Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему: «**ОПТИМІЗАЦІЯ РЕНДЕРИНГУ НА МОБІЛЬНИХ ПРИСТРОЯХ ДЛЯ ВІЗУАЛІЗАЦІЇ МОЛЕКУЛЯРНИХ СТРУКТУР**»

Виконав: студент 4-го року навчання,

Освітньої програми «Інженерія програмного забезпечення», 121

Пермяков Андрій Ігорович

Керівник Франків О.О.,_____

старший викладач

Рецензент _____

(прізвище та ініціали)

Секретар ЕК _____

«____» _____ 20____ р.

Київ – 2024

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Факультет інформатики
Кафедра інформатики

ЗАТВЕРДЖУЮ
Викладач кафедри інформатики ,
канд. фіз-мат. наук, доц. _____ Гороховський С.С.
(підпис)
„____”_____2024р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на дипломну роботу
студенту Пермякову Андрію Ігоровичу
факультету інформатики 4 курсу бакалаврської програми
ТЕМА: Оптимізація рендерингу на мобільних пристроях для візуалізації молекулярних структур

Зміст ТЧ до дипломної роботи:
Індивідуальне завдання
Анотація
Вступ
Розділ 1. Теоретичні основи
Розділ 2. Вибір та обґрунтування методу оптимізації
Розділ 3. Опис імплементації та результати
Висновки
Список посилань

Дата видачі „___” _____ 2024 р. Керівник_____
(підпис)
Завдання отримав _____
(підпис)

Календарний план виконання дипломної роботи

Тема: **Оптимізація рендерингу на мобільних пристроях для візуалізації молекулярних структур**

№ п/п	Назва етапу дипломої роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на дипломну роботу.	вересень 2023	
2.	Огляд літератури за темою роботи.	жовтень-листопад 2023	
3.	Проведення аналізу алгоритмів оптимізації рендерингу на графічному процесорі.	грудень 2023	
4.	Дослідження архітектури вихідного додатку на предмет можливості оптимізацій.	січень 2024	
5.	Імплементація обраного алгоритму й оцінка результатів.	січень-березень 2024	
6.	Написання пояснювальної роботи.	березень-квітень 2024	
7.	Аналіз отриманих результатів з керівником.	квітень 2024	
8.	Коригування роботи.	квітень-травень 2024	
9.	Створення слайдів для презентації та написання доповіді.	травень 2024	
10.	Захист дипломної роботи.	травень 2024	

Студент _____
Керівник _____
“ ” _____ _____

ЗМІСТ

АНОТАЦІЯ

Ця робота присвячена вивченню та оптимізації процесу рендерингу молекулярних структур на мобільних пристроях. Головною метою дослідження було покращення швидкодії візуалізації в реальному часі, що вимагає ефективної обробки та відображення складних геометричних форм за умов обмежених ресурсів.

Особлива увага була приділена інтеграції методу оптимізації Frustum culling у проєкт BioViewer, застосунок для iOS з відкритим кодом, реалізований на базі Metal API.

Робота передбачала дослідження відомих підходів до оптимізації ренденрингу, детальний аналіз архітектури програми, імплементацію вдосконалень і порівняння продуктивності до та після впровадження оптимізацій. Результати дослідження демонструють покращення в ефективності роботи застосунку без компромісів щодо якості результатів візуалізації.

ВСТУП

Сучасні мобільні пристрої забезпечують можливість для візуалізації складних даних. Завдяки передовим технологіям, новітні смартфони та планшети можуть відображати детальні інтеркативні зображення складних 3D-моделей, що кілка десятків років тому важко було б уявити навіть на стаціонарних комп'ютерах. Однак, незважаючи на значні технологічні досягнення, мобільні пристрої все ще є переносними приладами, які завжди будуть стикатись з певними обмеженнями обчислювальної потужності, ємності батареї та тепловиділенням, які можуть обмежувати їхню ефективність у виконанні складних завдань візуалізації.

Надмірне використання ресурсів, обмежених на мобільних пристроях, може серйозно негативно вплинути на досвід користувача. Наприклад:

- Погана частота кадрів може зробити інтерфейс додаток неприємним для використання, особливо в застосунках, де потрібна швидка взаємодія, як інтерактивні навчальні застосунки або ігри.
- Швидке розрядження батареї обмежує тривалість роботи додатка, що особливо критично для користувачів, які залежать від своїх мобільних пристроїв протягом дня без доступу до зарядки.
- Надмірне нагрівання пристрою не тільки створює неприємні тактильні відчуття, але й може скоротити загальний термін служби батареї та елементів пристрою.

Теорія обмежень, описана Ілаєм Голдраттом у романі «The Goal» [1] , — це підхід, який допомагає виявити найбільш критичні обмеження, що перешкоджають процесу або системі досягти своїх цілей, та почергово усувати ці обмеження для покращення продуктивності. Він основується на простій ідіомі: «весь ланцюг не міцніший за його найслабшу ланку».

У випадку з мобільним додатком для візуалізації молекулярних структур вищезгадане надмірне використання обмежених ресурсів може бути саме тією слабкою частиною, яка погіршує досвід користувача. Адже незалежно від того, наскільки візуально привабливий, багатофункціональний та корисний може бути застосунок, він може виявитись неприємним у використанні. Наприклад, якщо процес інтерактивного перегляду певної тривимірної моделі схожий на слайд-шоу радше ніж на плавне управління, користувачі швидко втратять інтерес. Так само, якщо температура пристрою зростає до такого рівня, що тактильний контакт з ним викликає дискомфорт, це суттєво погіршує досвід користувача. Крім того, якщо пристрій надто швидко розряджається – додатком скоріше за все не користуватимуться на регулярній основі, принаймні не мобільною версією.

Виходячи з тенденції популярності мобільних пристроїв, ***за мету даної роботи*** було поставлено дослідження та впровадження оптимізації рендерингу молекулярних структур на графічному процесорі. Це дозволить поліпшити досвід користувачів та може слугувати способом просування застосунку.

Мета роботи зумовила наступне ***наукове завдання:***

1. Провести систематичний огляд літератури та наявних технічних рішень для дослідження методів оптимізації рендерингу в контексті мобільних пристроїв.
2. У вихідному проєкті з вікритим кодом, який дозволяє тривимірну візуалізацію молекулярних структур, дослідити архітектуру.
3. Визначити комоненти, куди можливо впровадити оптимізації візуалізації.
4. Розробити алгоритми та імплементувати власне оптимізації.
5. Виконати порівняльний аналіз ефективності використання ресурсів застосунком до та після впровадження оптимізацій.
6. Проаналізувати результати експерименту та запропонувати можливі покращення для подальших досліджень.

Наукова новизна даного дослідження полягає у розробці та імплементації методів оптимізації рендерингу, адаптованих до специфіки мобільних пристроїв. Ці методи впроваджено в проєкт з відкритим кодом, який використовує власну низькорівневу імплеметацію рушія для рендерингу, спеціально розроблений для візуалізації молекулярних структур з використанням графічного процесора мобільного пристрою. Значущість цього дослідження полягає в можливості підвищувати ефективність рендерингу й мінімізувати використання обчислювальних ресурсів та енергії, при цьому забезпечуючи високу якість візуалізованих зображень.

Практичне значення результатів дослідження виявляється у підвищенні продуктивності мобільного додатку, що використовуються для візуалізації складних наукових даних. Таке поліпшення може зробити застосунок більш привабливим для користувачів, які вимагають надійних і ефективних інструментів для роботи з великими обсягами даних. Впровадження цих оптимізацій також може сприяти розширенню функціональних можливостей мобільних пристроїв у наукових дослідженнях та освіті, що робить їх ще більш ефективним інструментом для спеціалістів у галузях кристалографії, хімії, біології та в інших сферах.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ

1.1 Опис представлення молекулярних структур

Молекулярна структура описує фізичне розташування атомів у молекулі. Це охоплює як відстані між атомами, так і природу хімічних зав'язків, які їх утримують. Ця структура відіграє важливу роль для розуміння хімічних та біологічних властивостей молекул, а також їхніх взаємодій з іншими молекулами. Точність визначення молекулярних структур має критичне значення в багатьох галузях, включаючи хімію, фармацевтику, матеріалознавство, біологію тощо. Тривимірна візуалізація дозволяє наочно переглядати й детально досліджувати структуру будови молекули. На рисунку 1.1 у якості прикладу показано візуалізацію будови бурштинової кислоти, хімічна формула якої – CH_6O_4.

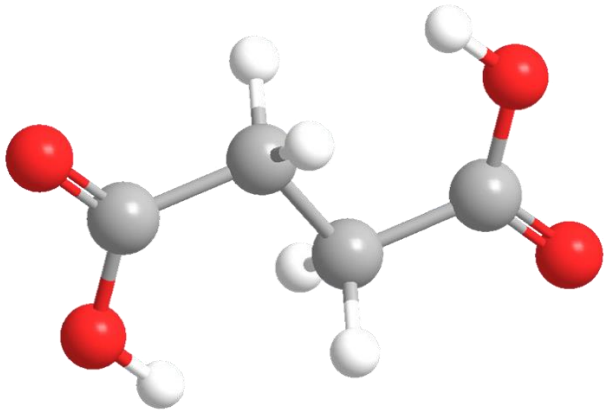

Рисунок 1.1. Приклад візуалізації будови бурштинової кислоти[2]

Мобільні застосунки дозволяють не тільки переглядати структури у тривимірному форматі, але й взаємодіяти з ними в реальному часі. Користувачі можуть повертати, масштабувати, досліджувати молекули з різних боків, налаштовувати кольори для окремих атомів тощо. Це забезпечує глибше розуміння їхньої структури та функцій. Такі застосунки відкривають нові

можливості для освіти та досліджень, дозволяючи студентам і науковцям вивчати складні молекулярні структури безпосередньо зі своїх смартфонів або планшетів. Для зберігання і передачі даних про молекулярні структури використовуються різні формати файлів, кожен з яких має свої особливості та призначення. Ось найпопулярніші з них:

- CIF (Crystallographic Information File)[3]: Цей формат широко використовується у кристалографії. Він є основним форматом для подання даних у наукових публікаціях та базах даних, таких як Cambridge Crystallographic Data Centre (CCDC).
- PDB (Protein Data Bank)[4]: Призначений для зберігання тривимірних структурних даних білків, нуклеїнових кислот, і великих біомолекулярних комплексів. Цей формат файлів є незамінним ресурсом у біоінформатиці і структурній біології.
- XYZ[5]: Простий текстовий формат, який містить координати атомів без інформації про зв'язки. Простота робить цей формат зручним для використання у багатьох молекулярних редакторах та візуалізаторах.

1.2 Модель повного заповнення простору (Space-Filling Model)

Модель повного заповнення простору, відома також як space-filling model[6], є одним із класичних методів візуалізації молекулярних структур. Ця модель відображає атоми у формі сфер, розміри яких визначаються відповідно до їх Ван-дер-Ваальсових радіусів. Це дає можливість візуально відтворити фізичний об'єм, який займає кожен атом у молекулі.

Ван-дер-Ваальсові радіуси описують розміри атомів, коли вони не утворюють хімічних зв'язків, але вступають у слабкі міжмолекулярні взаємодії, такі як ван-дер-ваальсові сили[7]. Ці радіуси представляють собою мінімальні відстані між

нереагуючими атомами і важливі для розуміння, наскільки близько атоми можуть зближуватися один до одного без формування зв'язків. Ван-дер-ваальсові радіуси вимірюються в ангстремах (Å), де 1 Å еквівалентно 10^{-10} метра.

Також використовуються різні кольори для різних типів атомів. Це допомагає легше ідентифікувати складові частини молекули та розуміти їхнє взаємне розташування. На рисунку 1.2 наведено приклад візуалізації за використання цього методу молекули води, хімічна формула якої H_2O.

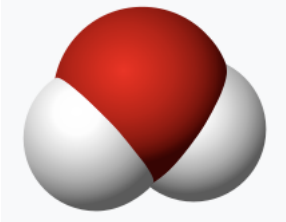

Рисунок 1.2. Приклад візуалізації молекули води у форматі моделі повного заповнення простору[8]. Білі атоми – гідроген, червоний – оксиген.

Попри візуальну інформативність, модель повного заповнення простору може створювати враження перевантаженості при візуалізації великих молекулярних комплексів. Численні перетини атомів у складних структурах можуть спричинити труднощі в сприйнятті загальної картини. Особливо на пристроях з невеликими розмірами екранів, таких як мобільні телефони та планшети.

Ідеальних підходів не існує, тому ця модель, хоч і має певні мінуси, усе ще залишається ключовим інструментом у молекулярній біології та хімії для навчання та досліджень.

1.3 Кулестрижнева модель (Ball-and-stick model)

Кулестрижнева модель є іншим з двох класичних методів візуалізації молекулярних структур. Вона також чудово підходить для наочного зображення

атомів та хімічних зв'язків між ними. У цій моделі атоми представлені у формі куль, розміри яких, на відміну від моделі повного заповнення простору, можуть відповідати або не відповідати Ван-дер-Ваальсовим радіусам, залежно від масштабу моделі. Зв'язки між атомами представлені стрижнями, тобто, циліндрами, що не тільки показують міжатомні відстані, але й тип зв'язку: одинарний, подвійний чи потрійний. Використання різних кольорів для куль та стрижнів допомагають розрізнити різні хімічні елементи та зв'язки. На рисунку 1.3 наведено приклад візуалізації за використання цього методу молекули води, хімічна формула якої H_2O.

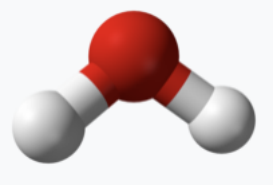

Рисунок 1.3. Приклад візуалізації молекули води у форматі кулестрижневої моделі[8]. Білі атоми – гідроген, червоний – оксиген.

Головна перевага кулестрижневої над моделлю повного заповнення простору полягає у її здатності детально показати типи хімічних зв'язків та їхнє розташування в тривимірному просторі. Це важливо для освітніх цілей та досліджень, оскільки структура зв’язків між атомами молекули тісно пов’язана з її хімічними властивостями. Візуалізація допомагає зрозуміти, як зв'язки впливають на, наприклад, реакційну здатність молекули та інші її властивості.
Хоча кулестрижнева модель є більш інформативною в плані опису з’вязків, вона може створювати некоректні уявлення про реальні розміри атомів. Особливо коли масштаб не відповідає Ван-дер-Ваальсовим радіусам. При візуалізації великих молекулярних комплексів ця модель також може здатися перевантаженою, що ускладнює візуальне сприйняття структурних деталей.

1.4 Метод імопсторів (Impostor Geometry)

Метод імпосторів полягає у створенні двовимірних зображень, які імітують тривимірні об'єкти. Це дозволяє зменшити обчислювальне навантаження без значної втрати візуальної якості.

Imposter geometry використовується для оптимізації рендерингу великих або складних сцен, зменшуючи кількість полігонів, які потрібно обробляти. Атоми або інші молекулярні компоненти представляються у вигляді «імпостерів» – спрощених зображень, які відображаються на плоских поверхнях, що завжди повернуті до користувача. Це дозволяє зберегти ілюзію тривимірності при значно менших вимогах до обчислювальних ресурсів. Рисунок 1.4 показує найпростіший випадок – кулю, відображену звичайним способом і з використанням геометрії імпосторів.

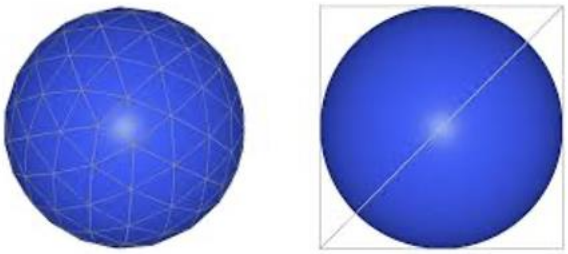

Рисунок 1.4. Зліва – звичайна куля, відоображена за допомогою 320 трикутників. Справа – куля за використання геометрії імпосторів, відображена за допомогою 2 трикутників[9].

У науках, де часто потрібно аналізувати великі об'єми даних, imposter geometry може значно підвищити продуктивність візуалізації. Завдяки меншій кількості деталей, які необхідно рендерити, можна швидше і ефективніше відтворювати великі молекулярні структури.

1.5 Метод «Ходинних кубів» (Marching Cubes)

Виникає очевидне питання: за яким алгоритмом переводити тривимірні моделі в двовимірні зображення? Відповідь на нього ще в 1987 році дали Вільям Лоренс та Харві Клайн у науковій статті під назвою «Marching cubes: A high resolution 3D surface construction algorithm»[10]. Описано алгоритм під назвою Marching Cubes. Простір ділиться на сітку одиничних кубів – вокселів. Очевидно, що кожен воксель має вісім вершин. Для кожного вокселя визначається, які з його восьми вершин перебувають всередині ізо-поверхні, а які — поза нею. Ізо-поверхнею у контексті цього документу називатимемо тривимірну форму, яка описує межі молекули.

Зрозуміло, що кожна вершина має рівно два стани: або всередині (0), або зовні (1) ізо-поверхні. Відповідно для кожної з восьми вершин маємо два значення, тому повну конфігурацію можна описати $2^8 = 256$ варіантами, тобто єдиною 8-бітною маскою. Тому індекс генерується шляхом об'єднання бінарних значень належності вершин вершин вокселя до поверхні у число від 0 до 255.

Окрім вершин, для побудови трикутників використовуються також середини ребер. Кожен воксель має 12 ребер, кожне з яких описане двома вершинами вокселя. Тут і вібувається апроксимація: якщо одна вершина ребра перебуває всередині ізо-поверхні, а інша – поза нею, то на цьому ребрі існує точка перетину ізо-поверхні. Незалежно від того де в реальному світі на відрізку ребра знаходиться ця точка, вона позначається рівно на його середині.

Таблиця шаблонів конфігурацій тримає в собі рівно 256 варіантів, тому кожен індекс відповідає шаблону конфігурації, який визначає, як з'єднати середини ребер куба для формування трикутників. Усього є 15 базових конфігурацій, показаних на рисунку 1.5, які з урахуванням симетрії утворюють 256 можливих

варіантів. Таблиця конфігурацій дозволяє швидко знайти відповідний набір трикутників для будь-якої конфігурації вокселя.

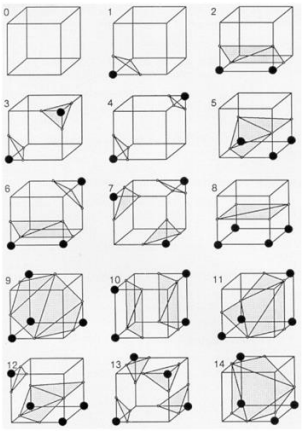

Рисунок 1.5. Показує 15 базових конфігурацій вокселів[10].

За допомогою шаблонів з таблиці конфігурацій точки перетину з'єднуються у трикутники. Ці трикутники формують маленьку частинку геометричної поверхні, що відповідає вокселю. Трикутники всіх вокселів об'єднуються, щоб сформувати повну ізо-поверхню в тривимірному просторі. Результатом є тривимірна фігура, що точно відтворює межі ізо-поверхні.

Початково алгоритм Marching Cubes був спеціально розроблений для візуалізації медичних сканів, таких як комп'ютерна томографія (КТ) та магнітно-резонансна томографія (МРТ). Завдяки здатності відтворювати складні форми з високою точністю, він швидко став популярним навіть за межами візуалізації медичних сканів.

Очевидно, що через апроксимації на етапі обрахувань вершин трикутників на ребрах, алгоритм може створювати артефакти у вигляді «ступінчастих» поверхонь. Особливо це помітно при низькій роздільній здатності (тобто, малій кількості вокселів). Таку проблему можна віришити за допомогою зменшення сторони вокселя або ж додаткової обробки для згладжування.

Ці особливості роблять Marching Cubes важливим, але водночас специфічним інструментом. Як і з практично будь-яким іншим інструментом, вибір для конкретного застосування має базуватися на оцінці його переваг і обмежень з урахуванням контексту задачі.

1.6 Vertex-Connected Marching Cubes (скорочено – VCMC)

Професори Dong Xu та Yang Zhang у статті «Generating Triangulated Macromolecular Surfaces by Euclidean Distance Transform»[10] представили альтернативу до Marching Cubes – Vertex-Connected Marching Cubes, тобто метод ходинних кубів з вершинним з'єднанням. Основна його відмінність від традиційного алгоритма полягає в способі формування трикутників. Оригінальна версія використовує вершини й середини ребер вокселів, тоді як ця версія базує трикутники суто на вершинах.

Цей підхід сильно зменшує кількість полігонів в фінальній репрезентації. На рисунку 1.6 показано 23 базові конфігурацій, які з урахуванням симетрії дають усі можливі 256 варіанти. Легко побачити, що за умови нормалізованих сторін вокселів, довжини сторін трикутників обмежуються трьома значеннями: 1 – сторона на ребрі, $\sqrt{2}$ – сторона на діагоналі грані та $\sqrt{3}$ – сторона на діагоналі всього вокселя.

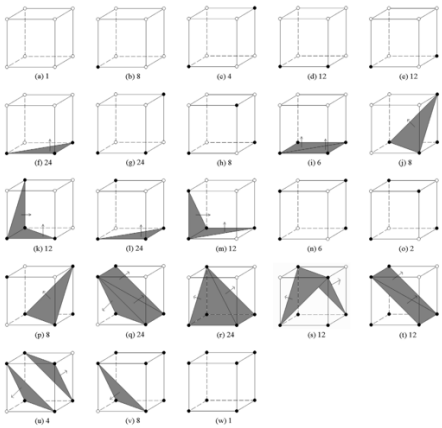

Рисунок 1.6. Показує 23 базові конфігурації вокселів[11]

Хоча використання лише вершин може дещо знизити точність відображення складних молекулярних структур, це також робить алгоритм більш гнучким. Зміною розміру вокселів можна адаптувати детальність візуалізації, що дозволяє досягти оптимального балансу між якістю зображення та вимогливістю до обчислювальних можливостей.

Використання простіших конфігурацій у VCMC дозволяє значно скоротити використання пам'яті та знизити навантаження як на графічний, так і на центральний. Це робить метод підходящим для обробки великих даних на системах з обмеженими ресурсами.

Таким чином VCMC є цінним інструментом для застосувань, де необхідно швидко обробляти великі об'єми тривимірних даних з регульованою точністю, таких візуалізація складних молекулярних структур на мобільних пристроях.

1.7 Переваги методу імпосторів у поєднанні з VCMC

У початковому проєкті, який було обрано для впровадження в нього подальших оптимізацій та покращень, використовується підхід імпосторів в поєднанні з

методом ходинних кубів з вершинним з’єднанням. Детальніше про застосунок описано у розділі 3.1 цього документу.

Метод імпосторів, зокрема використання білбордних квадратів, зменшує обчислювальне навантаження, дозволяючи системі відображати складні молекулярні структури з меншою кількістю обчислень. Білбордні квадрати – це двовимірні графічні елементи, які завжди обернені обличчям до спостерігача, незалежно від кута огляду. Кожен атом представляється у вигляді двовимірного зображення, яке зберігає ілюзію тривимірності, але вимагає значно менше ресурсів для обробки і відображення.

Vertex-Connected Marching Cubes дозволяє достатньо точно й оптимально відтворювати молекулярні поверхні, використовуючи сітку, яка генерується на основі обчислення відстаней між атомами. Цей метод детально відображає межі і форми молекул, що є важливим для розуміння їхньої біологічної функціональності та взаємодій. Таке поєднання технік дозволяє швидко і ефективно візуалізувати великі молекулярні структури, надаючи важливу інформацію для наукових досліджень та освітніх програм.

РОЗДІЛ 2. ВИБІР ТА ОБҐРУНТУВАННЯ МЕТОДУ ОПТИМІЗАЦІЇ

2.1 Огляд методів оптимізації рендерингу

Оптимізація рендерингу є важливою частиною розробки застосунків для візуалізації даних, тим паче в умовах мобільних пристроях, де ресурси обмежені. Для підвищення продуктивності і зменшення навантаження на обчислювальну систему вигадано різноманіття методів. У цьому підрозділі корото описані лише деякі з них.

Одним з таких методів є Level of Detail (LOD) - техніка, що адаптує деталізацію об'єктів в залежності від їхньої віддаленості від камери. Що далі від спостерігача об'єкт, то гірша якість і відповідно менше ресурсів іде на нього й навпаки. Якщо правильно зробити, вища продуктивність забезпечується без втрати якості візуалізації для користувача.

Інший підхід, Geometry Instancing - метод, який дозволяє ефективно рендерити велику кількість однотипних об'єктів. Власне оптимізація полягає в тому, що всі вони рендеряться одночасно єдиною операцією й це зменшує обчислювальне навантаження.

Техніка Occlusion Culling відбраковує з процесу рендерингу об'єкти, які перекриті іншими об'єктами і не видимі користувачу. Зрозуміло, що це зменшує кількість обчислень.

Метод Frustum Culling також допомагає відбракувати з обчислень об'єкти. На відміну від попереднього, цей підхід відкидає об'єкти, які знаходяться за межами поля зору камери.

Схожа на попередні дві, Backface Culling - оптимізація, яка не обробляє ті сторони полігонів, які повернені від користувача. Варто зауважити, що цей підхід може відкидати об'єкти не повністю, а лишень їх задню частину.

Техніка Mipmapping схожа на вищеописану LOD. Відрізняється вона тим, що зосереджується на оптимізації саме текстури, а не складності геометрії моделі. Цей метод полягає у створенні текстур різної роздільної здатності. Вони й підбираються відповідно до ступеню приближення або, наприклад, розміру екрану.

Підхід Dynamic Tessellation теж схожа на LOD. Проте, слово dynamic у назві означає, що геометрія моделі ускладнюється або спрощується динамічно в реальному часі. Тобто цей підхід не використовує заздалегідь заготовлені варіанти, як LOD.

У наступних розділах цього документу буде здійснено більш детальний огляд деяких вибраних методів оптимізації рендерингу, які виявилися найбільш релевантними для проєкту для візуалізації молекулярних структур. Зокрема, будуть розглянуті такі техніки, як Level of Detail (LOD), Geometry Instancing, Occlusion Culling, Frustum Culling, та Backface Culling. Кожен з цих підходів, як було описано в цьому розділі, має свої унікальні переваги та області застосування.

2.2 Техніка level of detail (LOD)

Техніка Level of Detail (LOD) є одним з ключових методів оптимізації в комп'ютерній графіці. Вона призначена для підвищення продуктивності рендерингу шляхом адаптування рівня деталізації об'єктів. Застосування LOD дозволяє системі витрачати менше ресурсів на об'єкти, які знаходяться далеко від камери та більше для тих, що розташовані ближче.

Загальна ідея LOD полягає в створенні кількох версій геометрії одного і того ж об'єкта з різним рівнем деталізації, що в принципі можна зрозуміти з назви. Для високодеталізованих моделей, що будуть показуватись з близької відстані, буде

більше полігонів. Для спрощених версій, які будуть використовуватись для далеких планів – менше. Алгоритм управління рівнем деталізації відстані від камери до об'єкта, а також можу враховувати кут огляду й можливо навіть поточні вимоги до продуктивності системи. Коли цей алгоритм якісно упроваджений в систему і їй в якийсь момент життєвого циклу стає недостатньо ресурсів, можна відносно легко погіршити якість деякиї або й усіх моделей.

На Рисунку 2.1 представлено зображення зайця, який відображається з чотирма різними рівнями деталізації. При поточному рівні збільшення лише лівий виглядає прийнятно, проте при достатньому віддаленні найправішу картинку буде важко або й неможливо відрізнити від найлівішої. При цьому різниця кількості полігонів на них – більш ніж в 900 разів.

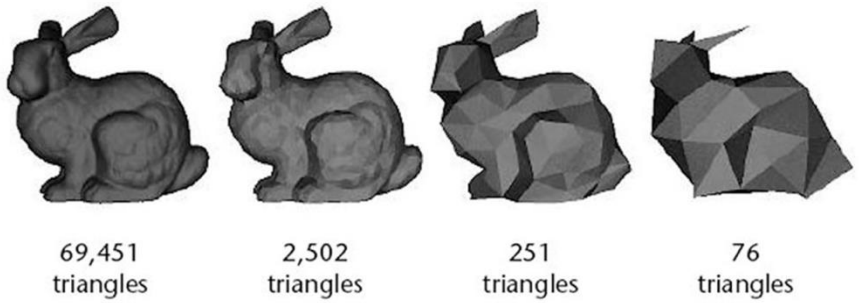

Рисунок 2.1. Моделі зайця з різною кількістю полігонів[12].

2.3 Техніка occlusion culling

Occlusion Culling – іще одна з класичних технік оптимізації в комп'ютерній графіці. Вона підвищує продуктивність рендерингу шляхом відбракування об'єктів, які не видимі користувачеві через перекриття іншими об'єктами. Застосування цього підходу може бути особливо корисним у великих і складних тривимірних середовищах, де об'єкти часто перекривають один одного, як-от для тривімірних візуалізацій молекул.

На Рисунку 2.2 представлено схематичну ілюстрацію того як occlusion culling визначає видимість об'єктів у просторі. Камера спрямована на сцену, де деякі об'єкти зображені пунктиром, оскільки вони перекриті іншим об'єктом і таким чином не потребують обробки в рамках поточного кадру рендерингу.

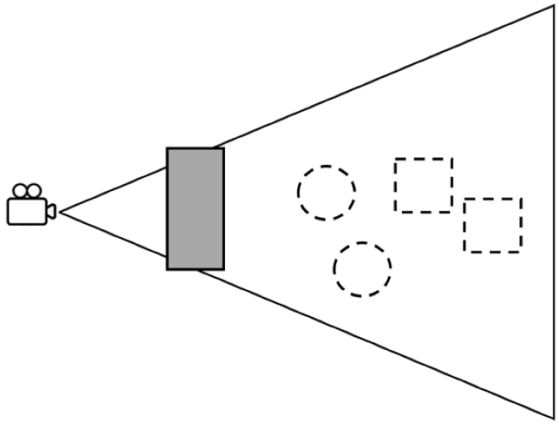

Рисунок 2.2. Occlusion culling. З камери видно область, обмежену трикутником. Сірий прямокутник загороджує інші об’єкти (замальовані пунктирними лініями), тому вони вібраковуються й не рендеряться.[13]

Розглянемо простий алгоритм occlusion culling. Маємо сцену, де камера спостерігає за множиною об'єктів. Кожен об'єкт може бути огорнутий у обмежувальну рамку (bounding box), який використовується для швидкої перевірки перекриття. Під час рендерингу, для кожного об'єкта спочатку виконується перевірка, чи є його обмежувальна рамка видимою. Якщо вона перекрита іншими об'єктами, то й весь об'єкт можна безпечно відбракувати з подальшого рендерингу. Такий підхід використано як один з доступних онлайн демо-прикладів[14] з книги Еріка Хейнса та Наті Гоффмана «Real-Time Rendering, Fourth Edition»[15].

Більш складні реалізації occlusion culling можуть включати використання додаткових специфічних структур даних, наприклад, дерев просторового

розбиття. Вони дозволяють ефективніше управляти великими сценами й оптимальніше визначати, які об'єкти повинні бути відображені, а які можуть бути ігноровані.

Ефективна імплементація occlusion culling може значно знизити кількість операцій рендерингу, необхідних для візуалізації сцени. Утім, важливо не забувати враховувати вплив алгоритмів на загальну продуктивність системи. Потрібно чітко розуміти чи дійсно такий підхід дає позитивний впив у конкретному проєкті, адже обрахування того які об’єкти відбраковувати теж далеко не безкоштовне.

2.4 Техніка frustum culling

Frustum culling є ще однією з фундаментальних технік у комп'ютерній графіці для підвищення ефективності рендерингу. Ця техніка забезпечує оптимізацію шляхом відбракування з процесу рендерингу об'єктів, які не потрапляють у поле зору камери.

Суть frustum culling полягає у визначенні видимості об'єктів відносно так званої «view frustum» – зрізаної піраміди зору камери. Формується вона з шістьох площин: права, ліва, нижня, верхня, ближнья та глибинна. Усі об'єкти, які не перетинаються з цією пірамідою, відбраковуються з рендерингу. Це й зменшує загальну кількість полігонів, які потрібно буде обробити. Таку зрізану піраміду продемонстровано на рисунку 2.3.

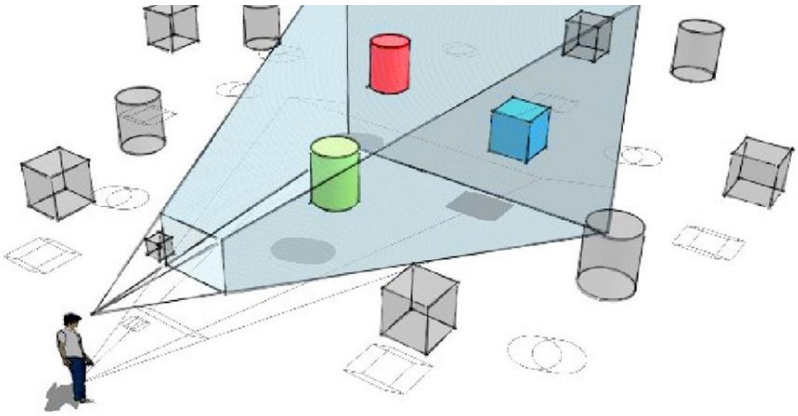

Рисунок 2.3. Приклад frustum culling. Об’єкти, які не попадають в зрізану піраміду зору позначені сірим і не рендеряться.

Процес реалізації frustum culling може включати обчислення перетинів об'єктів із площинами зрізаної піраміди зору за допомогою геометричних перевірок. У найпростішій реалізації це вимагає, щоб кожен об'єкт мав визначену обмежувальну рамку (bounding box), яка регулярно оновлюється залежно від позиції та орієнтації об'єкта у тривимірному просторі. За допомогою цих обмежувальних рамок легко і швидко перевіряти, чи потрапляє об'єкт до видимої зони.

Як і будь-яка оптимізаційна техніка, frustum culling вимагає обдуманого застосування. Неправильне використання може призвести до зайвих обчислень, що перевищують виграш від самого відбракування, а хибна імплементація – навіть до втрати певних візуальних деталей.

Як приклад не повністю обдуманого підходу можна навести один з підходів у книзі Трента Полака «Focus On 3D Terrain Programming»[16]. Ця кни га детально описує низькорівневе створення й управління 3д-ландшафтом за допомогою OpenGL. У розділі про frustum culling автор описує настпний алгоритм: простір розділяється на рівні куби. У цій книзі вони називаються патчами. Потім виконується перевірка для кожного з них на предмет того, чи він належить зірзаній підаміді зору. Алгоритм перевірки наступний: якщо жодна з 8 вершин

патча не належить простору зрізаної піраміди, то всю геометрію, яка належала цьому патчу можна відбракувати. На перший погляд здається, що логіка правильна, проте легко уявити куб простору, частина однієї площини якого належить зрізаній піраміді й при цьому жодна з вершин не належить. На рисунку 2.4 показано приклад ландшафту, де в лівому нижньому кутку відбракувались зайві трикутники через хибну імплементацію алгоритму.

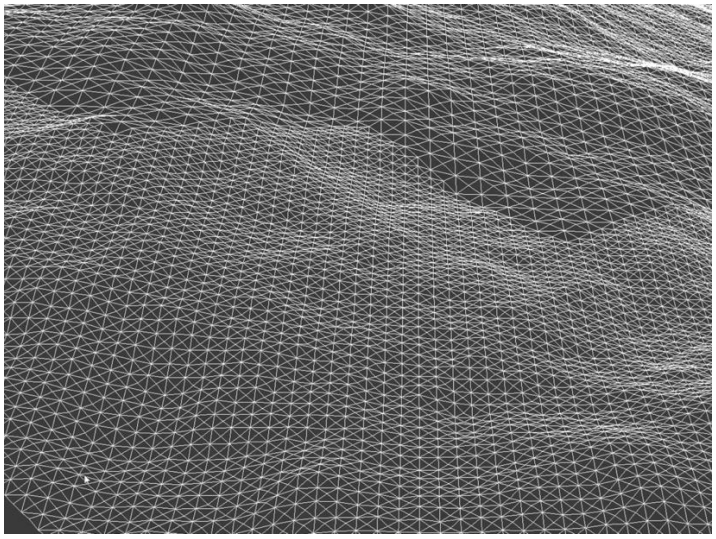

Рисунок 2.4. Приклад хибної імплементації frustum culling.

2.5 Техніка backface culling

Backface culling є технікою оптимізації в комп'ютерній графіці, яка вібраковує з процесу рендерингу ті полігони об'єктів, що звернені від користувача або камери. Цей метод базується на математичному аналізі нормалей полігонів. Нормаль полігону – це одиничний вектор, який перпендикулярний до його поверхні і вказує назовні об'єкту. Якщо кут між нормаллю полігону і напрямком зору камери є більшим ніж 90 градусів, це означає, що полігон звернений від користувача і його можна безпечно ігнорувати під час рендерингу.

Backface culling відсікає близько половини полігонів з видимої частини сцени. Це дозволяє сильно зменшити кількість обчислень, які виконує графічний процесор, що особливо ефективно у випадку великих і складних тривимірних об'єктів. На рисунку 2.5 демонструється результат backface culling

Рисунок 2.5. Приклад backface culling. Червоні пікселі будуть рендеритись, чорні – ні[17].

Однак, очевидно, техніка має певні обмеження. По-перше, backface culling може бути неефективним або навіть привести до помилок візуалізації у сценаріях, де полігони об'єкту є напівпрозорими. Додаткові обрахунки для перевірки таких випадків можуть переважити позитивний вплив оптимізації. По-друге, ця техніка може не принести покращення продуктивності, якщо більшість полігонів в сцені і так спрямовані до камери.

2.6 Обґрунтування вибору методу Frustum Culling

Очевидно, що можна комбінувати різні способи оптимізацій. На рисунку 2.6 наведено візуалізацію усіх вищеописаних підходів до відбраковки на єдиній сцені.

Кожен окремий підхід заслуговує окремої уваги до нюансів, потребує оцінки того, наскільки виграш від відбраковки чи зміни деталізації перевищує накладні

витрати на обрахунки алгоритмів та на власне вибір й імплементування цих алгоритмів. У цьому документі було вирішено зосередитися на одному типі оптимізації – frustum culling.

Основним фактором при виборі методу оптимізації, як це й має в ідеалі бути, стала специфіка імплементації проєкту, у який їх потрібно буде впроваджувати. Як вже було описано в розділі 1.7, використовується метод імпосторів, зокрема білбордних квадратів. Припинити рендерити білборди поза зрізаною пірамідою зору, особливо, коли користувач збільшує зображення, є першим кроком, який спадає на думку.

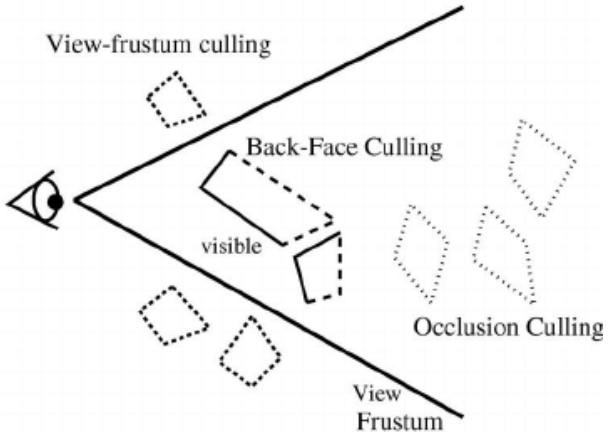

Рисунок 2.6. Комбінація з frustum, backface та occlusion culling[18].

Варто також зазначити, що у файлі README.md[19] проєкту frustum culling було відмічена як потенційна оптимізація, яка могла б бути реалізована в майбутньому. Це ще раз підтверджує важливість та актуальність даної техніки в рамках розглянутого проєкту. Тому вибір саме цього методу оптимізації став логічним продовженням дослідження можливостей підвищення ефективності рендерингу.

РОЗДІЛ 3. ОПИС ІМПЛЕМЕНТАЦІЇ ТА РЕЗУЛЬТАТИ

3.1 Опис обраного проєкту

У рамках практичної частини роботи були прийнято рішення покращити функціональність проєкту, що забезпечує рендеринг молекулярних структур на мобільних пристроях, який вже існує. Було обрано iOS застосунок BioViewer[19], який є продуктом з відкритим кодом. Завдяки введенню обраної техніки оптимізації відбулося як покращення досвіду користувача, так і розширення можливостей додатку.

Для побудови інтерактивних частин інтерфейсу користувача: кнопок навігації та налаштувань, він використовує SwiftUI – інноваційний іпростий спосіб створення інтерфейсів на всіх платформах Apple на мові Swift[20]. Найважливіше для нас тут те, що для рендерингу молекулярних структур використовується власна імплементація рушія на Metal.

Metal – це низькорівневий, високопродуктивний графічний та обчислювальний фреймворк, розроблений компанією Apple для iOS, macOS та tvOS. Він надає розробникам низьконакладний, практично прямий, доступ до графічного процесора (GPU). Це дозволяє максимально оптимізувати рендеринг графіки та виконання паралельних обчислень з мінімальними накладними витратами.

Термін «шейдер» в комп'ютерній графіці має походження з роботи, проведеній Робом Куком в Pixar. Він розробив перший програмований шейдер, який був важливим кроком у створенні фотореалістичних комп'ютерних зображень[21]. Їх назва може вводити в оману й вони часто асоціюються з тінями або освітленням в контексті графіки. Насправді ж шейдери – це спеціальзовані комп’ютерні програми, які контролюють більш широкий спектр властивостей поверхонь,

включаючи колір, текстуру і відблиски, використовуючи математичні формули для опису цих ефектів.

Для написання шейдерів у контексті Metal розроблено Metal Shading Language (MSL). Строго кажучи, MSL – це специфікація на основі C++14 із розширеннями та обмеженнями[22].

Як описано в розділі 2.6, було обрано впровадити в цей проєкт оптимізацію frustum culling. Наступні підрозділи цього документу присвячені роз'ясненням різних підходів і технік, які були застосовані для досягнення цього, проблем, що виникали й аналізу результатів.

3.2 Принципи роботи конвеєру рендерингу на Metal

Конвеєр рендерингу на Metal, або rendering pipeline, представляє собою послідовність обробки команд рендерингу, яка закінчується записом даних у цільові буфери для зображень. Основні етапи цього конвеєра включають вершинну стадію, стадію растеризації та фрагментну стадію. Вершинна та фрагментна стадії програмуються з використанням Metal Shading Language (MSL), тоді як стадія растеризації має фіксовану поведінку[23].

Конфігурація конвеєра рендерингу є найважливішем етапом в процесі визначення того, як дані оброблятимуться на GPU. Рендеринг починається з команди рендерингу – виклику на CPU функції drawPrimitives або її варіацій. Цей виклик визначає скільки вершин обробити та який тип графічних примітивів малювати, наприклад, трикутники або лінії.

На стадії вершин кожній вершині присвоюються певні дані за допомогою функції, означеної в MSL як `vertex`. Ця стадія приймає дані від CPU, які зазвичай передаються через буфери (наприклад, MTLBuffer), які можуть включати координати вершин, кольори, текстурні координати та інші атрибути. Дані в

таких буферах можуть бути організовані в структури або масиви. Загалом, MTLBuffer – це просто неперервний відрізок пам’яті, який можна об’явити на CPU й прочитати на GPU. Metal не знає нічого про його вміст, лише про розмір[24].

Після обробки необхідної кількості вершин оброблені та трансформовані дані проходять стадію растеризації. Тут визначається які пікселі на екрані відповідають кожному примітиву. Далі фрагментна стадія отримує дані від стадії растеризації. За допомогою функції фрагментів, означеної в MSL як `fragment`, на цій стадії визначаються кінцеві кольори пікселів для відображення на екрані.

3.3 Обчислення видимості на CPU з використанням буферів індексів

У вихідному проєкті дані про всі атоми зберігаються в єдиному MTLBuffer і рендеряться одночасно. Ініціація відмалювання сфер відбувається викликом функції drawIndexedPrimitives. Основна відмінність її від функції drawPrimitives полягає у тому, drawPrimitives дозволяє рендерити геометрію за допомогою прямого читання вершин з буфера, що призводить до дублювання даних для спільних вершин у різних примітивах. Наприклад, якщо два трикутники утворюють квадрат, то ця фукнція використовуватиме шість вершин. Натомість, drawIndexedPrimitives використовує індексний буфер, що дозволяє ефективно перевикористовувати вершини, значно зменшуючи обсяги переданих даних і оптимізуючи рендеринг. Тобто для того ж квадрату тут потрібно буде передати чотири вершини і шість індексів.

Найперший підхід для впровадження frustum culling, який було застосовано – відфільтрувати буфер індексів ще на CPU при описі конвеєру рендерингу.

Перша ітерація полягала в тому, щоб відрендерити лише першу половину всіх атомів – просто для того, щоб зрозуміти, чи такий підхід працюватиме.

На лістінгу 3.1 показано код для рендерингу першої половини. Примірник MTLBuffer, у якому зберігаються індекси примітивів для всіх атомів називається impostorIndexBuffer. На першому рядку обраховується половина його розміру – halfSize. У розділі 3.2 було пояснено, що сам фреймворк Metal знає лише про розмір буферу, але нічого про типи, які в ньому зберігаються.

```
let halfSize = impostorIndexBuffer.length / 2

guard let newBuffer = device.makeBuffer(length: halfSize)
else { return }

memcpy(
    newBuffer.contents(),
    impostorIndexBuffer.contents(),
    halfSize)

renderCommandEncoder.drawIndexedPrimitives(
    type: .triangle,
    indexCount: indexBufferRegion.length / 2,
    indexType: .uint32,
    indexBuffer: newBuffer,
    indexBufferOffset: indexBufferRegion.offset * MemoryLayout<UInt32>.stride
)
```

Лістінг 3.1. Код імплементації рендерингу першої половини примітивів.

Далі створюється новий буфер розміром з halfSize. Для створення MTLBuffer, як і для багатьох інших типів у Metal потрібно використати примірник MTLDevice і його методи-фабрики. Кожен MTLDevice репрезентує фізичний GPU[25]. Далі за допомогою системної функції мови Swift копіюється пам’ять з impostorIndexBuffer до newBuffer розміром у halfSize. Після цього викликом drawIndexedPrimitives на renderCommandEncoder ініціюється відмалювання індексованих примітивів. MTLRenderCommandEncoder, примірником якого є renderCommandEncoder – компонент, який відповідає за запис команд рендерингу до командного буфера: він встановлює буфери, вибирає потрібний MTLRenderPipelineState для вказання конкретних вершинних і фрагментних

функцій тощо. Зокрема, він викликає функції малювання такі, як `drawPrimitives` та в цьому випадку ` drawIndexedPrimitives`.

У `indexBufferRegion` зберігаються значення довжини й відступу для буферу індексів усіх примітивів усіх атомів.

У виклику ` drawIndexedPrimitives` встановлюємо наступні параметри: тип примітивів – трикутники; кількість індексів – половина від тих, що були; тип індекса – 32-бітне беззнакове ціле число; буфер – новий відфільтрований нами; відступ буфера – той же, що й був.

На рисунку 3.1 зображено поряд візуалізацію без цього підходу і таку, де рендериться лише перша половина примітивів. Очевидно, що це спрацювало, тому за логікою має спрацювати й відфільтрування примітивів, які стосуються атомів, що знаходяться поза камерою. Здавалося б, для цього потрібно просто написати саму логіку фільтрації й оптимізація готова.

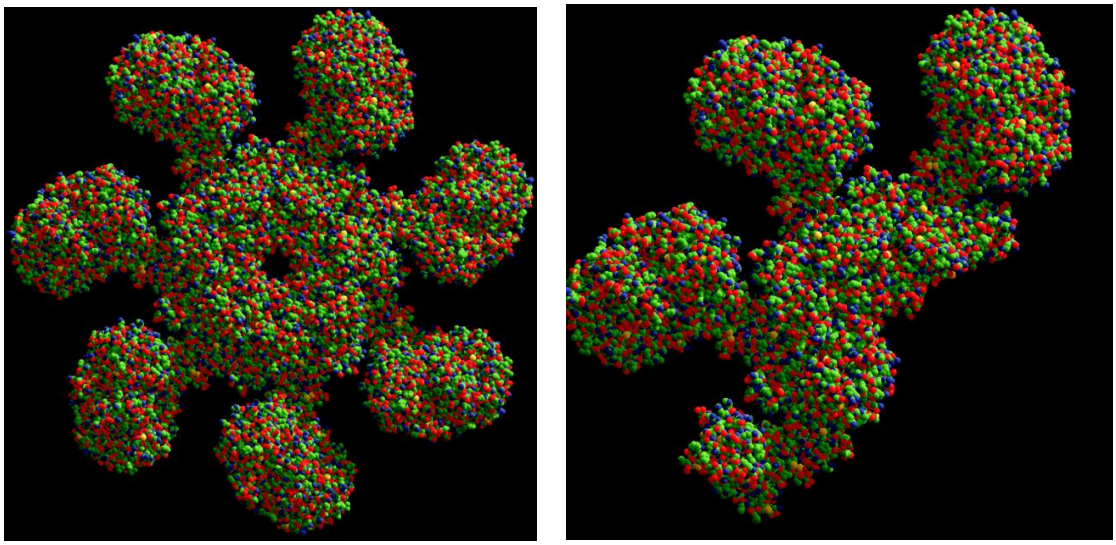

Рисунок 3.1. Рендеринг базової молекули повністю (зліва) та першої половини її примітивів (справа)

Наступною ітерацією було спробувати рендерити частину примітивів, проте в цей раз використовувати предикат для визначення того відкидати індекс чи ні.

Такий підхід, хоч і схожий на попередній, сильно відрізняється. Як описано в розділі 3.2, MTLBuffer – це просто відрізок в пам’яті. Тому прибрати з нього першу чи другу половину дуже легко й головне швидко, а от перебирати й створювати новий буфер з відфільтрованими відповідно до предикату значеннями – уже довше.

На лістингу 3.2 показано першу половину коду для створення нового буфера індексів, застосовуючи предикат для кожного індекса. Спочатку розраховується загальна кількість індексів totalIndices – довжина буфера в пам’яті ділена на розмірність типу 32-бітного цілого числа. Далі створюється типізований указник на буфер індексів. Потім ініціалізується порожній масив для відфільтрованих індексів, після чого відбувається прохід по всіх індексах і ті, які підходять за предикатом isInFrustum додаються до результату, який буде відправлятись на рендеринг.

```
let totalIndices = impostorIndexBuffer.length / MemoryLayout<UInt32>.stride
let srcPointer = impostorIndexBuffer.contents().assumingMemoryBound(to: UInt32.self)

var filteredIndices: [UInt32] = []

for i in 0..<totalIndices {
    if isInFrustum(index: i) {
        filteredIndices.append(srcPointer[i])
    }
}
```

Лістінг 3.2. Перша половина коду фільтрації буферу індексів за допомогою предикату.

На лістингу 3.3 – друга половина коду. Вираховується розмірність для нового буфера й відповідно ініціалізується сам newBuffer. Після цього він відправляється в якості аргумента до вищеописаної функції drawIndexedPrimitives.

```
let dataSize = filteredIndices.count * MemoryLayout<UInt32>.stride
guard let newBuffer = device.makeBuffer(bytes: filteredIndices, length: dataSize)
else { return }

renderCommandEncoder.drawIndexedPrimitives(
    type: .triangle,
    indexCount: indexBufferRegion.length,
    indexType: .uint32,
    indexBuffer: newBuffer,
    indexBufferOffset: indexBufferRegion.offset * MemoryLayout<UInt32>.stride)
```

Лістінг 3.3. Друга частина коду фільтрації буферу індексів за допомогою предикату.

Для тестування цього підходу спершу функція isInFrustum повертала значення true незалежно від параметру. Очікувано, уся структура рендерилась так, як і без застосування цього підходу. Проте вже тут стало зрозуміло, що накладні витрати на CPU переважають будь-яку можливу оптимізацію навіть без застосування алгоритму для обрахування належності атом до зрізаної піраміди зору. З описаним підходом програма працювала з продуктивністю в приблизно 8 кадрів на секунду. У тій же ситуації оригінальна програма видававла близько 90 кадрів на секунду. Очевидно, що з таким підходом далі рухатись не мало сенсу.

3.4 Оптимізація фільтрації буферу індексів на CPU

Підхід, описаний у підрозділі 3.3 виявився провальним через надмірні витрати на CPU. Тому було прийнято рішення спробувати оптимізувати фільтрацію індексів, яка відбувалась лінійно, за допомогою розподілу роботи на різні потоки.

У Swift існують різні способи роботи з багатопоточністю. Рекомендований – використання вбудованого механізму черг диспатчеризації – DispatchQueue. Для виконання коду з черги система самостійно обирає потік, який буде оптимально використати на даний момент, щоб мінімізувати затримки і забезпечити рівномірне розподілення робочого навантаження між ядрами процесора. Такі

черги є двох типів – послідновні й паралельні. Їх різниця зрозуміла з назв і очевидно, що для роботи на різні потоки потрібно використовувати паралельну. На лістінгу 3.4 показано першу частину оновленогу коду, який тепер розподіляє фільтрацію індексів на різні потоки. Кількість індексів і типізований указник на буфер визначаються так само, як і до цього. Машина, на якій відбувалось тестування мала вісім ядер, тому для початку кількість частинок, на які розбивати дані теж була рівна восьми. Далі вираховується розмір частинки chunkSize як кількість індексів ділена на кількість частин і виділяється пам’ять під двови виділяється пам’ять під двовимірний масив для проміжного збереження результатів. У циклі на паралельній DispatchQueue запускаються асинхронно обрахунки для кожної частинки.

```
let totalIndices = impostorIndexBuffer.length / MemoryLayout<Int32>.stride
let srcPointer = impostorIndexBuffer.contents().assumingMemoryBound(to: Int32.self)

let numChunks = 8
let chunkSize = totalIndices / numChunks
var chunks = Array(repeating: [Int32](), count: numChunks)

for chunkIndex in 0..<numChunks {
    queue.async(group: group) {
        let start = chunkIndex * chunkSize
        let end = (chunkIndex == numChunks - 1) ? totalIndices : start + chunkSize
        var subArray = [Int32]()
        subArray.reserveCapacity(chunkSize)
```

Лістінг 3.4 Перша частина коду оптимізації для паралельної фільтрації індексів.

Для того, щоб синхронізувати результат після того як усі частини будуть обраховані, використовується group, що є примірником DispatchGroup – інструменту для координації грип асинхронних завдань, що дозволить відслідкувати їх загальне завершення. Усередині блоку коду для окремої частинки вираховуються індекси її початку й кінця – start та end відповідно. Далі створюється порожній масив subArray для результату цієї порції. Потім за

допомогою функції reserveCapacity для нього алокується місце розміром з кількість елементів у цьому відрізку. Це потрібно для уникнення ітеративної реалокації масиву при великій кількості викликів append на ньому. Спосіб з послідовним визначенням й алокацією є найбільш оптимальним підходом, бо у Swift неможливо одночасно створити й алокувати місце для масиву без додавання елементів, на відміну від деяких інших мов програмування.

На лістингу 3.5 показано продовження коду зсередини блоку, переданого в `queue.async(group: group)`. У внутрішньому циклі від start до end береться індекс з буфера й перевіряється за допомогою предикату isInFrustum чи потрібно його рендерити. Якщо потрібно, то цей індекс записується до масиву. Після завершення внутрішнього циклу відфільтровану частку індексів потрібно записати до загального масиву. Через те, що ми досі знаходимось в контексті паралельної черги, записувати напряму в масив chunks не варто, адже може виникнути так звиний data race. Цей термін в багатопоточному програмуванні описує ситауцію, коли два або більше потоків одночасно намагаються записувати дані до одного місця без належної синхронізації, що може призвести до непередбачуваних результатів.

Цю проблему було вирішено за допомогою ще одного інструменту для роботи з багатопоточністю у Swift, який називають бар’єрною чергою. Рядок queue.async(flags: .barrier) означає, що блок, який переданий в нього, виконається на паралельній черзі так, що жоден інший блок у тій самій черзі не виконуватиметься одночасно з ним. Виконання блоку з прапорцем `barrier` гарантує наступну поведінку: блоки, які вже виконуються на черзі, завершать своє виконання, аж тоді буде виконаний бар’єрний блок. При цьому інші блоки, які ставляться на чергу, не починають своє виконання допоки не завершиться виконання бар’єрного[26].

```
            for i in start..<end {
                let rawIndex = srcPointer[i]
                if self.isInFrustum(index: Int(rawIndex)) {
                    subArray.append(rawIndex)
                }
            }

            queue.async(group: group, flags: .barrier) {
                chunks[chunkIndex] = subArray
            }
        }
    }

    group.wait()
    let filteredIndices = chunks.flatMap { $0 }
```

Лістінг 3.5. Друга частина коду фільтрації на різних потоках.

Рядок `group.wait()` змушує головний потік дочекатися виконання фільтрації всіх частинок, після чого за допомогою функції `flatMap` двовимірний масив згладжується до одновимірного. Далі відбувається все те саме, що й в розділі 3.3. Варіація фільтрації індексів на CPU з розпаралеленням вийшла помітно кращою: близько 30 кадрів на секунду, коли в ідентичних умовах послідовний варіант видавав жахливі 8 кадрів. Проте накладні витрати цього підходу досі залишаються дуже значними, адже без нього в цих же умовах продуктивність усе ще становила близько 90 кадрів на секунду.

3.5 Розрахунки для відбраковки на GPU

Після експериментів з фільтрацією індексів на CPU, результати яких виявились незадовільними, було прийнято рішення здійснювати відбраковку вже на GPU. Для цього в функції вершин `impostor_vertex` була зроблена проста перевірка: якщо предикат is_vertex_visible для позиції атома повертає істину, то змін не відбувається, а якщо хибу – то цей атом відкидається й не рендериться.
Фукнція is_vertex_visible показана на лістінгу 3.4. Вона приймає як аргументи позицію атома і його радіус.

```
bool is_vertex_visible(float4 position, float radius) {
    for (int i = 0; i < 6; i++) {
        if (dot(position, frustumPlanes[i]) < -2.0 * radius) {
            return false;
        }
    }

    return true;
}
```

Лістінг 3.4. Функія видимості атома.

Функція перевіряє, чи вершина разом зі своїм радіусом знаходиться всередині зрізаної піраміди зору, яка визначається через шість площин, які можна описати матрицею з формули 3.1.

$$M = \begin{bmatrix} -1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & -1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & 1000 \end{bmatrix}$$

Формула 3.1. Матриця площин зрізаної піраміди зору.

У векторному представленні площини зрізаної піраміди зору у форматі (a, b, c, d) перші три значення a, b, і c представляють нормаль вектору площини, а d – це зсув площини від початку координат. Для площин, перпендикулярних до осей X та Y, значення координати d встановлене як 1, оскільки це значення визначає мінімальну відстань від початку координат до площини у напрямках X та Y. Ці площини формують бокові грані зрізаної піраміди зору, де кожна площина зміщена на одиницю від центра камери для створення зорового поля.

Однак, остання координата d у площини (0, 0, 1, 1000), яка є глибинною площиною зору, має вище значення. Це пов’язано з тим, що камера зосереджена на відстані вздовж осі Z. Ця площина встановлює глибину сцени, за якою об’єкти не будуть рендеритися.

Скалярний добуток позиції атома в гомогенних координатах та площини, представленої у вищеописаному вигляді, показує з якого боку площини відносно нормалі знаходиться цей атом. Якщо цей добуток менший за від'ємне подвоєне значення радіуса атома, то атом з урахуванням його радіусу повністю розташований за межами площини, тобто поза областю, яку вона визначає. Таким чином і визначається, чи варто рендерити цей атом.

У більшості графічних систем, як зокрема й у Metal, використовується гомогенне представлення координат для опису вершин. Гомогенні координати мають додаткову (для тривімірного простору – четверту) координату w, яка використовується для проєктування точок в перспективі. Якщо встановити $w = 0$ для позиції вершини, вона стає точкою на нескінченності в перспективній проєкції, і як наслідок, графічний процесор автоматично ігнорує цю вершину під час рендерингу.

Цей підхід, на відміну від попередніх, практично не вплинув на кількість кадрів на секунду при перегляді повної структури. Проте з алгоритмом визначення належності до зрізаної піраміди зору була проблема: чим більше користувач збільшує зображення – тим більше артефактів з’являлось по боках візуалізації. Відкидались зайві примітиви, що можна побачити на рисунку 3.2. Проте це означає, що ми на правильному шляху. Залишилось лишень зрозуміти, що не так з алгоритмом.

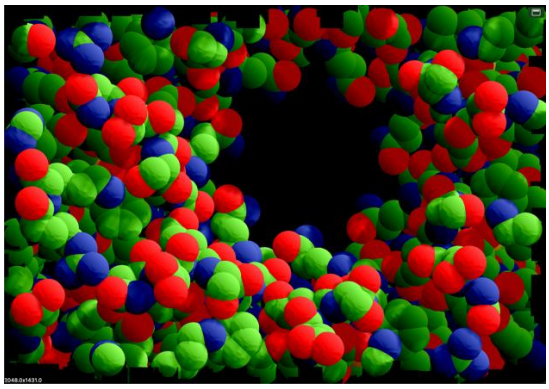

Рисунок 3.2 Візуалізація з імплементованим frustum culling.

3.6 Підвищення точності з використанням кута зору

Проблема, виявлена під час попередніх спроб оптимізації, полягала у тому, що зі збільшенням масштабу наявний алгоритм відкидав примітиви, які не мали виходити за межі зрізаної піраміди зору повністю, тим самим створюючи візуальні артефакти на краях візуалізації. Причина того, що в описаному в попередньому підрозділі алгоритмі відкидались зайві примітиви полягала в неврахуванні перспективного спотворення об'єктів залежно від кута зору. Це призводило до хибного визначення їх видимості.

Проблема вирішується шляхом введення множника, обрахованого на основі vertical field of view – вертикального кута зору камери. Цей кут поля зору, зазвичай вказаний у радіанах, визначає, наскільки широко камера «бачить» вертикально. Обраховується множник за формулою 3.2.

$$factor = \frac{1}{\tan(\frac{fov_v}{2})}$$

Формула 3.2. Множник перспективного спотворення.

Для коригування результатів відкидання радіус атома домножається на цей множник при виклику функції `is_vertex_visible`. Операції обрахування тангенсу й ділення для знаходження зворотнього до нього значення, тобто котангенсу, є недешевими, тому це оновлення додає певні накладні витрати. Проте вони очевидно є необхідними для збереження функціональності застосунку.

3.7 Результати оптимізації та подальші дослідження

Підхід, описаний у підрозділі 3.6, досяг бажаного результату. Візаульно досягнення показано на рисунку 3.3. В оригінальному проєкті завжди рендеряться всі атоми незалежно від приближення молекули, з оптимізацією кількість примітивів, які відмальовуються адаптується відповідно до того, що видно користувачу на екрані. Зображення, що показують зелені примітиви створені за допомгою Geometry viewer, переглядача геометрії. Це інструмент, вбудований в Xcode, який дозволяє переглядати геометрію сцени в реальному часі, досліджувати структуру буферів вершин, текстур та інших графічних ресурсів, які використовуються в додатку[27].

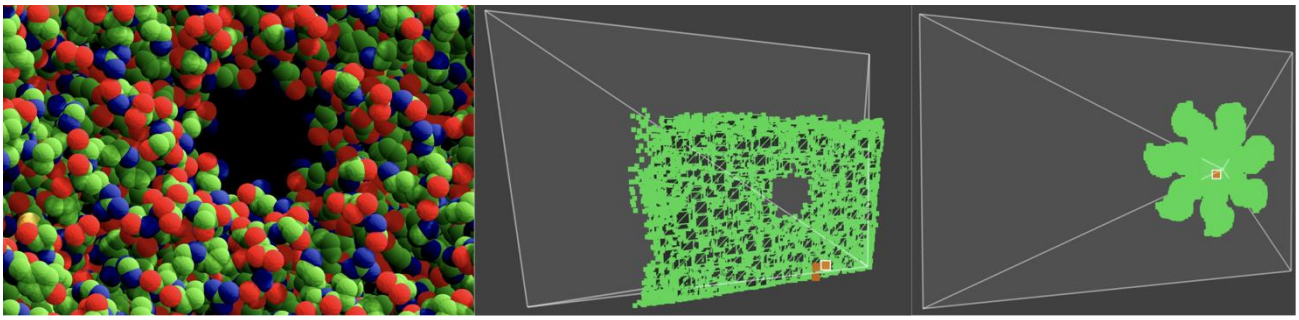

Рисунок 3.3. Результат, який видно користувачу (ліво); у сірій піраміді зору зеленими вершинами показані примітиви, які рендеряться з оптимізацією (центр) і без неї (право).

Отримані дані про кількість примітивів, які рендеряться свідчать про успішність експерименту. Завдяки оптимізації процесу відбракування невидимих атомів при збільшенні масштабу, знижено навантаження на графічний процесор. Це дозволяє використовувати звільнені ресурси для інших цілей. Наприклад, їх можна направити на створення більш деталізованих та якісних зображень у режимі збільшення.

Щодо подальших досліджень, варто спробувати не перевіряти кожен атом окремо, а радше розділити тривимірний простір на певні куби й перевіряти вже чи належить хоча б частина куба зрізаній піраміді зору й відкидати від рендерингу лише ті куби, які повністю знаходяться поза нею. Такий підхід є більш розширюваним. Він дозволить регулювати «агресивність» оптимізації за допомогою підлаштування розміру сторони куба. Таким чином більші куби означатимуть менше перевірок, проте більші об’єми відмальованих атомів, що не видимі користувачу й навпаки.

ВИСНОВКИ

У цій роботі було проведено ґрунтовне дослідження різноманітних методик оптимізації рендерингу на мобільних пристроях. Найбільше уваги було приділено алгоритму frustum culling для зменшення навантаження на графічний процесор при збільшенні масштабу візуалізації.

Основним досягненням роботи стала розробка та впровадження варіації підходу frustum culling для відкидання невидимих атомів, що за збільшення масштабу значно знизило кількість примітивів, які рендеряться. Важливо зазначати, що ця оптимізація була впроваджена в проєкт з відкритим кодом. Вона потенційно дозволить виділити більше ресурсів для, наприклад, поліпшення якості зображень при детальному вивченні молекулярних структур.

За результатами роботи можна виділити кілька напрямків для подальших досліджень. Перше, що спадає на думку, це вивчення можливостей інтеграції розробленої оптимізації з іншими методами, такими як occlusion culling та backface culling, для досягнення ще вищої продуктивності рендерингу.

Також є потенціал у дослідженні впливу різних алгоритмічних підходів до визначення видимості об'єктів на загальну продуктивність і якість візуалізації. Наприклад, варто спробувати альтернативний алгоритм, описаний наприкінці розділу 3.7, з розділенням тривімірного простору на куби.

Іще один варіант для подальших покращень і досліджень – адаптація досліджених технік для використання у віртуальній та доповненій реальності. Вимоги до швидкості візуалізації і реалістичності зображень у таких умовах є надзвичайно високими, тому їх оптимізація є ще більш актуальною.

Повертаючись до результатів поточного експерименту, можна зробити висновки, що він виявився успішним і при цьому показав перспективи як для подальшого вдосконалення запропонованої технології, так і для впровадження інших.

ДЖЕРЕЛА

1. Goldratt E. M. The Goal / Eliyahu M. Goldratt., 1984. – 384 с.
2. American Chemical Society. Molecule of the Week Archive: Succinic acid [Електронний ресурс] / American Chemical Society. – 2009. – Режим доступу до ресурсу: https://www.acs.org/molecule-of-the-week/archive/s/succinic-acid.html.
3. CCDC. A short guide to Crystallographic Information Files [Електронний ресурс] / CCDC – Режим доступу до ресурсу: https://www.ccdc.cam.ac.uk/media/MoreInformationAboutCIFsyntax.pdf.
4. Introduction to Protein Data Bank Format [Електронний ресурс] – Режим доступу до ресурсу: https://www.biostat.jhsph.edu/~iruczins/teaching/260.655/links/pdbformat.pdf.
5. UCSF Computer Graphics Laboratory. XYZ Format [Електронний ресурс] / UCSF Computer Graphics Laboratory. – 2014. – Режим доступу до ресурсу: https://www.cgl.ucsf.edu/chimera/docs/UsersGuide/xyz.html.
6. Space-filling model [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Space-filling_model.
7. Batsanov S. S. Van der Waals Radii of Elements [Електронний ресурс] / S. S. Batsanov. – 2001. – Режим доступу до ресурсу: https://physlab.lums.edu.pk/images/f/f6/Franck_ref2.pdf.
8. Properties of water [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Properties_of_water.
9. Rose A. Rendering in Mol [Електронний ресурс] / Alexander Rose. – 2022. – Режим доступу до ресурсу: https://www.khronos.org/assets/uploads/developers/presentations/rendering-in-molstar.pdf.

10. Lorensen W. E. MARCHING CUBES: A HIGH RESOLUTION 3D SURFACE CONSTRUCTION ALGORITHM [Електронний ресурс] / W. E. Lorensen, H. E. Cline. – 1987. – Режим доступу до ресурсу: https://dl.acm.org/doi/pdf/10.1145/37402.37422.
11. Xu D. Generating Triangulated Macromolecular Surfaces by Euclidean Distance Transform [Електронний ресурс] / D. Xu, Y. Zhang. – 2009. – Режим доступу до ресурсу: https://journals.plos.org/plosone/article/file?id=10.1371/journal.pone.0008140&type=printable.
12. Tiigimägi S. What is LOD: Level of Detail [Електронний ресурс] / Siim Tiigimägi – Режим доступу до ресурсу: https://3dstudio.co/3d-lod-level-of-detail/.
13. Lee E. Vertex Chunk-Based Object Culling Method for Real-Time Rendering in Metaverse [Електронний ресурс] / E. Lee, B. Shin. – 2023. – Режим доступу до ресурсу: https://www.mdpi.com/2079-9292/12/12/2601.
14. Sherif T. WebGL 2 Example: Occlusion Culling [Електронний ресурс] / Tarek Sherif. – 2017. – Режим доступу до ресурсу: https://github.com/tsherif/webgl2examples/blob/master/occlusion.html.
15. Akenine-Möller T. Real-Time Rendering, Fourth Edition / T. Akenine-Möller, E. Haines, N. Hoffman., 2018. – 1178 с.
16. Polack T. Focus On 3D Terrain Programming / Trent Polack., 2002. – 220 с. – (1).
17. Back face culling for linestrips [Електронний ресурс] – Режим доступу до ресурсу: https://stackoverflow.com/questions/28794883/back-face-culling-for-linestrips.
18. Chrysanthou Y. Three types of visibility culling techniques [Електронний ресурс] / Yiorgos Chrysanthou – Режим доступу до ресурсу:

https://www.researchgate.net/figure/Three-types-of-visibility-culling-techniques-1-View-Frustum-Culling-2-back-face_fig1_2440562.

19. Androp0v. BioViewer [Електронний ресурс] / Androp0v – Режим доступу до ресурсу: https://github.com/Androp0v/BioViewer/tree/main?tab=readme-ov-file.
20. SwiftUI [Електронний ресурс] – Режим доступу до ресурсу: https://developer.apple.com/xcode/swiftui/.
21. Rob Cook [Електронний ресурс]. – 2008. – Режим доступу до ресурсу: https://graphics.pixar.com/people/rob/.
22. Metal Shading Language Specification [Електронний ресурс] – Режим доступу до ресурсу: https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf.
23. Using a Render Pipeline to Render Primitives [Електронний ресурс] – Режим доступу до ресурсу: https://developer.apple.com/documentation/metal/using_a_render_pipeline_to_render_primitives.
24. MTLBuffer [Електронний ресурс] – Режим доступу до ресурсу: https://developer.apple.com/documentation/metal/mtlrenderpipelinestate.
25. MTLDevice [Електронний ресурс] – Режим доступу до ресурсу: https://developer.apple.com/documentation/metal/mtldevice.
26. C R. Dispatch Barrier in Swift [Електронний ресурс] / Ranga C. – 2022. – Режим доступу до ресурсу: https://medium.com/@ranga.c222/dispatch-barrier-in-swift-84779f49a291.
27. BabylonJS. DebugRenderedFrameMetal [Електронний ресурс] / BabylonJS – Режим доступу до ресурсу: https://github.com/BabylonJS/BabylonNative/blob/master/Documentation/DebugRenderedFrameMetal.md.