc* – це інструмент, який запускається на вихідних текстах програми *Qt*. Він інтерпретує певні макроси з коду *C++*, як анотації, і використовує їх для генерації додаткового коду *C++* з метаінформацією про класи, використовувані в програмі. Цю метаінформацію використовує *Qt* для надання функцій програмування, які не доступні нативно в *C++*: сигнали та слоти, інтроспекція та асинхронні виклики функцій.

– Мовні обгортки. *Qt* може використовуватися в декількох мовах програмування, таких як *Python, Javascript, C#* та *Rust*, за допомогою мовних обгорток; для *Qt* 5 існують обгортки для багатьох мов, а для *Qt* 4 - відповідно.

– Модулі *Qt*. Починаючи з *Qt* 4.0, фреймворк був розділений на окремі модулі. *Qt* тепер розділений на обов'язкові та додаткові модулі, такі як: *Qt Core, Qt GUI, Qt Widgets, Qt QML, Qt Quick, Qt Network, Qt Multimedia, Qt SQL, Qt WebEngine, Qt Test* та інші.

2.3. Специфікація розробки програмного інтерфейсу для використання комп’ютерної графіки

Комп'ютерна графіка широко використовується в повсякденному житті. Вчені використовують її для аналізу результатів моделювання. Інженери та архітектори використовують тривимірну графіку для створення віртуальних моделей. Кінематографісти створюють спецефекти або повністю анімовані фільми (наприклад, «Шрек», «Історія іграшок» і ін.). Останнім часом комп'ютерні ігри, які максимально використовують тривимірну графіку для створення віртуальних світів, також набули великої популярності.

Із поширенням комп'ютерної графіки виникли свої труднощі. У 1990-х роках розробка програмного продукту, який міг би працювати на різному графічному обладнанні, вимагала значних часових і фінансових витрат. Для кожного типу графічних адаптерів доводилося створювати окремі модулі, що інколи призводило до непотрібного розмноження програмного коду. Це значно сповільнювало розвиток і поширення комп'ютерної графіки.

Компанія *Silicon Graphics (SGI)* спеціалізувалася на створенні високотехнологічного графічного обладнання та програмних засобів. Будучи лідером у тривимірній графіці, *SGI* визнавала проблеми та бар'єри в зростанні ринку. Тому було прийнято рішення стандартизувати доступ до графічного обладнання на рівні програмного інтерфейсу.

Так з’явився програмний інтерфейс *OpenGL*, який стандартизує доступ до графічного обладнання, зміщуючи відповідальність за створення апаратного драйвера на виробника графічного пристрою. Це дозволило розробникам програмного забезпечення використовувати більш високий рівень абстракції від

графічного обладнання, що значно прискорило створення нових програмних продуктів та знизило витрати на них.

OpenGL – специфікація, що визначає незалежний від мови програмування кросплатформний програмний інтерфейс (*API*) для написання застосунків, що використовують *2D* та *3D*-комп'ютерну графіку [5]. Цей інтерфейс містить понад 250 функцій, які можуть використовуватися для малювання складних тривимірних сцен з простих примітивів. Широко застосовується індустрією комп'ютерних ігор і віртуальної реальності, у графічних інтерфейсах (*Compiz, Clutter*), при візуалізації наукових даних, в системах автоматизованого проєктування тощо.

На основному рівні *OpenGL* – це лише специфікація, або, іншими словами, документ, який описує набір функцій та їх точне виконання. Виробники обладнання, використовуючи цю специфікацію, створюють свої реалізації – бібліотеки функцій, що відповідають вказаному набору. Основна мета реалізації – ефективне використання можливостей обладнання. Якщо обладнання не може реалізувати певну функціональність, вона повинна бути програмно емульована. Виробники апаратури проходять тестування на відповідність (*conformance tests*), перш ніж їх реалізацію визнають як *OpenGL*-сумісну. Розробники програмного забезпечення повинні лише вивчити використання функцій, описаних у специфікації, оскільки реалізація залишається поза їхньою компетенцією. Ефективні реалізації *OpenGL* підтримуються для *Windows, Unix*-платформ і *Mac OS* і надаються виробниками відеоадаптерів, які активно використовують можливості останніх. Також існують відкриті реалізації специфікації *OpenGL*, наприклад, бібліотека *Mesa*. З ліцензійних питань *Mesa* вважається «неофіційною» реалізацією *OpenGL*, хоча вона повністю сумісна з нею на рівні коду та підтримує як програмну емуляцію, так і апаратне прискорення за наявності відповідних драйверів. Специфікацію *OpenGL* регулярно переглядає консорціум *ARB* (*Architecture Review Board*), сформований у 1992 році. Цей консорціум об'єднує компанії, які мають інтерес до створення широкодоступного *API*. За інформацією з офіційного сайту *OpenGL*, члени *ARB* з визначальним голосом на листопад 2004 року включають виробників професійного графічного обладнання, таких як *SGI, 3Dlabs, Matrox* та *Evans* & *Sutherland* (військові застосування), а також

виробників споживчого графічного обладнання *ATI* і *NVIDIA*, процесорний виробник *Intel*, і виробників комп'ютерів та комп'ютерного обладнання, таких як *IBM, Apple, Dell, Hewlett-Packard і Sun Microsystems*, а також одного з лідерів геймінгової індустрії, *id Software*. Однак *Microsoft*, один з ініціаторів консорціуму, покинула його в березні 2003 року. Крім постійних членів, щорічно до *ARB* запрошується багато інших компаній, які стають частиною *OpenGL ARB* протягом одного року. Це робить *OpenGL* відомим інтерфейсом загального призначення з великою кількістю можливостей завдяки широкому колу зацікавлених компаній.

OpenGL визначає своє завдання через дві ключові мети:

– спростити процес адаптації до різних *3D*-прискорювачів, надаючи розробникам єдиний *API*;

– забезпечити уніфіковану роботу навіть на різних апаратних платформах, використовуючи програмну емуляцію для реалізації відсутньої функціональності.

Основний принцип роботи *OpenGL* полягає в опрацюванні векторних графічних примітивів, таких як точки, лінії та багатокутники, з подальшою математичною обробкою отриманих даних і формуванням растрового зображення на екрані або в пам'яті. Всі ці процеси виконуються графічним конвеєром, що представляє собою дискретний автомат, що зображено нижче (рис. 2.1).

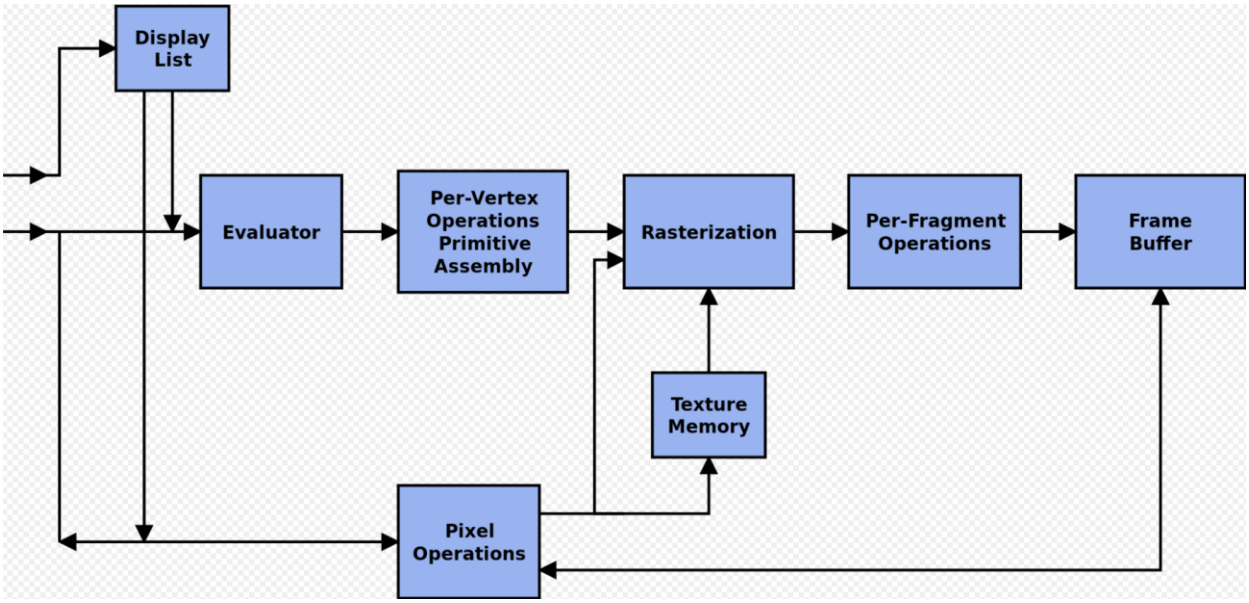


Рис. 2.1. Процеси графічного конвеєра

Більшість команд *OpenGL* поділяються на дві групи: додавання графічних примітивів до конвеєра або конфігурування його для різних трансформацій.

OpenGL є низькорівневим процедурним *API*, де програміст визначає точну послідовність кроків для побудови результуючого растрового зображення (імперативний підхід). Це відрізняє його від дескрипторних підходів, де сцена передається як структура даних і обробляється автоматично. Імперативний підхід вимагає від програміста глибокого розуміння тривимірної графіки та математичних моделей, але водночас надає волю для впровадження різноманітних інновацій.

2.4. Бібліотека алгоритмів обчислювальної геометрії для обробки 3*D*-даних

Геометричні алгоритми виникають у різних галузях комп’ютерних наук. Графіка та віртуальна реальність, комп'ютерне проєктування та виробництво, тривимірне моделювання, робототехніка, географічні інформаційні системи, комп'ютерне зорове сприйняття, відновлення форми, молекулярне моделювання та проєктування схем – це найвідоміші приклади. З результатів досліджень конкретних геометричних проблем в цих областях розробка та аналіз геометричних алгоритмів вивчається в галузі обчислювальної геометрії. За останні два десятиліття в цьому підрозділі проєктування алгоритмів було розроблено багато ефективних геометричних методів та структур даних. Але багато з цих технік ще не знайшли свого застосування, головним чином через те, що правильна реалізація навіть найпростіших з цих алгоритмів може бути вкрай складною задачею. Це передусім пов'язано з проблемою дегенерації та точності: теоретичні статті передбачають, що вхідні дані є взагалі взяті в загальному положенні і передбачають точні арифметичні операції з дійсними числами. Обидві припущення малоймовірно відповідають ситуації на практиці. Високорозвинені алгоритми призводять до додаткових труднощів тим, що їх часто важко розуміти та важко реалізовувати. З цих причин непрактично користувачам реалізовувати геометричні алгоритми з нуля. Для вирішення цієї ситуації потрібна бібліотека обчислювальної геометрії, яка надає правильні та ефективні повторно використовувані реалізації. Така бібліотека, названа *CGAL*

(*Computational Geometry Algorithms Library*). Метою бібліотеки є надання можливості використання в промисловості великого обсягу геометричних алгоритмів, розроблених в галузі обчислювальної геометрії. *CGAL* є ключовим інструментом для досягнення цієї мети і є основою для реалізації геометричних алгоритмів. Оскільки обчислювальна геометрія має багато потенційних областей застосування з різними потребами, гнучкість компонентів бібліотеки, особливо адаптивність та модульність, є важливими аспектами *CGAL*. Звісно, правильність, простота використання та ефективність були також метою дизайну.

CGAL – бібліотека алгоритмів обчислювальної геометрії є відкритим програмним забезпеченням, що містить набір алгоритмів обчислювальної геометрії, написана на *C++*. *CGAL* пропонує структури даних та алгоритми, такі як триангуляції, діаграми Вороного, алгоритм оболонки, розташування, триангуляція Делоне, створення сітки, пошук, аналіз форми, апроксимація, інтерполяція, полігони, клітинні комплекси та поліедри, аранжування кривих, обробка геометрії, алгоритми опуклої оболонки і багато інших. Усі ці структури даних та алгоритми працюють з геометричними об'єктами, такими як точки, сегменти і сітки та виконують геометричні операції над ними. Бібліотека *CGAL* відзначається високою оптимізацією та надійністю. Вона використовує різні методи оптимізації для забезпечення ефективності в роботі з великими обсягами даних. Крім того, бібліотека пропонує генератори геометричних об'єктів та функції просторового сортування, а також фреймворк для пошуку матриць і розв'язувач лінійних та квадратичних рівнянь. Вона також надає інтерфейси до програмного забезпечення сторонніх виробників, таких як бібліотеки для графічного інтерфейсу користувача *Qt* і бібліотека *Boost Graph Library*. *CGAL* доступний для використання в різних мовах програмування, включаючи *C++, Python* та інші. Це робить його дуже гнучким і дозволяє інтегрувати його в різноманітні проєкти. *CGAL* широко використовується у різних галузях, таких як комп'ютерна графіка, геоінформаційні системи, біомедичні дослідження, числове моделювання та інші [9].

2.5. Кросплатформний генератор сценаріїв складання

CMake – це розширюваний, відкрите програмне забезпечення, яке керує процесом збірки в операційній системі незалежно від компілятора. *CMake* виступає як альтернатива *Autotools* і використовується в таких проєктах, як *KDE, LLVM/Clang, MySQL, MariaDB, ReactOS і Blender*. Початковий код *CMake* написаний мовою *C++* і поширюється під ліцензією *BSD*. На відміну від багатьох кросплатформних систем, *CMake* призначений для використання разом зі стандартним середовищем збірки. Прості конфігураційні файли, розміщені в кожному каталозі джерела (називаються файлами *CMakeLists.txt*), використовуються для генерації стандартних файлів збірки (наприклад, *make*-файлів у *Unix* і проєктів/робочих просторів у *Windows MSVC*), які використовуються у звичайний спосіб. *CMake* може генерувати нативне середовище збірки, яке компілює вихідний код, створює бібліотеки, генерує обгортки та будує виконувані файли в довільних комбінаціях. *CMake* підтримує збірки на місці і зовнішні збірки, тому він може підтримувати кілька збірок з одного дерева джерела. *CMake* також підтримує статичні і динамічні збірки бібліотек. Ще однією цікавою функцією *CMake* є генерація кеш-файлу, який призначений для використання з графічним редактором. Наприклад, під час роботи *CMake* він знаходить файли, бібліотеки і виконувані файли, і може зустріти додаткові директиви збірки. Ця інформація збирається в кеш, який може бути змінений користувачем до генерації нативних файлів збірки [10].

CMake призначений для підтримки складних ієрархій каталогів та додатків, залежних від кількох бібліотек. Наприклад, *CMake* підтримує проєкти, що складаються з кількох інструментів (тобто бібліотек), де кожен інструмент може містити кілька каталогів, а додаток залежить від цих інструментів та додаткового коду. *CMake* також може обробляти ситуації, коли виконувані файли повинні бути зібрані для генерації коду, який потім компілюється і лінкується в кінцевий додаток. Оскільки *CMake* є відкритим програмним забезпеченням і має просту, розширювану конструкцію, його можна розширювати за необхідністю для підтримки нових

функцій. Використання *CMake* є простим. Процес збірки контролюється створенням одного або кількох файлів *CMakeLists.txt* в кожному каталозі (включаючи підкаталоги), які складаються проєкту. Кожен файл *CMakeLists.txt* складається з однієї або декількох команд. Кожна команда має форму *COMMAND* (аргументи...), де *COMMAND* – це назва команди, а аргументи - це список аргументів, розділених пробілами. *CMake* надає багато заздалегідь визначених команд, але якщо потрібно, ви можете додати власні команди. Крім того, досвідчений користувач може додати інші генератори *make*-файлів для певної комбінації компілятора/ОС. (В даний момент підтримуються *Unix* і *MSVC++*, інші розробники додають підтримку для інших компіляторів/ОС.) Ви можете ознайомитись зі сторінкою прикладів, щоб дізнатися більше подробиць.

2.6. Менеджер пакетів для завантаження та керування бібліотеками

Vcpkg – це безкоштовний менеджер пакетів *C/C++* для завантаження та керування бібліотеками. В ньому можна вибрати з понад 1500 відкритих бібліотек для завантаження та збірки за один крок або додати свої власні приватні бібліотеки, щоб спростити процес збірки. *Vcpkg* підтримується командою *Microsoft C++* та співробітниками з відкритим кодом. *Vcpkg* працює з обраною операційною системою, системою збирання, цільовими архітектурами, інтегрованою середовищем розробки (*IDE*), редактором та процесом постійної інтеграції. Бібліотеки будуються з вихідних кодів і можуть бути налаштовані. *Vcpkg* дозволяє: забезпечувати однорідність між локальними розробниками та робочими процесами *CI/CD*, синхронізувати свою екосистему залежностей із будь-яким розробником чи компанією, кешувати бінарні файли для швидкого використання, уникати проблем з розрішенням конфліктів залежностей, відтворювати ідентичні збірки для всіх розробників, машин *CI* та контейнерів [13].

Висновки за розділом

C++ – мова програмування високого рівня, розроблена Б’ярном Страуструпом. Синтаксис *C++* значною мірою успадкований від мови *C*. Він додає до свого попередника функції об’єктно-орієнтованого програмування, такі як класи, абстракція, інкапсуляція, успадкування та поліморфізм.

QT – кросплатформний інструментарій розробки програмного забезпечення (ПЗ) мовою програмування *C++*. Дозволяє запускати написане за його допомогою ПЗ на більшості сучасних операційних систем (ОС), просто компілюючи текст програми для кожної операційної системи без зміни початкового коду.

OpenGL – специфікація, що визначає незалежний від мови програмування кросплатформний програмний інтерфейс (*API*) для написання застосунків, що використовують *2D* та *3D*-комп'ютерну графіку. На основному рівні *OpenGL* – це лише специфікація, або, іншими словами, документ, який описує набір функцій та їх точне виконання. Виробники обладнання, використовуючи цю специфікацію, створюють свої реалізації – бібліотеки функцій, що відповідають вказаному набору.

CGAL – Бібліотека алгоритмів обчислювальної геометрії є відкритим програмним забезпеченням, що містить набір алгоритмів обчислювальної геометрії, написана на *C++*.

CMake – це розширюваний, відкрите програмне забезпечення, яке керує процесом збірки в операційній системі незалежно від компілятора. *CMake* призначений для підтримки складних ієрархій каталогів та додатків, залежних від кількох бібліотек.

Vcpkg – це безкоштовний менеджер пакетів *C/C++* для завантаження та керування бібліотеками. *Vcpkg* працює з обраною операційною системою, системою збирання, цільовими архітектурами, інтегрованою середовищем розробки (*IDE*), редактором та процесом постійної інтеграції.

РОЗДІЛ 3
АРХІТЕКТУРА ПРОГРАМНОГО ДОДАТКУ

Архітектура програмного забезпечення представляє собою набір структур, необхідних для врахування системи програмного забезпечення та процес створення таких структур і систем. Кожна структура включає елементи програмного забезпечення, їх взаємозв'язки та властивості як елементів, так і взаємозв'язків. Подібно до архітектури будівлі, архітектура програмної системи служить зразком для системи та проєкту розробки. Ці зразки можуть використовуватися керівництвом проєкту для уточнення завдань, які необхідно виконати командами та учасниками процесу. Основна мета архітектури програмного забезпечення – прийняття фундаментальних структурних рішень, які буде важко змінити після впровадження. Варіанти архітектурних рішень включають конкретні структурні опції у дизайні програмного забезпечення. Наприклад, системи управління ракетою-носієм *Space Shuttle* повинні бути швидкими і надійними. Тому вибір відповідної мови обчислень у реальному часі є обов'язковим. Також може бути зроблено вибір на користь кількох резервних та незалежно вироблених копій програми, які запускаються на незалежних обчислювальних пристроях і перевіряють результати між собою.

Основною ідеєю програмної архітектури є концепція зменшення складності системи шляхом використання абстракцій та чіткого визначення повноважень. Дотепер не існує загальної згоди щодо чіткого визначення терміну «архітектура програмного забезпечення».

Існують взаємозалежності між різними частинами будь-якого програмного коду. Класи потребують інших класів, функції викликають одна одну і так далі. При зростанні проєкту кількість таких взаємозалежностей збільшується. Вимоги до проєкту змінюються, розробники іноді приймають поспішні та не завжди оптимальні рішення. Неефективне управління залежностями може спричинити руйнування проєкту: зростання складності коду, його часті поломки, менша гнучкість і важкість

повторного використання. Це може призвести до спаду швидкості розробки та опору проєкту до змін.

Характеристики непродуктивного проєкту:

– закритість (*rigid*): система уперто опирається змінам через велику кількість взаємозалежностей, що робить внесення змін витратним та часозатратним;

– нестійкість, крихкість (*fragile*): система ламається в непередбачених місцях, навіть якщо зміни внесені раніше не торкнулись конкретних компонентів;

– нерухомість або монолітність (*not reusable*): система побудована так, що використання окремих компонентів неможливе;

– в'язкість (*high viscosity*): код проєкту робити легше неправильно, ніж правильно;

– невиправдані повторення (*high code duplication*): проєкт надто обширний через недостатнє застосування абстракцій;

– надмірна складність (*overcomplicated design*): проєкт включає рішення, що не є очевидно корисними, ускладнюючи розуміння та розвиток системи.

Протягом тривалого часу розумні індивіди визначили деякі ключові принципи ООП, які, якщо дотримуватися, сприяють створенню ефективної архітектури:

– висока зчепленість коду (*high cohesion*): код, що відповідає за конкретну функціональність, повинен бути локалізований в одному місці;

– низька зв'язаність коду (*low coupling*): класи повинні мінімізувати залежність від інших класів;

– вказуй, а не питай (*tell, don't ask*): класи мають володіти даними і методами для операцій з цими даними, не цікавлячись іншими класами;

– не розмовляй з незнайомцями (d*on't talk to strangers*): класи повинні знати тільки своїх безпосередніх сусідів, щоб код був стійким та чистим.

Ці рекомендації призначені для того, щоб розташувати класи логічно, об'єднати сильні зв'язки та встановити чіткі межі у коді.

Тим не менш, ці принципи можуть бути розмитими, і тому виникла певна набір чітких правил, якими слід керуватися при створенні архітектури:

– принцип персональної відповідальності (s*ingle responsibility principle*): кожен клас повинен мати тільки одну відповідальність;

– принцип відкриття-закриття (*open-closed principle*): класи повинні бути відкритими для розширень, але закритими для модифікацій;

– принцип підстановки Барбари Лісков (*liskov substitution principle*): дочірні класи можна використовувати через інтерфейси базових класів без знання конкретного типу дочірнього класу;

– принцип інверсії залежностей (*dependency inversion principle*): модулі верхнього рівня не повинні залежати від модулів нижнього рівня, абстракції повинні бути незалежними від подробиць;

– принцип відділення інтерфейсу (*interface segregation principle*): клієнти повинні залежати тільки від тих методів інтерфейсу, які вони фактично використовують.

В розробці програмного забезпечення використовуються «типові рішення» або «шаблони проєктування» (*design patterns*) як основа проєктування. Шаблони проєктування, або патерни, є часто вживаними архітектурними конструкціями, які надають рішення загальних проблем проєктування у конкретному контексті та пояснюють сутність цього рішення.

Патерн не є готовим проєктним зразком, який може бути прямо втілений в код, але, скоріше, це опис або модель для того, як ефективно вирішити завдання, забезпечуючи при цьому можливість застосування в різних сценаріях.

Об'єктно-орієнтовані шаблони часто вказують на взаємодію та відносини між класами або об'єктами, не конкретизуючи при цьому, які саме класи чи об'єкти будуть використовуватися в конкретних застосуваннях. Відмінною особливістю є те, що алгоритми не розглядаються як шаблони, оскільки вони вирішують завдання обчислення, а не проєктування.

Архітектурна модель дозволяє розглядати програму на високому рівні абстракції, і вона може бути представлена різними способами, такими як блок-схеми, діаграми компонентів, діаграми пакетів тощо. Ця модель визначає, як система

розділена на компоненти, як вони взаємодіють між собою, і як вони взаємодіють з іншими системами чи компонентами.

На високому рівні архітектурна модель може включати в себе такі елементи, як клієнтські та серверні компоненти, бази даних, інтерфейси користувача, модулі безпеки та інші. Вона визначає загальну структуру системи і надає основу для подальшого проєктування та розробки програмного додатку [14].

3.1. Визначення вимог та функціональності програмного додатку

У цьому розділі розглядається важливий етап в розробці програмного додатку – визначення функціональних вимог. Детальне розуміння потреб користувачів та ключових функцій додатку є визначальним етапом, який слід враховувати при подальшій розробці. У цьому розділі будуть представлені визначені вимоги та описана функціональність програмного додатку [15].

3.1.1. Мета та завдання

Метою розроблюваного додатку є створення зручного та ефективного інструменту для відображення тривимірних об'єктів, представлених у форматі *OBJ*. Додаток має застосовуватися в галузях комп'ютерної графіки, дизайну та інших областях, де візуалізація *3D*-об'єктів є важливою.

Завданнями додатку є:

– забезпечити можливість завантаження та відображення *OBJ* файлів;

– забезпечити інтерактивні можливості взаємодії з відображеною *3D*-сценою;

– реалізувати елементи управління для зміни ракурсу та налаштувань відображення.

– забезпечити можливість керування освітленням на *3D*-сцені;

– забезпечити можливість аналізування об'єктів та їх параметрів;

– забезпечити можливість асинхронного завантаження декількох *3D*-моделей.

3.1.2. Функціональні вимоги

До функціональних вимог програмного додатку відносяться:

– завантаження файлів (користувач повинен мати можливість завантажити *OBJ* файли в додаток);

– відображення *3D*-сцени (система повинна ефективно відображати *3D*-об'єкти на екрані з можливістю їхньої взаємодії);

– управління камерою (користувач повинен мати зручні елементи управління для зміни положення та ракурсу камери);

– вибір об'єктів (користувач повинен мати можливість виділяти та взаємодіяти з окремими об'єктами на сцені);

– видалення об’єктів (користувач повинен мати змогу видаляти об’єкти зі сцени);

– маніпуляції над об’єктом (користувач повинен мати змогу повертати, переміщувати та віддаляти об’єкт за потреби).

3.1.3. Нефункціональні вимоги

До функціональних вимог програмного додатку відносяться:

– ефективність (додаток повинен працювати ефективно навіть з великими об'ємами даних);

– масштабованість (додаток повинен зберігати максимальну продуктивність під час завантаження великої кількості файлів);

– сумісність (додаток має підтримувати *Windows*, *UNIX* та *Mac* операційні системи);

– надійність (додаток має працювати без збоїв у системі);

– зручність (інтерфейс повинен бути зрозумілим та зручним для користувача);

– локалізація (додаток повинен мати можливість відображати інтерфейс на різних мовах).

3.1.4. Структура

Додаток структуровано у відповідності до концепції *Model-View-Controller (MVC)*, де різні компоненти будуть взаємодіяти між собою для досягнення ефективної та розширюваної архітектури (рис. 3.1).

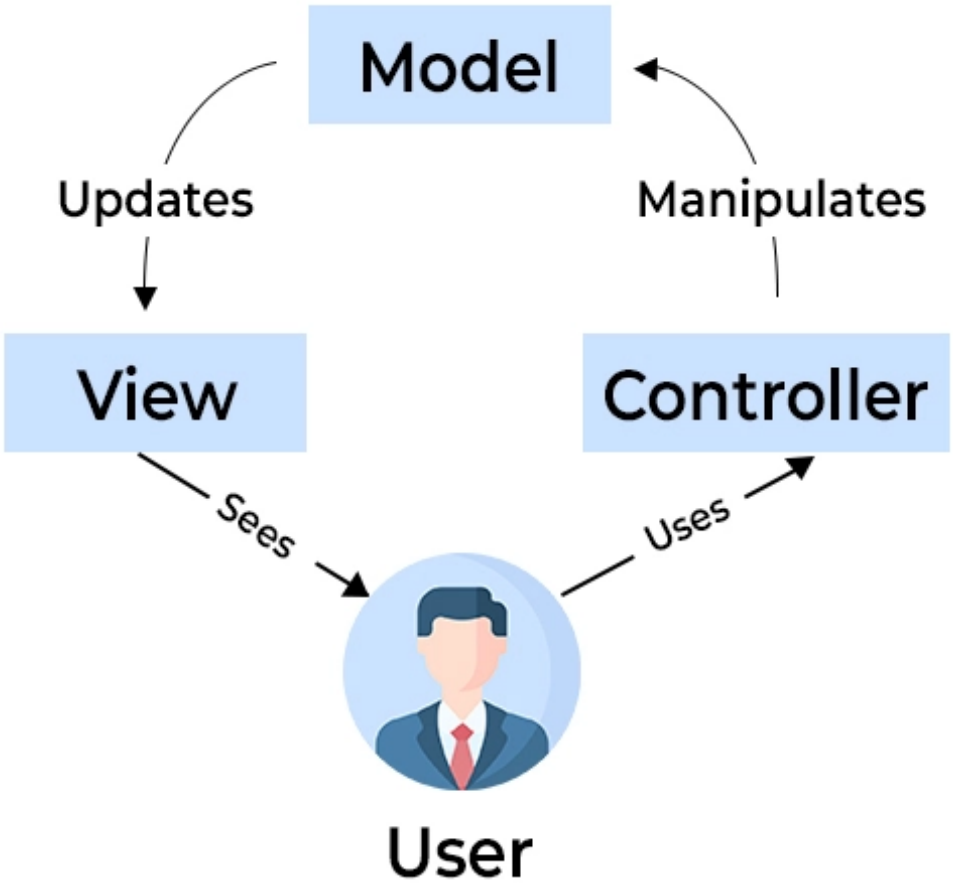


Рис. 3.1. Схема архітектури *MVC*

У шаблоні архітектури *MVC (Model-View-Controller)*, як видно з назви, визначаються три основні компоненти: модель, представлення і контролер.

Представлення відповідає за відображення інформації, яка надходить у систему або виходить з неї. Модель представляє собою «суть» системи і відповідає за внутрішні алгоритми та розрахунки.

Контролер виступає як зв'язок між представленням і моделлю, дозволяючи їм взаємодіяти. Контролер отримує дані від користувача і передає їх моделі, а також отримує повідомлення від моделі та передає їх представленню.

У відношенні до інтернет-додатків існує погляд, що деякі частини контролера і представлення об'єднані через відображення та введення інформації, які здійснюються браузером. Зокрема, це стосується відображення та одночасного введення інформації через браузер [12].

Конкретизація компонентів наведена нижче.

– Представлення це модуль для відображення інформації. В додатку представленням є графічний інтерфейс, що забезпечує візуалізацію та взаємодію із користувачем. Він включає в себе елементи управління такі як: сцену, кнопки, поля введення, поля тексту та інші.

– Контролер це модуль для управління введенням і виведенням даних. Контролер визначає типи даних, отриманих від моделі, передає їх у представлення і приймає дані від користувача для подальшої передачі в модель. В додатку контролером виступає логіка, що включає обробник подій та взаємодії між моделлю та представленням, а також обробку та введення користувача, що забезпечує реакцію на дії користувача та взаємодію з іншими компонентами.

– Модель це модуль, який здійснює розрахунки на основі введених від користувача даних і передає результати в контролер. Результат не повинен містити інформацію, що стосується відображення. В додатку моделлю є бібліотека *CGAL*, за допомогою якої буде відбуватися: зчитування та обробка вхідних файлів; конструювання поверхневої сітки, яка відображатиметься на сцені.

3.2. Діаграма компонентів

Діаграма компонентів – це один з видів структурних діаграм у мові моделювання *Unified Modeling Language* (*UML*), яка використовується для візуалізації та опису архітектури системи або програмного додатку за допомогою її компонентів та їх взаємодії.

Діаграма компонентів вказує на структурні елементи системи та зв'язки між ними. Кожен компонент представляє фізичну або логічну частину системи, яка може бути незалежно розглядати, заміняти або покращувати. Компоненти можуть включати програмні модулі, бібліотеки, файли, об'єкти, апаратне забезпечення та інші структурні одиниці.

Діаграма включає в себе такі елементи:

– компоненти, що представляють фізичні або логічні частини системи (програмний код, бібліотеки, модулі, файли, або інші важливі частини системи), які

можна згрупувати та ідентифікувати, що дозволяє аналізувати та організувати їх взаємодію;

– інтерфейси, що вказують на взаємодію між компонентами або як користувачі можуть взаємодіяти із компонентами та описують методи або служби, які один компонент може використовувати або надавати для інших компонентів;

– залежності, що вказують на взаємозалежність між компонентами, тобто які компоненти потрібні для роботи інших, що допомагає визначити порядок завантаження та виконання компонентів та їх взаємні відносини.

Діаграми компонентів використовуються для полегшення розуміння структури системи та розподілу функціональності між різними частинами програми чи апаратного забезпечення. Ці діаграми можуть бути корисним інструментом для комунікації між розробниками та іншими учасниками проєкту [16].

Діаграму компонентів для програмного додатку, що працює із *3D*-даними зображено нижче (рис. 3.2).

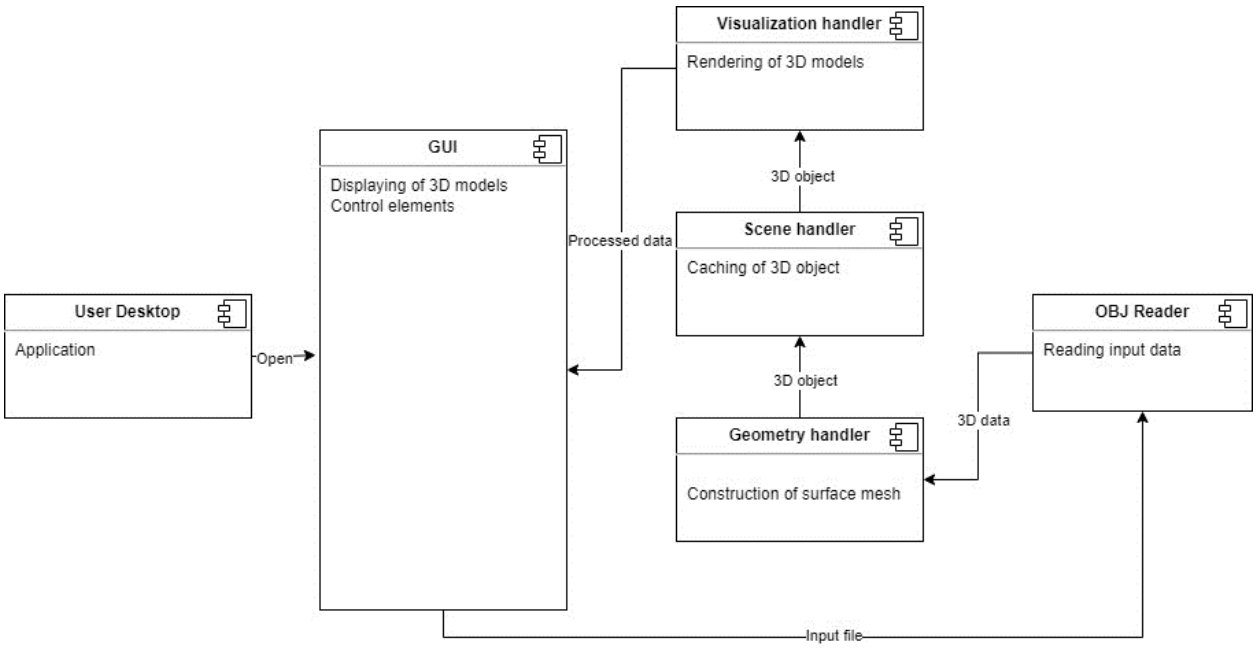


Рис. 3.2. Діаграма компонентів програмного додатку

Наведена діаграма містить такі компоненти:

– *User Desktop* це компонент, який представляє собою настільний комп’ютер користувача, де встановлений додаток;

– *GUI* це компонент графічного інтерфейсу користувача;

– *Obj Reader* це компонент, за допомогою якого відбувається зчитування вхідного файлу формату *.OBJ;*

– *Geometry Handler* це компонент, за допомогою якого відбувається конструювання *3D*-об’єкту;

– *Scene handler* це компонент, за допомогою якого відбувається кешування *3D*-об’єкту для подальшого використання без необхідності зчитування;

– *Visualization handler* це компонент, що представляє собою бібліотеку *OpenGL* та необхідний для візуалізації *3D*-об’єкту.

3.3. Діаграма прецедентів

Діаграма прецедентів (*Use Case Diagram*) є однією з ключових складових моделювання системи за допомогою мови *UML (Unified Modeling Language).* Вона дозволяє аналізувати та визначати взаємодію між зовнішніми сутностями (акторами) та функціональністю системи (прецедентами). Хоча діаграма прецедентів сама по собі не призначена для проєктування інтерфейсу, але вона може надавати цінний контекст для його розробки. Інтерфейс користувача визначається тим, як користувачі взаємодіють із системою. При аналізі діаграми прецедентів можна виявити основні функції та взаємодії, які повинні бути підтримані інтерфейсом [17].

Ось деякі унікальні аспекти діаграми прецедентів:

– ідентифікація ключових взаємодій, яка дозволяє ідентифікувати ключові взаємодії між користувачами та системою, що є важливим етапом при визначенні вимог до системи;

– акцент на зовнішніх сутностях (акторах), що підкреслює важливість взаємодії системи з оточуючим середовищем та екосистемою користувачів;

– узагальнення функціональності на високому рівні, що полегшує зрозуміння великого обсягу інформації для зацікавлених сторін;

– визначення сценаріїв забезпечує моделювати сценарії взаємодії між користувачами та системою, що становить основу для подальшого розроблення функціональності;

– орієнтовність на взаємодію є відмінною особливістю, яка полягає в тому, що діаграма прецедентів концентрується на взаємодії зовнішніх сутностей і їх впливі на систему, виокремлюючи ключові аспекти з точки зору користувачів;

– основа для визначення вимог, що забезпечує визначення функціональних вимог до до системи, які можуть слугувати фундаментом для подальших етапів розробки, таких як проєктування та реалізація.

Діаграму прецедентів до програмного додатку, що працює із *3D*-даними, зображено нижче (рис. 3.3).

Задіяні актори:

– «користувач» це актор, який користується додатком;

– «система» це актор, який обробляє події, викликані користувачем та проводить маніпуляції в разі завантаження моделі.

У «користувача» є можливість відкриття додатку, що в свою чергу дозволить завантажувати *3D*-моделі та маніпулювати камерою, встановленої на *3D*-сцені, включаючи такі операції, як: переміщення та масштабування. Завантажування моделі використовує такі виклики «системи», як: відкриття/парсинг/закриття вхідного файлу, конструювання поверхневої сітки, що включає триангуляцію та валідацію результуючого об’єкту, рендеринг *3D*-об’єкту, обробник помилок. В разі успішного завантаження моделі, «користувач» матиме змогу: побачити візуалізований *3D*-об’єкт; змінити режим малювання об’єкту (каркас або твердість), видимість об’єкту та матеріал, що використовується для рендерингу; побачити інформацію про відкритий *3D*-об’єкт, що включає в себе: кількість вершин, кількість ребер, кількість граней, довжину, ширину, висоту, унікальний номер та ім’я; маніпулювати об’єктом за допомогою поворотів та переміщення; скинути всі зміни до стандартних налаштувань; видалити об’єкт із програми із вивільненням охопленої пам’яті.

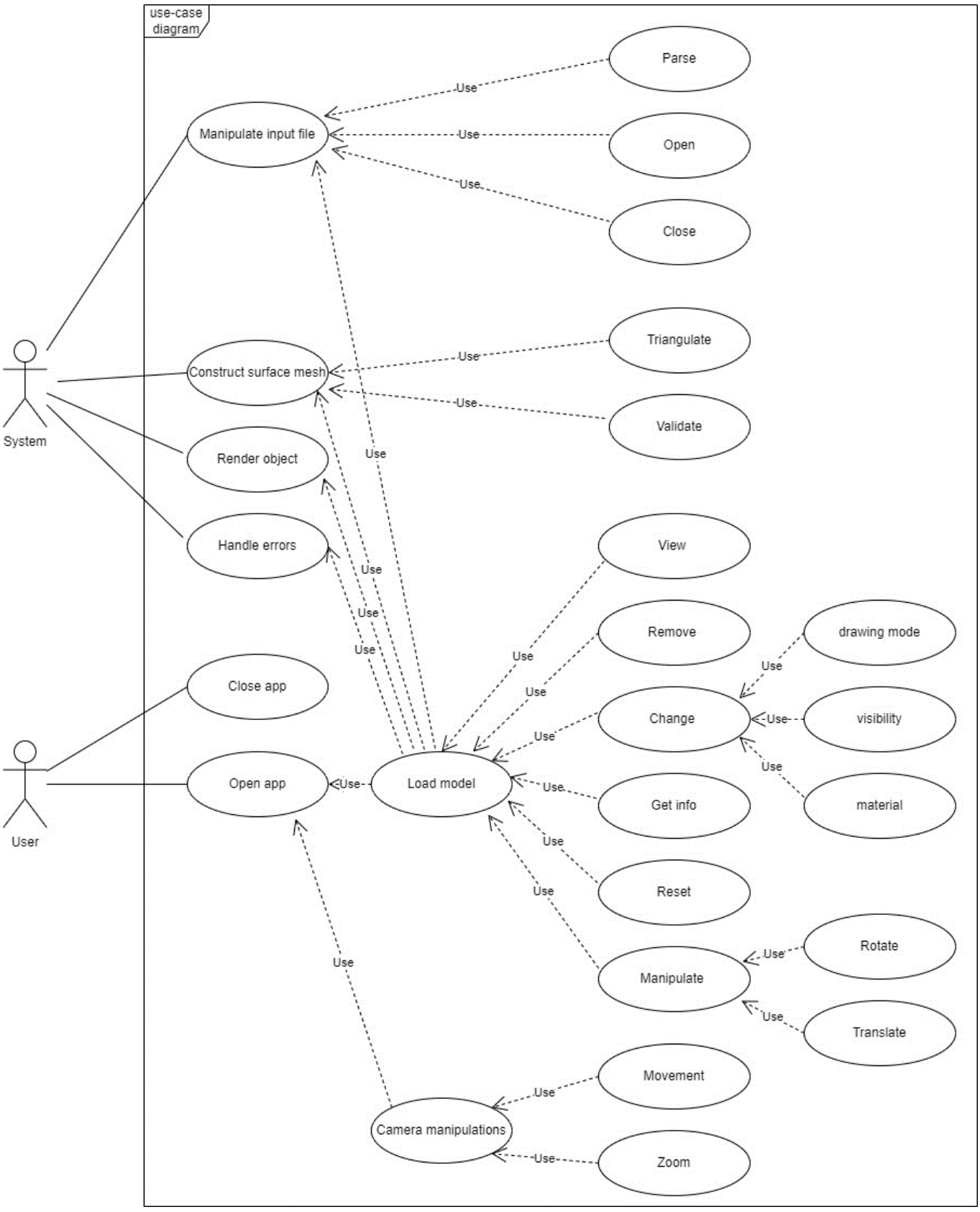


Рис. 3.3. Діаграма прецедентів програмного додатку

3.4. Блок-схема

Блок-схема – це графічне представлення послідовності операцій або процесу за допомогою стандартизованих геометричних фігур та стрілок. Кожен блок представляє конкретну операцію, дію або обробку даних, а стрілки вказують напрямок потоку управління або передачі даних між блоками. Блок-схеми широко використовуються в програмуванні, інженерії, бізнес-аналітиці, управлінському консалтингу та інших галузях для візуалізації, розуміння та документування алгоритмів, процесів чи систем.

Переваги блок-схеми:

– простота сприйняття (легко розуміється та використовується особами без глибоких технічних знань);

– ефективність (дозволяє швидко виявляти та вирішувати проблеми у великих системах чи процесах);

– інтуїтивність (зручна для комунікації між різними учасниками проєкту);

– структурованість (дозволяють структурувати інформацію в логічний порядок, допомагаючи визначити послідовність кроків).

Недоліки блок-схеми:

– обмежена придатність для складних алгоритмів та високе деталювання (для дуже складних або деталізованих алгоритмів блок-схеми можуть стати громіздкими та важкими для розуміння);

– неефективність для текстової інформації (не завжди ефективні для відображення текстової інформації або деталей, що не піддаються графічному представленню);

– можливість непорозуміння взаємозв'язків (іноді взаємозв'язки між блоками можуть бути неправильно зрозумілі або призвести до непорозумінь).

Нижче зображено блок-схему для програмного додатку, що працює із *3D*-даними (рис. 3.4).

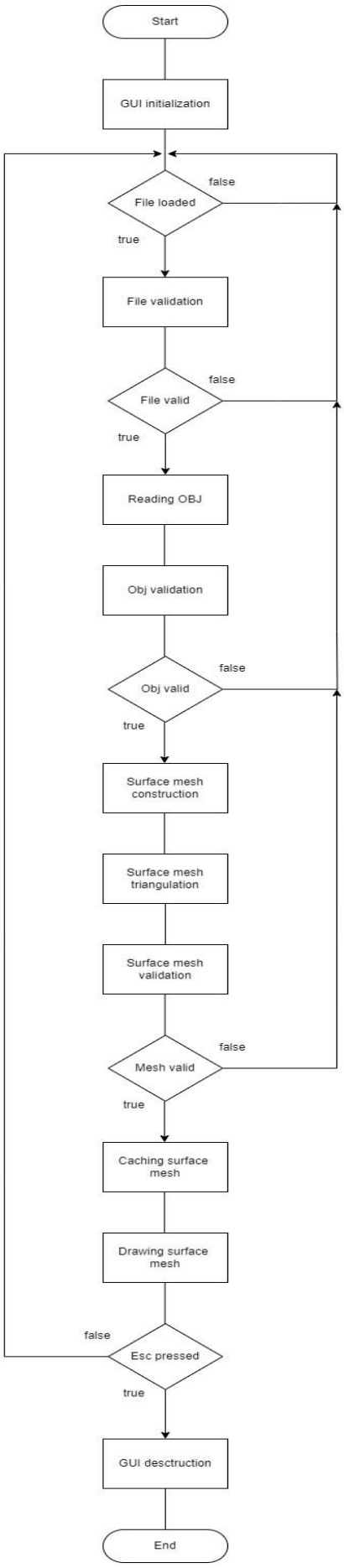


Рис. 3.4. Блок-схема програмного додатку

Висновки за розділом

Архітектура програмного забезпечення представляє собою набір структур, необхідних для врахування системи програмного забезпечення та процес створення таких структур і систем. Кожна структура включає елементи програмного забезпечення, їх взаємозв'язки та властивості як елементів, так і взаємозв'язків. Основна мета архітектури програмного забезпечення – прийняття фундаментальних структурних рішень, які буде важко змінити після впровадження**.**

Програмний додаток структуровано у відповідності до концепції *Model-View-Controller (MVC)*, де різні компоненти взаємодіють між собою для досягнення ефективної та розширюваної архітектури.

Діаграма компонентів – це один з видів структурних діаграм у мові моделювання *Unified Modeling Language* (*UML*), яка використовується для візуалізації та опису архітектури системи або програмного додатку за допомогою її компонентів та їх взаємодії. Діаграма компонентів вказує на структурні елементи системи та зв'язки між ними. Кожен компонент представляє фізичну або логічну частину системи, яка може бути незалежно розглядати, заміняти або покращувати.

Діаграма прецедентів (*Use Case Diagram*) є однією з ключових складових моделювання системи за допомогою мови *UML*. Вона дозволяє аналізувати та визначати взаємодію між зовнішніми сутностями (акторами) та функціональністю системи (прецедентами). Хоча діаграма прецедентів сама по собі не призначена для проєктування інтерфейсу, але вона може надавати цінний контекст для його розробки.

Блок-схема – це графічне представлення послідовності операцій або процесу за допомогою стандартизованих геометричних фігур та стрілок.

РОЗДІЛ 4
ВІЗУАЛІЗАЦІЯ ТА ОБРОБКА 3D-МОДЕЛЕЙ

Розділ представляє ключовий етап дослідження, спрямованого на створення ефективного інтерфейсу для взаємодії з тривимірними об'єктами у програмному додатку. У розділі розглянуто: структуру форматів файлів для описання геометрії; аспекти візуалізації, які визначають вигляд та поведінку *3D*-моделей; алгоритми бібліотеки *CGAL* для конструювання поверхневої сітки, що визначає форму та структуру віртуальних об'єктів та графічний конвеєр.

4.1. Геометрія *3D*-моделей

Геометрія *3D*-моделей включає в себе опис просторових об'єктів за допомогою точок, векторів та поверхонь. У геометрії *3D*-моделей використовуються різні математичні підходи, такі як векторна алгебра, матриці, криві та поверхні Без'є. Ці методи дозволяють точно представляти об'єкти та їх просторові взаємозв'язки. Застосування геометрії *3D*-моделей у сучасному комп'ютерному середовищі розширюється від відтворення фізично реалістичних об'єктів до візуалізації складних даних та інтерактивної взаємодії з віртуальними середовищами. Перш за все, потрібно ознайомиться із базовими поняттями, що постійно використовуються для роботи із *3D*-моделями.

4.1.1. Вершина

У комп'ютерній графіці термін «вершина» використовується для позначення точки або вузла в тривимірному або двовимірному просторі. Вершини є базовими будівельними блоками для створення графічних об'єктів, таких як моделі *3D*-об'єктів чи форми в двовимірному просторі. У тривимірній графіці вершини часто використовуються для визначення позицій точок у просторі *XYZ* (або *XYZW* в деяких випадках). Кожна вершина може також мати інші атрибути, такі як: позиція (координати вершини у просторі), колір (зазвичай дифузні або дзеркальні значення

RGB, що представляють колір поверхні або попередньо обчислену інформацію про освітлення), текстурні координати (*UV*-координати, що керують відображенням текстури поверхні), нормалі (визначають наближену криву поверхню в місці розташування вершини, яка використовується для розрахунків освітлення) і так далі. При створенні графічних об'єктів, таких як тривимірні моделі, вершини зазвичай з'єднуються для створення граней, які утворюють поверхню об'єкта. Ці грані можуть мати текстури, матеріали та інші властивості, що визначають зовнішній вигляд об'єкта. Таким чином, вершина є фундаментальним елементом при роботі з комп'ютерною графікою, і вона допомагає визначати геометричну структуру та вигляд графічних об'єктів [18].

4.1.2. Текстура та її розгортка

Текстура в контексті комп'ютерної графіки – це зображення або зразок, який використовується для покриття поверхні *3D*-моделі чи *2D*-графічного об'єкта. Текстури додають деталі та візуальні ефекти до об'єктів, роблячи їх більш реалістичними та привабливими. Основні аспекти текстур включають: зображення (фотографію, малюнки, графічні патерни, тощо), текстурні координати, *UV*-розгортку, розмір та роздільність, тип (дифузні, нормальні, висотні) та дисплейні картки.

Текстурні координати – це пари чисел (зазвичай два або три), які вказують на конкретну точку на текстурі, пов'язаній з вершиною графічного об'єкта. Вони використовуються в комп'ютерній графіці для визначення того, як текстура повинна бути накладена на графічний об'єкт або його поверхню. Коли вершини графічного об'єкта описуються, до кожної вершини можуть бути прикріплені текстурні координати. Зазвичай ці координати лежать в діапазоні від 0 до 1, де (0, 0) вказує на верхній лівий кут текстури, а (1,1) – на нижній правий. При рендерингу графічного об'єкта, програмне забезпечення або апаратне забезпечення графіки використовує ці текстурні координати, щоб визначити, яку частину текстури використовувати для кожної точки на поверхні об'єкта. Це дозволяє накладати різні текстури на різні частини об'єкта і визначати, як текстура буде розташована та виглядати на ньому. Текстурні координати використовуються для створення реалістичних зображень і

можуть бути використані для ефектів, таких як: розгортка текстур, оточуюче освітлення та інші аспекти візуалізації в комп'ютерній графіці. Нижче зображено накладання текстури на *3D*-модель (рис. 4.1).

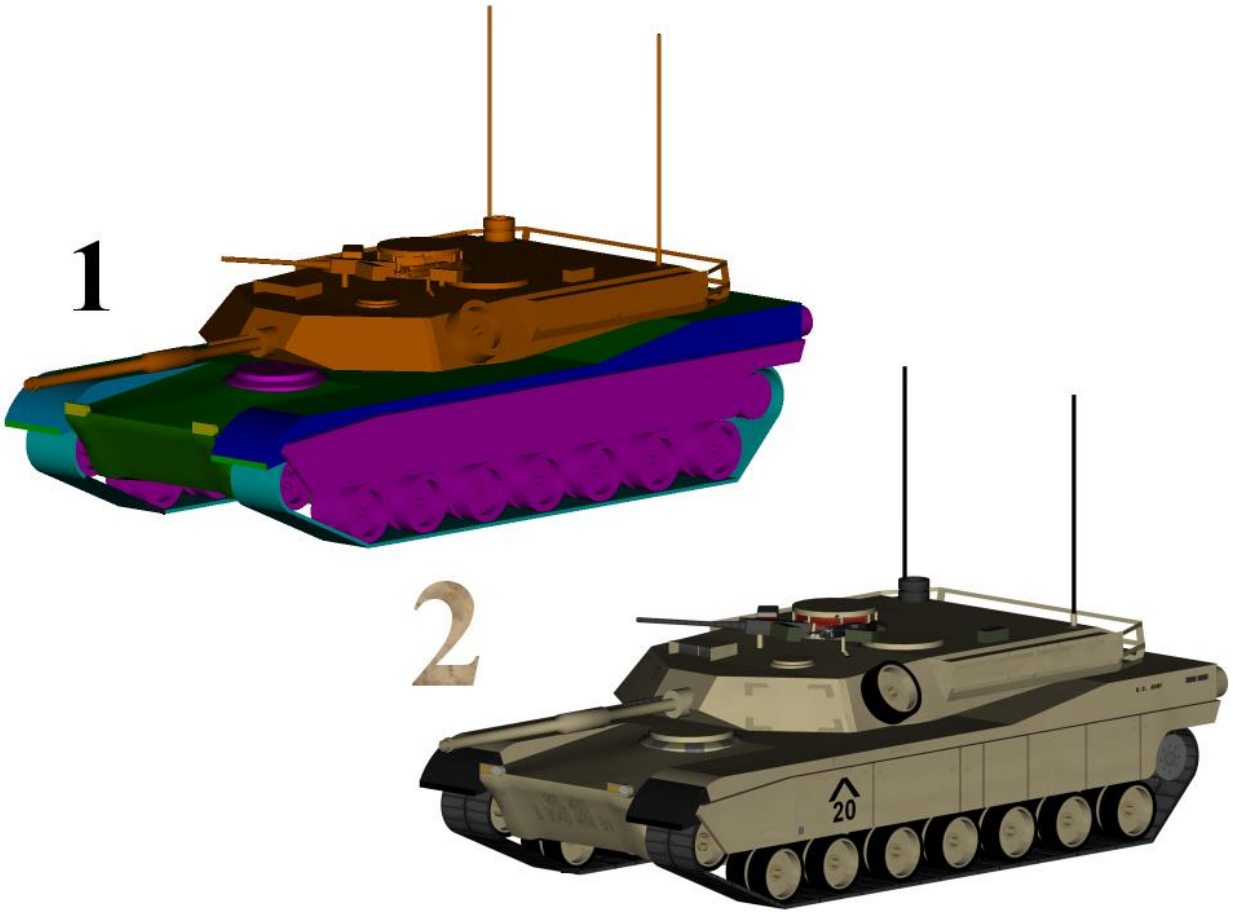


Рис. 4.1. Вигляд *3D*-моделі із текстурою та без неї

UV-mapping (розгортка текстури) – це процес створення відповідності між точками на поверхні *3D*-моделі та їхніми текстурними координатами (часто позначаються як *U* та *V*). *UV-mapping* описує, які частини текстури відповідають кожній частині поверхні моделі. Також ця техніка включає в себе визначення текстурних координат для кожної точки на поверхні об'єкта так, щоб текстура належним чином накладалася на модель при рендерингу [19]. У контексті *3D*-моделювання та графіки, термін використовується для опису процесу призначення текстурних координат вершинам моделі. Отже, *UV-mapping* – це процес, а текстурні координати – це конкретні значення, які отримують точки моделі під час цього процесу (рис. 4.2).

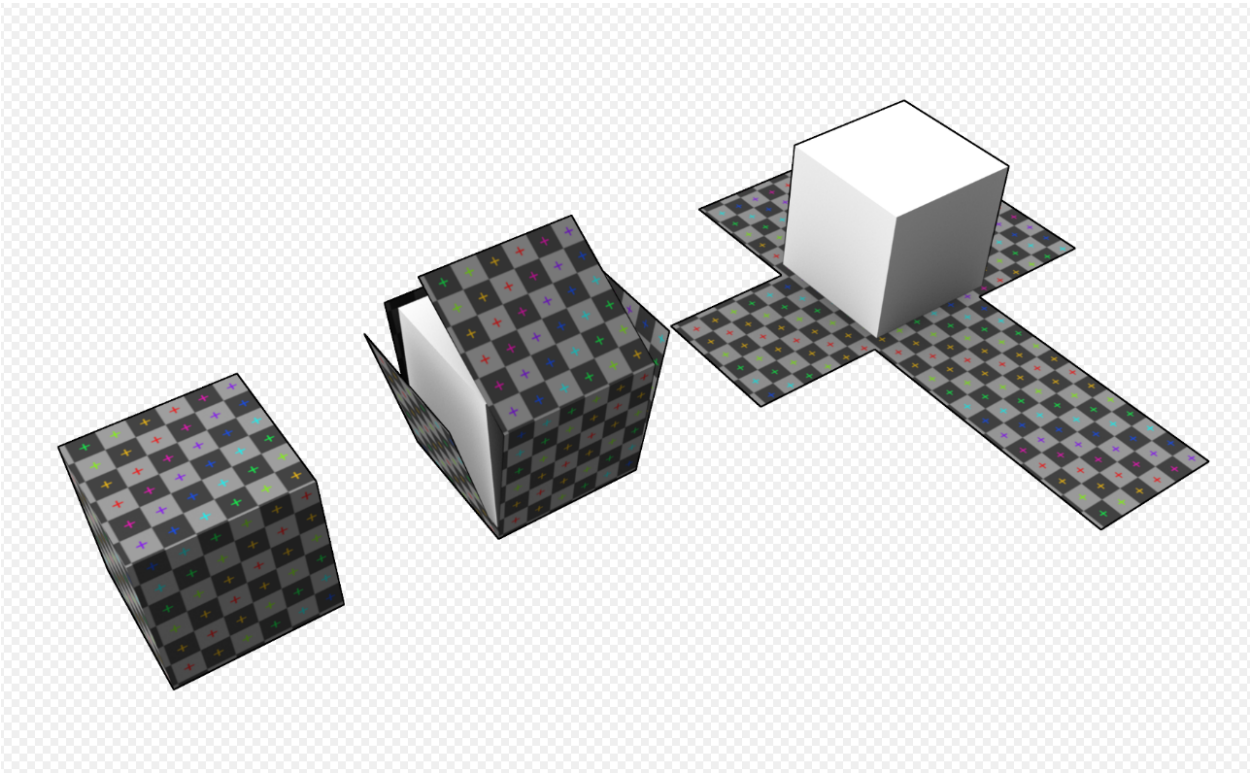


Рис. 4.2. Приклад накладання текстурної розгортки на куб

Існує кілька факторів, які визначають якість процесу *UV-mapping*, які іноді можуть взаємно конфліктувати.

– Максимальне використання площі текстури. Хоча деякі ситуації вимагають повного використання площі текстури, існує конфлікт між максимальною деталізацією та системними обмеженнями, що може потребувати зниження розмірів текстур на краях.

– Відсутність дисбалансу у деталізації. Важливо уникати областей на текстурі, де деталізація занадто велика чи, навпаки, недостатня.

– Відсутність геометричних спотворень. Необхідно уникати областей на текстурі, де геометричні артефакти виглядають аномально чи викривлено.

– Подібність до стандартних ракурсів. Зручно, коли розгортка текстури відображає об'єкт з позицій, які часто використовуються при його створенні чи фотографуванні.

– Вдале розташування «швів». «Шви», які відповідають ребрам об'єкта, можуть бути корисними для натурального розриву поверхні, але вони повинні бути розташовані ретельно.

– Часткова симетричність об'єктів. Важливо досягти збалансованого поєднання симетричних та асиметричних ділянок розгортання, оскільки симетрія поліпшує деталізацію, а асиметрія надає об'єкту життєвості.

4.1.3. Нормаль

Нормаль в комп'ютерній графіці – це вектор, який перпендикулярний до поверхні геометричного об'єкта. Кожна точка на поверхні об'єкта може мати свою нормаль, і цей вектор вказує на напрямок від поверхні в зовнішній простір (рис. 4.3).

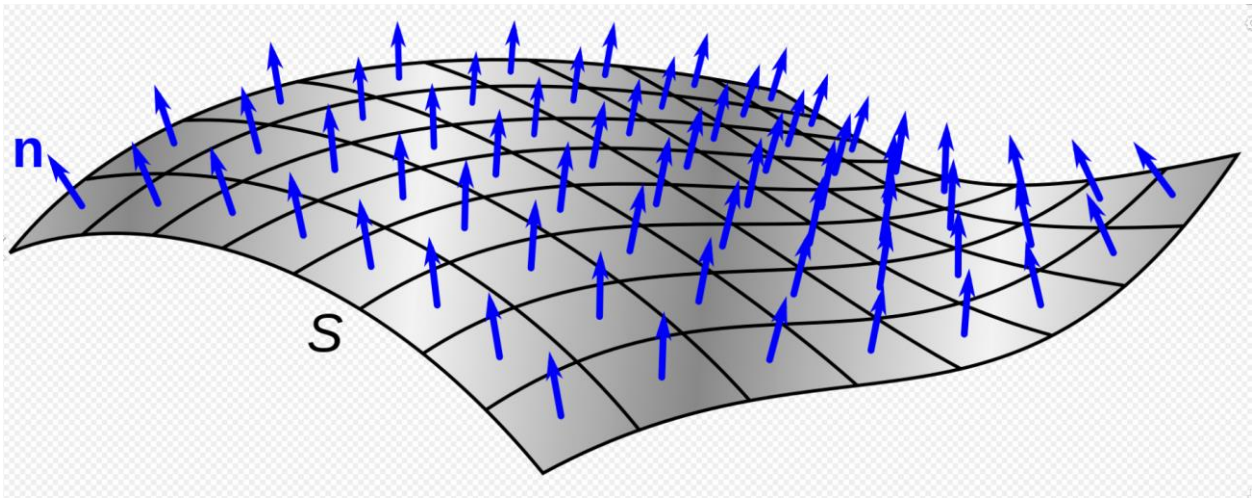


Рис. 4.3. Вектори нормалей у вершинах поверхні

Використовуючи нормалі, можна вирішувати різні завдання в комп'ютерній графіці, зокрема в області обчислення освітленості та визначення тіней. Основна ідея полягає в тому, що, знаючи напрямок нормалі в кожній точці поверхні, можна визначити, як світло взаємодіє з цією поверхнею. Наприклад, для обчислення освітленості в конкретній точці можна використовувати відношення між напрямком нормалі та напрямком світла. Це дозволяє визначити, наскільки поверхня направлена до або від світла, і враховувати це при рендерингу зображення. Нормалі також використовуються при визначенні ефектів відбиття та розсіювання світла, а також при вирішенні задач реалістичного моделювання поверхні об'єктів. У тривимірній графіці нормалі є важливим елементом для досягнення реалістичного візуального вигляду графічних об'єктів [20].

4.2. Формат *OBJ*

Формат *OBJ* – це формат, який визначає геометрію *3D*-поверхні одного чи кількох об'єктів. Перший раз цей формат був використаний компанією *Wavefront Technologies* близько 1990 року, і була опублікована специфікація для підтримки взаємодії. У вступі до специфікації зазначалося: «Файли об'єктів визначають геометрію та інші властивості для об'єктів *Advanced Visualizer* від *Wavefront*. Файли об'єктів також можуть використовуватися для передачі геометричних даних між *Advanced Visualizer* та іншими додатками». З середини 1990-х років формат на основі *ASCII* розглядається як вендор-нейтральний і називається «*Wavefront OBJ*», «*Alias/Wavefront OBJ*» або просто «*OBJ*». У 2020 році цей формат залишається широко використовуваним, особливо для *3D*-друку об'єктів різних кольорів. Зауваження: оригінальна специфікація файлу *OBJ* використовує пов'язаний файл бібліотеки матеріалів *Wavefront (MTL)* для визначення кольорів і текстур. Текстури та кольори, вказані іменами в файлі *MTL* та викликаються в файлі *OBJ*.

Формат файлу *OBJ* підтримує визначення геометрії поверхонь об'єктів за допомогою полігональних сіток або за допомогою вільних кривих та поверхонь. Логічною концепцією в моделі *OBJ* є «елемент». У специфікації перераховані такі елементи: точка (*p*), лінія (*l*), грань (*f*), крива (*curv*), *2D*-крива (*curv2*) та поверхня (*surf*). Елементи створюються з вершин; вершини повинні бути перераховані першими в файлі, щоб елементи могли посилатися на них за номером, використовуючи порядок їх появи в файлі. Елементи можуть опціонально організовуватися в групи для зручності або для керування застосунками відображення.

Формат файлу *OBJ* має просту фізичну структуру, що складається з рядків, що починаються з ключових слів. Після кожного ключового слова вказуються відповідні параметри та значення. Рядки, які починаються з «#» – є коментарями. Довгі рядки можна розбити, використовуючи символ зворотнього слеша в кінці рядків для продовження. Згідно з описом формату файлу *Wavefront OBJ* у енциклопедії форматів графічних файлів: «Найчастіше зустрічаються файли *OBJ*, що містять лише

полігональні грані. Щоб описати полігон, файл спершу описує кожну точку (вершину) за допомогою ключового слова «*v*», а потім описує грань за допомогою ключового слова «*f*». Рядок команди грані містить нумерацію точок у грані, що представляють індекси точок із списку, в порядку їх входження в файл». Не всі програми *3D*-програмного забезпечення можуть обробляти поверхні вільної форми, які визначаються математичними рівняннями. З тих, що можуть, підтримка може бути обмеженою до неоднорідних раціональних *B*-сплайнів (*NURBS*), хоча специфікація *OBJ* дозволяє різні моделі.

Нижче приведено простий приклад файлу *OBJ* для куба, взятий зі специфікації (рис. 4.4).

```
v 0.000000 2.000000 2.000000
v 0.000000 0.000000 2.000000
v 2.000000 0.000000 2.000000
v 2.000000 2.000000 2.000000
v 0.000000 2.000000 0.000000
v 0.000000 0.000000 0.000000
v 2.000000 0.000000 0.000000
v 2.000000 2.000000 0.000000
f 1 2 3 4
f 8 7 6 5
f 4 3 7 8
f 5 1 4 8
f 5 6 2 1
f 2 6 7 3
```

Рис. 4.4. Приклад файлу *OBJ* для відображення куба

Вершини для полігональної моделі в файлі *OBJ* можуть мати три типи даних:

– «*v*» визначає положення вершини в трьох вимірах (*x, y, z*) (обов’язково три числа з плаваючою комою);

– «v*n*» є нормаллю вершини, напрямний вектор, пов'язаний з вершиною, використовується для забезпечення плавного затінення (опціонально три числа з плаваючою комою);

– «*vt*» є координатою текстури, також відомою як *UV*-координат, які використовуються під час відображення для визначення того, як розфарбувати тривимірну поверхню за допомогою пікселів з текстурної карти *2D,* наприклад, зображення у форматі *PNG* (зазвичай два числа з рухомою комою).

Файли *OBJ*, що містять лише елементи граней та пов'язані дані вершин, широко підтримуються програмами для обробки *3D*-об’єктів, як для імпорту, так і для експорту. Тим не менше, можуть виникати проблеми взаємодії при застосуванні кольорів та текстур за допомогою пов'язаного файлу *MTL*. Файл *OBJ* звертається до файлу *MTL* у вказаній декларації «*mtllib*», а визначення об'єктів (полігональних чи вільних форм) у файлі *OBJ* може посилатися на конкретне ім'я матеріалу з використанням ключового слова «*usemtl*». Деякі програми просто не підтримують файли *MTL*. Початкова програма *Advanced Visualizer* вимагала, щоб *файл MTL* був в тому ж каталозі, що й файл *OBJ*, і файли карт текстур також були в тому ж каталозі. Деякі програми приймають шляхи до файлів в інших каталогах, інші експортують складові файли в окремі каталоги, і користувачам слід переміщати їх перед успішним використанням. Ще одним потенційним джерелом проблем взаємодії з файлами *MTL* є використання ключових слів, які не визначені в оригінальній специфікації *MTL,* для власного використання або для підтримки нових підходів до *3D*-рендерингу.

Кілька програм підтримують конвенцію представлення кольорів в моделі *OBJ* без використання файлу *MTL*. Ця конвенція підтримує техніку, відому як колірування на вершинах, в якій кожній вершині присвоюється колір, а пікселі на поверхні меша розфарбовуються шляхом інтерполяції на основі відстані від вершин поверхні. Цей підхід особливо корисний для деяких застосувань. Коли *3D*-моделі створюються шляхом сканування (наприклад, для патологічного зразка) або фотограмметрії (наприклад, для артефакту культурної спадщини у *3D*), можливо створити один файл, який включає точну інформацію про кольори. Ще однією ситуацією, коли цей підхід є цінним, є моделі, які використовують базові кольори для ідентифікації частин моделі дизайну або передачі іншої інформації за допомогою штучних кольорів, наприклад, для анатомічної моделі. Наявність одного файлу може спростити робочі процеси та зберігання в ситуаціях, коли карти текстур не потрібні. Формат *OBJ* може

бути використаний як формат початкового стану, створений за допомогою процесу *3D*-сканування. Часто він служить як формат середнього стану, створений як вихід з програмного забезпечення, що підтримує *3D*-дизайн. Модель *OBJ* часто використовується в процесі *3D*-друку або для подальшого використання в більш розвинених програмах *3D*-моделювання для включення в *3D*-сцену чи анімацію. Моделі часто обмінюються для подальшого використання у форматі *OBJ*.

Формат файлу *OBJ* може кодувати геометрію поверхні *3D*-моделі, використовуючи багатокутну сітку та/або геометрію довільної форми, використовуючи будь-який із ряду методів визначення ділянок викривленої поверхні: базова матриця, крива Безьє, *B*-сплайн, кардинальний сплайн або ряд Тейлора. Файл *OBJ* може визначати кілька об’єктів, але не сцену, яка включає положення освітлення або попередньо визначені положення перегляду. Також формат *OBJ* не підтримує попередньо визначені анімаційні послідовності або скелетні структури.

Файли *OBJ*, завдяки своїй структурі списку, можуть посилатися на вершини, нормалі тощо або за їх абсолютною позицією (1 представляє першу визначену вершину, *N* представляє *N*-у визначену вершину), або їх відносним положенням (-1 представляє останню визначену вершину). Однак не все програмне забезпечення підтримує останній підхід, і навпаки, деякі програми за своєю суттю записують лише останню форму (через зручність додавання елементів без необхідність перерахувати зміщення вершин тощо), що іноді призводить до несумісності, що є потенційною проблемою формату [21].

4.3. *3D*-сітка та її види

3D-сітка (*3D-mesh*) – це сукупність вершин, ребер і граней, які разом утворюють тривимірний об'єкт. Вершини – це координати в тривимірному просторі, ребра – з'єднують дві сусідні вершини, а грані (також називають полігонами) обгортають ребра, утворюючи поверхню об'єкта. Найбільш поширеними полігонами в *3D*-сітках є трикутники та чотирикутники. *3D*-сітка є фундаментальним компонентом тривимірної графіки, що представляє геометрію поверхні тривимірного об'єкта.

Фактично – це колекція вершин, ребер і граней, які визначають форму та структуру тривимірного об'єкта. За допомогою *3D*-сітки можна створювати моделі з витонченими деталями та текстурами, що робить їх реалістичними. Ця технологія також дозволяє маніпулювати цими об'єктами використовуючи техніки, які раніше були неможливими. Це робить їх невід'ємним інструментом для дизайнерів та інженерів, які повинні швидко і точно створювати складні моделі.

Меші є одними з найважливіших компонентів у тривимірному моделюванні. Основною ідеєю є те, що меш – це будь-який об'єкт, яким митець може маніпулювати, щоб створювати різні форми та образи. При створенні сітки, зазвичай першим кроком є створення форми. Можна уявити це як створення контуру або каркасу, який потім можна буде заповнити кольором і текстурою. Це має сенс, якщо уявляти собі, що один з способів заповнення чогось на папері – спочатку обвести його.

У тривимірному моделюванні використовуються три основні компоненти, які складають меш. Перший компонент – це вершини (*vertices*) сітки, другий – ребра (*edges*) сітки, і третій – грані (*faces*) сітки (рис. 4.5).

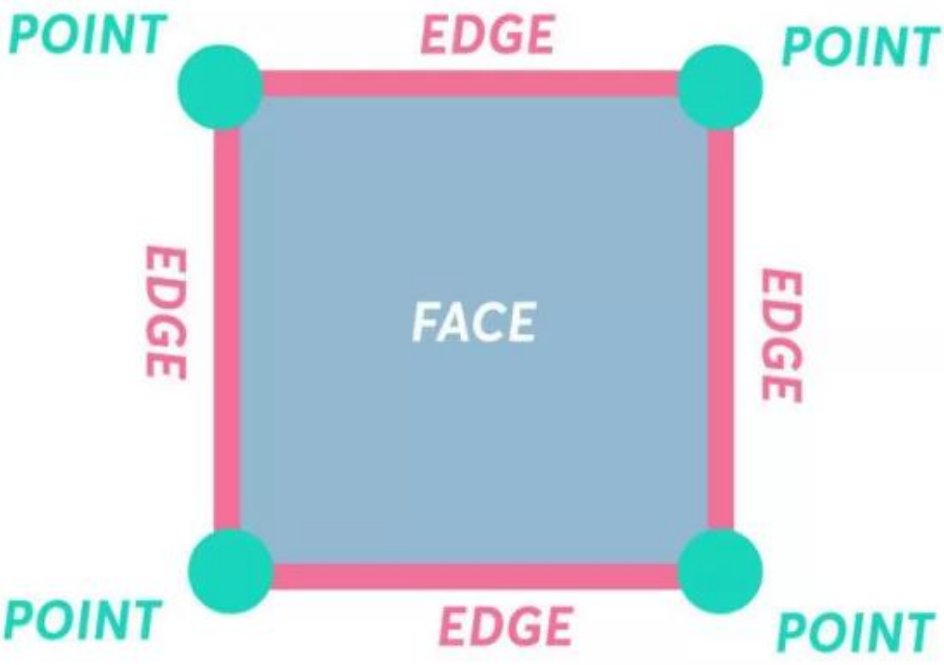


Рис. 4.5. Основні компоненти сітки

Вершини використовуються для створення форми об'єкта і їх можна з'єднати за допомогою ребер. Вершини сітки, зазвичай, представляють собою набір підказок, які складають зовнішній край. Вершини визначають форму або контур чого-небудь у тривимірному моделюванні, схоже на те, як ручка створює лінії на папері при обведенні форм. Якщо є більше однієї точки, існує можливість їх з'єднати, і це з'єднання потім називається ребром. Ребра потім з'єднуються в точках або близько до них, утворюючи грані, які складають *3D*-модель. Ребра можна відрізнити від точок тим, що точка часто представлена невеликою крапкою, а ребро найчастіше зображується як лінія. Подумайте про квадрат як приклад. У квадрата чотири точки. Між цими чотирма точками проведемо чотири ребра, які їх з'єднують. Тепер, коли є «замкнута» форма, можна сказати, що є також грань. У тривимірному моделюванні будь-яка замкнута форма чи полігон називається гранню. Щоб грань сформувалася в моделі, повинно бути більше трьох точок і трьох ребер. Таким чином, можна сказати, що найбільш простою формою грані в тривимірному моделюванні є трикутник. Отже, *3D*-сітка – це набір точок, ребер і граней, які визначають форму та структуру *3D*-об’єкта. Ці точки називаються вершинами, і вони з’єднані лініями, які називаються ребрами, утворюючи багатокутники, які називаються гранями, які потім використовуються для створення поверхні об’єкта.

У тривимірному моделюванні меш складається з граней у формі полігонів – зазвичай трикутників або квадратів, але також можуть бути шестикутники. Існує кілька типів тривимірних меш-моделей, які відрізняються способом їх створення та використанням. Нижче наведено кілька з найважливіших.

– Полігональні моделі складаються з полігонів, з'єднаних вершинами та ребрами. Полігональні моделі є найбільш поширеними меш-моделями в тривимірному моделюванні (рис. 4.6).

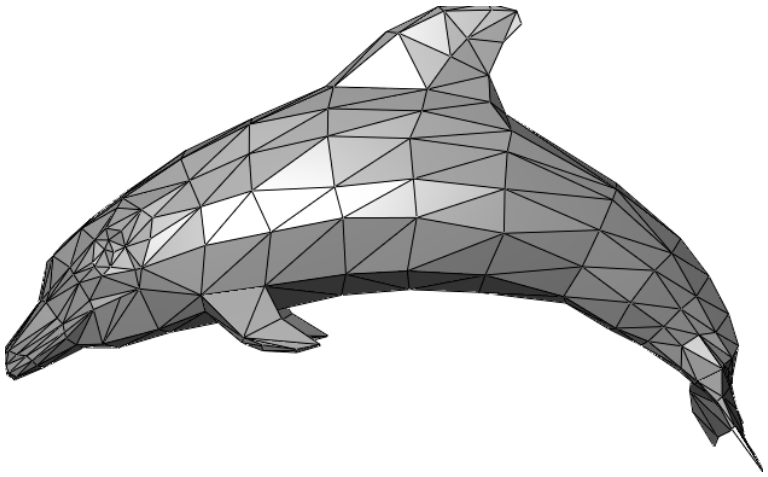


Рис. 4.6. Приклад трикутної полігональної меш-моделі

Якщо полігональний меш складається з трикутників (найпростіша форма мешу), то кажуть, що він має трикутну топологію. Якщо полігональний меш складається з квадратів замість трикутників, то кажуть, що він має квадратну топологію. Якщо він має кутову топологію, то він складається з полігонів з більше ніж чотирма ребрами. Частіше квадратна топологія виявляється кращою, ніж трикутна топологія. Якщо полігональна сітка складається з квадратів, то вона матиме парну кількість ребер. Це означає, що під час *3D*-друку поверхня має бути абсолютно плоскою для будь-якої заданої площини, щоб зберегти точність і якість. Якщо меш складається з квадратів, його також легше анімувати.

– *NURBS*-моделі (*NURBS* означає «*Non-Uniform Rational B-Spline*») є математичним представленням тривимірних об'єктів. Зазвичай використовується для створення кривих, які не можна виразити будь-яким іншим типом. Сітка *NURBS* – це сітка, яка містить криві. *NURBS*-моделі є більш плавними та гнучкими, ніж полігональні моделі, але вони також складніші та вимагають більше обчислювальних ресурсів.

– Моделі поверхні підрозділу – це комбінація багатокутників та *NURBS*-моделей. Вони використовують обидві техніки для створення деталізованих та гладких поверхонь. Частіше за все складаються з багатьох квадратів. Насправді вони складаються з багатьох квадратів. Це означає, що на площині поверхні є певна кількість спотворень (викривлення), що ускладнює точний і високоякісний *3D*-друк.

Після створення *3D*-моделі сітки наступним кроком часто є додавання текстур і матеріалів, щоб надати моделі реалістичного вигляду пізніше в *3D*-візуалізації. Текстури – це двовимірні зображення, які проєктуються на поверхню сітчастої моделі для додавання кольорів, візерунків і деталей. Матеріали, з іншого боку, визначають оптичні властивості поверхні, такі як відбиття, прозорість і шорсткість. Часто необхідно знайти баланс між деталізацією та продуктивністю тривимірної сітчастої моделі. У багатьох випадках для досягнення найкращої якості та точності спочатку створюється високополігональна модель. Потім цю модель можна оптимізувати та перетворити на модель з низьким полі шляхом зменшення кількості багатокутників без шкоди для надто високої якості зображення. Ця техніка, також відома як скорочення багатокутників або оптимізація сітки, є важливою частиною *3D*-моделювання та дозволяє створювати моделі ефективніше та з використанням менших ресурсів. Високополігональні моделі – це тривимірні сітчасті моделі з великою кількістю полігонів. Вони відрізняються високою деталізацією та реалістичністю візуалізації та часто використовуються в областях, де потрібна висока якість і точність зображення, наприклад у кіноіндустрії, архітектурній візуалізації або дизайні продуктів. Однак високополігональні моделі також потребують більше обчислень і вимагають більше ресурсів для візуалізації та відображення, що робить їх менш придатними для деяких програм, особливо коли продуктивність використовуваного апаратного забезпечення обмежена. Низько полігональні моделі – це тривимірні сітчасті моделі з невеликою кількістю полігонів. Зазвичай вони менш деталізовані та реалістичні, ніж високополігональні моделі, але також менш обчислювально-інтенсивніші та ресурсо-ефективніші. Низько полігональні моделі в основному використовуються в ігровій індустрії або онлайн-конфігураторах, де важлива апаратна продуктивність і швидке відтворення *3D*-об’єктів. Вони також часто використовуються для фонових елементів і об’єктів, які не знаходяться на передньому плані в сцені і тому потребують меншої деталізації.

В контексті даної роботи в програмі будуть використовуватися полігональні моделі із трикутною топологією. Дані моделі братимуться із об’єктного файлу, а конструювання відбуватиметься за допомогою класу *Surface_mesh* бібліотеки *CGAL*.

4.4. Структура даних для представлення полігональної моделі

Клас *Surface_mesh* є реалізацією структури даних напівребер і може використовуватися для представлення багатогранної поверхні. Це альтернатива пакетам *CGAL Halfedge Data Structures* і *3D Polyhedral Surface*. Основна відмінність полягає в тому, що він базується на індексах, а не на основі покажчиків. Крім того, механізм додавання інформації до вершин, напівребрів, ребер і граней набагато простіший і виконується під час виконання, а не під час компіляції. Оскільки структура даних використовує цілочисельні індекси як дескриптори для вершин, напівребер, ребер і граней, вона займає менше пам’яті, ніж версія, заснована на 64-розрядних покажчиках. Оскільки індекси є суміжними, їх можна використовувати як індекс у векторах, які зберігають властивості. Коли елементи видаляються, вони лише позначаються як видалені, і для справжнього їх видалення потрібно викликати функцію збирання сміття [22].

4.4.1. Використання

Основний клас *Surface_mesh* надає чотири вкладені класи, які представляють основні елементи структури даних напівребра:

– *Surface_mesh::Vertex_index;*

– *Surface_mesh::Halfedge_index;*

– *Surface_mesh::Face_index;*

– *Surface_mesh::Edge_index.*

Ці типи є лише обгортками для цілих чисел, і їхньою основною метою є гарантування безпеки типів. Вони можуть бути створені за замовчуванням, що призводить до отримання недійсного елемента. Нові елементи можна додавати та видаляти з *Surface_mesh* за допомогою набору функцій низького рівня, які не підтримують з'єднаність. Однак є виняток - *Surface_mesh::add_face()*, яка намагається додати нову грань до мешу (визначену послідовністю вершин) і невдається, якщо операція не є топологічно валідною. У такому випадку повернений *Face_index* є *Surface_mesh::null_face().*

Приклад додавання грані до мешу наведено нижче (рис. 4.7).

```
typedef Surface_mesh<Point> Mesh;
Mesh m;
Mesh::Vertex_index u = m.add_vertex(Point(0,1,0));
Mesh::Vertex_index v = m.add_vertex(Point(0,0,0));
Mesh::Vertex_index w = m.add_vertex(Point(1,0,0));
m.add_face(u, v, w);
```

Рис. 4.7. Приклад додавання грані до мешу

Оскільки *Surface_mesh* базується на індексах, класи *Vertex_index*, *Halfedge_index*, *Edge_index* та *Face_index* не мають методів доступу до з'єднаності чи властивостей. Для отримання цієї інформації слід використовувати функції екземпляра *Surface_mesh*, з якого вони були створені.

Нижче наведено приклад, який демонструє створення простої *Surface_mesh* додаючи дві грані, і перевірку, що грань коректно додана до мешу (рис. 4.8).

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Surface_mesh.h>

typedef CGAL::Simple_cartesian<double> K;
typedef CGAL::Surface_mesh<K::Point_3> Mesh;
typedef Mesh::Vertex_index vertex_descriptor;
typedef Mesh::Face_index face_descriptor;

int main()
{
  Mesh m;

  // Add the points as vertices
  vertex_descriptor u = m.add_vertex(K::Point_3(0,1,0));
  vertex_descriptor v = m.add_vertex(K::Point_3(0,0,0));
  vertex_descriptor w = m.add_vertex(K::Point_3(1,1,0));
  vertex_descriptor x = m.add_vertex(K::Point_3(1,0,0));

  m.add_face(u,v,w);
  face_descriptor f = m.add_face(u,v,x);
  if(f == Mesh::null_face())
  {
    std::cerr<<"The face could not be added because of an orientation error."<<std::endl;
    f = m.add_face(u,x,v);
    assert(f != Mesh::null_face());
  }
  return 0;
}
```

Рис. 4.8. Приклад створення простого мешу

4.4.2. З’єднання

Surface_mesh є структурою даних, орієнтованою на ребра, яка може зберігати інформацію про взаємодію вершин, ребер і граней. Кожне ребро представлено двома напівребрами з протилежною орієнтацією. Кожне напівребро зберігає посилання на прилеглу грань і прилеглу вершину. Крім того, воно зберігає посилання на наступне

та попереднє напівребро, прилегле до його прилеглої грані. Для кожної грані та кожної вершини зберігається прилегле напівребро. Напівребра не зберігають індекс протилежного напівребра, оскільки *Surface_mesh* зберігає протилежні напівребра послідовно в пам'яті.

Наведена нижче фігура (рис. 4.9) демонструє функції, які дозволяють навігувати в поверхневому меші: *Surface_mesh::opposite(), Surface_mesh::next(), Surface_mesh::prev(), Surface_mesh::target()*, і *Surface_mesh::face()*. Крім того, функція *Surface_mesh::halfedge()* дозволяє отримати напівребро, пов'язане з вершиною та гранню. За необхідності можна використовувати також вільні функції з тими ж назвами, визначені в пакетах *CGAL* та *Boost Graph Library*.

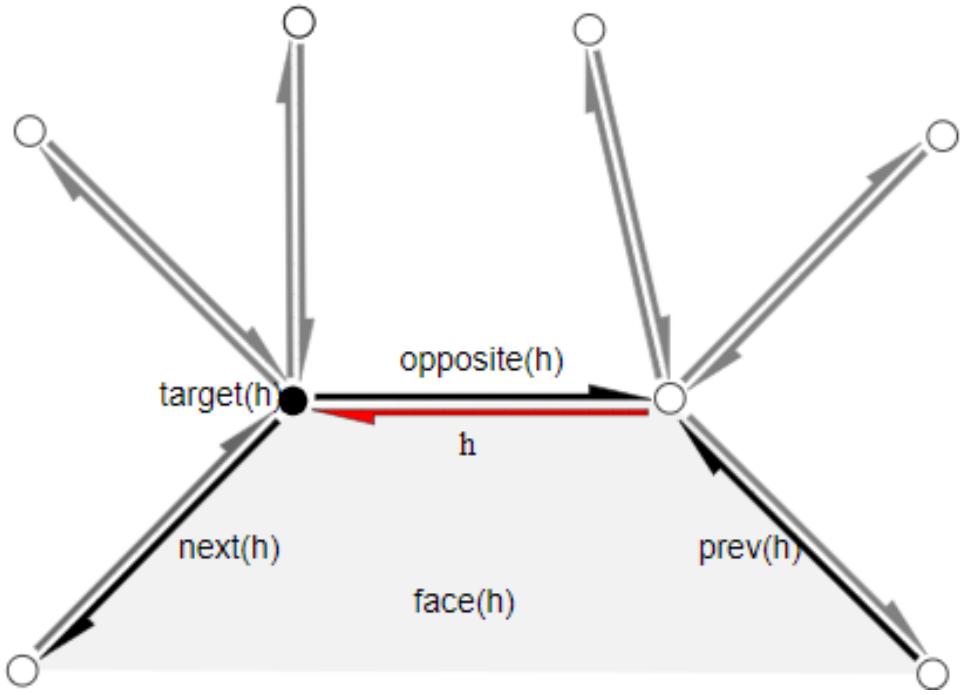


Рис. 4.9. Зв'язність половинних ребер і вершин у поверхневій сітці, видимій ззовні

Напівребра, прилеглі до грані, утворюють цикл. Залежно від того, з якого боку дивиться на поверхню, послідовність напівребер може виглядати, як орієнтована за годинниковою стрілкою або проти годинникової стрілки. Коли мова йде про орієнтацію обходу, ми розглядаємо поверхню так, що напівребра навколо грані орієнтовані проти годинникової стрілки (див. рис. 4.9).

Surface_mesh надає діапазони ітераторів для переліку всіх вершин, напівребер, ребер і граней. Він надає методи-члени, які повертають діапазони елементів, сумісних з бібліотекою *Boost.Range*.

4.4.3. Циркулятори

Навколо граней та вершин надаються циркулятори (*circulators*) як класові шаблони в пакеті *CGAL* та бібліотеці *Boost Graph*. Циркулятори навколо граней в основному викликають *Surface_mesh::next()* для переміщення від напівребра до напівребра проти годинникової стрілки навколо грані, і при розіменуванні повертають напівребро, або прилеглу вершину, або протилежну грань. *Circulators* навколо цільової вершини ребра в основному викликають *Surface_mesh::opposite(Surface_mesh::next())* для переміщення від напівребра до напівребра за годинниковою стрілкою навколо тієї самої цільової вершини.

Усі *circulators* реалізують інтерфейс *BidirectionalCirculator*. Крім того, вони також підтримують перетворення в *bool* для більш зручної перевірки на пустоту.

4.4.4. Властивості

Surface_mesh надає механізм для визначення нових властивостей для вершин, напівребер, ребер і граней під час виконання програми. Кожна властивість ідентифікується рядком та його ключовим типом. Всі значення властивості зберігаються як послідовні блоки пам'яті. Посилання на властивості стають недійсними, коли до структури даних додаються нові елементи ключового типу або коли виконується функція *Surface_mesh::collect_garbage()*. Властивості елемента будуть існувати після видалення елемента. Спроба доступу до властивості через недійсний елемент призведе до невизначеної поведінки. Одна властивість підтримується за замовчуванням, а саме «*v:point*». Значення цієї властивості повинно бути надано при додаванні нової точки до структури даних за допомогою *Surface_mesh::add_vertex()*. До властивості можна звертатися напряму за допомогою *Surface_mesh::points()* або *Surface_mesh::point(Surface_mesh::Vertex_index v)*.

Коли елемент видаляється, він лише позначається як видалений, і він дійсно видаляється, коли викликається *Surface_mesh::collect_garbage()*. Збірка сміття також дійсно видаляє властивості цих елементів.

З'єднаність також зберігається властивостями, а саме властивостями з іменами «*v:connectivity*», «*h:connectivity*» і «*f:connectivity*». Аналогічно для мітки видаленого елемента, де у нас є «*v:removed*», «*e:removed*» і «*f:removed*».

Напівребро також зберігає посилання на грань, а саме на її інцидентну грань. Напівребро *h* вважається на кордоні, якщо воно не має інцидентної грані, тобто якщо *sm.face(h) == Surface_mesh::null_face()*. Ребро вважається на кордоні, якщо хоча б одне з його напівребер знаходиться на кордоні. Так само вершина вважається на кордоні, якщо хоча б одне з її інцидентних напівребер знаходиться на кордоні. У вершини в *Surface_mesh* є лише одне пов'язане напівребро. Якщо користувач впевнений, що пов'язане напівребро є напівребром на кордоні, в разі, якщо вершина знаходиться на кордоні, немає потреби переглядати всі інцидентні напівребра функцією *is_border()* для вершин. Щоб перевірити лише чи пов'язане напівребро знаходиться на кордоні, функцію *Surface_mesh::is_border(Vertex_index v, bool check_all_incident_halfedges = true)* потрібно викликати з параметром *check_all_incident_halfedges = false*.

Користувач відповідає за правильне встановлення напівребра, пов'язаного з вершиною після застосування операції, яка може зробити цю властивість недійсною. Функції *Surface_mesh::set_vertex_halfedge_to_border_halfedge(Vertex_index v), Surface_mesh::set_vertex_halfedge_to_border_halfedge(Halfedge_index h),* і *Surface_mesh::set_vertex_halfedge_to_border_halfedge()* дозволяють встановлювати напівребро на кордоні для одного елемента *v*, для всіх елементів на кордоні грані *h*, і для всіх елементів поверхневого мешу відповідно.

4.4.5. Управління пам’яттю

Управління пам'яттю є в *Surface_mesh* є напівавтоматичним. Пам'ять збільшується при додаванні більше елементів до структури, але не скорочується, коли елементи видаляються. При додаванні елементів і випадку, коли вичерпано поточну ємність вектора, вектор перерозподілить пам'ять. Оскільки дескриптори в основному є індексами, після перерозподілу вони вказують на той самий елемент. При видаленні елемента він лише відзначається як видалений. Внутрішньо він додається до списку вільних елементів, і коли ви додаєте елементи до поверхневого мешу, вони беруться

зі списку вільних, якщо він не пустий. Для всіх елементів надається функція для отримання кількості використаних елементів, а також кількості використаних і видалених елементів. Для вершин це функції *Surface_mesh::number_of_vertices()* і *Surface_mesh::number_of_removed_vertices()* відповідно.

Ітератори, такі як *Surface_mesh::Vertex_iterator*, перелічують лише елементи, які не відзначені як видалені. Щоб дійсно зменшити використовувану пам'ять, потрібно викликати *Surface_mesh::collect_garbage()*. Важливо враховувати, що при зборі сміття елементи отримують нові індекси. У випадку утримання дескрипторів вершин вони, ймовірно, більше не вказують на правильні вершини.

4.4.6. Використання в додатку

У цьому підрозділі розглядається конкретне використання функціоналу бібліотеки *CGAL* для обробки та аналізу тривимірних мешів у програмному додатку, який розроблено в рамках даної магістерської роботи. Зокрема, надається опис функції *constructMeshFromObj*, яка відповідає за конструювання тривимірної поверхні з вхідного файлу у форматі *OBJ*.

У бібліотеці *CGAL* реалізований зчитувач *OBJ* формату, тому потрібно викликати лише функцію із передачею необхідних параметрів. Але алгоритм зчитування виглядає наступним чином:

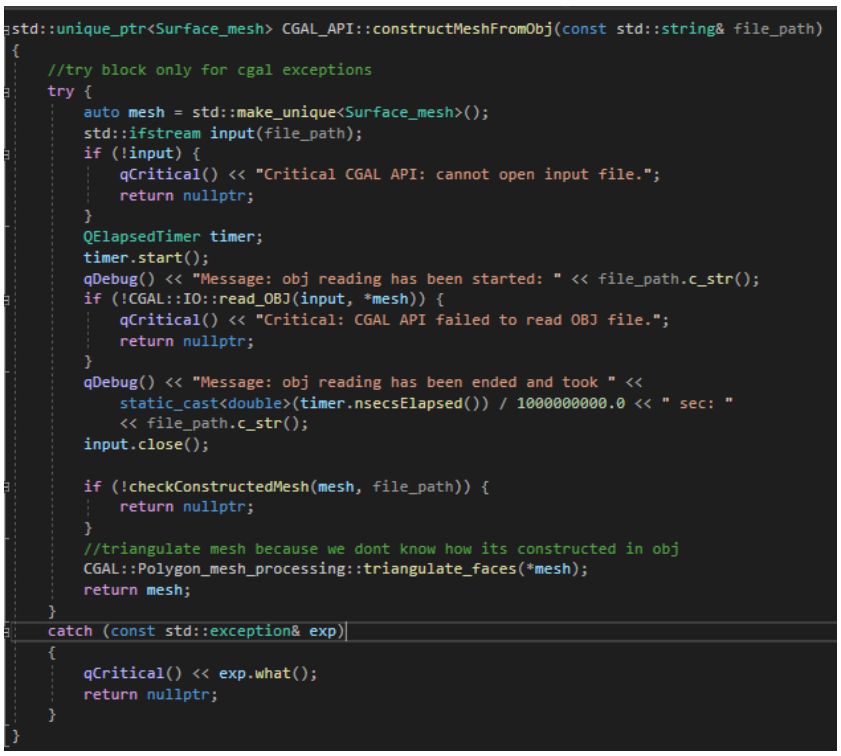
– відкриття *OBJ* файлу в потоці;

– ітерація по кожному рядку та пошук ключового слова, що представляє виконувану операцію (*v*, *f*);

– в разі знаходження вершини викликаємо метод *add_vertex* структури *Surface_mesh* та передаємо координати знайденої вершини;

– в разі знаходження грані викликаємо метод *add_face* структури *Surface_mesh* та передаємо список перечислених вершин.

Так як алгоритм вже реалізований всередині бібліотеки *CGAL*, достатньо викликати метод *CGAL::IO:Read_OBJ(input, mesh)* та передати відкритий *OBJ* файл в потоці першим параметром, а другим – новостворений об’єкт *Surface_mesh*. Реалізовація функції *constructMeshFromObj* зображена нижче (рис. 4.10).



```
std::unique_ptr<Surface_mesh> CGAL_API::constructMeshFromObj(const std::string& file_path)
{
    //try block only for cgal exceptions
    try {
        auto mesh = std::make_unique<Surface_mesh>();
        std::ifstream input(file_path);
        if (!input) {
            qCritical() << "Critical CGAL API: cannot open input file.";
            return nullptr;
        }
        QElapsedTimer timer;
        timer.start();
        qDebug() << "Message: obj reading has been started: " << file_path.c_str();
        if (!CGAL::IO::read_OBJ(input, *mesh)) {
            qCritical() << "Critical: CGAL API failed to read OBJ file.";
            return nullptr;
        }
        qDebug() << "Message: obj reading has been ended and took " <<
            static_cast<double>(timer.nsecsElapsed()) / 1000000000.0 << " sec: "
            << file_path.c_str();
        input.close();

        if (!checkConstructedMesh(mesh, file_path)) {
            return nullptr;
        }
        //triangulate mesh because we dont know how its constructed in obj
        CGAL::Polygon_mesh_processing::triangulate_faces(*mesh);
        return mesh;
    }
    catch (const std::exception& exp)
    {
        qCritical() << exp.what();
        return nullptr;
    }
}
```

Рис. 4.10. Функція створення поверхневої сітки із вхідного *OBJ* файлу

Функція включає блок обробки виняткових ситуацій, щоб виявити та вивести інформацію про помилки, які можуть виникнути під час роботи з бібліотекою *CGAL*. Це сприяє забезпеченню стійкості та надійності програмного додатку при взаємодії з *CGAL*. Також функціональність вимагає наявності файлу у форматі *OBJ* з правильною структурою, інакше може виникнути помилка при спробі зчитування.

Після успішного створення об’єкту *Surface_mesh* потрібно обрахувати вектори нормалей для кожної вершини мешу та параметризувати поверхню *3D*-об’єкту.

У області імен *Polygon_mesh_processing* бібліотеки *CGAL* також вже реалізовано метод для обчислення вектора нормалей на вершині, тому достатньо ітеруватися по кожній вершині сітки та викликати метод *CGAL::Polygon_mesh_processing::compute_vertex_normal(vertex, mesh)* із переданою вершиною та вхідною сіткою.

Алгоритм обчислення вектору нормалей в кожній вершині може виглядати наступним чином:

– обчислення вектору нормалей для кожної грані у меші. Нормаль грані можна обчислити як векторний добуток двох векторів сторін трикутника;

– для кожної вершини потрібно акумулювати нормалі суміжних граней. Це можна зробити пройшовши кожну грань і додаючи її нормалі до відповідних вершин;

– нормалізувати накопичені нормалі в кожній вершині, щоб отримати нормалі вершин.

Нижче проілюстровано функцію обчислення нормалі вершини за допомогою реалізованого функціоналу бібліотеки *CGAL* (рис. 4.11).

```
QVector3D CGAL_API::computeVertexNormal(const std::unique_ptr<Surface_mesh>& mesh, const CGAL::SM_Vertex_index& vertex)
{
    if (nullptr == mesh)
        return { 0, 0, 0 };
    auto vertex_normal = CGAL::Polygon_mesh_processing::compute_vertex_normal(vertex, *mesh);
    return QVector3D(
        static_cast<float>(CGAL::to_double(vertex_normal.x())),
        static_cast<float>(CGAL::to_double(vertex_normal.y())),
        static_cast<float>(CGAL::to_double(vertex_normal.z()))
    );
}
```

Рис. 4.11. Функція обчислення нормалі вершини

UV-mapping є більш складною операцією для реалізації вручну, але, на щастя, вона також реалізована в бібліотеці *CGAL*. В *CGAL* цей алгоритм має назву: планарна параметризація триангульованих поверхневих сіток. Параметризація поверхні полягає в знаходженні однозначного відображення з відповідної області на поверхню. Хороше відображення – це таке, яке мінімізує або викривлення кутів (конформна параметризація), або викривлення площі (еквіареальна параметризація) у певному сенсі. *CGAL* фокусується на параметризації трикутних поверхонь, які є гомеоморфними диску або сфері, а також на кусково-лінійних відображеннях на плоску поверхню.

Незважаючи на те, що основною мотивацією перших методів параметризації було їхнє застосування до текстурного відображення, зараз їх часто використовують для відображення більш складних модуляційних сигналів (таких як нормалі, прозорість, відображення або сигнали модуляції світла), апроксимації розсіяних даних, репараметризації поверхонь сплайнів, виправлення моделей *CAD*, апроксимації поверхонь та ремешінгу. Функція *parameterize()* застосовує метод параметризації поверхні за замовчуванням, а саме *Floater Mean Value Coordinates* до підключеного компонента сітки типу *TriangleMesh* з межею, заданою половиною *bhd*.

Нижче проілюстровано функцію параметризації поверхні, що повертає файл із координатами текстур для кожної вершини, за допомогою бібліотеки *CGAL* (рис. 4.12).

```cpp
#include <CGAL/Polygon_mesh_processing/measure.h>
#include <CGAL/Surface_mesh_parameterization/parameterize.h>
#include <CGAL/Surface_mesh_parameterization/IO/File_off.h>
std::ofstream uvMapping(const std::unique_ptr<Surface_mesh> &mesh)
{
    typedef boost::graph_traits<Surface_mesh>::vertex_descriptor vertex_descriptor;
    typedef boost::graph_traits<Surface_mesh>::halfedge_descriptor halfedge_descriptor;
    typedef boost::graph_traits<Surface_mesh>::face_descriptor face_descriptor;
    typedef Kernel::Point_2 Point_2;

    halfedge_descriptor bhd = CGAL::Polygon_mesh_processing::longest_border(*mesh).first;
    // The UV property map that holds the parameterized values
    typedef Surface_mesh::Property_map<vertex_descriptor, Point_2> UV_pmap;
    UV_pmap uv_map = mesh->add_property_map<vertex_descriptor, Point_2>("h:uv").first;
    CGAL::Surface_mesh_parameterization::parameterize(*mesh, bhd, uv_map);
    std::ofstream out("result.off");
    CGAL::Surface_mesh_parameterization::IO::output_uvmap_to_off(*mesh, bhd, uv_map, out);
    return out;
}
```

Рис. 4.12. Функція параметризації поверхні із використанням бібліотеки *CGAL*

Результат процесу параметризації бібліотеки *CGAL* можна побачити нижче (рис. 4.13).

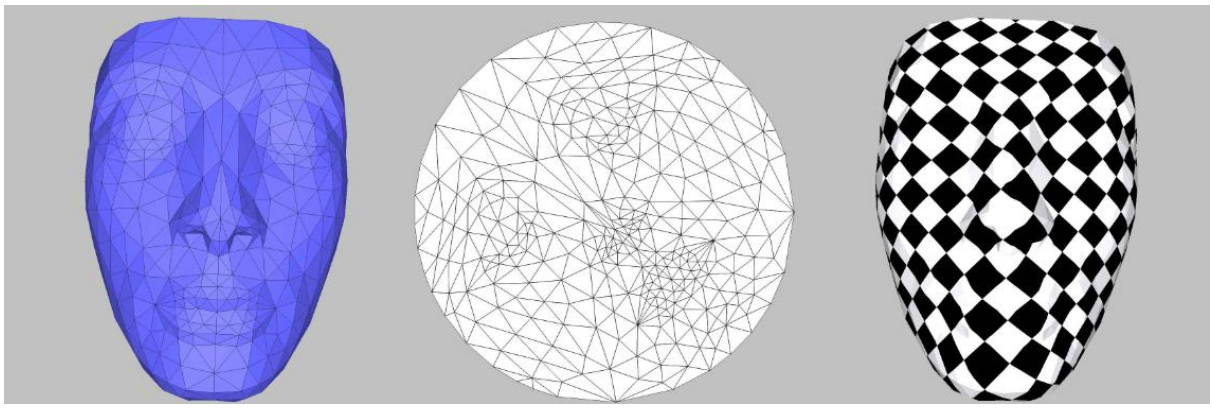


Рис. 4.13. Вхідний меш, параметризований простір, текстурований меш

Після виконання всіх вищезгаданих операцій отримуємо вектор вершин, кожна з якого складається із позиції в *3D*-просторі, нормалі та текстурної координати. Далі цей вектор потрібно використовувати для візуалізації об’єкту на сцені.

4.5. Графічний конвеєр

Графічний конвеєр, також відомий як конвеєр відтворення, є структурою в графіці комп'ютерів, яка визначає необхідні процедури для перетворення тривимірної

(*3D*) сцени в двовимірне (*2D*) зображення на екрані. Після того, як створено *3D*-модель, будь то для відеоігор чи будь-якої іншої форми тривимірної комп'ютерної анімації, графічний конвеєр перетворює модель в візуально сприйнятний формат на дисплеї комп'ютера. Завдяки залежності від конкретного програмного забезпечення, конфігурацій обладнання та бажаних характеристик відображення, універсального графічного конвеєру для всіх випадків не існує. Тим не менше, інтерфейси програмування застосунків графіки (*API*), такі як *Direct3D* та *OpenGL*, розроблені для стандартизації загальних процедур та контролю за графічним конвеєром конкретного прискорювача обладнання. Ці *API* надають абстракційний рівень над основним обладнанням, звільняючи програмістів від необхідності писати код, який напряму спрямований на різні прискорювачі графіки, такі як *AMD, Intel, Nvidia* та інші. Модель графічного конвеєра зазвичай використовується в реальному часі відтворення. Часто більшість етапів конвеєра реалізовані на рівні обладнання, що дозволяє здійснювати спеціальні оптимізації. Термін «конвеєр» використовується в схожому розумінні для конвеєра в процесорах: окремі етапи конвеєра виконуються паралельно, поки будь-який з етапів має все необхідне.

Зазвичай термін «*3D*-конвеєр» вказує на найпоширенішу форму відтворення *3D*-графіки на комп'ютері, відому як полігональне моделювання, що відрізняється від технік відтворення променів і рейкастингу. У рейкастингу промінь виходить з точки, де знаходиться камера, і якщо цей промінь вдаряється в поверхню, розраховується колір і освітлення точки на поверхні, де промінь вдарився.

Графічний конвеєр можна розділити на такі основні етапи: програма, геометрія та растеризація (рис. 4.14).

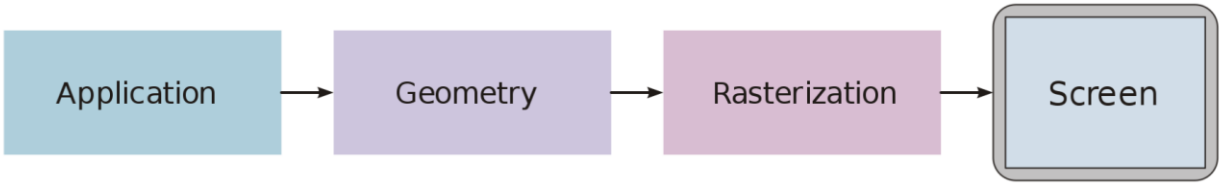


Рис. 4.14. Етапи графічного конвеєра

Програма – етап застосування виконується програмним забезпеченням на основному процесорі (ЦП). Під час етапу вносяться зміни до сцени за необхідності,

наприклад, в результаті взаємодії користувача за допомогою введених пристроїв або під час анімації. Нова сцена з усіма її примітивами, зазвичай трикутниками, лініями та точками, потім передається на наступний етап конвеєра. Приклади завдань, які зазвичай виконуються на етапі застосування, включають виявлення зіткнень, анімацію та техніки прискорення за допомогою просторових схем підрозділу, що також використовуються для зменшення кількості основної пам'яті, необхідної у конкретний момент часу.

Етап геометрії (з геометричним конвеєром), відповідальний за більшість операцій з полігонами та їх вершинами (з вершинним конвеєром), може бути розділений на наступні п'ять завдань: перетворення моделі та камери (*model & camera transformations*), освітлення (*lighting*), проєкція (*projection*), відсікання (*clipping*), перетворення вікна-перегляду (*window-viewport transformation*).

Процес растеризації [7], останній етап перед фрагментним конвеєром шейдерів, включає у себе перетворення всіх примітивів у дискретні фрагменти. На цьому етапі, який є частиною графічного конвеєра, точки сітки отримують назву фрагментів, кожен з яких відповідає одному пікселю в буфері кадру та, відповідно, одному пікселю екрану. Фрагменти можуть мати кольорове відображення та освітлення. Також необхідно визначити видимий фрагмент у випадку перекриття полігонів, для чого використовується буфер глибини (*Z*-буфер). Колір фрагмента залежить від освітлення, текстури та інших властивостей матеріалу видимого примітиву, що часто піддаватиметься інтерполяції з властивостей вершин трикутника. На етапі растеризації, якщо доступний, виконується фрагментний шейдер (також відомий як піксельний шейдер) для кожного фрагмента об'єкта. Якщо фрагмент є видимим, його можна змішати із вже існуючими значеннями кольору на зображенні у випадку використання прозорості чи семплінгу. Щоб уникнути поступової растеризації примітивів, використовується подвійна буферизація, і після повного завершення растеризації зображення копіюється в область видимої пам'яті для відображення.

Сучасні відеокарти використовують програмований конвеєр, який керується шейдером і має можливість безпосередньо взаємодіяти з окремими етапами обробки. З метою розширення функціоналу та звільнення основного процесора, додаткові

етапи обробки були переміщені до конвеєра та графічного процесора. Шейдер – програма, яка обчислює необхідні рівні світла, темряви та кольору під час рендерингу тривимірної сцени. Шейдери виникли для виконання різноманітних спеціалізованих функцій в графіці комп'ютерних ефектів, а також для загальнопризначених обчислень на графічних обчислювальних пристроях. Більшість шейдерів програмують для (та виконують на) графічного процесора (*GPU*), хоча це не є строгим обов'язком. За допомогою шейдерів можна змінити позицію та колір всіх пікселів, вершин та/або текстур, які використовуються для створення остаточного зображення [23]. Найпоширеніші шейдери наведено нижче.

– Вершинні шейдери – взаємодіють з інформацією про вершини геометричних об'єктів, такі як їхні координати в просторі та інші характеристики. Застосовуються для різноманітних операцій, таких як трансформація вершин, визначення текстурних координат та розрахунок освітлення.

– Геометричні шейдери – обробляють не одну, а цілу групу вершин, утворюючи примітиви. Можуть взаємодіяти з інформацією про суміжні вершини та генерувати нові примітиви, спрощуючи завдання центральному процесору.

– Фрагментні (піксельні) шейдери – працюють з фрагментами зображення, визначаючи їхні атрибути, такі як кольори та текстурні координати. Застосовуються для формування фрагментів зображення та створення різних візуальних ефектів.

У контексті даної роботи в програмі використовуються вершинний та фрагментний шейдери.

Нижче проілюстровано код вершинного шейдеру, що використовується в додатку (рис. 4.15).

Вершинний шейдер виконує необхідні обчислення для відображення тривимірних об'єктів на екрані, включаючи перетворення координат вершин, обчислення нормалей та визначення позицій вершин у просторі камери та проекційному просторі. Ці дані потім будуть використовуватися у фрагментному шейдері для подальших обчислень, таких як текстурування та освітлення. Шейдер повертає перетворену вершину із локального простору у простір камери та екрану.

Нижче проілюстровано код фрагментного шейдеру, що використовується в додатку (рис. 4.16).

Фрагментний шейдер реалізує освітлення за допомогою моделі Фонга для обох напрямного джерела світла (переднього). Він враховує амбієнтне, дифузне та відбите світло, а також використовує параметри для налаштування сили кожного компонента освітлення та повертає результуючий колір пікселя на екрані.

```
#version 330 core

layout(location = 0) in vec3 inPosition;
layout(location = 1) in vec3 inNormal;
layout(location = 2) in vec2 inTexture;

uniform mat4 viewMatrix;
uniform mat4 projectionMatrix;
uniform mat4 modelMatrix;

out vec3 Normal;
out vec3 FragPos;

void main() {
    FragPos = (modelMatrix * vec4(inPosition, 1.0)).xyz;
    Normal = mat3(transpose(inverse(modelMatrix))) * inNormal;
    gl_Position = projectionMatrix * viewMatrix * vec4(FragPos, 1.0);
}
```

Рис. 4.15. Вершинний шейдер

```
#version 330 core

in vec3 FragPos;     // Fragment position in world coordinates
in vec3 Normal;      // Normal vector at the fragment

out vec4 FragColor;

uniform vec3 lightDirectionFront; // Direction of the front directional light
uniform vec3 lightDirectionBack;  // Direction of the back directional light
uniform vec3 lightColor;
uniform vec3 objectColor;
uniform float ambientStrength;
uniform float specularStrength;
uniform float shininess;

void main()
{
    // Calculate ambient lighting for both light sources
    vec3 ambientFront = ambientStrength * lightColor;

    // Calculate the direction from the fragment to both light sources
    vec3 lightDirFront = normalize(-lightDirectionFront); // Negative because it's a light direction

    // Calculate diffuse lighting for both light sources
    float diffFront = max(dot(Normal, lightDirFront), 0.0);
    vec3 diffuseFront = diffFront * lightColor;

    // Calculate the view direction (camera direction)
    vec3 viewDir = normalize(-FragPos);

    // Calculate specular lighting using the Phong reflection model for both light sources
    vec3 reflectDirFront = reflect(-lightDirFront, Normal);
    float specFront = pow(max(dot(viewDir, reflectDirFront), 0.0), shininess);
    vec3 specularFront = specularStrength * specFront * lightColor;

    // Combine lighting components from both light sources
    vec3 result = (ambientFront + diffuseFront + specularFront) * objectColor;

    // Output the final fragment color
    FragColor = vec4(result, 1.0);
}
```

Рис. 4.16. Фрагментний шейдер

Модель Фонга (*Phong reflection model*) – це модель освітлення, яка використовується у комп'ютерній графіці для симуляції взаємодії світла з поверхнею. Була запропонована Бліном Фонгом (*Bui-Tuong Phong*) у 1973 році. Ця модель

дозволяє враховувати амбієнтне, дифузне та відбите світло для кожного фрагмента поверхні.

Основні компоненти моделі Фонга наведено нижче.

– Амбієнтне (фонове) освітлення (*Ambient Lighting*) – компонент освітлення, який відображає рівномірне, розсіяне світло, яке розсипане навколо сцени. Це світло не має конкретного джерела та рівномірно освітлює всю сцену.

– Дифузне (розсіююче) освітлення (*Diffuse Lighting*) – компонент освітлення, який враховує розсіювання світла в різних напрямках при зіткненні з поверхнею. Цей компонент залежить від кута між напрямком світла та нормаллю до поверхні.

– Відбите світло (*Specular Reflection*) – компонент освітлення, що відображає «блиск» або відбиття світла від гладкої поверхні. Цей ефект стає більш помітним при розташуванні джерела світла під кутом до лінії погляду (*view direction*).

Модель Фонга використовує параметри, такі як коефіцієнти дифузного та відбитого світла, сила блиску (*shininess*), а також враховує напрямки світла та погляду, щоб реалістично відтворити освітлення на поверхні. Використання цієї моделі дозволяє отримати більш фотореалістичні зображення, оскільки вона враховує деякі аспекти взаємодії світла з поверхнею, які не враховуються більш простими моделями, такими як модель Ламберта (рис. 4.17).

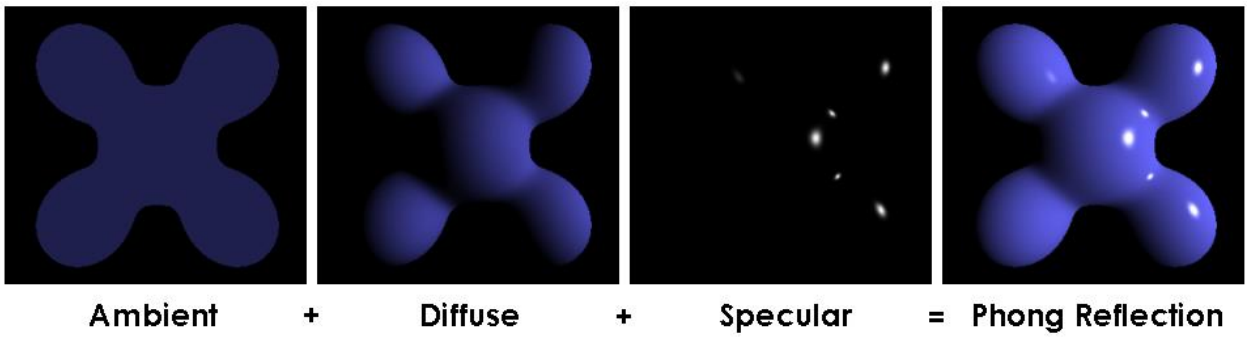


Рис. 4.17. Приклад використання моделі Фонга

Висновки за розділом

У комп'ютерній графіці термін «вершина» використовується для позначення точки або вузла в тривимірному або двовимірному просторі. Вершини є базовими

будівельними блоками для створення графічних об'єктів, таких як моделі *3D*-об'єктів чи форми в двовимірному просторі.

Текстурні координати – це пари чисел (зазвичай два або три), які вказують на конкретну точку на текстурі, пов'язаній з вершиною графічного об'єкта. Вони використовуються в комп'ютерній графіці для визначення того, як текстура повинна бути накладена на графічний об'єкт або його поверхню.

Нормаль в комп'ютерній графіці – це вектор, який перпендикулярний до поверхні геометричного об'єкта. Кожна точка на поверхні об'єкта може мати свою нормаль, і цей вектор вказує на напрямок від поверхні в зовнішній простір

Формат файлу *OBJ* підтримує визначення геометрії поверхонь об'єктів за допомогою полігональних сіток або за допомогою вільних кривих та поверхонь. Логічною концепцією в моделі *OBJ* є «елемент». У специфікації перераховані такі елементи: точка (*p*), лінія (*l*), грань (*f*), крива (*curv*), *2D*-крива (*curv2*) та поверхня (*surf*). Елементи створюються з вершин; вершини повинні бути перераховані першими в файлі, щоб елементи могли посилатися на них за номером, використовуючи порядок їх появи в файлі.

3D-сітка (*3D-mesh*) – це сукупність вершин, ребер і граней, які разом утворюють тривимірний об'єкт. Вершини – це координати в тривимірному просторі, ребра – з'єднують дві сусідні вершини, а грані (також називають полігонами) обгортають ребра, утворюючи поверхню об'єкта.

Графічний конвеєр, також відомий як конвеєр відтворення, є структурою в графіці комп'ютерів, яка визначає необхідні процедури для перетворення тривимірної (*3D*) сцени в двовимірне (*2D*) зображення на екрані. Графічний конвеєр перетворює модель в візуальносприйнятний формат на дисплеї комп'ютера.

РОЗДІЛ 5
АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

5.1. Мануальне тестування

Нижче зображено головне вікно програмного додатку, що отримуємо в разі відкриття (рис. 5.1).

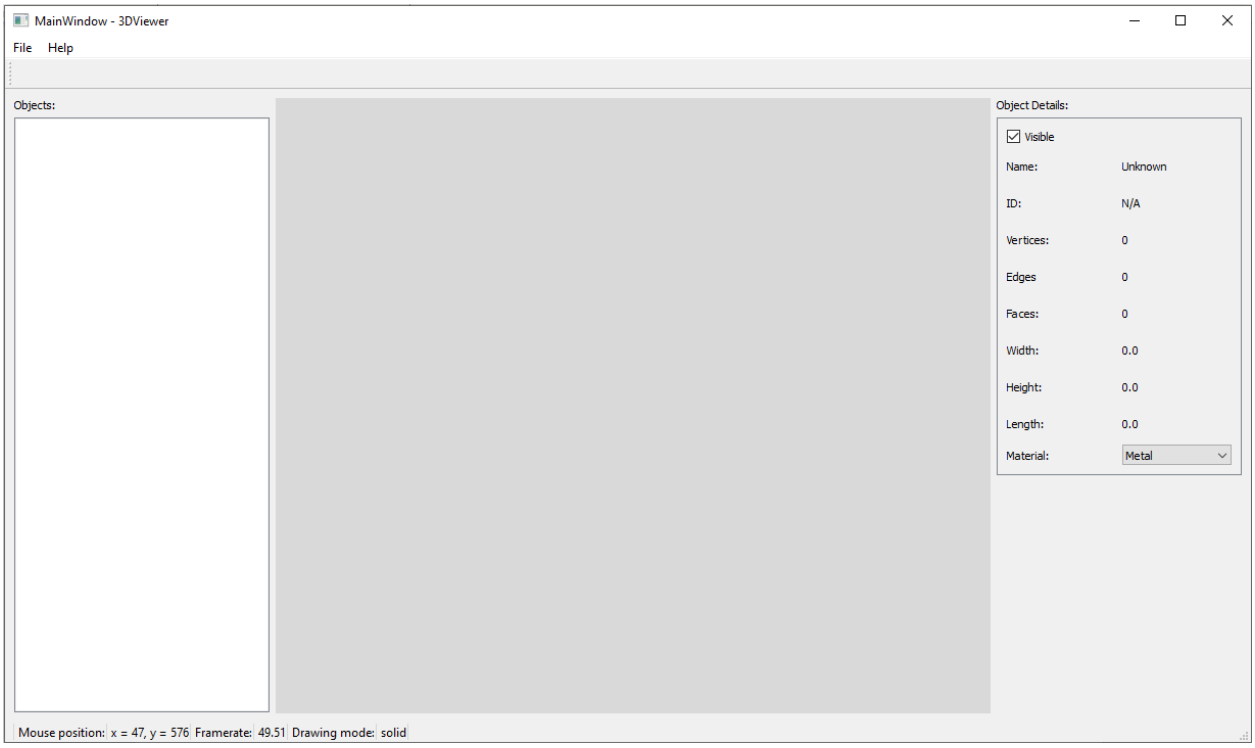


Рис. 5.1. Головне вікно програмного додатку

На головному вікні бачимо:

– файлове меню;

– меню допомоги;

– деревовидна структура відкритих файлів;

– область перегляду моделей;

– рядок стану, що включає в себе: позицію курсора миші на області перегляду, частоту кадрів, режим малювання;

– рамку із даними про обраний об’єкт, а саме: видимість, ім’я (береться із імені файлу), унікальний ідентифікатор, кількість вершин, кількість ребер, кількість граней, ширину, висоту, довжину, матеріал.

Через меню допомоги (рис. 5.2) можна відкрити вікно «гарячих» клавіш (рис. 5.3) та інформацію про автора (рис. 5.4).

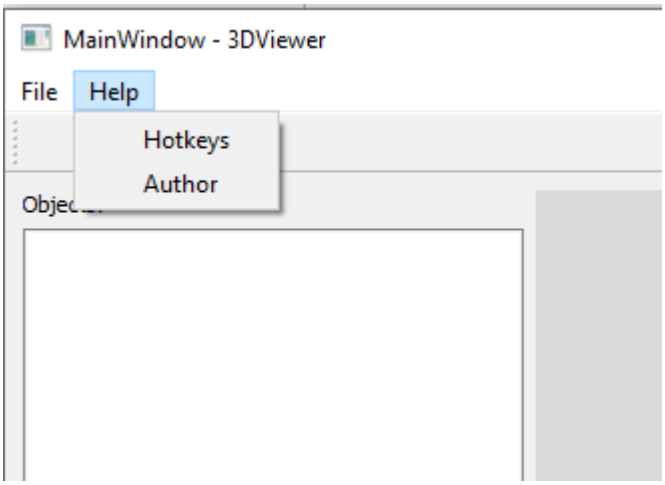


Рис. 5.2. Меню допомоги

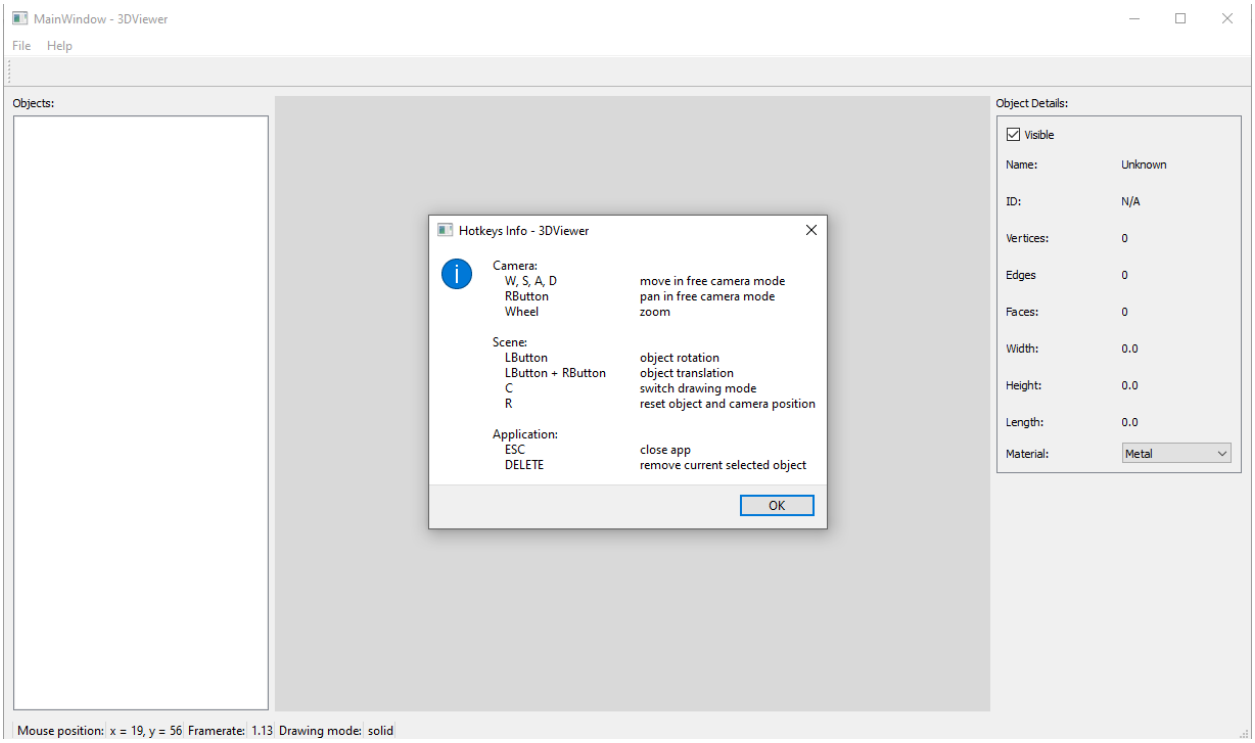


Рис. 5.3. Вікно «гарячих» клавіш

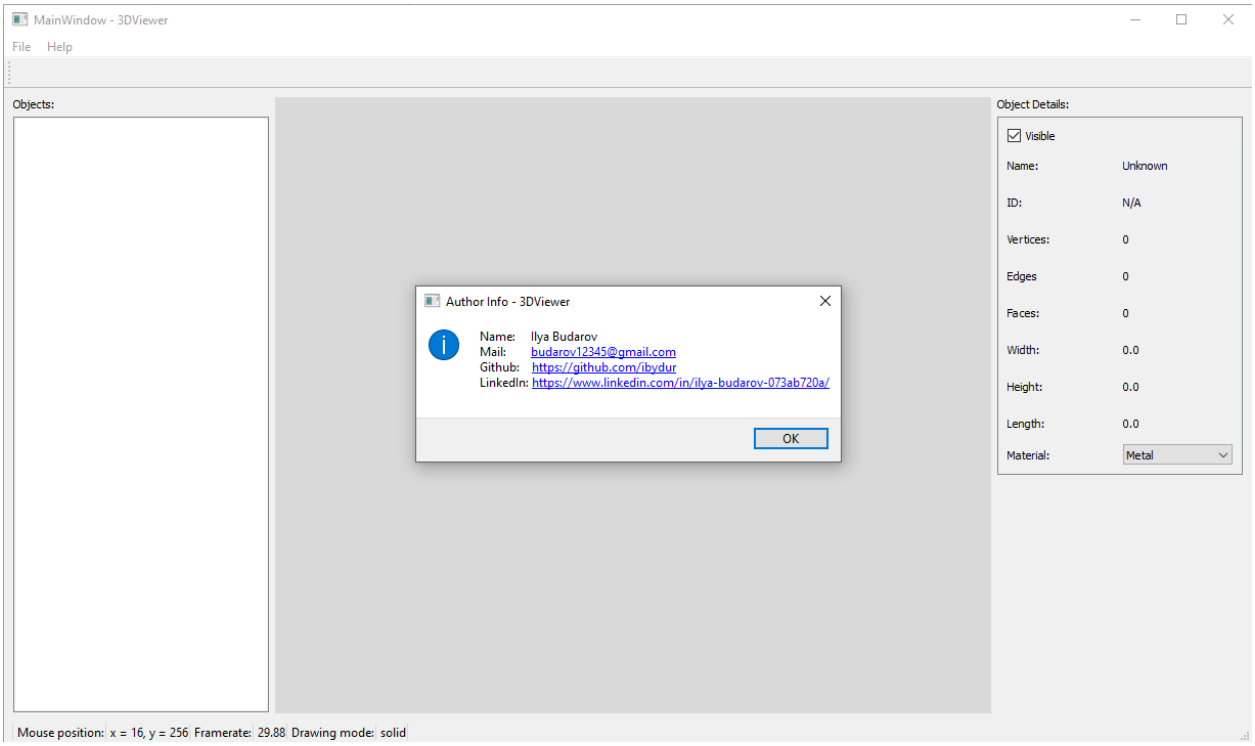


Рис. 5.4. Вікно інформації про автора

Через файлове меню (рис. 5.5) можна відкрити файлове вікно (рис. 5.6) та закрити додаток.

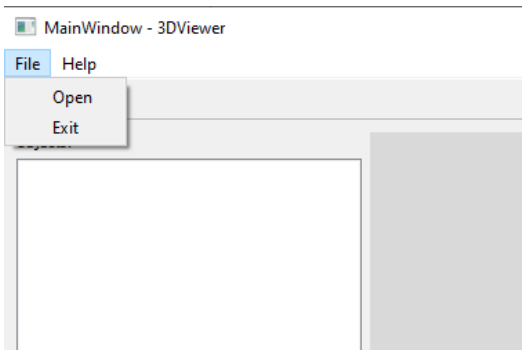


Рис. 5.5. Файлове меню

У фільтрах файлового вікна налаштований вибір лише відкриття файлів формату *OBJ*, тому інші формати не вдасться відкрити. Після вибору файлу *HeadPrime10c.obj*, у рядку стану з'явиться інформація про те, що цей файл завантажується (рис. 5.7), але за рахунок того, що завантаження файлів відбувається в окремому потоці, тобто головне вікно програми не блокується після вибору файлу, у користувача є можливість взаємодіяти із головним вікном програми чи відкрити ще один файл, доки очікується завантаження першого.

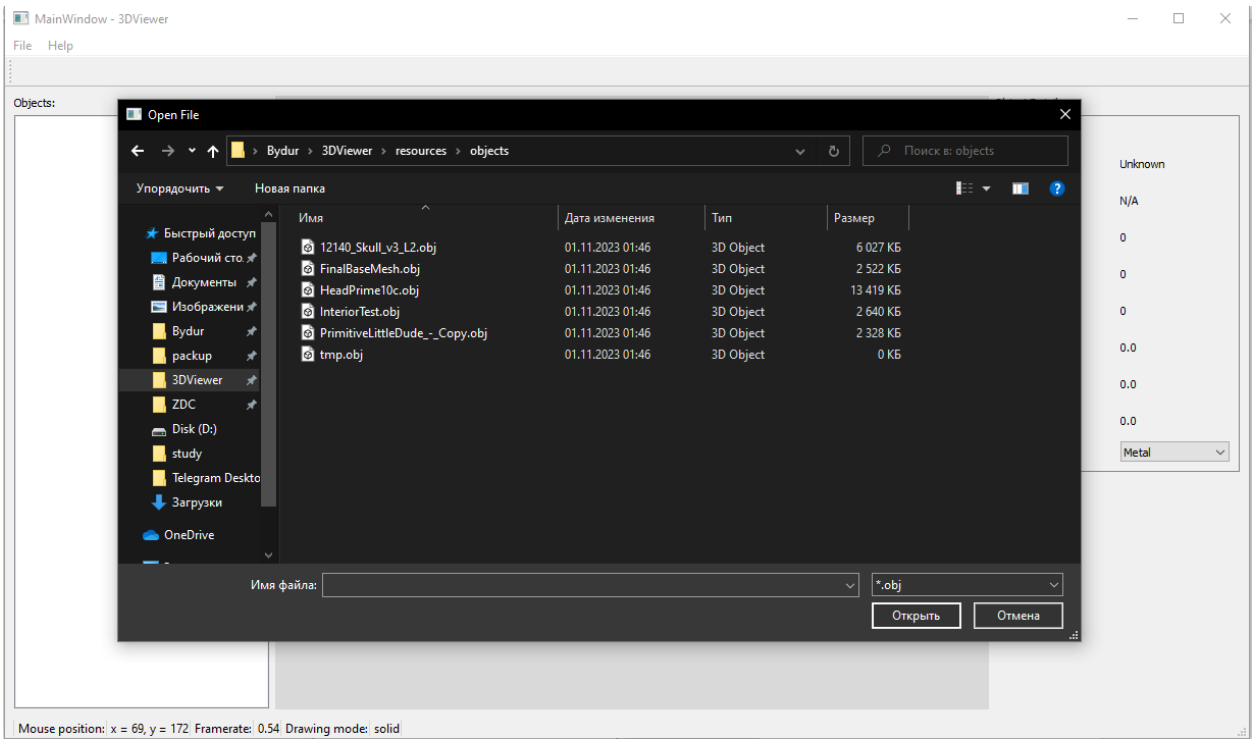


Рис. 5.6. Файлове вікно

Рис. 5.7. Очікування відкриття файлу *HeadPrime10c.obj*

Після успішного відкриття файлу можемо побачити візуалізований об’єкт та детальну інформацію про нього (рис. 5.8).

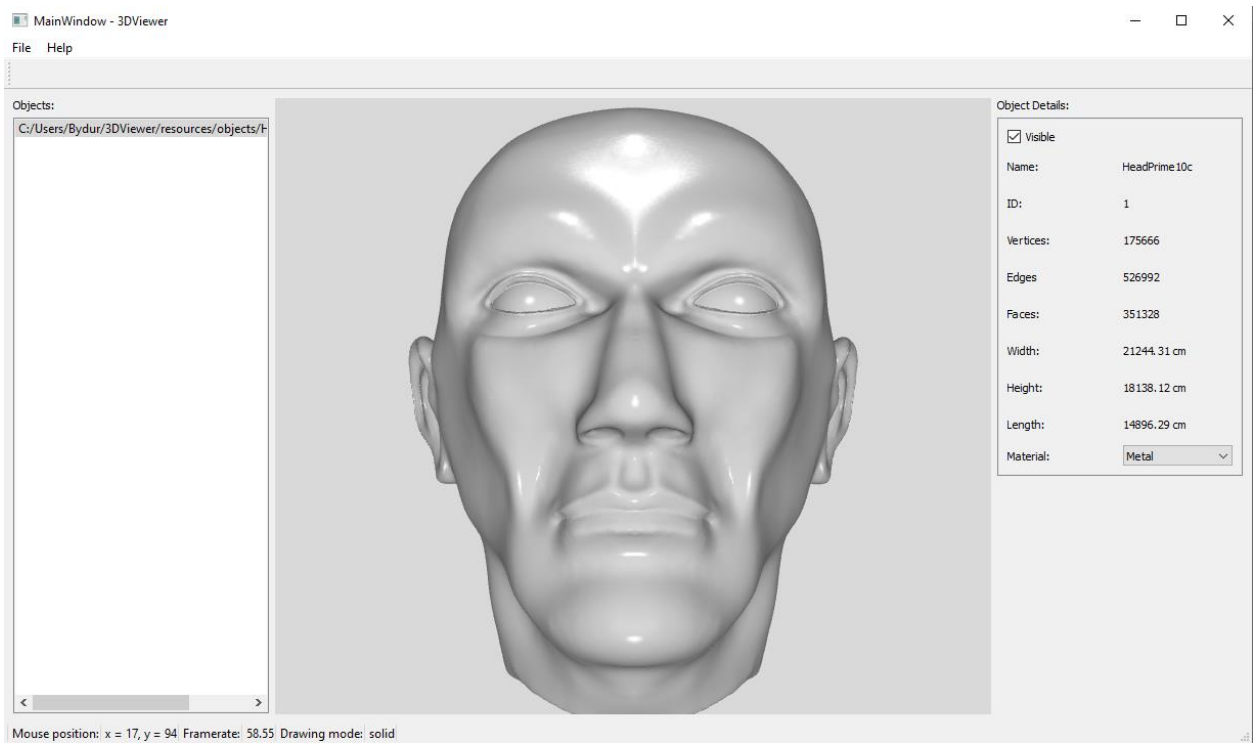


Рис. 5.8. Успішно завантажений об’єкт

Далі відкриємо файл *12140_Skull_v3_L2.obj* та змінемо для нього тип матеріалу із «*Metal*» на «*Skelet*» (рис. 5.9).

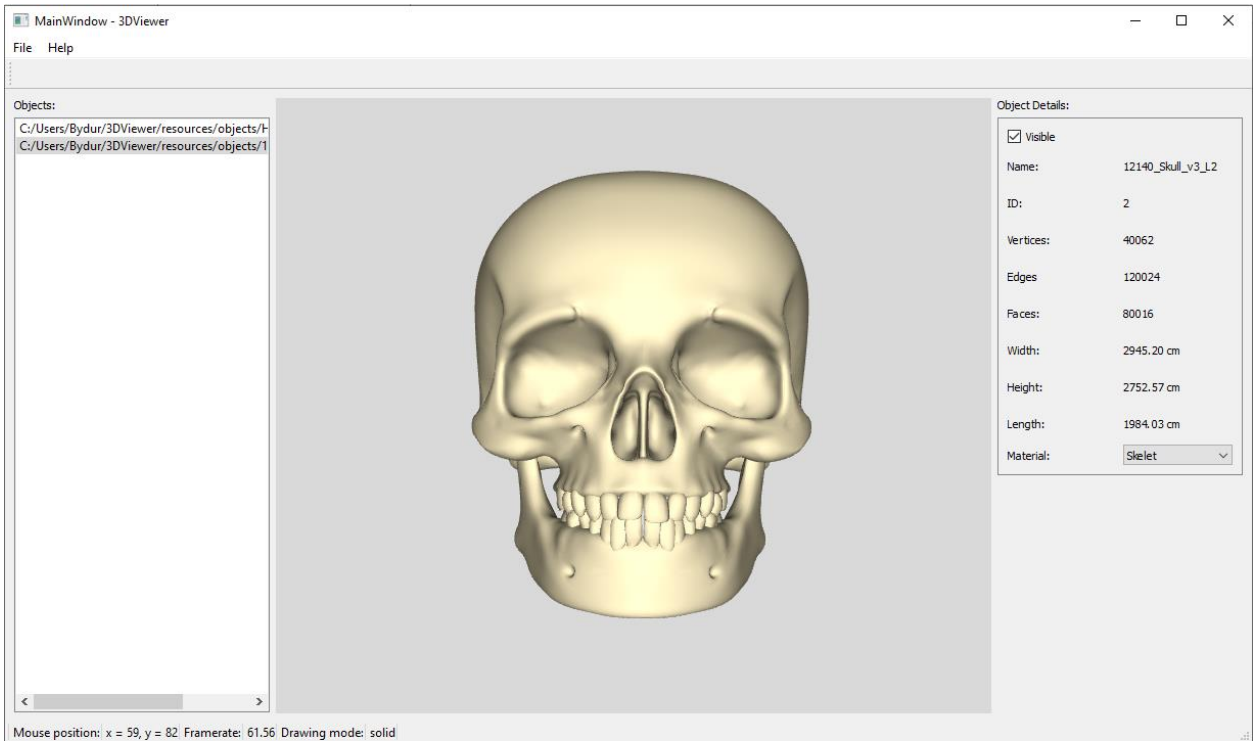


Рис. 5.9. Зміна матеріалу

Для мануального тестування доступний також консоль, де можна бачити інформацію про успіх відкриття файлу, а також час, витрачений на геометричні операції. У разі відкриття файлу без даних або із некоректною топологією, на консолі з’явиться відповідна інформація про це. Нижче зображено тестування відкриття пустого файлу із назвою «*tmp.obj*» (рис. 5.10).

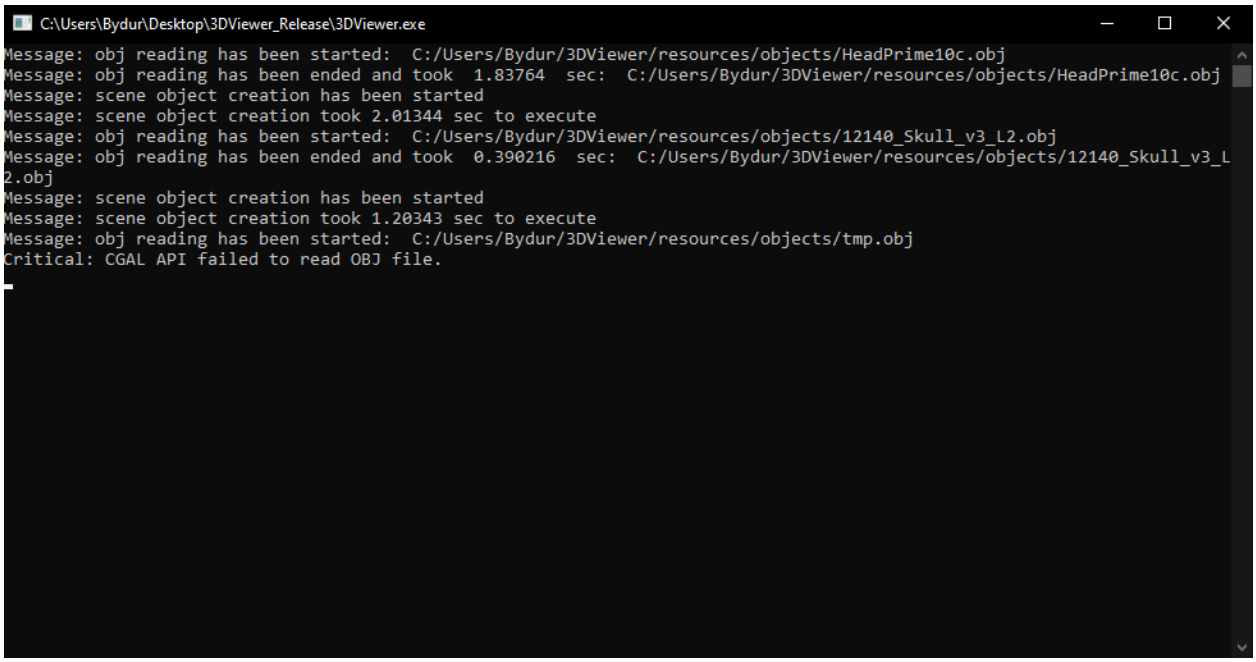


Рис. 5.10. Тестування відкриття пустого файлу

Знявши прапорець «*Visible*» даний об’єкт залишиться обраним у деревовидній структуру та можна буде бачити його детальну інформацію, але він буде невидимим на області перегляду (рис. 5.11).

Також у користувача є можливість взаємодії з програмою через ряд інтерактивних опцій, спрямованих на більш глибоке вивчення об'єкта та його оточення (рис. 5.11). Зокрема, він може виконувати повороти як самого об'єкта, так і камери, що дозволяє переглядати його з різних кутів. Переміщення об'єкта та камери дозволяє користувачу активно взаємодіяти з віртуальним середовищем, змінюючи їхнє положення. Окрім того, можливість збільшення та зменшення відстані до об'єкта надає зручність при дослідженні деталей.

Рис. 5.11. Зняття прапорця «*Visible*» для об’єкту «*12140_Skull_v3_L2.obj*»

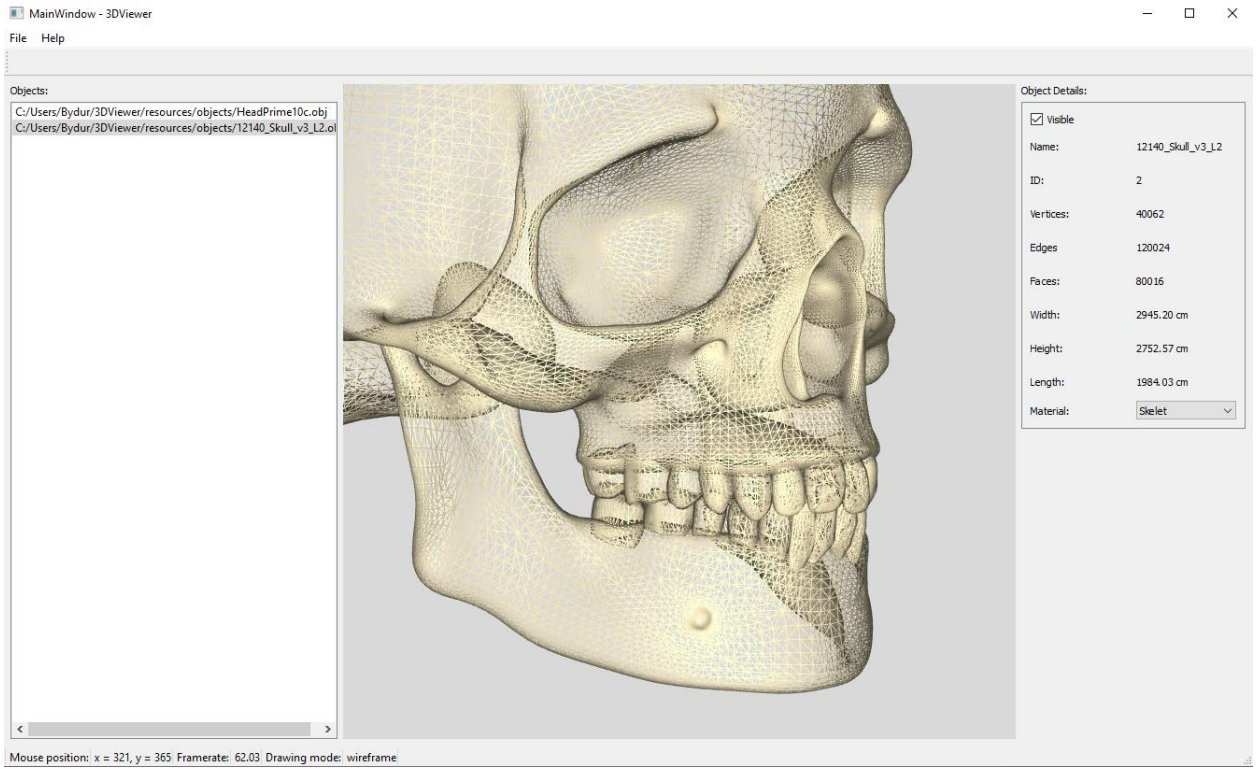


Рис. 5.12. Застосування маніпуляцій до об’єкту та камери

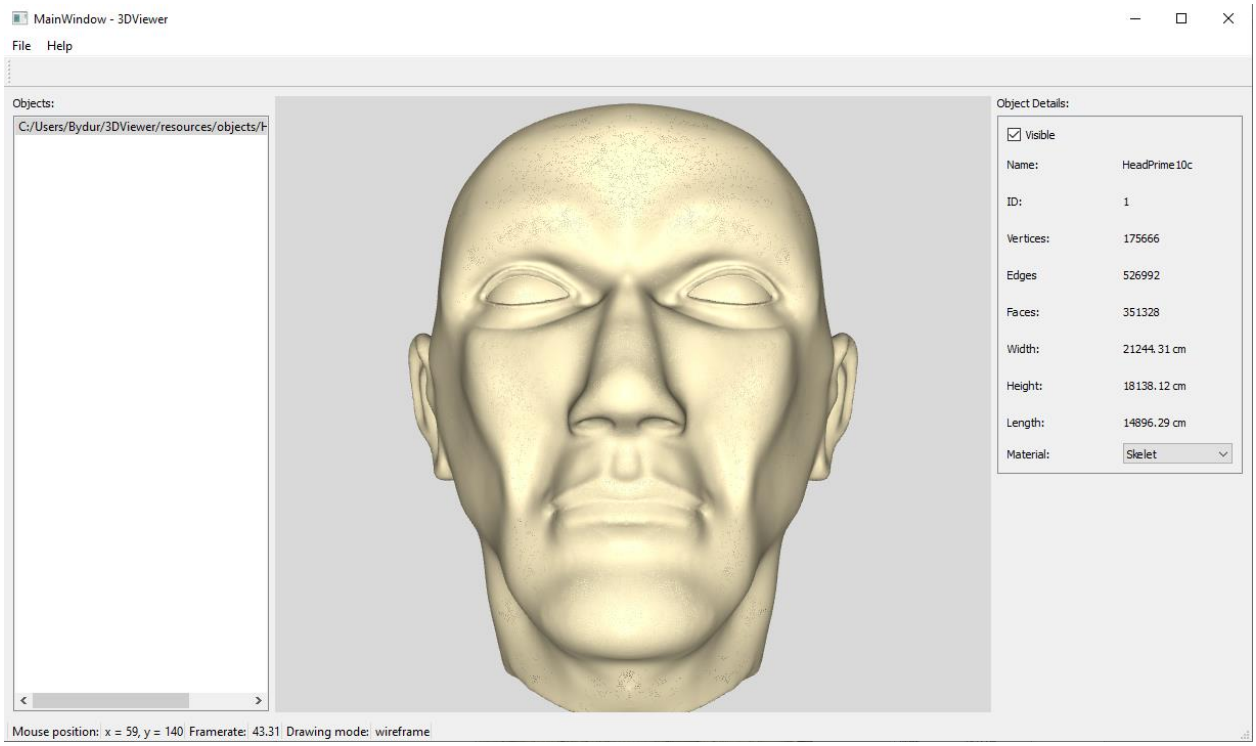


Рис. 5.13. Видалення об’єкту *12140_Skull_v3_L2.obj* із програми

Також користувач має змогу видалити об’єкт із програми та звільнити зайняту ним пам’ять, натиснувши кнопку «*DELETE*» на обраний об’єкт у деревовидній структурі. Після видалення, обраним об’єктом стане той, що був доданий останнім (див. рис. 5.13).

5.2. Вдосконалення

Як і будь-який інший програмний продукт, додаток для роботи із *3D*-даними потребує вдосконалень для заохочування більш широкої аудиторії користувачів. Подальшою метою буде розширення можливостей додатку та покращення користувацького досвіду через інтеграцію нового функціоналу та оптимізацію вже наявного.

До покращення функціоналу відносяться:

– вдосконалення можливостей програми шляхом розширення списку підтримуваних форматів для завантаження та відображення *3D*-об'єктів. Додавання підтримки інших форматів, таких як *STL*, *FBX* чи *glTF*, *STEP* та *AMF* розширить сферу застосування додатку;

– додавання можливостей анімації об'єктів на сцені. Інтеграція системи ключових кадрів дозволить користувачу створювати та переглядати рухомі *3D*-об'єкти, що відкриє нові перспективи для використання у галузі візуалізації та симуляцій;

– вдосконалення системи взаємодії з об'єктами на сцені. Додавання нових інструментів для вибору, обрання та взаємодії з об'єктами, таких як вибір за допомогою області, розширені опції редагування тощо;

– додавання можливості одночасного завантаження декількох об’єктів на сцену із можливістю застосування булевих операцій (*union, difference, intersection*).

Головною цінністю програмного додатку може стати створення алгоритму ретопології поверхні *3D*-об’єкту. Ретопологія – це процес створення тривимірної моделі із оптимізованою полігональною сіткою. Якщо вона не є оптимізованою, об'єкт матиме нерівну поверхню, неправильну геометрію і не буде придатний для текстурування та рендерингу [24]. Тому ретопологія є важливою для отримання візуальновисокоякісних комп'ютерних графік. Крім того, вона вирішальна для створення анімації, а також для візуалізації, готової до використання в доповненій та віртуальній реальності, оскільки ці типи контенту вимагають оптимізованих легких графічних елементів. Нижче зображено приклад квадратичної ретопології *3D*-об’єкту (рис 5.14).

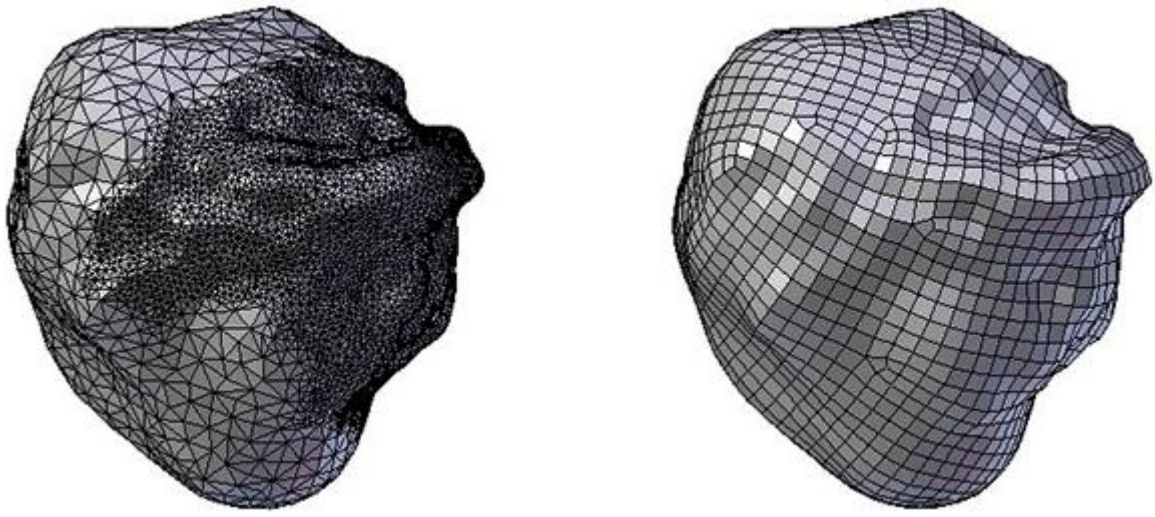


Рис. 5.14. Приклад квадратичної ретопології сітки

Головна ідея полягає в тому, що таким чином можна зберігати деталі, які ми отримуємо від роботи з високою роздільності, але все ще генерувати модель, яка плавно працює з анімацією. Наприклад, може бути модель, яка складається із 100 000 вершин, але завдяки алгоритму ретопології ми отримаємо на виході 1000 вершин, не втрачаючи структури та якості. Звичайно, що алгоритм ретопології реалізований в програмних застосунках, таких як: *Autodesk 3ds Max software, Autodesk Maya software, Modo, Blender, Houdini, Cinema4D* (табл. 4.1).

Таблиця 5.1

Порівняльна таблиця

Програмне Забезпечення	**Алгоритм Ретопології**	**Основні Функції**	**Підтримка Форматів**	**Ліцензія**
Autodesk 3ds Max	*Quad Draw, ProOptimizer*	Ретопологія поверхонь, оптимізація мешу	*OBJ, FBX, STL*	Платна
Autodesk Maya	*Quad Draw*	Ретопологія з використанням чотирикутників	*OBJ, FBX, STL*	Платна
Modo	*Topology Pen, RetopoMesh*	Ретопологія з використанням пенсилів та інструментів *RetopoMesh*	*LXO, OBJ, FBX*	Платна
Blender	*BSurfaces, Retopoflow*	Вирізання, розташування чотирикутників, інструменти *Retopoflow*	*OBJ, FBX, STL*	Безкоштовна
Houdini	*TopoBuild, PolyDraw*	Ретопологія на основі вузлів та полігонів, інструмент *PolyDraw*	*OBJ, FBX, Alembic*	Платна

Закінчення таблиці 5.1

1	**2**	**3**	**4**	**5**
Cinema4D	*RetopoRoom*	Інтерактивна ретопологія з інструментом *RetopoRoom*	*OBJ, FBX, STL*	Платна

Більшість із цих програм є комерційними, що говорить про конкурентну спроможність завдяки створенню якісного алгоритму ретопології, що включатиме в себе: коректну ретопологію як для цилідричний форм, так і для спіральних, можливість контролю кількістю результуючих вершин та симетричне створення. До наявного функціоналу програмного додатку для роботи із *3D*-моделями має бути додано експорт *OBJ* формату, щоб користувач міг застосувати ретопологію до вхідного файлу, та отримати коректну топологію на виході у вигляді *OBJ* файлу.

Висновки за розділом

Після завершення розробки програмного додатку, досягнуто функціональні та нефункціональні вимоги, які були описані у розділі 3.1. Аналіз програмного додатку включав в себе: оцінку продуктивності, інтерфейсу користувача, функціональність обробки помилок, розширюваність та можливості розвитку.

Загальний висновок після аналізу програмного коду та тестування підтверджує, що всі аспекти програмного застосунку були належним чином перевірені та всі «вузькі місця» програми відповідають необхідним перевіркам.

До покращення функціональності додатку відноситься: розширення списку підтримуваних форматів, додавання можливостей анімації об’єктів, вдосконалення системи взаємодії із об’єктами на сцені, додавання можливості одночасного завантаження декількох об’єктів на сцену із можливістю застосування булевих операцій. Головною цінністю програмного додатку може стати створення алгоритму ретопології поверхні *3D*-об’єкту.

ВИСНОВКИ

3D-моделювання – це процес розробки математичного координатного представлення будь-якої поверхні об'єкта в трьох вимірах за допомогою спеціалізованого програмного забезпечення шляхом маніпулювання ребрами, вершинами та полігонами в модельованому тривимірному просторі. *3D*-моделі представляють фізичне тіло за допомогою набору точок в тривимірному просторі, які з'єднані різними геометричними об'єктами, такими як трикутники, лінії, криві поверхні і так далі.

Однією з ключових проблем у роботі з тривимірними даними є необхідність уніфікації та стандартизації процесів обробки цих даних. Різноманітність форматів та структур тривимірних об'єктів ускладнює роботу фахівців та унеможливлює обмін даними між різними платформами та програмними продуктами. У зв'язку з цим, однією з ключових мет цього дослідження є розробка універсальних рішень, які сприятимуть інтеграції та обробці тривимірних даних незалежно від їхнього формату.

Залежно від використаного програмного забезпечення та технічних характеристик комп'ютера, тривимірні моделі можна створити за допомогою кількох різних процедур, таких як моделювання на основі алгоритмів, ручне малювання (на комп'ютері) чи сканування.

Існує три способи представлення *3D*-моделей: полігональне моделювання (*polygonal modeling*), криволінійне моделювання (*curve modeling*) та цифрове ліплення (*digital sculpting*).

Етапами розробки *3D*-моделей є: концептуалізація, розробка основної геометрії, модифікація полігонів та топології, додавання текстур, кольорів та інших деталей, рендеринг та пост-процесинг.

3D-моделювання застосовується в: архітектурі та будівництві, комп’ютерних іграх, медицині та біології, промисловості та виробництві, науці та дослідженнях, географії та картографії, рекламі та маркетингу.

Протягом виконання дипломного дослідження було досліджено та вдосконалено програмний додаток, спрямований на обробку та відображення *3D*-об'єктів у форматі *OBJ* із використанням: *Qt* для створення графічного інтерфейсу користувачі; *C++* для написання коду; *CGAL* для застосування геометричних алгоритмів; *OpenGL* для графічної візуалізації об’єктів; *Cmake* для забезпечення кросплатформності; *vcpkg* для встановлення залежних бібліотек.

З врахуванням функціональних вимог, визначених в проєкті, розроблено інноваційні покращення, спрямовані на розширення можливостей додатку та поліпшення його користувацького інтерфейсу.

Додаток структуровано у вигляді шаблону проєктування *Model-View-Controller,* де різні компоненти взаємодіють між собою для досягнення ефективної та розширюваної архітектури.

Під час проєктування архітектури розроблено:

– діаграму компонентів, яка використовується для візуалізації та опису архітектури системи або програмного додатку за допомогою її компонентів і їх взаємодії та вказує на структурні елементи системи та зв'язки між ними;

– діаграму прецедентів, що є однією з ключових складових моделювання системи, яка дозволяє аналізувати та визначати взаємодію між зовнішніми сутностями (акторами) та функціональністю системи (прецедентами). та надає цінний контекст для його розробки інтерфейсу користувача;

– блок-схему, що надає графічне представлення послідовності операцій або процесу за допомогою стандартизованих геометричних фігур та стрілок.

Особлива увага була приділена оптимізації програмного продукту, що призвело до покращення продуктивності та якості взаємодії користувача з додатком. Інтеграція нових технологій графічного відображення та механізмів кешування стали важливим кроком у забезпеченні ефективності.

Приділено увагу геометричним складовим *3D*-моделей, а саме:

– вершина це базовий «будівельний блок» для створення графічних об'єктів, таких як моделі *3D*-об'єктів чи форми в двовимірному просторі. У тривимірній

графіці вершини часто використовуються для визначення позицій точок у просторі *XYZ* (або *XYZW* в деяких випадках);

– текстура це зображення або зразок, який використовується для покриття поверхні *3D*-моделі чи *2D*-графічного об'єкта. Текстури додають деталі та візуальні ефекти до об'єктів, роблячи їх більш реалістичними та привабливими;

– нормаль це вектор, який перпендикулярний до поверхні геометричного об'єкта. Кожна точка на поверхні об'єкта може мати свою нормаль, і цей вектор вказує на напрямок від поверхні в зовнішній простір.

Детально описано такі геометричні структури даних:

– формат файлу *OBJ,* що підтримує визначення геометрії поверхонь об'єктів за допомогою полігональних сіток або за допомогою вільних кривих та поверхонь. Формат файлу *OBJ* має просту фізичну структуру, що складається з рядків, що починаються з ключових слів. Після кожного ключового слова вказуються відповідні параметри та значення;

– *3D*-сітку, що є сукупністю вершин, ребер і граней, які разом утворюють тривимірний об'єкт. Вершини – це координати в тривимірному просторі, ребра – з'єднують дві сусідні вершини, а грані обгортають ребра, утворюючи поверхню об'єкта. *3D*-сітка є фундаментальним компонентом тривимірної графіки, що представляє геометрію поверхні тривимірного об'єкта;

– графічний конвеєр, також відомий як конвеєр відтворення, що є структурою в графіці комп'ютерів, яка визначає необхідні процедури для перетворення тривимірної (*3D*) сцени в двовимірне (*2D*) зображення на екрані. Графічний конвеєр перетворює модель в візуально-сприйнятний формат на дисплеї комп'ютера.

Проведено мануальне тестування та проаналізовано:

– оцінку продуктивності, що складається із вивчення швидкодії та витрат ресурсів та тестування завантаження та відображення об'єктів різного розміру та складності для визначення ефективності додатку;

– інтерфейсу користувача, що складається із вивчення зручності та ергономіки. Додаток надає інтуїтивно зрозумілі елементи управління та забезпечує зручний спосіб взаємодії з користувачем під час взаємодії із об'єктами;

– функціональність обробки помилок, що складається із тестів на виявлення та коректне вирішення хиб, що можуть виникати під час завантаження чи обробки об'єктів у форматі *OBJ*;

– розширюваність та можливості розвитку, що складається із вивчення можливості додавання нових функцій, а також гнучкості системи для адаптації до нових вимог.

Загальний висновок після аналізу програмного коду та тестування підтверджує, що всі аспекти програмного застосунку були належним чином перевірені. Всі «вузькі місця» програми відповідають необхідним перевіркам, і усі можливі виключні ситуації, пов'язані з роботою програми, контролюються.

Проаналізовано можливе покращення додатку, що включає в себе такі аспекти:

– вдосконалення можливостей програми шляхом розширення списку підтримуваних форматів для завантаження та відображення *3D*-об'єктів. Додавання підтримки інших форматів, таких як *STL, FBX* чи *glTF*, *STEP* та *AMF* розширить сферу застосування додатку;

– додавання можливостей анімації об'єктів на сцені. Інтеграція системи ключових кадрів дозволить користувачу створювати та переглядати рухомі *3D*-об'єкти, що відкриє нові перспективи для використання у галузі візуалізації та симуляцій;

– вдосконалення системи взаємодії з об'єктами на сцені. Додавання нових інструментів для вибору, обрання та взаємодії з об'єктами, таких як вибір за допомогою області, розширені опції редагування тощо;

– додавання можливості одночасного завантаження декількох об’єктів на сцену із можливістю застосування булевих операцій (*union, difference, intersection*).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. *3D-modeling* [Електронний ресурс] / Вікіпедія – Електрон. дані – Режим доступу *https://en.wikipedia.org/wiki/3D_modeling* (дата звернення 03.11.2023). – Назва з екрану.

2. *Polygonal modeling* [Електронний ресурс] / Вікіпедія – Електрон. дані – Режим доступу *https://en.wikipedia.org/wiki/Polygonal_modeling* (дата звернення 03.11.2023). – Назва з екрану.

3. *Digital sculpting* [Електронний ресурс] / Вікіпедія – Електрон. дані – Режим доступу *https://en.wikipedia.org/wiki/Digital_sculpting* (дата звернення 04.11.2023). – Назва з екрану.

4. *QT* [Електронний ресурс] / Електрон. дані – Режим доступу *https://www.qt.io/* (дата звернення 15.11.2023). – Назва з екрану.

5. *OpenGL* [Електронний ресурс] / Електрон. дані – Режим доступу *https://www.opengl.org/* (дата звернення 10.11.2023). – Назва з екрану.

6. *Rendering* [Електронний ресурс] / Вікіпедія – Електрон. дані – Режим доступу *https://en.wikipedia.org/wiki/Rendering_(computer_graphics)* (дата звернення 04.11.2023). – Назва з екрану.

7. *Rasterization* [Електронний ресурс] / Вікіпедія – Електрон. дані – Режим доступу *https://en.wikipedia.org/wiki/Rasterisation* (дата звернення 04.11.2023). – Назва з екрану.

8. *How is a 3D Model of a Product Made?* [Електронний ресурс] / Електрон. дані – Режим доступу *https://cgifurniture.com/3d-rendering-guide/3d-model-creation-workflow/* (дата звернення 13.11.2023). – Назва з екрану.

9. *CGAL* [Електронний ресурс] / Електрон. дані – Режим доступу *https://www.cgal.org/* (дата звернення 13.11.2023). – Назва з екрану.

10. *CMake* [Електронний ресурс] / Електрон. дані – Режим доступу *https://cmake.org/* (дата звернення 11.11.2023). – Назва з екрану.

11. *Stroustrup Bjarne. The C++ Programming Language / Bjarne Stroustrup. – 3rd. ed. – New Jersey : AT&T Labs Murray Hill*, 1997. – 1022 p. – 937 c.

12. *Eric Freeman, Elisabeth Freeman, Kathy Sierra and Bert Bates. Head First Design Patterns. – 1005 Gravenstein Highway North, Sebastopol : O'Reilly Media, Inc*., 2004. – 643 c.

13. *Vcpkg* [Електронний ресурс] / Електрон. дані – Режим доступу *https://vcpkg.io/en/* (дата звернення 12.11.2023). – Назва з екрану.

14. *Neal Ford, Mark Richards, Pramod Sadalage and Zhamak Dehghani. Software Architecture: The Hard Parts. – 1005 Gravenstein Highway North, Sebastopol* : *O'Reilly Media, Inc*., 2021. – 425 c.

15. Функціональні та нефункціональні вимоги проєкту [Електронний ресурс] / Електрон. дані – Режим доступу *https://visuresolutions.com/uk/* (дата звернення 13.11.2023). – Назва з екрану.

16. *Component diagram* / Вікіпедія – Електрон. дані – Режим доступу *https://en.wikipedia.org/wiki/Component_diagram* (дата звернення 14.11.2023). – Назва з екрану.

17. *Precedence diagram* / Вікіпедія – Електрон. дані – Режим доступу *https://en.wikipedia.org/wiki/Precedence_diagram_method* (дата звернення 15.11.2023). – Назва з екрану.

18. *Vertex* / Вікіпедія – Електрон. дані – Режим доступу *https://en.wikipedia.org/wiki/Vertex_(geometry)* (дата звернення 16.11.2023). – Назва з екрану.

19. *Texture mapping* / Вікіпедія – Електрон. дані – Режим доступу *https://en.wikipedia.org/wiki/Texture_mapping* (дата звернення 16.11.2023). – Назва з екрану.

20. *Normal* / Вікіпедія – Електрон. дані – Режим доступу *https://en.wikipedia.org/wiki/Normal_(geometry)* (дата звернення 16.11.2023). – Назва з екрану.

21. *Wavefront OBJ* [Електронний ресурс] / Електрон. дані – Режим доступу *https://www.loc.gov/preservation/digital/formats/fdd/fdd000507.shtml#:~:text=The%20O*

BJ%20format%20can%20be,a%203D%20scene%20or%20animation (дата звернення 20.11.2023). – Назва з екрану.

22. *Surface mesh* [Електронний ресурс] / Електрон. дані – Режим доступу *https://doc.cgal.org/latest/Surface_mesh/index.html* (дата звернення 22.11.2023). – Назва з екрану.

23. *Graphics pipeline* [Електронний ресурс] / Електрон. дані – Режим доступу *https://en.wikipedia.org/wiki/Graphics_pipeline* (дата звернення 24.11.2023). – Назва з екрану.

24. *Retopology* [Електронний ресурс] / Електрон. дані – Режим доступу *https://cgifurniture.com/3d-max-retopology-for-furniture-visualization/* (дата звернення 20.11.2023). – Назва з екрану.

25. Положення про дипломні роботи (проєкти) випускників Національного Авіаційного Університету. Київ: НАУ, 2017.

26. Документація. Звіти у сфері науки і техніки. Структура і правила оформлення. ДСТУ 3008-95. Київ.