НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування і спеціалізованих комп‘ютерних систем

«На правах рукопису»

УДК 004.925.3

«До захисту допущено»

Завідувач кафедри
_____ Віталій РОМАНКЕВИЧ
_____
(підпис) (ініціали, прізвище)
“___”_____2023 р.

Магістерська дисертація

на здобуття ступеня магістра
за освітньо-науковою програмою «Системне програмування та спеціалізовані комп’ютерні системи»

зі спеціальності **123 «Комп'ютерна інженерія»**

на тему: Метод візуалізації об’єктів в комп’ютерних системах засобами API Vulkan

Виконав:
студент II курсу, групи КВ-11мн
(шифр групи)

Новіков Олександр Олегович
(прізвище, ім’я, по батькові) (підпис)

Керівник ст.викладач каф. СПСКС, к.т.н. Коляда К.В.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент.
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2023 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет прикладної математики
Кафедра системного програмування і спеціалізованих комп'ютерних систем

Рівень вищої освіти – другий (магістерський)
Спеціальність 123 «Комп'ютерна інженерія»
Освітньо-наукова програма «**Системне програмування та спеціалізовані комп'ютерні системи**»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Віталій РОМАНКЕВИЧ
(підпис) (ініціали, прізвище)
«___» _____2023 р.

ЗАВДАННЯ
на магістерську дисертацію студента

Новіков Олександр Олегович

1. Тема дисертації Метод візуалізації об'єктів в комп'ютерних системах засобами API Vulkan ,
керівник проєкту ст.викладач каф. СПСКС, к.т.н. Коляда К.В. ,
затверджені наказом по університету від «30» березня 2023р. №1359-с
2. Термін подання студентом проєкту
3. Об'єкт дослідження: комп'ютерні системи візуалізації об'єктів.
4. Предмет дослідження: методи візуалізації прозорих об'єктів за допомогою засобів API Vulkan.
5. Перелік завдань, які потрібно розробити : проаналізувати існуючі методи візуалізації прозорих об'єктів; дослідити можливості API Vulkan; розробити програмне забезпечення для порівняння методів; обгрунтувати вибір методу для візуалізації прозорих об'єктів та оптимізувати його роботу за допомогою засобів API Vulkan.
6. Орієнтований перелік графічного (ілюстрованого) матеріалу: презентація.
7. Дата видачі завдання: 20.10.2021р

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів	Примітка
1.	Вивчення літератури за тематикою дисертації	24.10.2021	
2.	Визначення структури магістерської дисертації	28.12.2021	
3.	Робота над першим розділом магістерської дисертації; проведення наукового дослідження	19.01.2022	
4.	Підготовка матеріалів першого розділу дипломного проекту	12.05.2022	

5.	Проведення наукового дослідження; робота над другим розділом магістерської дисертації; розроблення програмного забезпечення	03.09.2022	
6.	Проведення наукового дослідження; робота над статтею за результатами наукового дослідження	14.10.2022	
7.	Проведення наукового дослідження; робота над третім розділом магістерської дисертації	30.11.2022	
8.	Підготовка матеріалів четвертого розділу магістерської дисертації	18.01.2023	
9.	Завершення роботи над основною частиною магістерської дисертації; підготовка ілюстративного матеріалу; підготовка матеріалів доповіді на конференції ПМК-2018	29.03.2023	
10.	Оформлення текстової і графічної частини магістерської дисертації	30.04.2023	
11.	Попередній розгляд МД на кафедрі	04.05.2023	

Студент _____ Олександр НОВІКОВ
(підпис) (ініціали, прізвище)

Науковий керівник дисертації _____ Костянтин КОЛЯДА
(підпис) (ініціали, прізвище)

РЕФЕРАТ

Актуальність теми. Комп’ютерна візуалізація – це галузь в якій відбуваються постійні зміни. Зумовлено це неспинним розвитком та модифікаціями архітектури графічних процесорів. Зміни в апаратних частинах комп’ютерних систем призводять до потреби в перегляді методів розробки програмного забезпечення, які використовуються на цих самих системах. Нові можливості графічних процесорів та інструментів взаємодії з ними можуть бути використані для покращення існуючих методів візуалізації об’єктів.

Об’єктом дослідження є комп’ютерні системи візуалізації об’єктів.

Предметом дослідження є методи візуалізації прозорих об’єктів засобами API Vulkan.

Мета роботи: Оптимізація методу візуалізації прозорих об’єктів за допомогою засобів API Vulkan. Створення програмного забезпечення для порівняння існуючих методів.

Наукова новизна полягає в актуалізації методів візуалізації прозорих об’єктів за допомогою сучасних засобів, які надає API Vulkan. Надання шляхів оптимізації методу візуалізації прозорих об’єктів.

Практична цінність полягає у надані аналізу існуючих методів візуалізації прозорих об’єктів та в оптимізації методу візуалізації прозорих об’єктів за допомогою спеціалізованих засобів API Vulkan, що покращує роботу комп’ютерних систем, які призначені для візуалізації тривимірних сцен.

Апробація роботи. Основні положення і результати роботи були представлені та обговорювались на 89й Міжнародній науковій конференції молодих учених, аспірантів і стедунтів від НУХТ та на конференції від Молодіжної Наукової ліги. Було опубліковано статтю у видані Міжнародного наукового журналу «Грааль Науки» №26 – Вінниця-Відень, 2023. (OUCI, Copernicus)

Структура та обсяг роботи. Магістерська дисертація складається з вступу, трьох розділів та висновків.

У вступі описується загальна задача методів візуалізації прозорих об’єктів, актуальність та практичне застосування цих методів.

У першому розділі наведено аналіз існуючих методів вирішення проблеми візуалізації прозорих об’єктів за допомогою розробленого програмного забезпечення.

У другому розділі наведено опис API Vulkan та аналіз засобів цього інстументу, які можуть бути використані для оптимізації візуалізації об’єктів.

У третьому розділі описуються способи оптимізації методу візуалізації за допомогою засобів API Vulkan та наведені отримані результати.

У висновках надані результати проведеної роботи.

Робота представлена на 72 аркушах, містить посилання на список використаних літературних джерел.

Ключові слова: рендеринг, комп’ютерна візуалізація, прозорі об’єкти, тривимірна симуляція, Vulkan, GPU.

ABSTRACT

Relevance of the topic. Computer visualization is an industry that is constantly changing. This is due to the constant development and modification of the architecture of graphics processors. Changes in the hardware of computer systems require revision of software development methods used on these same systems. New capabilities of GPUs and tools for interacting with them can be used to improve existing methods of object visualization

The object of research is the method of visualizing transparent objects.

The subject of the research is methods of visualizing transparent objects using the Vulkan API.

Objective: Analysis of existing methods for visualizing transparent objects. Creating software for comparing methods. Optimization of the method for visualizing transparent objects using Vulkan API tools.

The scientific novelty: upgrade of the methods of visualizing transparent objects using modern tools provided by the Vulkan API. And providing ways to optimize the chosen method of visualizing transparent objects.

The practical value of the results obtained in the work is that the proposed method of hardware and software ray tracing makes it possible to visualize a realistic image in three-dimensional simulation tasks. The developed method makes it possible to render scenes with a sufficient level of speed for real-time rendering based on compared to other rendering methods based on the ray tracing algorithm. The performance improvement is about 15-20 percent.

Approbation of work. The main provisions and results of the work were presented and discussed at the 89th International Scientific Conference of Young Scientists, Postgraduates and Students from NUPh and at the conference of the Youth Scientific League. An article was published in the Grail of Science.

Structure and scope of work. The master's thesis consists of an introduction, three chapters and conclusions.

The introduction describes the general task of methods for visualizing transparent objects, the relevance and practical application of these methods.

The *first section* existing methods for solving the problem of visualizing transparent objects was analyzed using the developed software.

The *second section* describes the Vulkan API and analyzes the tools of this tool that can be used to optimize object visualization.

The *third section* describes the ways to optimize the visualization method using the Vulkan API and presents the results obtained

The *results* of the work are presented in the conclusions.

The work is presented on 72 pages, contains links to the list of references used.

Keywords: rendering, graphics, GPU, three-dimensional simulation, transparent objects, Vulkan.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

API – Прикладний програмний інтерфейс;

RGB – опис кольору на основі червоного, заленого, синього кольорів;

CPU – центральний процесор, або ж ЦЗ;

GPU – графічний процесор;

MSAA – метод зглажування зображенння;

ВСТУП

Майже кожного року відбуваються зміни в архітектурі графічних процесорів та в інструментах взаємодії інженерів з апаратною частиною комп'ютерних систем. Це впливає на підходи реалізації методів візуалізації об'єктів, які використовуються в програмному забезпечені для відтворення реалістичних зображень. Такі зміни впливають на актуальність використання певних методів візуалізації та надають можливості їх покращення.

Однією з найбільших змін в області комп'ютерної візуалізації за останні часи стало поява нового API для роботи з графічним процесором під назвою Vulkan. У порівнянні з його попередниками цей інструмент надає більше засобів низькорівневої взаємодії з графічним процесором, які можуть бути використані інженерами для розробки програмного забезпечення у напрямку комп'ютерної візуалізації. Наприклад, Vulkan надає можливість регулювати виділення пам'яті, обирати тип пам'яті і тд.

За мету даної роботи взято реалізація, дослідження та аналіз існуючих методів візуалізації прозорих об'єктів. Оптимізації методу візуалізації прозорих об'єктів на комп'ютерних системах з використання засобів, що надає API Vulkan.

1. АНАЛІЗ МЕТОДІВ ВІЗУАЛІЗАЦІЇ ПРОЗОРИХ ОБ’ЄКТІВ

1.1. Опис задачі візуалізації прозорих об’єктів

Проблема задачі візуалізації прозорих об’єктів полягає у тому, щоб відтворити реалістичне зображення таких прозорих об’єктів як скло, рідина, туман і тд (рис. 1.1 та рис. 1.2).

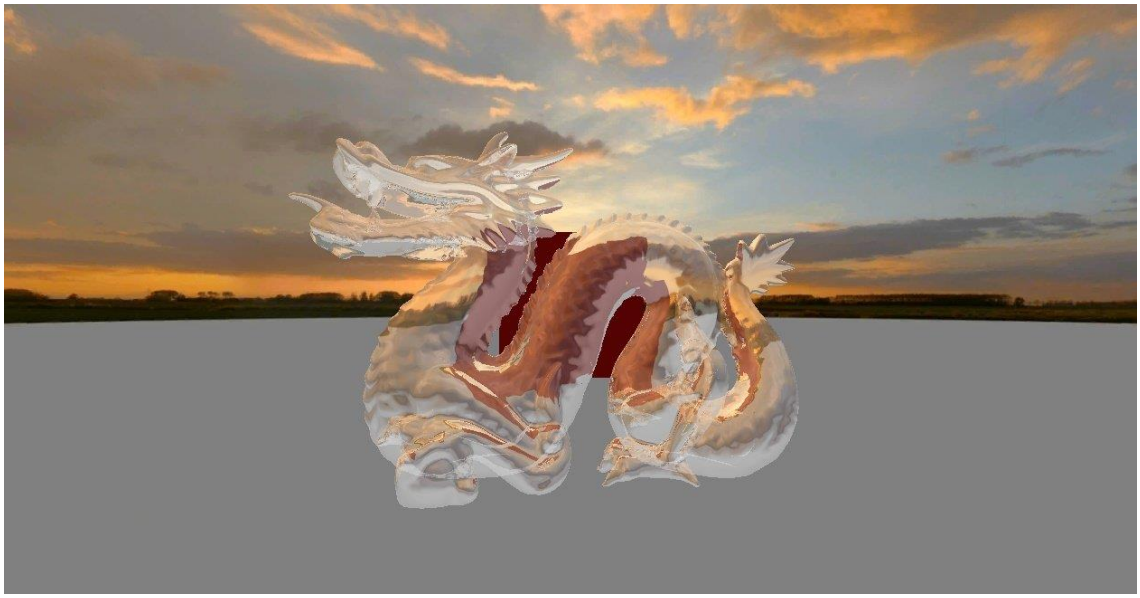

Рисунок 1.1 – приклад відтворення прозорого об’єкту

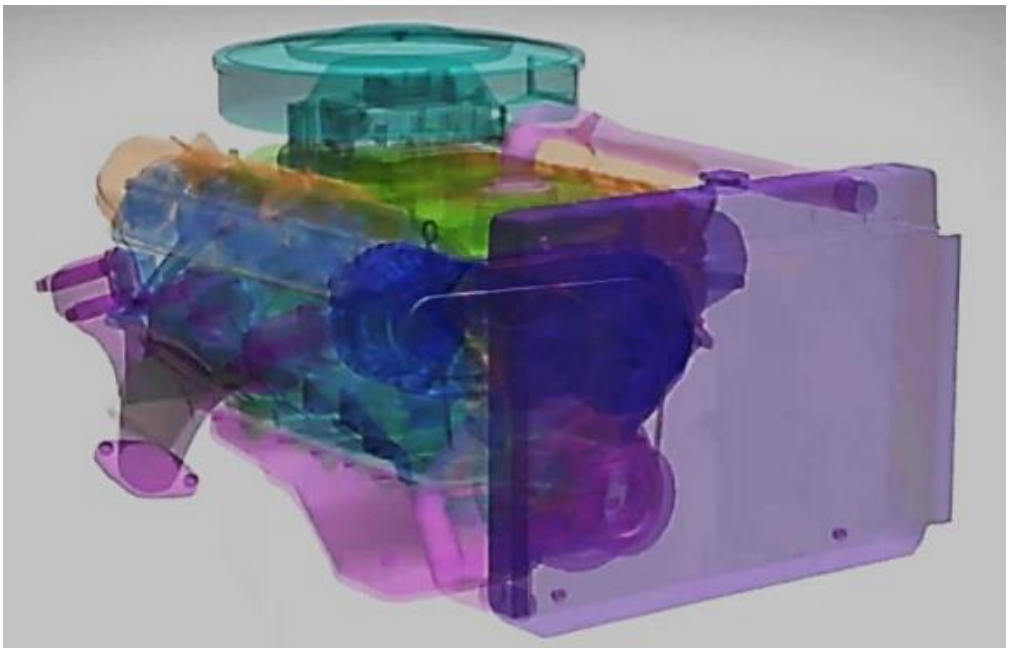

Рисунок 1.2 – використання методів візуалізації прозорих об’єктів для моделювання

Незважаючи на те, що проблема розглядається уже протягом довгого часу, досі не існує апаратного вирішення даної задачі. Не існує можливості задати параметри режиму роботи графічного процесору для візуалізації прозорих об’єктів, так як, наприклад, вирішується питання обробки глибини сцени. Тестування глибини працює шляхом порівняння значень глибини (Z-значень) пікселів об'єктів, що візуалізуються, з буфером глибини, який є структурою даних, що зберігає значення глибини попередньо відображених пікселів. Буфер глибини ініціалізується максимальним значенням глибини, зазвичай це значення дорівнює 1.0, і під час обчислення кожного пікселя його значення глибини порівнюється зі значенням, що зберігається в буфері глибини на тій ж самій позиції. Якщо значення глибини показує, що частина об’єкту за яку відповідає піксель ближче до глядача, ніж значення у буфері глибини, піксель промальовується і вже його значення глибини записується до буфера. В іншому випадку піксель відкидається, тобто цей піксель не приймає участь у відтворенні зображення. Всі ці операції реалізовано на стороні графічного процесору, інженер програмного забзпечення лише задає параметри тестування глибини, як початкова глибина, що записується в буфер, або умову за якій відкидається фрагмент (глибина більше або менше за вже записаний у буфер) [9].

Обчислення прозорість об’єктів досі не реалізовано на графічних процесорах, оскільки фізично не має можливості записувати інформацію про рівень прозорості об’єкту, оскільки кожна координата єкрану (кінцевого зображення) може мати нескінченну кількість інформації, оскільки об’єктів на комплексній сцені може бути багато, вони можуть знаходитися один за одним. Тому існує багато різноманітних програмних методів, які пропонують рішення поставленої задачі.

Ці методи поділяються на два класи: упорядкована прозорості та неупорядкована прозорість.

Методи упорядкованої прозорості є тривіальними. Ідея цього класу алгоритмів заключена у його назві. Для того щоб відтворити сцену із прозорими

об’єктами, ці об’єкти спочатку сортуруються у заданому порядку відносно камери, яка направлена на сцену [7].

Основним недоліком сортування об’єктів на сцені перед тим як їх відмальовувати є використання центрального процесору. Навантаження центрального процесору зменшує продуктивність роботи комп’ютерної системи. Оскільки процесор виконує різноманітні задачі під час роботи комп’ютерної системи, це призводить до очікування графічним процесором системи на результати сортування. Через це графічний процесор може простоювати в очікувані, що не є ефективним рішенням, особливо в ситуації візуалізації в реальному часі.

Другим класом методів візуалізації прозорих об’єктів є неупорядкована прозорість. Ці методи не потребують попереднього сортування об’єктів на сцені для відтворення бажаного реалістичного ефекту прозорості. Кожен з цих методів має свої переваги і недоліки [7].

Деякі з методів неупорядкованиї прозорості використовують сортування не об’єктів, а фрагментів які обробляються графічним процесором. Де, фрагмент - це набір значень, створених растеризатором. Кожен фрагмент являє собою сегмент растрованого примітиву. Розмір фрагмента пов'язаний з площею пікселя, але растеризація може створювати декілька фрагментів з одного трикутника на піксель, залежно від різних параметрів мультидискретизації. Створюється принаймні один фрагмент для кожної області пікселів, яку покриває примітив, що растеризується.

Методи які використовують описану вище техніку називають точні методи. Завдяки ним отримують більш реалістичні результати. Інші ж методи називають наближеними методами. Вони не виконують сортування взагалі, навіть сортування фрагментів, а обчислюють наближені значення. Що в свою чергу зменшує якість отриманого результату, особливо у комплексних сценах, але при цьому зменшується використання обчислювальних можливостей апаратного забезпечення [7].

Неточні методи досягають збільшення швидкодії за рахунок часткової або повної відмови від сортування фрагментів. Дві основні техніки які використовуються в методах такого роду: стохастична прозорість та адаптивна прозорість [8].

Стохастична прозорість - це техніка, що використовується в комп'ютерній графіці для досягнення реалістичного та ефективного відображення прозорих матеріалів, таких як, наприклад, скло або вода. Ця техніка працює шляхом випадкової вибірки прозорості поверхні для кожного пікселя, замість того, щоб виконувати змішування кольорів поверхні з фоном.

Стохастична прозорості базується на ідеї того, що в реальному світі прозорість поверхні визначається випадковим розташуванням її складових частинок або молекул. Імітуючи цей випадковий процес під час візуалізації сцени, стохастична прозорість може створювати більш реалістичні та природні зображення.

Традиційні методи візуалізації прозорих об’єктів, такі як альфа-змішування, можуть бути затратними за ресурсами комп’ютерних систем і при цьому можуть призводити до видимих артефактів, наприклад, коли кілька прозорих поверхонь накладаються одна на одну. Стохастична прозорість дозволяє ефективно відтворювати складні сцени з декількома прозорими поверхнями, використовуючи імовірнісний підхід для визначення того, які поверхні мають бути видимими в кожному пікселі [5].

Адаптивна прозорість - це техніка, що використовується в комп'ютерній графіці для відтворення прозорості об’єктів на сцені у спосіб, що балансує між обчислювальною ефективністю та візуальною якістю. Цей підхід працює шляхом вибіркового обчислення прозорості кожного пікселя на основі його впливу на якість кінцевого зображення.

Адаптивна прозорість обчислює прозорість лише найважливіших пікселів, при цьому пропускаючи обчислення значень пікселів, які не маю великого впливу на результат кінцевого зображення сцени. Це досягається за допомогою

набору евристик для оцінки внеску кожного пікселя в кінцеве зображення на основі таких факторів, як його глибина, відстань від камери та рівень контрасту з навколишніми пікселями [7].

Вибірково обчислюючи прозорість пікселів, адаптивна прозорість може зменшити загальні обчислювальні витрати на обчислення прозорих об'єктів, водночас створюючи візуально реалістичн результати. При цьому адаптивну прозорість також можна комбінувати з іншими методами рендерингу, такими як стохастична прозорість або глибинний пілінг. Глибинний пілінг використовується для подолання обмежень традиційного підходу з буфером глибини, коли прозорі об’єкти не можуть бути відтворені правильно, оскільки вони не мають чітко визначеного порядку відносно непрозорих поверхонь. При глибинному пілінгу прозора геометрія обчислюється багато разів у шарах, де кожен шар представляє певний діапазон глибини. Значення глибини прозорої геометрії зберігаються у декількох текстурах, по одній на кожен шар. Після рендерингу всіх шарів значення кольору та глибини кожного пікселя змішуються у порядку проходження шарів, починаючи з задньої частини сцени і рухаючись вперед. Цей процес повторюється доти, доки всі прозорі поверхні не будуть змішані. Кінцевим результатом є набір текстур, які представляють змішані прозорі поверхні. Ці текстури можна об'єднати з непрозорими поверхнями, щоб отримати остаточне зображення. Глибинний пілінг може задіювати багато обчислювальних ресурсів комп’ютерної системи, оскільки вимагає декількох проходів по сцені для кожного шару. Однак, ця технака забезпечує надійне і точне відтворення прозорості об’єктів на сцені [5].

1.2 Опис програмного забезпечення розробленого для порівняння методів візуалізації прозорих об’єктів

Для аналізу та порівняння методів візуалізації прозорих об’єктів в комп’ютерних системах, у ході роботи над дисертацією було розроблено програмне забезпечення в якому реалізовано основні методи візуалізації прозорих об’єктів.

Програмне забезпечення випадковим чином створює сцену на якій відображаться об’єкти, сфери з різним кольором, розміром та коефіцієнтом прозорості.

Для розробки програмного забезпечення було обрану мову програмування високого рівня C++. Ця мова програмування широко використовується в комп'ютерній графіці для розробки різноманітних додатків. Це компільована мова, тобто вихідний код компілюється в машинний, що збільшує ефективність роботи програмного забезпечення розробленого з використання мови C++.

C++ дозволяє інженерам розробляти програмне забезпечення для комп’ютерних систем, яке виконується безпосередньо на апаратному забезпеченні цих систем, що робить її ідеальною для розробки високопродуктивних додатків для відображення трьохвимірних сцен [10].

C++ надає інженерам програмного забезпечення низькорівневий доступ до апаратного забезпечення персонального комп'ютера, такого як відеокарта. Це дозволяє використовувати різноманітні можливості графічного процесору [10].

C++ дозволяє легко організовувати та керувати складними частинами коду програмного забезпечення, оскільки є об’єктно орієнтовною мовою програмування. Це особливо корисно для розробки великомасштабних графічних додатків, таких як відеоігри [10].

C++ має велику базу стандартних бібліотек, що надає широкий спектр функцій і структур даних, що можуть бути використані в розробці програмного забезпечення для відтворення трьохвимірних сцен. Використання цих бібліотек допомагає інженерам заощадити час і зробити розробку програмного забезпечення більш ефективною [10].

C++ є крос-платформною мовою, отже програмне забезпечення, розроблене на C++, може бути скомпільованим для запуску на комп’ютерних системах з різними операційними системами та апаратними платформами, включаючи Windows, Linux та macOS [10].

Загалом, C++ є потужною та універсальною мовою програмування, яка добре підходить для розробки високопродуктивного програмного забезпечення для вирішення проблем комп'ютерної графіки [10].

Для створення інтерфейсу користувача було використано ImGUI. ImGUI розшифровується як Immediate Mode Graphical User Interface (рис. 1.3). Ця бібліотека призначена для створення багатофункціонального графічного інтерфейсу при розробці програмного забезпечення, що використовується в реальному часі, таких як, наприклад, відеоігри, симулятори та програми моделювання.

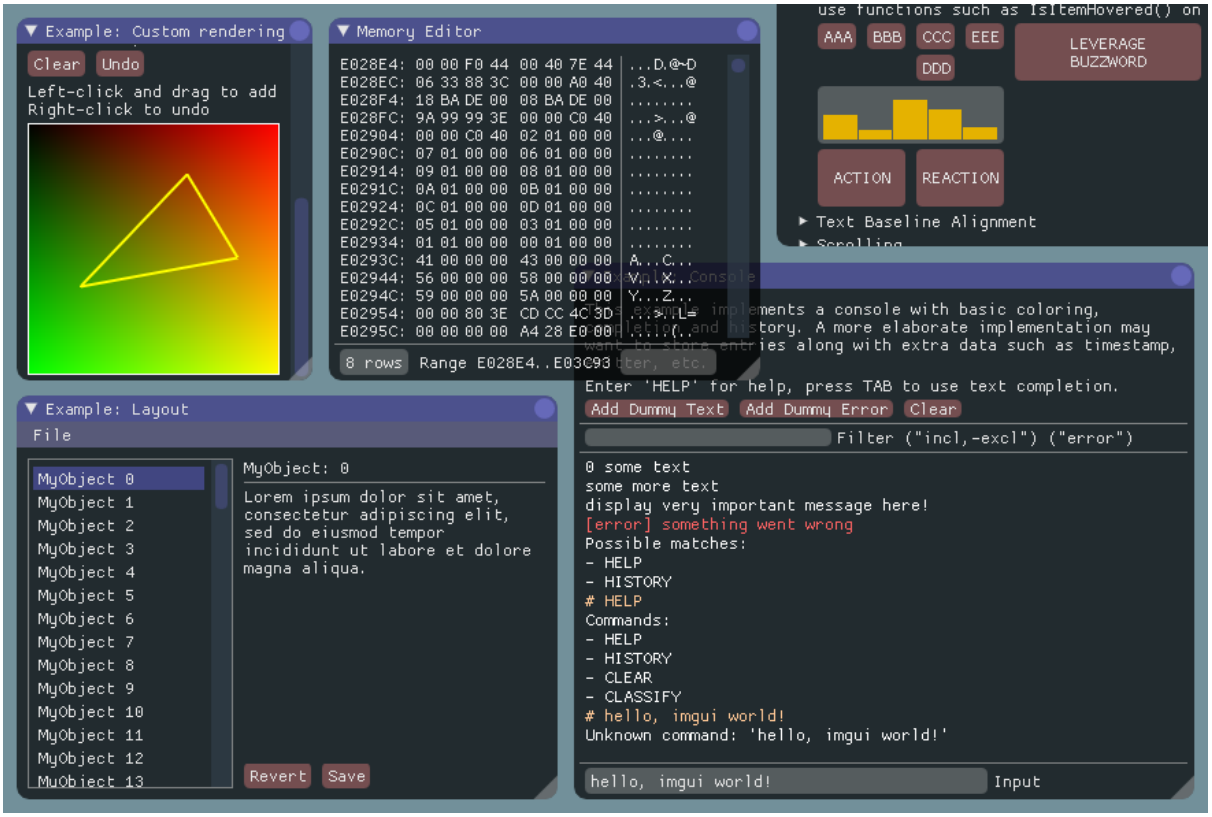

Рисунок 1.3 – приклад інтерфейсу користувач створеного за допомогою Dear ImGUI

Основна концепція ImGUI полягає в тому, що інтерфейс створюється в режимі негайного доступу. Це означає, що цей інтерфейс не створюється шляхом створення віджетів, а потім малювання їх на екрані. Замість цього інтерфейс створюється для кожного кадру, на основі поточного стану програми. В ImGUI інтерфейс описується за допомогою набору функцій безпосереднього режиму, які можна викликати для малювання різних типів віджетів, таких як кнопки, прапорці, поля введення тексту і так далі. Ці функції піклуються про низькорівневі деталі малювання віджетів на екрані, такі як створення необхідної геометрії та обробка даних, введених користувачем.

Однією з головних переваг ImGUI є його простота і легкість у використанні. Він має невеликий обсяг коду і може бути легко інтегрований в існуючі кодові бази. Оскільки інтерфейс створюється "на льоту", він також є дуже гнучким і може адаптуватися до потреб програми під час виконання.

ImGUI реалізовано на C++, але він також має прив'язки до інших мов, таких як C, Python та Lua. Це бібліотека з відкритим вихідним кодом і широко використовується в індустрії розробки ігор, а також в інших сферах, таких як наукова візуалізація та аналіз даних.

На рис. 1.4 зображено розроблений інтерфейс програмного забезпечення.

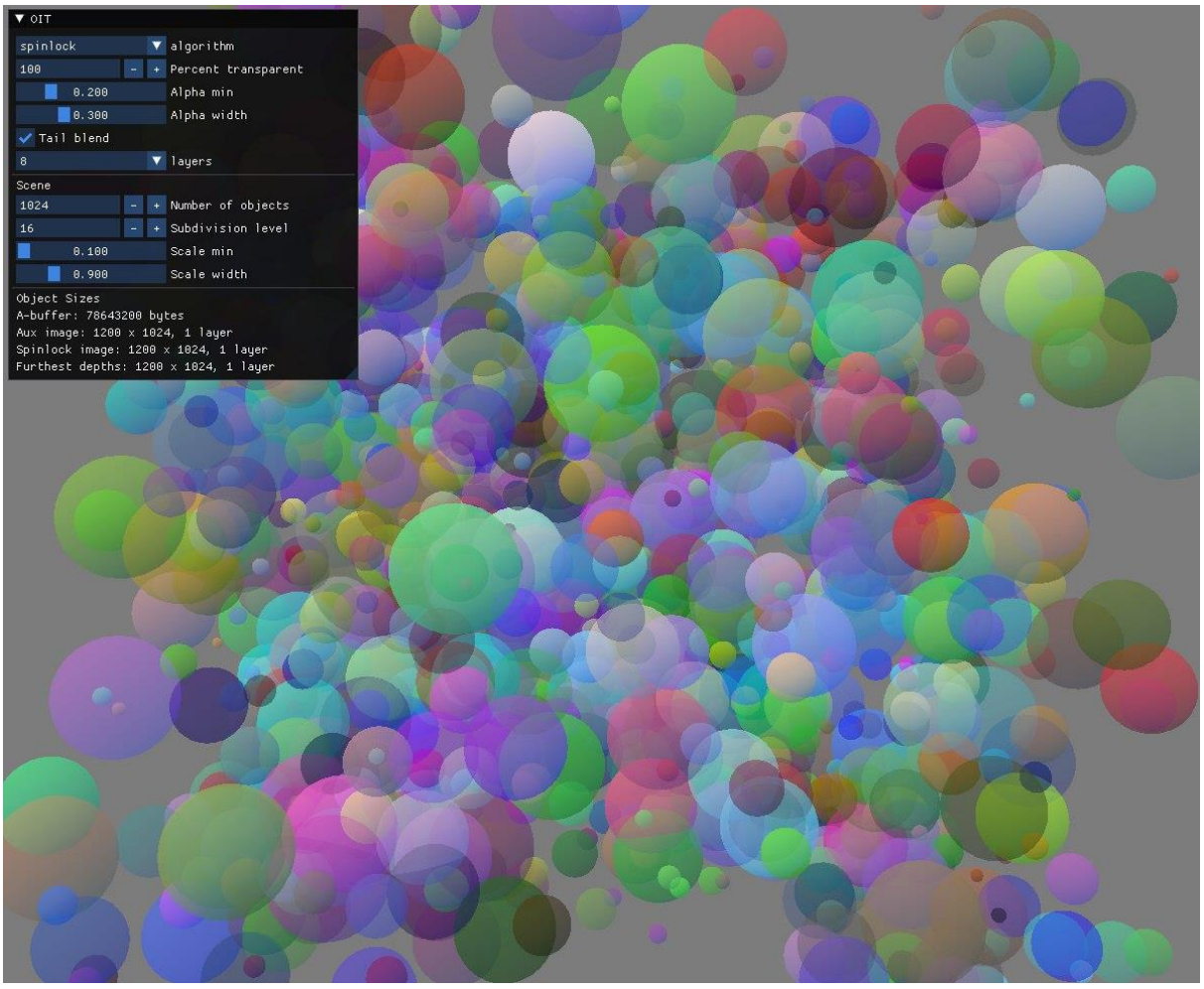

Рисунок 1.4 – Вікно програмного забезпечення розробленого для порівняння методів візуалізації прозорих об’єктів

На рис 1.5 зображено опції які може обирати користувач програмного забезпечення. Algorithms відповідає за вибір методу, який буде використовуватися для відтворення прозорих об’єктів на сцені. Percent transparent встановлює відношення між кількістю прозорих об’єктів та непрозорих об’єктів. Рівень прозорості об’єктів встановлюється на основі

значень Alpha min та Alpha width. Відповідно, прозорість обирається випадково у діапазоні [alpha min, alpha min + alpha width]. Прапорець відповідає за включення режиму tail blend, про який описується далі. Layers (N) встановлює кількість фрагментів з якими працює обраний метод візуалізації. Для контролю кількості об’єктів, які створюються на сцені, використовується Number of objects. Subdivision регулює кількість полігонів з яких складаються сфери, що відображаються на сцені. Scale min та Scale width регулюють розміри об’єктів. У нижній області інтерфейсу користувача надається інформація про ресурси, які використовуються обраним алгоритмом візуалізації прозорих об’єктів.

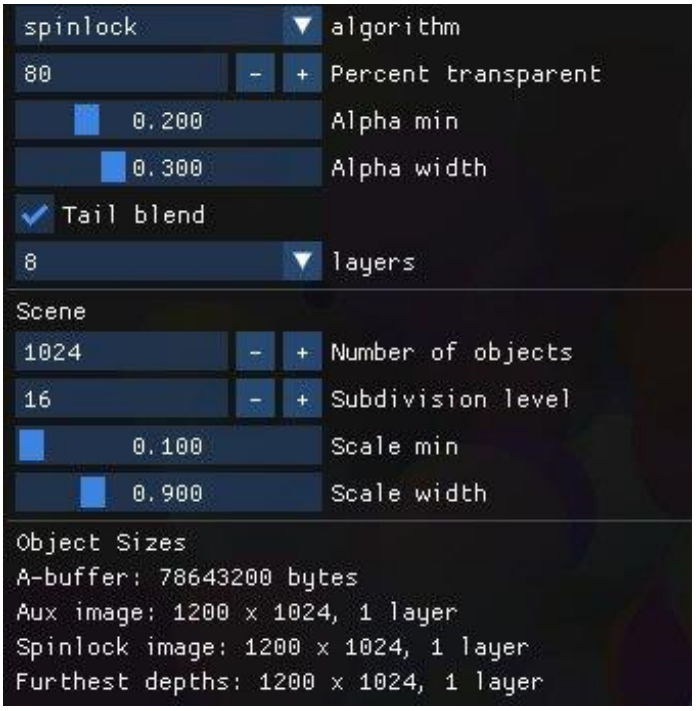

Рисунок 1.5 – зображення вікна налаштувань програмного забезпечення

1.3 Simple-метод

Для зберігання інформації в цьому методі створюється буфер, який буде відповідати розмірам зображення сцени. Глибина цього буферу буде визначати точність методу. Глибина позначається N. Це кількість фрагментів (елемент відображення об’єкту на сцені), які відповідають пікселю результуючого зображення. Шейдер зберігає значення кольорів та глибини перших N фрагментів та змішенне значення кольорів фрагментів, що залишилися (цей

процес називається tail blend). Tail Blend має на меті покращити продуктивність рендерингу прозорих об'єктів. Замість того, щоб видаляти всі фрагменти геометрії, tail blend видаляє лише останній фрагмент кожного об'єкта. Цей фрагмент потім змішується з попередньо обрахованими пікселями в буфері кадру [5].

Наприклад, маємо значення N = 2, при цьому одному пікселю відповідає 4 фрагменти (пари кольору та глибини). Тобто є чотири пари значень записані в такому порядку: [C1, 0.35], [C2, 0.2], [C3, 0.6], [C4, 0.1]. Де C1 - це значення кольору у системі RGB. Програма зберігає значення перших двох пар, оскільки в даному випадку N = 2. Отже [C1, 0.35] та [C2, 0.2] буде збережено в створенний буфер, який називається A-буфер. Кольори фрагментів, що залишилися, будуть змішані між собою ([C3, 0.6] та [C4, 0.9]), після чого записані у буфер. Тобто буде виконана операція tail blend. Далі відбувається сортування значень, які були записані в A-буфер, за глибиною. Сортування відбувається в послідовності від найдальшого від камери до найближчого. Після цього значення кольорів будуть змішані. На рис. 1.6 наведено діаграму роботу даного методу.

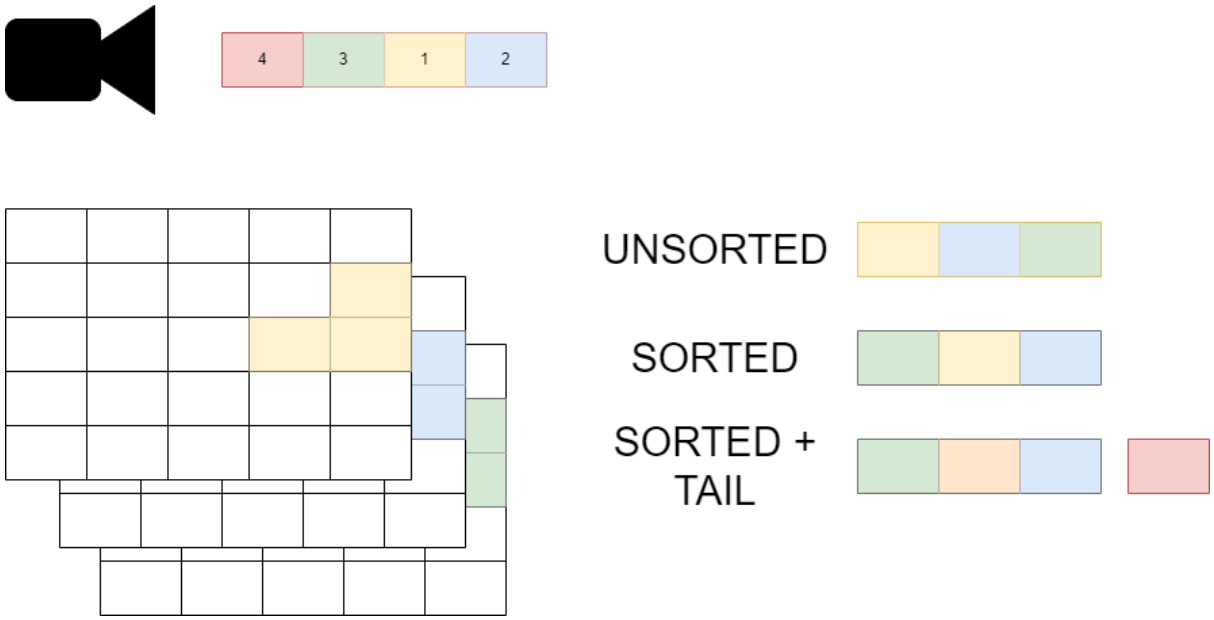

Рисунок 1.6 – Simple-метод при N = 3

Оскільки кількість записаних значень обмежено значенням N та в буфер записуються перші пари в порядку надходження до відповідного буфера, без урахування глибини, то можлива поява артефактів (рис. 1.7). Артефакт – це ненавмисний візуальний елемент або спотворення, що з'являється в кінцевому зображенні. Артефакти можуть бути спричинені різними факторами, такими як помилки в алгоритмі рендерингу, обмеження апаратного забезпечення або проблеми з вхідними даними. До найпоширеніших артефактів у комп'ютерній графіці належать аліасинг (нерівні краї або східчасті візерунки), мерехтіння, смуги (видимі смуги кольору або яскравості) і спотворення (розтягнення, викривлення або розриви зображення). Артефакти можуть знижувати загальну якість зображення, тому важливо мінімізувати їх появу.

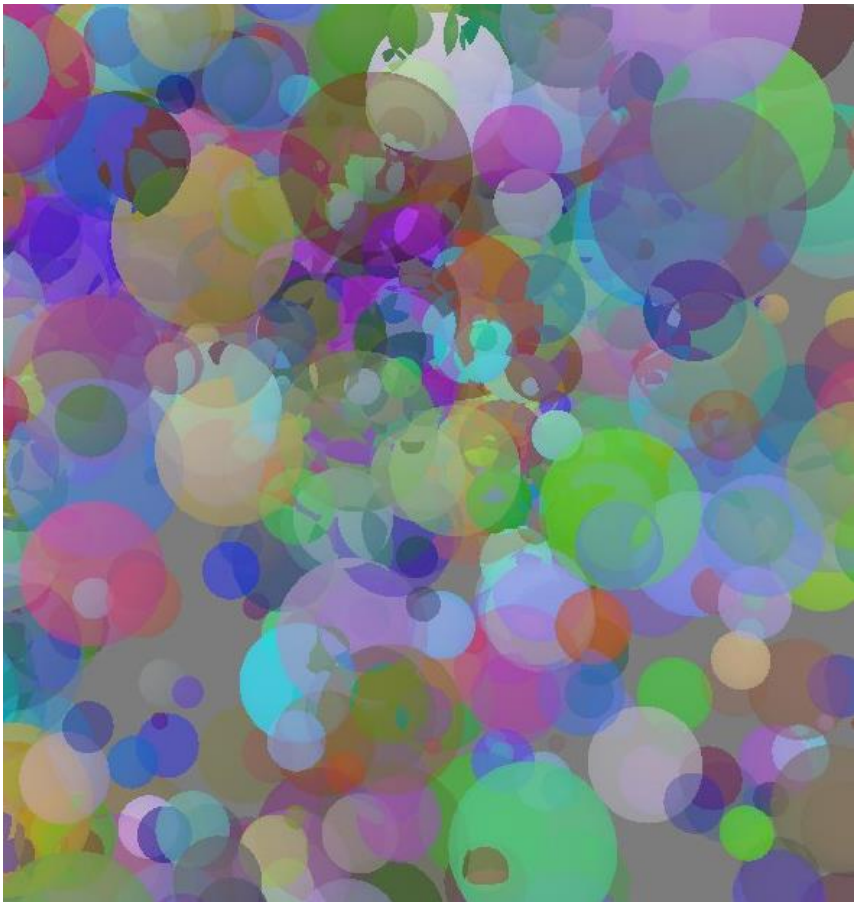

Рисунок 1.7 – приклад мерехтіння при низькому значені N

З наведеного прикладу можна побачити умову появи такої проблеми. Найближчим фрагментом до камери є [C4, 0.1], але з причини того, що цей

фрагмент був відкинутий і записаний у хвіст, то розташування фрагменту відносно камери не буде враховуватися і призведе до непередбачуваних візуальних ефектів, тобто до артефакту. Якщо сцена не є комплексною, відсутній перетин багатьох об’єктів між собою, то даний артефакт буде рідкістю. При збільшенні значення N, ймовірність появи такої проблеми зменшується. Але це потребує більшої кількості ресурсів для зберігання та обробки більшої кількості пар значень в A-буфері. На рис. 1.8 зображено візуалізації прозорих об’єктів при N = 32.

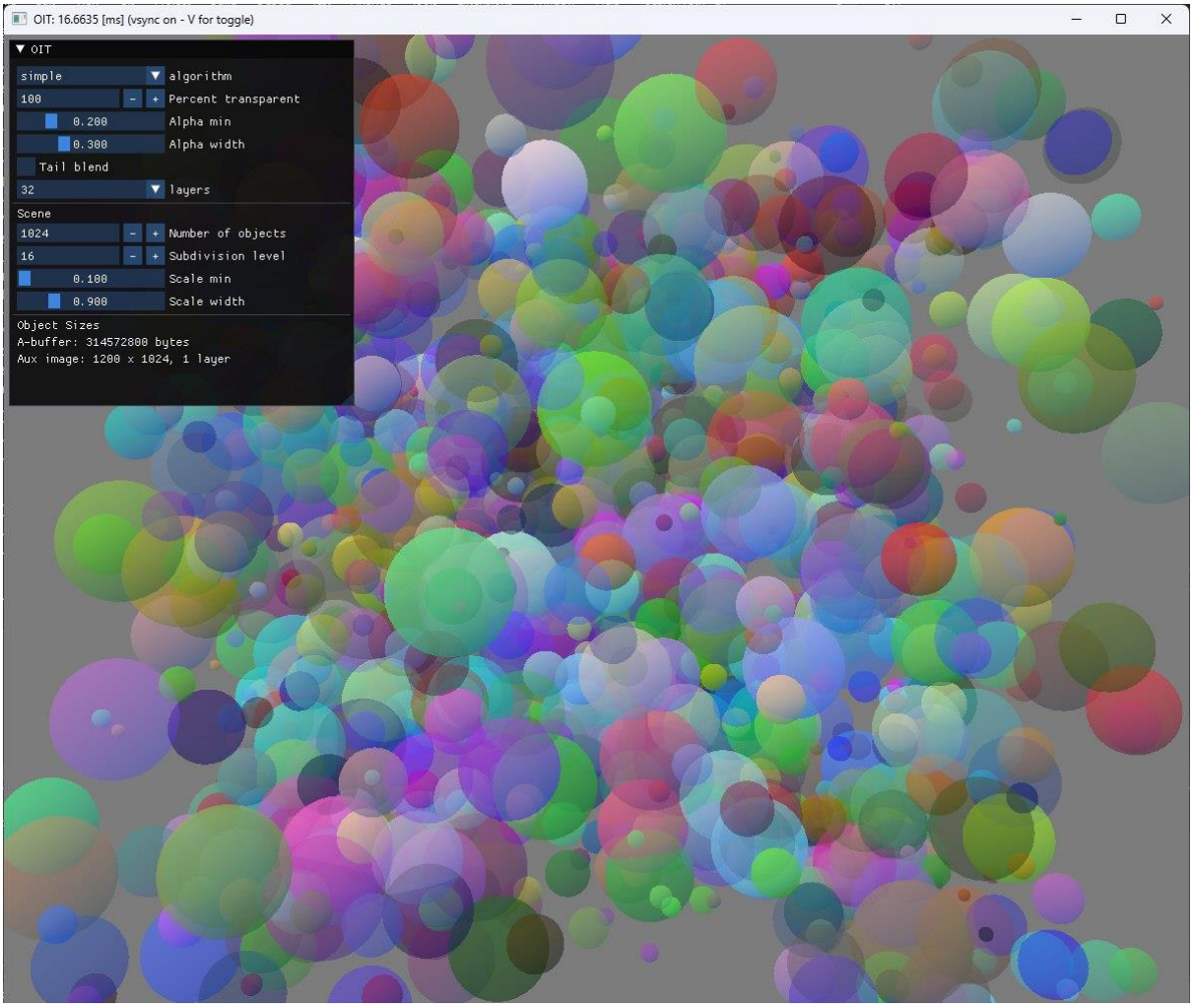

Рисунок 1.8 - візуалізація прозорих об’єктів за допомогою Simple-методу

1.4 Spinlock-метод

Для вирішення проблему артефакту описаного в Simple-методі необхідно виконувати сортування фрагментів. Але оскільки графічний процесор обробляє інформацію багатопоточно, то таке сортування небезпечно з точки зору втрати інформації в критичних зонах (зонах де потоки можуть одночасно змінювати інформацію). Це вирішується використанням так званого спінлоку. Для кожного пікселя створюється змінна яка контролює чи відбувається обробка інформації певним потоком. Коли поток заходить у критичну зону змінна набуває значення 1, а на виході - змінна набуває значення 0. Фрагменти, які виходять за N, будуть змішані між собою і записані в буфер [5].

Хоча цей алгоритм і вирішує проблему артефактів, які виникають без сортування фрагментів за глибиною. Але його не рекомендовано використовувати, оскільки він не є оптимальним рішеням. Багато потоків будуть перебувати в очікуванні, що порушує головну логіку роботи графічного процесору. На рис. 1.9 видно результат роботи даного алгоритму.

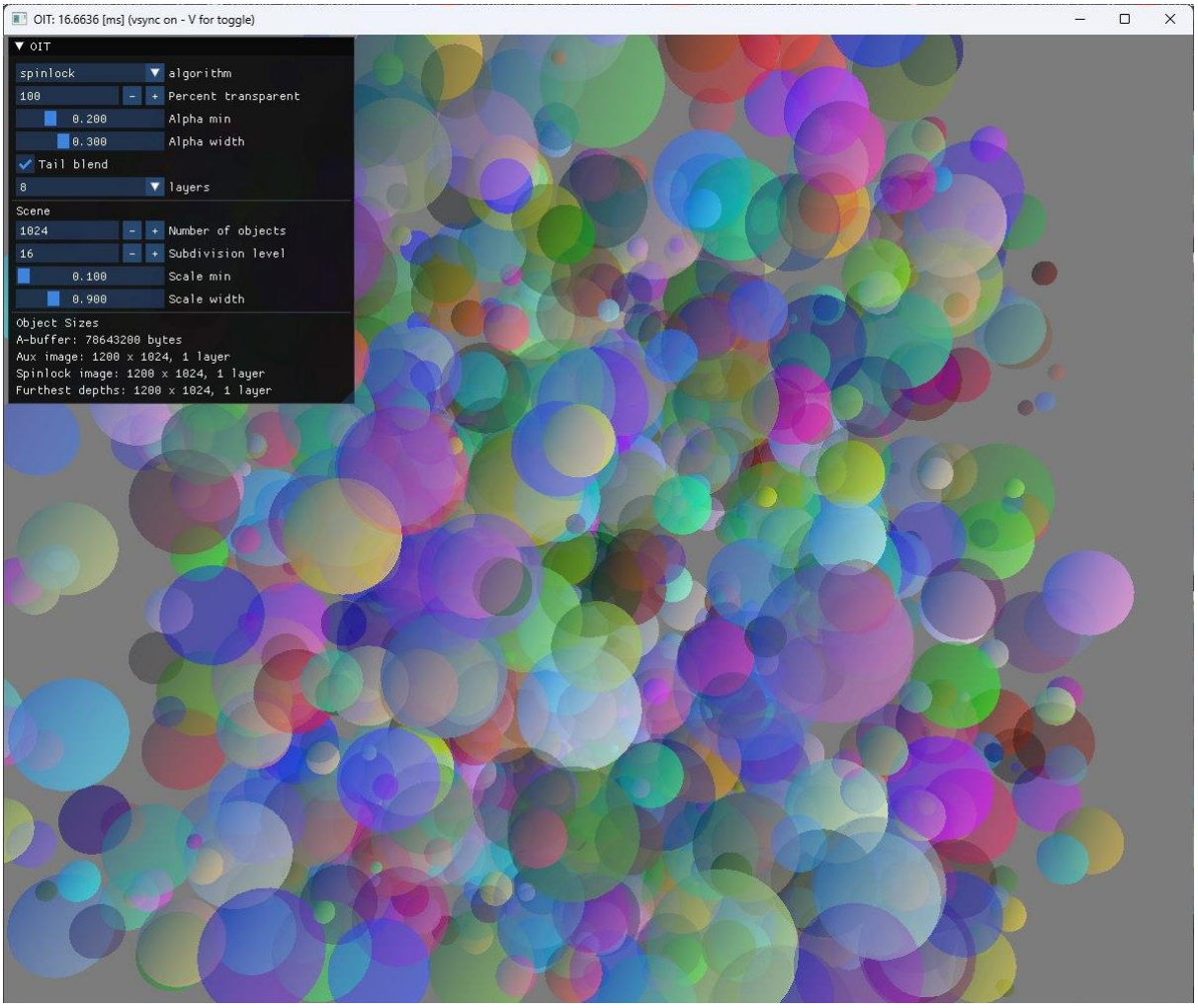

Рисунок 1.9 - візуалізація прозорих об’єктів за допомогою SpinLock-методу

1.5 LinkedList-метод

Ідея методу полягає у тому, щоб зберігати інформацію про фрагменти для кожного пікселю на екрані у зв’язаних між собою списках.

Зв'язаний список - це лінійна колекція елементів даних, які називаються вузлами, кожен з яких вказує на наступний вузол у послідовності. Зв'язаний список - це динамічна структура даних, в якій вузли розміщуються в пам'яті в міру необхідності, що дозволяє ефективно використовувати пам'ять [10].

Кожен вузол зв'язаного списку містить два поля: дані, що зберігаються у вузлі, і посилання, або вказівник, на наступний вузол у списку. Перший вузол

списку називається заголовком, а останній вузол списку має нульове посилання, що вказує на кінець списку.

Зв'язані списки мають кілька переваг над масивами, зокрема можливість вставляти і видаляти дані в постійний час, а також ефективно зберігати дані різного розміру. Однак зв'язані списки мають і певні недоліки, зокрема потребу в додатковій пам'яті для зберігання вказівників і неможливість ефективного доступу до певного елемента списку без обходу всього списку.

Для реалізації такого підходу вирішення проблеми візуалізації прозорих об’єктів необхідно використовувати два буфери: головний та вузловий буфер. Головний буфер має розмір відповідний до розміру екрану на якому будується зображення. В ньому записується індекс вузлового буферу, який відповідає за останній записаний фрагмент у цьому пікселі. Вузловий буфер зберігає в собі інформацію про напівпрозорі фрагменти. Ця інформація складається з трьох частин: колір фрагменту, глибина та індекс попереднього фрагменту [7]. На рис. 1.10 зображено процес побудови буферів.

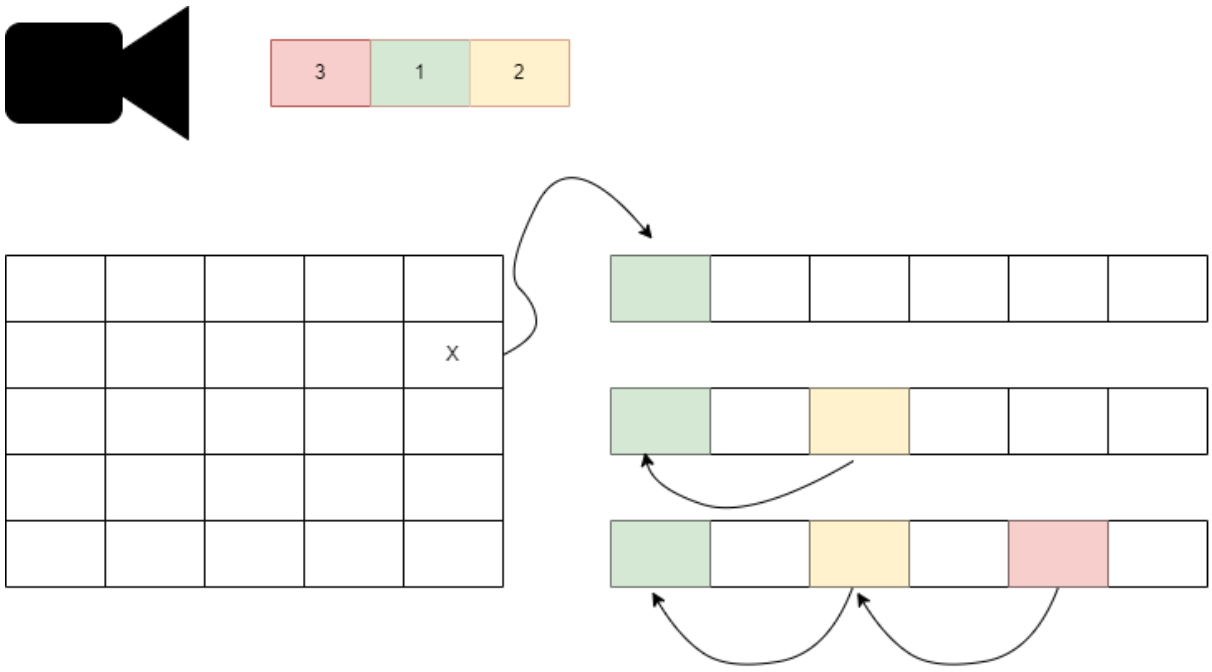

Рисунок 1.10 – Побудова буферів в LinkedList-методі

Після того як всі фрагменти було пройдено, головний буфер і вузловий буфер побудовані, можна починати виводити зображення на екран. Для кожного пікселя, починаючи із індексу записаного у головному буфері, виконується прохід по створеному зв’язаному списку та збирається інформація про всі фрагменти, які відповідають даному пікселю. Після чого ця інформація записується у тимчасовий масив (рис. 1.11). Ці фрагменти сортуються, їх колір змішується та отримується кінцеве значення пікселя на екрані [7].

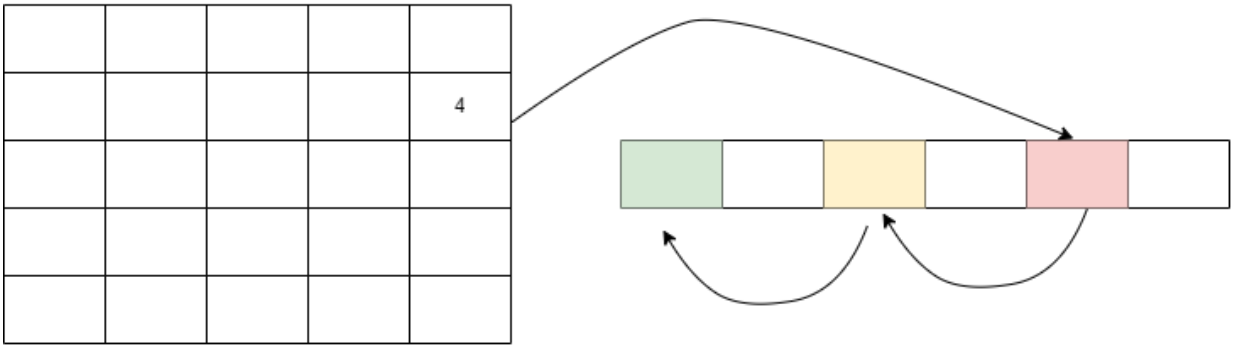

Рисунок 1.11 – Прохід по створеному списку для отримання кінцевого значення кольору пікселя

Перевагою даного методу є те, що не обмежується максимальна можлива кількість фрагментів для окремого пікселя. Тобто один піксель може мати 1 фрагмент, а інший 20. В попередньо описаних методах, кожен піксель мав конкретне максимальне значення фрагментів (обмежувалося значення N), а це збільшує шанс появи артефактів при отримані зображення. На рис. 1.12 представлено зображення отримане з допомогою LinkedList-методу візуалізації.

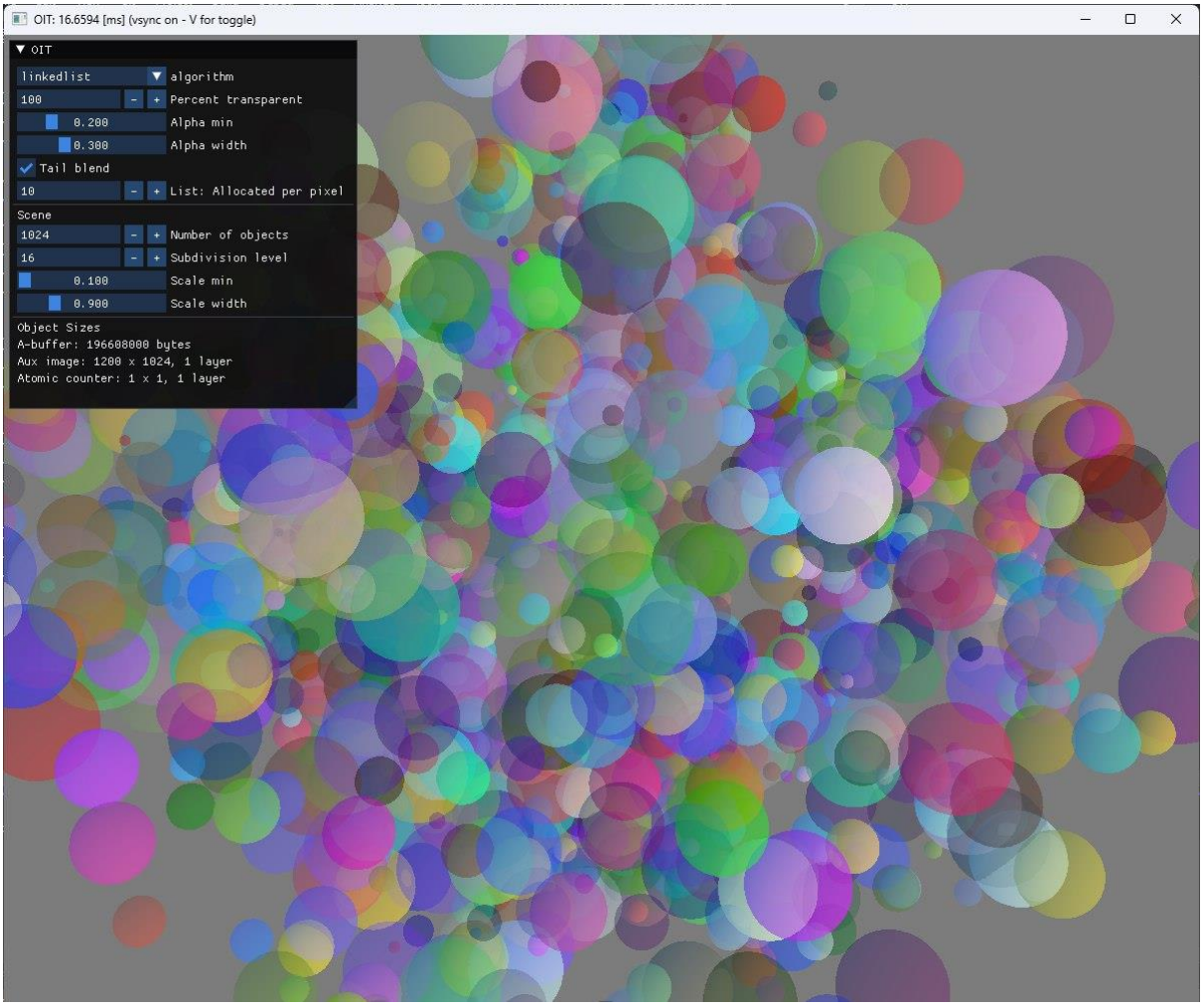

Рисунок 1.12 - візуалізація прозорих об’єктів за допомогою LinkedList-методу

1.6 Loop32-метод

Особливістю цього алгоритму є використання атомарної операції, що виконує пошук мінімального значення в заданому проміжку пам’яті.

Цей метод виконує два обчислювальних проходи для визначення кінцевого значень пікселя на екрані. На першому проході виконується сортування фрагментів за глибиною використовуючи атомарну операцію для пошуку мінімуму. На другому проходів, використовуючи відсортовані позиції із попереднього проходу будується A-buffer з кольорами. Кольори, що не увійшли в діапазон розміру буферу (N) змішуються між собою [7]. Після чого виконується

обчислення кінцевого значення кольору пікселя, виконуючи змішування описане в розділі 1.3.

Цей метод схожий на Simple метод. Але в Loop32 відбувається сортування, що зменшує ймовірність появи небажаних ефектів. А використання атомарних операції гарно вирішує проблему критчних зон, що виникає при паралельних обчислення на графічному процесорі.

Недоліком цього алгоритму є те що він не підтримує MSAA. MSAA розшифровується як "Multisample Anti-Aliasing" - це техніка, що використовується в комп'ютерній графіці для зменшення візуальних артефактів, відомих як згладжування. Згладжування виникає, коли роздільна здатність зображення недостатня для точного відображення деталей об'єкта, що відображається, що призводить до нерівних країв, мерехтливих візерунків та інших візуальних спотворень [5].

MSAA працює, беручи кілька зразків кожного пікселя зображення, а потім об'єднуючи ці зразки для отримання більш плавного і точного представлення зображення. Кількість зразків на піксель визначається рівнем MSAA, який використовується; наприклад, 2x MSAA бере два зразки на піксель, 4x MSAA - чотири зразки на піксель і так далі (рис 1.13).

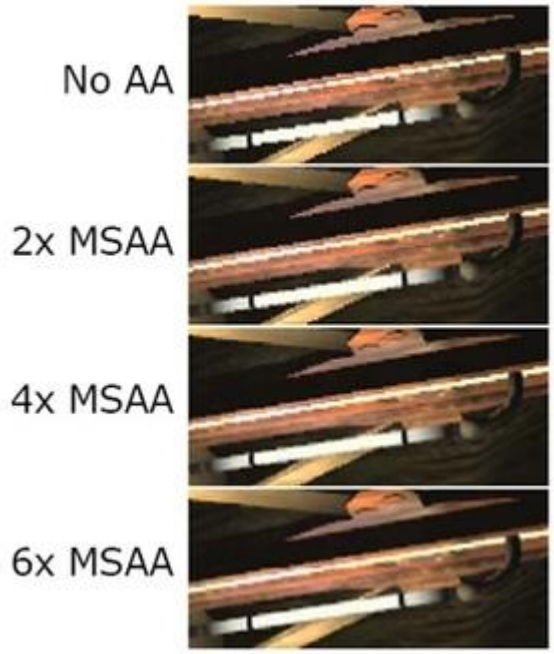

Рисунок 1.13 – приклад зображення з різною кількість зразків, що використуваються в MSAA

Під час процесу рендерингу MSAA відбирає зразки з декількох точок в межах пікселя і обчислює середній колір (рис 1.14). Порівнюючи колір у кожній з цих точок, алгоритм здатен згладжувати краї об'єктів на зображенні та зменшувати нерівності зображення. Результатом є більш реалістичне, високоякісне зображення, яке виглядає набагато більш плавним і природним, ніж зображення, відтворене без згладжування.

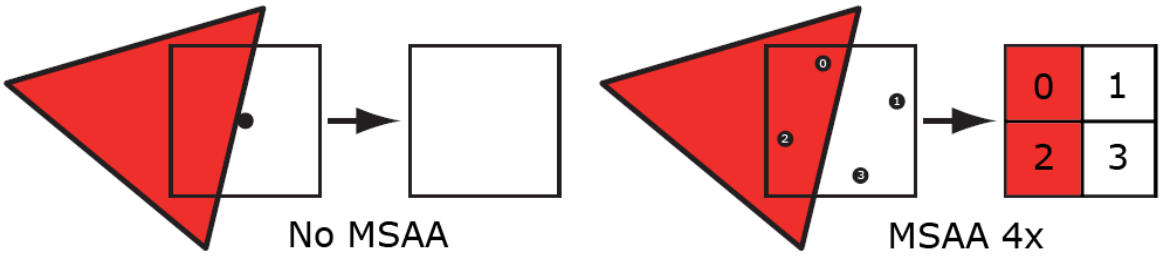

Рисунок 1.14 – вибір кольору пікселя на основі MSAA

MSAA широко використовується в сучасних програмах комп'ютерної графіки, включаючи відеоігри, програмне забезпечення для 3D-моделювання та інші інструменти візуалізації. Однак він може бути дорогим в обчислювальному плані, оскільки вимагає додаткової обчислювальної потужності для рендерингу кожного пікселя з декількома вибірками.

Алгоритм Loop32 потребує два проходи, що в свою чергу означає більше операцій та взаємодії процесора з драйвером графічного процесору. Це призводить до зменшення ефективності програмного забезпечення для комп’ютерних систем. На рис. 1.15 відображено сцену створену з використання Loop32-методу.

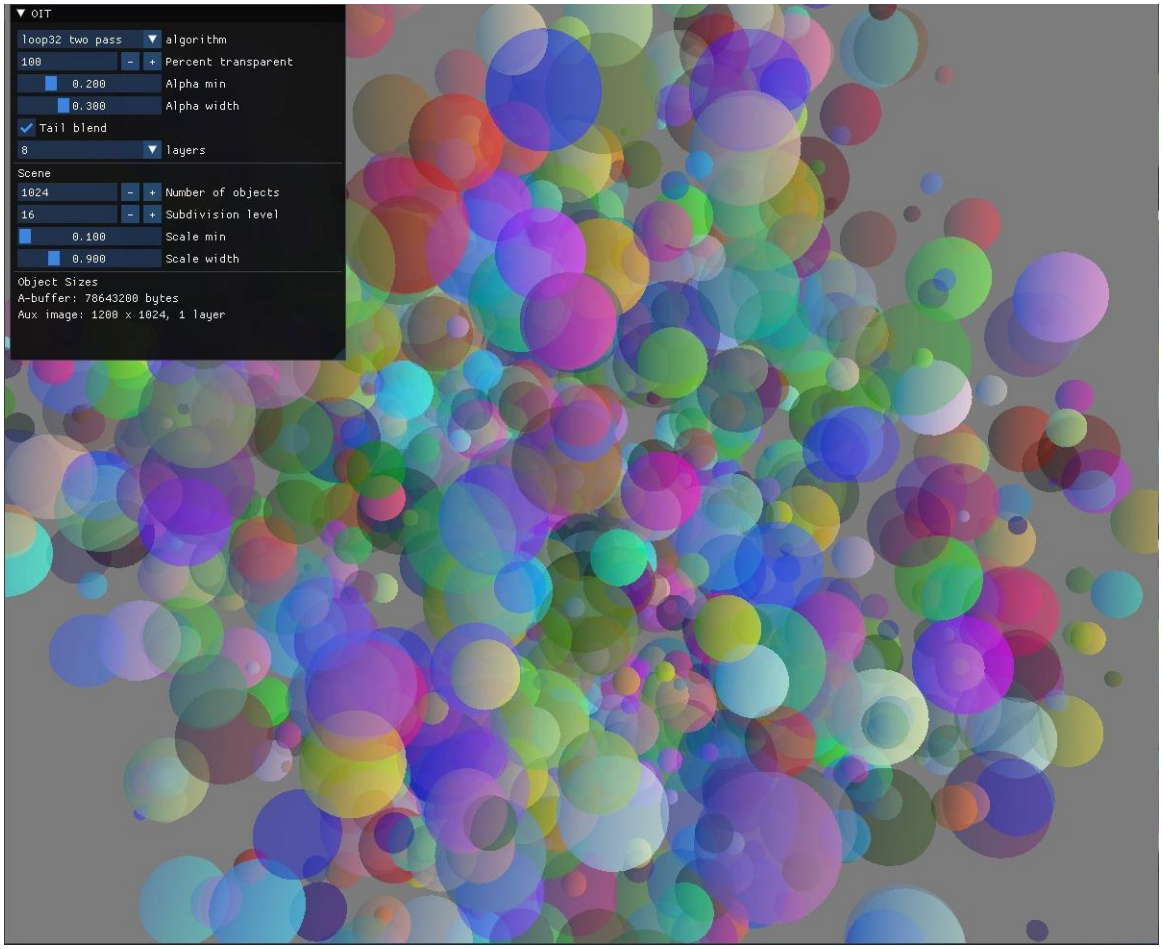

Рисунок 1.15– візуалізація прозорих об’єктів за допомогою Loop32-методу

1.7 Loop64-метод

Це модифікація методу Loop32, яка використовує атомарну операцію, що працює з 64 бітними числами. Використання такої операції дає змогу зменшити кількість необхідних проходів з двох до одного. Такий результат отримується завдяки тому, що ми можемо зберігати колір та глибину фрагменту разом, та сортувати їх за допомогою атомарної операції. Але не всі графічні процесори підтримуються таку операцію [7].

1.8 WeigtedBlended-метод

WeightedBlended-метод є приближеним, на відміну від інших представлених методів.

В даному методі використовуються вагові коєфіцієнти, які обраховуються заданою інженером функцією. Функція обирається в залежності від сцени, яку необхідно водобразити. В різних ситуаціях кожна функція буде мати свої недоліки та переваги. На основі отриманих коефіцієнтів виконується зміщування фрагментів для отримання кінцевого значення кольору пікселя [8].

Метод потребує створення двох буферів: значення кінцевого кольору outColor та значення видимості outReveal. Остаточний результат буде описуватися формулою [8]:

$$Result = vec4\left(\frac{outColor.rgb}{outColor.a}, 1 - outReveal\right),$$

де $outColor = (w_0 * c_0) + (w_1 * c_1) + \cdots + (w_n * c_n)$,

$outReveal = (1 - alpha_0) * (1 - alpha_1) + \cdots + (1 - alpha_n)$.

Хоча метод і є приближеним, але при правильно обраній функції для знаходження вагових коефіцієнтів фрагментів в результаті отримуються достатньо реалістичні зображення прозорих об’єктів на сцені. При цьому цей метод потребує набагато менше апаратних ресурсів, у порівнянні з точними методами описаними до цього. Як обчислювальних можливостей, так і пам’яті.

Так цей метод не є чутливим до появи артефактів. На рис. 1.16 показано результат отриманий при використанні даного методу.

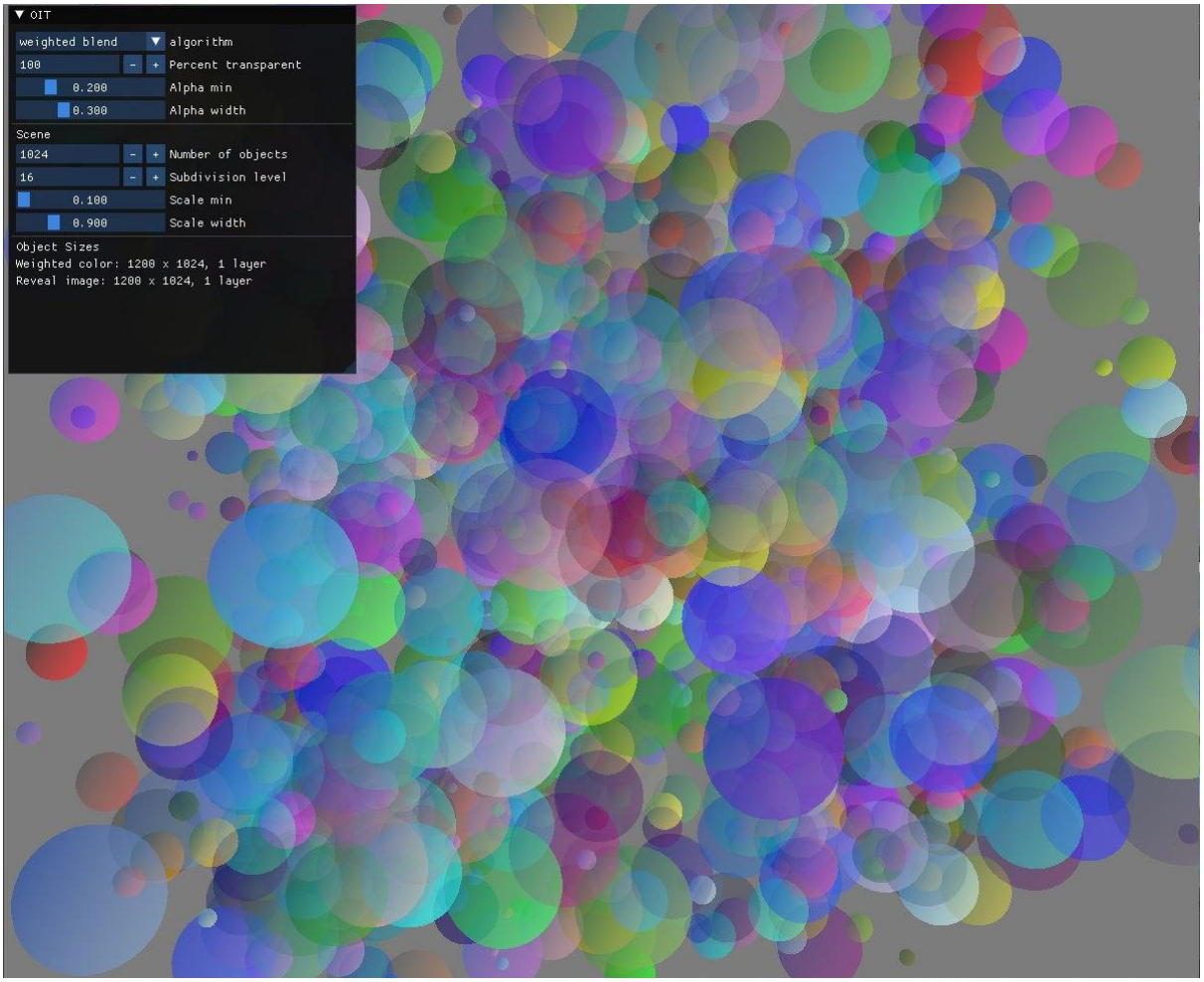

Рисунок 1.16 – візуалізації прозорих об’єктів за допомогою WeightedBlended-методу

1.9 Порівняння методів

Simple-метод є не складним у реалізації. Але через таку простоту, цей метод схильний до появи артефактів, що є великою проблемою при створенні реалістичних зображень на комп’ютерних системах.

Для вирішення цієї проблеми необхідно виконувати сортування. Spinlock-метод виконує сортування завдяки блокуванні критичних зон, але це призводить до простою ресурсів графічного процесору, що є не допустимим, оскільки сильно впливає на продуктивність роботи комп’ютерної системи.

Loop32-метод та LinkedList-метод вирішують проблеми описані вище виконуючи сортування без блокування роботи графічного процесору. Це є великою перевагою.

WeightedBlended-метод виконує приближені обчислення кінцевого значення кольору пікселя. Але при правильно підібраній функції, яка обраховує вагові коефіцієнти, кінцеве зображення сцени з наявністю прозорих об’єктів буде достатньо реалістичним. При цьому даний метод потребує найменше ресурсів в порівнянні з іншими методами.

Нижче наведено порівняльну таблицю розглянутих методів (табл. 1.1), графік залежності часу необхідного на обчислення кадру від N (рис. 1.17) та графік необхідної кількості пам’яті від N (рис. 1.18).

Таблиця 1.1 – Порівняння методів зображення прозорих об’єктів

Назва	**Залежність точності**	**Підтримка MSAA**	**Байтів на піксель**	**Сортування фрагментів**	**Кількість проходів**
Simple	N та початкового порядку фрагментів	Так	8 * N + 4 байт	Ні	1
LinkedList	N та розміру A-буферу	Так	16(для фрагменту) + 4 байт	Так	1
Loop32	N	Ні	8 * N + 4 байт	Так	2
Spinlock	N	Так	8 * N + 12 байт	Так	1
WeigthedBlended	Від обраної функції апроксимації	Так	20 байт	Так	1

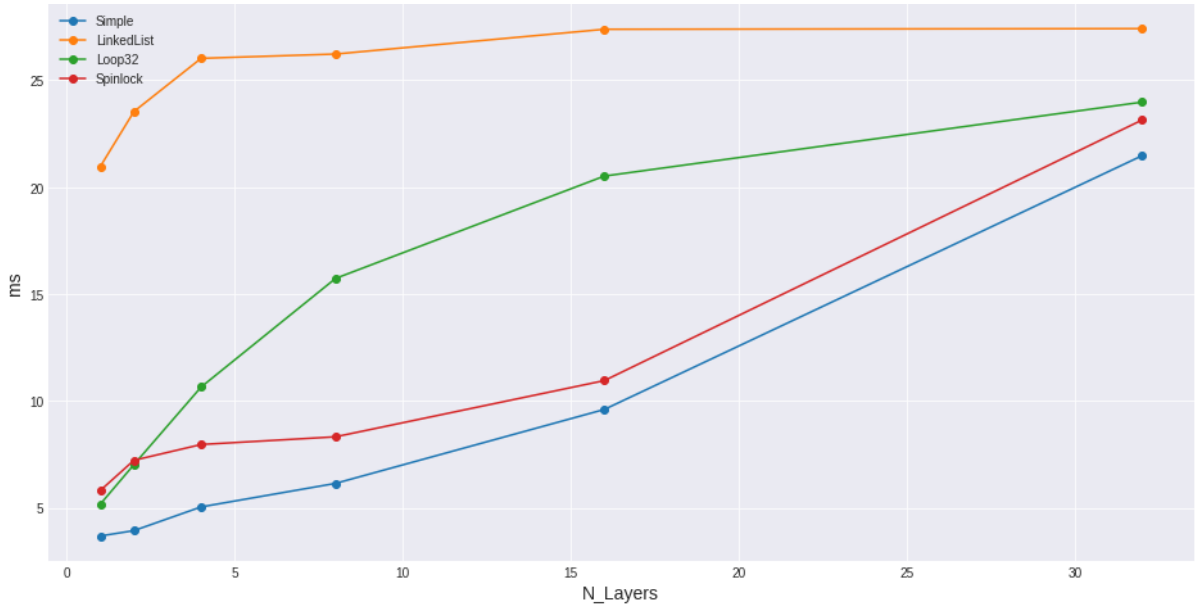

Рисунок 1.17 – Час на відтворення зображення

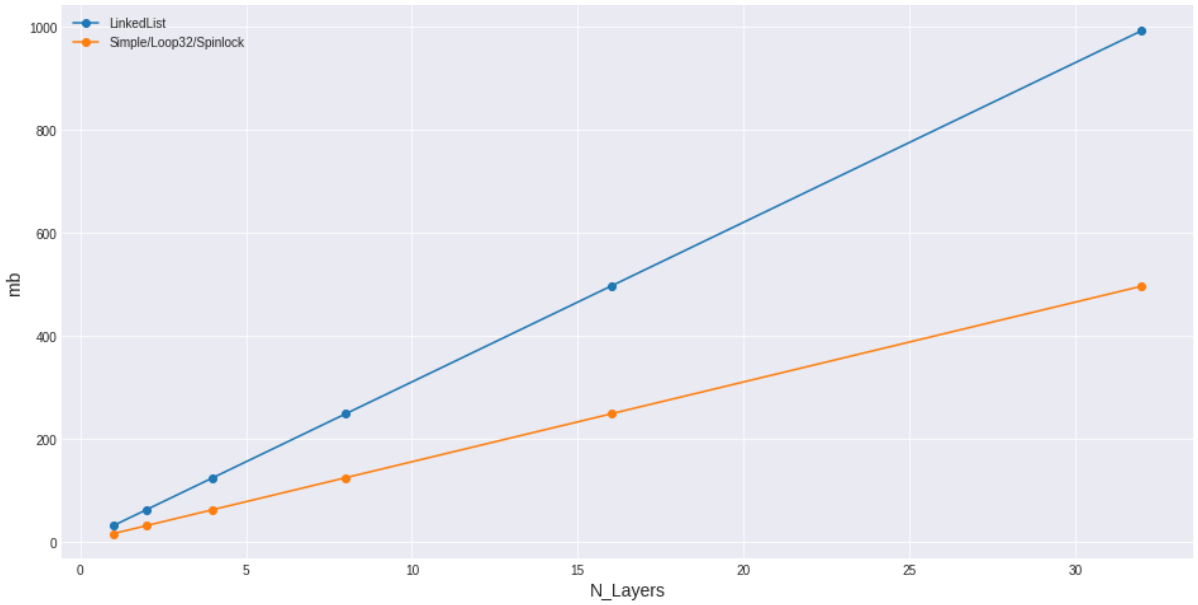

Рисунок 1.18 – Кількість затраченої пам’яті

Висновки до розділу

У цьому розділі було розглянуто п’ять різних методів зображення прозорих об’єктів. Для порівняння було розроблено програмне забезпечення за допомогою засобів Vulkan, у якому реалізовані усі описані методи.

Simple метод є найбільш тривіальним. Через те що в ньому відсутнє сортування фрагментів він схильний до появи артефактів. LinkedList вирішує цю проблему створюючи зв’язні списки та сортуючи фрагменти. Але використання списку створює залежність точності обчислень від кількості необхідної пам’яті для A-buffer, що може стати проблемою. Loop32, ще один точний алгоритм, який вирішує проблему артефактів. Але для отримання результату необхідно виконувати два обчислювальних проходи. Це може бути оптимізовано за допомогою засобів Vulkan API. Spinlock метод використовує засоби синхронізації між потоками, але це призводить до того, що частини GPU можуть простоювати, що не є припустимим. Тому цей метод є найгіршим із представлених. Weighted Blended є приблизним методом, єдиний з розглянутих. Хоча він і є приближеним, але він видає достатньо реалістичні результати. Рівень реалістичності залежить від обраної функції апроксимації. При цьому він потребує найменше пам’яті. Тому, якщо ресурси комп’ютерних систем є обмежені, цей метод є гарним вибором.

Для оптимізації за допомогою API Vulkan було обрано алгоритм Loop32.

2. VULKAN API ТА ЗАСОБИ ДЛЯ ВІЗУАЛІЗАЦІЇ

2.1 Опис Vulkan API

Vulkan - це низькорівневий API, який надає можливості прямого доступу до ресурсів графічного процесу та контролю його роботи. У порівнянні з OpenGL та Direct3D має менші витрати ресурсів та затримку при виконанні графічних команд [3].

Vulkan працює на більшості сучасних операційних систем, в порівнянні з OpenGL та Direct3D, які використовуються лише для певних систем. Vulkan працює на Android, Linux, BSD Unix, Respberry Pi, Tize, Windows. MoltenVK надає можливість використовувати Vulkan на macOs, iOS та tvOS, за допомогою взаємодіє з Metal API від Apple.

Vulkan використовує менше ресурсів центрального процесору в порівнянні з іншими графічними API. В часності, це досягається завдяки реалізації пакетної обробки команд [3].

Пакетна обробка команд в API Vulkan - це техніка, яка дозволяє програмі записати послідовність команд один раз і виконати їх кілька разів з мінімальними накладними витратами. Це може підвищити продуктивність і зменшити використання процесора за рахунок уникнення надлишкових викликів API і змін стану. Пакетна обробка команд може бути досягнута за рахунок використання буферів команд і пулів команд у Vulkan [3].

Буфер команд - це об'єкт, що зберігає список команд, які можна поставити в чергу на виконання. Пул команд - це об'єкт, який виділяє пам'ять для буферів команд і керує їхнім життєвим циклом. Буфери команд можуть бути первинними або вторинними, залежно від рівня їхньої гнучкості та залежності. Первинні буфери команд можуть бути подані безпосередньо в чергу, тоді як вторинні буфери команд можуть виконуватися тільки як частина первинного буфера команд.

Щоб використовувати пакетну обробку команд у Vulkan, програма повинна виконати наступні кроки:

1. Створити пул команд за допомогою функції vkCreateCommandPool().

2. Виділити один або декілька буферів команд з пулу команд за допомогою функції vkAllocateCommandBuffers().

3. Запис команд у командні буфери за допомогою різних функцій vkCmd*().

4. Подати буфери команд у чергу за допомогою функції vkQueueSubmit().

5. Дочекатися завершення виконання буферів команд у черзі за допомогою функцій vkQueueWaitIdle() або vkWaitForFences().

6. Звільнити буфери команд функцією vkFreeCommandBuffers() або скинути їх функцією vkResetCommandBuffer() для повторного використання.

7. Знищити пул команд за допомогою функції vkDestroyCommandPool(), коли він більше не потрібен.

Vulkan розроблявся з ідеєю багатопоточної роботи центрального процесору. Це пришвидшує роботу програмного забезпечення та зменшує простоювання процесору. Для OpenGL 4 та Direct3D 11 не оптимальне використання ресурсів CPU було великою проблемою. Оскільки вони пристосовані лише для використання одноядерного центрального процесору.

Ще одною особливістю Vulkan є використання SPIR-V (Standard Portable Intermediate Representation).

SPIR-V - це відкрита стандартна проміжна мова для паралельних обчислень і графіки, розроблена Khronos Group. Вона використовується Vulkan, OpenGL та OpenCL як загальне представлення для шейдерів та ядер, що забезпечує крос-платформну портативність та інтероперабельність. SPIR-V розроблено так, щоб його можна було розширювати, налагоджувати та оптимізувати для підвищення продуктивності та надійності [3].

SPIR-V переводить код написаний інженером в бінарний код.

SPIR-V підтримує різні моделі виконання, такі як вершина, фрагмент, обчислення, трасування променів і затінення сітки. Кожна модель виконання визначає, як викликається функція і які вбудовані змінні доступні.

SPIR-V також підтримує різні можливості, такі як шейдер, ядро, геометрія, тесселяція, атомарне зберігання, операції з підгрупами та запит променів. Можливості використовуються для оголошення того, які функції потрібні або використовуються модулем. Різні API можуть надавати різні набори можливостей для модулів SPIR-V.

Для створення модулів SPIR-V розробники можуть використовувати різні мовні інтерфейси, такі як GLSL, HLSL, OpenCL C або C++. Ці інтерфейси можуть компілювати вихідний код у бінарні модулі SPIR-V, які потім можуть бути сприйняті драйверами Vulkan, OpenGL або OpenCL. Крім того, розробники можуть використовувати такі інструменти, як SPIR-V Tools або SPIR-V Cross, щоб збирати, розбирати, оптимізувати, компонувати, перевіряти або аналізувати модулі SPIR-V.

SPIR-V - це потужна та гнучка проміжна мова, яка дозволяє розробникам писати шейдери та ядра для різних API та платформ з високою продуктивністю та портативністю. Це пришвидшує ініціалізацію застосунків, оскільки дозволяє зберігати скомпільовані програми. Також зменшує розмір програм драйверів для графічних процесорів, бо зникає необхідність створення компіляторів для різних мов програмування шейдерів.

2.2 Порівняння з OpenGL

Silicon Graphics випустила відкриту графічну бібліотеку OpenGL у червні 1992 року як API з відкритим вихідним кодом для забезпечення інтерфейсу для графічного програмування. З роками, завдяки вдосконаленню та впровадженню інших розширень, вона перетворилася на міжмовний API [6].

Поворотним моментом стало впровадження розширення версії 4.2, яке дозволило використовувати обчислювальні шейдери, що уможливило

синхронізацію потоків і спільне використання пам'яті. Це перетворило OpenGL на те, чим він є сьогодні - API для апаратно-прискореного рендерингу 2D і 3D векторної графіки.

Khronos Group, AMD та DICE випустили Vulkan у 2016 році як крос-платформний API з відкритим вихідним кодом, що надає інтерфейс для програмування та рендерингу відеоігор та інших додатків 3D-графіки в реальному часі. Це мала бути сучасна версія OpenGL, яка б відповідала швидкозмінним потребам ігрової індустрії, а також використовувала всі можливості сучасних графічних процесорів. Спочатку він був відомий як Next Generation OpenGL, а потім був перейменований на Vulkan.

Дизайн OpenGL робив його придатним для графічних робочих станцій з прямим рендерингом. Як наслідок, він не міг бути придатним для усіх настільних платформ, незалежно від операційної системи. Потокова модель перешкоджає паралельній генерації та виконанню команд. Отже, вона не могла працювати на певних платформах, таких як планшети, смартфони та консолі [6].

На відміну від OpenGL та інших попередніх API, Vulkan в першу чергу призначений для роботи на різних платформах, особливо на мобільних і консолях. Його багаторівневий дизайн дозволяє перевіряти, профілювати та налагоджувати код, не впливаючи на продуктивність; отже, він може підтримувати інноваційне використання крос-вендорних GPU-інструментів.

Завдяки своїй багаторівневій структурі Vulkan пропонує прямий контроль над графічним процесором - можливість, яку вимагають найскладніші ігрові рушії. Він також покладається на буфери команд, які є заздалегідь згенерованими командами, таким чином усуваючи необхідність компіляції або перевірки під час кожного циклу рендерингу. В результаті, це значно зменшує навантаження на процесор [11].

У OpenGL виконані виклики не кешуються, тому для зміни та перевірки стану драйвера потрібно багато звернень до режиму ядра. З іншого боку, Vulkan використовує буфери команд, що покращує використання ресурсів процесора.

Друга відмінність - це функція паралельної генерації буферів у Vulkan. Завдяки їй Vulkan використовує потужність всіх ядер процесора, з чим OpenGL боровся, враховуючи, що він був розроблений в той час, коли багатоядерні процесори були нечуваними. Оскільки сучасні процесори мають щонайменше вісім ядер, Vulkan є єдиним API, який може належним чином використовувати можливості, які вони пропонують.

Обчислювальні шейдери допомагають максимально ефективно використовувати потужність графічного процесора (GPU) замість того, щоб залежати лише від потужності центрального процесора системи. Однак, ці API керують шейдерами по-різному.

OpenGL використовує мову високого рівня C++ з компілятором GLSL. Компілятор працює на рівні драйвера і пише шейдери, які потім транслюються під час виконання програми в машинний код для графічного процесора.

Vulkan, однак, скорочує цей процес, використовуючи стандартне портативне проміжне представлення SPIR-V. Він має компілятор-транслятор для GLSL, HLSL і LLVM, який перетворює коди з усіх інших високорівневих API у SPIR-V. Потім Vulkan приймає готовий до виконання двійковий проміжний вхід і використовує його на етапі шейдерування. На додаток до GLSL, Vulkan підтримує різні специфічні мови, інструменти, фреймворки та ядра OpenGL C.

Рівень абстракції в OpenGL дуже високий, оскільки більшість операцій відбувається за лаштунками. Вони включають управління ресурсами, перевірку та валідацію помилок, попередню компіляцію шейдерів та деякі синхронізації. Такий неявний дизайн гарантує, що розробнику не потрібно писати код для кожної дії, але також робить його менш передбачуваним; ви ніколи не знаєте, чи буде ваш код робити те, що він повинен робити, поки він не зробить. Крім того, драйвер планує надсилання завдань на обробку, що призводить до переривань рендерингу [6].

Vulkan відрізняється цим від OpenGL. Vulkan - це явний API, що означає, що драйвер не відстежує ресурси і не визначає їхні взаємозв'язки. Це робиться

на рівні програми і вимагає від розробника написання коду для кожного процесу. Перевага цього методу полягає в тому, що завдання подаються на виконання заздалегідь. Як результат, обробка завдань у Vulkan є передбачуваною, оптимізованою та безперервною. Таким чином, рендеринг у Vulkan не страждає від затримок і збоїв, як це часто буває в OpenGL.

OpenGL, як неявний API, залишає відповідальність за управління ресурсами та відстеження стану на рівні драйверів. Як результат, прикладний рівень у OpenGL дуже тонкий. Рендеринг займає більше часу, ніж якби більшість завдань з розподілу ресурсів виконувалася на прикладному рівні.

З іншого боку, Vulkan розроблений так, щоб бути максимально наближеним до графічного обладнання. Щоб досягти цього, він залишає управління ресурсами, логікою та станами для додатків. Дозволяючи прикладному рівню розподіляти ресурси, Vulkan надає розробнику програми повний контроль над комп'ютерними ресурсами хост-системи. Таким чином, надання програмному забезпеченню прямого доступу до графічного процесора робить рендеринг на GPU максимально швидким і покращує продуктивність процесора.

Ще однією ключовою відмінністю між двома API є рівень контролю, який має розробник над пам'яттю системи. У OpenGL такий контроль відсутній, оскільки драйвери використовують внутрішні евристики для розподілу ресурсів системи, включаючи пам'ять. Основною проблемою такого режиму керування пам'яттю є те, що він залежить від постачальника і може призвести до неоптимального розподілу або затримок при переміщенні ресурсу.

Vulkan показує різні доступні типи пам'яті в системі і дозволяє розробнику розподіляти і звільняти пам'ять на свій розсуд. Програміст повинен вибрати тип пам'яті, який найкраще підходить для передбачуваної мети. Окрім типів пам'яті, Vulkan також надає розробнику доступ до доступних розширень, черг буферів команд та фізичних пристроїв. Тому, оскільки немає втручання драйвера,

розробник може змінювати поведінку Vulkan відповідно до вимог програмного забезпечення [3].

OpenGL має дуже погану багатопоточність, що ускладнює використання переваг обчислювальної потужності сучасних процесорів. Через це він також має високі накладні витрати на процесор. Тут Vulkan йде протилежною траєкторією. Він спеціально розроблений для використання повної багатопотоковості процесора в явному вигляді. Завдяки навмисному управлінню ресурсами, завдання розділені на різні потоки, від створення до виконання.

Портування програмного забезпечення між десктопними та мобільними платформами неможливе у OpenGL. Натомість існує дві окремі версії API: API для настільних комп'ютерів (OpenGL) та OpenGL ES, вбудований API для мобільних платформ. Але навіть у цьому випадку пріоритет часто надається десктопним платформам. Крім того, API GL розвиваються значно повільніше, ніж апаратне забезпечення та технології GPU, тому навіть найновіші реалізації часто придатні лише для GPU попереднього покоління.

Vulkan має уніфікований API для всіх платформ, оскільки інтегрує як десктопні, так і мобільні растеризатори. Він також має кращі інструменти інтеграції, ніж OpenGL, оскільки розробник може самостійно ввімкнути рівень діагностики та валідації. Все це робить десктопну та мобільну версії API схожими, що робить можливим портування ігор на різні платформи. Також, на відміну від OpenGL, Vulkan надає пріоритет мобільним платформам.

У OpenGL перевірка помилок є автоматичною функцією рівня драйверів. Як наслідок, програми, які працюють добре, можуть постраждати, якщо перевірка помилок не спрацьовує під час виконання. Надмірна та випадкова перевірка помилок також перевантажує комп'ютер. Тоді як у Vulkan перевірка помилок є надбудовою, яку можна вмикати та вимикати за потреби, увімкнувши або вимкнувши перевірку помилок та інші шари валідації. В ідеалі, для цілей налагодження, ці шари у Vulkan вмикаються на етапі розробки, а потім вимикаються на етапі релізу. Vulkan, як правило, є більш тонким API, ніж

OpenGL. Перевагами тонкого API є низькі накладні витрати та високий рівень контролю управління ресурсами. Таким чином, розробники на Vulkan можуть додавати фреймворки та допоміжні бібліотеки до прикладного рівня. Крім того, завдяки низьким накладним витратам, продукти, побудовані на ньому, втрачають дуже мало продуктивності та контролю над API. Фактично, на Vulkan можна побудувати OpenGL.

Основна перевага OpenGL над Vulkan полягає в тому, що він економить розробнику кілька годин, оскільки рівень драйверів виконує розподіл ресурсів за лаштунками. У Vulkan, однак, розробник повинен писати код для кожної передбачуваної дії, вибираючи тип пам'яті, розподіл обсягу пам'яті, потік і т.д. Як результат, Vulkan вимагає більше рядків коду для однієї і тієї ж програми, ніж OpenGL. Більшість людей використовують це як аргумент, чому машинне кодування того не варте.

Однак у випадках коли висока апаратна ефективність є необхідністю, додатковий час, витрачений на написання коду того вартий. Крім того, це потрібно робити лише один раз, а не для кожної програми через створення буферів команд. Також Vulkan пропонує багаторівневу екосистему, щоб розробники могли вибрати рівень програмування, з яким вони можуть впоратися. Експерти та ті, кому потрібна максимальна гнучкість і контроль, задовольняться використанням безпосередньо Vulkan. Ті, хто бажає пришвидшити етап розробки, обирають бібліотеки та шари. Більшість користувачів Vulkan належать до цієї категорії. І, нарешті, остання категорія - ті, хто використовує Vulkan для оптимізації вже готових ігрових рушіїв.

2.3 Основні кроки для відтворення зображення за допомогою Vulkan API

Крок 1 - Вибір екземпляра та фізичного пристрою. Програмне забезпечення розроблене за допомогою Vulkan починається з налаштування Vulkan API використовуючи VkInstance. VkInstance створюється за допомогою функції vkCreateInstance(). Ця функція використовує структуру

VkInstanceCreateInfo в яку записується інформація про необхідний функціонал. Можна вказати версію Vulkan, що використовується для реалізації програмного забезпечення; вказати необхідні розширення для Vulkan. Після створення VkInstance з’являється можливість виконувати запит на апаратне забезпечення комп’ютерної системи. Це дозволяє обрати один або декілька VkPhysicalDevice. В API Vulkan VkPhysicalDevice представляє фізичний графічний процесор або будь-який інший апаратний прискорювач, доступний в системі. Це абстракція фізичного пристрою, який додаток може використовувати для рендерингу. Об'єкт VkPhysicalDevice містить інформацію про властивості пристрою, такі як назва, тип, ідентифікатор виробника, ідентифікатор пристрою та версія драйвера. Він також надає інформацію про підтримувану версію API Vulkan, властивості пам'яті, підтримувані функції та розширення. Програми можуть перерахувати всі доступні фізичні пристрої в системі за допомогою функції vkEnumeratePhysicalDevices(). Функція повертає кількість доступних фізичних пристроїв і масив дескрипторів VkPhysicalDevice, які можна використовувати для подальшої обробки. Зазвичай програми вибирають відповідний фізичний пристрій на основі вимог програми, таких як бажані можливості рендерингу, підтримувані функції та розширення. Після вибору фізичного пристрою програма може створити логічний пристрій за допомогою функції vkCreateDevice(), щоб встановити з'єднання з фізичним пристроєм і виділити ресурси для рендерингу [11].

Крок 2 – Логічні пристрої (VkDevice) та черги. Після вибору потрібного апаратного пристрою потрібно створити VkDevice. У Vulkan API VkDevice представляє логічний пристрій, який керує доступом до фізичного пристрою (представленого VkPhysicalDevice) і надає інтерфейс для створення та керування об'єктами Vulkan, такими як буфери, зображення, конвеєри та командні буфери. Об'єкт VkDevice створюється програмою за допомогою функції vkCreateDevice() після вибору відповідного фізичного пристрою. Це основний об'єкт, з яким взаємодіє програма для виконання операцій рендерингу. Об'єкт VkDevice також

надає інформацію про властивості пристрою, такі як підтримувані функції, розширення, властивості пам'яті та черги. Програма може запитувати ці властивості за допомогою різних функцій vkGet*(). За допомогою об'єкта VkDevice програмне забезпечення може створювати та керувати різними об'єктами Vulkan, такими як буфери команд, буфери, зображення та конвеєри. Також при створені VkDevice необхідно вказати, які сімейства черг (VkQueue) будуть використувавутися. Більшість операцій, що виконуються у Vulkan, наприклад, команди малювання та операції з пам'яттю, виконуються асинхронно, шляхом розміщення їх у VkQueue. Черги розподіляються з сімейств черг (QueuesFamily), де кожне сімейство підтримує певний набір операцій. Наприклад, можуть існувати окремі сімейства для графічних, обчислювальних операцій та операцій передачі пам'яті. Наявність сімейств черг також може бути використано як визначальний фактор при виборі фізичного пристрою [11].

Крок 3 – створення VkSurface та VkSwapchainKHR. В API Vulkan VkSurface - це абстрактне представлення нативної поверхні платформи, такої як вікно або малюнок, що використовується як ціль рендерингу для додатків Vulkan. VkSurface створюється програмним забезпеченням за допомогою специфічних для платформи API або розширень, таких як Win32 API для Windows або XCB чи Xlib API для Linux. Після створення об'єкта VkSurface з’являється можливість отримати його властивості, зокрема розмір і формат, за допомогою функцій vkGetPhysicalDeviceSurfaceCapabilitiesKHR() і vkGetPhysicalDeviceSurfaceFormatsKHR(). Ці функції приймають об'єкти VkPhysicalDevice та VkSurface як вхідні параметри і повертають набір властивостей, які програма може використовувати для налаштування графічного конвеєра. VkSurface потім використовується програмним забезпеченням для створення VkSwapchain. VkSwapchainKHR - це набір зображень, які використовуються для рендерингу на поверхню, створену за допомогою VkSurface. Ланцюжок підкачки створюється за допомогою функції vkCreateSwapchainKHR(), яка приймає на вхід VkDevice, VkSurface і набір

параметрів конфігурації. Параметри конфігурації VkSwapchainKHR включають формат поверхні, колірний простір, розмір зображень, кількість зображень у VkSwapchainKHR, прапорці використання та режим представлення. Ці параметри використовуються для створення набору зображень, придатних для рендерингу на поверхню. Після створення VkSwapchainKHR програма може отримати зображення за допомогою функції vkGetSwapchainImagesKHR(). Ця функція приймає VkDevice та об'єкт VkSwapchainKHR як вхідні параметри і повертає масив об'єктів VkImage, які представляють зображення у своп-ланцюгу [11].

Крок 4 - VkImageView і VkFramebuffer. VkImageView - це подання об'єкта VkImage, яке описує, як зображення має інтерпретуватися графічним конвеєром. Зображення може мати декілька представлень, кожне з яких має різні налаштування, наприклад, різні формати або колірні простори, що дозволяє використовувати його в різних частинах конвеєра. VkImageView створюється за допомогою функції vkCreateImageView(), яка приймає VkDevice, структуру VkImageViewCreateInfo та вказівник на виділену ділянку пам'яті для зберігання створеного об'єкта VkImageView. Структура VkImageViewCreateInfo містить інформацію про те, як слід інтерпретувати зображення, включаючи посилання на саме зображення, формат зображення, аспект зображення для перегляду (колір, глибина, трафарет), діапазон зображення для перегляду та будь-які додаткові прапори, які можуть вплинути на зображення. VkImageView можна використовувати для прив'язки зображення до графічного конвеєра, створивши дескриптор VkImageView у шейдера. Дескриптор визначає тип ресурсу, номер зв'язування та етапи шейдеру, які матимуть доступ до ресурсу [11].

Крок 5 – VkRenderPass. VkRenderPass являє собою набір вкладень, підпроходів і залежностей, які описують схему операцій відтворення зображення. Це фундаментальна концепція, яка описує послідовність операцій відтворення зображення, що виконуються протягом одного кадру. Об'єкт VkRenderPass створюється за допомогою функції vkCreateRenderPass(), яка

отримує VkDevice, структуру VkRenderPassCreateInfo та вказівник на виділену ділянку пам'яті для зберігання створеного об'єкта VkRenderPass. Структура VkRenderPassCreateInfo описує вкладення, що використовуються під час проходу рендеру, підпроходи, які будуть використані для виконання операцій рендеру, та залежності між підпроходами. Вкладення представлені об'єктами VkAttachmentDescription, які описують формат, кількість вибірок та використання зображень, які будуть використані як вкладення під час проходу рендеру. Підпроходи (VkSubpass) представлені об'єктами VkSubpassDescription, які описують набір вкладень, що будуть використані під час операцій відтворення зображення, а також будь-які операції вводу, виводу або розв'язання, які будуть виконуватися над вкладеннями. Залежності представлені об'єктами VkSubpassDependency, які описують вимоги до впорядкування та синхронізації між підпроцесами. Вони використовуються для того, щоб гарантувати, що вихідні дані одного підпроцесу будуть доступні як вхідні для іншого підпроцесу. Після створення VkRenderPass може бути використаний для початку проходу рендерингу за допомогою функції vkCmdBeginRenderPass(). Під час проходу рендеру підпроходи виконуються у порядку, визначеному залежностями, причому кожен підпрохід використовує вкладення, визначені об'єктом VkRenderPass. Функція vkCmdNextSubpass() використовується для переходу між підпроходами, а функція vkCmdEndRenderPass() - для завершення проходу рендеру і звільнення пов'язаних з ним ресурсів [11].

Крор 6 – Графічний конвеєр. Графічний конвеєр у Vulkan налаштовується шляхом створення об'єкта VkPipeline. Він описує конфігурований стан відеокарти, наприклад, розмір VkViewport та роботу буфера глибини, а також стан, який можна запрограмувати за допомогою об'єктів VkShaderModule. Об'єкти VkShaderModule створюються з байтового коду шейдерів. Драйверу також потрібно знати, які цілі рендерингу будуть використовуватися у конвеєрі, що ми вказуємо, посилаючись на VkRenderPass [11].

Однією з найбільш характерних особливостей Vulkan у порівнянні з існуючими API є те, що майже всі конфігурації графічного конвеєра потрібно задавати заздалегідь. Це означає, що якщо необхідно перейти на інший шейдер або трохи змінити розташування вершин, то доведеться повністю перестворювати графічний конвеєр. Це означає, що необхідно заздалегідь створювати багато об'єктів VkPipeline для всіх можливих комбінацій, необхідних для операцій рендерингу. Динамічно можна змінювати лише деякі базові конфігурації, як розмір області перегляду та колір прозорість. Усі стани також мають бути описані явно, наприклад, не існує стану змішування кольорів за замовчуванням. Перевага полягає у тому, що оскільки еквівалент компіляції виконується заздалегідь, а не в реальному часі роботи програмного забезпечення, для драйвера існує більше можливостей для оптимізації, а продуктивність під час виконання є більш передбачуваною, оскільки великі зміни станів, такі як перемикання на інший графічний конвеєр, описано дуже явно [11].

Крок 7 – VkCommandPool та VkCommandBuffer. VkCommandPool - це об'єкт для керування пам'яттю, яка використовується для зберігання буферів команд. Він створюється за допомогою функції vkCreateCommandPool(), яка отримує VkDevice, структуру VkCommandPoolCreateInfo та вказівник на виділену ділянку пам'яті для зберігання створеного об'єкта VkCommandPool. VkCommandPool використовується для виділення та звільнення об'єктів VkCommandBuffer. VkCommandBuffer представляє послідовність команд, які можуть бути подані до черги на виконання. Об'єкти VkCommandBuffer створюються з пулу команд VkCommandPool за допомогою функції vkAllocateCommandBuffers() і знищуються за допомогою функції vkFreeCommandBuffers(). Об'єкти VkCommandBuffer записуються за допомогою функцій vkBeginCommandBuffer() та vkEndCommandBuffer(), які позначають початок та кінець процесу запису відповідно. Під час процесу запису команди додаються до буфера команд за допомогою різноманітних функцій, які відповідають різним командам API Vulkan, що можуть бути використані для

малювання геометрії, встановлення стану конвеєра та маніпулювання буферами і зображеннями. Після того, як VkCommandBuffer було записано, його можна відправити у чергу на виконання за допомогою функції vkQueueSubmit(). За один виклик vkQueueSubmit() можна відправити декілька об'єктів VkCommandBuffer, що дозволяє ефективно розподіляти роботу [11].

Крок 8 – Основний цикл. Команди необхідні для обчислення зображення записуються в командний буфер. Після чого описується основиний цикл програми. Для початку в циклу отримується наступний доступний VkImage, який може бути використаний для запису інформації отриманих з обчислень зображення. Це виконується за допомогою vkAcquireNextImageKHR(), що повертає VkImage з VkSwapchainKHR. Після рендеренгу зображення виконується функція vkQueuePresentKHR(), що повертає VkImage до VkSwapchainKHR та виконує вивід зображення на екран (VkSurface). Операції, які подається в чергу виконується асинхронно. Тому необхідно виконувати синхронізацію, щоб забезпечити правильний порядок виконання необхідних команд. Це виконується в основному циклі за допомогою VkSemaphore та VkFence [11].

2.4 Графічний конвеєр в Vulkan API

Графічний конвеєр - це фундаментальна концепція комп'ютерної графіки. На високому рівні графічний конвеєр можна уявити як низку етапів, які перетворюють вхідні дані на вихідні. У випадку рендерингу зображення вхідні дані зазвичай складаються з геометричної інформації (наприклад, вершин) та інформації про текстуру, тоді як вихідні дані складаються з кінцевого зображення, яке може бути відображене на екрані. Графічний конвеєр поділяється на дві основні стадії: стадія програми та стадія драйвера/апаратного забезпечення. На етапі програми програміст визначає, що має робити відеокарта. Зазвичай це передбачає створення низки команд рендерингу, які вказують відеокарті, як перетворити вхідні дані у бажані вихідні дані. Ці команди

рендерингу, як правило, організовані у серію викликів API. На етапі драйвер/апаратне забезпечення відеокарта фактично виконує операції рендерингу, визначені програмою. Це включає низку низькорівневих операцій, які оптимізовано для конкретного обладнання та драйвера, що використовується. Драйвер перетворює команди рендерингу, визначені програмою, у серію інструкцій графічного процесора, які можуть бути виконані відеокартою. Ці інструкції зазвичай виконуються паралельно багатьма різними процесорами відеокарти, що забезпечує високу продуктивність рендерингу [3].

Основні етапи графічного конвеєру зображено на рис. 2.1.

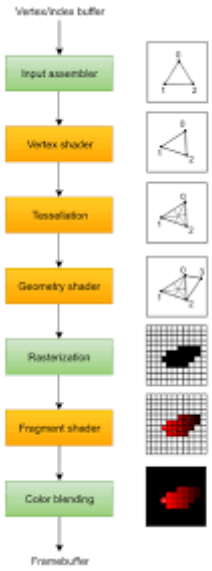

Рисунок 2.1- Стадії графічного конвеєру

Етап Input Assembly (Збірка Вхідних Даних) - це перший етап графічного конвеєра в API Vulkan. Цей етап приймає вхідні дані від центрального процесора і збирає їх у дані вершин для наступного етапу конвеєра. Вхідні дані зазвичай складаються зі списку вершин, які визначаються їхнім положенням у тривимірному просторі, а також іншими необов'язковими атрибутами, такими як колір, нормальний вектор або координати текстури. Вершини можуть бути організовані різними способами, наприклад, у вигляді списку, смуги або віяла, залежно від потреб програми. Після того, як вершини визначені, вони передаються вхідному асемблеру, який створює послідовність примітивів з

даних про вершини. Примітив - це базова геометрична фігура, така як точка, лінія або трикутник. Вхідний асемблер створює ці примітиви, комбінуючи вершини відповідно до порядку і топології, визначеної програмою. Результатом роботи вхідного асемблера є послідовність примітивів, які передаються на етап вершинного шейдера. Вершинний шейдер відповідає за обробку кожної вершини в послідовності, застосовуючи будь-які перетворення або обчислення, необхідні для процесу рендерингу. Загалом, етап Input Assembly є критично важливим першим кроком у графічному конвеєрі, оскільки він приймає сирі вхідні дані і готує їх для подальшої обробки графічним процесором. Організовуючи вершини у примітиви, вхідний асемблер гарантує, що графічний процесор може ефективно обробляти дані паралельно, що є ключем до досягнення високої продуктивності в сучасних графічних додатках [3].

Етап Vertex Shader (Вершинний Шейдер) - це другий етап графічного конвеєра в API Vulkan, наступний за етапом Збірки вхідних даних. Етап Вершинний Шейдер приймає послідовність примітивів, створену на етапі Input Assembly, і обробляє кожну вершину в послідовності окремо. Вершинний шейдер - це програмований блок, який працює на графічному процесорі і відповідає за застосування перетворень та обчислень до кожної вершини в послідовності. Вершинний Шейдер отримує на вхід одну вершину з послідовності разом з будь-якими додатковими даними, які програма пов'язала з цією вершиною, такими як координати текстури, колір або нормальні вектори. Потім Вершинний Шейдер застосовує до вершини одне або декілька перетворень, таких як масштабування, обертання або переведення, а також будь-які обчислення або операції, необхідні для процесу рендерингу. Результатом етапу Вершинного Шейдера є нова послідовність вершин, які були перетворені та оброблені відповідно до специфікацій програми. Ці перетворені дані про вершини передаються на наступний етап графічного конвеєра для подальшої обробки. Загалом, етап Vertex Shader є критично важливим компонентом графічного конвеєра, оскільки він дозволяє застосовувати складні та динамічні

перетворення до кожної вершини в послідовності. Завдяки запуску Vertex Shader на GPU, а не на CPU, графічний конвеєр може досягти високої продуктивності та ефективності, що дозволяє рендерити складні 3D-сцени в реальному часі в сучасних графічних додатках [3].

Етап Tessellation Shader (Тесселяція) - це необов'язковий етап у графічному конвеєрі Vulkan API, який слідує за етапом Vertex Shader. Основною метою етапу Tessellation Shader є генерація нових вершин і примітивів на основі вхідної геометрії, що може покращити деталізацію і складність зображення, яке рендериться. Тесселяція передбачає розбиття більшого примітиву, наприклад, трикутника або квадрата, на менші примітиви (рис. 1.1).Цей процес можна використовувати для деталізації поверхні, розбиваючи її на менші трикутники або квадрати, або для створення складнішої геометрії, наприклад, криволінійних поверхонь або неправильних форм. Етап Тесселяції складається з двох програмованих блоків: Hull (Корпусного) і Domain (Доменного) шейдерів. Hull Shader відповідає за створення нових примітивів і визначення їхнього розміру та форми, тоді як Domain Shader відповідає за визначення положення кожної нової вершини в межах створених примітивів. Вхідними даними для етапу Tessellation Shader є послідовність вершин, згенерованих на етапі Vertex Shader, разом з будь-якими додатковими даними, пов'язаними з цими вершинами. На виході етапу Tessellation Shader виходить нова послідовність вершин, яка може бути створена шляхом розбиття вхідної геометрії на менші примітиви. Загалом, етап Теселяція можна використовувати для додавання деталей і складності зображенню, що дає змогу реалістичніше і динамічніше відтворювати складні 3D-сцени. Однак, оскільки вона передбачає створення нової геометрії на льоту, теселяція може бути дорогою в обчислювальному плані і може вимагати додаткової апаратної підтримки для досягнення оптимальної продуктивності [3].

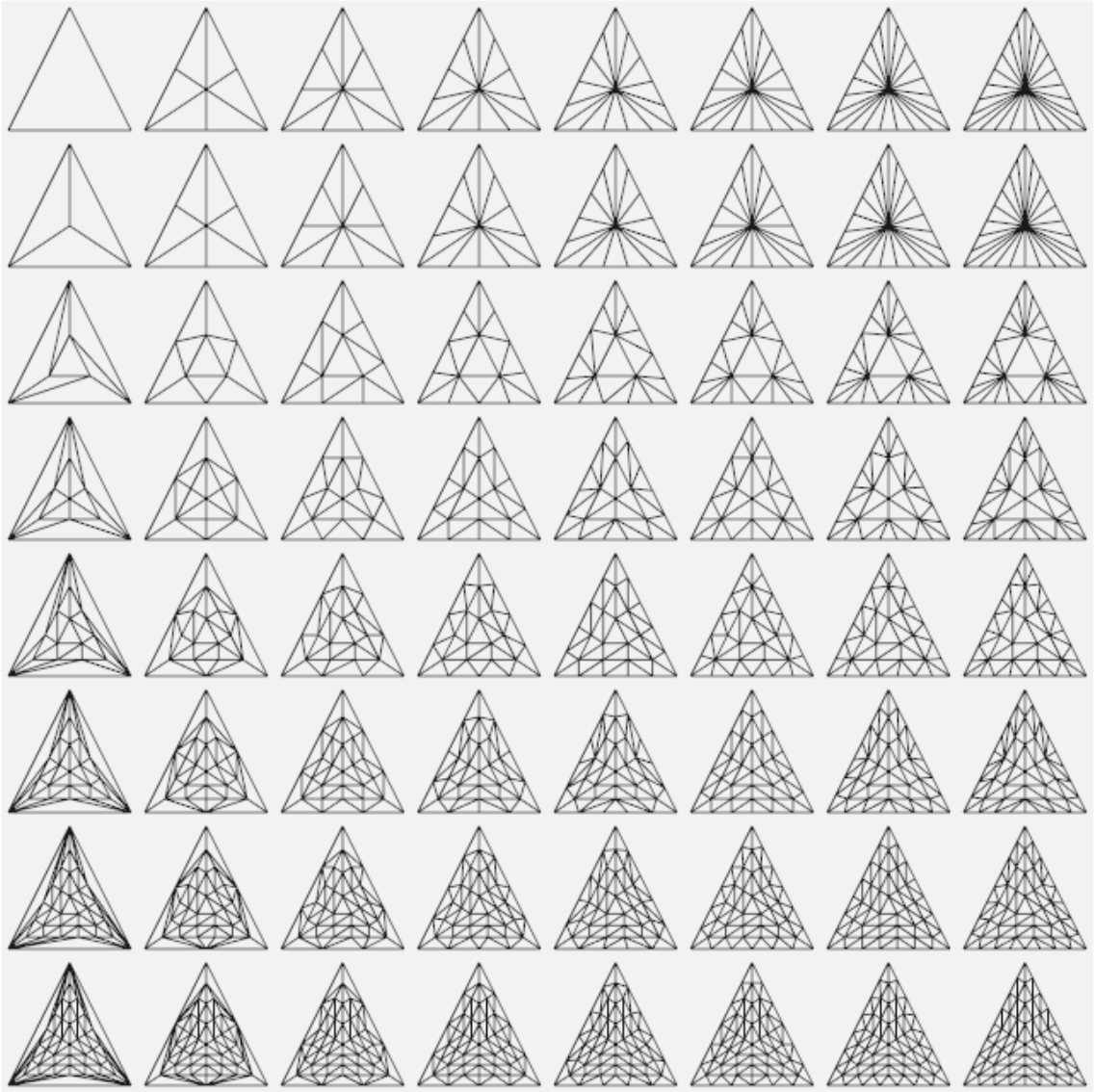

Рисунок 2.2 - приклад роботи тесселяції

Етап Geometry Shader (Геометричний Шейдер) - це ще один необов'язковий етап у графічному конвеєрі API Vulkan, який слідує за етапом Tessellation Shader, якщо він присутній. Основною метою етапу Geometry Shader є обробка примітивних збірок, згенерованих етапом вхідної збірки, і виведення нових примітивних збірок (рис. 2.2). На вхід етапу Geometry Shader подається збірка примітивів, яка може складатися з точок, ліній, трикутників або інших типів примітивів. Потім геометричний шейдер може виводити нові збірки примітивів різних типів або навіть повністю відкидати певні примітиви на основі певних умов, визначених користувачем. Цей етап бути використаний для різноманітних завдань, таких як генерація частинок, виконання процедурно

згенерованої анімації або реалізація алгоритмів виявлення зіткнень. Він також може бути використаний для створення додаткової геометрії для тесселяційних примітивів або виконання інших складних ефектів. Як і інші програмовані етапи шейдерів, Геометричний Шейдер забезпечує високий ступінь кастомізації та гнучкості у рендерингу графіки. Однак, він також може бути дорогим в обчислювальному плані і вимагає ретельної оптимізації для досягнення оптимальної продуктивності. Варто зазначити, що не всі відеокарти підтримують сцену Geometry Shader, і її доступність може залежати від конкретного обладнання та конфігурації драйверів [3].

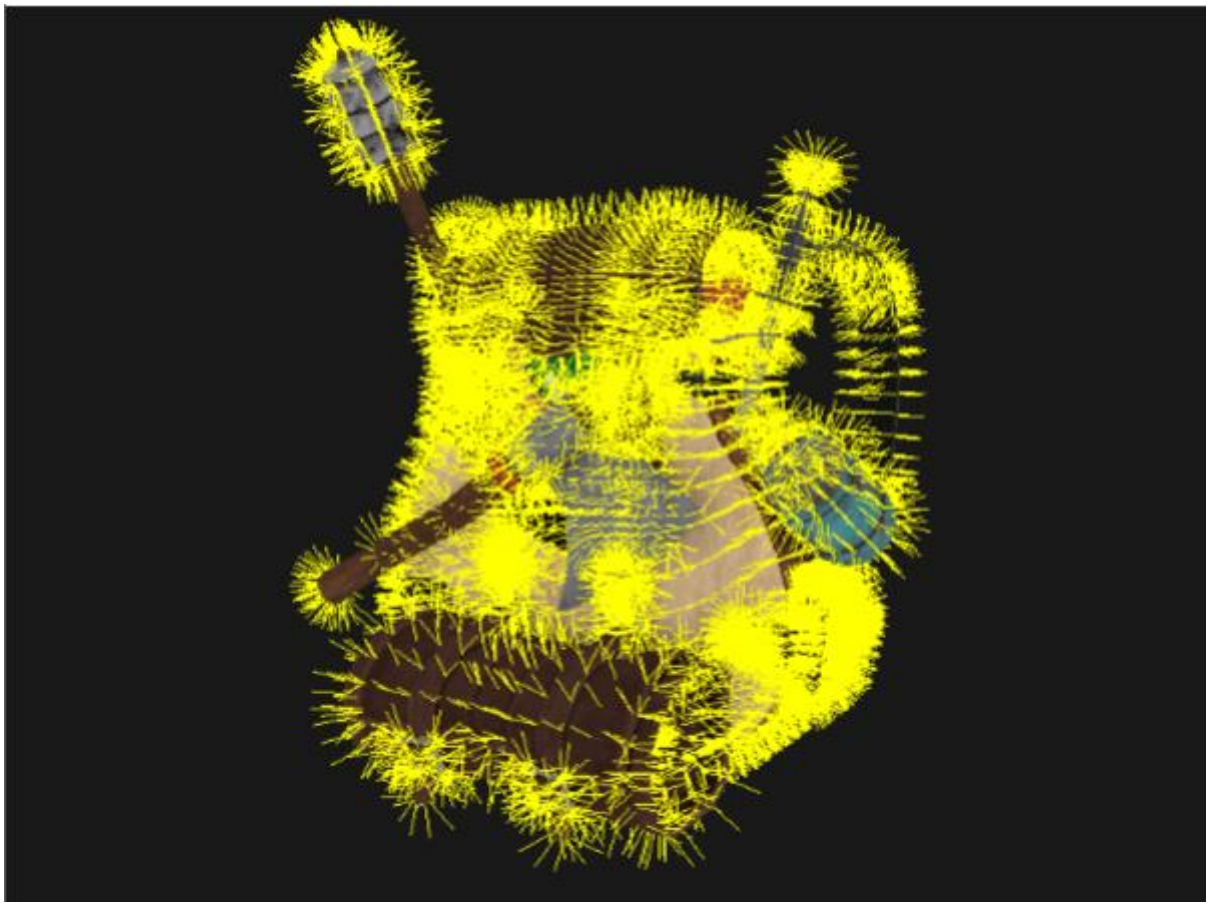

Рисунок 2.2 – використання шейдеру геометріх для відображення нормалей об’єкту

Етап Rasterization (Растеризації) є критично важливою частиною конвеєра обробки графіки в API Vulkan, який слідує за етапами вершинного шейдера і геометричного шейдера. Його основне завдання - взяти оброблені вершини з

попередніх етапів і перетворити їх у растрове зображення, яке можна вивести на екран (рис. 2.3). На етапі Растеризації вершини перетворюються на фрагменти, які по суті є елементами розміром з піксель, що представляють невелику частину кінцевого зображення. Кожен фрагмент обробляється растеризатором, який визначає, чи буде він видимим на екрані, виконуючи низку тестів, таких як відсіювання, відсікання та перевірка глибини. Якщо фрагмент проходить всі тести, він надсилається на етап Fragment Shader, де на основі вхідних даних шейдерів визначається його колірне значення. Цей процес повторюється для кожного фрагмента зображення, в результаті чого на екран виводиться фінальне зображення. Етап растеризації відіграє вирішальну роль у визначенні візуальної якості та продуктивності графічної програми. Важливо ретельно налаштувати растеризатор, щоб збалансувати компроміси між візуальною точністю та продуктивністю. В API Vulkan розробники мають високий ступінь контролю над етапом растеризації за допомогою різних налаштувань і опцій, таких як ввімкнення/вимкнення тестування глибини, вказівка розміру вікна перегляду і встановлення рівня мультидискретизації. Така гнучкість дозволяє розробникам оптимізувати свої програми для широкого спектру апаратних конфігурацій і роздільних здатностей дисплеїв [3].

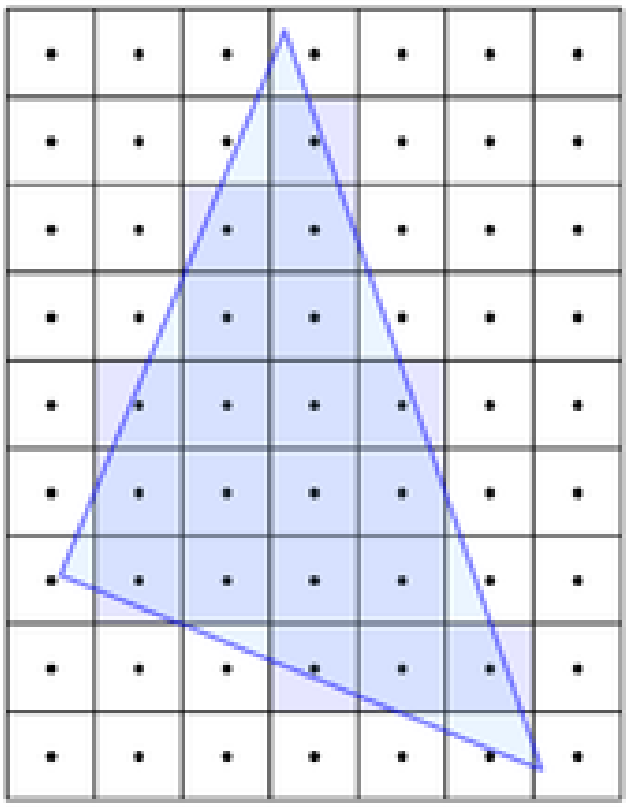

Рисунок 2.3 – результат єтапу растеризації

Етап Fragment Shader (Фрагментний шейдер) - це наступний етап конвеєра обробки графіки в API Vulkan після етапу Растеризація. Основною функцією цього етапу є визначення остаточного кольору кожного пікселя або фрагмента, який пройшов етап растеризації, на основі його вхідних даних (рис. 2.4). На цьому етапі вхідні дані для кожного фрагмента інтерполюються з даних вершин, які були перетворені на попередніх етапах. Фрагментний шейдер приймає ці інтерпольовані дані як вхідні і виконує різні обчислення для визначення остаточного значення кольору фрагмента. Цей шейдер також може виконувати інші операції, такі як накладання текстур, освітлення та прозорість. Фрагментний шейдер добре програмується, що дозволяє розробникам писати власні шейдери, які реалізують широкий спектр візуальних ефектів. Важливо зазначити, що

продуктивність фрагментного шейдера може мати значний вплив на загальну продуктивність графічної програми. Тому оптимізація шейдера фрагментів є критично важливим завданням для досягнення високої частоти кадрів і плавності ігрового процесу. В API Vulkan розробники мають тонкий контроль над етапом фрагментного шейдера за допомогою різних налаштувань і опцій, таких як встановлення режиму змішування, вказівка поведінки тестування глибини і контроль рівня згладжування. Такий рівень контролю дозволяє розробникам налаштовувати свої графічні додатки для широкого спектру апаратних конфігурацій і роздільних здатностей дисплеїв [3].

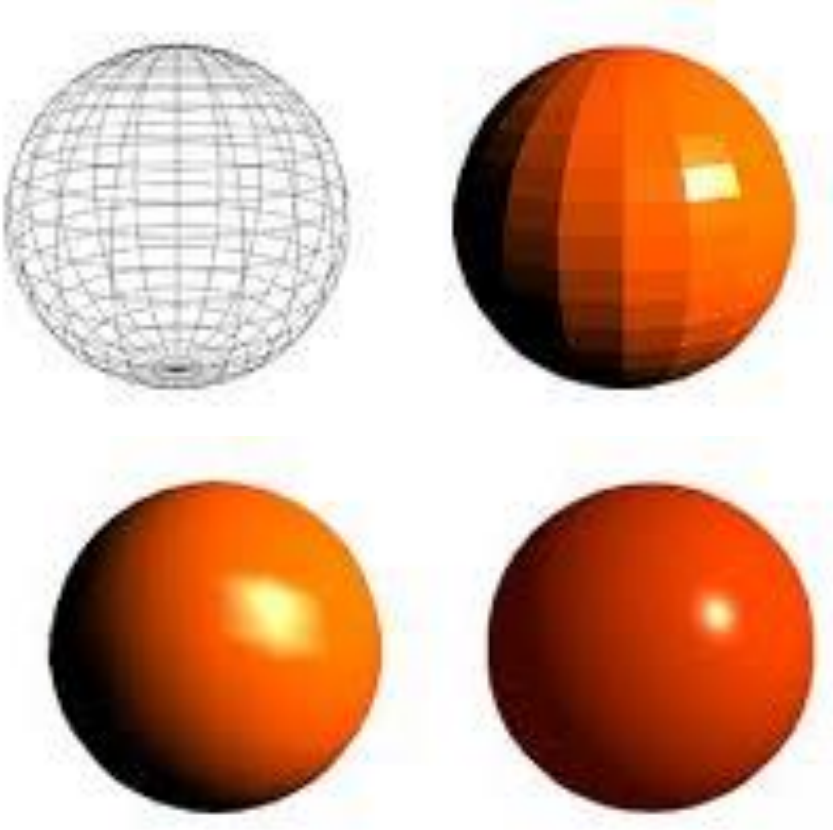

Рисунок 2.4 – результати обрахунку освітлення у вершиному шейдері та у фрагментному шейдері (знизу зліва та знизу справа відповідно)

Етап Color Blending (змішування кольорів) є завершальним у графічному конвеєрі API Vulkan. Його основна функція полягає у змішуванні вихідних

значень кольорів (рис. 2.5), згенерованих фрагментним шейдером, зі значеннями кольорів, вже наявними у фреймбуфері, для отримання остаточного зображення, яке буде відображено на екрані. Етап змішування кольорів працює на попіксельній основі, беручи вихідні значення кольорів, згенеровані шейдером фрагментів, і застосовуючи набір операцій, які визначають, як ці значення поєднуються з наявними значеннями кольорів у буфері кадрів. Ці операції можуть включати змішування кольорів на основі різних факторів, таких як прозорість, або виконання логічних операцій, таких як побітове І, АБО та XOR. В API Vulkan розробники мають тонкий контроль над операціями змішування завдяки використанню набору рівнянь змішування і коефіцієнтів змішування. Ці рівняння і коефіцієнти можуть бути використані для вказівки того, як повинні поєднуватися значення кольорів, і можуть бути використані для створення широкого спектру візуальних ефектів, таких як прозорість, згладжування і корекція кольору. Важливо зазначити, що продуктивність етапу змішування кольорів може мати значний вплив на загальну продуктивність графічної програми. Тому оптимізація операцій змішування є критично важливим завданням для досягнення високої частоти кадрів і плавного ігрового процесу. Розробники можуть оптимізувати операції змішування за допомогою таких методів, як зменшення кількості операцій змішування або використання спрощених рівнянь змішування, які вимагають менше обчислень [3].

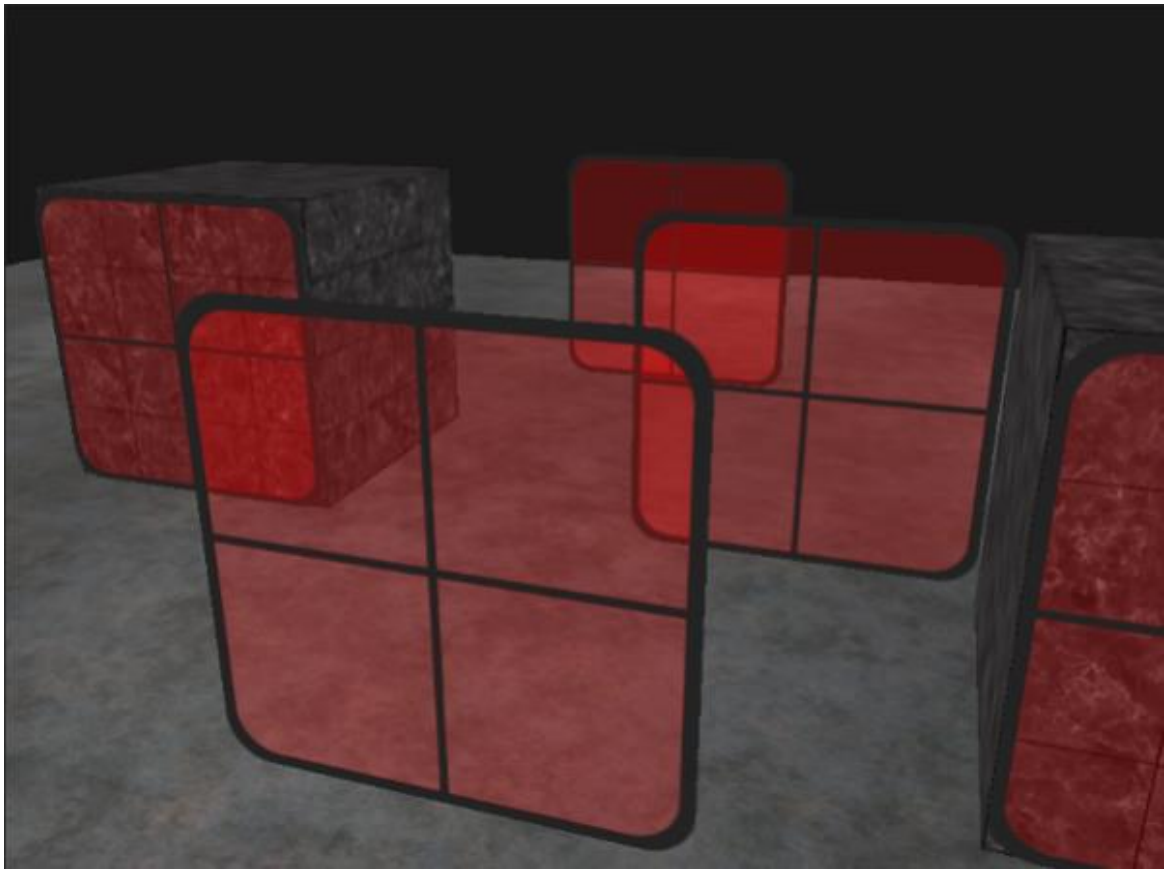

Рисунок 2.5 – Використання єтапу змішування кольорів для відображення прозорих об’єктів

2.5 Subpass в Vulkan API

У Vulkan API Subpass - це частина RenderPass, яка представляє собою автономну операцію рендерингу. RenderPass може мати один або декілька Subpass, кожен з яких визначає набір вхідних, вихідних і проміжних вкладень, а також операції рендерингу, які будуть виконуватися над цими вкладеннями. Основна мета Subpass - оптимізувати процес рендерингу, дозволяючи драйверу та апаратному забезпеченню оптимізувати схеми доступу до пам'яті та зменшити передачу даних між об'єктами рендерингу. Групуючи операції рендерингу в підпроцеси, Vulkan може краще керувати залежностями між різними частинами процесу рендерингу, що може призвести до покращення продуктивності та зменшення використання пам'яті [3]. Subpass визначає наступні властивості:

- Input Attachments: слугують вхідними даними для підпроцесу.

- Color Attachments: слугують вихідними даними для підпроцесу, що містять дані про колір.
- Depth-Stencil Attachment: слугує виходом для підпроцесу, що містить дані про глибину та/або трафарет.
- Preserve Attachments: VkImageView, який не використовується як вхідний або вихідний, але вміст якого потрібно зберегти для наступних підходів.
- Subpass Dependencies: Описує залежності між різними Subpass.

Subpass є унікальною особливістю API Vulkan і надають потужний інструмент для оптимізації продуктивності рендерингу та підвищення ефективності використання пам'яті.

Subpass може бути використаний для реалізації методу відкладеного рендерингу.

Відкладенний рендеринг базується на відкладеному обчислені складних процесів відображенння об’єктів, наприклад, обчислення світла. Цей мотод складається з двох проходів: обчислення геометрії сцени та обчислення освітлення сцени. В першому обході отримутється інформації про розташування об’єктів на сцені, колір об’єктів, нормальні вектори вершин об’єктів, тощо [5]. Вся ця інформація записується в буфер, який називається G-буфер (рис. 2.6).

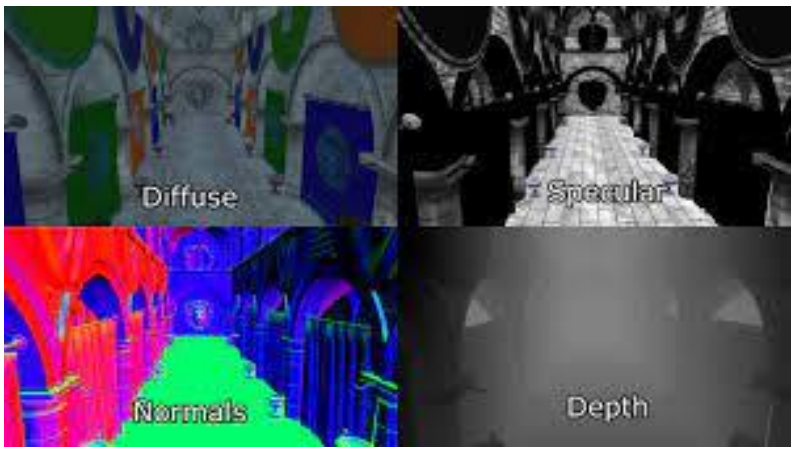

Рисунок 2.6 – приклад інформації записаної в G-буфер на першому проході відкладеного рендерингу

Інформація отримана під час обчислень в першому проході використовується при обчислені освітлення об’єктів на сцені під час другого проходу.

Основною перевагою цього підходу є те, що обчислення освітлення відбувається лише на тих фрагментах, які будуть відображатися на екрані. Усі інші фрагменти були відкинуті під час першого проходу, завдяки тестування глибини, тощо. Це зменшує кількість обчислень необхідних для вібдображення схеми [7].

Недоліком такого методу є необхідність збереження великої кількості інформації в G-буфері для подальшого обчислення під час другого проходу. Також з таким підходом до відображення сцени не є можливим використання змішування фрагментів. Як і не так просто реалізувати MSAA.

Оскільки відкладений метод передбачає кілька проходів, то використовуючи Subpass, проміжні дані можна зберігати в пам'яті і передавати безпосередньо між проходами без необхідності записувати і зчитувати інформацію в пам’яті вишого рівня графічного процесору (рис. 2.7). Це може значно оптимізувати використання пам'яті графічного процесору та підвищити ефективність роботи програмного забезпечення для комп’ютерних систем.

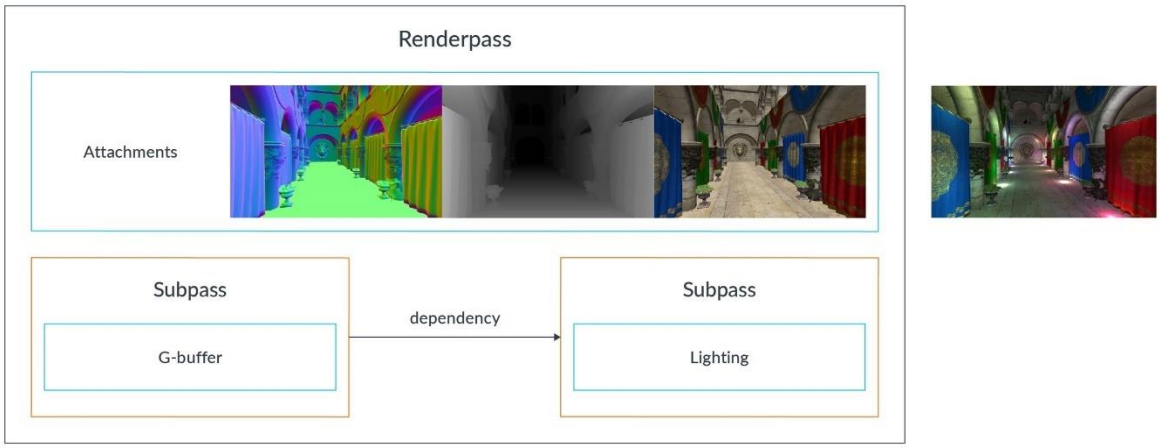

Рисунок 2.7 – реалізація Відкладеного Рендеренгу на основі VkSubpass

Також Subpass може бути використаний для реалізації ефектів постобробки, таких як глибина різкості зображення, фільрація зображення, розмиття, корекції кольору зображення, тощо. Ці ефекти виконуються в кінці обчислення результуючого зображення сцени, коли вже отримано усю інформації про об’єкти та освітлення на заданій сцені. Кожен ефект може бути реалізовано як окремий підпрохід з власними вхідними та вихідними вкладеннями, що дозволяє драйверу оптимізувати схеми доступу до пам'яті та мінімізувати передачу даних між проходами.

Tile-Based рендеринг - це техніка, що використовується в мобільних графічних процесорах для підвищення продуктивності та зниження енергоспоживання. Використовуючи підпроходи для визначення операцій рендерингу в межах кожного такого тайлу, Vulkan може оптимізувати шаблони доступу до пам'яті і зменшити обсяг даних, що передаються між GPU і CPU, в результаті чого підвищується продуктивність і знижується енергоспоживання.

У деяких випадках може знадобитися виконати рендеринг до декількох вихідних буферів одночасно. Subpass можна використовувати для визначення вкладень для кожного вихідного буфера та операцій рендерингу, які потрібно виконати для кожного буфера. Це може покращити продуктивність, дозволяючи драйверу оптимізувати схеми доступу до пам'яті та зменшити передачу даних між буферами.

2.6 Засоби синхронізаціх в Vulkan API

API Vulkan надає кілька інструментів синхронізації для забезпечення правильного впорядкування команд і даних між CPU і GPU, а також між різними частинами графічного конвеєра.

VkSemaphore (Семафори) використовуються для сигналізації подій між різними частинами графічного конвеєра, наприклад, між CPU і GPU, або між різними буферами команд. Вони легкі та ефективні і можуть використовуватися для керування порядком виконання різних операцій [3].

VkFence (Огорожі) використовуються для сигналізації про завершення виконання певного набору команд на графічному процесорі. Їх можна використовувати для синхронізації доступу до ресурсів між CPU і GPU або між декількома буферами команд. Загородження сигналізує графічному процесору, коли передача або програма завершена за допомогою vkSignalFence. Програма може очікувати на сигнал огорожі за допомогою vkWaitForFences, яка блокує виконання, доки не буде подано сигнал або не настане таймаут. Стан огорожі також може бути скинутий програмою за допомогою vkResetFences, який встановлює огорожу у стан «без сигналу» [3].

Events (Події) схожі на семафори, але можуть використовуватися для сигналізації про певні моменти у виконанні буфера команд. Їх можна використовувати для керування порядком виконання різних частин буфера команд або для синхронізації доступу до ресурсів між різними частинами конвеєра. Події можна встановити за допомогою команд vkCmdSetEvent або vkSetEvent, а скинути - за допомогою команд vkCmdResetEvent або vkResetEvent. На події можна чекати за допомогою таких команд, як vkCmdWaitEvents або vkWaitForFences. Події корисні для координації складних операцій, які включають декілька черг, наприклад, передачу даних між графічними та обчислювальними чергами, або реалізацію умовного рендерингу на основі певних критеріїв. Події також можна використовувати для реалізації тонкої синхронізації між операціями хоста і пристрою, наприклад, для оновлення буфера або зображення. Щоб використовувати події, вам потрібно створити їх за допомогою vkCreateEvent, вказати маски етапів, які вказують, коли подія встановлюється або очікується, і знищити їх за допомогою vkDestroyEvent, коли вони більше не потрібні. Ви також повинні переконатися, що події є видимими для черг, які їх використовують, використовуючи бар'єри пам'яті або бар'єри конвеєра [3].

Pipeline Barriers (Бар'єри конвеєра) використовуються для синхронізації доступу до ресурсів між різними частинами графічного конвеєра. Вони можуть

бути використані для забезпечення дотримання залежностей даних між різними етапами конвеєра або для запобігання гонки при доступі до спільних ресурсів [3].

RenderPass Dependencies (Залежності RenderPass) використовуються для синхронізації доступу до ресурсів між різними Subpass в одному RenderPass. Вони можуть бути використані для забезпечення дотримання залежностей даних між різними підпроходами або для запобігання перевантажень при доступі до спільних ресурсів. Залежність RenderPass складається з чотирьох елементів: підпрохід джерела, підпрохід призначення, маска сцени джерела та маска сцени призначення. Підпроходи джерела та призначення визначають, які підпроходи беруть участь у залежності. Маска вихідного етапу визначає, які етапи конвеєра вихідного підпроцесу повинні завершитися до того, як залежність буде задоволена. Маска етапу призначення вказує, які етапи конвеєра підпроцесу призначення повинні дочекатися виконання залежності, перш ніж вони зможуть розпочати роботу [3]. Залежність RenderPass можна використовувати для вираження різних типів залежностей, наприклад:

- Залежність між двома підпроцесами, які використовують один і той самий вкладення з різними макетами. Наприклад, підпроцес, який записує до вкладення як кольорове вкладення, і інший підпроцес, який читає з нього як вхідне вкладення.
- Залежність між підходом і зовнішнім буфером команд, який працює з тим самим вкладенням. Наприклад, підпроцес, який записує до вкладення як кольоровий додаток, і зовнішній буфер команд, який копіює його до іншого зображення.
- Залежність між двома підпроцесами, які використовують різні вкладення, але повинні синхронізувати їх виконання. Наприклад, підпроцес, який виконує тестування глибини, та інший підпроцес, який виконує тестування трафарету, використовують різні вкладення, але повинні гарантувати, що вони не перетинаються.

Усі ці інструменти синхронізації надають гнучкий і потужний спосіб контролювати виконання команд і забезпечують правильне впорядкування даних і операцій в API Vulkan

2.6.1 Використання VkSemaphores

У Vulkan API семафор - це примітив синхронізації, який можна використовувати для координації виконання декількох завдань у чергах команд. Семафор може перебувати в одному з двох станів: з сигналом або без сигналу. VkSemaphore - це дескриптор об'єкта семафора, який можна створити за допомогою функції vkCreateSemaphore. Об'єкт VkSemaphore представляє точку у виконанні буферу команд, де певне завдання завершилося. Семафори зазвичай використовуються для синхронізації виконання декількох буферів команд або для координації отримання та звільнення ресурсів, таких як зображення свопчейнів. Для сигналізації семафора можна використати функцію vkQueueSubmit, яка надсилає буфер команд до черги команд з об'єктом VkSemaphore, вказаним у полі pSignalSemaphores структури VkSubmitInfo. Щоб дочекатися сигналу семафора, функція vkQueueSubmit може бути використана для подання командного буфера з об'єктом VkSemaphore, вказаним у полі pWaitSemaphores структури VkSubmitInfo. Семафори також можна використовувати разом з огорожами для синхронізації виконання буферів команд у декількох потоках або процесах [11].

Семафор може бути використаним для того, щоб гарантувати, що зображення з VkSwapchainKHR не читається і не записується під час його презентації. Семафор може сигналізувати про завершення рендерингу і готовність зображення до виводу на екран.

Коли декілька буферів команд потрібно виконати у певному порядку, семафори можна використовувати для забезпечення необхідної синхронізації. Наприклад, якщо один буфер команд записує в ресурс, який потім зчитується

іншим буфером команд, семафор може бути використаний, щоб гарантувати, що другий буфер команд не буде виконуватися, поки не завершиться перший.

Семафори можна використовувати для синхронізації виконання операцій CPU та GPU. Наприклад, семафор може бути використаний для того, щоб гарантувати, що буфер не буде записано в CPU, поки він читається GPU.

Для того, щоб переконатися, що команди для певного кадру виконуються у правильному порядку, можна використовувати семафор, який сигналізує про завершення попереднього кадру. Це може бути використано для реалізації потрійної буферизації, де використовуються три кадри, щоб гарантувати, що GPU завжди має кадр для роботи, поки CPU готує наступний.

2.6.2 Використання VkFences

VkFence (Огорожа)- це об'єкт синхронізації, який допомагає програмному забезпеченю дізнатися, коли поданий буфер команд завершив виконання. Він використовується для координації між CPU і GPU або між декількома потоками, які отримують доступ до одного і того ж ресурсу.

Огорожа може перебувати в одному з двох станів: сигналізована або несигналізована. Коли огорожа створюється, вона спочатку перебуває у несигналізованому стані. Потім програма може приєднати огорожу до буфера команд за допомогою функції vkQueueSubmit(). Як тільки буфер команд завершить виконання, огорожа буде сигналізована. Програма може використовувати функцію vkWaitForFences() для очікування сигналу про появу огорожі. Ця функція блокує потік, що викликає, доки не буде сигналізовано вказану огорожу. Це корисно, коли програмі потрібно зачекати, поки графічний процесор завершить виконання певного набору команд, перш ніж продовжити роботу. Альтернативно, функція vkGetFenceStatus() може бути використана для запиту стану огорожі без блокування. Ця функція повертає поточний стан огорожі, з сигналом або без сигналу. Це корисно, коли програма хоче періодично перевіряти стан огорожі, не блокуючи основний потік. Огорожі зазвичай

використовуються у поєднанні з буферами команд і чергами для синхронізації доступу до ресурсів, таких як буфери і зображення. Вони також корисні для реалізації багатопотокового рендерингу, коли різні потоки подають буфери команд до однієї черги і повинні координувати свій доступ до спільних ресурсів [11].

Висновки до розділу

Vulkan – це сучасний API для взаємодії з графічним процесором, основною перевагою якого є низькорівневість. Це надає інженерам можливість будувати більш оптимізоване під конкретні задачі, в яких швидкість виконання є першим пріоритетом, програмне забезпечення для комп'ютерних систем.

До появи Vulkan більшість операцій були реалізовані на стороні драйверу графічного процесору, що не давало інженерам чіткого розуміння деталей роботи розробленого програмного забезпечення. Це приводило до неоптимальних рішень, при реалізації.

За допомогою використання засобів Vulkan API можна реалізувати та оптимізувати метод візуалізації програмного забезпечення. VkSubpass, який є унікальним для Vulkan, може пришвидшити роботу програмного забезпечення, оскільки оптимізовує використання ресурсів у ситуації з декількома проходами для обчиселння кінцевого зображення. Це може бути використано в Loop32-методі описаному у розділі 1.

3. ОПТИМІЗАЦІЇ LOOP32-МЕТОДУ ЗА ДОПОМОГОЮ ЗАСОБІВ VULKAN API

3.1 Покращення алгоритму Loop32 за допомогою оптимального використання VkDescriptorSet та VkBuffer

Управління VkDescriptorPool та VkDescriptorSet має одне з вирішальних значень при розробці програмного забезпечення з використанням Vulkan API.

VkDescriptorSet - це потужний інструмент, який дозволяє передавати інформацію на графічний процесор для рендерингу динамічних об'єктів і матеріалів, що використовують текстури. Однак у складних програмах кількість VkDescriptorSets може бути значною, і вони можуть часто змінюватися залежно від об'єктів і текстур у сцені. Для вирішення цієї проблеми, можна створити кілька пулів VkDescriptorPools для кожного кадру. Однак такий підхід може спричинити перевантаження центрального процесора через часті виклики таких функцій, як vkResetDescriptorPool, vkAllocateDescriptorSets та vkUpdateDescriptorSets. Ці виклики, навіть, можуть займати більше часу, ніж сам розрахунок кадру [2, 3, 4].

Для оптимізації використання VkDescriptorSet можна реалізувати систему кешування, яка повторно використовує вже створені VkDescriptorSets. Один із способів зробити це - використовувати хеш-таблицю з буферами та зображеннями як ключами. Цей метод значно покращує продуктивність, зменшуючи час, необхідний для обчислення кадру. Однак, оскільки VkDescriptorPool не очищується з кожним кадром, необхідно реалізувати систему, яка відстежує невикористані набори дескрипторів і видаляє їх. Такий підхід зменшив час необхідний для обчислення фінального кадру з 11.96 мс до 10.84 мс. Тобто покращення швидкодії алгоритму становить майже 10%.

Інший підхід полягає в тому, щоб правильно керувати буферами, записуючи інформацію про всі схожі об'єкти в один буфер замість того, щоб створювати буфер для кожного об'єкта [2, 4]. Це зменшує кількість VkDescriptorSets, а отже і викликів vkUpdateDescriptorSets та vkAllocateDescriptorSets, що додатково оптимізує продуктивність. Як показало

тестування, поєднання кешування і керування буферами може бути корисним для великих, складних сцен. Використання такого буферу у випадку з візуалізацією прозорих об’єктів покращує швидкодію алгоритму на 7%. Час обчислення кадру став 11.09 мс, замість 11.96 мс.

3.2 Кешування графічного конвеєру

Vulkan API має функцію збереження схеми графічного конвеєра, яку можна використовувати повторно, щоб прискорити створення подібних графічних конвеєрів у майбутньому. Це може значно підвищити швидкість роботи програмного забезпечення.

При створенні графічного конвеєра у Vulkan, першим кроком є компіляція шейдерів (VkShaderModule). Цей процес може зайняти багато часу, особливо при генерації зображень в реальному часі. Для прискорення створення графічного конвеєра можна використовувати кешування. Таким чином зменшується час, необхідний для побудови кадру. Vulkan має вбудований об'єкт VkPipelineCache, який можна використовувати під час створення графічного конвеєра, наприклад, за допомогою vkCreateGraphicsPipelines. Цей об'єкт представляє із себе хешований контейнер, який зберігає необхідну інформацію про графічний конвеєр. Крім того, Vulkan дозволяє отримати двійкову інформацію об'єкта VkPipelineCache і зберегти її у файл на диску, щоб використовувати у наступних запусках програми.

Тестування використання кешування та збереження графічного конвеєру показали, що використання така техніка може скоротити час створення конвеєра приблизно вдвічі. Коли кешування конвеєр не використовується, час, необхідний для створення конвеєра, становить 50.4 мс. Однак, при використанні кешування, цей час значно скорочується до 24,7 мс. Таким чином, кешування графічного конвеєру є важливою функцією Vulkan, яка може значно підвищити ефективність програмного забезпечення [2, 3].

3.3 Оптимізація за допомогою використання засобу VkSubpass

Одним з унікальних інструментів Vulkan є VkSubpass. Ця функція дозволяє розділити VkRenderPass на окремі логічні частини, що дозволяє оптимізувати роботу графічного процесора. Використовуючи підпроходи замість проходів рендерингу, графічний процесор може виконувати обчислення більш ефективно, що призводить до покращення продуктивності програмного забезпечення.

Loop32-метод виконує два проходи для обчислення кінцевого результату. На першому виконується сортування фрагментів за значенням глиби. На другому – обчислюється значення кінцевого кольору пікселя на результуючому зображені.

Необхідність у двох проходах є недоліком даного методу, але Vulkan надає можливість оптимізувати роботу алгоритму за допомогою використання VkSubpass.

Нижче наведено частину програмного коду, яка відповідає за створення VkRenderPass з двома Subpass для реалізації Loop32 методу.

```
VkAttachmentDescription colorAttachment = {};
colorAttachment.format = VK_FORMAT_R8G8B8A8_UNORM;
colorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
colorAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
colorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
colorAttachment.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;

VkAttachmentReference colorAttachmentRef = {};
colorAttachmentRef.attachment = 0;
colorAttachmentRef.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

VkSubpassDescription subpass1 = {};
subpass1.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
subpass1.colorAttachmentCount = 1;
subpass1.pColorAttachments = &colorAttachmentRef;

VkSubpassDescription subpass2 = {};
subpass2.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
subpass2.colorAttachmentCount = 1;
subpass2.pColorAttachments = &colorAttachmentRef;
subpass2.inputAttachmentCount = 1;
subpass2.pInputAttachments = &colorAttachmentRef;

VkRenderPassCreateInfo renderPassInfo = {};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
renderPassInfo.attachmentCount = 1;
renderPassInfo.pAttachments = &colorAttachment;
renderPassInfo.subpassCount = 2;
VkSubpassDescription subpasses[] = {subpass1, subpass2};
renderPassInfo.pSubpasses = subpasses;

VkRenderPass renderPass;
vkCreateRenderPass(device, &renderPassInfo, nullptr, &renderPass);

VkFramebufferCreateInfo framebufferInfo = {};
framebufferInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
framebufferInfo.renderPass = renderPass;
framebufferInfo.attachmentCount = 1;
framebufferInfo.pAttachments = &colorAttachmentView;
framebufferInfo.width = width;
framebufferInfo.height = height;
framebufferInfo.layers = 1;


VkFramebuffer framebuffer;
vkCreateFramebuffer(device, &framebufferInfo, nullptr, &framebuffer);

VkPipeline pipeline1, pipeline2;

VkCommandBufferAllocateInfo cmdBufAllocInfo = {};
cmdBufAllocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
cmdBufAllocInfo.commandPool = cmdPool;
cmdBufAllocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
cmdBufAllocInfo.commandBufferCount = 1;

VkCommandBuffer cmdBuf;
vkAllocateCommandBuffers(device, &cmdBufAllocInfo, &cmdBuf);
```

Існує два підходи до розділення етапів розрахунку кадру. Перший передбачає створення двох окремих VkRenderPasses і передачу інформації між ними. Другий підхід полягає у створенні одного VkRenderPass з двома VkSubpasses. Різниця між цими двома методами полягає у кількості інформації, що передається між різними рівнями пам'яті комп'ютерної системи. При використанні дво окремих VkRenderPass інформація зчитується зі швидкістю 2905 мегабайт на секунду, а записується - 2289 мегабайт на секунду. А при реалізації двох VkSubpass у рамках одного VkRenderPass інформація зчитується зі швидкістю 825 Мб/с, а записується - 788 Мб/с. Ця різниця особливо важлива на мобільних пристроях, де передача інформації є енергоємною операцією [1, 2, 3, 4].

3.4 Використання бар’єрів графічного конвеєру

Vulkan використовує бар'єри графічного конвеєра для синхронізації спільних ресурсів, що використовуються командами, які виконуються на графічному процесорі. Ці бар'єри поділяються на три типи: VkMemoryBarrier, VkBufferMemoryBarrier та VkImageMemoryBarrier, залежно від ресурсу, який вони обробляють. Бар'єр, доданий під час запису команди, встановлює залежність між командами, записаними до і після бар'єру.

Залежно від типу використовуваного бар'єру, час розрахунку кадру може змінюватися. Наприклад, використання бар'єру BOTTOM_OF_PIPE_BIT - TOP_OF_PIPE_BIT займає 14.12 мс для рендерингу кадру, але блокує можливість одночасного виконання фрагментного та вершинного шейдерів. На противагу цьому, коли бар'єр COLOR_ATTACHMENT_OUTPUT_BIT дорівнює VERTEX_SHADER_BIT, час рендерингу кадру залишається 14.12 мс, оскільки у вершинному шейдері другого проходу не відбувається жодних обчислень. Тому для відкладеного рендерингу, де виконуються обчислення шейдерів фрагментів, найоптимальнішим бар'єром є COLOR_ATTACHMENT_OUTPUT_BIT -

FRAGMENT_SHADER_BIT, що призводить до часу прорахунку кадру - 10,31 мс [2, 4].

3.5 Оптимальний метод синхронізації центрального та графічного процесорів

Для синхронізації CPU і GPU у Vulkan доступні різні методи. Одним з простих, але неефективних методів є використання vkQueueWaitIdle або vkDeviceWaitIdle. Ці команди змушують процесор чекати, доки пристрій або черга на пристрої не завершить виконання всіх команд. Хоча цей метод надійний, він призводить до того, що графічний процесор простоює, що може спричинити значну затримку в процесі рендерингу. Коли паралельне виконання вершинної та фрагментної стадій графічного конвеєра для різних кадрів зупиняється, це призводить до збільшення часу, необхідного для обчислення кадру.

Однак, альтернативним методом є використання об'єктів VkFence. Ці об'єкти створюються спеціально для того, щоб сповіщати центральний процесор про те, що графічний процесор готовий до отримання наступних інструкцій. Такий підхід усуває проблему простою графічного процесора, оскільки центральний процесор точно знає, коли надсилати команди, необхідні для обчислення наступного кадру. Це дозволяє CPU надсилати команди, не чекаючи, поки GPU завершить попередні завдання, що призводить до більш ефективного процесу рендерингу.

Щоб продемонструвати вплив цих двох методів, було проведено тестування на двох прикладах. У першому прикладі vkDeviceWaitIdle використовувався перед початком кожного нового кадру. Це змушувало графічний процесор завершувати всю роботу до початку наступного кадру, що призводило до простою ресурсів графічного процесора. В результаті час, витрачений на обчислення кадру, склав 15.1 мс, що значно більше, ніж у другому прикладі.

У другому прикладі об'єкти VkFence використовувалися для повідомлення CPU про готовність GPU до отримання наступних інструкцій, що призвело до більш ефективного процесу рендерингу. Час, витрачений на обчислення кадру,

склав 9.45 мс, що помітно швидше, ніж у першому прикладі. Отже, очевидно, що використання об'єктів VkFence є набагато ефективнішим способом синхронізації CPU і GPU у Vulkan [2, 4].

Висновок до розділу

У розділі було розглянуто декілька варіантів оптимізації методу візуалізації прозорих об’єктів, а саме Loop32-метод, з використанням засобів, які надає Vulkan API.

Оскільки Loop32-метод виконує два обчислювальних проходи для отримання кінцевого результату зображення, то доцільним є використання VkSubpass для оптимізації роботи програмного забезпечення. За допомогою такої реалізації вдалося зменшити кількість пам’яті, що зчитується та записується на графічному процесорі, у 70%. Таке покращення особливо є важливим для комп’ютерних систем з необхідністю автономної роботи, оскільки покращую енергоефективність системи.

Використання засобів VkDescriptorSet, що описане у розділі, зменшую час необхідний на обрахунок кінцевого зображення майже у 10%.

Збереження графічного конвеєру за допомогою кешування та використання цієї інформацію зменшує час створення нового графічного конвеєру приблизно у два рази. Це зменшує витрати ресурсів на ініціалізацію програмного забезпечення при запуску та перед обрахунком кожного кадру.

Правильне використання бар’єрів при синхронізації роботи графічного процесору зменшує час обрахунку зображення на 35%. Бо це дає змогу тримати постійну нагрузку на графічний процесор.

Використання засобу VkFence для синхронізації центрального та графічного процесорів призводить до зменшення час обрахунку кадру на 14%.

За отриманими результатами можна побачити, що ефективне використання засобів API Vulkan дає можливість оптимізувати роботу комп’ютерних систем для візуалізації тривимірних сцен.

ВИСНОВКИ

У дисертації було розглянуто метод візуалізації прозорих об’єктів та запропоновано шляхи оптимізації методу за допомогою використання засобів, що надає Vulkan API.

Було проаналізовано та порівняно декілька методів візуалізації прозорих об’єктів. Більшість методів є точними методами, тобто такими, що враховують усі об’єкти. Один із них є приближеним, Для проведення порівняння роботи всіх методів було розроблено програмне забезпечення за допомогою мови програмування C++. Аналіз результатів порівнянн показує, що Simple-метод є найменш ефективним з представлених, оскільки він схильний до прояву артефактів і використовує значну кількість ресірсів графічного процесору. WeightedBlended-метод надає приближені результати обчислень прозорості об’єктів на сцені, але при цьому має найвищу швидкодію та використовує найменше пам’яті в порінянні з усіма представленими методами візуалізації прозорих об’єктів. Spinlock-метод використовує блокування процесів для уникнення втрати інформації в критичних зона, коли різні процеси одночасно працюють з одною сомою ділянкою пам’яті. Це призводить до простою ресурсів графічного процесору, що шкодить паралельним обчислення. LinkedList-метод потребує значної кількості пам’яті для отрамання реалістичних результатів та уникнення появи артефактів. Основним недоліком Loo32-методу є використання двох обчислювальних проходів для отримання кінцевого результат. Але це може бути оптимізовано з використання засобів, що надає Vulkan. Тому цей метод і був використаний для досліджень в даній роботі.

Впровадження VkSubpass для реалізації Loop32-методу зменшило об’єми інформації, що проходять через графічний процесор, в двічі. Це особливо важливо для автономних систем, оскільки значно збілшує їх енергоефективність. Впровадження системи кешування VkDescriptorSet зменшило час необхідний для обчислень на 10%. А використання VkFence для синхронізації роботи центрального процесора та графічного процесора, зменшило час на 14%.

Зберігання та кешування графічного конвеєру (VkPipeline) зменшило час ініціалізації програмного забезпечення перед обчисленням кадру майже у два рази. Правильне використання бар’єрів пришвидвишо роботу методу на 35%.

Засоби Vulkan API надають можливість детального контролю ресурсами графічного процесору, що і було використано в ході роботи для оптимізації методу візуалізації на комп’ютерних системах.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Новіков Олександр, Оптимізація Loop32-методу для візуалізації напівпрозорих об’єктів на основі Vulkan. 89 Міжнародна наукова конференція молодих учених, аспірантів і студентів від НУХТ. 2023. 334 с. URL: http://conferencenuft.ho.ua/Books%20of%20abstracts/2023/Part%202.pdf
2. Олександр Новіков, Костянтин Коляда. Практика використання інструментів Vulkan API для підвищення продуктивності програмного забезпечення. CISP Conference «SCIENTIFIC VECTOR OF VARIOUS SPHERE’ DEVELOPMENT: REALITY AND FUTURE TRENDS». 2023. 237-241 с. URL: https://archive.journal-grail.science/index.php/2710-3056/article/view/1136
3. Vulkan Specification. URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html
4. Arm Developer. Best Practices. URL: https://developer.arm.com/documentation/101897/0301/Optimizing-application-logic/Vulkan-GPU-pipelining
5. Eric Haines, Naty Hoffman, Tomas Möller (2019). Real-Time Rendering Third Edition
6. Joey de Vries, Kendall & Welling, Learn OpenGL - Graphics Programming: Learn modern OpenGL graphics programming in a step-by-step fashion, 2020.
7. R. Stuart Ferguson, Practical Algorithms for 3D Computer Graphics, Second Edition, 2013.
8. Morgan McGuire and Louis Bavoil. Weighted Blended Order-Independent Transparency. *Journal of Computer Graphics Techniques (JCGT).* 2013. Vol. 2, No. 2. P. 122-141. URL: https://jcgt.org/published/0002/02/09/
9. Steve Marschner, Peter Shirley, Fundamentals of Computer Graphics, 2021. 716 с.

10. Stanley B. Lippman, Josée Lajoie, Barbara E. Moo, C++ Primer, 2012. 976 c.
11. Graham Sellers, John Kessenich, Vulkan Programming Guide, 2016, 480 c.