

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
ФАКУЛЬТЕТ ПРОГРАМНИХ СИСТЕМ ТА ПРИКЛАДНОЇ МАТЕМАТИКИ
Кафедра системного програмування і спеціалізованих комп'ютерних систем

До захисту допущено:
Завідувач кафедри
_____ Віталій РОМАНКЕВИЧ
(підпис)
“ ” _____ 2026 р.

Дипломний проєкт
на здобуття ступеня бакалавра
за освітньо-професійною програмою
«Системне програмування та спеціалізовані комп'ютерні системи»
спеціальності 123 «Комп'ютерна інженерія»

на тему: Графічний рушій реального часу з використанням GPU-керованого рендерингу та нейронних мереж на базі Vulkan API

Виконав: студент IV курсу, групи КВ-23
Литвин Станіслав Романович

(підпис)

Керівник: доц. каф. СПСКС, к.т.н.,
Клятченко Ярослав Михайлович

(підпис)

Консультант з нормоконтролю, доц. каф. СПСКС, к.т.н.
Клятченко Ярослав Михайлович

(підпис)

Рецензент: доц. каф. ПЗКС, к.т.н.,
Заболотня Тетяна Миколаївна

(підпис)

Засвідчую, що у цьому дипломному
проєкті немає запозичень з праць інших
авторів без відповідних посилань.

Студент

(підпис)

Київ – 2026 року

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

ФАКУЛЬТЕТ ПРОГРАМНИХ СИСТЕМ ТА ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування і спеціалізованих комп'ютерних систем

Рівень вищої освіти – перший (бакалаврський)

Освітньо-професійною програма

**«Системне програмування та спеціалізовані
комп'ютерні системи»**

Спеціальність 123 «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ

Завідувач кафедри

(підпис) Віталій РОМАНКЕВИЧ

“ ____ ” _____ 2026 р.

ЗАВДАННЯ

на дипломний проєкт студента

Литвина Станіслава Романовича

1. Тема проєкту «Графічний рушій реального часу з використанням GPU-керованого рендерингу та нейронних мереж на базі Vulkan API», керівник проєкту Клятченко Я.М., кандидат технічних наук, доцент кафедри системного програмування і спеціалізованих комп'ютерних систем, затверджені наказом по університету від «28» травня 2026 р. № 1984-С

2. Термін подання студентом проєкту _____

3. Вихідні дані до проєкту див. Технічне завдання

4. Зміст пояснювальної записки:

- 1) Аналіз існуючих рішень та обґрунтування теми дипломного проєкту;
- 2) Аналіз та вибір засобів реалізації;
- 3) Архітектурні особливості графічного рушія;
- 4) Тестування та оцінка ефективності розробленого графічного рушія.

5. Перелік графічного матеріалу:

- 1) Структурна схема графічного конвеєра рендерингу;
- 2) Алгоритм двоетапного оклюзійного відсікання екземплярів;
- 3) Структурна схема процесу навчання MLP-моделі для апроксимації навколишнього затінення;
- 4) Алгоритм формування вхідних ознак для MLP-моделі.

6. Консультанти розділів проєкту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Клятченко Я.М., доц. каф. СПіСКС, к.т.н		

7. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1.	Вивчення літератури за тематикою проєкту	06.02.2026	
2.	Розроблення та узгодження технічного завдання	20.02.2026	
3.	Аналіз існуючих рішень	10.03.2026	
4.	Підготовка матеріалів першого розділу дипломного проєкту	24.03.2026	
5.	Підготовка матеріалів другого розділу дипломного проєкту	08.04.2026	
6.	Підготовка матеріалів третього розділу дипломного проєкту	30.04.2026	
7.	Підготовка матеріалів четвертого розділу дипломного проєкту	15.05.2026	
8.	Підготовка графічної частини дипломного проєкту	20.05.2026	
9.	Оформлення документації дипломного проєкту	25.05.2026	
10.	Попередній огляд матеріалів диплому на кафедрі	29.05.2026	

Студент

(підпис)

Станіслав ЛИТВИН

Керівник проєкту

(підпис)

Ярослав КЛЯТЧЕНКО

АНОТАЦІЯ

Кваліфікаційна робота включає пояснювальну записку (93 с., 45 рис., 4 табл., 6 додатків).

Об'єкт розробки – графічний рушій реального часу з використанням GPU-керованого рендерингу та нейронних мереж на базі Vulkan API.

Рушій дозволяє: завантажувати тривимірні моделі та матеріали сцени; виконувати обробку геометрії на основі Mesh Shading та мешлетів; рендерити сцену з урахуванням тіней, глобального освітлення та нейромережевого навколишнього затінення. Для реалізації використано мову програмування C++, графічний API Vulkan, мову шейдерного програмування Slang, а також бібліотеки meshoptimizer, AMD FidelityFX SPD і Dear ImGui.

В ході розробки:

- Проведено аналіз сучасних графічних рушіїв та методів нейромережевої апроксимації в комп'ютерній графіці;
- Побудовано архітектуру GPU-керованого конвеєра рендерингу;
- Реалізовано механізми відсікання невидимої геометрії;
- Використано трасування променів для реалізації ефектів освітлення;
- Розроблено нейромережевий метод апроксимації навколишнього затінення;
- Проведено тестування продуктивності та оцінено ефективність реалізованих рішень.

Розроблений графічний рушій може використовуватися як основа для інтерактивної 3D-візуалізації, ігрових сцен, симуляційних середовищ та дослідження сучасних методів рендерингу реального часу.

Ключові слова:

ГРАФІЧНИЙ РУШІЙ, GPU-КЕРОВАНИЙ РЕНДЕРИНГ, ТРАСУВАННЯ ПРОМЕНІВ, VULKAN API, MESH SHADING, RTX NEURAL SHADING, SLANG.

ABSTRACT

The qualification work includes an explanatory note (93 p., 45 fig., 4 tables, 6 appendices).

The object of development is a real-time graphics engine using GPU-driven rendering and neural networks based on the Vulkan API.

The engine allows loading three-dimensional models and scene materials; performing geometry processing based on Mesh Shading and meshlets; rendering the scene with shadows, global illumination, and neural network ambient occlusion. The implementation utilized the C++ programming language, the Vulkan graphics API, the Slang shader programming language, and the meshoptimizer, AMD FidelityFX SPD, and Dear ImGui libraries.

During the development:

- An analysis of modern graphics engines and neural network approximation methods in computer graphics was conducted;
- The architecture of a GPU-driven rendering pipeline was built;
- Mechanisms for culling invisible geometry were implemented;
- Ray tracing was used to implement lighting effects;
- A neural network method for approximating ambient occlusion was developed;
- Performance testing was conducted and the effectiveness of the implemented solutions was evaluated.

The developed graphics engine can be used as a basis for interactive 3D visualization, game scenes, simulation environments, and research into modern real-time rendering methods.

Keywords:

GRAPHICS ENGINE, GPU-DRIVEN RENDERING, RAY TRACING, VULKAN API, MESH SHADING, RTX NEURAL SHADING, SLANG.

[illegible]

[illegible]

[illegible]

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ГАЛУЗЬ РОЗРОБКИ.....	2
2. ПІДСТАВА ДЛЯ РОЗРОБКИ	2
3. МЕТА І ПРИЗНАЧЕННЯ РОБОТИ	2
4. ДЖЕРЕЛА РОЗРОБКИ.....	2
5. ТЕХНІЧНІ ВИМОГИ.....	3
5.1. Вимоги до програмного продукту, що розробляється	3
5.2. Вимоги до апаратного забезпечення	3
5.3. Вимоги до програмного та апаратного забезпечення користувача	3
6. ЕТАПИ РОЗРОБКИ	4

					ІАЛЦ.467800.002 ТЗ							
Змн.	Лист	№ докум.	Підпис	Дата	Графічний рушій реального часу з використанням GPU-керованого рендерингу та нейронних мереж на базі Vulkan API Технічне завдання					Літ.	Лист	Листів
Розроб.		Литвин С.Р.										
Перевір.		Клятченко Я.М.									1	4
										КПІ ім. Ігоря Сікорського ФПСІМ КВ-23		
Н. контр.		Клятченко Я.М.										
Затверд.		Романкевич В.О.										

1. НАЙМЕНУВАННЯ ТА ГАЛУЗЬ РОЗРОБКИ

Назва розробки: «Графічний рушій реального часу з використанням GPU-керованого рендерингу та нейронних мереж на базі Vulkan API».

Галузь застосування: комп'ютерна графіка та інтерактивна 3D-візуалізація ігрових і симуляційних сцен із використанням сучасних можливостей графічних процесорів.

2. ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на дипломне проектування на здобуття першого (бакалаврського) рівня вищої освіти, затверджене кафедрою системного програмування і спеціалізованих комп'ютерних систем Національного технічного університету України «Київський Політехнічний Інститут імені Ігоря Сікорського».

3. МЕТА І ПРИЗНАЧЕННЯ РОБОТИ

Метою даного дипломного проєкту є розробка графічного рушія реального часу з використанням GPU-керованого рендерингу та нейронних мереж для підвищення продуктивності обробки тривимірних сцен і ефективної апроксимації навколишнього затінення.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом інформації є технічна та науково-технічна література з комп'ютерної графіки, технічна документація Vulkan API, матеріали конференцій SIGGRAPH і GDC, а також відкриті технічні матеріали NVIDIA, AMD та Khronos Group.

					ІАЛЦ.467800.002 ТЗ	Лист
						2
Зм	Лист	№ докум.	Підп.	Дата		

5. ТЕХНІЧНІ ВИМОГИ

5.1 Вимоги до програмного продукту, що розробляється

- Підтримка сучасних можливостей Vulkan API;
- Рендеринг тривимірних сцен у режимі реального часу;
- Наявність GPU-керованого конвеєра рендерингу;
- Наявність нейромережевого навколишнього затінення;
- Можливість використання трасування променів для ефектів освітлення;
- Можливість роботи при різних роздільних здатностях кадру;
- Забезпечення ефективного керування пам'яттю та ресурсами графічного процесора;
- Наявність засобів вимірювання продуктивності.

5.2 Вимоги до апаратного забезпечення

- Процесор: 4-ядерний Intel Core i3 / AMD Ryzen 3 або вище;
- Оперативна пам'ять: 8 ГБ;
- Графічний процесор: NVIDIA GeForce RTX 20xx або новіший.

5.3 Вимоги до програмного та апаратного забезпечення користувача

- Операційна система: Windows 10/11;
- Встановлений драйвер графічного процесора NVIDIA.

6. ЕТАПИ РОЗРОБКИ

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів
1.	Вивчення літератури за тематикою проєкту	06.02.2026
2.	Розроблення та узгодження технічного завдання	20.02.2026
3.	Аналіз існуючих рішень	10.03.2026
4.	Підготовка матеріалів першого розділу дипломного проєкту	24.03.2026
5.	Підготовка матеріалів другого розділу дипломного проєкту	08.04.2026
6.	Підготовка матеріалів третього розділу дипломного проєкту	30.04.2026
7.	Підготовка матеріалів четвертого розділу дипломного проєкту	15.05.2026
8.	Підготовка графічної частини дипломного проєкту	20.05.2026
9.	Оформлення документації дипломного проєкту	25.05.2026
10.	Попередній огляд матеріалів диплому на кафедрі	29.05.2026

[illegible]

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ.....	4
ВСТУП	7
1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ОБҐРУНТУВАННЯ ТЕМИ ДИПЛОМНОГО ПРОЄКТУ	8
1.1 Огляд сучасних графічних рушіїв	8
1.1.1 Unreal Engine.....	8
1.1.2 Unity.....	10
1.1.3 Godot.....	12
1.2 Нейромережеві методи апроксимації в комп'ютерній графіці	14
1.3 Обґрунтування теми дипломного проєкту	17
2. АНАЛІЗ ТА ВИБІР ЗАСОБІВ РЕАЛІЗАЦІЇ	18
2.1 Вибір графічного API	18
2.2 Вибір мови програмування	20
2.3 Вибір мови програмування шейдерів	23
2.4 Вибір допоміжних бібліотек та інструментів реалізації	24
2.5 Висновки до розділу	27
3. АРХІТЕКТУРНІ ОСОБЛИВОСТІ ГРАФІЧНОГО РУШІЯ	28
3.1 Архітектура GPU-керованого рендерингу	28
3.1.1 Графічний конвеєр на основі Mesh Shading	28
3.1.2 Організація доступу до ресурсів.....	32
3.2 GPU-кероване відсікання геометрії.....	36
3.2.1 Відсікання за пірамідою видимості.....	37
3.2.2 Оклюзійне відсікання	39
3.2.3 Відсікання трикутників	42
3.2.4 Розподіл мешлетів.....	44

					ІАЛЦ.467800.004 ПЗ		
Змн.	Лист	№ докум.	Підпис	Дата			
Розроб.		Литвин С.Р.			Графічний рушій реального часу з використанням GPU-керованого рендерингу та нейронних мереж на базі Vulkan API Пояснювальна записка	Літ.	Лист
Перевір.		Клячченко Я.М.					
Н. контр.		Клячченко Я.М.				1	93
Затверд.		Романкевич В.О.				КПІ ім. Ігоря Сікорського ФПСІМ КВ-23	

3.3 Відкладений рендеринг	47
3.3.1 Формування буфера видимості.....	49
3.3.2 Відкладена інтерполяція атрибутів	51
3.4 Трасування променів	52
3.4.1 Керування прискорювальними структурами	53
3.4.2 Реалізація тіней від джерел світла.....	56
3.5 Нейромережеве навколишнє затінення	60
3.5.1 Основи навколишнього затінення	61
3.5.2 Архітектура нейронної мережі та прискорення обчислень	62
3.5.3 Процес навчання	64
3.6 Глобальне освітлення	68
3.7 Масштабування та реконструкція зображення	74
3.8 Висновки до розділу	78
4. ТЕСТУВАННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ РОЗРОБЛЕНОГО ГРАФІЧНОГО РУШІЯ	79
4.1 Тестування GPU-керованого конвеєра	79
4.2 Оцінка нейромережевого навколишнього затінення	85
4.3 Висновки до розділу	89
ВИСНОВКИ.....	90
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	91

ДОДАТКИ

Додаток А. ІАЛЦ.467800.005 Д1. Графічний конвеєр рендерингу. Схема структурна.

Додаток Б. ІАЛЦ.467800.006 Д2. Алгоритм двоетапного оклюзійного відсікання екземплярів. Схема алгоритму.

Додаток В. ІАЛЦ.467800.007 Д3. Процес навчання MLP-моделі для апроксимації навколишнього затінення. Схема структурна.

Додаток Г. ІАЛЦ.467800.008 Д4. Алгоритм формування вхідних ознак для MLP-моделі. Схема алгоритму.

Додаток Д. ІАЛЦ.467800.009 Д5. Фрагменти програмного коду.

Додаток Е. ІАЛЦ.467800.010 Д6. Презентація дипломного проєкту.

					ІАЛЦ.467800.004 ПЗ	Лист
						3
Зм	Лист	№ докум.	Підп.	Дата		

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

Буфер видимості – техніка 3D-рендерингу, що відокремлює визначення видимих поверхонь на екрані від їх подальшого затінення.

Буфер глибини – буфер, у якому зберігаються значення глибини фрагментів сцени відносно камери.

Відмалювання – процес обробки геометрії графічним конвеєром для формування фрагментів або пікселів зображення.

Дескриптор – невеликий блок даних, що описує ресурс GPU або спосіб доступу до нього у форматі, придатному для використання графічним процесором.

Екземпляр – окреме розміщення геометричного об'єкта в сцені з власними параметрами трансформації та матеріалу.

Мешлет – компактна група вершин і трикутників, що використовується як одиниця обробки геометрії на графічному процесорі.

Навколишнє затінення – оцінка локального затінення поверхні залежно від близькості навколишньої геометрії.

Оклюдійне відсікання – вилучення з обробки об'єктів або частин геометрії, перекритих іншими поверхнями сцени.

Піраміда видимості – область простору, що визначає частину сцени, видиму камерою.

Шейдер – програма, що виконується на графічному процесорі та описує обробку графічних або обчислювальних даних.

API (Application Programming Interface) – програмний інтерфейс, що визначає набір засобів для взаємодії між програмними компонентами.

AR (Augmented Reality) – технологія доповненої реальності.

BLAS (Bottom-Level Acceleration Structure) – нижній рівень прискорювальної структури для трасування променів.

BVH (Bounding Volume Hierarchy) – ієрархія обмежувальних об'ємів для прискорення пошуку перетинів із геометрією.

CPU (Central Processing Unit) – центральний процесор, що виконує основні керувальні та обчислювальні операції системи.

DDGI (Dynamic Diffuse Global Illumination) – метод динамічного дифузного глобального освітлення.

DLSS (Deep Learning Super Sampling) – технологія масштабування та реконструкції зображення з використанням нейромережових методів.

FP16 – 16-бітний формат чисел із плаваючою комою.

GLSL (OpenGL Shading Language) – мова шейдерного програмування.

GPU (Graphics Processing Unit) – графічний процесор, призначений для паралельної обробки графічних та обчислювальних задач.

HLSL (High-Level Shading Language) – мова шейдерного програмування.

HZB (Hierarchical Z-Buffer) – ієрархічний буфер глибини.

LDS (Local Data Share) – локальна пам'ять для обміну даними між потоками в межах робочої групи GPU.

MAE (Mean Absolute Error) – середня абсолютна похибка.

Mesh Shading – модель обробки геометрії з використанням програмованих шейдерних етапів.

MLP (Multilayer Perceptron) – багатошаровий перцептрон, тип повнозв'язної нейронної мережі.

NDC (Normalized Device Coordinates) – нормалізовані координати пристрою.

NNAO (Neural Network Ambient Occlusion) – нейромережовий метод апроксимації навколишнього затінення.

NRC (Neural Radiance Cache) – нейромережовий метод кешування та апроксимації непрямого освітлення.

RMSE (Root Mean Square Error) – середньоквадратична похибка.

RPP (Rays Per Pixel) – кількість променів, що трасуються для одного пікселя.

Slang – мова шейдерного програмування.

SPD (Single Pass Downsampler) – метод швидкого формування послідовності MIP-рівнів текстури за один прохід на графічному процесорі.

SPIR-V (Standard Portable Intermediate Representation) – проміжне представлення шейдерних програм, що використовується у Vulkan.

SSAO (Screen-Space Ambient Occlusion) – екранно-просторовий метод наближеного обчислення навколишнього затінення.

SSIM (Structural Similarity Index Measure) – метрика оцінювання структурної подібності між двома зображеннями.

TLAS (Top-Level Acceleration Structure) – верхній рівень прискорювальної структури, що містить екземпляри геометрії для трасування променів.

VR (Virtual Reality) – технологія віртуальної реальності.

					ІАЛЦ.467800.004 ПЗ	Лист
Зм	Лист	№ докум.	Підп.	Дата		6

ВСТУП

Сучасні інтерактивні візуальні системи дедалі частіше використовуються для відображення складних тривимірних середовищ, з якими користувач взаємодіє безпосередньо під час роботи програми. До таких систем належать комп'ютерні ігри, симулятори, засоби віртуальної реальності, інженерні та навчальні комплекси. Для них важливо не лише забезпечити достатній рівень деталізації й реалістичності сцени, а й підтримувати стабільне формування кадрів без помітних затримок.

Зі збільшенням деталізації моделей, кількості об'єктів і складності освітлення зростає навантаження на рендеринговий конвеєр. Одним із чинників, що може обмежувати продуктивність, є значна участь центрального процесора у підготовці кадру. Якщо обробка об'єктів, перевірка видимості та формування команд відмалювання виконуються переважно на CPU, то масштабування сцени може ускладнюватися навіть за наявності продуктивного графічного процесора. Для зменшення таких обмежень застосовуються GPU-керовані підходи, а важливе місце серед них займає Mesh Shading, що дає змогу ефективніше організовувати обробку геометрії та краще використовувати паралельну архітектуру GPU.

Поряд з оптимізацією геометричного етапу актуальним напрямом є застосування нейронних методів рендерингу. Вони дають змогу наближено відтворювати складні ефекти освітлення, затінення, фільтрації та реконструкції зображення з меншою обчислювальною вартістю, що є особливо важливим для інтерактивних систем із жорсткими обмеженнями часу формування кадру.

Метою даного дипломного проєкту є розробка рушія для дослідження сучасних підходів до формування зображення тривимірних сцен. Результатом роботи стала програмна система з GPU-керованою організацією рендерингу, ефективною обробкою геометрії сцени та моделлю навколишнього затінення, побудованою на основі нейронних мереж.

1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ОБҐРУНТУВАННЯ ТЕМИ ДИПЛОМНОГО ПРОЄКТУ

1.1 Огляд сучасних графічних рушіїв

Графічний рушій є основою програмної системи, що відповідає за формування зображення, організацію сцени, керування ресурсами та взаємодію з апаратними можливостями графічного процесора. У сучасних інтерактивних застосунках рушій зазвичай не обмежується лише рендерингом, а також містить засоби роботи з об'єктами сцени, матеріалами, освітленням, введенням користувача, анімацією, фізичним моделюванням і редакторським інструментарієм. Завдяки цьому графічні рушії застосовуються не лише в ігровій індустрії, а й у симуляторах, навчальних середовищах, системах візуалізації, архітектурних проєктах та застосунках віртуальної й доповненої реальності.

Сучасні рушії відрізняються не лише набором функцій, а й внутрішньою архітектурою, рівнем абстракції, підходом до організації рендерингового конвеєра та можливостями розширення. Тому їх доцільно аналізувати не за одним показником, а за сукупністю характеристик: якістю графіки, продуктивністю, підтримкою платформ, мовами програмування, складністю освоєння, доступністю низькорівневого керування, можливостями оптимізації та гнучкістю роботи з GPU [1].

1.1.1 Unreal Engine

Unreal Engine – це графічний рушій, розроблений компанією Epic Games, який використовується для створення комп'ютерних ігор, інтерактивних 3D-застосунків, архітектурної візуалізації, віртуального виробництва, кіноіндустрії, а також VR- та AR-рішень. Рушій орієнтований на проєкти з високими вимогами до візуальної якості, складної тривимірної сцени,

реалістичного освітлення та масштабованого контенту. Інтерфейс редактора Unreal Engine 5 наведено на рисунку 1.1.

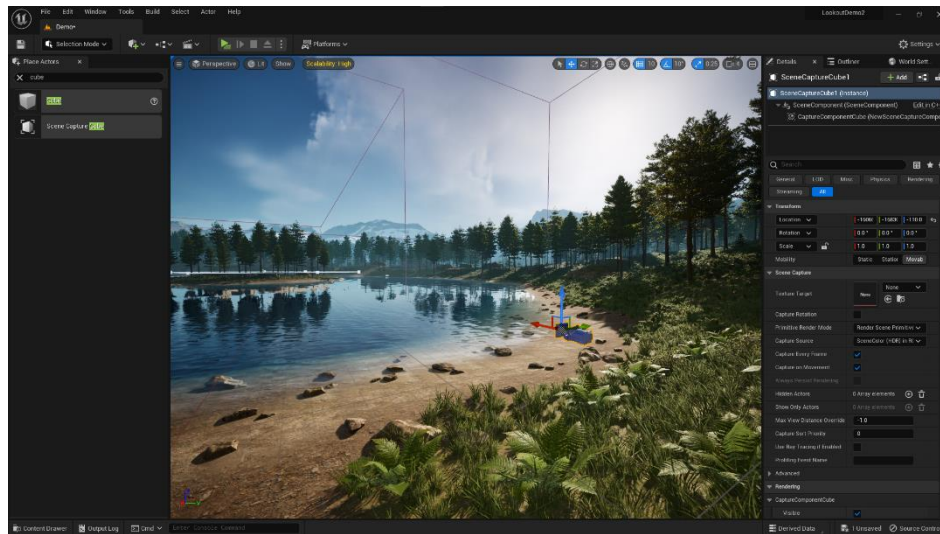


Рисунок 1.1 – Інтерфейс редактора Unreal Engine 5

Однією з ключових переваг Unreal Engine є наявність сучасних технологій рендерингу, зокрема Lumen і Nanite. Lumen забезпечує динамічне глобальне освітлення та відбиття світла в реальному часі, що дає змогу формувати більш природне освітлення сцени без обов'язкового попереднього запікання світлових карт. Nanite призначений для ефективної роботи з високодеталізованою геометрією та зменшує потребу в ручному створенні кількох рівнів деталізації для об'єктів сцени. Завдяки цьому Unreal Engine є ефективним інструментом для побудови графічно насичених сцен, великих відкритих світів і проєктів із високим рівнем деталізації.

Рушій підтримує два основні підходи до реалізації логіки застосунку. Перший підхід передбачає використання мови C++, що забезпечує високу продуктивність, доступ до внутрішніх механізмів рушія та можливість створення складних програмних систем. Другий підхід базується на системі Blueprints – візуальному середовищі програмування, яке дає змогу швидко створювати прототипи та реалізовувати логіку без написання низькорівневого

коду. Це робить Unreal Engine доступним не лише для програмістів, а й для технічних художників, дизайнерів рівнів та інших учасників розробки.

До переваг Unreal Engine належать розвинений інструментарій редактора, підтримка процедурної генерації контенту, засоби роботи з великими ігровими світами, підтримка віртуального виробництва, потужна система матеріалів і велика екосистема готових рішень. Водночас така універсальність має і певні недоліки. Рушій характеризується складною архітектурою та значною кількістю вбудованих підсистем, що підвищує поріг входу й ускладнює аналіз внутрішніх механізмів. Для ефективної роботи з великими проєктами розробнику необхідно розуміти принципи організації ресурсів, особливості рендерингового конвеєра та механізми оптимізації. Повноцінне використання можливостей Unreal Engine також потребує продуктивного апаратного забезпечення, зокрема сучасного графічного адаптера, достатнього обсягу оперативної пам'яті та потужного центрального процесора.

У контексті дослідження власної архітектури графічного рушія важливим обмеженням є те, що Unreal Engine залишається готовою універсальною платформою з усталеною внутрішньою структурою. Це може створювати надмірні накладні витрати для задач, де потрібен мінімальний і повністю контрольований рендеринговий конвеєр. Реалізація експериментальних обчислювальних етапів вимагає глибокого втручання у внутрішні механізми рушія. У зв'язку з цим Unreal Engine є потужним готовим рішенням, однак не завжди є оптимальною основою для побудови компактної дослідницької платформи з повним контролем над рендерингом.

1.1.2 Unity

Unity – це кросплатформний графічний рушій, розроблений компанією Unity Technologies. Він широко застосовується для створення двовимірних і тривимірних ігор, інтерактивних застосунків, симуляторів, навчальних систем,

візуалізацій. Unity підтримує велику кількість платформ, зокрема Windows, Linux, macOS, Android, iOS, WebGL та ігрові консолі. Інтерфейс редактора Unity наведено на рисунку 1.2.

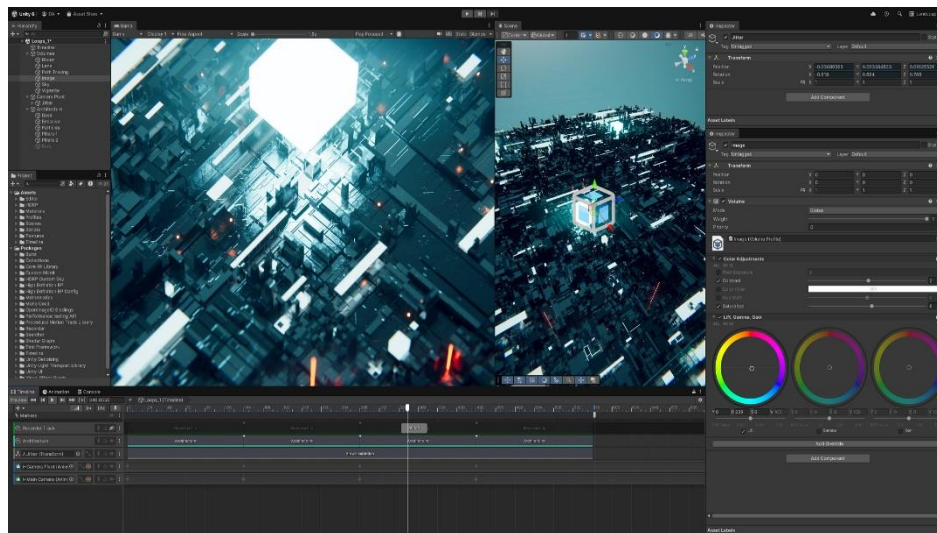


Рисунок 1.2 – Інтерфейс редактора Unity

Важливою перевагою Unity є відносно низький поріг входу. Редактор рушія має зрозумілу структуру та містить засоби для роботи зі сценою, ієрархією об'єктів, компонентами, ресурсами проєкту та переглядом результату виконання програми. Завдяки цьому Unity активно використовується незалежними розробниками, невеликими командами, освітніми закладами та комерційними студіями.

Основною мовою програмування в Unity є C#. Вона є достатньо зручною для швидкого створення прикладної логіки та водночас підтримує об'єктно-орієнтовані механізми, необхідні для побудови складніших систем. Використання C# спрощує процес розробки, проте в окремих високонавантажених задачах може поступатися нативним рішенням за рівнем продуктивності. Це особливо помітно в системах, чутливих до часових затримок, великої кількості об'єктів або частих оновлень логіки.

Суттєвою особливістю Unity є наявність кількох рендерингових конвеєрів: Built-in Render Pipeline, Universal Render Pipeline та High Definition

Render Pipeline. Universal Render Pipeline орієнтований на баланс між якістю графіки та продуктивністю, тому його часто застосовують у мобільних і кросплатформних проєктах. High Definition Render Pipeline призначений для застосунків із підвищеними вимогами до якості візуалізації та використовується переважно на потужних персональних комп'ютерах і консолях. Наявність кількох конвеєрів дає змогу адаптувати рушій до різних класів задач, однак водночас ускладнює сумісність окремих шейдерів, плагінів і графічних інструментів.

Водночас Unity не є повністю GPU-керованим рушієм у своїй базовій архітектурі. Хоча рушій підтримує окремі GPU-орієнтовані механізми, зокрема Compute Shader, GPU Instancing, DrawMeshInstancedIndirect та GPU Resident Drawer, ці засоби переважно використовуються для оптимізації окремих етапів рендерингу, зменшення кількості викликів відмалювання та зниження навантаження на центральний процесор. Вони не змінюють загальну модель рушія на повністю GPU-керований графічний конвеєр, у якому підготовка сцени, відсікання, вибір рівнів деталізації та формування команд відмалювання виконуються переважно на графічному процесорі.

1.1.3 Godot

Godot – це графічний рушій із відкритим вихідним кодом, який поширюється за ліцензією MIT. Його важливою перевагою є можливість безкоштовного використання в навчальних, дослідницьких і комерційних проєктах без сплати роялті. Завдяки відкритості, простоті розгортання та активному розвитку Godot став популярним рішенням серед незалежних розробників, невеликих команд і користувачів, яким потрібен гнучкий інструмент для створення 2D- та 3D-застосунків. Інтерфейс редактора Godot наведено на рисунку 1.3.

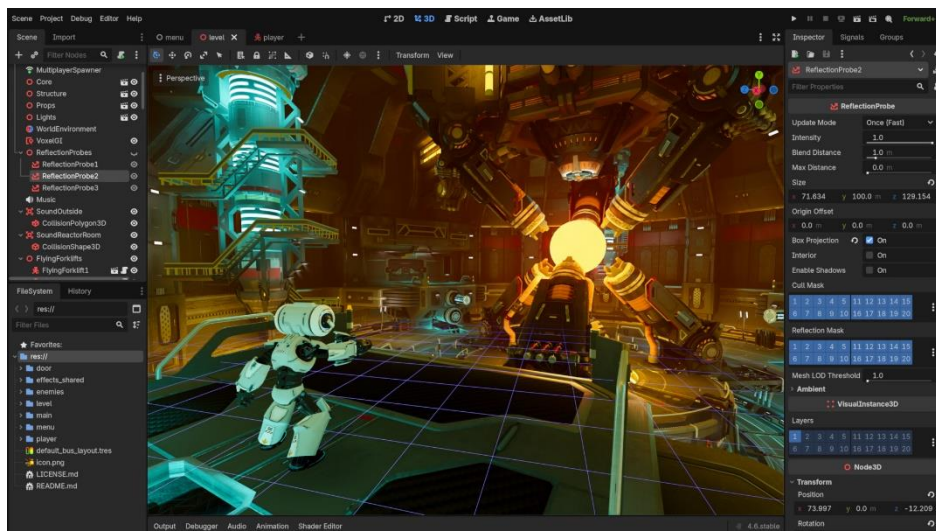


Рисунок 1.3 – Інтерфейс редактора Godot

Архітектура Godot побудована навколо концепції сцен і вузлів. У межах такого підходу об'єкти проєкту формуються як ієрархії вузлів, кожен із яких виконує окрему функцію: відповідає за відображення графіки, фізичну взаємодію, анімацію, звук, освітлення, роботу камери або виконання скриптів. Сцена в Godot може розглядатися як самостійна структура, яку можна повторно використовувати, вкладати в інші сцени та комбінувати з іншими елементами проєкту. Це спрощує організацію складних об'єктів і сприяє модульності розробки.

Для реалізації логіки Godot підтримує кілька підходів. Основною мовою рушія є GDScript, орієнтована на швидке створення ігрової та прикладної логіки. Також доступна підтримка C#, а для задач, що потребують вищої продуктивності або інтеграції нативного коду, використовується механізм GDExtension. Завдяки цьому Godot може застосовуватися як для швидкого прототипування, так і для створення спеціалізованих рішень із власними низькорівневими компонентами.

Особливістю Godot є можливість використовувати рушій не лише як готовий інструмент розробки, а й як систему, внутрішню реалізацію якої можна аналізувати та змінювати. Розробник має доступ до коду окремих підсистем, тому за потреби може адаптувати поведінку редактора, механізмів сцени,

рендерингу або інтеграції нативних модулів. Це підвищує гнучкість рушія та робить його зручним для випадків, коли стандартного функціоналу недостатньо або потрібне точніше налаштування під вимоги конкретного проєкту.

Попри ці переваги, Godot має й певні обмеження. Його екосистема поки що менш розвинена порівняно з Unreal Engine та Unity: рушій має меншу кількість готових комерційних плагінів, навчальних матеріалів, промислових прикладів використання та впроваджень у великих студіях. У складних тривимірних проєктах це може збільшувати обсяг ручного налаштування, оптимізації та доопрацювання окремих підсистем. Хоча Godot підтримує механізми на зразок MultiMesh для ефективнішого відображення великої кількості однотипних об'єктів, такі засоби не забезпечують повноцінної GPU-керованої організації рендерингу, а лише оптимізують окремі випадки роботи з геометрією.

1.2 Нейромережеві методи апроксимації в комп'ютерній графіці

Останніми роками в комп'ютерній графіці дедалі активніше застосовуються нейромережеві методи. Їх використання пов'язане з тим, що багато класичних алгоритмів формування зображення мають високу обчислювальну складність і потребують значних ресурсів для отримання якісного результату в реальному часі. У рендерингу такі методи можуть використовуватися для таких задач:

- реконструкція зображення;
- усунення шуму після трасування променів або трасування шляху;
- апроксимація освітлення та навколишнього затінення;
- відновлення деталей сцени на основі просторових даних;

На відміну від традиційних підходів, які здебільшого ґрунтуються на аналітичних формулах, геометричних побудовах або стохастичному семплюванні, нейронні мережі дають змогу наближено відтворювати складні

залежності на основі попереднього навчання або адаптації під час роботи системи. У задачах рендерингу це особливо корисно тоді, коли точне обчислення певної величини є занадто дорогим, але допустиме використання її наближеної оцінки. Таким чином, нейромережа може виконувати роль апроксиматора функції освітлення, затінення, матеріалу або іншої характеристики сцени.

Одним із характерних прикладів нейромережевої апроксимації в комп'ютерній графіці є Neural Radiance Cache. Метод призначений для наближеного обчислення непрямого освітлення під час трасування шляху [2]. У класичному трасуванні шляху для отримання реалістичного глобального освітлення потрібно трасувати багато світлових шляхів, оскільки непряме освітлення формується внаслідок багаторазових відбиттів світла від поверхонь сцени. За обмеженої кількості вибірок це призводить до появи шуму, а збільшення кількості променів суттєво підвищує обчислювальні витрати.

Ідея такого підходу пов'язана з розвитком методів кешування освітлення. У класичних підходах раніше обчислена світлова інформація повторно використовувалася для прискорення розрахунку непрямого освітлення та зменшення кількості дорогих обчислень під час трасування [3]. NRC розвиває цей принцип, однак замість зберігання великої кількості окремих кеш-записів використовує компактну нейронну модель. Така модель апроксимує значення розсіяного радіансу й фактично подає інформацію про освітлення у вигляді неперервного функціонального представлення.

Під час рендерингу така модель може використовуватися для завершення частини світлових шляхів без їх подальшого повного трасування. Це дає змогу зменшити кількість обчислень, скоротити довжину шляхів і знизити рівень шуму в результаті. Важливою особливістю NRC є здатність адаптуватися до динамічних сцен, оскільки навчання може виконуватися безпосередньо під час рендерингу. Завдяки цьому метод може враховувати зміни геометрії, матеріалів, джерел світла та положення камери. Порівняння результатів

звичайного трасування шляху, методу NRC та еталонного зображення наведено на рисунку 1.4.



Рисунок 1.4 – Порівняння результатів рендерингу: трасування шляху, метод NRC та еталонне зображення

Іншим прикладом нейромережевої апроксимації є Neural Network Ambient Occlusion, призначений для наближеного обчислення навколишнього затінення на основі локальних ознак сцени [4]. У традиційних методах навколишнього затінення рівень оклюзії визначається за даними про глибину, нормалі поверхонь і набором евристичних правил вибірки. У нейромережевому підході ця залежність відтворюється навченою моделлю, яка аналізує вхідні атрибути фрагмента та його оточення й формує оцінку локального затінення.

Перевага такого підходу полягає в тому, що складна залежність між геометричними ознаками сцени та рівнем локального затінення не задається вручну, а формується під час навчання моделі. Це дає змогу отримати компактне наближення ефекту оклюзії, яке може виконуватися без повного відтворення дорогого трасування променів. Водночас якість результату залежить від добору вхідних ознак, репрезентативності навчальних даних і здатності моделі узагальнювати різні геометричні конфігурації сцени.

Отже, нейромеревеві методи апроксимації є перспективним напрямом розвитку комп'ютерної графіки, оскільки дають змогу замінювати частину дорогих обчислень швидким наближеним оцінюванням. Для їх ефективного застосування важливо мати контроль над графічним конвеєром, підготовкою даних, організацією пам'яті та взаємодією між CPU і GPU.

1.3 Обґрунтування теми дипломного проєкту

Проведений аналіз показує, що сучасні графічні рушії надають розвинений інструментарій для створення інтерактивних тривимірних застосунків. Unreal Engine орієнтований на високу якість візуалізації та роботу зі складними сценами, Unity вирізняється зручністю використання, кросплатформністю й великою екосистемою, а Godot є відкритим і гнучким рішенням, придатним для навчальних, дослідницьких та інди-проєктів.

Попри ці переваги, готові рушії не завжди є оптимальною основою для дослідження нових підходів до побудови рендерингових систем. Їхня архітектура зазвичай орієнтована на широкий спектр прикладних задач, тому значна частина внутрішніх механізмів прихована за високорівневими API або тісно пов'язана з уже наявними підсистемами. Це ускладнює реалізацію власних схем підготовки відмалювання, нестандартних методів, спеціалізованої організації ресурсів і експериментальних обчислювальних етапів.

З огляду на це доцільним є розроблення власного графічного рушія, орієнтованого не на максимальну універсальність, а на дослідження сучасних підходів до організації рендерингової системи. Такий рушій дасть змогу сформувати архітектуру відповідно до поставлених завдань, зменшити залежність від процесорної підготовки відмалювання, ефективніше використовувати можливості графічного процесора та забезпечити гнучкість для подальшої інтеграції нейромеревевих методів апроксимації.

2. АНАЛІЗ ТА ВИБІР ЗАСОБІВ РЕАЛІЗАЦІЇ

2.1 Вибір графічного API

Графічний API визначає спосіб взаємодії рушія з графічним процесором. Через нього створюються буфери, текстури, дескриптори та графічні пайплайни, формуються команди відмалювання, задаються правила синхронізації та організовується виконання графічних і обчислювальних задач. Тому для графічного рушія API є не просто засобом виведення зображення, а основою, від якої залежить структура рендеринг-конвеєра, робота з GPU-ресурсами та можливість низькорівневої оптимізації.

Для невеликих застосунків або навчальних проєктів часто достатньо простіших рішень, у яких більшість внутрішніх деталей прихована від розробника. Такий підхід прискорює початкову реалізацію, але обмежує можливості тонкого налаштування продуктивності та ускладнює впровадження нестандартних рендерингових підходів. Оскільки в межах цього проєкту передбачається створення власного рушія з гнучкою структурою графічного конвеєра, доцільно розглядати сучасні низькорівневі API, зокрема Direct3D 12, Metal та Vulkan. Такі API краще відповідають задачам, у яких важливо не лише отримати зображення, а й контролювати спосіб його формування на рівні окремих проходів.

Direct3D 12 – це графічний API компанії Microsoft, орієнтований насамперед на Windows та Xbox. Він підтримує явне керування командними списками, чергами виконання, дескрипторами, станами ресурсів і синхронізацією. Direct3D 12 широко використовується в сучасних ігрових проєктах і має розвинену інструментальну підтримку в середовищі Windows. Його доцільно застосовувати тоді, коли цільова платформа розробки чітко пов'язана з екосистемою Microsoft.

Metal – це графічний API компанії Apple, призначений для macOS, iOS та tvOS. Він забезпечує ефективну роботу з GPU на пристроях Apple і добре

враховує особливості відповідної апаратно-програмної платформи. Завдяки цьому Metal є ефективним рішенням для застосунків, які розробляються саме під екосистему Apple. Водночас така спеціалізація робить його менш універсальним для рушія, який має зберігати гнучкість архітектури та можливість подальшого розвитку поза межами однієї платформи.

Vulkan – це відкритий кросплатформний графічний API, розроблений Khronos Group. Його головна відмінність від Direct3D 12 і Metal полягає в меншій залежності від конкретної операційної системи або програмної екосистеми. Завдяки цьому Vulkan є придатним варіантом для рушія, архітектура якого не повинна бути жорстко пов'язана з однією платформою. З технічної точки зору Vulkan дає змогу самостійно визначати порядок підготовки кадру, структуру графічних і обчислювальних проходів, спосіб передавання даних на GPU та взаємодію між окремими етапами конвеєра. Це є суттєвою перевагою для розроблюваного рушія, оскільки дозволяє не обмежуватися готовою високорівневою абстракцією, а формувати рендеринг-підсистему відповідно до поставлених задач.

Окремою перевагою Vulkan є використання SPIR-V – проміжного представлення шейдерних програм, що дає змогу застосовувати різні мови шейдерного програмування та компіляційні інструменти. Такий підхід дає змогу організувати гнучкішу шейдерну кодову базу та спростити подальше розширення графічної підсистеми новими обчислювальними етапами. Водночас така гнучкість супроводжується високою складністю реалізації: значна частина роботи, яка у високорівневих API виконується автоматично, у Vulkan покладається на розробника. Це збільшує обсяг службового коду й підвищує вимоги до проектування рушія, однак у межах даного проекту така складність є виправданою, оскільки вона забезпечує потрібний рівень контролю для побудови власної керованої рендеринг-підсистеми.

2.2 Вибір мови програмування

Реалізація графічного рушія передбачає роботу з великою кількістю низькорівневих і високорівневих компонентів. З одного боку, рушій має ефективно взаємодіяти з графічним API, керувати пам'яттю, ресурсами GPU, буферами, текстурами та командами відмалювання. З іншого боку, його кодова база повинна залишатися структурованою, розширюваною та придатною для підтримки, оскільки система містить багато взаємопов'язаних підсистем.

Через це мова програмування повинна поєднувати дві групи властивостей. Перша пов'язана з продуктивністю: компіляція в ефективний машинний код, передбачуване виконання, контроль над пам'яттю та відсутність небажаних затримок під час роботи в реальному часі. Друга стосується організації проекту: наявність засобів для модульної побудови системи, опису складних абстракцій, керування життєвим циклом ресурсів і зручної інтеграції з зовнішніми бібліотеками.

Для вибору мови реалізації ядра рушія було проаналізовано C, C++, C# та Rust. Вони відрізняються рівнем абстракції, моделлю виконання, способом керування пам'яттю, зрілістю екосистеми та зручністю інтеграції з низькорівневими графічними API. Порівняння цих мов дає змогу визначити варіант, який найкраще відповідає вимогам до продуктивності, контрольованості та масштабованості ядра рушія.

Мова C забезпечує високу продуктивність, прямий доступ до пам'яті та мінімальний рівень прихованих механізмів виконання. Завдяки цьому вона може бути ефективною для реалізації невеликих низькорівневих компонентів або окремих модулів, де важливі простота виконання, мінімальні накладні витрати та прямий контроль над даними. Проте для побудови повноцінного графічного рушія її можливості є обмеженими. У такій системі необхідно реалізовувати менеджери ресурсів, обгортки над графічним API, систему команд, модулі рендерингу, завантаження даних і службові інструменти. Без

вбудованої підтримки класів, шаблонів, просторів імен та зручних механізмів керування життєвим циклом об'єктів архітектура на C швидко ускладнюється і потребує значного обсягу допоміжного коду.

C++ зберігає ключові переваги C, зокрема високу продуктивність, компіляцію в машинний код і можливість прямої роботи з пам'яттю, але водночас надає значно ширші засоби для побудови складних програмних систем. Класи, шаблони, простори імен, стандартні контейнери, перевантаження функцій і механізми автоматичного керування ресурсами дозволяють організовувати код рушія більш структуровано та підтримувано. Це важливо для реалізації підсистем, які містять багато взаємозалежних об'єктів: ресурсів GPU, командних списків і службових модулів.

C# має інший профіль застосування. Ця мова забезпечує зручний синтаксис, розвинену стандартну бібліотеку, об'єктно-орієнтовану модель і високий рівень абстракції. Вона добре підходить для прикладної логіки, інструментальних систем, редакторів і швидкого прототипування. Саме тому C# активно використовується в рушіях на рівні користувацької логіки, зокрема в Unity. Однак для ядра власного графічного рушія ця мова є менш придатною. Виконання в керованому середовищі, наявність збирача сміття та додатковий шар між програмою й апаратними ресурсами ускладнюють точне керування пам'яттю та можуть негативно впливати на стабільність часу кадру.

Rust є сучасною альтернативою для створення продуктивних програм без використання збирача сміття. Його модель володіння ресурсами спрямована на безпечну роботу з пам'яттю та зменшення кількості помилок, пов'язаних із некоректним доступом до даних. Це робить Rust перспективним варіантом для низькорівневих і високопродуктивних задач. Водночас для розробки графічного рушія важлива не лише сама мова, а й зрілість екосистеми, кількість перевірених бібліотек, приклади використання у великих графічних системах і зручність інтеграції з наявними інструментами рендерингу. За цими критеріями

Rust поки що поступається C++, тому його використання в межах цього проєкту є менш доцільним.

Після розгляду особливостей кожної мови доцільно узагальнити їх за ключовими критеріями, важливими для реалізації ядра графічного рушія. Таке порівняння наведено в таблиці 2.1.

Таблиця 2.1 – Порівняльна характеристика мов програмування

Критерій	C	C++	C#	Rust
Продуктивність	Висока	Висока	Середня	Висока
Контроль пам'яті	Повний	Повний	Обмежений	Повний
Архітектурна гнучкість	Низька	Висока	Висока	Висока
Екосистема	Зріла	Розвинена	Розвинена	Зростаюча
Інтеграція з Vulkan	Пряма	Пряма	Через обгортки	Через обгортки

Як видно з таблиці 2.1, кожна з розглянутих мов має власну сферу доцільного застосування. Мова C забезпечує високу продуктивність і прямий контроль над пам'яттю, однак має обмежені засоби для побудови складної архітектури. C# є зручним для прикладної логіки та інструментальних систем, проте наявність керованого середовища виконання і збирача сміття знижує його придатність для ядра рушія реального часу. Rust має сильну модель безпечної роботи з пам'яттю, але його екосистема для великих графічних систем поки що менш зріла порівняно з C++. Таким чином, найбільш збалансованим варіантом є C++, оскільки він поєднує високу продуктивність, контроль над ресурсами, розвинені засоби архітектурної організації та зручну інтеграцію з Vulkan.

2.3 Вибір мови програмування шейдерів

Мови шейдерного програмування є основним засобом опису обчислень, що виконуються на програмованих етапах сучасного графічного конвеєра. За їх допомогою реалізуються вершинна, фрагментна, обчислювальна та інші стадії обробки даних на GPU. Традиційно такі мови орієнтувалися на компактні шейдерні програми з жорсткими вимогами до продуктивності, тому їх розвиток відбувався під значним впливом апаратних особливостей графічних процесорів.

Найбільш поширеними традиційними мовами шейдерного програмування є GLSL та HLSL. Вони забезпечують доступ до програмованих етапів графічного конвеєра та тривалий час залишалися основним інструментом створення шейдерів. Проте зі зростанням складності рендеринг-систем виявилось, що класичних можливостей цих мов не завжди достатньо для підтримки великих шейдерних кодових баз. Для сучасних рушіїв важливими стають модульність, повторне використання коду, контроль видимості, узагальнення та інші механізми, характерні для сучасних мов програмування загального призначення.

Додатково ситуацію ускладнює розвиток Vulkan API та SPIR-V, що підвищує вимоги до інструментів шейдерного програмування. У великих рендеринг-системах це проявляється не лише у зростанні обсягу шейдерного коду, а й у проблемі великої кількості його варіантів, що впливає на процес компіляції, підтримки та супроводу. У зв'язку з цим доцільно розглядати сучасніші засоби програмування шейдерів, орієнтовані на побудову масштабованих і підтримуваних кодових баз.

Одним із таких засобів є Slang – відкрита кросплатформна мова шейдерного програмування та компіляційна інфраструктура, орієнтована на сучасні задачі рендерингу. Slang розширює підхід, характерний для HLSL, і додає засоби, що полегшують організацію складних шейдерних систем [5].

До основних можливостей Slang, важливих для цього проєкту, належать:

- модульна організація шейдерного коду;
- наявність інтерфейсів і узагальнень;
- підтримка автоматичного диференціювання;
- зручність супроводу великої шейдерної кодової бази.

На відміну від традиційних підходів, де похідні доводиться реалізовувати вручну або виносити обчислення до окремих ML-фреймворків, Slang інтегрує автоматичне диференціювання на рівні мови, системи типів та компілятора. Це дозволяє використовувати похідні безпосередньо в шейдерному коді та поєднувати задачі комп'ютерної графіки з методами градієнтної оптимізації. На рисунку 2.1 наведено приклад використання автоматичного диференціювання в Slang для обчислення похідної зображення сцени за зміщенням об'єкта вздовж осі Y [6].

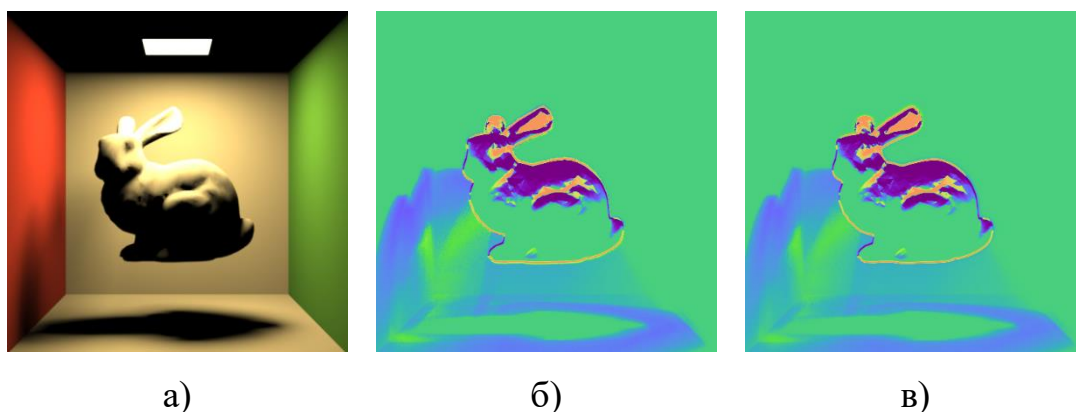


Рисунок 2.1 – Порівняння обчислення похідної: а) вихідне зображення; б) диференціювання засобами Slang; в) еталонний результат

2.4 Вибір допоміжних бібліотек та інструментів реалізації

Окрім графічного API та мов програмування, важливу роль у реалізації графічного рушія відіграє вибір допоміжних бібліотек та інструментів. Вони дозволяють ефективніше реалізувати окремі підсистеми, зменшити обсяг допоміжного коду та використати готові оптимізовані рішення для типових

задач. Однією з таких задач є швидке формування набору рівнів із різною роздільною здатністю на основі вихідного ресурсу. У класичному підході кожен наступний рівень формується окремо на основі попереднього, що вимагає кількох послідовних проходів обробки. Така схема є відносно простою для реалізації, проте створює додаткові накладні витрати, пов'язані з багаторазовим запуском шейдерів, проміжною синхронізацією та повторним доступом до ресурсів.

Для ефективнішого розв'язання цієї задачі у межах проєкту використовується AMD FidelityFX Single Pass Downsampler. Основна ідея цього підходу полягає в тому, що декілька MIP-рівнів можуть формуватися в межах одного проходу, а не послідовно, рівень за рівнем [7]. Порівняння послідовної генерації MIP-рівнів і підходу SPD наведено на рисунку 2.2.

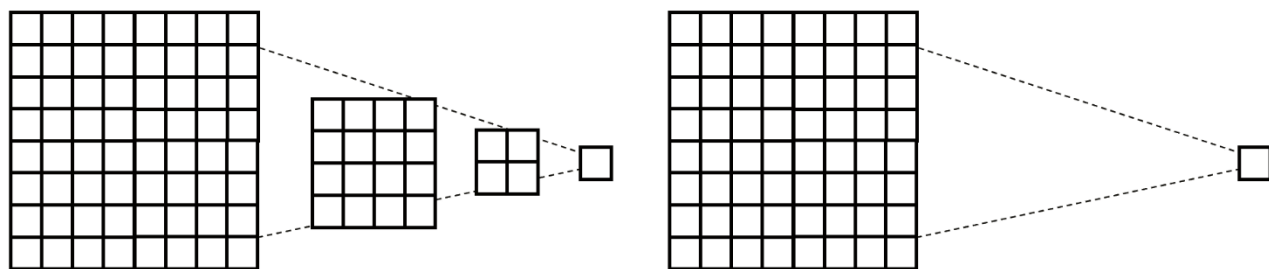


Рисунок 2.2 – Послідовна генерація MIP-рівнів у порівнянні з SPD

До основних переваг використання AMD SPD можна віднести:

- Можливість ефективнішого планування виконання GPU-навантаження завдяки відсутності жорсткої бар'єрної синхронізації між етапами побудови;
- Локальний обмін даними між рівнями за допомогою LDS та механізмів міжпоточної взаємодії, що зменшує інтенсивність звернень до глобальної пам'яті;
- Високий рівень завантаженості GPU за рахунок кращого використання обчислювальних ресурсів під час формування рівнів.

Це робить SPD доцільним рішенням для задач, де потрібно швидко формувати ієрархічні представлення текстур або буферів у межах рендеринг-конвеєра.

Окрім роботи з текстурними та буферними ієрархіями, важливою задачею є оптимізація геометрії сцени. Продуктивність обробки геометрії залежить не лише від кількості трикутників, а й від того, як організовані вершинні та індексні буфери, наскільки ефективно повторно використовуються вершини та чи забезпечується локальність доступу до пам'яті. Якщо геометрія зберігається без індексів, однакові вершини можуть дублюватися, що збільшує обсяг даних і створює додаткове навантаження під час їх передавання та обробки на GPU. Використання індексного представлення зменшує таку надлишковість, а подальша оптимізація порядку вершин і трикутників дає змогу краще використовувати кеші та зменшити кількість зайвих операцій.

Для виконання цих задач у проєкті використовується бібліотека meshoptimizer. Вона надає готові алгоритми для переіндексації сіток, оптимізації використання кешу вершин, покращення локальності вибірки вершинних даних, спрощення геометрії та побудови мешлетів [8]. Це особливо важливо для сучасного рендеринг-конвеєра, де геометрія часто групується в компактні кластери для ефективнішої обробки на GPU.

Ще одним важливим елементом реалізації є службовий інтерфейс для налагодження та контролю параметрів рушія. Під час розробки необхідно оперативно змінювати параметри рендерингу, переглядати статистику кадру, аналізувати стан ресурсів, тестувати окремі підсистеми та вмикати різні режими візуалізації. Без такого інтерфейсу ці дії довелося б виконувати через зміну коду, перекомпіляцію або виведення діагностичних даних у текстовому вигляді, що суттєво уповільнює розробку.

Для створення такого інтерфейсу використовується Dear ImGui [9]. Ця бібліотека працює за принципом негайного режиму, коли елементи інтерфейсу формуються безпосередньо під час побудови поточного кадру. Такий підхід

добре підходить для внутрішніх інструментів рушія, оскільки дозволяє швидко додавати панелі налагодження, перемикачі режимів, графіки продуктивності та перегляд службових буферів без розробки окремої повноцінної UI-системи.

2.5 Висновки до розділу

Даний розділ обґрунтовує вибір основних засобів реалізації графічного рушія реального часу. Для графічної підсистеми обрано Vulkan, оскільки він забезпечує низькорівневий контроль над GPU-ресурсами, підтримує графічні та обчислювальні задачі, використовує SPIR-V як проміжне представлення шейдерних програм і не прив'язує архітектуру рушія до однієї програмної екосистеми. Попри вищу складність реалізації, Vulkan дає змогу самостійно організовувати рендеринг-конвеєр, керувати синхронізацією та визначати структуру графічної підсистеми.

Основною мовою програмування визначено C++, оскільки вона поєднує високу продуктивність, контроль над пам'яттю та засоби побудови складної програмної архітектури. Порівняно з C вона краще підходить для реалізації великих підсистем рушія, порівняно з C# не залежить від керованого середовища виконання, а порівняно з Rust має зрілішу екосистему для графічного програмування. Для шейдерної частини обрано Slang, що підтримує модульну організацію коду, HLSL-подібний синтаксис, автоматичне диференціювання та компіляцію до SPIR-V.

Також визначено допоміжні бібліотеки й інструменти: AMD FidelityFX Single Pass Downsampler для ефективного формування MIP-рівнів, meshoptimizer для оптимізації геометричних даних і підготовки мешлетів, а також Dear ImGui для створення службового інтерфейсу налагодження. У сукупності ці засоби створюють основу для реалізації рушія, у якому робота з GPU-ресурсами, шейдерним кодом, геометрією та інструментами налагодження організовується в межах єдиної продуктивної архітектури.

					ІАЛЦ.467800.004 ПЗ	Лист
Зм	Лист	№ докум.	Підп.	Дата		27

3. АРХІТЕКТУРНІ ОСОБЛИВОСТІ ГРАФІЧНОГО РУШІЯ

3.1 Архітектура GPU-керованого рендерингу

Архітектура рендерингу визначає спосіб, у який дані сцени перетворюються у фінальне зображення. Вона охоплює організацію геометрії, доступ до ресурсів, вибір видимих об'єктів, виконання шейдерних програм, формування проміжних буферів і передавання результатів між окремими етапами графічного конвеєра.

Основна ідея GPU-керованого рендерингу полягає в перенесенні частини рішень щодо подальшої обробки сцени з центрального процесора на графічний процесор. У такій архітектурі GPU не лише виконує шейдерні обчислення, а й формує проміжні дані, які визначають наступні етапи рендеринг-конвеєра: списки видимої геометрії, лічильники елементів, діапазони обробки та аргументи непрямих викликів. Завдяки цьому CPU виконує переважно роль координатора проходів, тоді як обсяг подальшої роботи визначається результатами попередніх GPU-обчислень.

3.1.1 Графічний конвеєр на основі Mesh Shading

Графічний конвеєр – фундаментальна концепція комп'ютерної графіки. На високому рівні його можна розглядати як послідовність етапів, що перетворюють вхідні дані на кінцеве зображення. У випадку рендерингу такими вхідними даними зазвичай є геометрична інформація, зокрема вершини та індекси, а також текстурні дані, тоді як вихідними даними є сформоване зображення, яке відображається на екрані.

Традиційний графічний конвеєр растеризації включає низку фіксованих і програмованих стадій (рис. 3.1). У мінімальній конфігурації він складається зі стадій Input Assembler, Vertex Shader, Rasterizer та Fragment Shader.

Зм	Лист	№ докум.	Підп.	Дата

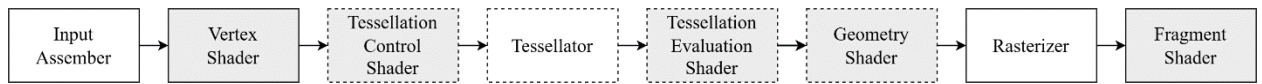


Рисунок 3.1 – Традиційний графічний конвеєр

Основні стадії традиційного конвеєра виконують такі функції:

- Input Assembler: стадія з фіксованою функцією, яка зчитує вершинні та індексні дані з буферів і формує з них примітиви відповідно до заданої топології. На цьому етапі визначається базовий порядок подання геометрії для подальшої обробки;
- Vertex Shader: програмована стадія, що виконує обробку окремих вершин. Зазвичай на цьому етапі здійснюються трансформації координат, передавання нормалей, тангентів, текстурних координат та інших атрибутів. Основним обмеженням цієї стадії є модель «одна вхідна вершина – одна вихідна вершина»;
- Rasterizer: стадія з фіксованою функцією, яка перетворює геометричні примітиви у фрагменти. Вона виконує проєкцію у координати екрана, відсікання, визначення покриття пікселів та інтерполяцію атрибутів між вершинами;
- Fragment Shader: програмована стадія, що обчислює параметри фрагмента перед записом у цільові буфери. На цьому етапі зазвичай виконуються вибірка текстур, обчислення освітлення, застосування матеріальних параметрів і формування кольору пікселя.

Окрім базових стадій, традиційний конвеєр може включати стадії теселяції та геометричний шейдер. Вони застосовуються у випадках, коли необхідно динамічно змінювати форму геометрії, генерувати нові примітиви або змінювати вже наявні під час рендерингу. Tessellation Control Shader відповідає за підготовку параметрів патча для подальшої теселяції. Tessellator виконує підрозбиття вхідного примітива на дрібніші елементи у нормованому просторі. Tessellation Evaluation Shader обчислює положення нових вершин на

основі результатів теселяції. Geometry Shader працює на рівні примітива та дає змогу генерувати, модифікувати або відкидати вершини перед растеризацією.

Попри широке використання, традиційний конвеєр має низку архітектурних обмежень. Його стадії є жорстко визначеними, а можливість керувати обробкою геометрії обмежується рамками окремих шейдерних етапів. Зокрема, класична схема не дає достатньої гнучкості для групової обробки компактних блоків геометрії, ефективного відсікання на рівні малих кластерів трикутників або ручного керування формуванням вихідних примітивів. Це стає особливо важливим у сценах із великою кількістю дрібних об'єктів або складною геометрією, де продуктивність залежить не тільки від кількості трикутників, а й від способу їх організації перед растеризацією.

Для подолання цих обмежень у розробленому русії використовується Mesh Shading [10]. Цей підхід замінює традиційні стадії обробки геометрії, що передують растеризації, на більш гнучку програмовану модель. Замість Input Assembler, Vertex Shader, стадій теселяції та Geometry Shader використовуються Task Shader і Mesh Shader, як показано на рисунку 3.2.

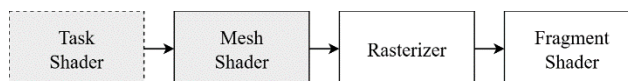


Рисунок 3.2 – Графічний конвеєр на основі Mesh Shading

Task Shader і Mesh Shader за моделлю виконання наближені до обчислювальних шейдерів. Вони працюють групами потоків, можуть використовувати спільну пам'ять і дозволяють програмно керувати формуванням геометрії. Водночас ці стадії залишаються частиною графічного конвеєра, тому їхній результат може безпосередньо передаватися у растеризатор. Це робить Mesh Shading зручним підходом для GPU-керованої архітектури, у якій підготовка геометрії виконується без надмірної участі CPU.

Task Shader може використовуватися для попередньої обробки груп геометрії. На цій стадії доцільно виконувати відсікання мешлетів, визначати

кількість робочих груп, формувати дані для подальшої обробки та передавати у Mesh Shader лише ті блоки геометрії, які потенційно можуть вплинути на фінальне зображення. Таким чином, частина невидимої або непотрібної геометрії може бути відкинута ще до детальної обробки вершин і трикутників.

Mesh Shader – основна стадія формування геометрії в цьому конвеєрі: він зчитує дані мешлета, обробляє вершини, формує примітиви та передає їх у растеризатор. Керування отримує ціла робоча група потоків – одні обчислюють позиції вершин, інші паралельно записують індекси примітивів. Це дає змогу процедурно генерувати або відкидати геометрію прямо всередині шейдера, без зайвих прогонів конвеєра.

Базовою структурною одиницею Mesh Shading є мешлет – обмежений набір просторово та топологічно близьких вершин і трикутників. Замість того щоб обробляти модель як суцільний потік окремих примітивів, геометрія поділяється на невеликі незалежні частини, придатні для паралельної обробки на GPU. Практична ефективність мешлетів пов'язана з тим, що просторово та топологічно суміжні трикутники групуються в межах одного блоку, що підвищує повторне використання вершин, покращує локальність доступу до даних і спрощує виконання відсікання на рівні груп геометрії. Якщо мешлет повністю не потрапляє в область видимості або перекритий іншою геометрією, його можна відкинути як єдиний блок, не виконуючи повну обробку всіх його примітивів.

Для досягнення максимальної продуктивності у рушії встановлено обмеження розміру мешлета – 64 вершини та 126 трикутників. Такі параметри є практично доцільними для сучасних графічних процесорів, оскільки дозволяють ефективно заповнювати робочі групи й водночас не перевищувати апаратні обмеження на кількість вихідних вершин і примітивів [11]. Занадто великі мешлети можуть зменшувати ефективність паралельної обробки та ускладнювати відсікання, оскільки в одному блоці накопичується забагато геометрії. Занадто малі мешлети, навпаки, збільшують кількість робочих груп і

можуть призводити до додаткових накладних витрат. Результат поділу геометрії сцени на мешлети наведено на рисунку 3.3. Різними кольорами позначено окремі групи трикутників, які можуть оброблятися незалежно одна від одної.



Рисунок 3.3 – Візуалізація поділу геометрії сцени на мешлети

3.1.2 Організація доступу до ресурсів

Ефективна організація доступу до ресурсів є важливою складовою рендерингу. Під час формування кадру шейдери постійно звертаються до ресурсів і параметрів матеріалів, тому спосіб організації таких звернень безпосередньо впливає на кількість змін стану та накладні витрати підготовки рендерингу. Базова модель роботи з ресурсами у Vulkan ґрунтується на використанні дескрипторних наборів. Дескриптор описує ресурс, доступний шейдеру, наприклад текстуру, семплер, буфер або зображення. Такі дескриптори об'єднуються в набори, які прив'язуються до графічного або обчислювального конвеєра перед виконанням відповідних команд. Порівняно з API попередніх поколінь ця модель є гнучкішою, однак вона все одно передбачає участь центрального процесора у підготовці та зміні прив'язок ресурсів.

У контексті GPU-керованого рендерингу класична схема дескрипторних наборів має кілька суттєвих обмежень:

- Ресурси мають бути заздалегідь згруповані у дескрипторні набори, що ускладнює динамічний вибір текстур, буферів і зображень безпосередньо в шейдері;
- Групування об'єктів часто доводиться виконувати за ознакою спільних ресурсів, що ускладнює організацію відмалювання сцен із великою кількістю унікальних матеріалів;
- Структура доступу до ресурсів жорстко задається layout конвеєра, тому додавання нових типів ресурсів або варіантів матеріалів вимагає зміни відповідних конфігурацій;
- Часті зміни стану можуть створювати додаткові накладні витрати, особливо у випадку великої кількості різнорідних матеріалів.

Гнучкішу організацію доступу до ресурсів забезпечує беззв'язна модель. У такій схемі текстури та буфери розміщуються в єдиному глобальному дескрипторному просторі, а шейдери звертаються до них за числовими індексами. Це усуває потребу в переприв'язуванні дескрипторних наборів для кожної окремої групи об'єктів або матеріалів. CPU відповідає за формування та оновлення таблиць ресурсів, тоді як вибір конкретного ресурсу виконується вже на GPU під час виконання шейдера. Реалізація такої моделі ґрунтується на індексації дескрипторів із підтримкою нерівномірного доступу до ресурсів. Індиксація дескрипторів дає змогу створювати великі масиви дескрипторів і звертатися до їхніх елементів безпосередньо в шейдерному коді. Нерівномірна індексація потрібна у випадках, коли різні потоки в межах однієї групи виконання можуть звертатися до різних ресурсів.

Для уніфікації доступу до різних типів GPU-ресурсів у рушії додатково застосовується розширення `VK_EXT_mutable_descriptor_type`. Воно дозволяє для одного зв'язування визначати кілька допустимих типів дескрипторів, замість створення окремих конфігурацій для кожної категорії ресурсів. У

поєднанні з індексацією дескрипторів це спрощує побудову глобальної таблиці ресурсів, зменшує кількість окремих layout-конфігурацій і робить систему доступу до ресурсів гнучкішою для подальшого розширення рушія.

Щоб така схема працювала, дані сцени мають бути організовані так, щоб шейдери могли за індексом екземпляра послідовно отримати інформацію про меш, матеріал і пов'язані з ним ресурси. Тому в рушії сцена подана як набір взаємопов'язаних індексованих масивів (рис. 3.4).

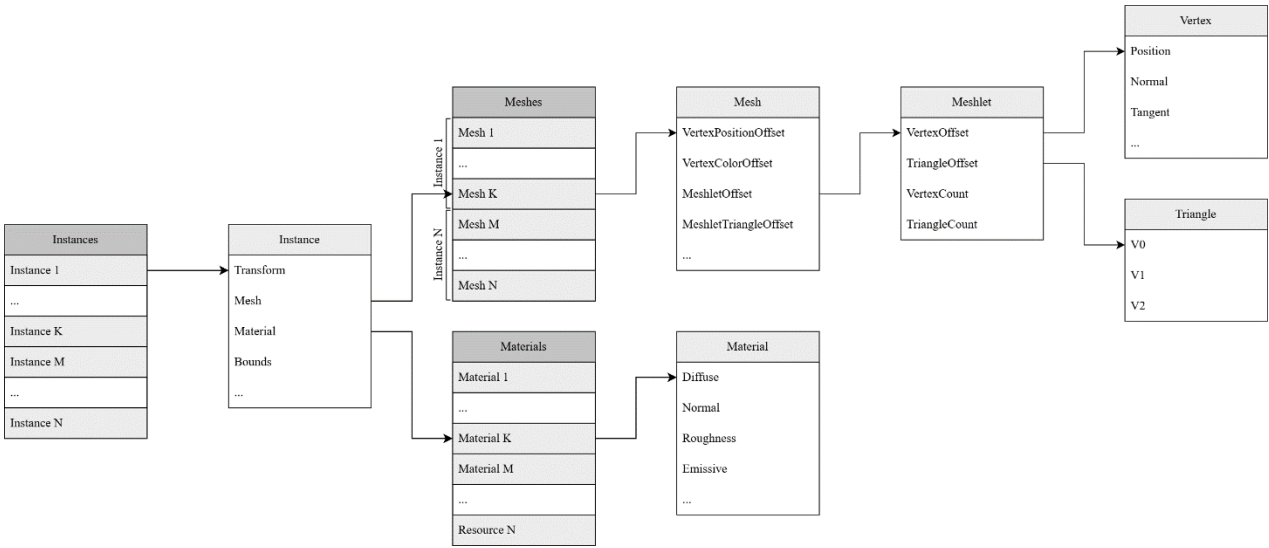


Рисунок 3.4 – Структура даних сцени та зв'язки між її компонентами

Кожен елемент масиву Instances посилається на структуру Instance, у якій зберігаються матриця трансформації, індекс меша, індекс матеріалу, дані про обмежувальні об'єми та інші параметри, необхідні для рендерингу. Такий поділ дозволяє відокремити дані конкретного екземпляра від геометрії та матеріалу, що спрощує повторне використання однакових моделей із різними трансформаціями або параметрами поверхні.

Матеріальні дані винесено в окремий масив Materials. Кожен запис Material описує параметри поверхні, зокрема базовий колір, коефіцієнти шорсткості та металічності, а також містить індекси дескрипторів текстур, необхідних для подальшого затінення.

Геометрія сцени зберігається через масив Meshes, де кожен запис Mesh задає зміщення до відповідних ділянок уніфікованих буферів: позицій вершин, атрибутів, мешлетів, індексів трикутників та інших допоміжних даних. Сам меш поділяється на набір мешлетів, а структура Meshlet визначає локальні зміщення до списків вершин і трикутників та їхню кількість. На нижчому рівні використовуються структури Vertex і Triangle: перша містить атрибути вершини, а друга задає індекси вершин, з яких формується примітив.

Під час завантаження моделі рушій переносить її геометричні дані до уніфікованих буферів і створює відповідні записи Mesh та Meshlet. Якщо матеріал використовує текстури або інші GPU-ресурси, їхні дескриптори розміщуються у глобальній таблиці, а отримані індекси записуються в структуру Material. У результаті під час рендерингу шейдери можуть знаходити потрібні геометричні та матеріальні дані без переприв'язування ресурсів з боку CPU (рис. 3.5).

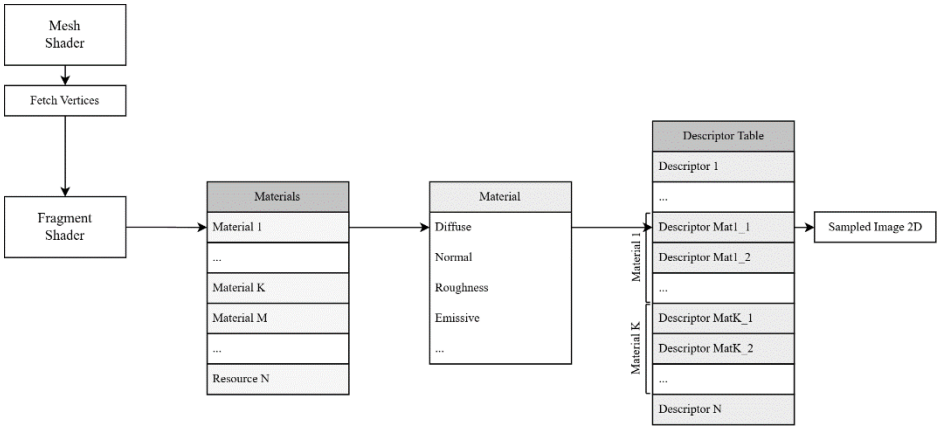


Рисунок 3.5 – Схема беззв’язного доступу до ресурсів на GPU

Завдяки такій організації відпадає потреба формувати окремі batch-групи об’єктів лише за ознакою спільного набору текстур або матеріалів. Шейдери працюють із єдиним дескрипторним простором і можуть динамічно вибирати потрібні ресурси для кожного екземпляра або фрагмента. Це зменшує кількість

змін стану, скорочує залежність від процесорної підготовки кадру та дозволяє ефективніше відтворювати сцени з великою кількістю унікальних об'єктів.

3.2 GPU-кероване відсікання геометрії

Відсікання невидимої геометрії є однією з ключових технік рендерингу в реальному часі, оскільки дає змогу зменшити обсяг обчислень ще до етапу растеризації. Його застосування особливо важливе у сценах із великою кількістю екземплярів, мешлетів і трикутників, де обробка навіть потенційно невидимих об'єктів створює зайве навантаження на GPU. У розробленому русії для цього використовується двоетапний підхід до оклюзійного відсікання [12], загальну структуру якого показано на рисунку 3.6.

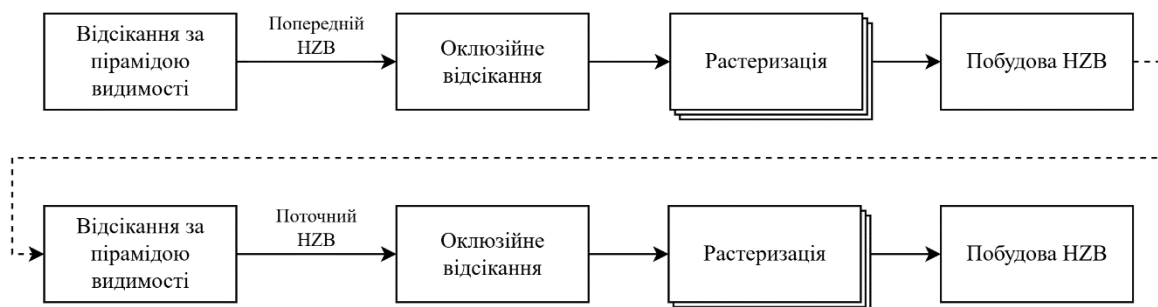


Рисунок 3.6 – Схема двоетапного оклюзійного відсікання

Такий підхід використовує часову когерентність сцени: у більшості випадків положення камери, об'єктів і характер їхнього перекриття змінюються поступово, тому дані про глибину з попереднього кадру можна використати для первинної оцінки видимості без очікування повного формування буфера глибини поточного кадру.

На першому проході виконується відсікання екземплярів і мешлетів за пірамідою видимості та оклюзійна перевірка. Геометрія, яка проходить ці перевірки, передається на растеризацію. Після цього на основі отриманого буфера глибини формується проміжний ієрархічний буфер глибини поточного

кадру. На другому проході проміжний буфер глибини поточного кадру використовується для повторної перевірки геометрії, яка була відсічена під час першого проходу. Це дозволяє повернути в обробку об'єкти, що могли стати видимими через рух камери, зміну положення об'єктів або оновлення глибинної інформації. Мешлети, які після повторної перевірки визнаються видимими, також передаються на растеризацію. Після завершення другого проходу формується фінальний ієрархічний буфер глибини, який використовується в наступному кадрі.

3.2.1 Відсікання за пірамідою видимості

Відсікання за пірамідою видимості є початковим етапом перевірки геометрії. На цьому рівні з подальшої обробки виключаються об'єкти, які повністю знаходяться поза областю, що потрапляє в поле зору камери (рис. 3.7). Така перевірка дозволяє швидко зменшити кількість об'єктів, для яких надалі потрібно виконувати складніші перевірки.

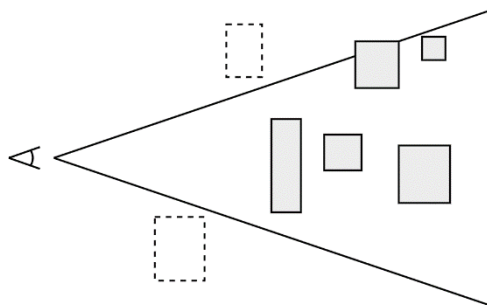


Рисунок 3.7 – Принцип відсікання за пірамідою видимості

Алгоритм перевірки видимості об'єкта реалізовано на основі його обмежувального паралелепіпеда, заданого центром і напіврозмірами (рис. 3.8). Замість аналізу всієї геометрії об'єкта достатньо перевірити вісім вершин цього об'єму. Для цього використовується потрійний цикл за координатами x , y і z , які набувають значень -1 та 1 , утворюючи всі комбінації кутів паралелепіпеда.

Кожна вершина трансформується у простір відсікання за допомогою матриці `worldViewProj`, після чого її положення подається в однорідних координатах (x, y, z, w) . У такому представленні компоненти x, y і z описують положення вершини, а w визначає масштаб, відносно якого виконується перевірка меж простору відсікання та подальший перехід до нормалізованих координат пристрою. Саме тому під час обходу вершин додатково визначаються мінімальне та максимальне значення w , які зберігаються у змінних `minW` і `maxW`.

```
for (int z = -1; z <= 1; z += 2)
    for (int y = -1; y <= 1; y += 2)
        for (int x = -1; x <= 1; x += 2)
        {
            float4 clipPos = mul(float4(center + extents *
                float3(x, y, z), 1.0f), worldViewProj);
            minW = min(minW, clipPos.w);
            maxW = max(maxW, clipPos.w);
            planesMin = min(planesMin,
                float4(clipPos.xy, -clipPos.xy) - clipPos.w);

            if (clipPos.w > 0.0f)
            {
                float3 ndcPos = clipPos.xyz / clipPos.w;
                rectMin = min(rectMin, ndcPos);
                rectMax = max(rectMax, ndcPos);
            }
        }

float minZ = maxW * zScaleOffset.x + zScaleOffset.y;
float maxZ = minW * zScaleOffset.x + zScaleOffset.y;
isCulled = (maxW <= minZ) || (maxZ <= 0.0f) || any(planesMin > 0.0f);
```

Рисунок 3.8 – Фрагмент шейдера відсікання за пірамідою видимості

Для бічних площин піраміди видимості використовуються стандартні умови простору відсікання:

$$\begin{aligned} -w &\leq x \leq w, \\ -w &\leq y \leq w, \end{aligned} \quad (3.1)$$

де w – однорідна координата вершини;

x, y – координати вершини у просторі відсікання.

На основі цих нерівностей формується набір перевірок для лівої, правої, верхньої та нижньої площин. У шейдері для цього використовується змінна `planesMin`, яка накопичує мінімальні значення відповідних виразів для всіх восьми вершин обмежувального об'єму. Якщо після обходу всіх вершин для певної площини відповідне значення залишається додатним, це означає, що об'єкт повністю розташований поза цією площиною.

Додатково аналізується положення об'єкта по глибині. На основі значень w та параметрів матриці проєкції обчислюються граничні значення $\min Z$ та $\max Z$, які використовуються для перевірки відносно ближньої та дальньої площин видимості. Після цього оновлюються межі прямокутника `rectMin` і `rectMax`, які задають мінімальні та максимальні координати об'єкта у просторі NDC. Кожен прямокутник охоплює проєкцію відповідного об'єкта (рис. 3.9) та використовується як компактна апроксимація його екранної області.

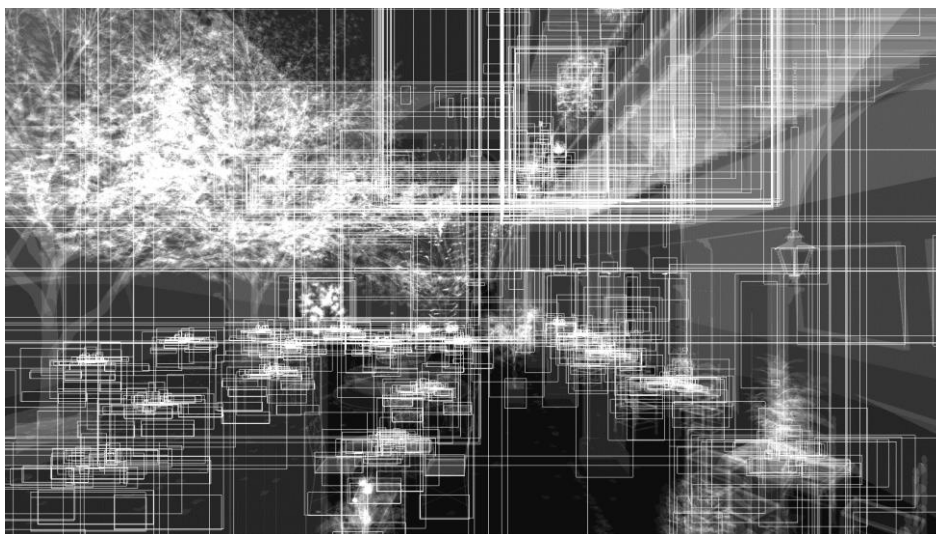


Рисунок 3.9 – Обмежувальні прямокутники екземплярів у екранному просторі

3.2.2 Оклюзійне відсікання

Після відсікання за пірамідою видимості в обробці залишаються лише ті об'єкти та мешлети, які мають екранну проєкцію. Проте це ще не означає, що вони реально видимі у кадрі. Частина геометрії може повністю перекриватися

ближчими поверхнями сцени, тому її передавання на растеризацію створює зайві витрати. Для усунення таких випадків використовується оклюзійне відсікання – перевірка видимості за вже наявною інформацією про глибину сцени (рис. 3.10).

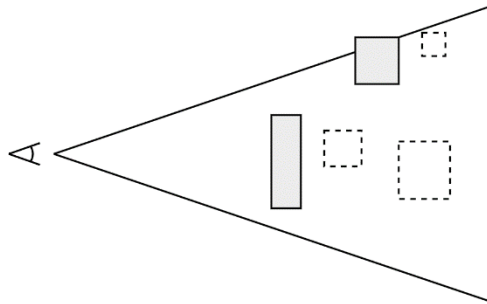


Рисунок 3.10 – Принцип оклюзійного відсікання

Основою перевірки є ієрархічний буфер глибини [13]. Він будується на основі повнорозмірного буфера глибини та містить набір MIP-рівнів зі спадною роздільною здатністю. У розробленому русії формування такого буфера виконується за один прохід із використанням AMD FidelityFX Single Pass Downsampler, розглянутого в підрозділі 2.4. Приклад перших рівнів HZB наведено на рисунку 3.11.



Рисунок 3.11 – Ієрархічне представлення буфера глибини

Кожен наступний рівень ієрархічного буфера глибини агрегує блоки 2×2 текселів попереднього рівня. На відміну від звичайних MIP-рівнів кольорових текстур, для глибини використовується не усереднення, а редукція за мінімальним значенням. За умови зворотної Z-буферизації це значення відповідає найбільш віддаленій поверхні в межах відповідного блока. Така побудова забезпечує консервативність перевірки: геометрія відсікається лише тоді, коли вся її екранна область перекрита вже наявними поверхнями.

Процедура перевірки (рис. 3.12) базується на виборі такого MIP-рівня, на якому проєкція обмежувального прямокутника об'єкта покриває область розміром не більше 4×4 текселів. Це досягається шляхом обчислення різниці координат та аналізу старшого значущого біта за допомогою функції `firstbithigh`. Використання надто грубого рівня дає змогу виконати перевірку швидше, проте через консервативний характер побудови HZB це часто призводить до хибного визначення прихованих об'єктів як видимих. Навпаки, використання детальнішого рівня підвищує точність, але потребує більшої кількості текселів для повного покриття обмежувального прямокутника.

```
int2 mipLevelXY = firstbithigh(rectPixels.zw - rectPixels.xy);
int mipLevel = max(max(mipLevelXY.x, mipLevelXY.y) - 1, 0);

int2 texelDelta = (rectPixels.zw >> mipLevel) - (rectPixels.xy >> mipLevel);
if (any(texelDelta > 3))
    mipLevel++;

rectPixels >=> mipLevel;
float2 texelSize = rcp(hzbSize) * (1u << mipLevel);
float4 xCoords = (min(rectPixels.x + int4(0, 1, 2, 3), rectPixels.z)
    + 0.5f) * texelSize.x;
float4 yCoords = (min(rectPixels.y + int4(0, 1, 2, 3), rectPixels.w)
    + 0.5f) * texelSize.y;

for (int y = 0; y < 4; ++y)
    for (int x = 0; x < 4; ++x)
    {
        minZ = min(minZ, GetHZBTexture().SampleLevel(HZBSampler,
            float2(xCoords[x], yCoords[y]), mipLevel));
    }

isOccluded = minZ > rectMax.z;
```

Рисунок 3.12 – Фрагмент шейдера оклюзійного відсікання

Серед вибраних значень визначається $\min Z$. За зворотної Z -буферизації це найбільш віддалене значення серед потенційних оклюдерів у перевірюваній області. Далі воно порівнюється з rectMax.z , тобто з найближчою до камери глибиною обмежувального об'єму. Критерій оклюзійного відсікання для такого випадку можна записати у вигляді:

$$\min Z > \text{rectMax.z}, \quad (3.2)$$

де $\min Z$ – мінімальне значення глибини в обраній області;

rectMax.z – найбільше значення глибини серед проєктованих вершин обмежувального об'єму.

Виконання умови означає, що для відповідної екранної області в HZB вже зафіксовано значення глибини, яке відповідає оклюдеру, розташованому ближче до камери, ніж обмежувальний об'єм об'єкта, що перевіряється. За таких умов об'єкт класифікується як повністю оклюдований і виключається з подальшої обробки на етапі рендерингу.

3.2.3 Відсікання трикутників

Після відсікання екземплярів і мешлетів додаткова оптимізація виконується на рівні окремих трикутників. Цей етап потрібний для того, щоб зменшити кількість примітивів, які передаються в растеризатор після обробки mesh-шейдера. Навіть якщо мешлет загалом є видимим, частина його трикутників може не впливати на фінальне зображення.

На цьому рівні використовуються такі перевірки:

- відсікання задніх граней;
- відсікання вироджених трикутників;
- відсікання трикутників, що повністю виходять за межі екрана;
- відсікання субпіксельних трикутників.

Першою виконується перевірка орієнтації трикутника (рис. 3.13). Для цього за координатами його вершин у просторі відсікання обчислюється

визначник, знак якого дозволяє встановити, чи є примітив фронтальним або належить до задніх граней [14]. Якщо матеріал не є двостороннім, трикутники, орієнтовані від камери, відсікаються. Вироджені примітиви з нульовою площею також виключаються, оскільки не створюють корисного внеску в зображення.

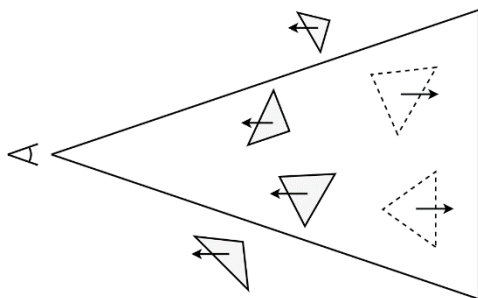


Рисунок 3.13 – Принцип відсікання задніх граней трикутника

Для трикутників, що залишилися після перевірки орієнтації, виконується перехід до нормалізованого екранного простору. На основі трьох вершин обчислюється двовимірний обмежувальний прямокутник. Якщо цей прямокутник повністю розташований поза межами екрана, трикутник не може бути растеризований у поточному кадрі і відсікається.

Завершальною виконується перевірка субпіксельних трикутників, екранна проєкція яких є меншою за один піксель або наближається до цього розміру (рис. 3.14). Для цього обмежувальний прямокутник масштабується до розмірів вікна перегляду, після чого порівнюються округлені мінімальні та максимальні координати [15]. Якщо вони збігаються хоча б по одній осі, трикутник вважається субпіксельним і не передається на растеризацію.

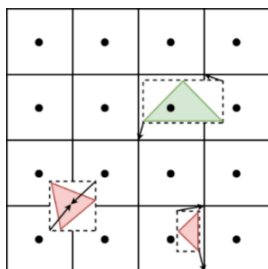


Рисунок 3.14 – Принцип відсікання субпіксельних трикутників

Зм	Лист	№ докум.	Підп.	Дата

ІАЛЦ.467800.004 ПЗ

Реалізацію описаних перевірок наведено на рисунку 3.15. У межах цього етапу кожен потік шейдера обробляє окремий трикутник. Для цього зі структури Instance (підрозділ 3.1.2) зчитується відповідна матриця перетворення, після чого вершини примітива трансформуються. Отримані координати (pc0, pc1, pc2) використовуються для послідовного виконання всіх критеріїв відсікання.

```
float det = determinant(float3x3(pc0, pc1, pc2));
isCulled = (det >= 0) && (!IsTwoSided() || det == 0);

if (!isCulled)
{
    float2 ps0 = (pc0.xy / pc0.w) * 0.5f + 0.5f;
    float2 ps1 = (pc1.xy / pc1.w) * 0.5f + 0.5f;
    float2 ps2 = (pc2.xy / pc2.w) * 0.5f + 0.5f;

    minAABB = min(ps0, min(ps1, ps2));
    maxAABB = max(ps0, max(ps1, ps2));
    isCulled = any(minAABB >= 1.0f) || any(maxAABB <= 0.0f);
}

if (!isCulled)
{
    minAABB *= GetViewportSize();
    maxAABB *= GetViewportSize();
    isCulled = any(round(minAABB) == round(maxAABB));
}
```

Рисунок 3.15 – Фрагмент шейдера відсікання трикутників

3.2.4 Розподіл мешлетів

Після завершення етапів відсікання формується набір видимих мешлетів. Однак для подальшої растеризації цей набір потрібно додатково впорядкувати, щоб мешлети з однаковими параметрами обробки могли виконуватися спільно. Тому перед запуском графічного проходу виконується їх класифікація за типом растеризації, який визначає режим обробки геометрії під час відмалювання. Переважна кількість сцен має такі основні категорії:

- Opaque: непрозора геометрія;
- Masked: геометрія з альфа-тестуванням.

Водночас набір типів растеризації не є фіксованим і може розширюватися залежно від вмісту сцени: використаних матеріалів, шейдерних варіантів і режимів обробки, необхідних для відображення конкретних моделей.

Першим виконується етап класифікації мешлетів (рис. 3.16). На цьому кроці для кожного видимого мешлета визначається ідентифікатор відповідної групи на основі даних екземпляра та пов'язаного з ним матеріалу. Після цього обчислюється кількість мешлетів у кожній групі. Для зменшення кількості атомарних звернень до глобальної пам'яті використовується хвильова агрегація [16]. Потоки в межах однієї хвилі, для яких визначено однакову групу, об'єднуються, а сумарна кількість таких мешлетів додається до відповідного глобального лічильника однією атомарною операцією. У результаті кількість атомарних оновлень зменшується з рівня окремих мешлетів до рівня унікальних груп усередині хвилі.

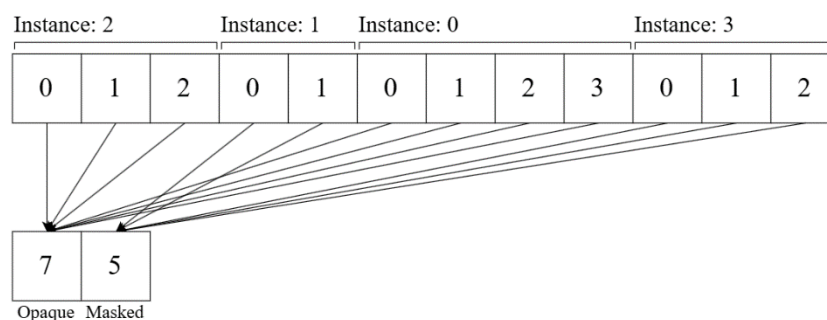


Рисунок 3.16 – Схема класифікації мешлетів

Після завершення класифікації виконується резервування діапазонів для кожної групи та ініціалізація даних для подальшого непрямого запуску (рис. 3.17). Для кожної групи зчитується кількість належних їй мешлетів, після чого за допомогою префіксної суми визначається початкове зміщення у спільному буфері індексів. Таким чином кожна група отримує власний безперервний діапазон пам'яті, у який надалі будуть записані індекси відповідних мешлетів. Одночасно створюється запис із параметрами mesh-шейдерного проходу: лічильником мешлетів, кількістю груп по осях Y та Z і зміщенням у буфері

індексів. Під час виконання команда `vkCmdDrawMeshTasksIndirectEXT` зчитує ці аргументи безпосередньо з GPU-буфера, тому подальша растеризація запускається на основі результатів попередніх GPU-проходів без повторної підготовки команд на CPU.

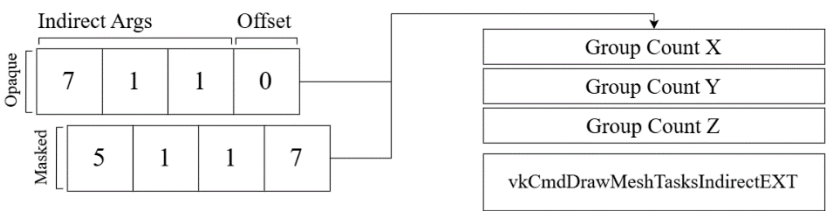


Рисунок 3.17 – Схема формування аргументів для непрямих викликів

Завершальним підготовчим етапом є розподіл індексів мешлетів за зарезервованими діапазонами (рис. 3.18). На цьому кроці кожен потік повторно визначає групу для свого мешлета, обчислює локальне зміщення всередині відповідного діапазону та записує індекс у виділену позицію. Як і на етапі підрахунку, тут використовується хвильова агрегація: для кожної унікальної групи в межах хвилі резервується суцільний піддіапазон, а конкретне положення мешлета в ньому визначається за допомогою внутрішньохвильових префіксних операцій.

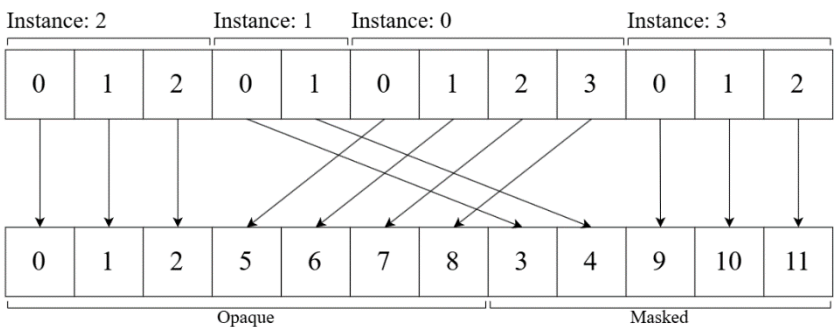


Рисунок 3.18 – Схема розподілу індексів мешлетів

У результаті формується впорядкований масив, у якому мешлети з однаковим режимом обробки розташовуються послідовно. Це забезпечує

компактне подання видимої геометрії та дає змогу виконувати подальшу растеризацію окремо для кожної групи без додаткового сортування. Під час запуску непрямих викликів кожна група обробляє власний безперервний діапазон мешлетів, а зміна параметрів матеріалу та режиму обробки відбувається лише між такими групами, а не між окремими елементами. Приклад результату такого розподілу наведено на рисунку 3.19, де показано геометрію, згруповану відповідно до типу растеризації.



Рисунок 3.19 – Результат класифікації геометрії за типом растеризації

3.3 Відкладений рендеринг

Рендеринг – це процес перетворення опису тривимірної сцени на двовимірне зображення, яке відображається на екрані. Під час цього процесу враховуються геометрія об’єктів, властивості матеріалів, положення камери, джерела освітлення та інші параметри, що визначають вигляд кадру. Спосіб організації рендерингу впливає на структуру графічного конвеєра, кількість шейдерних обчислень, обсяг проміжних буферів і зручність подальшого застосування ефектів освітлення та постобробки.

Найпростішим підходом є прямий (forward) рендеринг, за якого об’єкти безпосередньо передаються на графічний конвеєр, а обчислення матеріалу й

освітлення виконуються під час растеризації. Така схема є простою в реалізації та добре узгоджується з апаратним тестом глибини, однак має істотний недолік: значна частина обчислень виконується ще до остаточного визначення видимості фрагмента. У складних сценах це призводить до значних витрат на фрагментному етапі, особливо за наявності дорогих матеріалів і великої кількості джерел світла.

Подальшим розвитком цього підходу став forward+, у якому екран поділяється на плитки, а для кожної з них попередньо формується список джерел світла. Це зменшує кількість зайвих світлових обчислень, оскільки фрагментний шейдер працює не з усіма джерелами сцени, а лише з тими, що можуть впливати на відповідну ділянку екрана. Проте основна особливість прямого рендерингу зберігається: формування параметрів поверхні та обчислення освітлення все одно виконуються під час растеризації геометрії.

Альтернативою є відкладений (deferred) рендеринг. Його основна ідея полягає в розділенні геометричного етапу та етапу освітлення, що, на відміну від прямого рендерингу, дозволяє суттєво зменшити кількість надлишкових розрахунків і забезпечує гнучку підтримку довільної кількості джерел світла без повторного обходу геометрії. У класичному варіанті такий підхід складається з двох основних проходів:

- Перший прохід записує інформацію про геометрію поверхні, зокрема позицію, нормалі та інші атрибути, у кілька буферів кадру, які зазвичай називають G-буфером;
- Другий прохід виконується після завершення геометричного етапу для всієї сцени, зчитує збережені дані з G-буфера та використовує їх для обчислення освітлення.

Такий поділ дає змогу виконувати дорогі обчислення лише для видимих пікселів. Водночас класичний відкладений рендеринг має і недоліки: формування та зчитування G-буфера потребують високої пропускної здатності пам'яті, а частина записаних у нього фрагментів може бути перекрита іншими.

Частково це компенсується попереднім проходом глибини, але ціною повторної растеризації всієї геометрії сцени.

З огляду на ці обмеження в роботі використано відкладений підхід на основі буфера видимості [17]. Його відмінність від класичного G-буфера полягає в тому, що на геометричному етапі не зберігається повний набір атрибутів поверхні. Замість цього фіксується лише інформація, достатня для подальшого визначення видимого примітива. Така організація зменшує обсяг записів під час растеризації та переносить відновлення атрибутів, обчислення параметрів матеріалу й затінення на окремі відкладені проходи.

3.3.1 Формування буфера видимості

Практична доцільність буфера видимості особливо помітна під час рендерингу дрібної геометрії. Сучасні GPU зазвичай обробляють фрагменти групами розміром 2×2 пікселі, оскільки така організація потрібна для обчислення похідних у фрагментному шейдері. Якщо трикутник має малу екранну проєкцію і не покриває всю таку групу, частина потоків виконується як допоміжна: вони беруть участь у формуванні похідних, але не записують результат у фінальне зображення. У разі виконання текстурних вибірок, обчислення матеріалу або освітлення безпосередньо на цьому етапі частина роботи витрачається неефективно. Тому геометричний прохід у цьому підході має виконувати мінімальну кількість операцій.

Буфер видимості можна розглядати як розширення попереднього проходу глибини. Звичайний depth pre-pass визначає лише найближчий фрагмент за глибиною, тоді як буфер видимості додатково зберігає посилання на конкретний примітив. Завдяки цьому після завершення растеризації для кожного пікселя відомо не лише те, що певна поверхня є найближчою до камери, а й те, який саме трикутник її сформував.

Запис, використаний у реалізації, складається з індексу мешлета та локального індексу трикутника в його межах. Разом ці значення однозначно задають примітив, для якого в наступних проходах відновлюються вершинні дані та атрибути. Завдяки малому обсягу інформації такий запис можна упакувати в одну 32-бітну змінну (рис. 3.20). За прийнятого обмеження на максимальну кількість трикутників у мешлеті для локального індексу достатньо 7 бітів, тоді як решта 25 бітів відводиться під індекс мешлета.

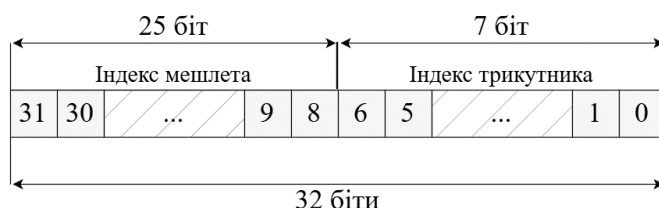
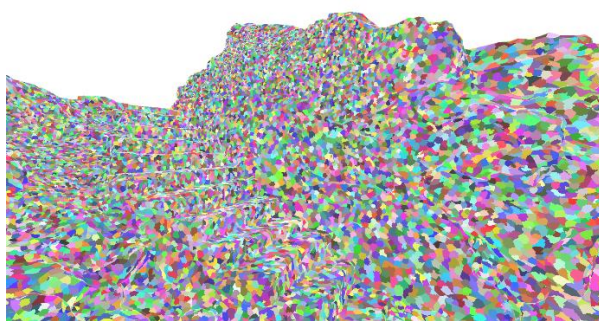
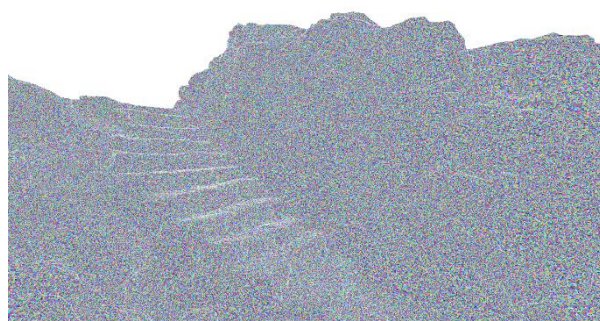


Рисунок 3.20 – Формат даних у буфері видимості

Приклад вмісту буфера видимості наведено на рисунку 3.21, де значення індексу мешлета та локального індексу трикутника для видимих трикутників сцени відображено різними кольорами.



а)



б)

Рисунок 3.21 – Візуалізація буфера видимості: а) індекс мешлета; б) індекс трикутника

3.3.2 Відкладена інтерполяція атрибутів

Після формування буфера видимості виконується відновлення атрибутів видимого трикутника для кожного пікселя. Оскільки в буфері зберігаються лише індекс мешлета та локальний індекс трикутника, барицентричні координати, їхні похідні та інші атрибути мають бути відновлені на цій основі. Математично цей етап ґрунтується на барицентричній інтерполяції. Якщо у вершинах трикутника задано деякий атрибут a_1, a_2, a_3 , то в афінному випадку його значення в точці (x, y) визначається як:

$$a(x, y) = \lambda_1(x, y)a_1 + \lambda_2(x, y)a_2 + \lambda_3(x, y)a_3, \quad (3.3)$$

де $\lambda_1(x, y), \lambda_2(x, y), \lambda_3(x, y)$ – барицентричні координати точки (x, y) , причому:

$$\lambda_1 + \lambda_2 + \lambda_3 = 1$$

Для трикутника з вершинами $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ в екранному просторі барицентричні координати є лінійними функціями координат точки (x, y) , тому їхні частинні похідні за x та y є сталими в межах трикутника:

$$\frac{\partial \lambda_1}{\partial x} = \frac{y_2 - y_3}{D}, \quad \frac{\partial \lambda_2}{\partial x} = \frac{y_3 - y_1}{D}, \quad \frac{\partial \lambda_3}{\partial x} = \frac{y_1 - y_2}{D}, \quad (3.4)$$

$$\frac{\partial \lambda_1}{\partial y} = \frac{x_3 - x_2}{D}, \quad \frac{\partial \lambda_2}{\partial y} = \frac{x_1 - x_3}{D}, \quad \frac{\partial \lambda_3}{\partial y} = \frac{x_2 - x_1}{D}, \quad (3.5)$$

$$D = (x_1 - x_3)(y_2 - y_3) - (x_2 - x_3)(y_1 - y_3), \quad (3.6)$$

де λ_i – барицентрична координата точки;

x_i, y_i – координати i -ї вершини трикутника в екранному просторі;

D – визначник, пропорційний подвоєній площі трикутника.

За перспективного проєктування для відновлення атрибутів застосовується перспективно-коректна інтерполяція. На відміну від афінного випадку, перспективно-коректні барицентричні координати є нелінійними, тому їхні похідні не є сталими й обчислюються аналітично за правилом диференціювання частки [18]. При цьому похідні афінних барицентричних координат, наведені у формулах (3.4) та (3.5), використовуються як проміжні величини. Такий підхід дає змогу отримати точні похідні без звернення до

апаратних інструкцій ddx/ddy , що є важливим для коректного визначення рівня деталізації текстур і забезпечення якісної фільтрації.

Після обчислення барицентричних координат та їх похідних виконується відновлення атрибутів видимого трикутника, зокрема текстурних координат, нормалі, дотичного вектора та позиції. На основі цих даних обчислюються параметри матеріалу, а отримані характеристики поверхні записуються до G-буфера для подальшого проходу освітлення.

3.4 Трасування променів

Одним із важливих напрямів розвитку сучасних графічних рушіїв є використання трасування променів як засобу розв'язання задач видимості та обчислення окремих складових освітлення. Якщо в попередніх підрозділах було розглянуто побудову GPU-керованого растеризаційного конвеєра, організацію відсікання геометрії та формування даних для відкладеного рендерингу, то на цьому етапі розглядається підхід, який доповнює растеризацію у задачах, де потрібна точніша перевірка просторових взаємодій між об'єктами.

Необхідність використання трасування променів у межах розробленого рушія зумовлена тим, що суто растеризаційні методи мають природні обмеження під час розрахунку ефектів, пов'язаних із видимістю. У растеризаційному конвеєрі інформація про сцену зазвичай обмежується тією геометрією, яка безпосередньо бере участь у формуванні поточного кадру, або тими даними, що були збережені у проміжних буферах. Через це реалізація відображень, оклюзії та інших подібних ефектів часто спирається на наближені методи, які працюють лише з частковою інформацією про сцену. Такий підхід є достатньо ефективним з точки зору швидкодії, однак не завжди забезпечує потрібний рівень точності та може призводити до появи характерних артефактів.

Трасування променів дає змогу виконувати перевірку відносно просторового представлення сцени, а не лише її екранного зображення. Це робить можливим формування точніших просторових запитів у тих підсистемах рушія, де важливо визначити наявність або відсутність перешкод між двома точками простору. Завдяки цьому трасування променів є доцільним насамперед для реалізації тіней та інших ефектів освітлення, для яких коректне визначення видимості є одним із ключових етапів обчислення.

3.4.1 Керування прискорювальними структурами

Для ефективного використання трасування променів геометрія сцени має бути подана у вигляді структури, придатної до швидкого обходу під час пошуку перетинів. Пряме тестування кожного променя з усіма трикутниками сцени є обчислювально не вигідним, тому використовується ієрархія обмежувальних об'ємів BVH, представлена засобами прискорювальних структур Vulkan. На відміну від екранних буферів растеризаційного конвеєра, така структура описує просторову організацію сцени й не обмежується лише геометрією, видимою в поточному кадрі. Це дозволяє виконувати запити видимості з довільної точки простору та в довільному напрямку.

Сцена для трасування променів будується на двох рівнях:

- BLAS описує геометрію окремого меша;
- TLAS містить екземпляри об'єктів сцени та посилання на відповідні BLAS.

Побудова BLAS виконується на основі трикутникової геометрії з використанням вершинного та індексного буферів. Для кожного меша формується опис геометрії, який містить формат вершин, адреси вершинних та індексних даних, крок між вершинами, тип індексів і кількість трикутників. Далі через `vkGetAccelerationStructureBuildSizesKHR` визначаються розміри цільового буфера структури та scratch-буфера, після чого створюються

відповідні GPU-ресурси і виконується команда `vkCmdBuildAccelerationStructuresKHR`.

Для статичної геометрії BLAS будується з прапорцями `PREFER_FAST_TRACE` та `ALLOW_COMPACTTION`. Перший орієнтує побудову на швидший обхід структури під час трасування променів, другий дозволяє після завершення побудови виконати окремий етап компактизації. Така конфігурація використовується для неанімованих мешів, оскільки вони будуються один раз і надалі багаторазово використовуються в запитах трасування.

У випадку анімованої геометрії використовується конфігурація `PREFER_FAST_BUILD` разом із `ALLOW_UPDATE`. Прапорець `ALLOW_UPDATE` дає змогу виконувати операцію рефіттингу – швидкого оновлення обмежувальних об’ємів BVH без повної реструктуризації дерева. Це важливо для об’єктів, положення вершин яких змінюється від кадру до кадру внаслідок анімації або скіннінгу. На відміну від растеризації, де деформація меша може застосовуватися безпосередньо під час проходження візуалізації, побудова BLAS потребує вже підготовлених позицій вершин, придатних для формування структури прискорення.

Структура верхнього рівня, TLAS, перебудовується динамічно, оскільки має відповідати поточному складу сцени та актуальним трансформаціям об’єктів. Це дозволяє враховувати переміщення, обертання, зміну кількості екземплярів і зв’язки з відповідними BLAS. Для TLAS використовується прапорець `PREFER_FAST_TRACE`, оскільки саме ця структура бере участь у всіх подальших запитах трасування променів. На цьому етапі створюються цільовий буфер, scratch-буфер і сама структура верхнього рівня, після чого виконується команда `vkCmdBuildAccelerationStructuresKHR`. Після завершення побудови встановлюються бар’єри між станами запису та читання BVH, що забезпечує коректну синхронізацію перед використанням прискорювальних структур у шейдерах трасування променів.

Окремим етапом є компактизація BLAS, яка дає змогу зменшити обсяг пам'яті, зайнятий статичними прискорювальними структурами. Після побудови початкова BLAS може містити надлишкові службові дані, необхідні лише під час створення структури.

Процес компактизації BLAS реалізовано як пакетну асинхронну операцію (рис. 3.22). Структури, для яких дозволено стискання, додаються до черги, з якої формується пакет поточної обробки обмеженого розміру. Це дозволяє виконувати компактизацію поступово й розподіляти пов'язані з нею операції на кілька проходів. У результаті зменшується навантаження на окремий кадр і знижується ймовірність різких піків часу виконання.

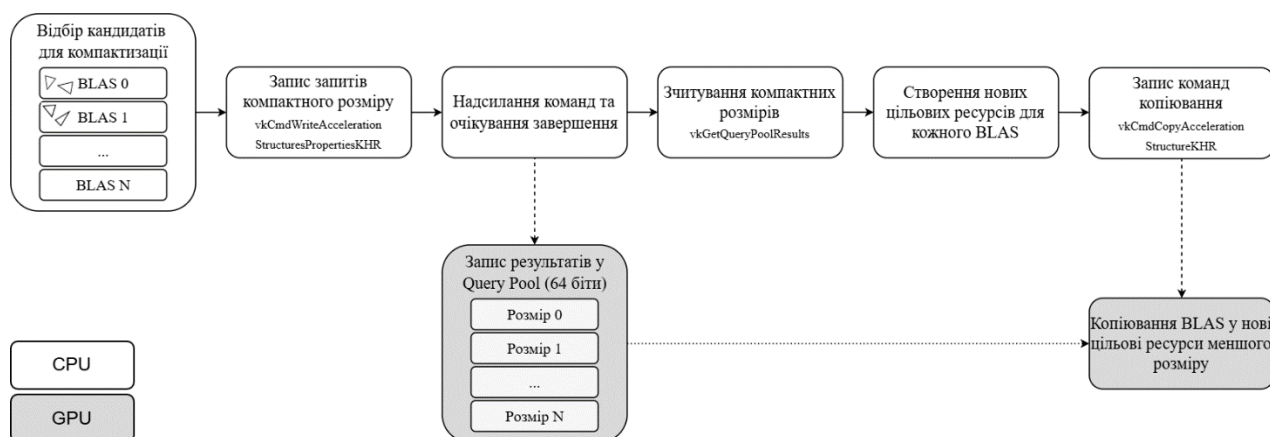


Рисунок 3.22 – Послідовність компактизації BLAS

Після копіювання початкові BLAS замінюються компактними версіями без зміни геометричного вмісту. Для шейдерів трасування це не змінює логіку доступу до сцени, однак зменшує обсяг відеопам'яті, необхідний для зберігання прискорювальних структур.

Завершальним етапом є створення дескриптора TLAS, який записується у глобальну таблицю дескрипторів і надалі використовується в шейдерах трасування променів. Це інтегрує TLAS у загальну схему беззв'язного доступу до ресурсів і дозволяє звертатися до неї за тією самою логікою, що й до інших ресурсів рушія.

3.4.2 Реалізація тіней від джерел світла

Використання карт тіней у класичних алгоритмах растеризації традиційно залишається одним із проблемних місць, оскільки якість результату тут суттєво залежить від роздільної здатності карти тіней, параметрів проєкції та вибору зміщення глибини. Це призводить до характерних артефактів, зокрема shadow acne, peter panning, східчастості меж тіні та нестабільності зображення під час руху камери. Карти тіней є одним із найстаріших прийомів у графіці реального часу, тому їх заміна більш коректним і фізично обґрунтованим методом визначення видимості дає суттєві переваги.

Реалізація трасованих тіней у найпростішій, жорсткій формі є досить прямолінійною: від поверхні в напрямку джерела світла випускається тіньовий промінь, і якщо промінь перетинає геометрію сцени, поверхня опиняється в тіні. У розробленій системі цей підхід спирається на буфер глибини, сформований під час геометричного проходу, для відновлення положення поверхні у світовому просторі. Відновлена позиція використовується як початок тіньового променя, а напрямок визначається положенням джерела світла відносно цієї точки. Результат побудови жорстких тіней наведено на рисунку 3.23.



Рисунок 3.23 – Жорсткі тіні, побудовані методом трасування променів

Зм	Лист	№ докум.	Підп.	Дата

ІАЛЦ.467800.004 ПЗ

Використання лише жорстких тіней не дозволяє досягти високого рівня реалізму, оскільки в реальних умовах практично не існує ідеально точкових джерел світла. Відсутність напівтіней позбавляє зображення глибини та створює неприродно різкі межі, які не відповідають фізичним властивостям світлових потоків. З цієї причини м'які тіні реалізуються шляхом випуску променів у формі конуса, орієнтованого на джерело світла. Такий підхід дозволяє моделювати площинний характер джерела і формувати область часткового затінення.

Для формування напрямку тіньового променя використовується сферичне параметричне подання. Вектор напрямку ω задається як:

$$\omega = (\sin \theta \cos \phi, \sin \theta \sin \phi, \cos \theta), \quad (3.7)$$

де θ – полярний кут;

ϕ – азимутальний кут.

Параметри θ та ϕ визначаються за співвідношеннями:

$$\begin{aligned} \cos \theta &= (1 - \xi_1) + \xi_1 \cos \theta_{max}, \\ \sin \theta &= \sqrt{1 - \cos^2 \theta}, \\ \phi &= 2\pi \xi_2, \end{aligned} \quad (3.8)$$

де $\xi_1, \xi_2 \in [0,1]$ – незалежні випадкові величини;

θ_{max} – максимальний кут розкриття конуса відносно його центральної осі.

Для величин ξ_1 і ξ_2 використовується просторово-часовий синій шум. Просторово-часовий синій шум, як і послідовності з низькою розбіжністю, спрямований на рівномірніше покриття області семплювання, але додатково має спектральні властивості, завдяки яким шум візуально менш помітний [19]. Це контрастує з білим шумом, якому властиві скупчення та порожнечі. Скупчення точок призводять до надлишковості даних, а порожні ділянки – до їх нестачі.

Формування якісних вибірок синього шуму безпосередньо під час виконання шейдера потребує додаткових обчислювальних витрат. Тому у реалізації використовуються заздалегідь підготовлені текстури масок синього

шуму, які під час рендерингу зчитуються як константні дані. Такий підхід дає змогу винести складнішу генерацію семплів на попередній етап, де можна приділити більше уваги їх якості, не збільшуючи суттєво вартість обробки кадру в реальному часі.

Ключові переваги просторово-часового синього шуму:

- Зниження кореляції між сусідніми пікселями порівняно з білим шумом та послідовностями з низькою розбіжністю;
- Формування темпорально узгодженої послідовності семплів для одного пікселя;
- Швидша збіжність результату при накопиченні семплів у часі порівняно з незалежними масками синього шуму;
- Зменшення мерехтіння та різких міжкадрових змін тіньового сигналу в динамічних сценах;
- Можливість коректного перезапуску темпорального накопичення без втрати якості послідовності завдяки тороїдальній прогресивності по часовій осі.

Коли від точки поверхні випускаються промені в різні точки джерела світла, на краях тіні починає проявлятися шум. Він виникає саме в області напівтіні, оскільки в повністю освітлених ділянках і в області повної тіні промені дають узгоджений результат щодо видимості джерела. Розмір напівтіні залежить від розміру та відстані до джерела світла, а також від розміру та положення об'єкта, що відкидає тінь. Чим ширший кут конуса, тим м'якшою буде напівтінь, але тим вищим буде рівень шуму. Зменшення цього шуму шляхом простого збільшення кількості променів швидко призводить до зростання вартості обчислень, тому в системі застосовується подальша фільтрація тіньового сигналу. Приклад рівномірного семплювання напрямків у межах конуса при одному промені на піксель із використанням просторово-часового синього шуму наведено на рисунку 3.24.



Рисунок 3.24 – Результат рівномірного семплювання в конусі при 1 RPP із використанням просторово-часового синього шуму

Для фільтрації м'яких тіней у системі використовується просторово-часовий денойзер, що об'єднує класифікацію областей, темпоральне накопичення та багатопрохідну просторову фільтрацію. За принципом роботи він близький до сучасних підходів, однак адаптований до швидкої обробки тіньового сигналу в задачі трасування променів. Це дозволяє зменшити шум, зберігаючи межі тіней і знижуючи рівень часових артефактів.

На початковому етапі виконується класифікація областей, у ході якої визначаються пікселі, що не потребують денойзингу, а також однорідні ділянки, для яких результат може бути записаний без повної фільтрації. Таким чином зменшується кількість зайвих обчислень і обробка зосереджується лише на тих областях, де тіньовий сигнал є шумним або неоднорідним.

Для решти пікселів застосовується темпоральна стабілізація з перепроєкцією історії та перевіркою її валідності. Якщо накопичені дані не відповідають поточному пікселю, вони відкидаються. Для коректно зіставлених пікселів оновлюються накопичені значення тіні, оцінка шуму та довжина історії, що забезпечує ефективне шумоподавлення між кадрами. Перепроєктована історія додатково обмежується на основі локальної

статистики сусідніх пікселів, знижуючи накопичення хибних значень і послаблюючи прояви шлейфів [20].

Після темпоральної обробки виконується кілька проходів просторової фільтрації. На кожному з них використовується двобічний фільтр, ваги якого визначаються просторовою близькістю, значенням тіні, глибиною та нормаллю поверхні. Завдяки цьому згладжування відбувається в межах однорідних областей і не призводить до значного розмивання меж тіні. Оцінка шуму використовується для керування інтенсивністю фільтрації, що дозволяє сильніше згладжувати зашумлені ділянки та зберігати деталі у стабільних областях. Послідовне виконання кількох проходів дає змогу усувати шум на різних просторових масштабах, а на завершальному етапі формується фінальне значення тіні (рис. 3.25).



Рисунок 3.25 – М'які тіні після застосування денойзера

3.5 Нейромережеве навколишнє затінення

Після етапу трасованих тіней у конвеєрі розглядається додатковий спосіб підсилення локальної форми поверхонь. На відміну від попереднього підрозділу, де оброблялась видимість конкретного джерела світла, тут використовується оцінка локального оточення пікселя.

3.5.1 Основи навколишнього затінення

Навколишнє затінення використовується для імітації ефекту, за якого ділянки поверхні, частково закриті іншою геометрією, отримують менше розсіяного світла з навколишнього середовища. Найбільш помітно цей ефект проявляється у заглибинах, кутах, місцях контакту об'єктів та на дрібних деталях геометрії. Без такого затемнення сцена може виглядати пласкою, навіть якщо основне освітлення та матеріали обчислюються коректно.

Фізично навколишнє затінення пов'язане з тим, що не всі напрямки над точкою поверхні є відкритими для освітлення. Якщо значна частина півсфери над поверхнею перекрита іншими об'єктами, точка отримує менше розсіяного світла і виглядає темнішою. Навпаки, відкриті ділянки поверхні мають вищий коефіцієнт доступності та освітлюються сильніше. Математично коефіцієнт навколишнього затінення для точки на поверхні можна подати через інтегрування видимості над півсферою напрямків:

$$A(p, n) = \frac{1}{\pi} \int_{\Omega} V(p, \omega) (n \cdot \omega) d\omega, \quad (3.9)$$

де p – точка на поверхні;

n – нормаль поверхні в точці p ;

Ω – півсфера напрямків над поверхнею;

ω – напрямок у межах півсфери;

$V(p, \omega)$ – функція видимості в напрямку ω ;

$(n \cdot \omega)$ – ваговий множник згідно із законом косинусів Ламберта.

Функція видимості $V(p, \omega)$ повертає значення 1, якщо напрямок ω не перекритий геометрією, і 0, якщо він заблокований. Множник $(n \cdot \omega)$ враховує орієнтацію поверхні та надає більшої ваги напрямкам, ближчим до нормалі n . Нормувальний коефіцієнт $\frac{1}{\pi}$ приводить результат до діапазону $[0; 1]$.

Повне обчислення такого інтеграла для кожного пікселя є занадто дорогим для рендерингу в реальному часі, тому на практиці застосовуються

наближені методи. Найпоширенішими наближеними методами є екранно-просторові алгоритми, зокрема SSAO, які оцінюють ступінь перекриття півсфери за даними буфера глибини. Такі методи добре інтегруються у відкладені конвеєри рендерингу та мають передбачувану обчислювальну вартість, однак не забезпечують високої фізичної точності.

Методи на основі трасування променів можуть давати точніший результат, оскільки працюють із геометрією сцени поза межами екранного простору. Проте їхня вартість залишається високою, особливо для застосувань, де потрібно підтримувати стабільну частоту кадрів. Через це в даному проєкті використовується нейромережевий підхід, який апроксимує значення навколишнього затінення на основі набору ознак, сформованих із буфера глибини, нормалей поверхні та локальної геометричної конфігурації сцени.

3.5.2 Архітектура нейронної мережі та прискорення обчислень

Для апроксимації навколишнього затінення використано багатошаровий перцептрон (рис. 3.26). Такий тип нейронної мережі є зручним для задач, у яких потрібно відновити нелінійне відображення між вхідними ознаками та одним або кількома вихідними значеннями. У цьому випадку вхідний вектор описує локальне оточення поточного пікселя, а виходом мережі є коефіцієнт навколишнього затінення.

Багатошаровий перцептрон є мережею прямого поширення. Дані проходять послідовно від вхідного шару через приховані шари до вихідного. Кожен шар виконує лінійне перетворення вхідного вектора, після чого застосовується нелінійна функція активації:

$$y_i = f(W_i x_i + b_i), \quad (3.10)$$

де y_i – вихід i -го шару;

x_i – вхідний вектор i -го шару;

W_i – матриця ваг шару;

b_i – вектор зміщень (bias);

f – функція активації.

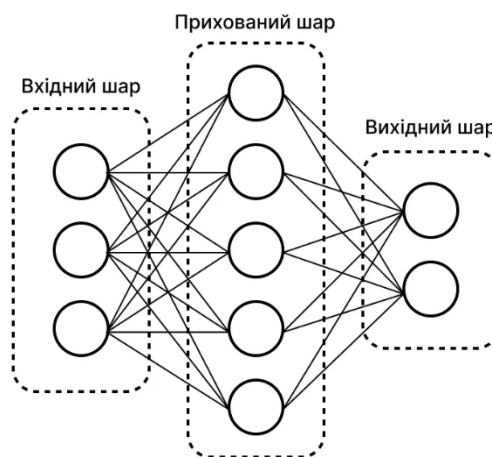


Рисунок 3.26 – Схематична модель багатошарового перцептрона

Вихід одного шару передається на вхід наступного, тому повне обчислення мережі зводиться до послідовності операцій множення вектора на матрицю, додавання зміщення та застосування нелінійності. Основними параметрами архітектури є розмір вхідного вектора, кількість прихованих шарів, кількість нейронів у кожному прихованому шарі та розмір вихідного шару.

Звичайне обчислення багатошарового перцептрона в шейдері є досить дорогим, оскільки кожен піксель потребує окремого прямого проходу через мережу. У задачі нейромережевого навколишнього затінення це особливо важливо: для кожного пікселя формується власний вхідний вектор ознак, але послідовність шарів MLP залишається однаковою. Для зменшення вартості такого інференсу використовується підхід RTX Neural Shading, що дозволяє розміщувати малі нейронні моделі безпосередньо в шейдерних програмах.

Апаратне прискорення забезпечується механізмом кооперативних векторів (Cooperative Vectors), реалізованим у Vulkan через розширення `VK_NV_cooperative_vector`. Воно орієнтоване на ефективне виконання векторно-матричних операцій, які є основою шарів MLP, і дає змогу:

- Спільно використовувати дані між сусідніми потоками у межах однієї підгрупи;
- Задіяти спеціалізовані апаратні блоки GPU для прискорення векторно-матричних операцій [21];
- Зменшити кількість проміжних звернень до пам'яті за рахунок кращої організації обчислень.

Додатковою перевагою є підтримка форматів зниженої точності, зокрема FP16. Для невеликої MLP-моделі, що прогнозує один коефіцієнт навколишнього затінення, такої точності достатньо, оскільки результат не потребує повного 32-бітного представлення на кожному проміжному етапі. Використання FP16 зменшує навантаження на регістри, кеш-пам'ять і пропускну здатність пам'яті графічного адаптера. У результаті модель може виконуватися як частина основного проходу затінення, не створюючи надмірної затримки між формуванням вхідних ознак і отриманням коефіцієнта затінення.

3.5.3 Процес навчання

Навчання нейронної мережі виконується до її використання в рушії. На цьому етапі підбираються ваги та зміщення MLP, які дозволяють за вхідними ознаками передбачати коефіцієнт навколишнього затінення. Після завершення навчання параметри фіксуються, а під час рендерингу виконується лише прямий прохід мережі.

Процес навчання складається з таких основних етапів:

1. Ініціалізація параметрів мережі;
2. Обчислення прогнозу під час прямого проходу;
3. Порівняння прогнозу з еталонним значенням за допомогою функції втрат;
4. Обчислення градієнтів методом зворотного поширення;

Зм	Лист	№ докум.	Підп.	Дата

ІАЛЦ.467800.004 ПЗ

5. Оновлення ваг і зміщень оптимізатором;
6. Повторення кроків 2-5 для навчальних пакетів протягом заданої кількості епох.

Основним джерелом геометричної інформації для формування вхідних ознак є буфер глибини. Нормаль поверхні реконструюється за глибиною, а не зчитується безпосередньо з G-буфера, оскільки вона може містити результат рельєфного текстуровання і відображати модифіковану орієнтацію поверхні, призначену для освітлення, а не фактичну геометричну форму сцени.

Вхідні ознаки поділяються на дві групи:

1. Глобальні ознаки, які описують центральний піксель і його найближче оточення;
2. Напрямкові ознаки, які характеризують локальну геометрію вздовж кількох напрямків вибірки.

До глобальних ознак належать параметри, що задають масштаб ефекту та положення поверхні відносно камери: радіус навколишнього затінення в екранному просторі, співвідношення глибини до радіуса вибірки та кут між нормальною поверхні й напрямком погляду. Додатково аналізуються значення глибини в сусідніх пікселях. На їх основі обчислюються різниці глибини та ознаки неперервності, які допомагають визначити локальний нахил поверхні, виявляти різкі межі геометрії та зменшувати помилкове затінення поблизу розривів глибини.

Для кожного напрямку виконується серія вибірок глибини в межах заданого радіуса навколо поточного пікселя. Положення вибірок зміщується, що зменшує регулярність шаблону семплювання. Після зчитування глибини вибрані точки відновлюються у просторі камери, і для них обчислюється просторове зміщення відносно центральної точки поверхні. На основі цих значень визначаються відстань до вибірок, зміщення вздовж напрямку погляду та оцінка можливого перекриття геометрією в цьому напрямку. Додатково враховується орієнтація нормалі відносно площини вибірки: для цього нормаль

проектується на відповідну площину, після чого оцінюються її кутове положення та довжина проєкції. Такі ознаки дають мережі інформацію про те, як розташована сусідня геометрія відносно поточної точки та наскільки вона може впливати на значення навколишнього затінення.

Для прихованих шарів використовується функція активації ReLU:

$$f(x) = \max(0, x), \quad (3.11)$$

де x – вхідне значення нейрона.

ReLU має просту форму та низьку обчислювальну вартість. Вона пропускає додатні значення без змін, а від’ємні замінює нулем. Така нелінійність добре підходить для глибоких мереж, оскільки сприяє ефективнішому поширенню градієнтів порівняно з насичуваними функціями активації.

Вибір функції активації впливає на спосіб ініціалізації ваг мережі. Ініціалізація Ксав’є Глорота добре узгоджується з симетричними активаційними функціями, однак для ReLU вона є менш придатною, оскільки ReLU занулює від’ємну частину сигналу та змінює його дисперсію під час проходження через шари. Тому для прихованих шарів було використано ініціалізацію Каймінга Хе, запропоновану для мереж із rectifier-активаціями. Це дозволяє узгодити початковий масштаб ваг із фактичною поведінкою ReLU, яка пропускає лише додатну частину вхідного сигналу.

Основна ідея цього методу полягає у виборі такого масштабу початкових ваг, який зберігає дисперсію сигналу під час прямого та зворотного проходів. Це зменшує ризик згасання або надмірного зростання активацій і градієнтів на початкових етапах навчання. Стандартне відхилення для ініціалізації визначається як:

$$\sigma_{He} = \sqrt{\frac{2}{N_{l-1}}}, \quad (3.12)$$

де N_{l-1} – кількість нейронів у попередньому шарі.

Вихідний шар використовує сигмоїдну функцію активації:

$$S(x) = \frac{1}{1 + e^{-x}}, \quad (3.13)$$

де $S(x)$ – значення вихідної активації;

x – вхідне значення функції.

Сигмоїдна функція переводить результат у діапазон $[0; 1]$, що відповідає природному діапазону коефіцієнта навколишнього затінення. Значення, близьке до 0, відповідає сильному затіненню, а значення, близьке до 1 – відкритій ділянці поверхні.

Для оцінки помилки під час навчання використовується функція втрат Губера, визначена формулою (3.14). Вона поєднує властивості середньоквадратичної та середньої абсолютної помилки: поблизу нуля поводить як квадратична функція, а для великих відхилень переходить до лінійного зростання. Це робить її стійкішою до шумових або нетипових прикладів у навчальних даних.

$$L_{\delta}(a) = \begin{cases} \frac{1}{2}a^2, & |a| \leq \delta, \\ \delta \left(|a| - \frac{1}{2}\delta \right), & |a| > \delta, \end{cases} \quad (3.14)$$

де $L_{\delta}(a)$ – значення функції втрат Губера;

$a = y - \hat{y}$ – помилка прогнозу;

y – еталонне значення;

$\hat{y}$ – прогноз мережі;

δ – пороговий параметр.

Оновлення параметрів виконується оптимізатором Adam. Він використовує згладжені оцінки першого та другого моментів градієнта, що дозволяє адаптивно масштабувати крок оптимізації для кожного параметра мережі.

Для керування швидкістю навчання використовується планувальник, що складається з трьох основних фаз: лінійного розгону, ділянки сталої швидкості та косинусного спадання. Така схема дозволяє уникнути різких змін параметрів

на початку навчання, зберегти достатню швидкість оптимізації на основній фазі та плавно стабілізувати модель наприкінці процесу.

На початковому етапі швидкість навчання поступово збільшується від нульового значення до базового. Ця фаза зменшує ризик нестабільних оновлень параметрів, які можуть виникати на перших ітераціях через випадкову ініціалізацію ваг. Після завершення розгону протягом заданої кількості кроків використовується стала базова швидкість навчання. Вона забезпечує активне зменшення функції втрат і дозволяє моделі швидко наблизитися до області оптимальних параметрів.

На завершальному етапі застосовується косинусне спадання швидкості навчання. У межах цієї фази швидкість плавно зменшується від базового значення до мінімального, що сприяє стабілізації параметрів моделі та зменшує ймовірність коливань поблизу мінімуму функції втрат. Косинусне спадання визначається за формулою:

$$\eta(t) = \eta_{\min} + (\eta_{\text{base}} - \eta_{\min}) \frac{1 + \cos(\pi r)}{2}, \quad (3.15)$$

де $\eta(t)$ – швидкість навчання на поточному кроці;

η_{base} – базова швидкість навчання;

$\eta_{\min}$ – мінімальна швидкість навчання;

r – нормалізована позиція всередині фази спадання, $r \in [0; 1]$.

Нормалізована позиція r визначає, наскільки далеко поточний крок знаходиться від початку фази спадання. На її початку $r = 0$, тому швидкість навчання дорівнює базовому значенню. Наприкінці фази $r = 1$, і швидкість досягає мінімального рівня. Після завершення косинусного спадання подальше навчання виконується з фіксованою мінімальною швидкістю.

3.6 Глобальне освітлення

Глобальне освітлення є важливою складовою фотореалістичного рендерингу, оскільки дає змогу враховувати багаторазові відбивання світла від

поверхонь сцени. Однак його точне обчислення методом трасування шляхів потребує значних обчислювальних ресурсів і тому залишається непридатним для більшості застосувань реального часу. З цієї причини традиційно використовують наближені методи.

Одним із традиційних рішень є використання попередньо запечених карт освітлення та статичних зондів опромінення. Проте такі методи мають суттєві обмеження: не підтримують коректне оновлення освітлення в динамічних сценах і не враховують оклюзію між зондом і точкою запиту, що спричиняє артефакти витоку світла. Для усунення цих недоліків у графічному рушії використано метод динамічного дифузного глобального освітлення DDGI [22].

До основних переваг методу DDGI належать:

- зменшення артефактів витоку світла;
- підтримка динамічних сцен;
- стабільність у часі;
- відсутність необхідності у складному денойзері;
- висока продуктивність у реальному часі.

Основна ідея методу полягає в тому, що простір сцени квантується регулярною тривимірною сіткою зондів, а непряме дифузне освітлення відновлюється за даними найближчих зондів з урахуванням видимості. На відміну від попередньо обчислених полів опромінення, дані зондів оновлюються під час виконання програми, що дає змогу реагувати на зміни освітлення, переміщення об'єктів і зміну конфігурації сцени.

Зонди розміщуються рівномірно в межах робочого об'єму сцени. Кількість зондів по осях X, Y та Z задається вручну залежно від розмірів сцени, її просторової структури та бажаної якості непрямого освітлення. Приклад такого розміщення у сцені Intel Sponza наведено на рисунку 3.27. Зменшення кроку між зондами підвищує точність відновлення освітлення, але одночасно збільшує витрати пам'яті та час оновлення. Тому розмірність об'єму

вибирається окремо для кожної сцени як компроміс між якістю та продуктивністю.



Рисунок 3.27 – Візуалізація об'єму зондів

Конвеєр рендерингу при використанні DDGI складається з трьох послідовних етапів:

1. Трасування стохастичних променів із зондів;
2. Оновлення карт опроміненості та відстані зондів;
3. Рендеринг сцени з використанням оновлених даних зондів.

На першому етапі з кожного активного зонда у сцену випускається N променів, де кількість N підбирається індивідуально залежно від вимог до якості та доступної продуктивності. Чим більша кількість променів використовується для одного зонда, тим точніше оцінюється локальне освітлення, однак тим більшою є вартість трасування. Напрямки цих променів генеруються за методом спіралі Фібоначчі [23], який забезпечує квазі-рівномірний розподіл напрямків на сфері (рис. 3.28). Додатково набір напрямків стохастично повертається між кадрами, що покращує охоплення сфери та зменшує регулярні артефакти вибірки. Після визначення точок перетину з геометрією в місцях влучання обчислюється освітлення, а отримані результати використовуються для оновлення даних зондів.

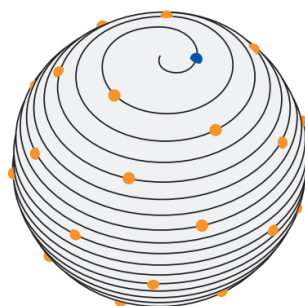


Рисунок 3.28 – Розподіл напрямків променів за методом спіралі Фібоначчі

Оскільки дані зонда мають напрямлений, тобто сферичний характер, їх потрібно відобразити у двовимірний текстурний простір. Існують різні способи параметризації одиничної сфери, зокрема сферичні карти та кубічні карти, проте октаедральне мапування забезпечує дуже вдале співвідношення між похибкою подання та обчислювальною складністю. Цей підхід ґрунтується на відображенні одиничних векторів на поверхню октаедра з подальшим проєктуванням граней на площину (рис. 3.29), що є зручним для текстурного зберігання даних на графічному процесорі.

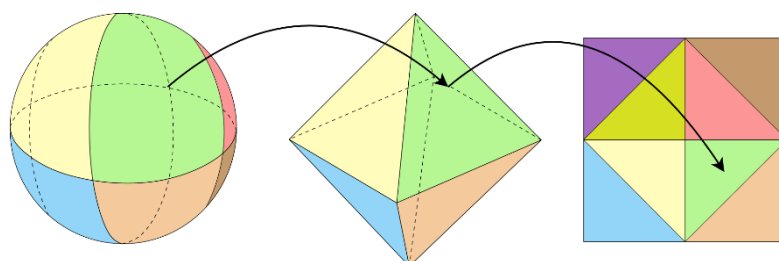


Рисунок 3.29 – Схема мапування сфери на октаедр із подальшим проєктуванням на площину

Кожен зонд подається парою текстур: картою опроміненості та картою відстані (рис. 3.30). Один зонд у карті опроміненості містить 6×6 корисних текселів, а один зонд у карті відстані – 14×14 корисних текселів. Більша роздільна здатність карти відстані є виправданою, оскільки саме вона кодує функцію видимості, для якої потрібна вища кутова точність. Зонди

впорядковуюються в атласі за тривимірними координатами сітки: горизонтальне положення визначається координатами X та Y , а вертикальне – координатою Z . Оскільки вибірка з атласу виконується з білінійною фільтрацією, навколо корисної області кожного зонда додається однопіксельна рамка. Значення в цій рамці дублюють крайові текселі октаедральної розгортки, що запобігає появі швів і чорних артефактів на її межах. Тому повний розмір області, яку займає один зонд в атласі, становить 8×8 текселів для карти опроміненості та 16×16 текселів для карти відстані.

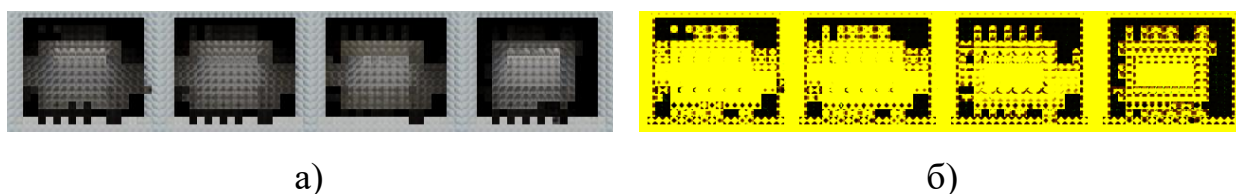


Рисунок 3.30 – Текстурний атлас даних зондів: а) опроміненість; б) відстань

Карта опроміненості зберігається у форматі `RGBA16_FLOAT`, а карта відстані – у форматі `RG16_FLOAT`. У карті відстані перший канал містить середню відстань до геометрії в певному напрямку, а другий – середнє значення квадрата відстані. Для ефективнішого використання динамічного діапазону під час зберігання опроміненості застосовується експоненціальне зважування.

Оновлення даних зондів виконується в кожному кадрі. Для карти опроміненості нове значення формується як зважена сума внесків трасованих променів. Після цього воно не записується безпосередньо, а змішується з попереднім значенням, що забезпечує часову стабільність і поступове накопичення непрямого освітлення. Аналогічно оновлюється карта відстані. Такий підхід дозволяє поступово адаптувати непряме освітлення до змін геометрії та освітлення сцени.

Ключовою відмінністю DDGI від класичних зондів опромінення є зберігання статистичної інформації про відстань до геометрії. Замість повного набору трасованих відстаней для кожного напрямку зберігаються середня

відстань та середнє значення квадрата відстані. На їх основі обчислюється дисперсія відстані:

$$\sigma^2 = E(d^2) - E(d)^2, \quad (3.16)$$

де $E(d)$ – середня відстань до геометрії;

$E(d^2)$ – середнє значення квадрата відстані.

Перед вибіркою непрямого освітлення, положення фрагмента зміщується вздовж нормалі поверхні та напрямку спостереження. Таке зміщення потрібне для зменшення самозатінення, що виникає через статистичний характер подання відстані. Записати його можна у вигляді:

$$x' = x + \beta_n n + \beta_v v, \quad (3.17)$$

де x – початкова позиція фрагмента;

n – нормаль поверхні;

v – напрямок спостереження;

x' – зміщена позиція;

β_n, β_v – коефіцієнти зміщення.

Після цього для точки визначається локальна комірка сітки, що складається з восьми сусідніх зондів, як показано на рисунку 3.31. Для кожного з цих зондів обчислюється вага, яка враховує положення точки всередині комірки, орієнтацію поверхні та статистичну видимість. Для оцінки довіри до зонда використовується тест Чебишова:

$$w = \begin{cases} 1, & r \leq E(d), \\ \frac{\sigma^2}{\left(\sigma^2 + (r - E(d))^2\right)}, & r > E(d), \end{cases} \quad (3.18)$$

де r – відстань від зонда до точки, що освітлюється;

$E(d)$ – середня відстань до геометрії в даному напрямку;

σ^2 – дисперсія відстані.

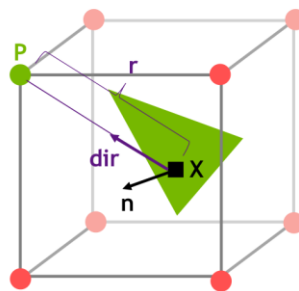


Рисунок 3.31 – Візуалізація обчислення освітлення в точці X за допомогою восьми сусідніх зондів

Чим сильніше відрізняється відстань до точки від середньої відстані до геометрії, яку «бачив» зонд, тим меншим буде внесок такого зонда. Саме цей механізм дозволяє суттєво зменшити артефакти витоку світла через непрозорі перешкоди. Після цього значення опроміненості від восьми сусідніх зондів підсумовуються та нормалізуються. Непряма опроміненість у точці X визначається як:

$$E(X) = \frac{\sum w_i E_i}{\sum w_i}, \quad (3.19)$$

де E_i – опроміненість i -го зонда;

w_i – повна вага i -го зонда.

Отриманий результат використовується для формування непрямої дифузної складової освітлення поверхні. Використання описаного підходу дозволяє досягти компромісу між фізичною точністю та обчислювальною складністю, забезпечуючи коректну візуалізацію взаємодії світла з об'єктами сцени в динаміці.

3.7 Масштабування та реконструкція зображення

Формування зображення у високій роздільній здатності суттєво підвищує навантаження на графічний процесор, оскільки вартість піксельних етапів рендерингу масштабується разом із площею буфера кадру. Під час переходу від 1920×1080 до 3840×2160 кількість пікселів зростає приблизно з 2,07 млн до 8,29

млн, тобто у чотири рази. Відповідно збільшується обсяг роботи, пов'язаний із затіненням, вибірками текстур, фільтрацією проміжних результатів та обчисленням освітлення, що може помітно зменшувати частоту кадрів.

Одним із практичних способів зменшення цього навантаження є рендеринг сцени у нижчій внутрішній роздільній здатності з подальшим відновленням зображення до цільового розміру. Просте просторове масштабування не забезпечує достатньої якості, оскільки працює лише з поточним кадром і не враховує міжкадрову інформацію. Унаслідок цього виникають розмиття контурів, втрата дрібних деталей і помітна ступінчастість на межах об'єктів. Для розв'язання цієї задачі в розробленому рушії використовується NVIDIA DLSS Super Resolution – нейромережевий метод масштабування та реконструкції зображення [24].

Етап реконструкції виконується після формування основного зображення сцени у внутрішній роздільній здатності та до фінальної постобробки. На цьому етапі рушій уже має сформовані буфери кольору, глибини та векторів руху, які використовуються для відновлення зображення у цільовій роздільній здатності. Такий підхід дозволяє виконувати найбільш затратні піксельні проходи у меншому буфері, а реконструкцію застосовувати лише до вже підготовленого проміжного результату.

До етапу реконструкції передаються колір, глибина та вектори руху. Колір є результатом рендерингу сцени у внутрішній роздільній здатності. Глибина описує просторове положення видимих поверхонь, а вектори руху визначають зміщення відповідної вибірки в екранному просторі між поточним і попереднім кадрами. Вектори руху обчислюються як різниця між положенням тієї самої точки поверхні у поточному та попередньому кадрах після її перетворення у простір проєкції. Після перспективного ділення отримується зміщення в екранних координатах, яке записується до буфера векторів руху та використовується для міжкадрового зіставлення. У реалізації вони зберігаються у форматі RG16_FLOAT, де перший канал містить горизонтальну складову

зміщення, а другий – вертикальну. Порівняно з 32-бітним представленням це зменшує обсяг пам'яті та пропускну здатність, зберігаючи достатню точність для реконструкції.

Окрім цього, аналогічно до методів часового згладжування, під час обчислення матриці проєкції застосовується субпіксельне зміщення. Його призначення полягає в тому, щоб кожен кадр рендерився з дещо іншим положенням вибірки всередині пікселя. Без такого зміщення послідовні кадри містили б майже однакову просторову інформацію. Зміщення проєкції на частину пікселя дозволяє отримувати додаткові субпіксельні вибірки сцени в часі, які надалі використовуються під час реконструкції зображення.

Для побудови послідовності змищень використовується послідовність Галтона з базами 2 та 3. Вона формує нерегулярний, але добре розподілений набір точок, що дозволяє ефективніше накопичувати субпіксельну інформацію, ніж регулярний шаблон [20]. Послідовність Галтона належить до класу низькодисперсних квазівипадкових розподілів, тобто її точки рівномірно заповнюють площу пікселя без виражених локальних скупчень. Координати поточної точки перетворюються у зміщення в нормалізованих координатах пристрою за формулою:

$$\begin{aligned}\Delta x &= \frac{2(S_x - 0.5)}{W}, \\ \Delta y &= \frac{2(S_y - 0.5)}{H},\end{aligned}\tag{3.20}$$

де $\Delta x, \Delta y$ – зміщення в просторі нормалізованих координат пристрою;

S_x, S_y – координати поточної точки послідовності Галтона;

W, H – ширина та висота внутрішнього буфера рендерингу відповідно.

На рисунку 3.32 показано перші 16 субпіксельних положень вибірки, сформованих за послідовністю Галтона з базами 2 та 3. Їх розташування демонструє поступове покриття площі пікселя без концентрації вибірок в окремих його ділянках.

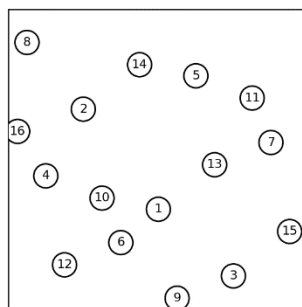


Рисунок 3.32 – Перші 16 субпіксельних позицій семплювання за послідовністю Галтона з базами 2 та 3

Кількість унікальних зміщень до повторення послідовності називається кількістю фаз. Вона залежить від співвідношення між цільовою та внутрішньою роздільною здатністю кадру і визначається за формулою:

$$Phases = 8 \cdot \left(\frac{H_t}{H_r} \right)^2, \quad (3.21)$$

де H_t – висота цільового зображення;

H_r – висота буфера рендерингу.

Квадратична залежність пояснюється тим, що зі зменшенням внутрішньої роздільної здатності один піксель буфера рендерингу відповідає більшій площі у фінальному кадрі. Тому для збереження достатнього покриття необхідно збільшувати кількість положень вибірки. Під час зміни розміру вікна або режиму якості внутрішня роздільна здатність оновлюється, після чого рушій відповідно оновлює і кількість фаз послідовності зміщень.

Якщо внутрішня роздільна здатність дорівнює цільовій, тобто масштабування становить 1:1, технологія працює в режимі DLAA. У цьому випадку вона використовується не для підвищення продуктивності, а для якісного часового згладжування, стабілізації зображення та відновлення дрібних деталей. При меншій внутрішній роздільній здатності застосовується DLSS Super Resolution, що дає змогу зменшити вартість піксельних проходів і відновити зображення до цільового розміру.

3.8 Висновки до розділу

Даний розділ формує технічну основу розробленого графічного рушія. Його архітектура побудована на GPU-керованому підході, за якого графічний процесор бере участь не лише у виконанні шейдерів, а й у підготовці даних для подальших етапів рендерингу. Центральний процесор у такій схемі переважно координує проходи та передає початкові дані, тоді як вибір видимої геометрії, групування мешлетів, формування діапазонів обробки та запуск непрямих викликів виконуються на GPU. Це зменшує накладні витрати процесорної частини та робить конвеєр придатним для сцен із великою кількістю геометрії, матеріалів і дрібних примітивів.

Запропонована структура поєднує декілька рівнів оптимізації, які послідовно зменшують обсяг зайвої роботи. Поділ моделей на мешлети дає змогу виконувати відсікання та розподіл геометрії на рівні компактних блоків, а не цілих об'єктів. Після цього буфер видимості дозволяє зберігати на геометричному етапі лише ідентифікатор видимого примітива, відкладаючи відновлення повних атрибутів до моменту, коли вже відомі потрібні пікселі. Завдяки цьому обсяг проміжних даних зменшується, а подальші етапи рендерингу виконуються на основі фактичного результату попередньої GPU-обробки.

Реалізований конвеєр також доповнюється методами, що підвищують якість фінального зображення. Трасування променів використовується для задач, де потрібна точніша перевірка просторової видимості, зокрема для м'яких тіней і динамічного глобального освітлення. Нейромережеве навколишнє затінення реалізовано на основі RTX Neural Shading, що дає змогу виконувати інференс MLP-моделі безпосередньо в шейдерному конвеєрі за ознаками з буфера глибини та локальної геометрії. Завершальним етапом є реконструкція зображення за допомогою DLSS Super Resolution, яка доповнює рушій засобами підвищення продуктивності та якості фінального кадру.

4. ТЕСТУВАННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ РОЗРОБЛЕНОГО ГРАФІЧНОГО РУШІЯ

Після реалізації основних підсистем рушія було проведено експериментальне тестування, спрямоване на перевірку його роботи в умовах складної тривимірної сцени. Основна увага приділялася тим аспектам, які безпосередньо визначають придатність рушія до рендерингу в реальному часі: швидкодії GPU-керованого конвеєра, ефективності відсікання геометрії та вартості окремих проходів формування кадру.

Для отримання точних вимірювань у рушії реалізовано власну систему профілювання продуктивності. Вона дає змогу фіксувати час виконання окремих GPU-проходів і виводити результати у службовому інтерфейсі. Завдяки цьому можна аналізувати не лише загальний час кадру, а й внесок конкретних етапів. Під час профілювання було використано стабільний режим роботи графічного процесора з фіксованою тактовою частотою. Це дозволило знизити вплив автоматичної зміни частот на результати вимірювань і зробити порівняння між окремими проходами більш коректним.

Тестування проводилося на платформі з такими характеристиками:

- операційна система: Windows 10;
- графічний процесор: NVIDIA GeForce RTX 5070 Ti;
- драйвер графічного процесора: версія 610.47.

4.1 Тестування GPU-керованого конвеєра

Для тестування було використано сцену Lumberyard Bistro, загальний вигляд якої наведено на рисунку 4.1. Ця сцена містить велику кількість об'єктів, матеріалів, архітектурних елементів, внутрішніх приміщень і ділянок зі щільним розташуванням геометрії. Вона є придатною для перевірки відсікання,

оскільки значна частина об'єктів у ній може перебувати поза межами поля зору камери або бути перекритою іншими поверхнями.

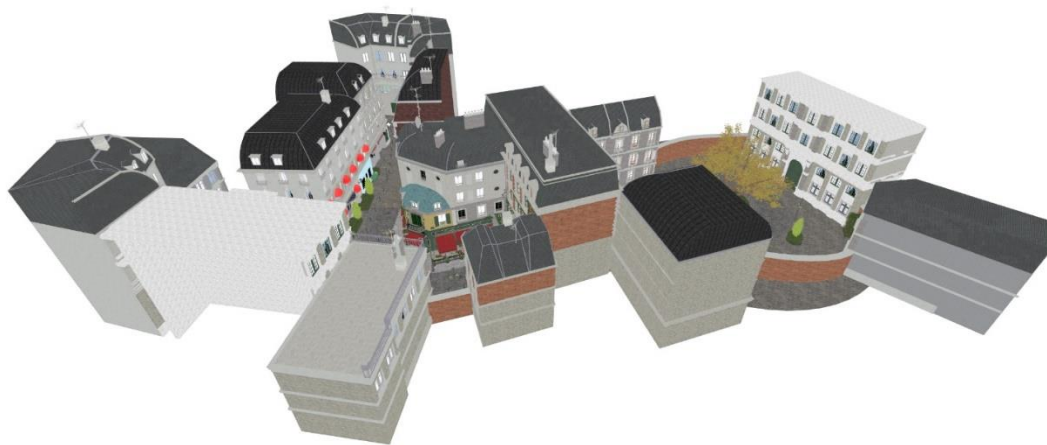


Рисунок 4.1 – Загальний вигляд тестової сцени

На рисунку 4.2 наведено результат рендерингу сцени, отриманий за допомогою розробленого графічного рушія. Рендеринг виконувався у роздільній здатності 2560×1440 , що забезпечує достатнє навантаження на графічний конвеєр і відповідає поширеному режиму роботи сучасних систем рендерингу реального часу.



Рисунок 4.2 – Результат рендерингу сцени Lumberyard Bistro

Зм	Лист	№ докум.	Підп.	Дата

ІАЛЦ.467800.004 ПЗ

Лист

80

Попередню візуальну оцінку ефективності відсікання виконано за допомогою режиму відображення перекриття пікселів. На рисунку 4.3 показано результат без використання алгоритмів відсікання, а на рисунку 4.4 – результат після їх застосування. Світліші ділянки відповідають областям, у яких один і той самий піксель багаторазово перекривається різними фрагментами геометрії. Темніші ділянки, навпаки, вказують на меншу кількість зайвої фрагментної роботи.

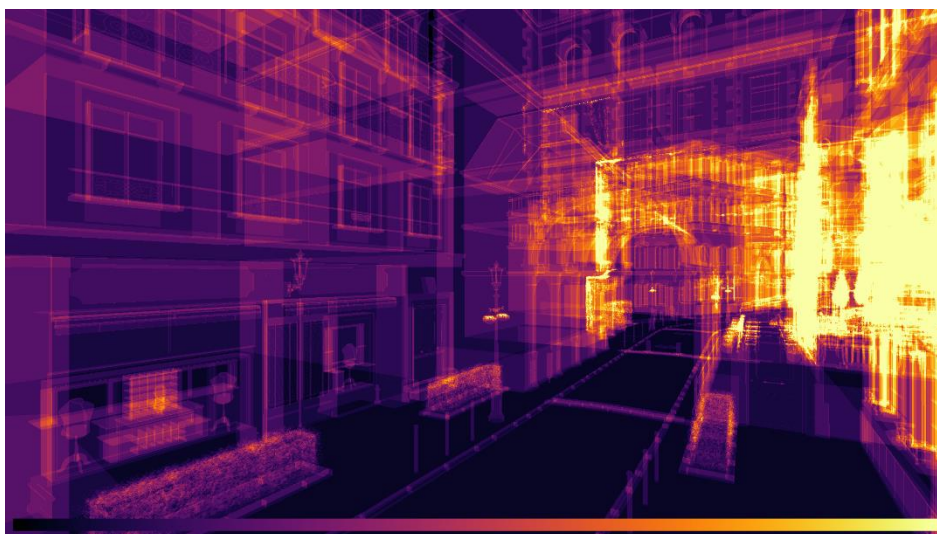


Рисунок 4.3 – Візуалізація перекриття пікселів без відсікання геометрії

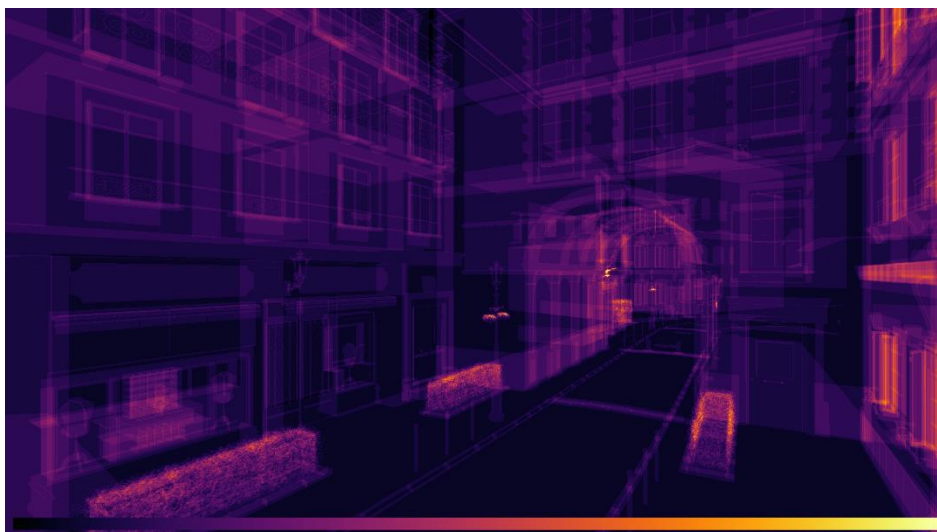


Рисунок 4.4 – Візуалізація перекриття пікселів із застосуванням GPU-керovanого відсікання

Для кількісної оцінки ефективності алгоритмів було проведено серію експериментів із поетапним додаванням методів відсікання. На першому етапі виконувався рендеринг сцени без будь-яких оптимізацій, що забезпечило базову точку порівняння. Далі послідовно додавалися такі етапи:

- відсікання за пірамідою видимості;
- оклюзійне відсікання;
- відсікання трикутників.

Такий підхід дозволяє оцінити внесок кожного з етапів окремо та визначити їх сумарний ефект. Результати експериментів наведено в таблиці 4.1.

Таблиця 4.1 – Результати тестування ефективності відсікання геометрії

	Кількість екземплярів	Кількість мешлетів	Кількість трикутників	Час растеризац ії (мс)
Без відсікання	3815	288924	23811722	1,26
Відсікання за пірамідою видимості	1929	131559	10757574	0,82
Оклюзійне відсікання	265	4596	252636	0,10
Відсікання трикутників	265	4596	96761	0,08

Отримані результати показують, що після застосування лише відсікання за пірамідою видимості зменшується обсяг геометрії приблизно вдвічі: кількість трикутників скорочується з 23,8 млн до 10,8 млн, а час растеризації – з 1,26 мс до 0,82 мс. Це свідчить про ефективне вилучення об’єктів, які не потрапляють у поле зору камери.

Найбільше скорочення забезпечує оклюзійне відсікання. Після його застосування кількість екземплярів зменшується з 1929 до 265, кількість мешлетів – зі 131559 до 4596, а кількість трикутників – з 10,8 млн до 252,6 тис.

Відповідно, час растеризації знижується з 0,82 мс до 0,10 мс. Такий результат пояснюється структурою сцени Lumberyard Bistro: велика частина геометрії розташована за стінами, фасадами, внутрішніми конструкціями та іншими ближчими об'єктами. Тобто вона формально може перебувати в полі зору камери, але фактично не впливає на сформоване зображення.

Відсікання трикутників додатково зменшує кількість примітивів з 252,6 тис. до 96,8 тис., тобто приблизно у 2,6 раза. Водночас час растеризації скорочується лише з 0,10 мс до 0,08 мс. Це пов'язано з тим, що зменшення кількості трикутників не дає прямо пропорційного скорочення часу виконання: самі перевірки видимості також мають накладні витрати. Тому цей етап дає додаткову оптимізацію, але його внесок менший порівняно з оклюзійним відсіканням.

Повна схема відсікання зменшує час растеризації з 1,26 мс до 0,08 мс, що відповідає прискоренню приблизно у 15,8 раза порівняно з режимом без оптимізації. Водночас ці результати залежать від просторової структури сцени, щільності розташування об'єктів і положення камери. Найбільший ефект оклюзійне відсікання дає за наявності значної кількості геометрії, перекритої ближчими поверхнями. Для відкритих сцен із меншою кількістю перекриттів приріст продуктивності буде нижчим.

Окремо було перевірено ефективність керування пам'яттю у підсистемі трасування променів на прикладі трьох сцен. Результати наведено в таблиці 4.2.

Таблиця 4.2 – Порівняння використання відеопам'яті BLAS-структурами до та після компактизації

	До компактизації (МБ)	Після компактизації (МБ)	Економія (%)
Intel Sponza	640,11	218,97	65,79
San Miguel	335,90	110,28	67,17
Lumberyard Bistro	322,02	109,07	66,13

Після компактизації BLAS-структури займають приблизно третину від початкового обсягу пам'яті. Водночас економія для різних сцен залишається близькою та в середньому становить приблизно 66,4%. Це свідчить про стабільність цього підходу для сцен із різною структурою геометрії.

Практичний ефект такої оптимізації полягає в тому, що зменшується базове навантаження на відеопам'ять у сценах із трасуванням променів. Це особливо важливо для рушія, де одночасно використовуються не лише BLAS і TLAS, а й текстури, матеріальні буфери, рендер-цілі, історичні буфери постобробки та дані освітлення. Звільнення значної частини пам'яті дає змогу або працювати зі складнішою геометрією, або залишати більший ресурсний запас для інших підсистем рендерингу.

Крім оцінки окремих оптимізацій, було виконано профілювання повного кадру (рис. 4.5). Вимірювання виконувалося для сцени з використанням глобального освітлення на основі сітки розміром $32 \times 16 \times 32$, тобто 16384 зонди, а для кожного зонда трасувалося 128 променів.

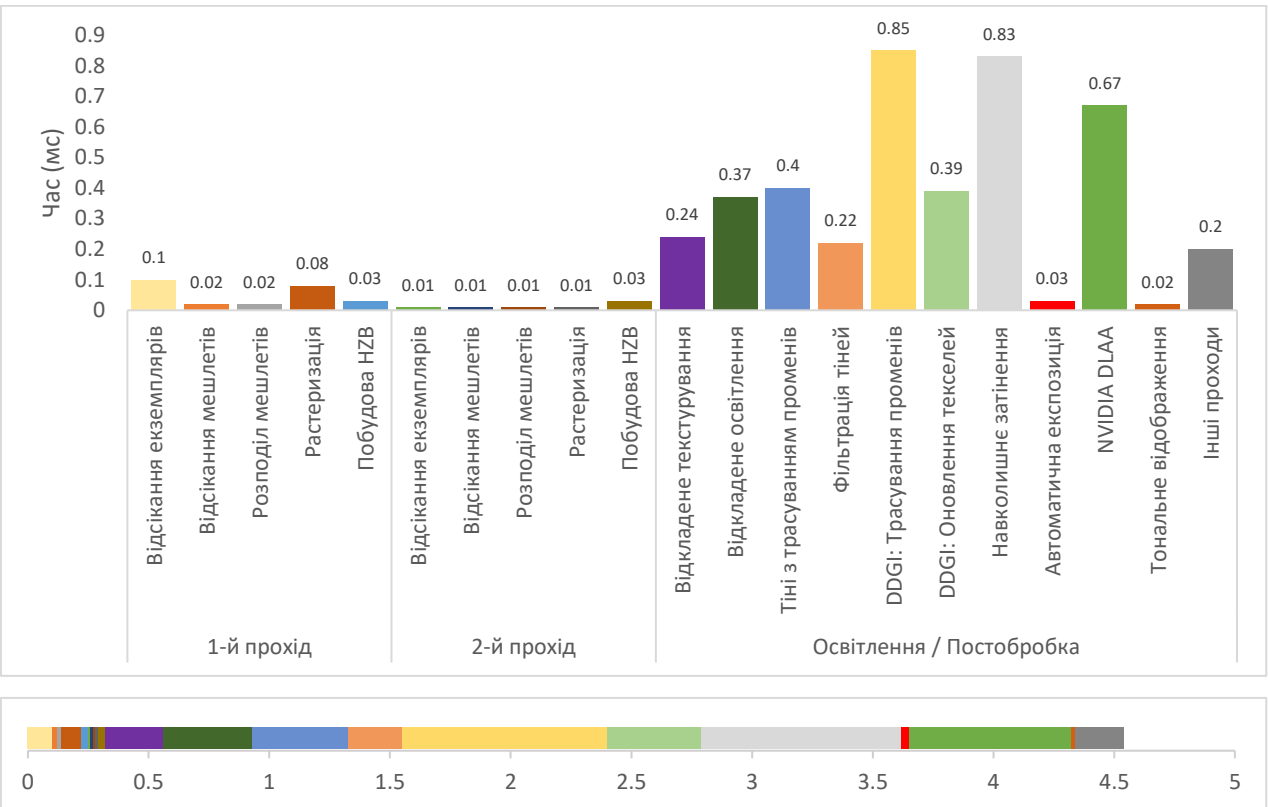


Рисунок 4.5 – Розподіл часу виконання кадру за проходами рендерингу

Розподіл часу має характерну для GPU-керованого конвеєра структуру: після відсікання геометрія перестає бути основним джерелом навантаження, а більша частина бюджету кадру переходить до проходів освітлення, трасування променів і екранної обробки.

4.2 Оцінка нейромережевого навколишнього затінення

Метою цього етапу є оцінка того, наскільки MLP-модель здатна наблизити еталонний результат за набором ознак, доступних під час формування кадру. Для експерименту було навчено три конфігурації мережі: 2×16 , 3×32 та 3×64 . У такому позначенні перше число визначає кількість прихованих шарів, а друге – кількість нейронів у кожному прихованому шарі. Як еталон використовувалося навколишнє затінення, отримане трасуванням променів за косинусно-зваженою півсферою.

На рисунку 4.6 наведено результати MLP-апроксимації навколишнього затінення. Зліва направо показано результати моделей 2×16 , 3×32 , 3×64 та еталонне навколишнє затінення.

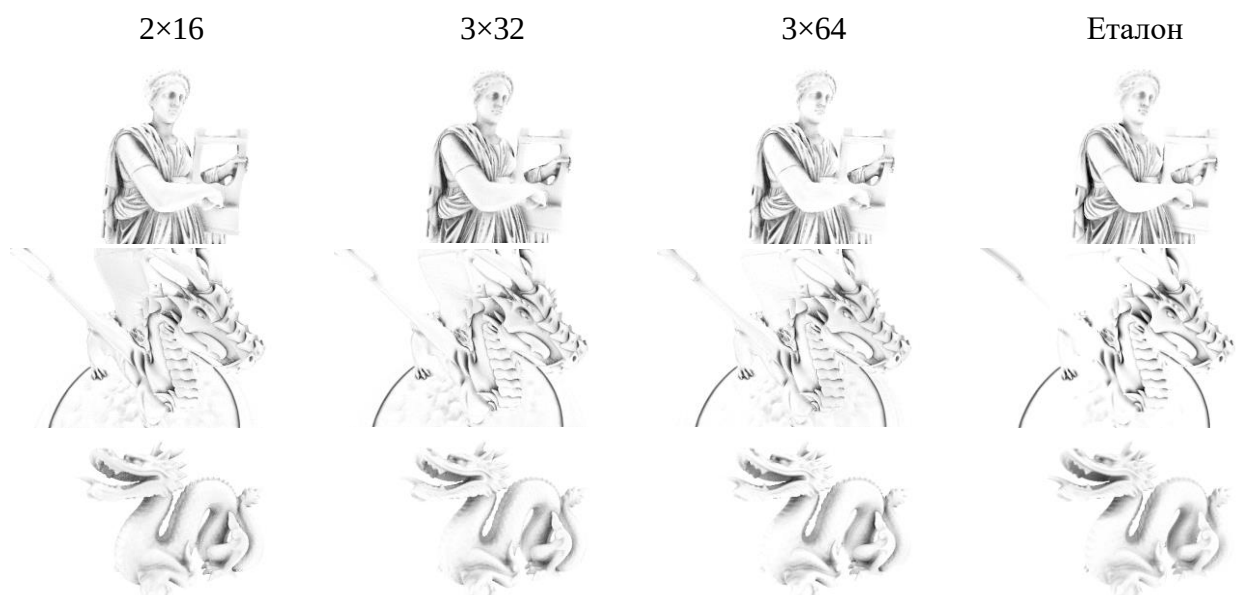


Рисунок 4.6 – Порівняння результатів MLP-апроксимації навколишнього затінення з еталонним результатом

Візуальне порівняння показує, що всі три конфігурації відтворюють загальний характер затінення, однак відрізняються рівнем усереднення та передаванням локального контрасту. Модель 2×16 має найменшу кількість параметрів, тому результат виглядає більш згладженим: темні ділянки менш виражені, а на переходах між освітленими та затіненими областями з'являється більше проміжних сірих значень. Конфігурація 3×32 зменшує цей ефект і краще зберігає форму зон затінення, однак усе ще поступається еталону в локальних деталях. Найближчим до еталонного результату є варіант 3×64, у якому темні області та дрібні перепади інтенсивності відтворюються точніше.

Для кількісного аналізу використано метрики MAE, RMSE та SSIM. Метрика MAE визначає середнє абсолютне відхилення між еталонним і прогнозованим значеннями навколишнього затінення:

$$MAE = \frac{1}{N} \sum_{i=1}^N |y_i - \hat{y}_i|, \quad (4.1)$$

де N – кількість пікселів;

y_i – еталонне значення;

$\hat{y}_i$ – значення прогнозу.

Метрика RMSE визначає корінь із середньоквадратичної похибки:

$$RMSE = \sqrt{\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2}, \quad (4.2)$$

де N – кількість пікселів;

y_i – еталонне значення;

$\hat{y}_i$ – значення прогнозу.

На відміну від MAE, ця метрика сильніше враховує великі локальні відхилення, оскільки різниця між еталонним і прогнозованим значеннями підноситься до квадрата.

SSIM – це метрика для оцінювання схожості між двома зображеннями, яка враховує зміни структурної інформації, яскравості та контрасту. На відміну від MAE та RMSE, що оцінюють переважно числові відмінності між піксельними значеннями, SSIM краще узгоджується з особливостями людського сприйняття зображень та визначається за такою формулою:

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}, \quad (4.3)$$

де x, y – порівнювані зображення;

μ_x, μ_y – середні значення яскравості відповідних локальних областей;

σ_x^2, σ_y^2 – дисперсії;

σ_{xy} – коваріація між зображеннями;

C_1, C_2 – сталі для стабілізації обчислень;

Значення SSIM, ближче до 1, відповідає більшій структурній подібності.

Результати кількісного аналізу наведено в таблиці 4.3.

Таблиця 4.3 – Порівняння MLP-моделей за якістю апроксимації

	MAE	RMSE	SSIM
2×16	0,0297	0,0476	0,8588
3×32	0,0245	0,0391	0,8716
3×64	0,0205	0,0338	0,8792

Збільшення розмірності MLP позитивно впливає на якість апроксимації, однак характер цього покращення є нерівномірним. Перехід від 2×16 до 3×32 дає найбільш помітний вииграш: зменшуються як середні відхилення, так і більші локальні помилки, що вказує на кращу здатність моделі відтворювати складніші залежності між вхідними ознаками та еталонним значенням. Подальше збільшення до 3×64 також покращує результат, але приріст стає менш вираженим. Це означає, що більша модель переважно уточнює вже відтворену структуру затінення, а не принципово змінює якість апроксимації.

На рисунку 4.7 наведено час інференсу навчених моделей і методу NNAO за трьох типових роздільних здатностей. Оскільки MLP обчислюється безпосередньо під час формування кадру, її вартість додається до загального часу рендерингу та впливає на придатність методу для роботи в реальному часі.

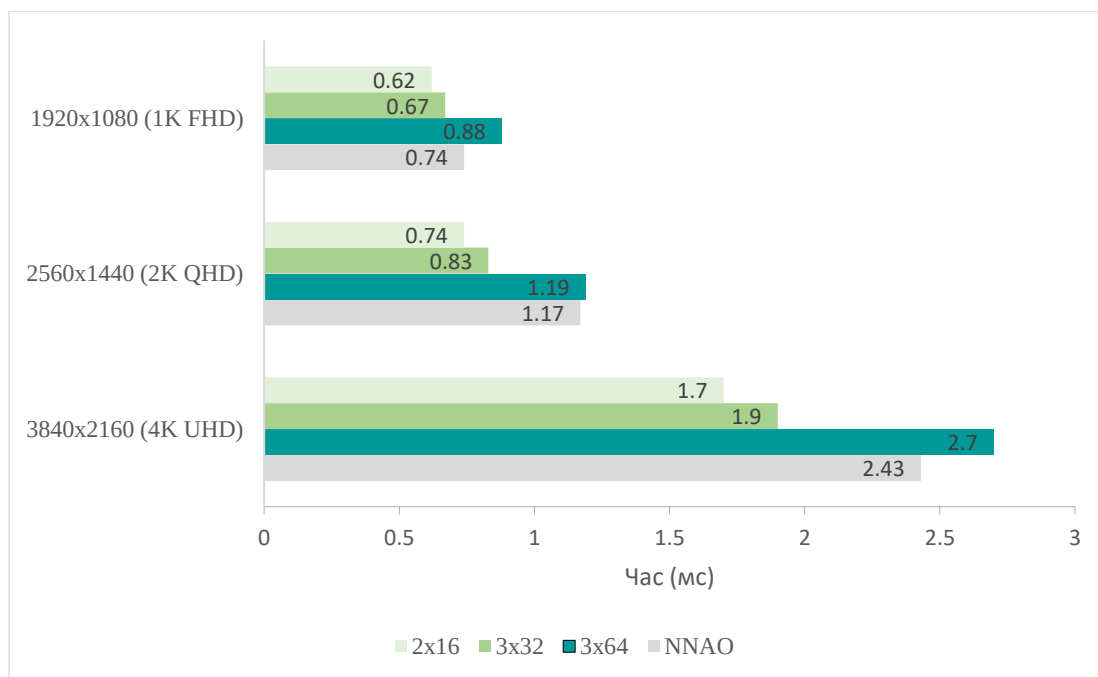


Рисунок 4.7 – Порівняння часу інференсу MLP-моделей та методу NNAO при різних роздільних здатностях рендерингу

Перехід від 2×16 до 3×32 супроводжується помірним зростанням часу виконання, тому така конфігурація зберігає прийнятну швидкодію навіть за вищої роздільної здатності. Натомість модель 3×64 створює значно більше навантаження на GPU, і її перевага за якістю вже не компенсує збільшення вартості інференсу. Водночас масштабування не є строго пропорційним кількості пікселів, оскільки на результат впливають планування потоків, доступ до пам'яті та заповнення обчислювальних блоків. З огляду на це, конфігурація 3×32 є найбільш доцільною для подальшого використання, оскільки забезпечує раціональний баланс між якістю апроксимації та вартістю інференсу.

4.3 Висновки до розділу

Результати тестування показали, що GPU-керована архітектура дає змогу суттєво зменшити навантаження на геометричну частину конвеєра. У тестовій сцені повна схема відсікання зменшила час растеризації з 1,26 мс до 0,08 мс, що відповідає прискоренню приблизно у 15,8 раз. Такий результат підтверджує доцільність перенесення відбору геометрії на GPU, оскільки значна частина невидимих об'єктів, мешлетів і примітивів вилучається ще до фінального відмалювання.

Подальший розвиток GPU-керованої частини рушія доцільно пов'язувати з більш гнучким способом організації роботи між проходами. Поточний підхід залишається прив'язаним до наперед заданої структури кадру, де послідовність етапів і синхронізація між ними визначаються заздалегідь. Перспективним напрямом є використання графів задач, у яких GPU може самостійно породжувати наступні одиниці роботи залежно від результатів попередніх обчислень. У Vulkan цей підхід представлений розширенням `VK_AMD_X_shader_enqueue`, однак через експериментальний статус і обмежену практичну підтримку воно поки не може розглядатися як стабільна основа для поточної реалізації рушія.

Нейронний рендеринг у межах проведеного тестування показав себе як перспективний спосіб зменшення вартості окремих дорогих етапів формування кадру. На прикладі навколишнього затінення було показано, що MLP-модель здатна отримувати результат, близький до еталонного трасованого обчислення, але з істотно меншою часовою вартістю. Водночас подальше покращення такого підходу обмежується природою екранного простору: модель не має доступу до прихованої або позакадрової геометрії, яку може враховувати трасування променів у просторі сцени. Отже, цей метод доцільно застосовувати як швидке наближене оцінювання, розуміючи його обмеження порівняно з трасованим еталоном.

ВИСНОВКИ

У межах дипломного проєкту було створено графічний рушій реального часу, призначений для дослідження сучасних підходів до організації рендерингу на графічному процесорі. Основний акцент у роботі зроблено на побудові GPU-керованого рендеринг-конвеєра, у якому підготовка геометрії, перевірка видимості та частина допоміжних обчислень виконуються безпосередньо на GPU.

Під час роботи було розглянуто можливості сучасних графічних рушіїв і нейромережових методів у комп'ютерній графіці. Аналіз показав, що готові рушії є ефективними для прикладної розробки, однак їхня універсальність і складна внутрішня структура обмежують можливості дослідження власних низькорівневих рендерингових схем. Це стало підставою для реалізації окремого рушія з повним контролем над організацією графічних і обчислювальних проходів.

Окрім геометричної частини, рушій було доповнено підсистемами трасування променів, динамічного глобального освітлення, нейромережевого навколишнього затінення та реконструкції зображення. Завдяки цьому розроблена система не обмежується базовою растеризацією, а охоплює кілька сучасних напрямів рендерингу реального часу: оптимізацію видимості, фізично точнішу перевірку освітлення, апроксимацію дорогих ефектів і підвищення якості фінального кадру.

Розроблений рушій може використовуватися як основа для подальших досліджень сучасних методів рендерингу. Його архітектура забезпечує можливість подальшого розширення, зокрема реалізацію нових графічних ефектів, удосконалення нейромережових моделей та впровадження більш гнучких механізмів планування роботи на GPU. Таким чином, поставлену мету дипломного проєкту було досягнуто.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Game Engine Architecture and Comparative Study of Different Game Engines / C. Vohera et al. // 2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT). IEEE, 2021. P. 1–6. URL: <https://doi.org/10.1109/ICCCNT51525.2021.9579618> (дата звернення: 25.05.2026).
2. Real-time Neural Radiance Caching for Path Tracing / T. Müller, F. Rousselle, J. Novák, A. Keller // ACM Transactions on Graphics. 2021. Vol. 40, No. 4. Article 36. P. 1–16. URL: <https://doi.org/10.1145/3450626.3459812> (дата звернення: 25.05.2026).
3. Radiance Caching for Efficient Global Illumination Computation / J. Křivánek, P. Gautron, S. Pattanaik, K. Bouatouch // IEEE Transactions on Visualization and Computer Graphics. 2005. Vol. 11, No. 5. P. 550–561. URL: <https://doi.org/10.1109/TVCG.2005.83> (дата звернення: 25.05.2026).
4. Neural Network Ambient Occlusion / D. Holden, J. Saito, T. Komura // SIGGRAPH Asia 2016 Technical Briefs. ACM, 2016. Article 9. P. 1–4. URL: <https://doi.org/10.1145/3005358.3005387> (дата звернення: 25.05.2026).
5. Foley T. Slang: Designing a Shading Language for Vulkan // Vulkanised 2024. NVIDIA, 2024. URL: <https://vulkan.org/user/pages/09.events/vulkanised-2024/vulkanised-2024-thereas-foley-nvidia.pdf> (дата звернення: 25.05.2026).
6. SLANG.D: Fast, Modular and Differentiable Shader Programming / S. Bangaru et al. // ACM Transactions on Graphics. 2023. Vol. 42, No. 6. Article 264. P. 1–28. URL: <https://doi.org/10.1145/3618353> (дата звернення: 25.05.2026).
7. AMD FidelityFX Single Pass Downsampler. AMD GPUOpen. URL: <https://gpuopen.com/fidelityfx-spd/> (дата звернення: 25.05.2026).
8. meshoptimizer. GitHub. URL: <https://github.com/zeux/meshoptimizer> (дата звернення: 25.05.2026).

9. Dear ImGui. GitHub. URL: <https://github.com/ocornut/imgui> (дата звернення: 25.05.2026).
10. Mesh Shading. Vulkan Documentation Project. URL: https://docs.vulkan.org/spec/latest/chapters/VK_NV_mesh_shader/mesh.html (дата звернення: 25.05.2026).
11. Kubisch C. Introduction to Turing Mesh Shaders // NVIDIA Technical Blog. 2018. URL: <https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/> (дата звернення: 25.05.2026).
12. Haar U., Aaltonen S. GPU-Driven Rendering Pipelines // SIGGRAPH 2015: Advances in Real-Time Rendering in Games. 2015. URL: http://advances.realtimerendering.com/s2015/aaltonenhaar_siggraph2015_combined_final_footer_220dpi.pdf (дата звернення: 25.05.2026).
13. Greene N., Kass M., Miller G. Hierarchical Z-Buffer Visibility // Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '93). ACM, 1993. P. 231–238. URL: <https://doi.org/10.1145/166117.166147> (дата звернення: 25.05.2026).
14. Olano M., Greer T. Triangle Scan Conversion using 2D Homogeneous Coordinates // Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware. ACM, 1997. P. 89–95. URL: <https://doi.org/10.1145/258694.258723> (дата звернення: 25.05.2026).
15. Wihlidal G. Optimizing the Graphics Pipeline With Compute // GDC 2016. Frostbite Engine / Electronic Arts. GDC Vault, 2016. URL: <https://www.gdcvault.com/play/1023109/Optimizing-the-Graphics-Pipeline-With> (дата звернення: 25.05.2026).
16. Lively D., Gruen H. Wave Programming in D3D12 and Vulkan // GDC 2017. AMD GPUOpen, 2017. URL: <https://gpuopen.com/download/GDC2017-Wave-Programming-D3D12-Vulkan.pdf> (дата звернення: 25.05.2026).
17. Burns C. A., Hunt W. A. The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading // Journal of Computer Graphics Techniques. 2013. Vol. 2,

No. 2. P. 55–69. URL: <https://jcgt.org/published/0002/02/04/> (дата звернення: 25.05.2026).

18. Schied C., Dachsbacher C. Deferred Attribute Interpolation for Memory-Efficient Deferred Shading // Proceedings of High Performance Graphics 2015. ACM, 2015. P. 43–49. URL: <https://doi.org/10.1145/2790060.2790066> (дата звернення: 25.05.2026).
19. Spatiotemporal Blue Noise Masks / A. Wolfe, N. Morrical, T. Akenine-Möller, R. Ramamoorthi // Eurographics Symposium on Rendering. The Eurographics Association, 2022. P. 117–126. URL: <https://doi.org/10.2312/sr.20221161> (дата звернення: 25.05.2026).
20. Karis B. High Quality Temporal Supersampling // SIGGRAPH 2014: Advances in Real-Time Rendering in Games. Epic Games, 2014. URL: <http://advances.realtimerendering.com/s2014/epic/TemporalAA.pptx> (дата звернення: 25.05.2026).
21. Bigler J., Hart D. Neural Rendering in NVIDIA OptiX Using Cooperative Vectors // NVIDIA Technical Blog. 2025. URL: <https://developer.nvidia.com/blog/neural-rendering-in-nvidia-optix-using-cooperative-vectors/> (дата звернення: 25.05.2026).
22. Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields / Z. Majercik, J.-P. Guertin, D. Nowrouzezahrai, M. McGuire // Journal of Computer Graphics Techniques. 2019. Vol. 8, No. 2. P. 1–30. URL: <https://jcgt.org/published/0008/02/01/> (дата звернення: 25.05.2026).
23. Spherical Fibonacci Mapping / B. Keinert, M. Innmann, M. Sängler, M. Stamminger // ACM Transactions on Graphics. 2015. Vol. 34, No. 6. Article 193. P. 1–7. URL: <https://doi.org/10.1145/2816795.2818131> (дата звернення: 25.05.2026).
24. NVIDIA DLSS. NVIDIA Developer. URL: <https://developer.nvidia.com/rtx/dlss> (дата звернення: 25.05.2026).

ДОДАТОК А

Графічний рушій реального часу з використанням GPU-керованого
рендерингу та нейронних мереж на базі Vulkan API

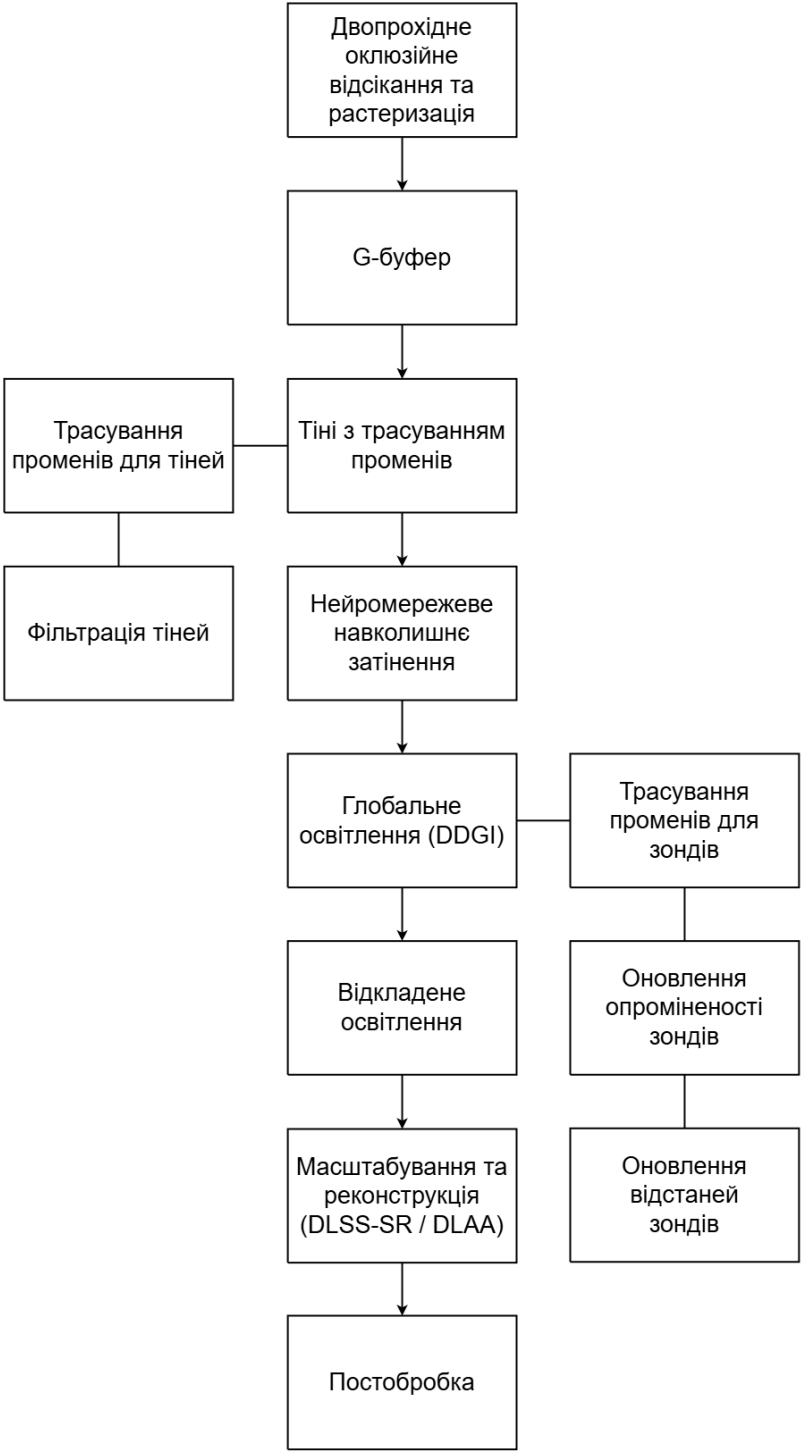
Графічний конвеєр рендерингу.

Схема структурна

ІАЛЦ.467800.005 Д1

Аркушів 1

Київ 2026



					ІАЛЦ.467800.005 Д1				
					Графічний конверс рендерингу. Схема структурна	Літ.	Маса	Масштаб	
Зм.	Арк.	№ докум.	Підпис	Дата					
Розробив		Литвин С.Р.							
Перевірив		Клятченко Я.М.							
						Арк.	Аркушів		
Н. контр.		Клятченко Я.М.			КПІ ім. Ігоря Сікорського, ФПСПМ KB-23				
Затв.		Романкевич В.О.							

ДОДАТОК Б

Графічний рушій реального часу з використанням GPU-керованого
рендерингу та нейронних мереж на базі Vulkan API

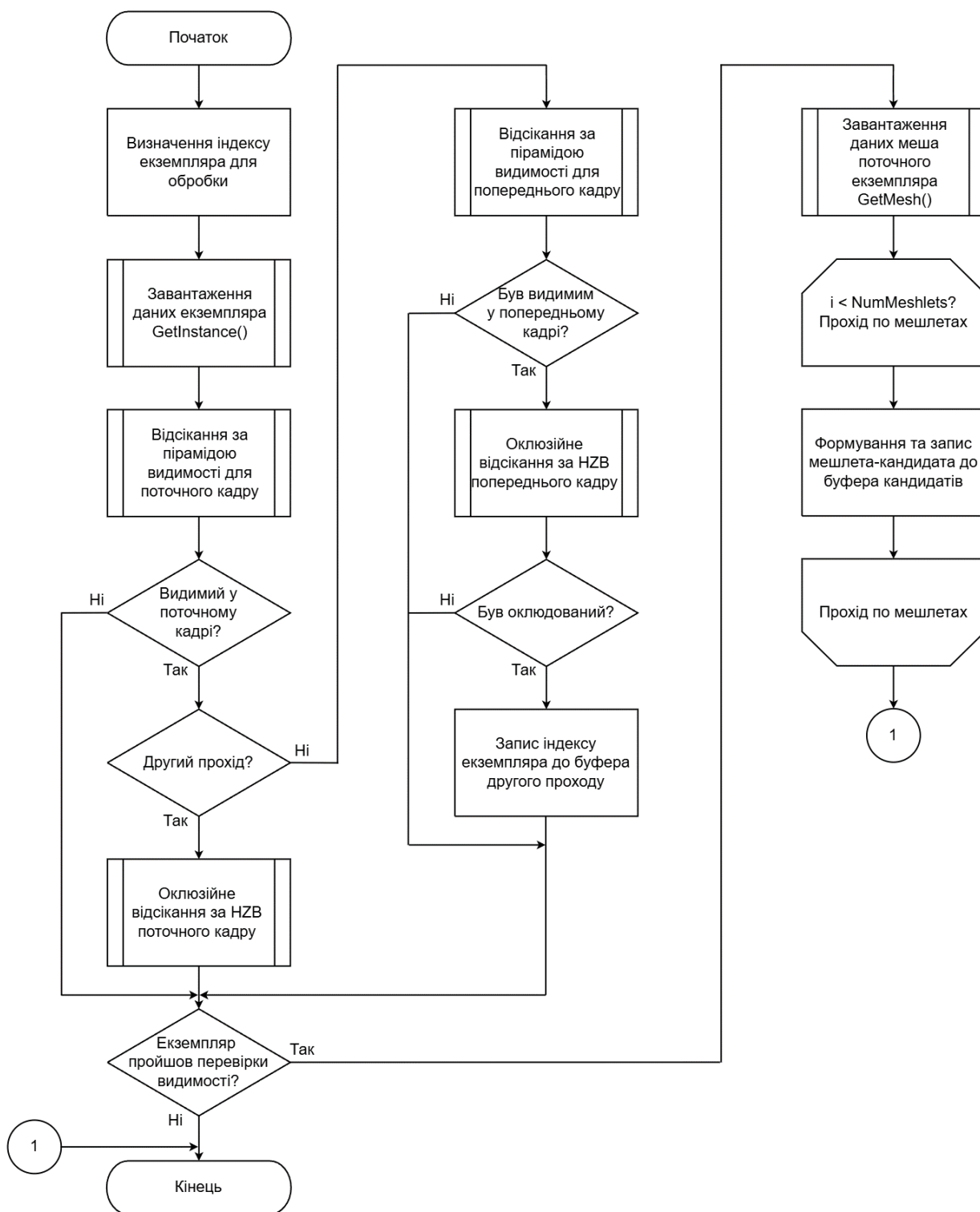
Алгоритм двоетапного оклюзійного відсікання екземплярів.

Схема алгоритму

ІАЛЦ.467800.006 Д2

Аркушів 1

Київ 2026



						ІАЛЦ.467800.006 Д2			
						Двоетапне оклюдійне відсікання екземплярів.			
Зм.	Арк.	№ докум.	Підпис	Дата		Схема алгоритму			
Розробив		Литвин С.Р.							
Перевірив		Клятченко Я.М.				КП ім. Ігоря Сікорського, ФГСПМ КВ-23			
Н. контр.		Клятченко Я.М.							
Затв.		Романкевич В.О.							

ДОДАТОК В

Графічний рушій реального часу з використанням GPU-керованого
рендерингу та нейронних мереж на базі Vulkan API

Процес навчання MLP-моделі для апроксимації навколишнього затінення.

Схема структурна

ІАЛЦ.467800.007 ДЗ

Аркушів 1

Київ 2026



						ІАЛЩ.467800.007 ДЗ				
						Процес навчання MLP-моделі для апроксимації навколишнього затинення.	Літ.	Маса	Масштаб	
Зм.	Арк.	№ докум.	Підпис	Дата		Схема структурна	Арк.		Аркушів	
Розробив		Литвин С.Р.								
Перевірив		Клятченко Я.М.								
Н. контр.		Клятченко Я.М.					КПІ ім. Ігоря Сікорського, ФПСПМ КВ-23			
Затв.		Романкевич В.О.								

ДОДАТОК Г

Графічний рушій реального часу з використанням GPU-керованого
рендерингу та нейронних мереж на базі Vulkan API

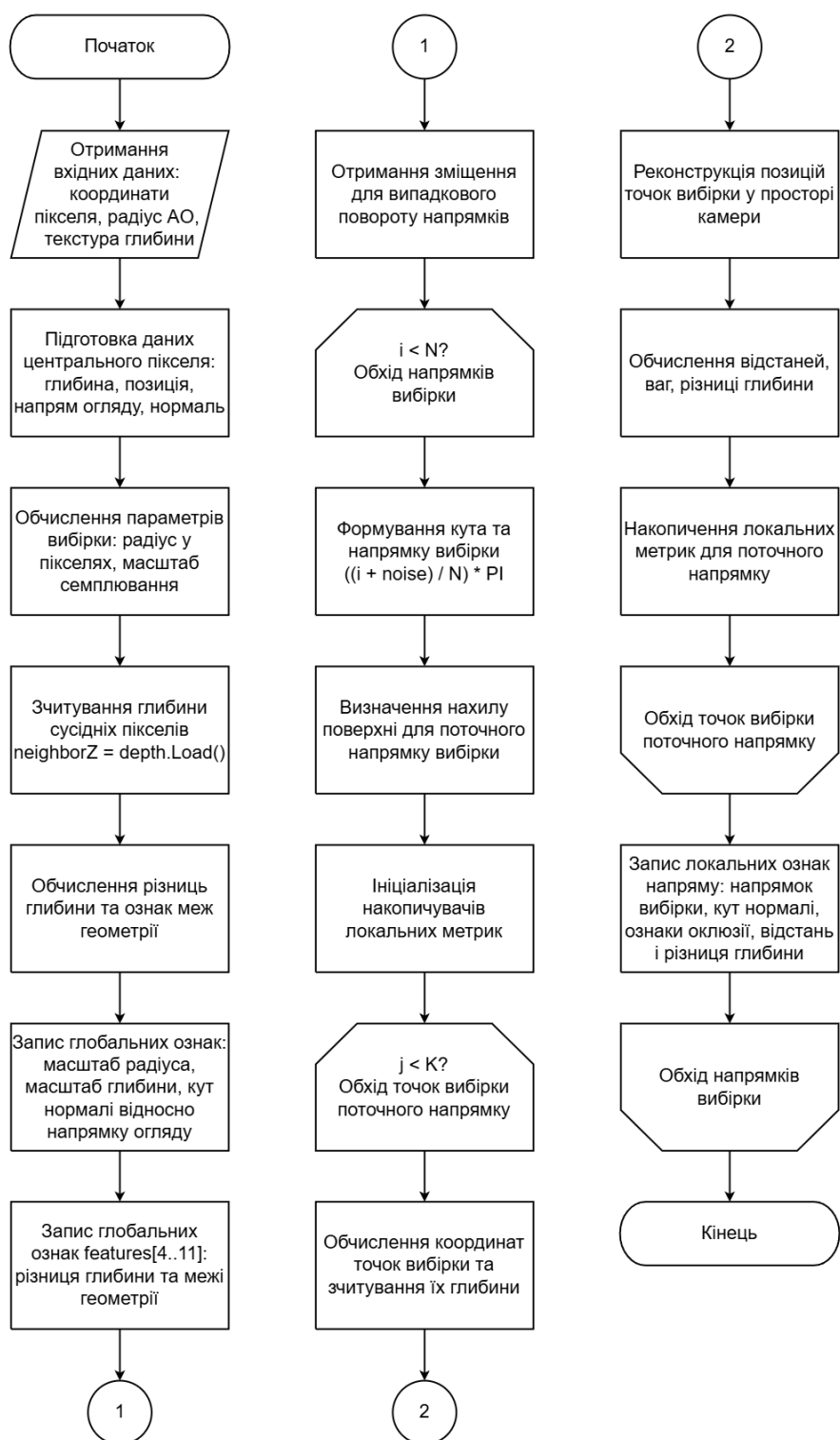
Алгоритм формування вхідних ознак для MLP-моделі.

Схема алгоритму

ІАЛЦ.467800.008 Д4

Аркушів 1

Київ 2026



						ІАЛЦ.467800.008 Д4					
						Формування вхідних ознак для MLP-моделі. Схема алгоритму	Літ.		Маса	Масштаб	
Зм.	Арк.	№ докум.	Підпис	Дата							
Розробив		Литвин С.Р									
Перевірив		Клятченко Я.М.									
							Арк.		Аркушів		
Н. контр.		Клятченко Я.М.					КПІ ім. Ігоря Сікорського, ФПСІМ КВ-23				
Затв.		Романкевич В. О.									

ДОДАТОК Д

Графічний рушій реального часу з використанням GPU-керованого
рендерингу та нейронних мереж на базі Vulkan API

Фрагменти програмного коду

ІАЛЦ.467800.009 Д5

Аркушів 7

Київ 2026

InstanceCulling.slang

```
namespace Visibility
{
    CullData FrustumOBB(float3 center, float3 extents, float4x4 world, float4x4 viewProj)
    {
        CullData cull = (CullData) 0;
        float minW = 1e30f;
        float maxW = -1e30f;
        float4 planesMin = float4(1.0f, 1.0f, 1.0f, 1.0f);
        float4x4 worldViewProj = mul(world, viewProj);

        float3 rectMin = float3(1.0f, 1.0f, 1.0f);
        float3 rectMax = float3(-1.0f, -1.0f, 0.0f);
        for (int z = -1; z <= 1; z += 2)
            for (int y = -1; y <= 1; y += 2)
                for (int x = -1; x <= 1; x += 2)
                {
                    float4 clipPos = mul(float4(center + extents * float3(x, y, z), 1.0f), worldViewProj);
                    minW = min(minW, clipPos.w);
                    maxW = max(maxW, clipPos.w);
                    planesMin = min(planesMin, float4(clipPos.xy, -clipPos.xy) - clipPos.w);

                    if (clipPos.w > 0.0f)
                    {
                        float3 ndcPos = clipPos.xyz / clipPos.w;
                        rectMin = min(rectMin, ndcPos);
                        rectMax = max(rectMax, ndcPos);
                    }
                }

        float minZ = maxW * zScaleOffset.x + zScaleOffset.y;
        float maxZ = minW * zScaleOffset.x + zScaleOffset.y;
        bool isCulled = (maxW <= minZ) || (maxZ <= 0.0f) || any(planesMin > 0.0f);
        cull.RectMin = rectMin;
        cull.RectMax = rectMax;
        if (minW <= 0.0f && maxW > 0.0f)
        {
            cull.RectMin = -1;
            cull.RectMax = +1;
        }
        else
        {
            cull.RectMin.z = minZ / maxW;
            cull.RectMax.z = maxZ / minW;
        }
        cull.IsVisible = !isCulled;
        return cull;
    }
}
```

```
bool Occlusion(CullData cullData, Texture2D<float> hzbTexture, float2 hzbSize)
{
    float3 rectMin = cullData.RectMin;
    float3 rectMax = cullData.RectMax;
    float depth = +1e30;
    float4 rectUV = float4(rectMin.xy, rectMax.xy);
    rectUV = saturate(rectUV * float2(0.5f, -0.5f).xyxy + 0.5f).xwzy;
    int4 rectPixels = int4(rectUV * hzbSize.xyxy);
    rectPixels.xy = max(rectPixels.xy, 0);
    rectPixels.zw = min(rectPixels.zw, hzbSize.xy);
    int2 mipLevelXY = firstbithigh(rectPixels.zw - rectPixels.xy);
    int mipLevel = max(max(mipLevelXY.x, mipLevelXY.y) - 1, 0);
    int2 texelDelta = (rectPixels.zw >> mipLevel) - (rectPixels.xy >> mipLevel);

    if (any(texelDelta > 3))
        mipLevel++;

    rectPixels >>= mipLevel;
    float2 texelSize = rcp(hzbSize) * (1u << mipLevel);
    float4 xCoords = (min(rectPixels.x + int4(0, 1, 2, 3), rectPixels.z) + 0.5f) * texelSize.x;
    float4 yCoords = (min(rectPixels.y + int4(0, 1, 2, 3), rectPixels.w) + 0.5f) * texelSize.y;

    for (int y = 0; y < 4; ++y)
        for (int x = 0; x < 4; ++x)
        {
            depth = min(depth, hzbTexture.SampleLevel(PointClampedSampler,
                float2(xCoords[x], yCoords[y]), mipLevel));
        }
    return cullData.IsVisible && !(depth > rectMax.z);
}
}
```

```

[numthreads(INSTANCE_CULLING_GROUP_SIZE, 1, 1)]
void InstanceCullCS(uint DispatchThreadId : SV_DispatchThreadID)
{
    #if FIRST_PHASE
        uint numInstances = InstanceCullData.NumInstances;
    #else
        uint numInstances = OccludedInstancesCounter[0];
    #endif

    if (DispatchThreadId >= numInstances)
        return;

    uint instanceIndex = DispatchThreadId;
    #if !FIRST_PHASE
        instanceIndex = OccludedInstances[instanceIndex];
    #endif
    InstanceData instance = GetInstance(instanceIndex);
    MeshData mesh = GetMesh(instance);

    bool isVisible = true;
    bool wasOccluded = false;
    #if FRUSTUM_CULLING
        CullData cull = Visibility::FrustumOBB(instance.LocalBoundsOrigin,
            instance.LocalBoundsExtents,
            instance.LocalToWorld,
            FrameData.ViewProj);
        isVisible = cull.IsVisible;
    #if FIRST_PHASE
        {
            uint instanceCount = WaveActiveCountBits(isVisible);
            if (WaveIsFirstLane())
            {
                InterlockedAdd(RenderStats[0], instanceCount);
            }
        }
    #endif
    #if OCCLUSION_CULLING
        if (isVisible)
        {
            #if FIRST_PHASE
                CullData prevCull = Visibility::FrustumOBB(instance.LocalBoundsOrigin,
                    instance.LocalBoundsExtents,
                    instance.LocalToWorld,
                    FrameData.PrevViewProj);
                if (prevCull.IsVisible)
                {
                    wasOccluded = !Visibility::Occlusion(prevCull, HZBTexture, InstanceCullData.HZBSize);
                }
                if (wasOccluded)
                {
                    uint occludedInstanceOffset = 0;
                    uint instanceCount = WaveActiveCountBits(true);
                    if (WaveIsFirstLane())
                    {
                        InterlockedAdd(OccludedInstancesCounter[0], instanceCount, occludedInstanceOffset);
                    }
                    occludedInstanceOffset = WaveReadLaneFirst(occludedInstanceOffset) + WavePrefixCountBits(true);

                    if (occludedInstanceOffset < MAX_INSTANCES)
                        OccludedInstances[occludedInstanceOffset] = instance.ID;
                }
            #else {
                uint instanceCount = WaveActiveCountBits(true);
                if (WaveIsFirstLane())
                {
                    {
                        InterlockedAdd(RenderStats[1], instanceCount);
                    }
                }
            }
        }
    #else
        isVisible = Visibility::Occlusion(cull, HZBTexture, InstanceCullData.HZBSize);
    #endif
    #endif
    #endif

    if (isVisible && !wasOccluded)
    {
        uint meshletIndex = 0;
        uint meshletCount = WaveActiveSum(mesh.MeshletCount);
    }
}

```

```

    if (WaveIsFirstLane())
    {
        InterlockedAdd(CandidateMeshletsCounter[0], meshletCount, meshletIndex);
    }
    meshletIndex = WaveReadLaneFirst(meshletIndex) + WavePrefixSum(mesh.MeshletCount);
    if (meshletIndex + mesh.MeshletCount < MAX_CANDIDATE_MESHLETS)
    {
        uint meshletOffset = 0;
        meshletCount = WaveActiveSum(mesh.MeshletCount);
        if (WaveIsFirstLane())
        {
            InterlockedAdd(CandidateMeshletsCounter[FIRST_PHASE ? 1 : 2], meshletCount, meshletOffset);
        }
        meshletOffset = WaveReadLaneFirst(meshletOffset) + WavePrefixSum(mesh.MeshletCount);
        uint meshletCandidateOffset = !FIRST_PHASE ? CandidateMeshletsCounter[1] : 0u;
        for (uint i = 0; i < mesh.MeshletCount; ++i)
        {
            VisibleMeshlet meshlet;
            meshlet.InstanceID = instance.ID;
            meshlet.MeshletIndex = i;
            CandidateMeshlets[meshletCandidateOffset + meshletOffset + i] = meshlet;
        }
    }
}
}
}

```

MeshletRasterize.slang

```

[outputtopology("triangle")]
[numthreads(MESH_SHADER_GROUP_SIZE, 1, 1)]
void RasterizeMS(
    in uint groupThreadID : SV_GroupIndex,
    in uint groupID : SV_GroupID,
    out vertices VertexAttributes verts[MESHLET_MAX_VERTICES],
    out indices uint3 triangles[MESHLET_MAX_TRIANGLES],
    out primitives PrimitiveAttributes primitives[MESHLET_MAX_TRIANGLES])
{
    uint meshletIndex = groupID;
    meshletIndex += RasterTypesOffsets[RasterData.BucketIndex].w;
    meshletIndex = MeshletIndices[meshletIndex];
    VisibleMeshlet candidate = VisibleMeshlets[meshletIndex];
    InstanceData instance = GetInstance(candidate.InstanceID);
    MeshData mesh = GetMesh(instance);
    Meshlet meshlet = LoadStaticData<Meshlet>(mesh, candidate.MeshletIndex, mesh.MeshletOffset);
    SetMeshOutputCounts(meshlet.VertexCount, meshlet.TriangleCount);

    for (uint i = groupThreadID; i < meshlet.VertexCount; i += MESH_SHADER_GROUP_SIZE)
    {
        uint vertexId = LoadStaticData<uint>(mesh, i + meshlet.VertexOffset, mesh.MeshletVertexOffset);
        VertexAttributes result = FetchVertexAttributes(mesh, instance.LocalToWorld, FrameData.ViewProj, vertexId);
        verts[i] = result;
    }
    #if TRIANGLE_CULLING
        VertexAttributes attr = FetchVertexAttributes(mesh, instance.LocalToWorld, RasterData.WorldToClip, vertexId);
        PosX[i] = attr.Position.x;
        PosY[i] = attr.Position.y;
        PosW[i] = attr.Position.w;
    #endif
    GroupMemoryBarrierWithGroupSync();

    uint visibleTriangles = 0;
    uint culledTriangles = 0;

    for (uint i = groupThreadID; i < meshlet.TriangleCount; i += MESH_SHADER_GROUP_SIZE)
    {
        Meshlet::Triangle tri = LoadStaticData<Meshlet::Triangle>(mesh, meshlet.TriangleOffset + i, mesh.MeshletTriangleOffset);
        triangles[i] = uint3(tri.V0, tri.V1, tri.V2);
        bool isCulled = false;

        #if TRIANGLE_CULLING
            float2 minAABB = 1.0f.xx;
            float2 maxAABB = 0.0f.xx;
            {
                float3 pc0 = float3(PosX[tri.V0], PosY[tri.V0], PosW[tri.V0]);
                float3 pc1 = float3(PosX[tri.V1], PosY[tri.V1], PosW[tri.V1]);
                float3 pc2 = float3(PosX[tri.V2], PosY[tri.V2], PosW[tri.V2]);

                float det = determinant(float3x3(pc0, pc1, pc2));
                isCulled = (det >= 0) && (!IsTwoSided() || det == 0);
            }
        #endif
    }
}

```

```

        if (!isCulled)
        {
            float2 ps0 = (pc0.xy / abs(pc0.z)) * 0.5f + 0.5f;
            float2 ps1 = (pc1.xy / abs(pc1.z)) * 0.5f + 0.5f;
            float2 ps2 = (pc2.xy / abs(pc2.z)) * 0.5f + 0.5f;

            minAABB = min(ps0, min(ps1, ps2));
            maxAABB = max(ps0, max(ps1, ps2));
            isCulled = any(minAABB >= 1.0f) || any(maxAABB <= 0.0f);
        }

        if (!isCulled)
        {
            minAABB *= GetViewportSize();
            maxAABB *= GetViewportSize();
            isCulled = any(round(minAABB) == round(maxAABB));
        }
    }
}

PrimitiveAttributes pri;
pri.PrimitiveID = i;
pri.VisibleIndex = meshletIndex;
pri.TriangleCulled = isCulled;
primitives[i] = pri;

visibleTriangles += !isCulled ? 1 : 0;
culledTriangles += isCulled ? 1 : 0;
}
}

```

MLPTraining.slang

```

public struct MLPFeatureExtractor
{
    public void Extract(float2 uv, uint2 pixelCoord, float radius, out rxns::HCoopVec<FeatureCount> features)
    {
        float invTapCount = 1.0f / (float)TapCount;
        float invRadius = rcp(radius);
        float depth = GetDepth(int2(pixelCoord));
        float3 viewPos = ScreenToViewPos(uv, depth);
        float3 viewDir = Normalize(-viewPos);
        float3 normal = Normalize(ReconstructViewNormalFromDepth(uv, DepthTexture));
        float radiusPixels = ViewRadiusToPixelRadius(radius, depth);
        float minScale = 1.3f / max(radiusPixels, Epsilon);
        float weightMul = -rcp(Epsilon);
        float weightAdd = radius * rcp(Epsilon) + 1.0f;
        int2 pixel = int2(pixelCoord);
        int2 maxPixel = int2(GetViewportSize()) - 1;

        float4 depths = float4(
            GetDepth(clamp(pixel + int2(-1, 0), int2(0, 0), maxPixel)),
            GetDepth(clamp(pixel + int2(1, 0), int2(0, 0), maxPixel)),
            GetDepth(clamp(pixel + int2(0, -1), int2(0, 0), maxPixel)),
            GetDepth(clamp(pixel + int2(0, 1), int2(0, 0), maxPixel)));
        float4 dz = depths - depth;
        float2 slope = 0.5f * float2(dz.y - dz.x, dz.w - dz.z);
        float4 predictedDz = dz + float4(slope.x, -slope.x, slope.y, -slope.y);
        float4 edge = saturate(1.25f - min(abs(dz), abs(predictedDz)) / max(depth * 0.011f, Epsilon));
        float4 dzScaled = dz * invRadius;

        features[0] = (half)saturate(log2(1.0f + max(radiusPixels, 0.0f)) * 0.20f);
        features[1] = (half)saturate(log2(1.0f + max(depth * invRadius, 0.0f)) * 0.20f);
        features[2] = (half)saturate(log2(1.0f + max(minScale, 0.0f) * 8.0f) * 0.25f);
        features[3] = (half)dot(normal, viewDir);

        [ForceUnroll]
        for (uint i = 0; i < 4; i++)
        {
            features[4 + i] = (half)(sign(dzScaled[i]) * log2(1.0f + max(abs(dzScaled[i]), Epsilon) * 8.0f) * 0.3125f);
            features[8 + i] = (half)(edge[i] * 2.0f - 1.0f);
        }

        float2 noise = BlueNoiseVec2(pixelCoord, 0);
        float2 halfTexel = 0.5f * GetInvViewportSize();
        float3 defaultTangent = Normalize(float3(-viewDir.y, viewDir.x, 0.0f));

        [ForceUnroll]
        for (uint dir = 0; dir < DirectionCount; dir++)
        {

```

```

float angle = ((dir + noise.x) / (float)DirectionCount) * PI;
float s; float c;
sincos(angle, s, c);
float2 dirPx = float2(c, -s) * radiusPixels;
float3 dirVec = float3(c, s, 0.0f);
float3 tangent = dirVec - dot(dirVec, viewDir) * viewDir;
tangent = dot(tangent, tangent) > MinFloat ? Normalize(tangent) : defaultTangent;
float3 axis = cross(tangent, viewDir);
axis = dot(axis, axis) > MinFloat ? Normalize(axis) : float3(0.0f, 1.0f, 0.0f);
float3 projNormal = normal - axis * dot(normal, axis);
float projLen = max(dot(projNormal, projNormal), Epsilon);
float invProjLen = rsqrt(projLen);
float normalLen = projLen * invProjLen;
float normalAngle = sign(dot(tangent, projNormal)) * acos(saturate(dot(projNormal, viewDir) * invProjLen));

float2 maxVal = -1.0f.xx;
float2 sum = 0.0f.xx;
float2 weights = 0.0f.xx;
float2 distSum = 0.0f.xx;
float2 depthSum = 0.0f.xx;

[ForceUnroll]
for (uint tap = 0; tap < TapCount; tap++)
{
    float tapNoise = frac(noise.y + dir * TapCount + tap);
    float scale = (tap + tapNoise) * invTapCount + minScale;
    float2 offset = round(scale * dirPx) * GetInvViewportSize();
    float2 uv0 = clamp(uv + offset, halfTexel, 1.0f.xx - halfTexel);
    float2 uv1 = clamp(uv - offset, halfTexel, 1.0f.xx - halfTexel);
    float2 sampleDepth = float2(SampleDepth(uv0), SampleDepth(uv1));
    float3 dx = ScreenToViewPos(uv0, sampleDepth.x) - viewPos;
    float3 dy = ScreenToViewPos(uv1, sampleDepth.y) - viewPos;
    float2 lenSq = max(float2(dot(dx, dx), dot(dy, dy)), Epsilon.xx);
    float2 invLen = rsqrt(lenSq);
    float2 dist = sqrt(lenSq);
    float2 sampleWeight = saturate(dist * weightMul + weightAdd);
    float2 viewDepth = float2(dot(dx, viewDir), dot(dy, viewDir));
    float2 val = viewDepth * invLen * sampleWeight;
    maxVal = max(maxVal, val);
    sum += val;
    weights += sampleWeight;
    distSum += dist * invRadius * sampleWeight;
    depthSum += viewDepth * invRadius * sampleWeight;
}

float2 invWeights = rcp(max(weights, Epsilon.xx));
float2 avgVal = sum * invWeights;
float support = 0.5f * (weights.x + weights.y) * invTapCount;
float avgDist = 0.5f * dot(distSum, invWeights);
float2 avgDepth = depthSum * invWeights;

uint idx = GlobalFeatureCount + dir * LocalFeatureCount;
features[idx + 0] = (half)c;
features[idx + 1] = (half)s;
features[idx + 2] = (half)(normalAngle * INV_PI);
features[idx + 3] = (half)saturate(normalLen);
features[idx + 4] = (half)maxVal.x;
features[idx + 5] = (half)maxVal.y;
features[idx + 6] = (half)avgVal.x;
features[idx + 7] = (half)avgVal.y;
features[idx + 8] = (half)support;
features[idx + 9] = (half)avgDist;
features[idx + 10] = (half)avgDepth.x;
features[idx + 11] = (half)avgDepth.y;
}
};

[shader("compute")]
[numthreads(THREADGROUP_SIZE_X, 1, 1)]
void MLPAmbientOcclusionTrainingCS(uint2 threadID : SV_DispatchThreadID)
{
    if (threadID.x >= TrainingData.BatchSize) return;
    uint seed = SeedThread(threadID, (uint)GetViewportSize().x, FrameData.FrameIndex);
    uint2 samplePixel;
    samplePixel.x = (uint)(Random01(seed) * GetInvViewportSize().x);
    samplePixel.y = (uint)(Random01(seed) * GetInvViewportSize().y);
    samplePixel = min(samplePixel, (uint2)GetViewportSize() - 1);
    float2 sampleUV = ((float2)samplePixel + 0.5f) * GetInvViewportSize();

```

```

uint weightOffsets[NUM_TRANSITIONS] = rtxns::UnpackArray<NUM_TRANSITIONS_ALIGN4,
NUM_TRANSITIONS>(TrainingData.WeightOffsets);
uint biasOffsets[NUM_TRANSITIONS] = rtxns::UnpackArray<NUM_TRANSITIONS_ALIGN4,
NUM_TRANSITIONS>(TrainingData.BiasOffsets);
Network network = Network(MLPParams, MLPParamsGradients, weightOffsets, biasOffsets);

rtxns::HCoopVec<INPUT_NEURONS> inputFeatures;
MLPFeatureExtractor featureExtractor = MLPFeatureExtractor(DepthTexture, PointClampedSampler, FrameData.Proj, FrameData.InvProj);
featureExtractor.Extract(sampleUV, samplePixel, TrainingData.AORadius, inputFeatures);

var hiddenActivation = rtxns::mlp::ReLUAct<half, HIDDEN_NEURONS>();
var outputActivation = rtxns::mlp::SigmoidAct<half, OUTPUT_NEURONS>();
float referenceAO = ReferenceTexture.SampleLevel(PointClampedSampler, sampleUV, 0);
float predictedAO = network.forward(inputFeatures, hiddenActivation, outputActivation)[0];
float lossGradient = rtxns::mlp::LossDerivFromVector<1, rtxns::mlp::Huber<float, 1>>(referenceAO, predictedAO, LOSS_SCALE /
TrainingData.BatchSize);
network.backward(inputFeatures, hiddenActivation, outputActivation, rtxns::HCoopVec<OUTPUT_NEURONS>(lossGradient));
}

```

HZB.slang

```

#define FFX_GPU
#define FFX_HLSL
#include "Interop/SPD/ffx_core.h"
groupshared FfxUInt32 spdCounter;
groupshared FfxFloat32 spdIntermediateR[16][16];
groupshared FfxFloat32 spdIntermediateG[16][16];
groupshared FfxFloat32 spdIntermediateB[16][16];
groupshared FfxFloat32 spdIntermediateA[16][16];

FfxFloat32x4 SpdLoadSourceImage(FfxInt32x2 tex, FfxUInt32 slice)
{
    RWTexture2D<float> HZBOutput_0 = ResourceDescriptorHeap[SpdData.HZBOutputIndex[0].x];
    return HZBOutput_0[tex];
}

FfxFloat32x4 SpdLoad(FfxInt32x2 tex, FfxUInt32 slice)
{
    globallycoherent RWTexture2D<float> HZBOutput_6 = ResourceDescriptorHeap[SpdData.HZBOutputIndex[6].x];
    return HZBOutput_6[tex];
}

void SpdStore(FfxInt32x2 pix, FfxFloat32x4 outValue, FfxUInt32 mip, FfxUInt32 slice)
{
    if (mip == 5)
    {
        globallycoherent RWTexture2D<float> HZBOutput_6 = ResourceDescriptorHeap[SpdData.HZBOutputIndex[6].x];
        HZBOutput_6[pix] = outValue.r;
    }
    else
    {
        RWTexture2D<float> HZBOutput = ResourceDescriptorHeap[SpdData.HZBOutputIndex[mip + 1].x];
        HZBOutput[pix] = outValue.r;
    }
}

void SpdIncreaseAtomicCounter(FfxUInt32 slice)
{
    globallycoherent RWStructuredBuffer<uint> spdGlobalAtomic = ResourceDescriptorHeap[SpdData.GlobalAtomicIndex];
    InterlockedAdd(spdGlobalAtomic[slice], 1, spdCounter);
}

FfxUInt32 SpdGetAtomicCounter()
{
    return spdCounter;
}

void SpdResetAtomicCounter(FfxUInt32 slice)
{
    globallycoherent RWStructuredBuffer<uint> spdGlobalAtomic = ResourceDescriptorHeap[SpdData.GlobalAtomicIndex];
    spdGlobalAtomic[slice] = 0;
}

FfxFloat32x4 SpdReduce4(FfxFloat32x4 v0, FfxFloat32x4 v1, FfxFloat32x4 v2, FfxFloat32x4 v3)
{
    return min(min(v0, v1), min(v2, v3));
}

FfxFloat32x4 SpdLoadIntermediate(FfxUInt32 x, FfxUInt32 y)
{

```

```

    return FfxFloat32x4(spdIntermediateR[x][y], spdIntermediateG[x][y], spdIntermediateB[x][y], spdIntermediateA[x][y]);
}

void SpdStoreIntermediate(FfxUInt32 x, FfxUInt32 y, FfxFloat32x4 value)
{
    spdIntermediateR[x][y] = value.x;
    spdIntermediateG[x][y] = value.y;
    spdIntermediateB[x][y] = value.z;
    spdIntermediateA[x][y] = value.w;
}

#include "Interop/SPD/ffx_spd.h"
[numthreads(256, 1, 1)]
void HZBBuildCS(uint3 WorkGroupId : SV_GroupID, uint LocalThreadId : SV_GroupIndex)
{
    SpdDownsample(
        WorkGroupId.xy,
        LocalThreadId,
        SpdData.Mips,
        SpdData.NumWorkGroups,
        WorkGroupId.z,
        SpdData.WorkGroupOffset
    );
}

```

ДОДАТОК Е

Графічний рушій реального часу з використанням GPU-керованого
рендерингу та нейронних мереж на базі Vulkan API

Презентація дипломного проєкту
ІАЛЦ.467800.010 Д6

Аркушів 9

Київ 2026

ГРАФІЧНИЙ РУШІЙ РЕАЛЬНОГО ЧАСУ З ВИКОРИСТАННЯМ GPU-КЕРОВАНОГО РЕНДЕРИНГУ ТА НЕЙРОННИХ МЕРЕЖ НА БАЗІ VULKAN API

Автор: студент групи KB-23, Литвин Станіслав Романович
Керівник: доцент каф. СПСКС, Клятченко Ярослав Михайлович

АКТУАЛЬНІСТЬ ТЕМИ

- Зростання складності 3D-сцен:
 - Велика кількість об'єктів і геометрії;
 - Складні матеріали та освітлення;
 - Вимога стабільного часу кадру.
- Обмежена гнучкість готових рушіїв;
- Потреба у власному рушії:
 - Контроль над рендеринг-проходами;
 - Інтеграція експериментальних методів.



2

ПОСТАНОВКА ЗАДАЧІ

Мета: розробити графічний рушій з GPU-керованою архітектурою рендерингу та реалізувати сучасні підходи до візуалізації тривимірних сцен.

Завдання:

- Обрати засоби реалізації графічного рушія;
- Спроекувати GPU-керовану архітектуру рендеринг-конвеєра;
- Реалізувати механізми відсікання невидимої геометрії;
- Реалізувати основні проходи формування зображення;
- Інтегрувати нейромережеві методи;
- Оцінити продуктивність та якість результатів.

3

КОНВЕЄР РЕНДЕРИНГУ

- CPU — координація кадру
- GPU — формування обсягу роботи
- Гібридний підхід:
 - Растеризація;
 - Трасування променів;
 - Нейромережеві проходи.
- Визначення видимості перед затіненням
- Відкладений рендеринг
- Пряме та непряме освітлення
- Реконструкція та постобробка фінального кадру



4

GPU-КЕРОВАННИЙ РЕНДЕРИНГ

Mesh Shading

- Заміна класичних стадій обробки геометрії:
 - Task Shader / Mesh Shader.
- Геометрія поділяється на мешлети:
 - Компактні блоки трикутників;
 - Незалежна паралельна обробка на GPU.
- Видимість на рівні підмножин меша
- meshoptimizer: побудова та оптимізація мешлетів
- Розмір мешлета: 64 вершини / 126 трикутників

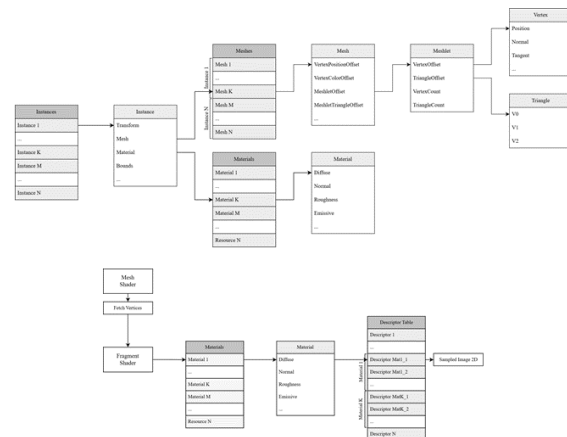


5

GPU-КЕРОВАННИЙ РЕНДЕРИНГ

Bindless

- Дані сцени зберігаються в індексованих GPU-буферах:
 - Екземпляри;
 - Меші;
 - Мешлети;
 - Матеріали.
- Матеріали зберігають індекси текстур у глобальній таблиці дескрипторів
- Шейдери отримують потрібні дані через індекси, без окремого прив'язування ресурсів для кожного об'єкта

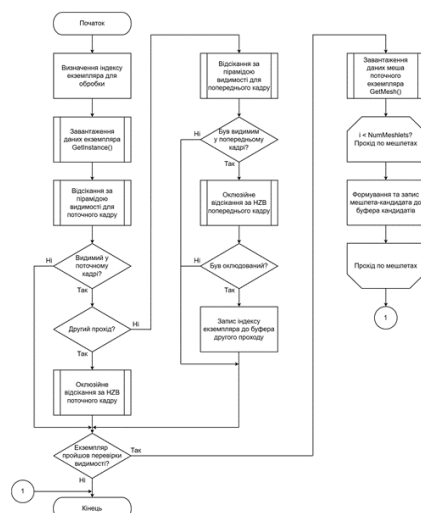


6

GPU-КЕРОВАНИЙ РЕНДЕРИНГ

Відсікання

- Обчислювальні GPU-проходи:
 - 1 потік = 1 екземпляр / мешлет
- Рівні відсікання:
 - Екземпляри сцени;
 - Мешлети видимих екземплярів.
- 1-й прохід: перевірка за HZB попереднього кадру
- 2-й прохід: повторна перевірка геометрії, відсіченої на першому етапі



7

GPU-КЕРОВАНИЙ РЕНДЕРИНГ

Відсікання

- За пірамідою видимості:**
 - Обмежувальний об'єм — орієнтований паралелепіпед
 - Перевіряється відносно 6 площин камери:
 - Ліва / Права;
 - Верхня / Нижня;
 - Ближня / Дальня.
- Оклюдійне відсікання:**
 - HZB для швидкої перевірки глибини
 - AMD FidelityFX SPD — один прохід, всі MIP-рівні
 - Кожен рівень HZB узагальнює глибину блоків 2x2
 - Вибір MIP-рівня: покриття до 4x4 текселів
 - Консервативна перевірка оклюзії

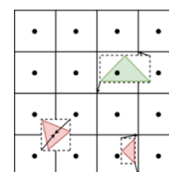
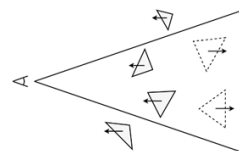


8

GPU-КЕРОВАНИЙ РЕНДЕРИНГ

Відсікання

- На рівні трикутників:**
 - SV_CullPrimitive у Mesh Shader
 - Вхідні дані: вершини трикутника у просторі відсікання
 - Послідовно перевіряються:
 - Задні грані;
 - Вироджені трикутники;
 - Повний вихід за межі екрана;
 - Субпіксельний розмір.

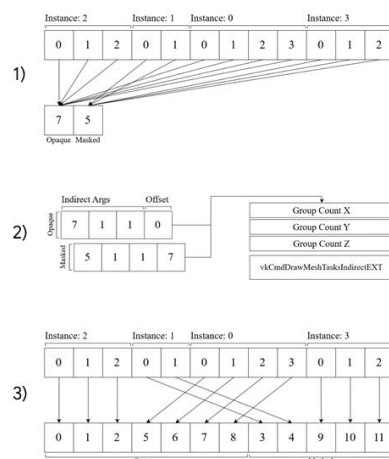


9

GPU-КЕРОВАННИЙ РЕНДЕРИНГ

Розподіл мешлетів

- Кожен мешлет класифікується за типом растеризації
- Виконується у три етапи:
 - Підрахунок кількості мешлетів у групах;
 - Резервування діапазонів у вихідному буфері;
 - Запис індексів мешлетів у відповідні діапазони.

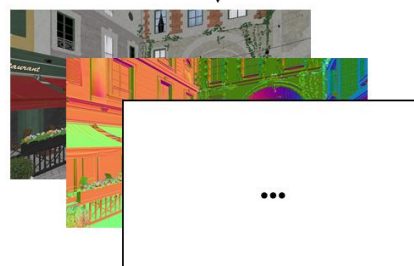
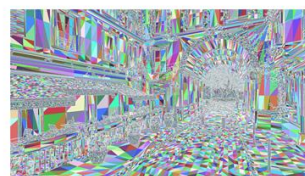
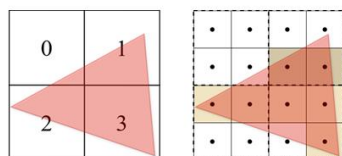


10

ВІДКЛАДЕНИЙ РЕНДЕРИНГ

Visibility Buffer

- Растеризація — запис 32-бітного ідентифікатора:
 - 25 біт — індекс мешлета;
 - 7 біт — локальний індекс трикутника.
- Атрибути поверхні відновлюються аналітично за барицентричними координатами
- Без надлишкового затінення та втрат на 2x2 фрагментних групах



11

ТІНІ З ТРАСУВАННЯМ ПРОМЕНІВ

- Жорсткі тіні:
 - Промінь у напрямку джерела світла;
 - Перевірка першого перетину з геометрією.
- М'які тіні:
 - Вибірка напрямку в межах конуса;
 - Ширина конуса визначає розмір напівтіні.

$$\omega = (\sin \theta \cos \phi, \sin \theta \sin \phi, \cos \theta),$$

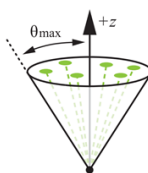
$$\cos \theta = (1 - \xi_1) + \xi_1 \cos \theta_{\max},$$

$$\sin \theta = \sqrt{1 - \cos^2 \theta},$$

$$\phi = 2\pi \xi_2,$$

де $\xi_1, \xi_2 \in [0, 1]$ — випадкові величини;

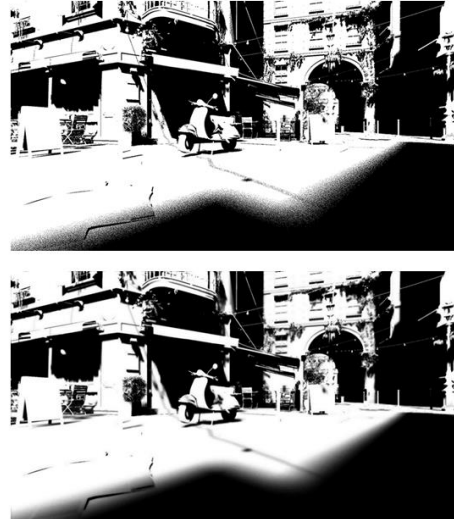
$\theta_{\max}$ — максимальний кут розкриття конуса



12

ТІНІ З ТРАСУВАННЯМ ПРОМЕНІВ

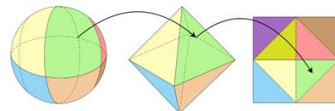
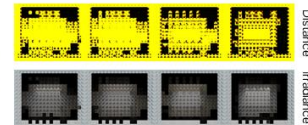
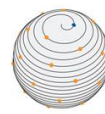
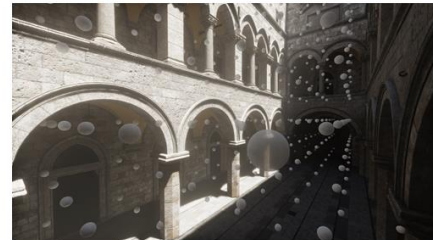
- Напрямки променів задаються просторово-часовим синім шумом (STBN):
 - Рівномірніше розміщення семплів;
 - Швидша збіжність під час накопичення;
 - Зменшення мерехтіння між кадрами.
- Фільтрація маски тіней:
 - Накопичення історії попередніх кадрів;
 - Багатопрхідне двобічне розмиття;
 - Edge-aware перевірки зберігають межі геометрії.



13

ГЛОБАЛЬНЕ ОСВІТЛЕННЯ

- Сцена покривається 3D-сіткою зондів
- Кожен зонд трасує промені по сферичному розподілу Фібоначчі
- Атласи зондів:
 - Irradiance — накопичене непряме освітлення;
 - Distance — інформація про видимість і відстані.
- Октаедричне кодування сферичних даних
- Освітлення точки: зважена інтерполяція між 8 найближчими зондами



$$E(X) = \frac{\sum w_i E_i}{\sum w_i}$$

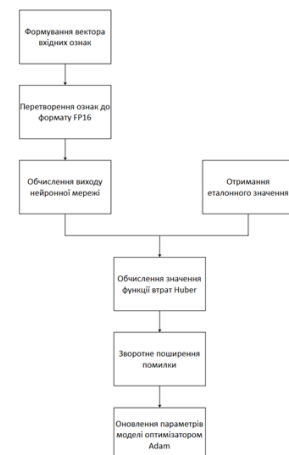
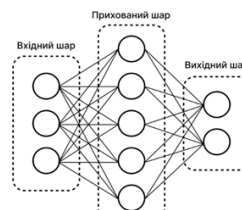
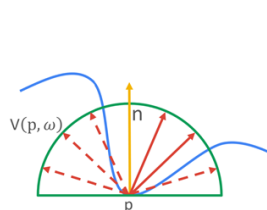
де E_i — опроміненість i -го зонда;
 w_i — повна вага i -го зонда;

14

НЕЙРОМЕРЕЖЕВЕ НАВКОЛИШНЄ ЗАТІНЕННЯ

- АО оцінює перекриття півсфери над точкою поверхні:

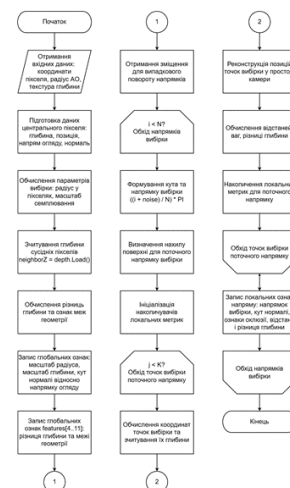
$$A(p, n) = \frac{1}{\pi} \int_{\Omega} V(p, \omega) (n \cdot \omega) d\omega$$
- MLP інтегровано в шейдери через RTX Neural Shading
- Приховані шари: ReLU, ініціалізація Каймінга Хе
- Вихідний шар: Sigmoid для діапазону [0;1]
- Навчання: Huber Loss, Adam та динамічне планування швидкості навчання



15

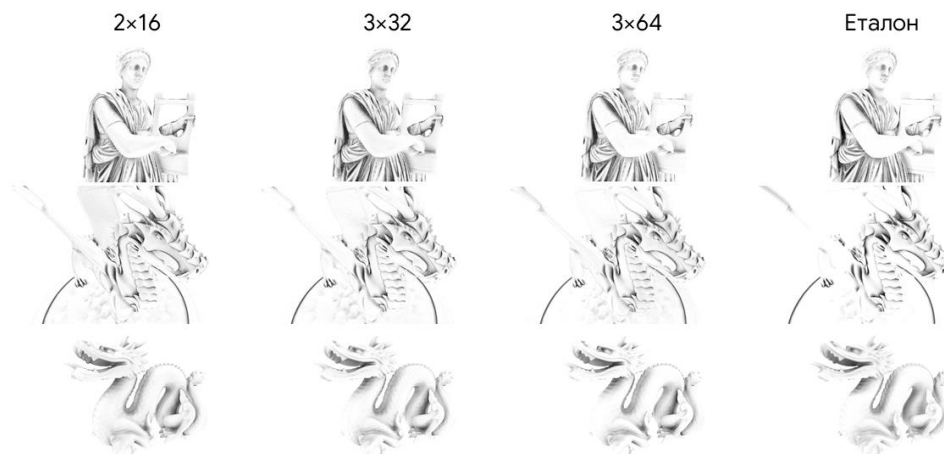
НЕЙРОМЕРЕЖЕВЕ НАВКОЛИШНЄ ЗАТІНЕННЯ

- Набір значень глибини / нормалі — недостатній:
 - Слабке узагальнення для різних форм геометрії;
 - Гало-артефакти біля меж об'єктів.
- Формуються дві групи ознак:
 - Глобальні — параметри центрального пікселя та його оточення;
 - Напрямкові — локальна геометрія вздовж напрямків вибірки.
- Ознаки описують:
 - Перепади глибини;
 - Відстані до сусідньої геометрії;
 - Можливе перекриття поверхні.
- Вихід — вектор ознак для MLP у форматі FP16



16

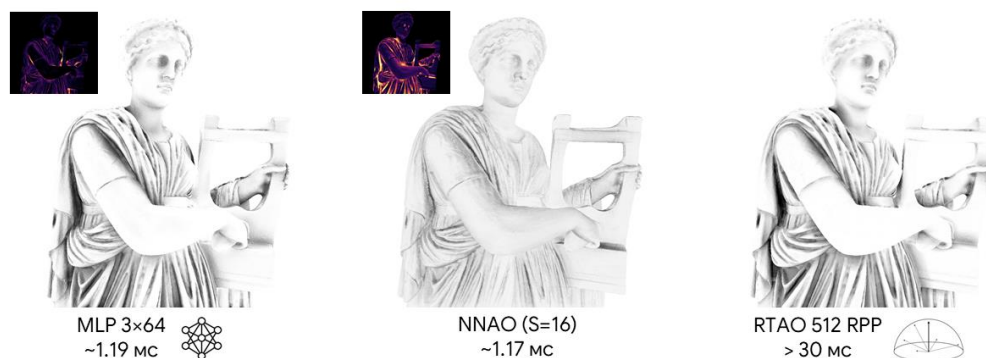
НЕЙРОМЕРЕЖЕВЕ НАВКОЛИШНЄ ЗАТІНЕННЯ



17

НЕЙРОМЕРЕЖЕВЕ НАВКОЛИШНЄ ЗАТІНЕННЯ

- Нейромережева апроксимація наближає результат до трасованого еталона
- Обмеження: екранний простір та залежність від доступних depth-ознак



18

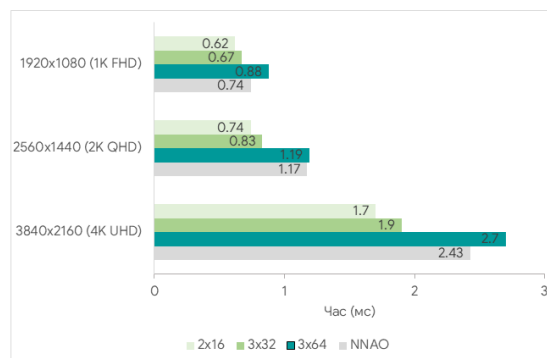
НЕЙРОМЕРЕЖЕВЕ НАВКОЛИШНЄ ЗАТІНЕННЯ

- Зі збільшенням розмірності мережі:
 - Зменшується значення похибки;
 - Краще зберігаються локальні деталі;
 - Зростає вартість інференсу.
- 3×32 — раціональний баланс між якістю апроксимації та швидкодією

$$MAE = \frac{1}{N} \sum_{i=1}^N |y_i - \hat{y}_i|$$

$$RMSE = \sqrt{\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2}$$

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}$$

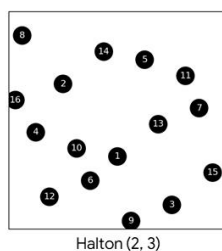


	MAE ↓	RMSE ↓	SSIM ↑
2×16	0.0297	0.0476	0.8588
3×32	0.0245	0.0391	0.8716
3×64	0.0205	0.0338	0.8792

19

МАСШТАБУВАННЯ ТА РЕКОНСТРУКЦІЯ

- Інтегровано NVIDIA DLSS Super Resolution
- Вхідні дані: колір, глибина, вектори руху, jitter-зміщення
- Jitter формується за послідовністю Halton (2, 3)
- Результат — вища якість фінального кадру при меншій вартості рендерингу



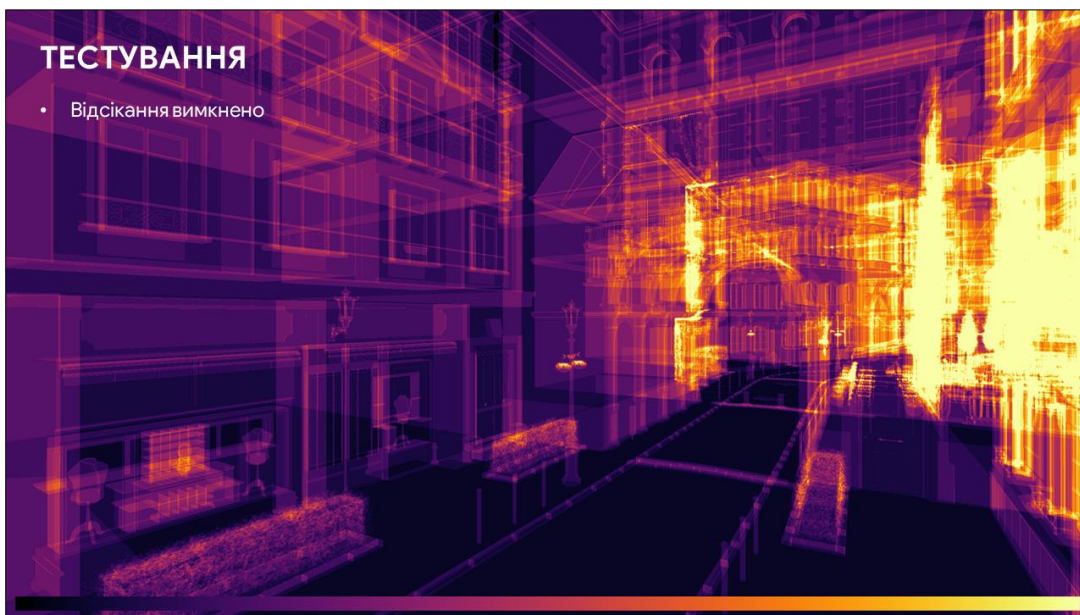
20



21

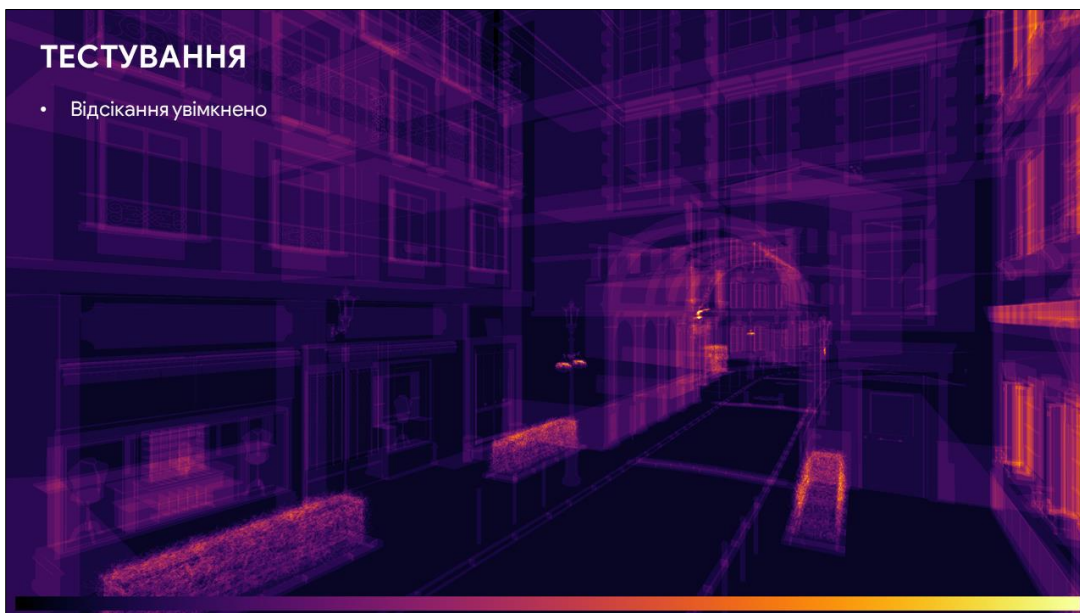
ТЕСТУВАННЯ

- Відсікання вимкнено



ТЕСТУВАННЯ

- Відсікання увімкнено

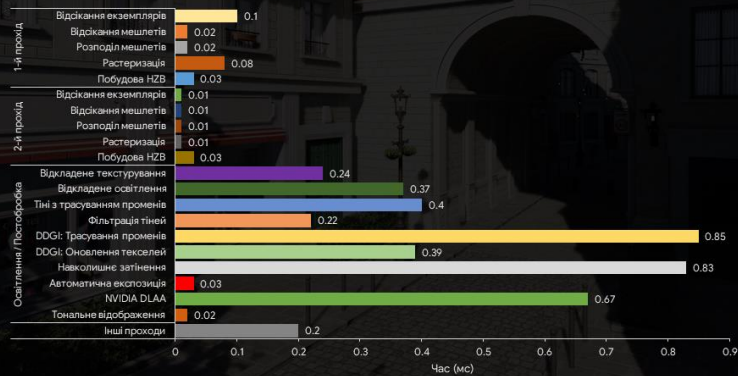


ТЕСТУВАННЯ

	Кількість екземплярів	Кількість мешлетів	Кількість трикутників	Час растеризації (мс)
Без відсікання	3815	288924	23811722	1,26
Відсікання за пірамідою видимості	1929	131559	10757574	0,82 (-35%)
Оклюдційне відсікання	265	4596	252636	0,10 (-88%)
Відсікання трикутників	265	4596	96761	0,08 (-20%)

ПРОДУКТИВНІСТЬ

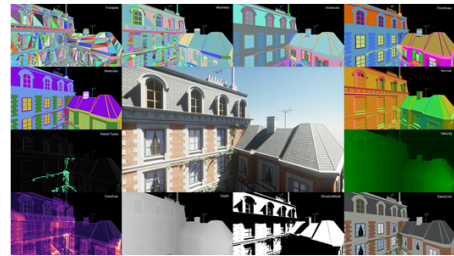
- Роздільна здатність: 2560×1440
- GPU: NVIDIA GeForce RTX 5070 Ti
- DDGI: 32×16×32, 128 променів на зонд



25

ВИСНОВКИ

- **Розроблений рушій забезпечує:**
 - Ефективне використання ресурсів GPU;
 - Масштабовану структуру рендеринг-конвеєра;
 - Можливість подальшого розширення.
- **Область використання:**
 - Інтерактивна візуалізація складних 3D-сцен;
 - Прототипування графічних застосунків;
 - Дослідження сучасних методів рендерингу.
- **Подальший розвиток:**
 - Variable Rate Shading (VRS);
 - GPU-кероване планування задач;
 - Розширення неймережевих методів на інші етапи рендерингу.



26