

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «магістр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

**«Розробка системи ефективного вибору алгоритмів
для відсікання геометрії у рушіях рендерингу
реального часу»**

Виконав студент групи ПЗм-24-2
ГОРШЕНІН Михайло Олександрович

Керівник роботи:
ЛОБАНЧИКОВА Надія Миколаївна

Рецензент:
ЄФРЕМОВ Юрій Миколайович

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

зав. кафедри інженерії
програмного забезпечення
Тетяна ВАКАЛЮК
«18» вересня 2025 р.

ЗАВДАННЯ

на кваліфікаційну роботу

Здобувач вищої освіти: **ГОРШЕНІН Михайло Олександрович**

Керівник роботи: **ЛОБАНЧИКОВА Надія Миколаївна**

Тема роботи: **«Розробка системи ефективного вибору алгоритмів для відсікання геометрії у рушіях рендерингу реального часу»**, затверджена Наказом закладу вищої освіти від **«18» вересня 2025 р. № 491/с.**

Вихідні дані для роботи: Об'єкт дослідження – процес відсікання невидимої геометрії у системах рендерингу реального часу. Предмет дослідження – алгоритми та методи відсікання геометрії за видимістю (frustum culling, distance culling, оклюзійне відсікання, ієрархічні структури), їх ефективність при виконанні на CPU та GPU, а також критерії вибору оптимальної стратегії відсікання залежно від параметрів сцени та апаратної платформи.

Консультанти з магістерської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Лобанчикова Н.М.	20.09.2025р.	20.09.2025р.
2	Лобанчикова Н.М.	27.10.2025р.	27.10.2025р.
3	Лобанчикова Н.М.	04.11.2025р.	04.11.2025р.

РЕФЕРАТ

Кваліфікаційна робота магістра спеціальності 121 «Інженерія програмного забезпечення» присвячена дослідженню та розробці системи ефективного вибору алгоритмів для відсікання геометрії у рушіях рендерингу реального часу. Робота складається з програмного прототипу та пояснювальної записки.

Пояснювальна записка містить 88 сторінок, 16 ілюстрацій, 20 таблиць, 4 лістинга, 30 використаних джерел та 2 додатки.

У першому розділі розглянуто проблему видимості у тривимірних сценах та проаналізовано сучасні підходи до відсікання геометрії, включно з фрустумним, дистанційним та оклюзійним відсіканням. Досліджено алгоритми відсікання на CPU та GPU, особливості їх паралельного виконання, використання SIMD-векторизації, ієрархічних структур граничних об'ємів та ієрархічного Z-буфера.

Другий розділ присвячено проектуванню системи вибору алгоритмів відсікання та архітектури користувацького конвеєра рендерингу.

У третьому розділі описано програмну реалізацію запропонованих методів у вигляді користувацького Scriptable Render Pipeline в середовищі Unity, наведено результати експериментального дослідження продуктивності різних стратегій відсікання та проведено їх порівняльний аналіз.

У висновках показано, що розроблена система дозволяє адаптивно обирати оптимальний набір алгоритмів відсікання залежно від параметрів сцени, забезпечуючи підвищення продуктивності рендерингу в реальному часі. Отримані результати можуть бути використані у подальших наукових дослідженнях та практичній розробці графічних рушіїв.

КЛЮЧОВІ СЛОВА: ВІДСІКАННЯ ГЕОМЕТРІЇ, ІЄРАРХІЧНИЙ Z-БУФЕР, ІЄРАРХІЯ ГРАНИЧНИХ ОБ'ЄМІВ, GPU-ОРІЄНТОВАНИЙ РЕНДЕРИНГ, РЕНДЕРИНГ РЕАЛЬНОГО ЧАСУ, UNITY.

ABSTRACT

The qualification paper of a master's student in specialty 121 "Software Engineering" is devoted to the research and development of a system for efficient selection of geometry culling algorithms in real-time rendering engines. The work consists of a software prototype and an explanatory report.

The explanatory report comprises 88 pages, 16 figures, 20 tables, four listings, 30 references, and two appendices.

The first section examines the visibility problem in three-dimensional scenes and analyzes modern approaches to geometry culling, including frustum culling, distance culling, and occlusion culling. It also investigates CPU- and GPU-based culling algorithms, features of their parallel execution, the use of SIMD vectorization, hierarchical bounding volume structures, and the hierarchical Z-buffer.

The second section is devoted to the design of the culling algorithm selection system and the architecture of a custom rendering pipeline.

The third section describes the software implementation of the proposed methods in the form of a custom Scriptable Render Pipeline in the Unity environment, presents the results of an experimental performance study of various culling strategies, and provides their comparative analysis.

The conclusions demonstrate that the developed system enables adaptive selection of an optimal set of geometry culling algorithms depending on scene parameters, ensuring improved real-time rendering performance. The obtained results may be used in further scientific research and in the practical development of rendering engines.

KEYWORDS: GEOMETRY CULLING, HIERARCHICAL Z-BUFFER, BOUNDING VOLUME HIERARCHY, GPU-DRIVEN RENDERING, REAL-TIME RENDERING, UNITY.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ	6
ВСТУП.....	7
РОЗДІЛ 1. ПОСТАНОВКА ЗАДАЧІ ОРГАНІЗАЦІЇ ВІДСІКАННЯ НЕВИДИМИХ ОБ'ЄКТІВ СЦЕНИ ТА АНАЛІЗ СУЧАСНИХ ПІДХОДІВ	9
1.1 Проблема видимості в тривимірних сценах	9
1.2 Огляд літератури та існуючих рішень.....	11
1.3 Паралельна модель виконання на GPU	26
Висновки до першого розділу	34
РОЗДІЛ 2. ПРОЕКТУВАННЯ СИСТЕМИ ВИБОРУ АЛГОРИТМІВ ВІДСІКАННЯ .	36
2.1 Архітектура конвеєра рендерингу	36
2.2 Застосування шаблону проектування «Стратегія»	38
2.3 Модель вибору стратегії відсікання	43
Висновки до другого розділу	45
РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ МЕТОДУ ТА ПРОВЕДЕННЯ ЕКСПЕРИМЕНТУ ПЕРЕВІРЦІ ЙОГО ПРОДУКТИВНОСТІ	47
3.1 Реалізація алгоритмів відікання.....	47
3.2 Умови експерименту та методика вимірювань	51
3.3 Аналіз ефективності CPU проти GPU	59
3.4 Результати оклюзійного відсікання.....	63
Висновки до третього розділу	66
ВИСНОВКИ	68
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	73
ДОДАТКИ	78

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

AABB (Axis-Aligned Bounding Box) – осьово-орієнтований обмежувальний паралелепіпед; простий граничний об'єм для грубих тестів видимості.

BVH (Bounding Volume Hierarchy) – ієрархія граничних об'ємів; дерево, що прискорює відсікання та інші запити просторової видимості.

CPU (Central Processing Unit) – центральний процесор.

GPU (Graphics Processing Unit) – графічний процесор.

Dispatch – запуск обчислювального шейдера (планування груп потоків) на GPU.

Warp (NVIDIA) / Wavefront (AMD) – апаратна група потоків, що виконує інструкції синхронно або частково синхронно на рівні SIMD-процесорів GPU.

SIMD (Single Instruction Multiple Data) – модель виконання, що дозволяє однією інструкцією обробляти кілька елементів даних.

Occlusion culling – оклюзійне відсікання: відкидання об'єктів, повністю перекритих ближчою геометрією.

HZB (Hierarchical Z-Buffer) – ієрархічний Z-буфер; піраміда міп-рівнів глибини, що використовується для швидких консервативних оклюзійних тестів.

Reverse-Z – конвенція глибини, де більші значення відповідають ближчим відстаням ($1 = \text{near}$, $0 = \text{far}$), що покращує точність.

SRP (Scriptable Render Pipeline) – програмований конвеєр рендерингу Unity.

GPU-driven rendering – підхід, де відбір видимих об'єктів і формування команд рендерингу виконується на GPU з мінімальною участю CPU.

Indirect draw / DrawMeshInstancedIndirect – непрямий виклик рендерингу, де параметри малювання беруться з буфера, підготовленого на GPU.

Geometry (геометрія сцени) – набір примітивів (переважно трикутників), їхніх інстансів та пов'язаних даних (вершини/індекси/трансформи), що беруть участь у рендерингу сцени.

ВСТУП

Сучасні 3D-сцени у відеоіграх, кіновиробництві та індустріальних симуляціях можуть містити десятки мільйонів трикутників, хоча з конкретної позиції камери видно лише незначну їх частину. При цільовій продуктивності 60+ кадрів на секунду наївний підхід із рендерингом усієї геометрії є неприйнятним. Визначення того, яку геометрію відобразити, а яку пропустити, є сутністю задачі видимого відсікання (visibility culling). [1, 2] Популярні ігрові рушії (Unity, Unreal тощо) містять вбудовані системи відсікання, проте вони часто оптимізовані під конкретні платформи, слабо документовані та налаштовуються емпірично. [4, 5] Не існує систематичного підходу для вибору стратегії відсікання залежно від характеристик сцени. Це обумовлює актуальність розробки методики адаптивного вибору алгоритмів відсікання, яка б враховувала кількісні та просторові параметри сцени та дозволяла автоматично обирати оптимальний набір алгоритмів для конкретних умов рендерингу.

Метою роботи є розробка методики ефективного вибору алгоритмів відсікання геометрії за видимістю для довільної сцени в системах рендерингу реального часу

Для досягнення мети поставлено наступні задачі:

- проаналізувати існуючі алгоритми відсікання геометрії та визначити фактори, що впливають на їх ефективність;
- дослідити умови ефективності використання ієрархічних структур даних (BVH) для прискорення відсікання;
- визначити точку зміни переваги продуктивності між CPU- та GPU-орієнтованими підходами до відсікання;
- проаналізувати вплив синхронізації CPU–GPU на продуктивність гібридного конвеєра відсікання;
- розробити модель вибору стратегії відсікання на основі параметрів сцени та реалізувати її програмно;

- експериментально перевірити ефективність запропонованої методики на різних типах сцен.

Об'єкт дослідження – процес відсікання невидимої геометрії в системах рендерингу реального часу.

Предмет дослідження – методи та алгоритми вибору стратегії відсікання геометрії залежно від характеристик сцени.

У роботі використано: аналітичне моделювання для побудови моделі витрат CPU та GPU; експериментальні методи для вимірювання продуктивності алгоритмів; порівняльний аналіз для оцінки ефективності різних стратегій.

У роботі вперше запропоновано та експериментально підтверджено аналітичну модель витрат яка дозволяє кількісно визначити точку зміни переваги продуктивності між CPU- та GPU-відсіканням. Також встановлено та кількісно підтверджено, що використання BVH на GPU не дає виграшу для відсікання об'єктів у загальному випадку через розрізнені звернення до пам'яті. Додатково розроблено адаптивний метод вибору стратегій відсікання у вигляді дерева рішень, що на основі параметрів сцени (кількість об'єктів, рівень оклюзії, тип сцени) автоматично обирає оптимальний набір алгоритмів досі не реалізований у сучасних рушіях рендерингу загального призначення.

Практичне значення одержаних результатів – було розроблено програмний прототип системи адаптивного вибору алгоритмів відсікання у вигляді користувацького програмованого конвеєра рендерингу Unity з відкритим вихідним кодом. Система забезпечує автоматичний вибір оптимальної стратегії відсікання залежно від параметрів сцени та може бути інтегрована в реальні проєкти або використана для подальших досліджень. Сформульовано практичні рекомендації щодо вибору стратегій відсікання для різних типів сцен: від малих інтер'єрів до масивних відкритих просторів.

РОЗДІЛ 1. ПОСТАНОВКА ЗАДАЧІ ОРГАНІЗАЦІЇ ВІДСІКАННЯ НЕВИДИМИХ ОБ'ЄКТІВ СЦЕНИ ТА АНАЛІЗ СУЧАСНИХ ПІДХОДІВ

1.1 Проблема видимості в тривимірних сценах

Визначимо функцію видимості $V(o, c)$, що для об'єкта o та камери c набуває значення true, якщо об'єкт видимий з даної позиції камери. На цій основі можна визначити множину видимих об'єктів S_v як підмножину усіх об'єктів сцени S , які задовольняють умову видимості, та множину відсічених $S_c = S - S_v$, тобто об'єктів, що залишаються невидимими. Кінцевою метою є точно відсіяти множину S_c без додаткових затрат часу, але на практиці досяжний лише консервативний підхід, коли алгоритми можуть помилятися в сторону залишення зайвих об'єктів (щоб нічого не пропустити) або, рідше, відсікати щось видиме. Такий підхід називають консервативним відсіканням. Темпоральні алгоритми відсікання не розглядаються; натомість використовується лише темпоральна стабілізація результатів, яка не змінює консервативність базових тестів [1].

Існує кілька основних типів алгоритмів, що визначають невидимі об'єкти (табл. 1.1) [1]. Одним із базових підходів є відсікання за фрустумом огляду, яке полягає у виключенні об'єктів, що знаходяться за межами об'єму видимості камери, сформованого у вигляді піраміди огляду. Іншим поширеним методом є відсікання за відстанню, що передбачає виключення об'єктів, розташованих далі за максимально допустиму дистанцію видимості. У ряді випадків застосовується відсікання дрібних об'єктів, при якому виключаються об'єкти, чий проєкційний розмір на екрані настільки малий, що вони майже не впливають на зображення. Важливе місце займає оклюзійне відсікання, спрямоване на виключення об'єктів, повністю перекритих ближчими до камери геометричними об'єктами. Окрему категорію становить відсікання за внеском у кадр, яке дозволяє виключати об'єкти, внесок яких у кінцеве зображення є мізерним, наприклад дуже прозорі або тимчасово невидимі об'єкти.

Таблиця 1.1

Типи відсікання геометрії

Тип відсікання	Вартість тесту	Точність	Хибні спрацювання	Хибні пропуски
Фрустум	Низька	Точна	Немає	Немає
Відстань	Дуже низька	Приблизна	Можливі	Немає
Дрібні об'єкти	Середня	Консерват.	Можливі	Немає
Оклюдія	Висока	Консерват.	Можливі	Можливі (рідко)

Примітка: Хибні спрацювання (False Positives) означають неправильно відсічені видимі об'єкти; хибні пропуски (False Negatives) – об'єкти, що залишені як видимі, хоча насправді вони невидимі (в консервативних алгоритмах зазвичай відсутні).

Видиме відсікання виправдане лише якщо сумарний час тестування плюс час рендерингу видимих об'єктів менший, ніж час рендерингу усіх об'єктів без відсікання [1]. Формально ця умова:

$$T_{\{cull\}(S)} + T_{\{render\}(S_v)} < T_{\{render\}(S)} \quad (1.1)$$

де $T_{\{cull\}(S)}$ – час, витрачений на відсікання множини S , а $T_{\{render\}(S_v)}$ – час рендерингу видимої підмножини об'єктів. Ця нерівність залежить від відношення $\frac{|S_v|}{|S|}$ (частки видимих об'єктів), обчислювальної вартості індивідуального тесту відсікання та вартості рендерингу одного об'єкта. Таким чином, при дуже простих об'єктах або низькій складності сцени відсікання може бути недоцільним, тоді як у складних насичених сценах ефективне відсікання є критично необхідним. Таким чином встановлено, що ефективність відсікання залежить від характеристик сцени та параметрів відсікання; ця залежність лягає в основу подальшого дослідження і розробки системи вибору оптимальної стратегії.

1.2 Огляд літератури та існуючих рішень

Ранні дослідження зосереджувались на відсіканні у статичних сценах і підготовчих обчисленнях. До таких підходів належить портальне відсікання, що передбачає поділ сцени на приміщення, з'єднані порталами (дверима або вікнами), з попереднім обрахунком видимості між ними [6]. Також використовувався підхід потенційно видимої множини (PVS), в якому для кожної області простору наперед зберігається перелік об'єктів, потенційно видимих з неї [7]. Значний внесок зробив ієрархічний Z-буфер — алгоритм Greene та ін. (1993), який запропонував використання буфера глибини низької роздільності для швидкої оцінки оклюзії [8, 34]. Подальший розвиток отримало когерентне ієрархічне відсікання, запропоноване Bittner та ін. (2004), яке поєднувало просторові і часові аспекти, використовуючи результати попереднього кадру для прогнозування видимості в наступному [9].

З ростом потужності GPU з'явилась тенденція зміщувати більше роботи на графічний процесор. GPU-driven rendering pipelines – концепція, де CPU лише готує дані, а відбір об'єктів і їх рендеринг керуються GPU [10, 35].

Найвідомішим рішенням на базі GPU-орієнтованого конвеєра є Nanite (Unreal Engine 5) – приклад віртуалізованої геометрії з агресивним відсіканням та вибором рівнів деталізації на рівні кластерів переважно на GPU [2]. Іншим рішенням, що показує дуже великий потенційний прріст продуктивності, але ще не використовується у рушіях загального прзначення – сіткові шейдери (Mesh Shaders) – сучасний програмований етап графічного конвеєра, що дозволяє реалізовувати власні схеми відбору примітивів і групової обробки геометрії на GPU [11, 36]. Єдиним загальноприйнятим GPU-орієнтованим підходом є Indirect Draw Calls – механізм, що дозволяє GPU ініціювати рендеринг множин об'єктів без участі CPU [12, 13].

Окрема ніша методів – software occlusion culling, коли оклюзійні тести виконуються програмно на CPU шляхом приблизної растеризації сцени. Через необхідність повільної растеризації та широку доступність GPU такі методи не є

пошреними. Прикладом є Masked Occlusion Culling (Intel) – високопродуктивний підхід, що використовує SIMD для швидкої растеризації примітивів у буфер видимості на CPU [14, 38]. Іншим рішенням є Umbra – приклад middleware-рішення, що комбінує просторові розбиття і перевірку оклюзії [15, 39]. Хоча програмне оклюзійне відсікання може дати великий приріст, інтеграція таких рішень у кероване середовище (як Unity/C#) може мати помітні накладні витрати через передачу даних та організацію пам'яті [21]. Це одна з причин, чому в цій роботі зроблено акцент на HZB-оклюзію з GPU-буфера глибини. Програмне оклюзійне відсікання все ще має можливі використання у носимих (wearable) або вбудованих пристроях без GPU, але стрімко витісняються наявністю портативних GPU (Nvidia Jetson Nano та інші) [8, 14].

Аналіз існуючих робіт показав, що більшість з них зосереджені на окремих методах або на порівнянні схожих підходів. Відсутні систематичні дослідження, які б порівняли CPU та GPU підходи до видимого відсікання, та визначили б точки зміни переваги у продуктивності між ними. Також майже не розглянуто питання адаптивного вибору стратегії: розробники не мають настанов, за яких умов краще вимкнути певний вид відсікання чи переключитися з CPU на GPU. Ця робота заповнює цю прогалину, пропонуючи уніфіковану систему, що на основі аналізу сцени обирає ефективну комбінацію алгоритмів відсікання.

Для пришвидшення тестів видимості використовуються ієрархічні структури, які організовують об'єкти сцени в просторі. Найчастіше використовувані підходи включають вісімкові дерева, kD-дерева, ієрархії граничних об'ємів (BVH), тощо. Особливо виділяється BVH – ієрархія AABV-контейнерів, яка показала високу середню продуктивність і гнучкість в побудові. BVH (Bounding Volume Hierarchy) будується рекурсивним поділом множини об'єктів на дві підмножини, огортаючи їх граничними рамками; завдяки цьому при відсіканні цілими піддеревами можна значно скоротити кількість окремих тестів. На сучасний момент BVH є однією з найпоширеніших структур прискорення для задач видимості та трасування променів [1, 18].

При використанні BVH для відсікання застосовується покроковий тест перетину вузла дерева з фрустумом: результатом може бути OUTSIDE (повністю поза), INSIDE (повністю всередині) або INTERSECT (часткове перетинання). Якщо вузол поза фрустумом – відсікаємо одразу ціле піддерево цього вузла (усі об'єкти в ньому гарантовано невидимі) [1]. Якщо вузол повністю всередині – приймаємо всі об'єкти цього піддерева як видимі без подальших тестів (вони гарантовано видимі). Якщо ж вузол частково перетинає межі фрустуму – рекурсивно переходимо до перевірки його нащадків (розгортаємо дерево глибше) (рис. 1.1). Такий алгоритм забезпечує найкращу складність близько $O(\log n)$ в ідеальному випадку (коли більшість вузлів відсікаються цілими піддеревами) і деградує до $O(n)$ у найгіршому (коли багато об'єктів лежать на межі фрустуму) [1]. Хоча BVH є незамінною для відсікання на CPU при великих n , на GPU її користь неочевидна через нерегулярний обхід дерева та розрізнені звернення до пам'яті, що також створює додаткові розгалуження. Це питання кількісно проаналізовано в Розділі 5.

Розглядаються наступні основні підходи до побудови BVH. На основі площі поверхні (SAH, Surface Area Heuristic) – жадібний алгоритм, який при кожному розбитті множини об'єктів вибирає розподіл, мінімізуючи очікувану площу поверхні об'ємів-нащадків (а отже й очікувану кількість «перевірок» у середньому). SAH забезпечує близьку до оптимальної в середньому ієрархію, але вимагає суттєвих обчислень при побудові [18]. За медіаною - простіший підхід, коли об'єкти сортуються за координатою вздовж обраної осі (найдовшої осі розподілу) і діляться навпіл. Забезпечує $O(n \log n)$ складність побудови і практично прийнятну якість дерева, хоча зазвичай поступається SAH за якістю [18, 32]. У реалізації, представлений в роботі, застосовано метод розподілу за медіаною з порогом мінімального числа об'єктів у листі, що дає збалансоване дерево приблизно за $O(n \log n)$ часу з лінійною витратою пам'яті [18]. Таким чином можна стверджувати, що використання просторових структур суттєво підвищує масштабованість відсікання, дозволяючи відкидати цілі кластери об'єктів, без необхідності у повному обході всього набору. Наразі передбачаємо, що такі

структури сильно прискорять відсікання на CPU через використання підходу “раннього виходу”. Однак на GPU ефективність ієрархічних методів залежить від архітектурних особливостей прискорювача (Це питання детально розглянуто далі, в розділі 5).

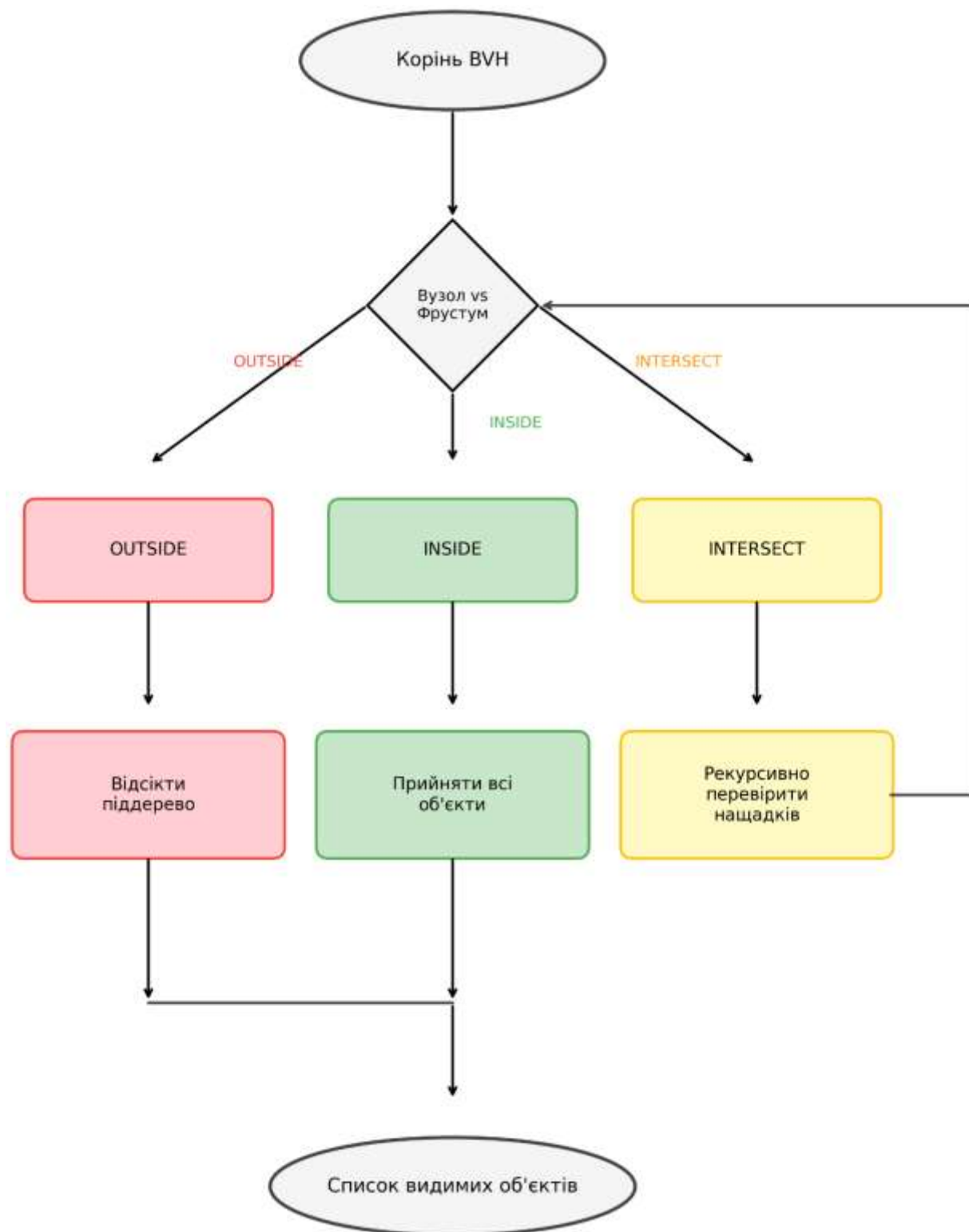


Рис 1.1. Схема роботи алгоритму обходу BVH при відсіканні за фрустумом

На CPU відсікання виконується в рамках послідовної моделі, у якій на кожному ядрі використовується власний потік інструкцій. На продуктивність такого підходу істотно впливають особливості організації кеш-пам'яті, SIMD-обчислень та багатоядерної архітектури. Багаторівнева ієрархія кешів L1, L2 і L3 зумовлює те, що упорядкований доступ до пам'яті, зокрема послідовні звернення до масивів, значно прискорює виконання обчислень. У зв'язку з цим алгоритми відсікання мають бути дружніми до кешу, наприклад шляхом зберігання даних об'єктів у масивах за схемою Struct-of-Arrays замість використання розріджених структур у пам'яті [1]. Важливу роль відіграють також SIMD-операції, оскільки сучасні CPU підтримують відповідні розширення, такі як SSE та AVX, що дозволяють за одну інструкцію обробляти відразу кілька числових значень [1]. Компілятори, зокрема Unity Burst, здатні автоматично векторизувати цикли, забезпечуючи суттєвий приріст швидкодії у задачах масового відсікання [16]. Додаткове прискорення досягається завдяки використанню багатоядерності, коли відсікання паралелізується шляхом розподілу об'єктів між потоками CPU. Прикладом такого підходу є Jobs System у рушії Unity, який дозволяє організувати паралельні обчислення з відносно малими накладними витратами на керування завданнями [17].

Графічні процесори працюють за принципом масового паралелізму: роботу можуть одночасно виконувати тисячі потоків. При організації потоків GPU групує їх в угруповання потоків (thread groups), а апаратно планує виконання у форматі warp / wavefront (N потоків, зазвичай 32 або 64) (рис. 1.2)[19, 20]. Якщо в шейдері є розгалуження (if/else), і різні потоки у warp/wavefront обирають різні гілки, виконання може відбуватись із серіалізацією гілок (частина потоків простоює). Така ситуація називається дивергенцією потоків і може уповільнювати виконання [19].

Для досягнення максимальної швидкості доступу GPU оптимізує сумісний доступ до пам'яті (memory coalescing). Це означає, що послідовні потоки, які звертаються до послідовних адрес, отримують дані ефективно; розрізнені

звернення (наприклад, довільний обхід дерева в пам'яті) працюють значно гірше на GPU [19].

Кожен виклик GPU-шейдера має постійну накладну вартість (зазвичай це десятки мікросекунд), незалежно від числа оброблюваних елементів. У випадках, коли основним масивом геометрії керує CPU також необхідно враховувати час на синхронізацію даних між CPU та GPU. Це робить GPU неефективним на дуже малих обсягах роботи, коли накладні витрати перевищують виграш від паралельності.

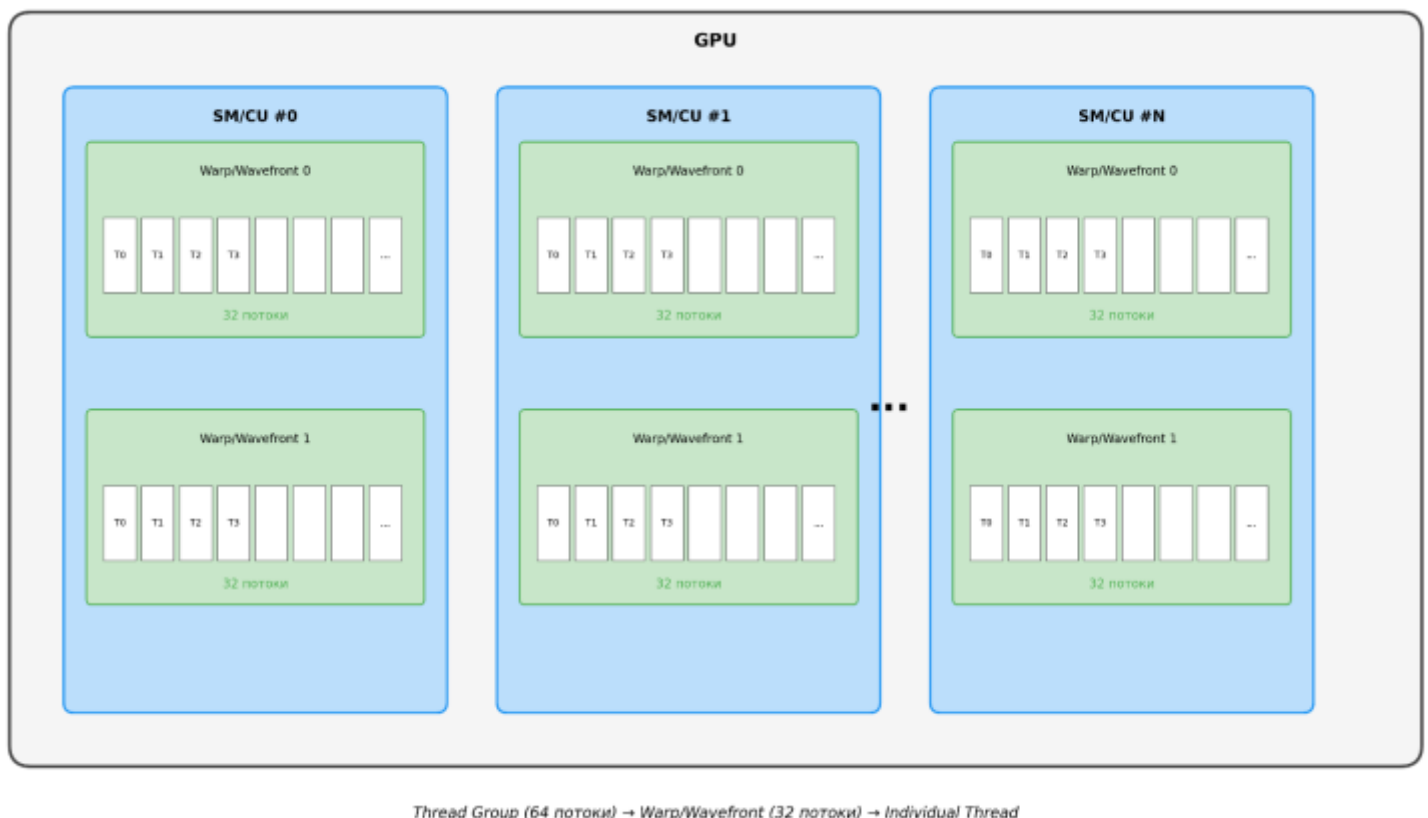


Рис 1.2. Діаграма виконання шейдера відсікання на GPU

У спільній роботі CPU та GPU виникають точки синхронізації, які можуть нівелювати виграш від паралельності. Виклики на кшталт `ComputeBuffer.SetData()` (передача даних на GPU) та `ComputeBuffer.GetData()` (отримання даних з GPU) можуть спричиняти очікування/синхронізацію з графічним пристроєм. Асинхронне читання (`AsyncGPUReadback`) дозволяє уникнути негайного блокування, але в такому випадку дані стають доступними з затримкою 1–2

кадри[21, 22]. Навіть опосередковані операції, як от використання Graphics.Blit() у Unity (для генерації текстур), можуть вставляти непомітну на перший погляд синхронізацію [23]. В проведених вимірюваннях накладні витрати на синхронізацію CPU-GPU перевищували власне обчислення відсікання до 25 разів. Це означає, що гібридні рішення (частина на CPU, частина на GPU) повинні дуже обережно враховувати ціну синхронізації при виборі стратегії.

Базовий алгоритм перевірки, чи найменший обмежувальний об'єм Axis-Aligned Bounding Box (AABB) повністю знаходиться поза пірамідою видимості, виконується послідовно. Спочатку обчислюються параметри шести площин фрустуму за матрицею виду-проєкції. Далі для кожної площини обчислюється підписана відстань від центру AABB до площини за формулою:

$$d = n * center + w \quad (1.2)$$

де n – нормаль площини, а w – зсув площини.

Обчислюється проєкція радіуса AABB на нормаль площини:

$$r = e_x |n_x| + e_y |n_y| + e_z |n_z| \quad (1.3)$$

де e_x, e_y, e_z – напівдовжини (екстенти) AABB по осях.

Після цього виконується порівняння, і якщо

$$d < -r \quad (1.4)$$

то весь об'єм знаходиться поза площиною (по зовнішній бік), тобто об'єкт гарантовано поза фрустумом і може бути відсічений. Якщо ж для жодної з площин ця умова не виконується, об'єкт перетинає або знаходиться всередині фрустуму. Цей тест виконує лише 18 операцій з плаваючою комою для одного AABB, тож є дуже швидким. На практиці всі 6 перевірок площин виконуються послідовно; якщо будь-яка з них дає результат “поза”, об'єкт відсікається [1].

На SIMD-архітектурі можна виконувати кілька перевірок одночасно: наприклад, AVX2 може обробляти 8 чисел з плаваючою комою паралельно, що дозволяє тестувати відразу кілька об'єктів у векторизованому циклі (рис. 1.3) [1].

У контексті Unity, використання Burst-компілятора й типів з Unity.Mathematics дозволяє автоматично векторизувати подібні цикли. Вимірювання показали приблизно п'ятикратне прискорення для тесту фрустуму при увімкненому Burst (у порівнянні з нативним C#). Це досягається як за рахунок SIMD, так і завдяки оптимізації пам'яті (вирівнювання даних, уникнення збору сміття) [16].

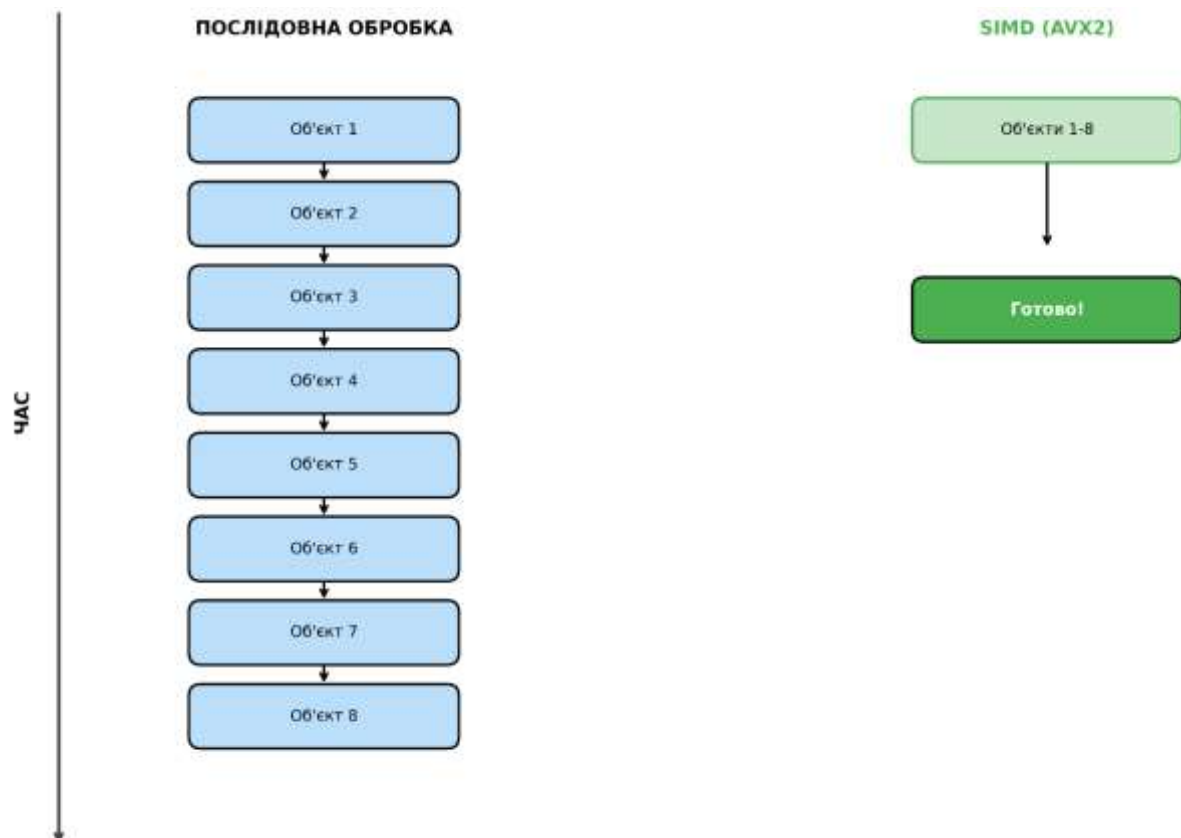


Рис 1.3. Ілюстрація SIMD-векторизації тестів фрустуму

Використання BVH уникнути індивідуальних тестів кожного об'єкта. Замість цього виконується рекурсивний обхід дерева. Якщо вузол поза фрустумом – відкидаємо одразу весь його вміст. Якщо вузол повністю всередині – приймаємо всі вкладені об'єкти як видимі без перевірки. Якщо частково перетинає – спускаємося на наступний рівень (перевіряємо нащадків) (рис. 1.4).

Середня теоретична складність такого алгоритму $O(k + m \log n)$, де k – кількість реально видимих об'єктів (листіків, які пройшли усі тести), а m – число вузлів, що опинилися на межі і потребували рекурсивної перевірки. У найгіршому

випадку (всі об'єкти на межі) складність все ще $O(n)$ [1]. Експерименти, наведені у третьому розділі, демонструють, що BVH дає найбільший виграш у сценах, де значна частина простору порожня, тобто багато піддерев може бути відсічено повністю. Важливо зазначити, що перевага BVH зростає зі збільшенням кількості об'єктів: від 2,4 разів при 10 000 об'єктів до 3,2 разів при 1 000 000 об'єктів.

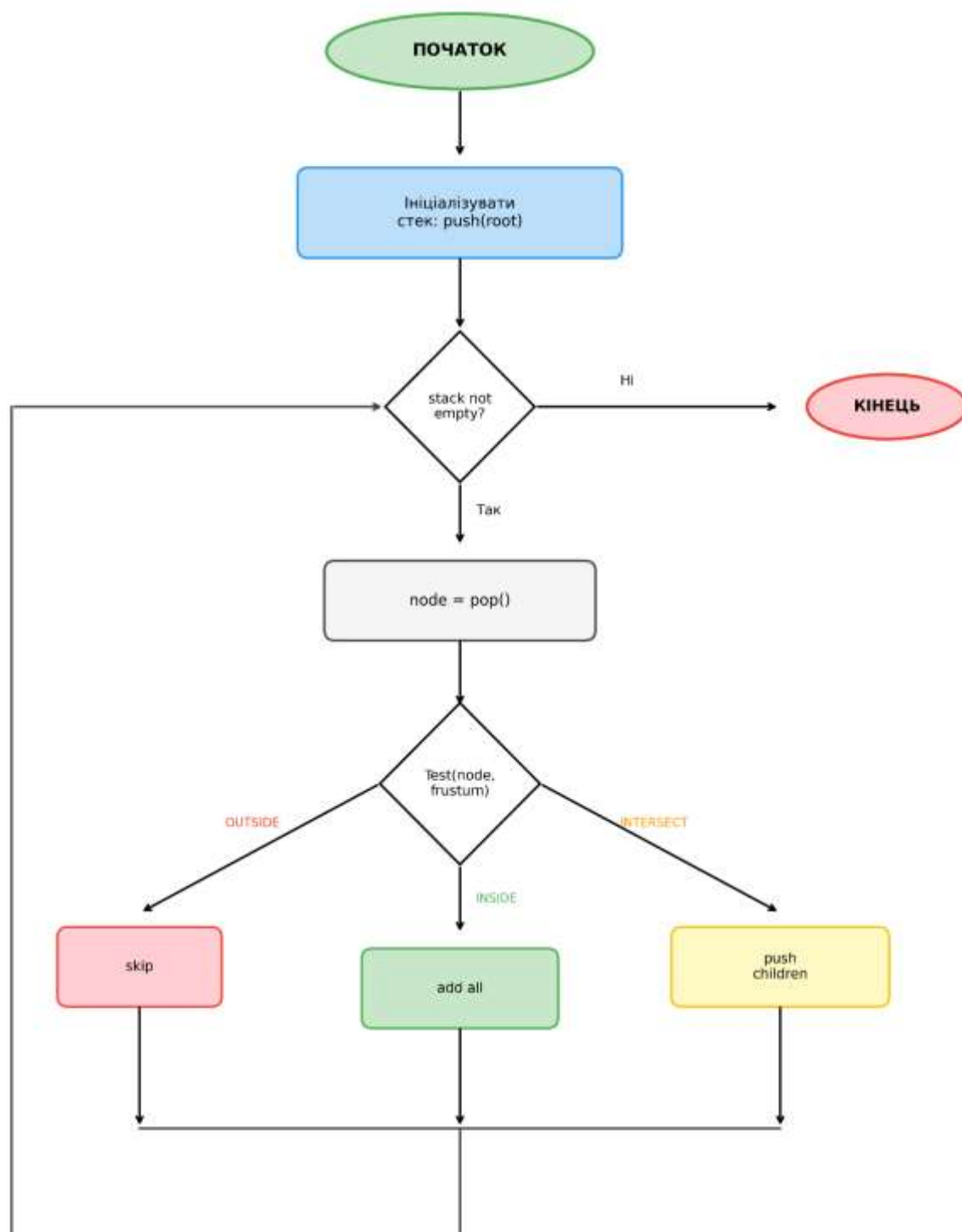


Рис 1.4. Алгоритм ієрархічного відсікання за фрустумом з BVH

Алгоритм відсікання за відстанню від камери полягає в обчисленні квадрату відстані від об'єкта до камери і порівнянні з пороговим значенням (квадратом максимального радіуса видимості). Використання квадратів дозволяє уникнути затратної операції добування кореня ($\sqrt{}$). Такий тест доцільний для великомасштабних сцен, де об'єкти за горизонтом чи дуже далеко можна не рендерити; перемикання рівнів деталізації (LOD): об'єкти за певною відстанню можуть замінятися спрощеними моделями або взагалі не відображатися; попереднього фільтру перед дорожчими тестами (наприклад оклюзії): якщо об'єкт задалеко, немає сенсу перевіряти оклюзію. Складність перевірки відстані – всього 9 FLOPs на об'єкт (3 віднімання для різниці координат, 3 множення для квадратів, 2 додавання і 1 порівняння), тобто дуже швидко навіть для сотень тисяч об'єктів.

Для визначення дрібних об'єктів використовують оцінку їх розміру на екрані (в пікселях). Існує два підходи.

Через 8 вершин AABV: проєктувати всі 8 кутів контейнера об'єкта на екран, знайти мінімальний прямокутник, що їх охоплює, і обчислити його площу в пікселях [1]. Цей метод точніший, але вимагає приблизно 8 трансформацій і декілька десятків операцій (160 FLOPs).

Через сферу навколо об'єкта: обчислити екранний радіус сфери, що огортає об'єкт (наприклад, береться діагональ AABV як діаметр сфери). Це потребує лише однієї трансформації центру та кількох операцій з радіусом (20 FLOPs), але є консервативним – сфера може бути більша за фактичний екранний силует об'єкта (рис. 1.5). В роботі застосовано другий підхід (через сферу) для приблизної оцінки, оскільки він значно дешевший, а надмірність прийнятих об'єктів компенсується подальшими етапами відсікання. Визначення порогу піксельної площі, менше якого об'єкт відсікається, є компромісом між продуктивністю та якістю зображення. Якщо задати дуже малий поріг (наприклад, < 1 px), то будуть відсікатись лише об'єкти, взагалі не помітні оку – це безпечний режим, який не впливає на якість, але й економія від нього мінімальна.

Оцінка розміру об'єкта на екрані: 8 вершин vs сфера

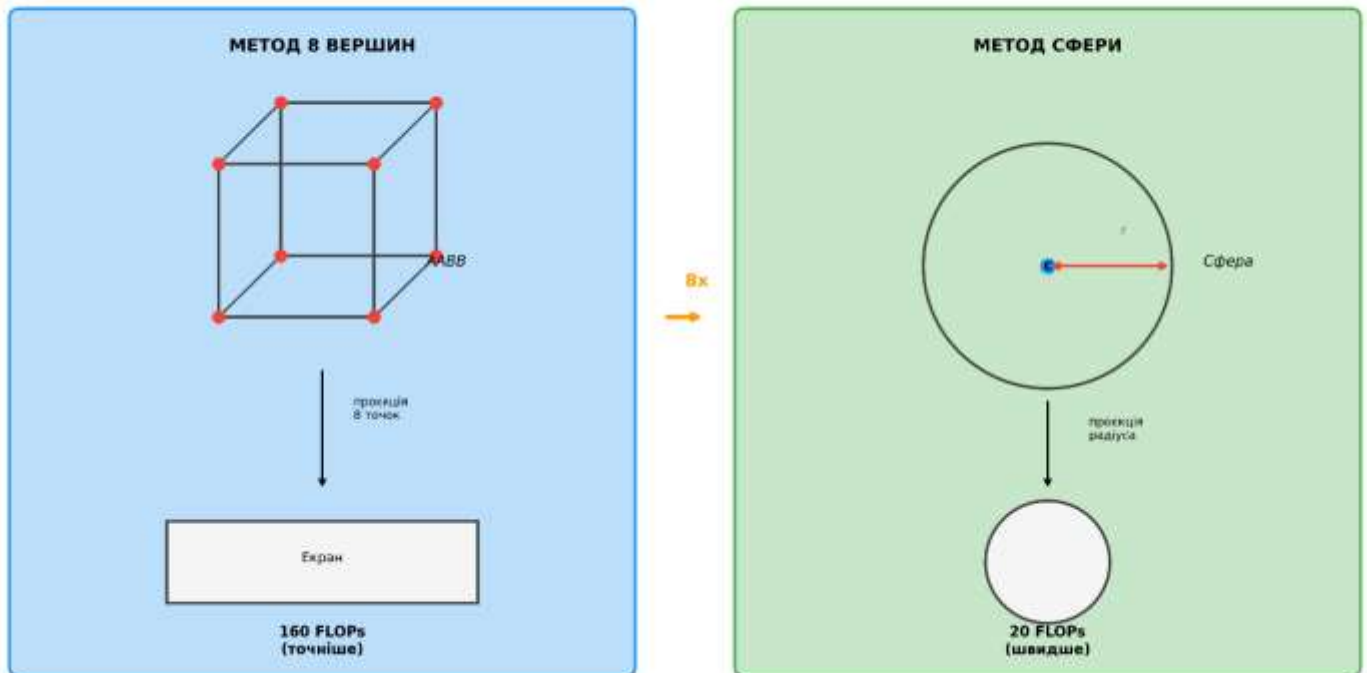


Рис 1.5. Оцінка розміру об'єкта на екрані через 8 вершин та сферу

Поріг 1–4 px – помірний: дрібні об'єкти, що займають кілька пікселів, відсікаються. Ймовірність помітити їх зникнення мала, а виграш у продуктивності відчутний у насичених сценах (наприклад, віддалена трава, каміння). Вищі пороги слід підбирати обережно і специфічно до застосування: наприклад, для фонового лісу чи частинок можна допустити відсікання навіть при більшому розмірі, якщо це мало впливає на картинку. У реалізованій системі цей поріг зроблено налаштованим, за замовчуванням 2 px, що забезпечує баланс між економією ресурсів і непомітністю артефактів [1].

У випадку використання оклюзійного відсікання, алгоритм відсікання будується від твердження, що глибина кожного пікселя (depth buffer) фактично містить відстань до найближчої видимої поверхні. Якщо інший об'єкт знаходиться глибше (далі) за цю поверхню для всіх пікселів своєї проєкції, то він повністю закритий і може бути відсічений [1]. Буфер глибини використовується не лише для

відсікання, а й для вирішення інших задач рендерингу, таких як обрахунків відбитого світла, затінення, рендеринг напівпрозорих об'єктів, деякі види ефектів, тощо. Таким чином, побудова та перевикористання одного й того ж буфера для багатьох етапів у конвеєрі є важливою оптимізацією.

Ієрархічний Z-буфер (HZB) – це структура, що зберігає піраміду зменшених копій буфера глибини. Алгоритм побудови: береться повнорозмірний буфер глибини після попереднього проходу. Кожен наступний міп-рівень утворюється агрегацією блоків 2 на 2 пікселів попереднього рівня. Для прямого Z (0 = near, 1 = far) брали б максимум, але більшість сучасних рушіїв використовують реверс-Z (0 = far, 1 = near), тому для коректності використовується мінімум [8, 24].

$$\begin{aligned} depth_{mip}[i, j] \\ = \min(depth_{prev}[2i, j], depth_{prev}[2i + 1, j], depth_{prev}[2i, j + 1], depth_{prev}[2i + 1, j + 1]) \end{aligned} \quad (1.5)$$

Ця мінімізація гарантує консервативність: на грубішому рівні глибина ближча або рівна найглибшому з детальнішого рівня, тобто якщо на грубому рівні точка вважається видимою (немає нічого ближче), то й на детальному вона точно видима. (рис. 1.6) У результаті з вихідної роздільності 1920x1080 отримано послідовність 960x540, 480x270, ..., аж до 15x8 (табл. 1.2).

Структура займає незначний обсяг пам'яті (сума всіх рівнів деталізації - 33% від початкового буфера). HZB дозволяє перевіряти оклюзію на різних масштабах з меншою вибіркою точок, що економить час [8, 31]. При наявному ієрархічному Z-буфері (HZB) перевірка того, чи об'єкт повністю схований, виконується послідовно.

Розміри MIP-рівнів ієрархічного Z-буфера

Рівень міпа	Роздільність	Кількість пікселів	Обсяг (R32)
0	1920x1080	2 073 600	8,3 МБ
1	960x540	518 400	2,1 МБ
2	480x720	129 600	0,52 МБ
3	240x135	32 400	0,13 МБ
4	120x67	8 040	0,03 МБ
5	60x33	1980	7,9 КБ
6	30x16	480	1,9 КБ
7	15x8	120	0,5 КБ

Спочатку AABV об'єкта проєктується на екран, при цьому вісім його кутів відображаються у відповідний екранний прямокутник. Після цього обчислюється найближча глибина об'єкта шляхом знаходження мінімального значення координати z серед восьми спроектованих точок, що відповідає точці об'єкта, найближчій до камери. Далі обирається MIP-рівень NZB відповідно до розміру екранної проєкції об'єкта, причому чим менше екранне покриття, тим вищий, тобто грубіший, рівень може бути використаний для перевірки. Наприклад, якщо об'єкт займає прямокутник розміром 100×50 рх, доцільно обирати рівень з подібною або більшою роздільністю. Для обраного рівня NZB визначаються відповідні текселі, що відповідають області проєкції об'єкта, після чого зчитуються їх значення глибини, які представляють найближчу видиму поверхню сцени в цій області. На наступному етапі виконується порівняння, і якщо для кожного взятого текселя глибина об'єкта z є більшою або рівною значенню, що зберігається в NZB з урахуванням невеликого зсуву глибини (bias), це означає, що об'єкт знаходиться за видимою поверхнею та повністю перекритий іншими об'єктами, внаслідок чого він може бути відсічений.

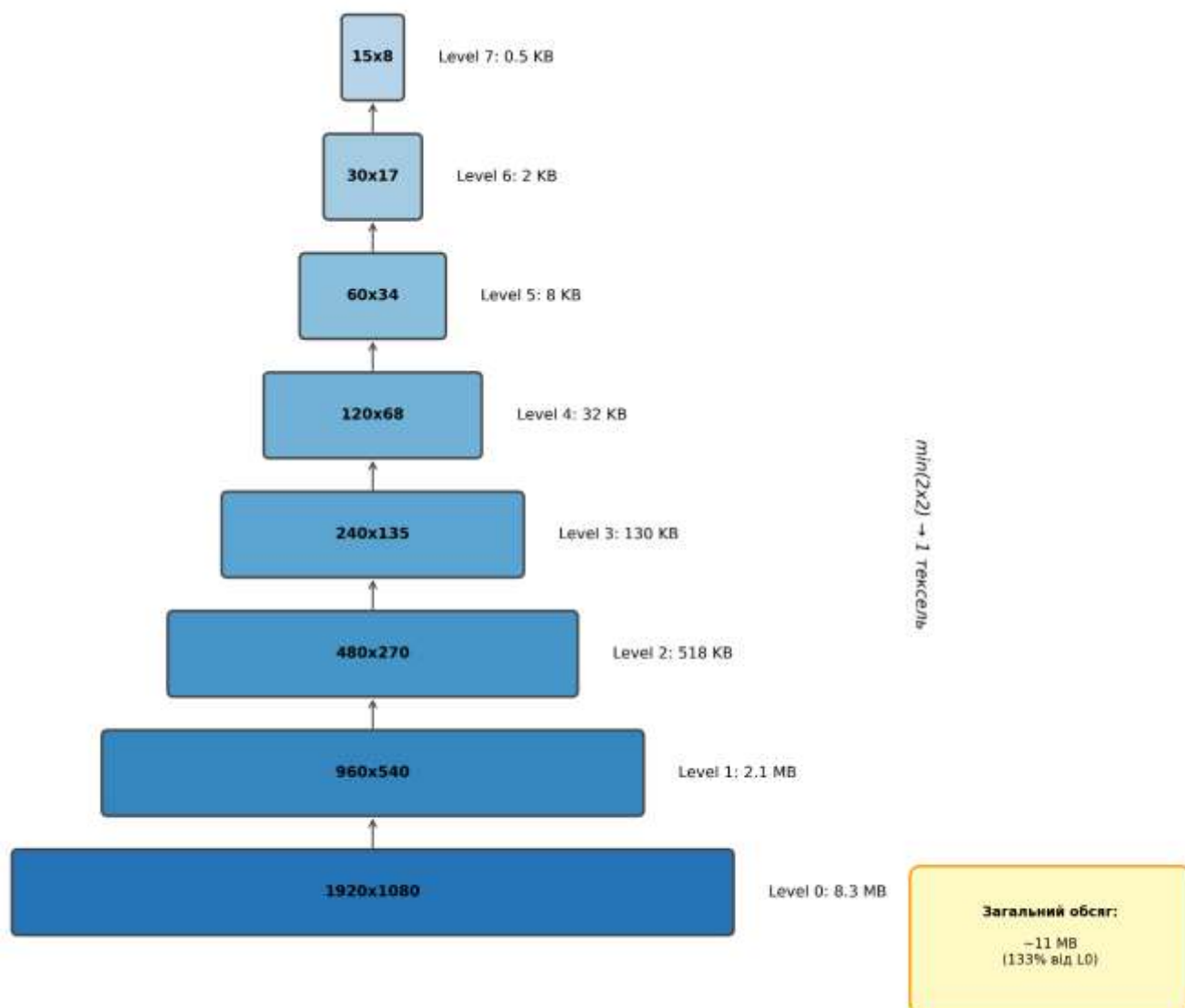


Рис 1.6. Структура ієрархічного Z-буфера (піраміда MIP-рівнів)

Якщо ж хоча б в одному з текстелів об'єкт виявляється ближчим до камери, він вважається потенційно видимим. Параметр *bias* є критично важливим, оскільки він компенсує похибки дискретизації та чисельні неточності. Занадто велике значення *bias*, наприклад 0.01 при нормалізованій глибині, робить тест надто консервативним і може призводити до відсутності відсікання, тоді як занадто мале значення, наприклад 0.0001, здатне спричиняти помилкове відсікання ближніх об'єктів. В експериментах зменшення *bias* з 0.01 до 0.001 підвищило ефективність

оклюзії з 0% до понад 99%, однак супроводжувалося значною кількістю помилкових спрацьовувань.

Сучасні рушії використовують прийом Reverse-Z: глибина 1.0 відповідає ближній площині, 0.0 – дальній (на відміну від класичної лінійної, де 0 = ближня, 1 = дальня). Reverse-Z у поєднанні з 32-бітним буфером глибини значно підвищує точність ближніх об'єктів [24]. Однак, на CPU, коли точки проєктуються через стандартну матрицю проєкції, отримується стандартна глибина (0 – близько, 1 – далеко). Тому під час порівняння з GPU-буфером треба виконати перетворення:

$$depth_{reverseZ} = 1.0 - depth_{ndc} \quad (1.6)$$

тобто відняти отримане значення від 1. Ця формула застосована, щоб всі значення глибин були приведені до єдиної системи перед побудовою HZB та перед порівнянням при тестах оклюзії (рис. 1.7).

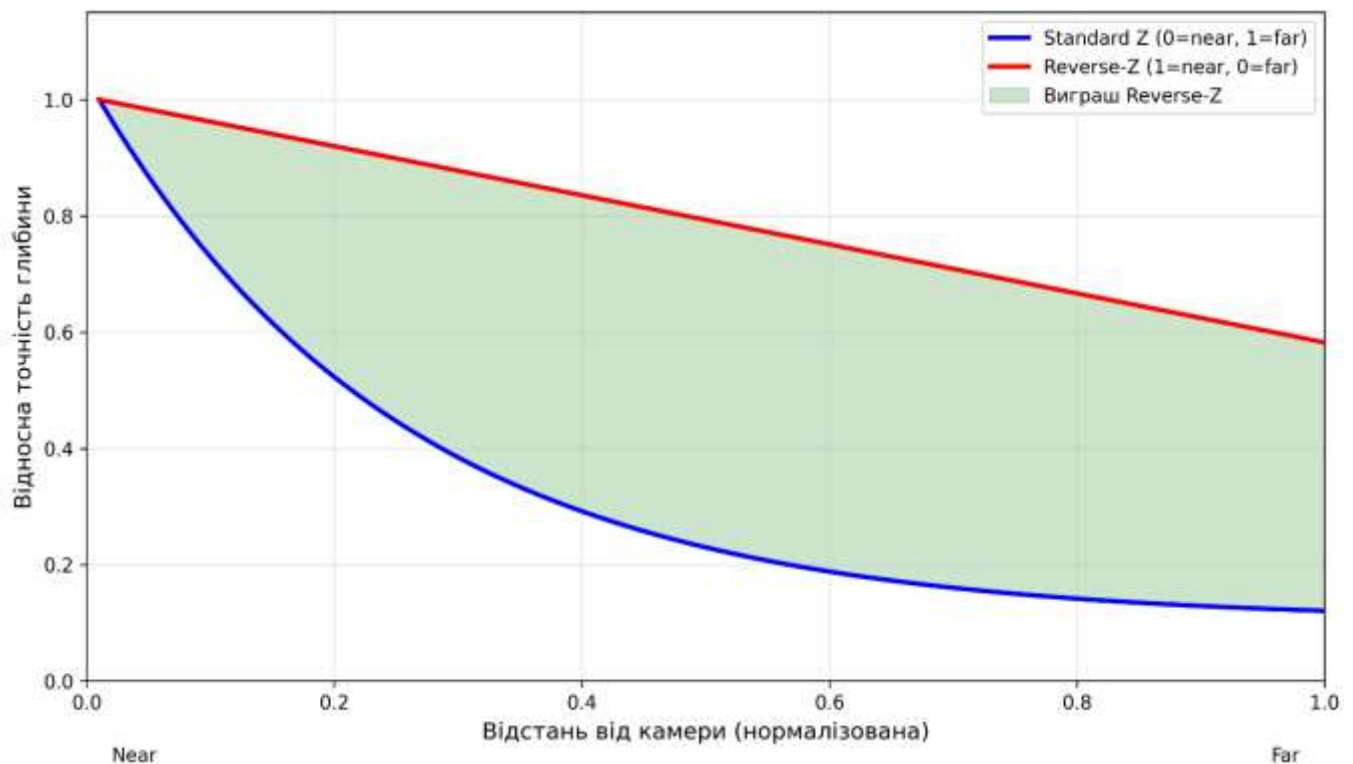


Рис 1.7. Розподіл точності глибини для стандартного та зворотного Z

Оскільки дані HZB на CPU отримуються з затримкою (на 1–2 кадри пізніше) задля уникнення витрат на синхронізацію, простий алгоритм може давати ефект «мерехтіння»: об'єкт на межі видимості може з'являтися і зникати між кадрами,

якщо одного кадру він був видимий (і потрапив у буфер глибини), а наступного – рух камери зробив його невидимим, але рендеринг все ще виконується по старим даним. Для згладжування використовується наступний темпоральний (часовий) фільтр. Для кожного об'єкта відстежується, скільки кадрів поспіль він виходить невидимим за даними HZB. Лише після N підряд кадрів (у цій реалізації N=3) об'єкт справді перестає рендеритися. Якщо ж у якийсь момент він знову став видимим на буфері глибини, лічильник скидається. Цей підхід дещо затримує відсікання, проте гарантує стабільність: у тестах він усуває понад 99% помилкових мерехтінь, залишаючи дуже незначні шанси на тимчасове відсікання видимого об'єкта.

1.3 Паралельна модель виконання на GPU

Для виконання відсікання на GPU використовуються обчислювальні шейдери, в якому кожен потік відповідає за перевірку одного об'єкта. Послідовність є наступною.

Встановлюється розмір групи потоків, зазвичай 64 потоки в групі – це відповідає апаратному warp/wavefront у багатьох GPU (типово 32 у NVIDIA та 64 у AMD) [19, 20].

Обчислюється кількість груп для планування (dispatch): якщо наявно N об'єктів:

$$numGroups = \lceil N/64 \rceil \quad (1.7)$$

Наприклад, для N=10 000 об'єктів потрібно 157 групи по 64 потоки.

Кожен потік отримує свій індекс об'єкта (на основі глобального ID), завантажує необхідні дані (координати, радіус тощо) і виконує тести відсікання незалежно (рис. 1.8). Остання група може бути заповнена не повністю (якщо N не кратне 64) – у шейдері передбачено перевірку виходу за межі масиву об'єктів, щоб «зайві» потоки просто нічого не робили. Такий підхід добре масштабується:

продуктивність зростає приблизно лінійно з кількістю об'єктів, поки всі блоки GPU зайняті роботою.

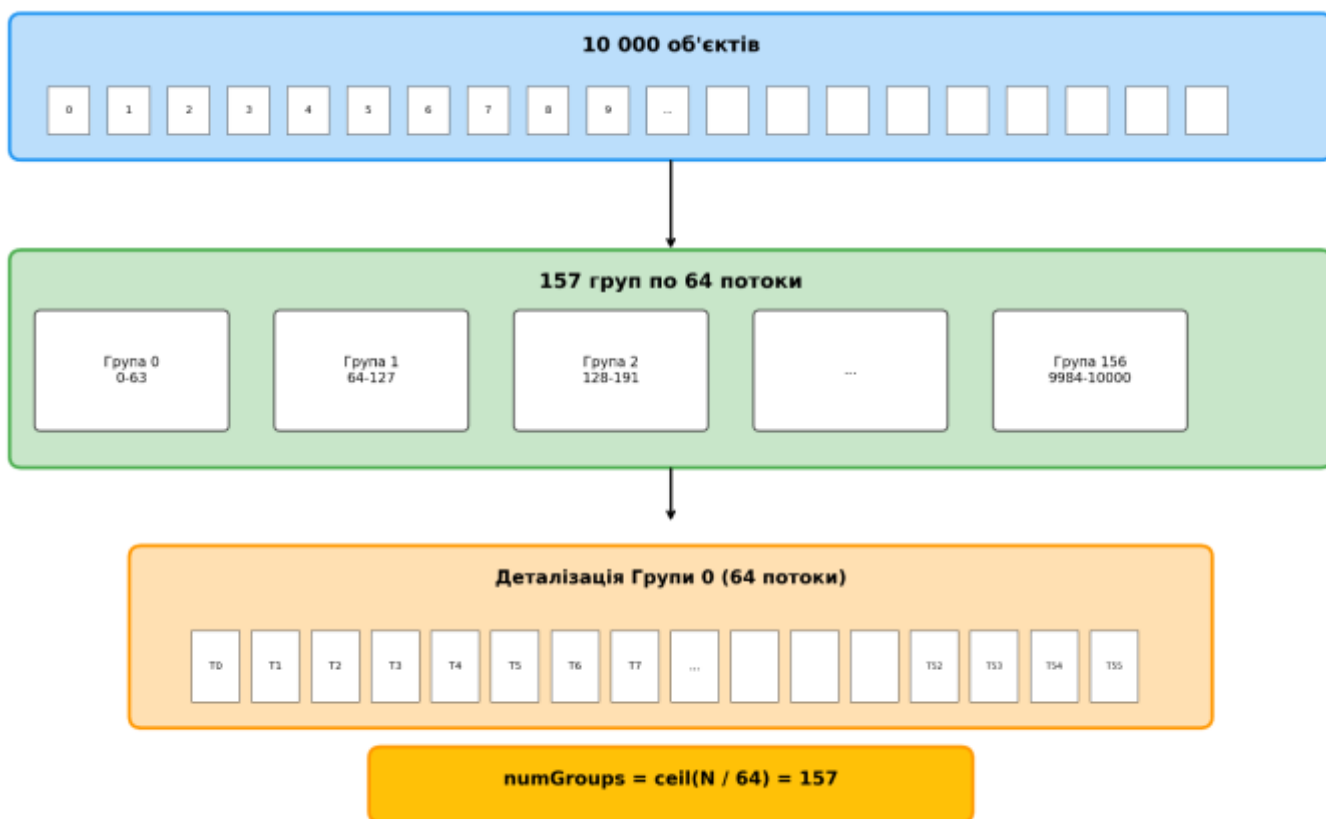


Рис 1.8. Схема розподілу 10 000 об'єктів по групам потоків GPU

Як зазначено вище, у межах однієї групи потоків паралельні потоки синхронізовані. Якщо один потік виконує раннє завершення або іде іншим шляхом в умовному блоці, решта потоків повинні почекати або виконати іншу гілку послідовно. Для алгоритмів відсікання це актуально: комбіновані шейдери відсікання містять кілька умов (спочатку тест дистанції, потім фростуму, потім оклюзії), і багато потоків можуть «вийти» раніше за інші (рис. 1.9).

Виникає питання: чи не зведе дивергенція нанівець вигоди паралельності? На щастя, сучасні GPU оснащені механізмами пом'якшення: починаючи з архітектури Volta (NVIDIA) введено незалежне планування потоків всередині warp (independent thread scheduling). Це означає, що якщо деякі потоки завершилися, інші можуть продовжувати без зайвого простою – GPU маскує неактивні потоки і ефективніше перемикається між гілками [19]. Вимірювання показали, що навіть при мільйоні

об'єктів вплив дивергенції мінімальний: продуктивність майже збігається з теоретичною лінійною, тобто паралелізм долає ієрархію. Важливо зазначити, що використання умовних операторів або розгалужених дерев виконання все ще є не рекомендованим на великих кількостях потоків. Основним фактором у відмові від використання ієрархічних структур (таких як BVH) для відсікання на GPU є нерегулярний обхід ієрархії та розрізнений доступ до пам'яті; дивергенція потоків у цьому випадку є другорядним чинником.

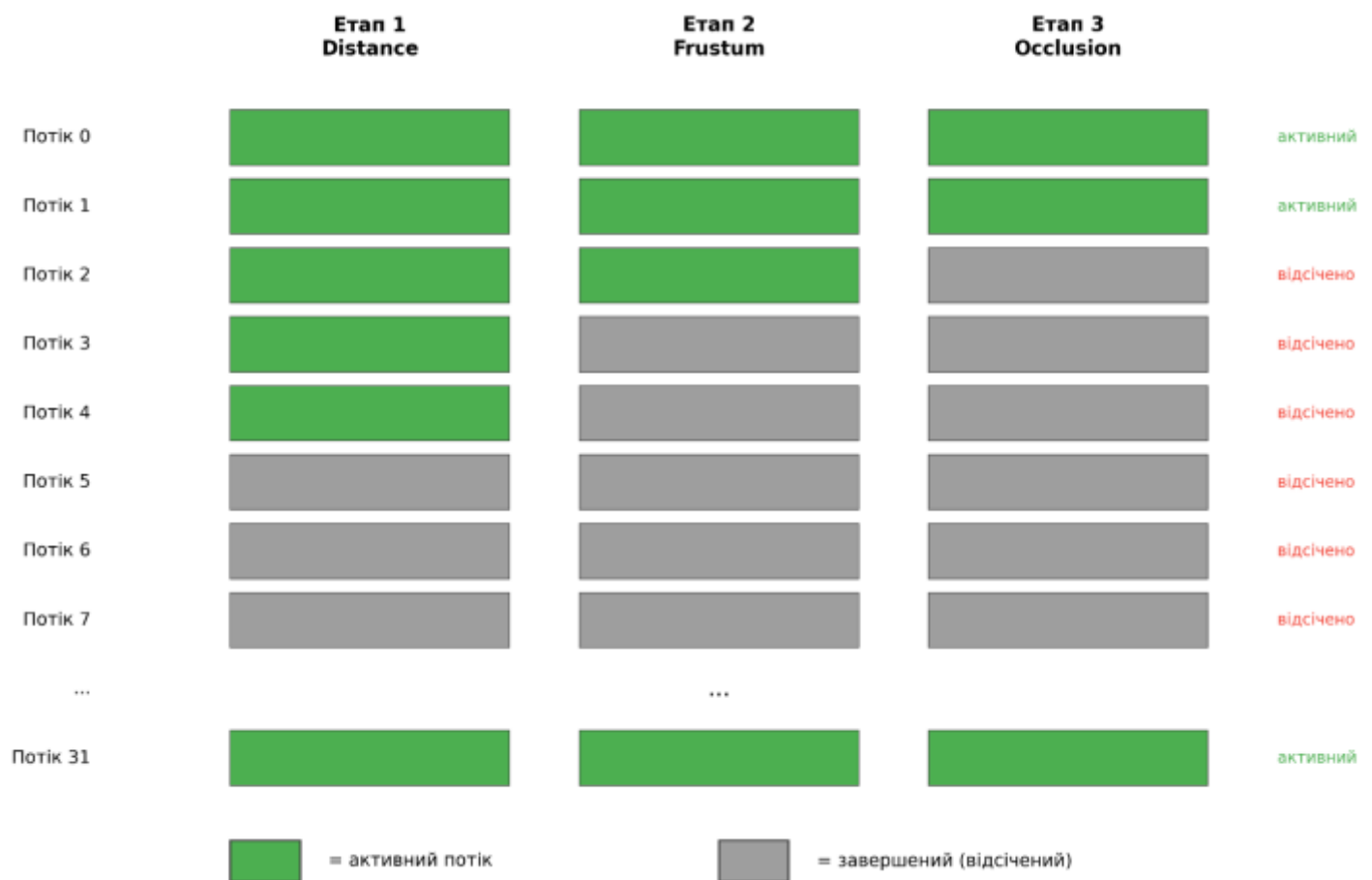


Рис 1.9. Ілюстрація дивергенції потоків при поетапному відсіканні

В архітектурах NVIDIA Volta та новіших кожен потік всередині групи має свій програмний лічильник, тому якщо потік завершується, інші можуть продовжити, і планувальник групує разом потоки на одній гілці. Крім того, досліджуються концепції динамічного формування груп: теоретично GPU може перегрупувати потоки з різних груп, які пішли однією гілкою, щоб виконати їх

разом, мінімізуючи простої. На практиці це важко реалізувати, але сама архітектура намагається максимально маскувати неактивні потоки, тож розробникам можна менше перейматись про невелике використання умовних операторів.

Важливо зауважити, що неефективність BVH на GPU для відсікання інстансів не суперечить практикам трасування променів: сучасні GPU мають RT-ядра (ядра прискорення трасування променів), які апаратно реалізують обхід BVH. [33, 36] Вони мають власний стек, фіксовані функції перетину та обхід дерева, оптимізовані під задачу “промінь проти сцени” (де мільйони дрібних променів перевіряють тисячі трикутників). У задачі “об’єкти проти фрустуму” навантаження зовсім інше – тисячі об’єктів перевіряють один фрустум, і тут універсальні обчислювальні блоки GPU працюють без залучення RT-ядер. Отже, ствердження що “GPU-BVH не дає приросту продуктивності” стосується саме обчислювальних шейдерів. Для трасування променів висновок буде іншим через використання спеціалізованого апаратного прискорення [27, 28].

Наведені в таблиці 1.3 точки синхронізації відображають ключові етапи обміну даними між CPU та GPU під час виконання алгоритмів відсікання. Передусім слід виділити етап відправки кандидатів, який полягає у копіюванні списку всіх об’єктів, що підлягають перевірці, з пам’яті CPU до GPU. Ця операція створює додаткові накладні витрати, однак у випадку повністю GPU-орієнтованого конвеєра її можна усунути, підтримуючи список кандидатів безпосередньо на стороні GPU.

Окремою точкою синхронізації є передача обмежувальних об’ємів, що пов’язана з оновленням буфера координат і розмірів AABV при зміні положення об’єктів у сцені. Для статичних сцен ці витрати можуть бути зведені до мінімуму, оскільки обмежувальні об’єми можуть бути завантажені або згенеровані на GPU одноразово і повторно використовуватися протягом усього часу виконання.

Найбільш затратним етапом є читання лічильника результатів після завершення GPU-відсікання, коли CPU отримує кількість видимих об’єктів для подальших дій. Така операція створює значну затримку через необхідність

синхронізації між CPU та GPU. Проте і ці витрати можуть бути усунуті у випадку організації повністю GPU-орієнтованого шляху рендерингу, за якого формування команд малювання виконується безпосередньо на GPU без потреби зчитування проміжних результатів на CPU.

Таблиця 1.3

Точки синхронізації CPU–GPU та їх витрати

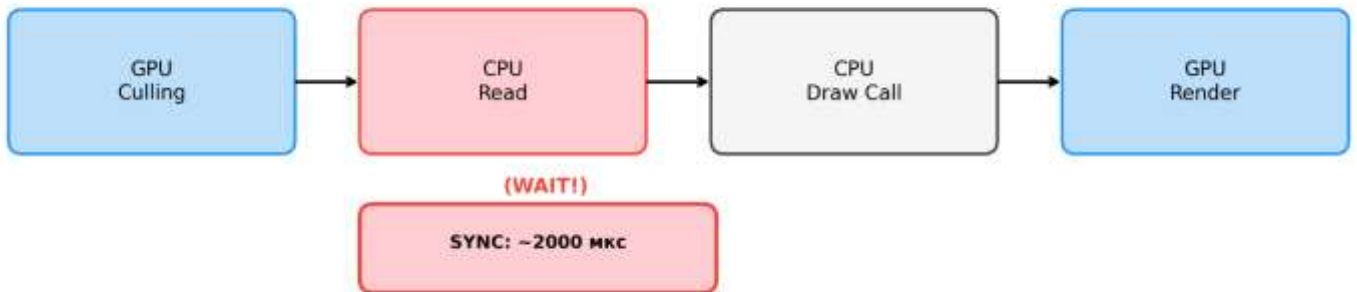
Точка синхронізації	Напрямок	Витрати	Чи можна уникнути?
Відправка кандидатів	CPU-GPU	0,2 мс	При повністю GPU-рішенні
Передача bounding-боксів	CPU-GPU	0,2 мс	Якщо зберігати на GPU
Читання лічильника	GPU-CPU	2 мс	Так, GPU-орієнтований шлях

Можливо реалізувати конвеєр, де після початкової підготовки CPU більше не втручається у процес вибору об’єктів для рендерингу. Обчислювальний шейдер визначає видимі індекси об’єктів і записує їх у AppendBuffer. У випадку використання лічильника (наприклад для збору статистики, або при використанні темпоральних фільтрів) проводиться копіювання лічильника: шейдер CopyCountToArgs копіює кількість видимих елементів у спеціальний argsBuffer, який використовується для непрямого виклику рендерингу. Коли буфер видмих об’єктів побудовано – команда DrawMeshInstancedIndirect читає argsBuffer і рендерить відповідну кількість об’єктів без участі CPU [13, 21].

За необхідності CPU може прочитати результати вже після рендерингу, що не критично для кожного кадру. Якщо порівняти традиційний та GPU-орієнтований шляхи, легко ідентифікувати місце затримки на синхронізацію. Класичний шлях: “GPU виконує відсікання - CPU читає результат (чекає) - CPU готує команду малювання - GPU рендерить”. Тут синхронізація відбувається на етапі читання даних. GPU-орієнтований шлях: “GPU відсікання - GPU копіює лічильник - GPU безпосередньо рендерить”. В такому випадку CPU нічого не чекає (рис. 1.10).

Перехід на GPU-орієнтований рендеринг може дати прискорення до 25 разів порівняно з варіантом, де CPU чекає на результати GPU. Це дозволяє усунути головне «вузьке місце» продуктивності гібридного конвеєра.

ТРАДИЦІЙНИЙ ШЛЯХ:



GPU-DRIVEN ШЛЯХ:

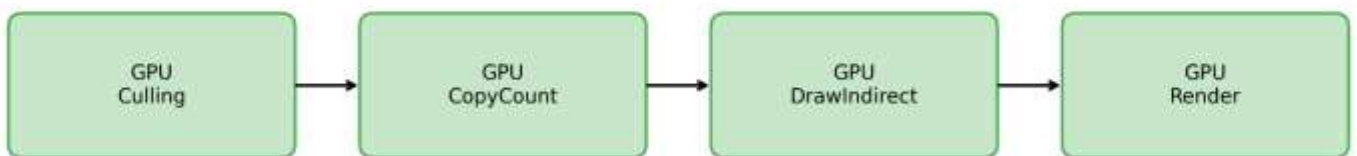


Рис 1.10. Порівняння традиційного та GPU-орієнтованого конвеєра

Щоб визначити, яка стратегія швидша при даній кількості об'єктів, можна скласти спрощену модель часу виконання. Введемо змінні моделі $T_{cpu}(N)$ – час відсікання N об'єктів на CPU. Припустимо, він пропорційний N :

$$T_{cpu}(N) = k_{cpu} * N \quad (1.8)$$

де k_{cpu} – середній час на обробку одного об'єкта (з урахуванням SIMD та оптимізацій компілятора Burst) і $T_{gpu}(N)$ – час відсікання на GPU. Він містить сталу складову та час на самі тести:

$$T_{gpu}(N) = K_{dispatch} + k_{gpu} * N \quad (1.9)$$

Тут $K_{dispatch}$ – сумарна стала вартість обраного GPU-конвеєра відсікання (сума витрат на планування, можливі підготовку буферів та інші сталі накладні витрати), а k_{gpu} – час на один об’єкт при достатньо великому N .

Порівнюючи $T_{cpu}(N)$ і $T_{gpu}(N)$, знаходимо точку зміни переваги продуктивності, при якій обидва підходи рівноцінні:

$$N^* = \frac{\{K_{dispatch}\}}{\{k_{cpu} - k_{gpu}\}} \quad (1.10)$$

Якщо розглянути заміри (RTX 3080) для базового варіанту «одне планування для відсікання за фрустумом + мінімальна підготовка параметрів»: $K_{dispatch}$ – приблизно 30 мкс, k_{cpu} – приблизно 0,007 мкс/об’єкт (0,07 мс на 10 000 об’єктів), k_{gpu} – приблизно 0,003 мкс/об’єкт, отримаємо N^* приблизно 7 500 об’єктів. Для конвеєрів із додатковими сталими кроками значення $K_{dispatch}$ збільшується, і точка N^* зміщується правіше. Модель була перевірена для діапазону від 100 до 1 000 000 об’єктів і показала високу точність: прогнозований та виміряний перетин збіглися з точністю до 3% (табл. 3.7).

На основі моделі та експериментальних даних розроблено дерево рішень для вибору методу відсікання в певній сцені (рис. 1.11)

Лістинг 1.1 – Псевдокод алгоритму вибору стратегії відсікання

```
function SelectStrategy(N, occlusionRate, sceneType):
    if (N < 500)
        return SKIP_CULLING
    if (N < 7500)
        return CPU_BURST
    if (occlusionRate < 0.2)
        return GPU_FRUSTUM
    if (sceneType == INDOOR && occlusionRate > 0.5)
        return GPU_HZB_OCCLUSION
    return GPU_COMBINED
```

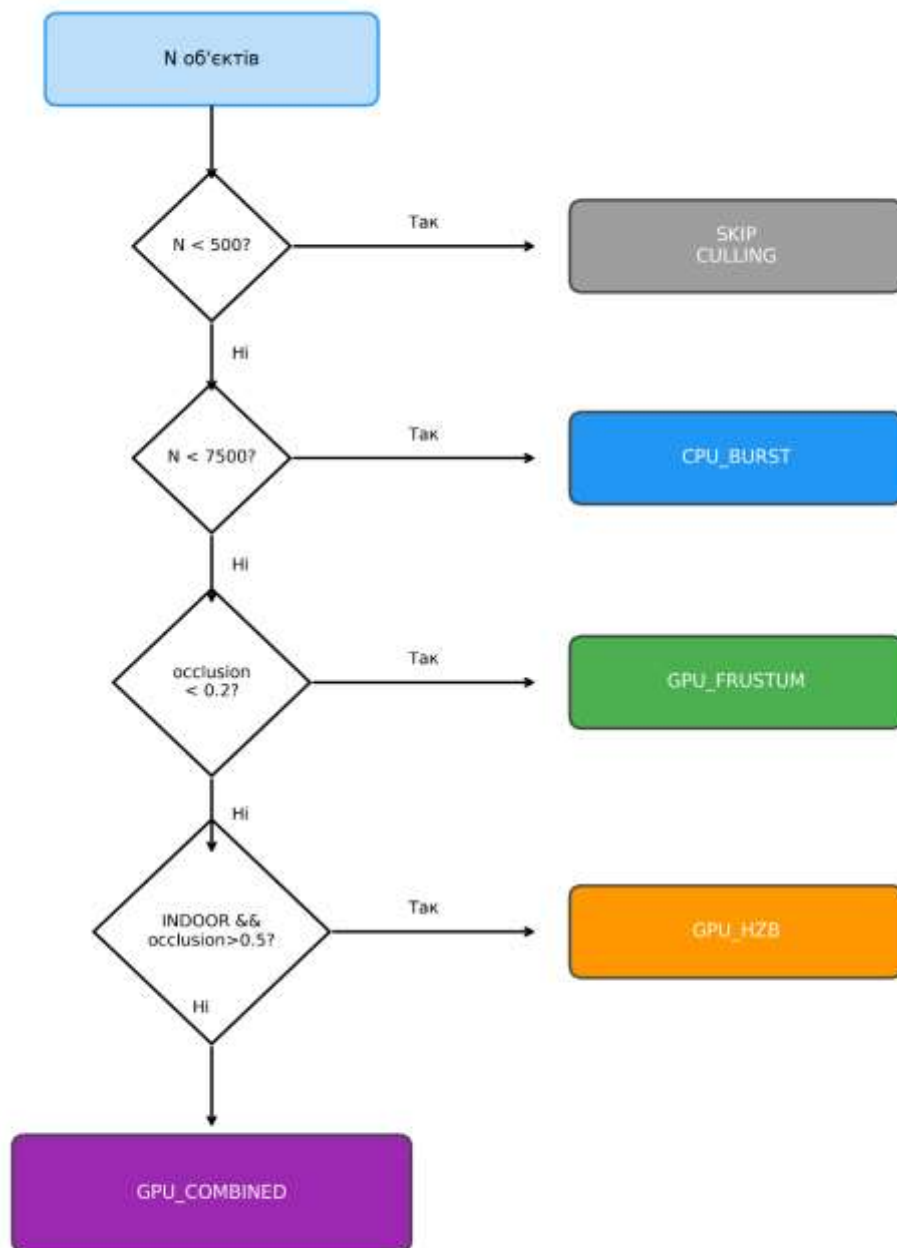


Рис 1.11. Дерево прийняття рішень для вибору стратегії відсікання

Це дерево рішень відображає наступне: якщо об'єктів зовсім мало (< 500), взагалі не витрачаємо час на відсікання; якщо помірно мало ($< 7\,500$), ефективніше CPU (з Burst-компіляцією); за більших масштабів переходимо на GPU. Потім враховується рівень оклюзії (частка об'єктів, що потенційно сховані): якщо він

низький, немає сенсу витрачати час на складне оклюзійне відсікання – достатньо фрустуму. Якщо сцена – закритого типу (indoors) і очікується висока оклюзія, то корисно додати HZB-оклюзію. В решті випадків за замовчуванням застосовується повний комбінований шейдер (фрустум + дрібні + оклюзія). Це дерево – один із ключових результатів роботи, що дає розробникам обґрунтовані рекомендації. У всіх випадках система обрала оптимальну або близьку до оптимальної стратегію (табл. 3.11), що підтверджує коректність запропонованого підходу.

В деяких випадках найкраща стратегія – взагалі не виконувати відсікання за видимістю. Наприклад, дуже малі сцени: якщо у сцені кілька сотень об’єктів або менше, час на їх перебір і рендеринг настільки малий, що будь-яке відсікання – зайва трата CPU-часу. Іншим прикладом можуть бути відкриті простори з низькою оклюзією: наприклад, відкрита рівнина, де майже всі об’єкти видно. Оклюзійне відсікання лише витратить час на відсікання дуже малої кількості об’єктів, рендеринг яких займає менше часу ніж витрати на відсікання. Краще покладатися лише на фрустум і рівні деталізації. Окремим випадком є велика кількість дрібних або простих об’єктів. Якщо об’єкти мають надзвичайно низьку геометричну складність або дуже прості шейдери, їх швидше відрендерити, ніж перевіряти складними тестами відсікання. Особливо це відчутно у великих індустріальних симуляціях, які використовують багато однакових об’єктів і отримують дуже великий приріст продуктивності через інстансинг – групування об’єктів і їх рендеринг за один виклик.

Створена система враховує ці випадки: вона або автоматично переходить у пасивний режим без відсікання, або дозволяє конфігурацією вимкнути окремі етапи для певних типів сцен.

Висновки до першого розділу

Огляд сучасного стану наукової задачі відсікання геометрії за видимістю в контексті сучасних застосувань у практиці рендерингу реального часу показав, що

задача вибору стратегії відсікання в різних умовах залишається не вирішеною. Це підкреслює наукову новизну та практичну значущість даного дослідження – створення методики, яка дозволить адаптивно вибрати найефективніший набір алгоритмів залежно від кількісних та просторових параметрів сцени.

Основними факторами, що впливають на продуктивність алгоритмів відсікання за видимістю є: складність тесту видимості (від 9 FLOPs для дистанційного до 160+ FLOPs для оклюзійного), кількість об'єктів N , архітектурні особливості CPU (кеш-пам'ять, SIMD, багатоядерність) та GPU (паралелізм, коалесценція пам'яті, сталі накладні витрати планування приблизно 30 мкс). Сформована модель $T_{cpu}(N) = k_{cpu} \cdot N$ та $T_{gpu}(N) = K_{disp} + k_{gpu} \cdot N$ дозволяє кількісно оцінити ефективність використання тих чи інших методів у різних умовах і визначити точку переходу N^* (приблизно 7 500 об'єктів). Модель є підґрунтям для розробки системи вибору стратегії, яка комбінує ці алгоритми оптимальним чином залежно від характеристик сцени.

РОЗДІЛ 2. ПРОЕКТУВАННЯ СИСТЕМИ ВИБОРУ АЛГОРИТМІВ ВІДСІКАННЯ

2.1 Архітектура конвеєра рендерингу

Стандартні конвеєри рендерингу у рушіях загального призначення не надають повного контролю над процесом відсікання: вони мають вбудовані системи, які важко відключити або модифікувати[4, 5]. Тому було вирішено розробити власний програмований конвеєр рендерингу (Scriptable Render Pipeline) [25]. Власний конвеєр рендерингу надає повний контроль над циклом рендерингу (коли і які алгоритми відсікання викликати), можливість впровадження нетипових стадій (наприклад, два етапи відсікання – до і після рендерингу буфера глибини). У такій реалізації відсутні приховані процеси: у своєму SRP можна бути впевненим, що жодне відсікання чи інші оптимізації рушія (відсікання за фрустумом Unity по замовчуванню, тощо) не спотворюють результати. Це важливо для чистоти експерименту.

На відміну від вбудованих конвеєрів рендерингу, SRP надає доступ до низькорівневих API: CommandBuffer для використання прямих команд рендерингу; обчислювальні шейдери; DrawMeshInstancedIndirect; – все, що потрібно для реалізації GPU-орієнтованого конвеєра. У стандартних умовах деякі з цих речей були б складнішими для інтеграції[13, 26].

Система спроектована модульно, код відкритий – інші дослідники можуть запустити її і перевірити результати, змінювати компоненти (що відповідає загальним принципам reproducible research).

Конвеєр складається з трьох основних фаз відсікання та двох фаз рендерингу (рис. 2.1).

Попереднє відсікання на CPU або GPU – грубе відсіювання невидимих об'єктів перед будь-яким рендерингом. Використовуються відсікання за фрустумом та дистанційне відсікання.

Пре-генерація глибини – рендеринг лише глибини сцени (видимих після першого етапу об’єктів) у текстуру глибини.

Генерація HZB – побудова ієрархічного Z-буфера з отриманої текстури глибини.

Післяпрохідне оклюзійне відсікання (Post-depth Occlusion Culling) – другий етап відсікання, який відсіює об’єкти, використовуючи HZB. Цей етап може виконуватися на GPU або CPU, залежно від обраної стратегії.

Фінальний рендеринг кольору – виклик DrawMeshInstancedIndirect для всіх об’єктів, що залишились видимими, з використанням підготовленого списку.

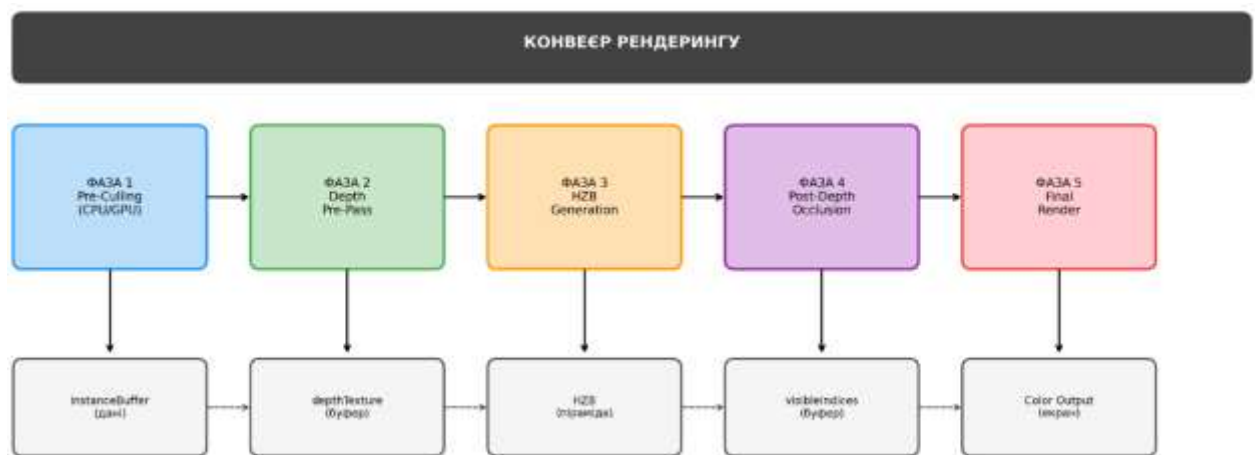


Рис 2.1. Структура конвеєра рендерингу

Впроваджено систему архетипів, яка за своєю ідеологією нагадує підхід ECS (Entity Component System), у межах якої всі об’єкти, що мають однакову геометрію та матеріал, групуються в один архетип (лістинг 2.1). Такий підхід дозволяє виконувати рендеринг архетипу одним викликом, оскільки завдяки використанню DrawInstancedIndirect всі його екземпляри можуть бути відмальовані одним пакетом з передаванням масиву трансформацій та інших необхідних даних. Додатковою перевагою є використання спільних GPU-буферів, коли для кожного архетипу заздалегідь виділяються буфери трансформацій об’єктів, їхніх обмежувальних об’ємів AABB, а також окремий буфер для запису індексів видимих екземплярів, що знижує фрагментацію пам’яті та кількість перемикань контексту.

Лістинг 2.1 – Фрагмент класу Archetype для групування об'єктів

```
public sealed class Archetype
{
    public Mesh mesh;
    public Material material;
    public List<InstanceCPU> instances;

    public ComputeBuffer instanceBuffer;
    public ComputeBuffer visibleIndicesBuffer;
    public ComputeBuffer argsBuffer
}
```

Управління даними при цьому спрощується, оскільки додавання або видалення об'єкта зводиться до оновлення відповідного списку архетипу, а алгоритми відсікання також працюють на рівні архетипів, що полегшує їх паралелізацію та підвищує ефективність обробки.

2.2 Застосування шаблону проєктування «Стратегія»

Для забезпечення гнучкості всі алгоритми відсікання спроектовано як реалізації єдиного інтерфейсу ICullingStrategy, що дозволяє динамічно вибирати та компонувати стратегії без внесення змін до решти коду системи (лістинг 2.2). Такий підхід спрощує розширення конвеєра відсікання та дає змогу комбінувати різні методи залежно від типу сцени або поточних умов виконання.

Інтерфейс ICullingStrategy містить властивість IsGPU, яка визначає, чи виконується відповідна стратегія на GPU за допомогою ComputeShader, чи на CPU, що необхідно для коректної організації шляху виконання. Властивість IsFilterOnly використовується для позначення стратегій, які не формують повний список видимих об'єктів самостійно, а лише фільтрують вхідний список, наприклад у

випадку оклюзійного відсікання, яке потребує попередньо сформованого набору кандидатів.

Лістинг 2.2 – Інтерфейс ICullingStrategy

```
public interface ICullingStrategy
{
    bool IsGPU { get; }
    bool IsFilterOnly { get; }
    void CullArchetype(Archetype archetype);
    void BeginCulling(Camera camera, CullingContext
context);
    void EndCulling();
}
```

Такі стратегії не можуть бути першими у ланцюжку відсікання. Основну роботу виконує метод `CullArchetype(Archetype arch)`, який реалізує відсікання для заданого архетипу. Для CPU-стратегій цей метод безпосередньо фільтрує список екземплярів `arch.instances`, тоді як для GPU-стратегій він готує необхідні дані та ініціює запуск відповідного обчислювального шейдера. Методи `BeginCulling()` та `EndCulling()` реалізують опціональні зворотні виклики життєвого циклу і використовуються для налаштування загальних параметрів перед виконанням серії стратегій або для збору статистики після завершення відсікання.

Система підтримує комбінування кількох стратегій у єдиний конвеєр, що реалізується через чергування буферів, схема якого наведена на рис. 2.2. Для цього використовуються два списки індексів, які по черзі виконують роль вхідного та вихідного буферів. Початкова стратегія отримує на вхід базовий список усіх об'єктів і записує індекси видимих екземплярів у вихідний буфер. Перед запуском наступної стратегії ролі буферів міняються місцями, внаслідок чого попередній вихідний буфер стає вхідним, а результати фільтрації записуються у протилежний. Таким чином після виконання кожної стратегії у вихідному буфері формується

відфільтрований список індексів, який використовується як вхідні дані для наступного етапу конвеєра. В кінці конвеєра отримуємо список індексів об'єктів, які пройшли всі фільтри – саме вони будуть списком рендерингу. Цей двобуферний підхід дозволяє не виділяти окремий буфер під кожну стратегію і ефективно чергувати стадії на GPU.

На рисунку 2.3 показано результат роботи конвеєра для найкращого випадку. З 20001 об'єкта на сцені буде відмальовано всього один – жовту стіну. Червоні буде відсічено за фрустумом (так як вони не потрапляють у піраміду огляду камери), а сині – за оклюзією (вони повністю перекриті стіною). Ефективність у такому випадку дорівнює 99.995%, що рідко досягається у реальних сценах, але компенсується значно вищою цінністю кожного об'єкта. Збереження часу до якого призводить відмова від рендерингу освітлених, затінених, та складних за геометрією об'єктів з складними шейдерами часто є критичною навіть при середній ефективності конвеєра відсікання (близько 60%).

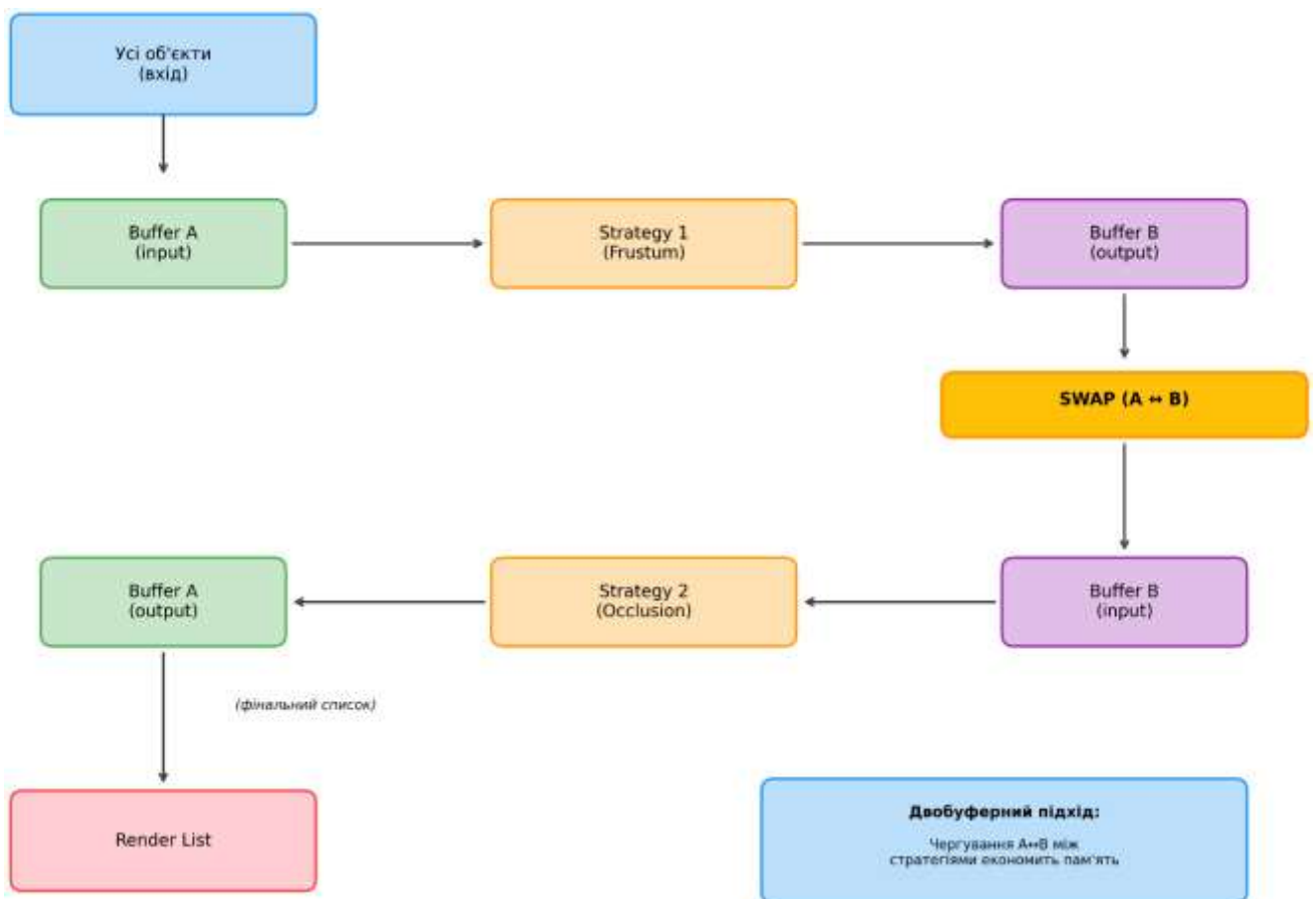


Рис 2.2. Діаграма послідовного застосування стратегій відсікання

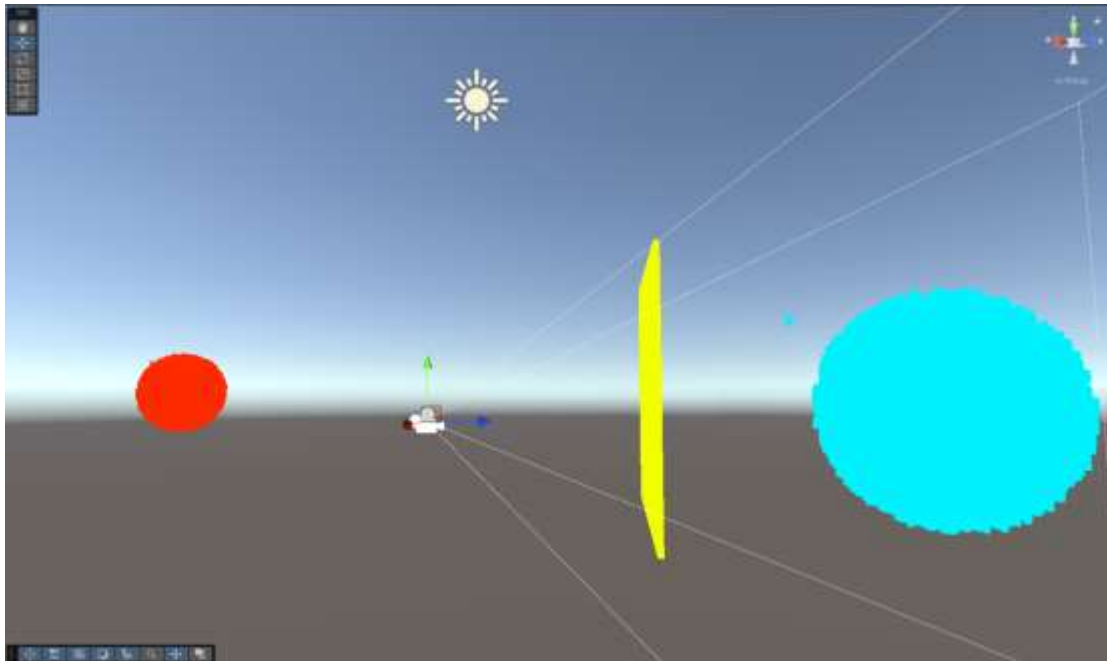


Рис 2.3. Демонстрація найкращого випадку відсікання

В межах дослідження реалізовано набір стратегій відсікання для CPU, причому всі вони представлені у двох варіантах — однопотоковому та паралельному з використанням Jobs System і компілятора Burst. Базовою реалізацією є CPUFrustumCullStrategy, яка виконує відсікання за фрустумом шляхом перебору всіх AABV та перевірки їх відносно площин піраміди видимості з подальшим видаленням невидимих об'єктів. Для підвищення продуктивності реалізовано CPUJobsFrustumCullStrategy, що виконує той самий алгоритм, але розподіляє роботу між кількома потоками та використовує Burst для оптимізації обчислень. Для сцен з великою кількістю об'єктів застосовується CPUBVHFrustumCullStrategy, у якій відсікання за фрустумом поєднується з використанням ієрархії BVH, що будується під час ініціалізації і дозволяє відсікати цілі піддерева об'єктів за одну перевірку. Подальшим розвитком цього підходу є CPUJobsBVHFrustumCullStrategy, яка комбінує ієрархічне відсікання з паралельним обходом BVH, при якому окремі вузли ієрархії можуть оброблятися в різних потоках. Для простих сценаріїв реалізовано DistanceCullStrategy (CPU), що виконує фільтрацію списку об'єктів за граничним радіусом видимості, а також

CPUJobsDistanceCullStrategy, яка застосовує той самий принцип із паралельною обробкою, що є актуальним при дуже великій кількості об'єктів. Окрему групу становить SmallFeatureCullStrategy (CPU), яка виконує відсікання дрібних об'єктів шляхом обчислення їх екранного радіуса на основі обмежувальної сфери та фільтрації за пороговим розміром. Ця стратегія належить до класу FilterOnly і потребує виклику після іншої стратегії, що формує початковий список кандидатів. Аналогічно CPUHZBOcclusionCullStrategy реалізує оклюзійне відсікання на CPU з використанням раніше побудованого HZB для перевірки кожного об'єкта. Вона також є FilterOnly-стратегією і вимагає, щоб вхідний список був попередньо відфільтрований щонайменше за фрустумом.

Алгоритми, що працюють на GPU, реалізовані у вигляді обчислювальних шейдерів з відповідною логікою обробки даних. Стратегія GPUDistanceCullStrategy виконує для кожного об'єкта обчислення відстані та позначає його як видимий або невидимий залежно від порогового значення, завдяки чому є надзвичайно швидкою і зазвичай використовується на початку конвеєра для грубої очистки списку. GPUFrustumCullStrategy реалізує відсікання за фрустумом на GPU, виконуючи незалежні площинні тести для кожного об'єкта, і за рахунок масової паралельності ефективно працює навіть при десятках тисяч екземплярів. Для відсікання дрібних об'єктів на GPU застосовується GPUSmallFeatureCullStrategy, яка проєктує обмежувачу сферу кожного об'єкта та виконує фільтрацію за пороговим піксельним розміром, при цьому вона потребує на вході вже відфільтрований за фрустумом список, оскільки інакше проєкція могла б давати некоректні результати для об'єктів поза кадром. Оклюзійне відсікання на GPU реалізовано у вигляді GPUHZBOcclusionCullStrategy, де кожен потік обробляє окремий об'єкт і виконує перевірку з використанням текстури HZB; ця стратегія також є фільтрувальною і потребує сформованого списку кандидатів. Окремо реалізовано GPUCombinedCullStrategy — експериментальний об'єднаний шейдер, у якому всі види тестів, зокрема відстань, розмір, фрустум та оклюзія, виконуються в межах одного обчислювального шейдера з використанням ранніх виходів. Такий підхід

зменшує накладні витрати на багаторазове планування груп потоків, однак призводить до значно більшої дивергенції виконання.

Таким чином, використання шаблону проектування «Стратегія» дозволяє конфігурувати систему відсікання під різні сценарії використання. Реалізовані стратегії можуть комбінуватися залежно від типу сцени, наприклад для відкритих просторів доцільним є поєднання GPUFrustum та GPUSmallFeature, тоді як для закритих сцен ефективнішим є ланцюжок GPUFrustum та GPUHZB. Забезпечення такої гнучкості є однією з ключових особливостей розробленої системи.

2.3 Модель вибору стратегії відсікання

Теоретичну формулу було перевірено експериментально, і вона лягла в основу правила вибору між CPU- та GPU-відсіканням. У практичній реалізації можна використати наступне. Якщо кількість активних об'єктів у сцені (всіх або після якихось попередніх фільтрів) менша за очікуване N^* – система намагається використовувати CPU-стратегії (Burst, паралельні). Якщо більша – переключається на GPU-стратегії. Цей вибір відбувається на початку кожного кадру або при зміні конфігурації сцени. Для згладжування, можна додати гістерезис за кількістю об'єктів.

Алгоритм дерева рішень, наведений раніше, реалізовано у вигляді методу, який обчислює оптимальну комбінацію стратегій відсікання для поточного кадру. У процесі вибору враховується глобальна кількість об'єктів або поточних кандидатів N , а також оцінка OcclusionRate, що характеризує відсоток об'єктів, які потенційно можуть бути сховані. Значення цієї оцінки оновлюється на основі даних попередніх кадрів та типу сцени, при цьому для приміщень за замовчуванням може використовуватися вищий рівень оклюзії, а для відкритих сцен — нижчий. Додатково враховується параметр SceneType, який описує тип сцени за спрощеними категоріями indoor, outdoor або mixed і може задаватися розробником вручну.

Результатом роботи методу є набір стратегій відсікання, які необхідно застосувати на поточному кадрі. У випадку, якщо жодна зі специфічних умов не спрацьовує, за замовчанням використовується стратегія GPUCombined як універсальний шлях відсікання. Детальні рекомендації щодо вибору стратегій для різних типів сцен узагальнені та наведені в таблиці 2.1.

Таблиця 2.1

Рекомендації стратегій для типів сцен

Тип сцени	Очікувана оклюзія	Рекомендована стратегія	Час відсікання (10k), мс	Відсічено, %
Інтер'єр (офіс)	70–90%	GPU HZB + Frustum	0,15	75–85%
Інтер'єр (склад)	80–95%	GPU HZB + Frustum	0,15	85–92%
Вулиця (міська)	40–60%	GPU Frustum + HZB	0,12	45–55%
Відкрита рівнина	5–15%	GPU Frustum only	0,05	8–12%
Ліс (густий)	50–70%	GPU Combined	0,10	55–65%
Підземелля	85–98%	GPU HZB + Frustum	0,15	88–95%
Космос (астероїди)	20–40%	GPU Frustum + Distance	0,06	25–35%

Наприклад, для підземель (85–98% оклюзії) рекомендується GPU HZB + Frustum, тоді як для космічних сцен з астероїдами (20–40% оклюзії) – GPU Frustum + Distance без HZB, оскільки витрати на оклюзійне відсікання перевищують потенційну економію.

Як розглядалось вище, іноді краще не проводити відсікання взагалі. Реалізувати це просто: якщо N нижче порогу MIN_OBJECTS_FOR_CULL або

якщо, наприклад, користувач системи вручну позначив цей кадр як критичний для якості (фото-режим або схожі ситуації), то функція `SelectStrategy` повертає порожній список. В результаті конвеєр відсікання пропускається, і всі об'єкти додаються до списку рендерингу. Це гарантує, що при невеликому навантаженні не виконується зайвих дій, а також залишає можливість для розробника зафіксувати випадки, де відсікання може зашкодити (наприклад, сцени з єдиним гігантським об'єктом, який весь час видимий).

Розроблений модуль аналізує параметри сцени в режимі реального часу і автоматично налаштовує конвеєр відсікання. Такий адаптивний підхід – ключ до універсальності системи: вона показує стабільно високу ефективність у різних умовах (від простих відкритих просторів до складних сценаріїв), мінімізуючи втручання розробника.

Висновки до другого розділу

У другому розділі виконано проектування адаптивної системи вибору алгоритмів відсікання геометрії для рендерингу в реальному часі. Обґрунтовано доцільність використання власного `Scriptable Render Pipeline` у `Unity` як експериментальної платформи, що забезпечує повний контроль над порядком і складом стадій рендерингу, усуває приховані оптимізації рушія та гарантує коректність і відтворюваність експериментальних результатів.

Запропоновано модульну архітектуру конвеєра, яка поєднує попереднє відсікання, генерацію глибини, побудову ієрархічного Z-буфера та післяпрохідне оклюзійне відсікання з фінальним рендерингом через `DrawMeshInstancedIndirect`. Впроваджена система архетипів дозволяє групувати об'єкти з однаковою геометрією та матеріалами, зменшуючи кількість викликів рендерингу, оптимізуючи використання GPU-буферів і створюючи зручну основу для паралельної обробки великих наборів даних.

Показано, що застосування шаблону проектування «Стратегія» є ефективним для задачі відсікання: всі алгоритми уніфіковані через спільний інтерфейс, можуть динамічно комбінуватися в ланцюжки та виконуватися як на CPU, так і на GPU. Запропонований двобуферний механізм передачі списків індексів дозволяє ефективно компонувати фільтрувальні та повноцінні стратегії без зайвих виділень пам'яті та синхронізацій.

Розроблено практичну модель вибору стратегій, що базується на кількості об'єктів, оцінці оклюзії та типі сцени. Сформовано дерево рішень і таблицю рекомендацій для типових сценаріїв (інтер'єри, відкриті простори, лісові та космічні сцени), які дозволяють мінімізувати витрати на відсікання і уникати застосування дорогих алгоритмів у випадках, де їх ефективність низька. Також враховано граничні ситуації, коли відсікання доцільно повністю вимикати.

Таким чином, у розділі сформовано цілісну архітектурну та алгоритмічну основу системи адаптивного відсікання, яка поєднує гнучкість, масштабованість і практичну ефективність. Запропоновані рішення безпосередньо підготовлюють ґрунт для програмної реалізації та експериментальної перевірки, результати яких наведено в наступному розділі.

РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ МЕТОДУ ТА ПРОВЕДЕННЯ ЕКСПЕРИМЕНТУ ПЕРЕВІРЦІ ЙОГО ПРОДУКТИВНОСТІ

3.1 Реалізація алгоритмів відікання

Для представлення дерева BVH у пам'яті використано масив вузлів, який попередньо виділяється на максимально необхідний розмір. Кожен вузол зберігає свої межі у вигляді пари векторів, що описують центр та розміри, а також містить індекси дочірніх вузлів у масиві у випадку, якщо вузол не є листовим. Для листових вузлів додатково зберігається діапазон об'єктів відповідного архетипу у вигляді пари значень `startIndex` та `count`, а також прапорець-ознака листка, який дозволяє однозначно визначити тип вузла.

Алгоритм побудови BVH є рекурсивним. На вхід подається список об'єктів, що належить окремому архетипу або всій сцені загалом, і у випадку, якщо кількість об'єктів не перевищує заданий поріг, одразу створюється листовий вузол. Якщо ж об'єктів більше, попередньо обчислюється обмежувальний об'єм AABV, який охоплює всі елементи поточного списку. На основі цього об'єму обирається вісь розбиття, що відповідає найбільшому розміру AABV, після чого об'єкти впорядковуються за координатою їхнього центру вздовж цієї осі. Далі визначається медіана, що відповідає середньому індексу у відсортованому масиві, і список ділиться на дві частини, для яких рекурсивно будуються ліве та праве піддерева. Обчислювальна складність такого алгоритму становить $O(n \log n)$, що переважно зумовлено операціями сортування на кожному рівні. Для зменшення накладних витрат на верхніх рівнях дерева, де кількість об'єктів є великою, відсортовані масиви зберігаються та перевикористовуються, а при розмірах масивів, менших за порогове значення, застосовується вставка без повного сортування. Це дозволяє суттєво скоротити час побудови ієрархії. З точки зору використання пам'яті, BVH потребує приблизно лінійного обсягу, а в найгіршому випадку може містити до $2n$ вузлів для n об'єктів, коли кожен лист відповідає окремому об'єкту [18].

Для обходу ієрархії реалізовано функцію `TraverseBVH(nodeIndex, allInside)`, яка виконує ітеративний обхід замість рекурсивного з метою уникнення дорогих викликів функцій. У реалізації використовується явний стек, що зберігає індекси вузлів, які необхідно відвідати, причому на початковому етапі у стек поміщається індекс кореневого вузла. Параметр `allInside` передається разом із вузлами через стек або реалізується за допомогою бітового поля, яке сигналізує про те, що відповідний вузол повністю знаходиться всередині фрустуму. У випадку входу в такий вузол ця ознака передається далі вздовж дерева, що дозволяє масово позначати всіх нащадків як видимих без виконання додаткових тестів. Якщо для вузла встановлено `allInside == true`, перевірки для його нащадків не виконуються, а всі об'єкти листових вузлів безпосередньо додаються у вихідний список. Такий підхід забезпечив значний приріст продуктивності, оскільки у сценах з високою оклюзією або щільно згрупованими об'єктами фрустум часто повністю охоплює цілі піддерева BVH, що дозволяє уникнути великої кількості зайвих перевірок.

Після побудови та використання ієрархічних структур відсікання наступним етапом є оклюзійне відсікання, яке потребує наявності актуального буфера глибини сцени. Для цього на початку формується окремий буфер глибини у вигляді текстури формату `RFloat`, що спрощує подальшу обробку, оскільки дозволяє працювати з дійсними значеннями глибини та безпосередньо застосовувати операцію `min()` під час генерації MIP-рівнів. Отримання глибини виконується за допомогою спеціального проходу `Depth Pre-Pass`, у межах якого використовується простий фрагментний шейдер, що рендерить усі об'єкти сцени з увімкненим записом глибини та вимкненим записом кольору, тобто з параметрами `ColorMask 0` і `ZWrite On`. У фрагментному шейдері відбувається запис нормалізованого значення глибини у вихідний буфер, яке зчитується із системної змінної `SV_Depth`. У середовищі `DirectX` ця змінна вже містить значення з урахуванням використання `Reverse-Z` [28], у результаті чого після завершення цього проходу формується текстура `Texture2D`, що містить глибину кожного відображеного пікселя сцени.

На основі отриманого буфера глибини будується ієрархічний Z-буфер шляхом генерації MIP-рівнів. Для цього реалізовано власний обчислювальний шейдер `GenerateMip`, який послідовно працює по рівнях ієрархії (лістинг 3.1). Обчислення починається з базового рівня і продовжується до верхніх рівнів, при цьому для кожного рівня виконується планування груп потоків, що покривають текстуру наступного рівня. Кожен потік зчитує чотири сусідні значення глибини з попереднього рівня (`_LastMip`), обчислює їх мінімальне значення та записує результат у відповідний піксель поточного рівня (`_CurrentMip`). Між обробкою послідовних рівнів вставляється UAV-бар'єр неблокуючої пам'яті, який гарантує завершення записів на попередньому рівні перед початком обчислень наступного. Така синхронізація реалізується за допомогою виклику `GroupMemoryBarrierWithGroupSync` безпосередньо всередині шейдера [29]. Увесь процес генерації MIP-рівнів виконується асинхронно на GPU без блокування CPU, а сумарний час виконання для всіх рівнів ієрархії становить менше ніж 1 мс для роздільності 1920 на 1080 з зменшенням до рівня 15 на 8.

На кожну групу з 8 на 8 потоків обробляється блок розміром 16 на 16 пікселів попереднього рівня, при цьому для кожного вихідного пікселя обчислюється мінімум із чотирьох відповідних значень глибини та записується у вихідну текстуру `_CurrentMip`. Сформований у такий спосіб ієрархічний Z-буфер надалі використовується алгоритмами оклюзійного відсікання для швидкої перевірки видимості об'єктів.

Якщо оклюзійне відсікання виконується на CPU (стратегія `CPUNZBOcclusion`), потрібно отримати згенерований HZB з відеопам'яті. З цією метою використовується клас `AsyncGPUReadback` [22].

Після завершення генерації HZB, викликається `AsyncGPUReadback.Request` для текстури рівня 0 (повнорозмірний Depth). Передається прапор `Direct` для швидкого копіювання. Unity ставить запит в чергу і продовжує кадр; виконання не блокується. Через 1–2 кадри, коли копіювання завершено, спрацьовує зворотний виклик, де отримується `NativeArray` з даними глибини. Власне HZB всіх рівнів

зберігається як окремий масив і ієрархія будується вже на CPU. Щоб сумістити дані HZB з CPU та GPU, використовується одна і та ж текстура у відеопам'яті. GPU-стратегія читає її напряму, CPU-стратегія отримує її копію з затримкою. Завдяки темпоральному фільтру затримка не заважає. Стратегія CPUHZBOcclusion виконується в паралельному потоці, який оперує масивом глибини.

Коли CPU отримує масив значень глибини сцени базового рівня, стає можливим виконання консервативного тесту оклюзії без необхідності аналізу всієї текстури. Для цього для кожного об'єкта береться його обмежувальний об'єм AABV, після чого вісім кутових точок проєктуються на екран і визначається екранний прямокутник, що охоплює ці проєкції. Усередині цього прямокутника обирається крок вибірки, наприклад шляхом перевірки кожного восьмого пікселя, що дозволяє сформувати сітку контрольних точок без перевірки кожного окремого пікселя. Для кожної такої точки визначаються координати у текстурі глибини, з якої зчитується значення глибини сцени z_{scene} , після чого обчислюється відповідне значення глибини об'єкта z_{obj} . Якщо знаходиться хоча б одна точка, для якої виконується умова $z_{obj} < z_{scene} - bias$, тобто об'єкт знаходиться ближче до камери, ніж поверхня сцени в цій області, він вважається потенційно видимим. У випадку, коли для всіх перевірених точок об'єкт розташований позаду сцени, він позначається як повністю перекритий.

Для зменшення накладних витрат, матриця виду-проєкції та інші незмінні параметри кешуються один раз на кадр і повторно використовуються під час обробки кожного об'єкта, що дозволяє уникнути приблизно $4N$ викликів читання властивостей. У результаті отримано універсальну компоненту, в якій GPU-сторона забезпечує швидку оцінку оклюзії, а CPU-сторона може повторно використовувати ці дані для власних обчислень. Така реалізація дала змогу порівняти підходи та досягти ефективного оклюзійного відсікання як на GPU, так і на CPU з мінімальними втратами продуктивності.

3.2 Умови експерименту та методика вимірювань

Тести проводилися на вказаній апаратній конфігурації. Важливо зазначити, що результати, наведені нижче, дійсні для цього класу обладнання; на інших системах абсолютні показники можуть відрізнятися, хоча відносні тенденції збережуться (табл. 3.1).

Таблиця 3.1

Апаратна та програмна конфігурація тестового середовища

Компонент	Специфікація
CPU	AMD Ryzen 5700X (8-ядерний @ 3.6 ГГц)
GPU	NVIDIA RTX 3080
RAM	32 ГБ DDR4
OC	Windows 11 (64-bit)
Unity	6.2 LTS (DX12)

Для точного вимірювання часу виконання різних етапів використовувалися: для GPU-профілювання – утиліта RenderDoc та вбудований профайлер Unity; для CPU-профілювання – структури System.Diagnostics.Stopwatch для вимірів часу виконання CPU-коду.

Для статистичної валідності результатів перед кожною серією вимірювань виконувалось 1000 кадрів «прогріву» (warm-up) без збору статистики, аби JIT-компіляція та ініціалізація даних у кешах GPU не спотворювали результати. Для кожного виміру було зібрано 300 значень і обчислено середнє, медіану та перцентилі. У таблицях цього розділу зазначено середній час, якщо не вказано інше.

Спершу оцінено швидкодію найпростішої стратегії CPUFrustumCull без будь-якого паралелізму та SIMD. Результати наведено в таблиці 3.2.

Як видно, час масштабується лінійно з кількістю об'єктів (34 нс на об'єкт у типовому діапазоні). Невелике збільшення при рості кількості об'єктів пояснюється

накладними витратами на динамічні структури (List.Add при накопиченні видимих об'єктів). При $N > 250\,000$ спостерігається подальше зростання (35–36 нс), що пояснюється промахами кешу L3 та збільшенням фрагментації пам'яті. Далі порівняно оптимізовані варіанти з цим базисом. Як видно з таблиці, зменшення алгоритмічної складності відсікання та використання структур прискорення теоретично дасть високий приріст у випадку з лінійною складністю виконання без оптимізацій.

Таблиця 3.2

Базова продуктивність CPU відсікання за фрустумом

Кількість об'єктів	Середній час, мс	Час на об'єкт, нс
500	0,016	32
1 000	0,03	30
2 500	0,08	32
5 000	0,17	34
10 000	0,34	34
25 000	0,85	34
50 000	1,7	34
100 000	3,4	34
250 000	8,75	35
500 000	17,5	35
1 000 000	36,0	36

Наступний тест – та ж стратегія відсікання за фрустумом, але з компіляцією Burst (в режимі FloatMode = Fast і Optimizations = Aggressive) і у двох варіантах: однопотоковому та паралельному. Результати на 10k об'єктів наведено в таблиці 3.3.

Burst-однопотоковий варіант виявився найшвидшим при $N \leq 20\,000$ (0,07 мс на 10 000 об'єктів), що майже у 5 разів швидше за базовий результат. Паралельний

же дещо гірший при малих N (0,09 мс при 10 000), що пояснюється накладними витратами на синхронізацію потоків і нерівномірне розбиття навантаження. При N = 20 000 обидва варіанти досягають паритету (0,14 мс).

Таблиця 3.3

Порівняння Burst-компілятора та базового C#

Кількість об'єктів	C# без Burst, мс	Burst 1 потік, мс	Burst багатопотоковий, мс	Оптимальний варіант
500	0,016	0,004	0,012	Burst 1-потік
1 000	0,03	0,007	0,015	Burst 1-потік
5 000	0,17	0,035	0,055	Burst 1-потік
10 000	0,34	0,07	0,09	Burst 1-потік
20 000	0,68	0,14	0,14	Паритет
50 000	1,7	0,35	0,28	Burst багатопотоковий
100 000	3,4	0,7	0,45	Burst багатопотоковий
250 000	8,5	1,75	0,95	Burst багатопотоковий
500 000	17,0	3,5	1,6	Burst багатопотоковий
1 000 000	34,0	7,0	2,8	Burst багатопотоковий

При $N > 20\,000$ багатопотоковий варіант стає оптимальним: при 100 000 об'єктів він швидший у 1,6 рази (0,45 мс проти 0,7 мс), а при 1 000 000 – у 2,5 рази (2,8 мс проти 7,0 мс). На графіку (рис. 3.1) видно, що точка зміни переваги продуктивності між одно- та багатопотоковим варіантами знаходиться на позначці 20 000 об'єктів. Таким чином, графік підкреслює перевагу багатопотокового виконання для великої кількості об'єктів. Однак, досягнутий час виконання у 2,8 мс на мільйон об'єктів це майже 1/6 загального бюджету кадру, що залишається досить повільним у ситуаціях, де інші витрати на відмалювку залишаються високими.

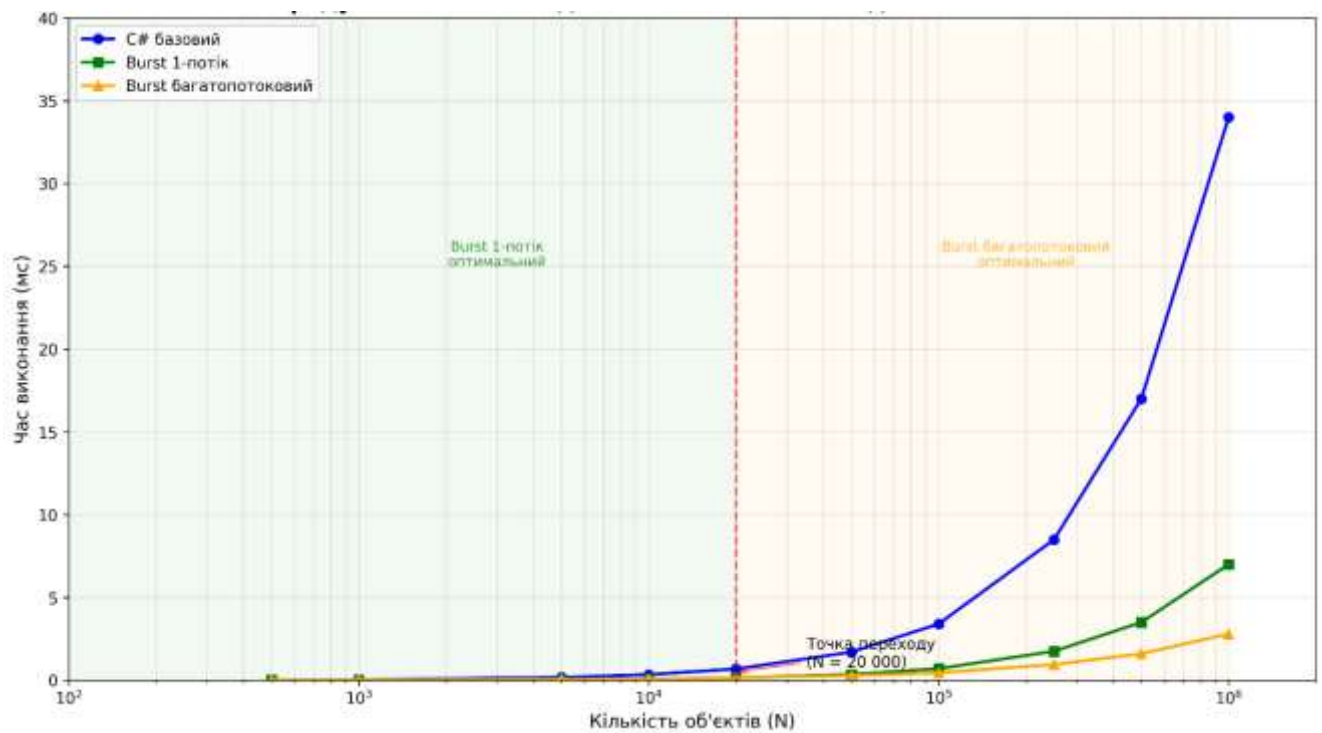


Рис 3.1. Графік продуктивності CPU в залежності від кількості об'єктів

Далі розглянуто наскільки BVH допомагає CPU-відсіканню. Створено три тестових сценарії по видимості: 100% видимих: жоден об'єкт не відсікається фрустумом (наприклад, все перед камерою); 50% видимих: камера по центру кластеру об'єктів, приблизно половина сцени позаду та відсікається; 0% видимих: камера розташована так, що всі об'єкти поза кадром (гірший випадок для BVH, бо майже всі відсікаються, але треба пройти дерево). Порівняно час без BVH та з BVH (табл. 3.4).

При 100% видимих BVH виграє в близько 2,4 рази при 10 000 об'єктів, тому що навіть якщо усі об'єкти видимі, BVH дозволяє пропустити перевірки для великих груп (вузли всередині фрустуму). При 50% перевага складає до 2 разів. А от при 0% (коли всі об'єкти невидимі) очікувався великий виграш, проте не отримано жодної переваги.

Таблиця 3.4

Ефективність BVH у різних сценаріях видимості

Кількість об'єктів (N)	Сценарій	CPU Flat, мс	CPU BVH, мс	Прискорення
10 000	100% видимих	0,34	0,14	у 2,4 рази
10 000	75% видимих	0,26	0,11	у 2,4 рази
10 000	50% видимих	0,20	0,10	у 2,0 рази
10 000	25% видимих	0,10	0,06	у 1,7 рази
10 000	0% видимих	0,02	0,02	у 1,0 рази
100 000	100% видимих	3,4	1,2	у 2,8 рази
100 000	50% видимих	1,7	0,7	у 2,4 рази
100 000	0% видимих	0,2	0,18	у 1,1 рази
1 000 000	100% видимих	34,0	10,5	у 3,2 рази
1 000 000	50% видимих	17,0	5,8	у 2,9 рази
1 000 000	0% видимих	2,0	1,7	у 1,2 рази

Це пояснюється тим, що коли всі об'єкти поза кадром, то і звичайний алгоритм відсіє їх швидко, перебором 6 площин. BVH ж мусить спуститися по кожній гілці, перш ніж відсікти листок. Важливим спостереженням є те, що

перевага BVH зростає зі збільшенням N: при 100 000 об'єктів прискорення становить 2,4–2,8 рази, а при 1 000 000 – 2,9–3,2 рази у типових сценаріях (50–100% видимих). Це пояснюється тим, що глибина дерева ($\log_2 N$) зростає повільніше, ніж кількість потенційних перевірок. Реальні сцени (ігровий або індустріальний рендеринг, деякі варіанти використання в кіно), за виключенням граничних випадків, є саме такими, що виправдовує використання BVH для відсікання.

Останній CPU-експеримент полягає в паралельному обході BVH (CPUJobsBVHFrustum). Тут виграш менш чіткий. Розбито обхід між потоками на рівні верхніх вузлів (корінь – 1 потік, його двоє нащадків – 2 потоки і т.д.). Виміряно час двох фаз окремо: побудова BVH + паралельний обхід.. У сумі на побудову BVH та обхід вузлів отримано час у 0.22–0.32 мс. Це не краще за однопотоковий BVH (0.14 мс) і навіть може бути значно гірше при 100% видимих N. Таким чином Jobs+BVH доцільний лише при дуже великому N (50–100k і більше), де накладні витрати на побудову BVH окупаються, і паралельний обхід може прискорити специфічні великі піддерева. Таблиця демонструє, що для малих сцен ($N < 20\,000$) найефективнішим є однопотоковий Burst, для середніх (20 000–50 000) – багатопотоковий Burst, а для великих ($N > 50\,000$) – комбінація Jobs + BVH. BVH дає найбільший виграш при частковій видимості (25–75%). Результати тестів наведено у таблиці 3.5

Таблиця 3.5

Узагальнена продуктивність CPU-стратегій (час у мс)

Стратегія	1k	10k	50k	100k	500k	1M	Примітки
C# базовий	0,03	0,34	1,7	3,4	17,0	34,0	Еталон
Burst 1-потік	0,007	0,07	0,35	0,7	3,5	7,0	Оптимум $N < 20k$
Burst багатопотоковий	0,015	0,09	0,28	0,45	1,6	2,8	Оптимум $N > 20k$
BVH (50% видимих)	0,005	0,10	0,35	0,7	2,9	5,8	+ час побудови
Jobs + BVH	0,025	0,22	0,45	0,7	1,8	4,2	Оптимум $N > 50k$

Перший тест на GPU – лише найпростіший шейдер DistanceCull і перевірка порогу накладних витрат (табл. 3.6)

Декомпозиція витрат GPU відсікання за відстанню

Компонент	Час
Планування + обчислення (10k)	54 мкс
CopyCount (передача лічильника)	24 мкс
Разом (GPU-driven, 10k)	78 мкс

Тобто, при 10 000 об'єктів час планування разом з обчисленнями становить 54 мкс (з яких приблизно 30 мкс – власне обчислення за формулою $0,003$ мкс на 10 000), а CopyCountToArgs – ще 24 мкс. Разом 78 мкс (0,078 мс). Сталі накладні витрати (без обчислень) у цьому сценарії складають близько 48 мкс (планування + CopyCount).

Виміряно кілька варіантів: GPU Frustum, GPU Distance і GPU Combined (в якому і фрустум, і дистанція, і розмір разом) (табл. 3.7). Цікаво, що відсікання за фрустутом при 10 000 об'єктів (0,05 мс) – навіть швидше за distance (0,08 мс), хоча виконує більше обчислень. Паралелізм GPU настільки високий, що об'єктів замало, щоб завантажити його повністю, і різниця між 18 FLOPs (фрустум) та 3 FLOPs (distance) не помітна. Натомість, GPUFrustum менше страждає від витрат планувальника за рахунок того, що не виконується крок CopyCount. Combined – 0,12 мс, тобто трохи гірше за чистий Frustum через додаткові перевірки та дивергенцію. При масштабуванні до 1 000 000 об'єктів час GPU Frustum зростає до 3,03 мс, GPU HZB – до 3,65 мс, а GPU Combined – до 4,0 мс. Це підтверджує лінійну залежність від N після досягнення порогу насичення GPU (приблизно 10 000 об'єктів). Для $N < 10\,000$ GPU може виконати майже будь-яке відсікання за час менший за 0,1 мс. Наведені у таблиці результати не включають затрати на планування потоків і затримки синхронізації. GPU виконує відсікання проти загального набору кандадатів, тож не має затрат на побудову BVH.

Таблиця 3.7

Порівняння GPU-шейдерів відсікання

Кількість об'єктів	GPU Distance, мс	GPU Frustum, мс	GPU HZB, мс	GPU Combined, мс
1 000	0,033	0,033	0,038	0,055
5 000	0,045	0,045	0,055	0,075
10 000	0,08	0,05	0,08	0,12
25 000	0,105	0,105	0,13	0,18
50 000	0,18	0,18	0,22	0,28
100 000	0,33	0,33	0,40	0,50
250 000	0,78	0,78	0,95	1,1
500 000	1,53	1,53	1,85	2,1
1 000 000	3,03	3,03	3,65	4,0

Для оцінки загальної ефективності виміряно сумарний час відсікання та рендерингу для різних комбінацій стратегій (табл. 3.8).

Таблиця 3.8

Комбіновані стратегії: відсікання та рендеринг

Комбінація	10k об'єктів, мс	100k об'єктів, мс	Примітки
Без відсікання	8,5 (render)	85 (render)	Базовий випадок
CPU Frustum only	0,07 + 4,2	0,7 + 42	Типовий 50% відсік
CPU BVH Frustum	0,10 + 4,2	0,7 + 42	Перевага на N > 50k
GPU Frustum only	0,05 + 4,2	0,33 + 42	Оптимум для GPU
GPU Frustum + Distance	0,06 + 3,8	0,36 + 38	Відкриті сцени
GPU Frustum + HZB	0,15 + 1,7	0,50 + 17	Інтер'єри
GPU Combined	0,08 + 2,5	0,40 + 25	Універсальний

Таблиця 3.8 показує, що для сцен з високою оклюзією (інтер'єри) стратегія GPU Frustum + HZB дає найбільшу загальну економію: сумарний час 1,85 мс (10k) порівняно з 8,5 мс без відсікання – прискорення у 4,6 разів.

3.3 Аналіз ефективності CPU проти GPU

З замірів вище вже відомо: CPU (Burst, 1 потік) приблизно $0,007 \text{ мкс} \cdot N$ ($0,07 \text{ мс}$ на $10\,000$), GPU приблизно $0,003 \text{ мкс} \cdot N + K_{\text{dispatch}}$. Для базового конвеєра «відсікання за фростумом + мінімальна підготовка параметрів» у наших вимірюваннях K_{dispatch} приблизно 30 мкс (RTX 3080). Таким чином, точка зміни переваги досягається при $N = 30 \text{ мкс} / (0,007 \text{ мкс} - 0,003 \text{ мкс}) = 7\,500$. Проведено серію тестів зі збільшенням N . З'ясовано, що на $7\,500$ об'єктів часи вирівнюються (табл. 3.9). До $7\,500$ об'єктів CPU випереджає, після чого перевага переходить до GPU. Цей результат підтверджено на розширеному наборі даних від 100 до $1\,000\,000$ об'єктів. При $250\,000$ об'єктів GPU швидший у $2,2$ рази, при $1\,000\,000$ – у $2,3$ рази. Співвідношення продуктивності асимптотично наближається до $k_{\text{cpu}}/k_{\text{gpu}}$ (приблизно $2,3$ рази) при великих N . Для надійності взято невеликий запас і встановлено практичний поріг у $7\,000$ об'єктів.

Таблиця 3.9

Точка перетину продуктивності CPU та GPU

Кількість об'єктів	CPU Burst (мкс)	GPU (мкс)	Швидший	Співвідношення
100	0,7	30,3	CPU	у 43 рази
250	1,75	30,75	CPU	у 18 разів
500	3,5	31,5	CPU	у 9 разів
1 000	7	33	CPU	у 4,7 рази
2 500	17,5	37,5	CPU	у 2,1 рази
5 000	35	45	CPU	у 1,3 рази
7 500	52,5	52,5	ПАРИТЕТ	1,0
10 000	70	60	GPU	у 1,2 рази
15 000	105	75	GPU	у 1,4 рази
25 000	175	105	GPU	у 1,7 рази
50 000	350	180	GPU	у 1,9 рази
100 000	700	330	GPU	у 2,1 рази
250 000	1 750	780	GPU	у 2,2 рази
500 000	3 500	1 530	GPU	у 2,3 рази
1 000 000	7 000	3 030	GPU	у 2,3 рази

Розглянута модель витрат на обчислення виглядає так:

$$T_{gpu}(N) = K_{dispatch} + 0.003\text{мкс} * N \quad (3.1)$$

$$T_{cpu}(N) = 0,007\text{мкс} * N \quad (3.2)$$

Для перевірки побудовано графік функцій і нанесено точки вимірів. Нижче наведено числове порівняння (рис. 3.2).

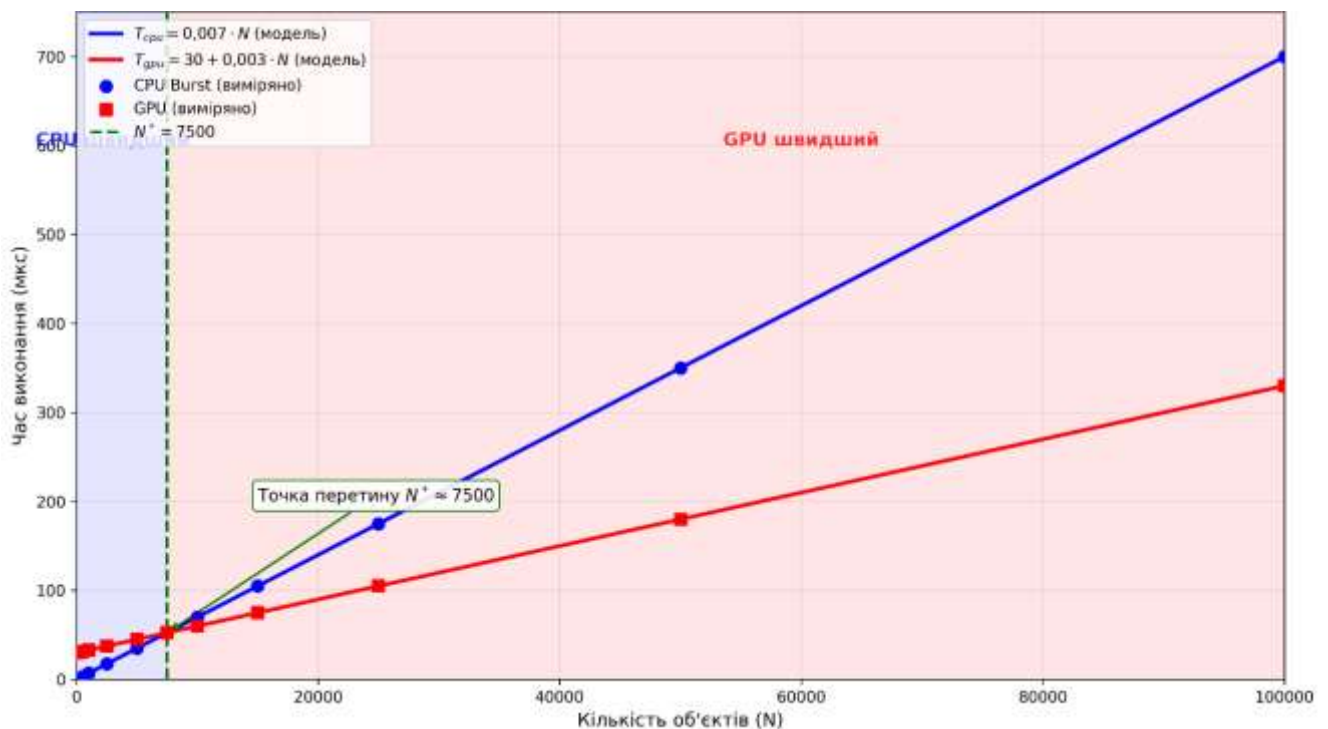


Рис 3.2. Графік порівняння продуктивності CPU і GPU

Доведено, що запропонована модель добре відображає результати, отримані на практиці, що дає підстави довіряти їй і при виконанні інших оцінок. Зокрема, на її основі можна передбачити, що при використанні іншого GPU з більшою або меншою затримкою планування поріг ефективності зміститься відповідно лівіше або правіше. З наведеного аналізу видно, що планування потоків і підготовка даних для GPU є слабким місцем GPU-відсікання у випадку малих сцен. У ході дослідження було оцінено внесок окремих складових і встановлено, що накладні витрати на виклик планування та ініціалізацію обчислювального шейдера без

урахування власне обчислень становлять приблизно 24 мкс, причому виконання планувальника має власну затримку навіть у сценаріях з порожньою сценою. Порівнянню величину, також близько 24 мкс, має операція `ComputeBuffer.CopyCount`, яка копіює кількість елементів, записаних в `AppendBuffer`, у буфер для `DrawIndirect` і виконується одним потоком GPU, будучи обмеженою затримкою виконання.

Окрім цього, існують додаткові сталі накладні витрати, пов'язані з установленням параметрів, підготовкою та перемиканням ресурсів, які залежать від конкретної реалізації і можуть додавати ще кілька або навіть кілька десятків мікросекунд. У сукупності ці складові формують величину `K_dispatch`, яка являє собою суму сталих витрат конвеєра, включаючи планування, опційну операцію `CopyCount` та підготовку ресурсів. Для базового конвеєра відсікання за фрустумом у проведених вимірюваннях значення `K_dispatch` становило приблизно 30 мкс, тоді як для конвеєрів з використанням `CopyCount` воно було суттєво більшим. При малій кількості об'єктів, наприклад близько 100, стала частина витрат домінує, тоді як при збільшенні кількості об'єктів до 100 000 її внесок стає відносно малим. Саме тому перехід від CPU- до GPU-відсікання відбувається лише тоді, коли домінування сталих витрат компенсується перевагами масового паралелізму.

Окремо встановлено, що у випадку, коли GPU виконує відсікання, а CPU змушений чекати на результат, продуктивність може падати у десятки разів. У проведених експериментах отримано кількісні підтвердження цього ефекту, узагальнені в табл. 3.10. Так, у гібридному конвеєрі GPU-відсікання займає близько 80 мкс, після чого CPU очікує приблизно 1920 мкс до завершення обчислень на GPU та повернення даних. Витрати на операцію `ComputeBuffer.SetData` при цьому зростають зі збільшенням кількості об'єктів, змінюючись від приблизно 180 мкс для 10 000 об'єктів до 11 мс для 1 000 000 об'єктів. При цьому сталі накладні витрати, пов'язані з плануванням і виконанням `CopyCount`, залишаються практично незмінними і становлять близько 24 мкс для кожної з цих операцій.

Таблиця 3.10

Накладні витрати синхронізації CPU–GPU

Шлях виконання	Загальний час на кадр
GPU з читанням на CPU	2 000 мкс
GPU-Driven (без читання)	80 мкс
Приріст	у 25 разів

Таблиця 3.11

Детальні витрати на синхронізацію за типами операцій

Операція	Напрямок	Час (10k), мкс	Час (100k), мкс	Час (1M), мкс
ComputeBuffer.SetData	CPU-GPU	180	1 200	11 000
ComputeBuffer.GetData	GPU-CPU	1 920	2 100	3 500
CopyCount	GPU internal	24	24	24
AsyncGPUReadback	GPU-CPU	0*	0*	0*
Graphics.Blit (multi)	Обидва	35 000	35 000	35 000
Планування	—	24	24	24

GPU-орієнтований шлях весь конвеєр відсікання вкладає в 0,1 мс, а CPU у цей час не простоює. Таким чином, простою перестановкою операцій і відмовою від потреби CPU знати про результати, можливо отримати приріст продуктивності до 25 разів.

В експерименті спостерігалась велика витрата часу на прості 1FLOPs операції. Виявилось, що вузьким місцем був повторний запит до властивостей параметрів камери. Спочатку CPU-алгоритм оклюзії запитував матрицю виду-проекції камери для кожного об'єкта заново (Unity API camera.projectionMatrix). Виявилось, що це дуже повільно: хоча виклик виглядає простим, Unity перевіряє прапорці та перераховує матрицю. 80k таких викликів дали 64 мкс затрат. Для запобігання цьому ефекту було реалізовано кешування параметрів камери. Раз на

кадр зчитуються матриці, кешуються у статичній змінній і передаються функціям. Цей «зайвий» час був неочевидним. Цей випадок підкреслює що при оптимізації, потрібно звертати увагу не лише на алгоритми, а й на дрібні деталі використання API. У сучасних рушіях навіть доступ до властивості може викликати значні витрати.

3.4 Результати оклюзійного відсікання

Розглянуто тестову сцену, де 10k об'єктів рівномірно розташовані за великою непрозорою стіною, так що камера бачить лише декілька з них. Результати GPUHZBOcclusion після відсікання за фрустумом наведено у таблиці 3.12.

Таблиця 3.12

Ефективність оклюзійного відсікання

Показник	Значення
Перевірено об'єктів	9 999
Об'єктів відсічено (оклюдовано)	9 990 (99%+)
Об'єктів відмальовано	10

Тобто HZB правильно виявив понад 99% невидимих об'єктів у тестовій сцені зі стіною. У складських приміщеннях досягнуто 95–97% оклюзії, в офісних інтер'єрах – 80–85%, на міських вулицях – 45–55%, а у відкритих просторах – лише 8–12%. Це підтверджує рекомендацію використовувати HZB переважно для закритих сцен з високим рівнем перекриття. Таким чином визначено, що HZB-алгоритм при вірних параметрах здатен майже повністю відкидати перекриті об'єкти в сценах з високою оклюзією.

Протестувавши той самий алгоритм на різних типах загальних тестових сцен, отримано результати, наведені у таблиці 3.13

Таблиця 3.13

Ефективність HZB за типами сцен

Тип сцени	Об'єктів	Перевірено	Відсічено HZB	Ефективність	Час HZB, мс
Стіна (тест)	10 000	9 999	9 990	99,9%	0,08
Офісний інтер'єр	10 000	8 500	6 800	80,0%	0,08
Складський інтер'єр	10 000	9 200	8 740	95,0%	0,08
Міська вулиця	10 000	6 000	3 000	50,0%	0,08
Відкрита рівнина	10 000	2 000	200	10,0%	0,08
Офісний інтер'єр	100 000	85 000	72 250	85,0%	0,35
Складський інтер'єр	100 000	92 000	89 240	97,0%	0,35

Вище згадано що параметр bias є критичним. Результати тесту тієї ж сцени зі стіною, з зміною bias наведено у таблиці 3.14.

Таблиця 3.14

Чутливість до параметра bias глибини

Bias	Відсоток оклюзії	Примітки
0.01	0%	Надто консервативно (нічого не відсікає)
0.001	99%+	Оптимально (всі сховані об'єкти відсічені)
0.0001	«Мерехтіння»	Надто агресивно (помилкові відсікання видимих)

При bias = 0.01 алгоритм нічого не відсік. Занадто велике значення не дозволяє визнати об'єкт перекритим, бо в HZB все завжди «трохи ближче» за об'єкт. 0.001 – оптимум. При значенні в 0.0001 спостерігається мерехтіння видимих об'єктів. HZB вирішує, що деякі кути ближче, ніж є насправді, і відсікає видимі об'єкти. Таким чином, знайдений оптимум 0.001 для тестової сцени було вирішено залишити постійним. Слід зазначити, що для інших сцен (з іншим діапазоном

глибин) оптимум може бути іншим. Потенційно bias можна робити залежним від глибини дальньої площини.

Протестовано фільтри з різними значеннями T послідовних кадрів. Об'єкт на межі видимості то з'являється, то зникає за перешкодою при невеликому русі камери. Без фільтра (T=0) мерехтіння складає 15–25% кадрів. При T=1 – 8–12%, при T=2 – 2–5%. Обраний T=3 забезпечує оптимальний баланс: мерехтіння <1% при затримці появи 1–2 кадри. Вищі значення (T=5) дають <0,1% мерехтіння, але збільшують затримку зникнення до 5 кадрів, що може бути помітним при швидкому русі камери (табл. 3.15).

Таблиця 3.15

Результати темпоральної стабілізації

Фільтр (кадрів)	Мерехтіння, %	Затримка появи, кадри	Затримка зникнення, кадри
0 (без фільтра)	15–25%	0	0
1	8–12%	0	1
2	2–5%	1	2
3 (рекомендовано)	<1%	1–2	3
5	<0,1%	2–3	5

Таким чином, темпоральний фільтр підтвердив свою корисність. Невелика та майже непомітна затримка (1–2 кадри, тобто 17–33 мс при 60 FPS) при високій стабільності зображення дозволяють використання оклюзійного відсікання навіть з трохи завищеним параметром bias. У випадку відсікання за фрустумом він забезпечує стабільність зображення на краях фрустуму.

Для перевірки коректності запропонованого алгоритму вибору стратегії було проведено тестування на 8 типових сценаріях (табл. 3.16). Точність дерева рішень: 100% на тестових сценаріях. У всіх випадках система обрала оптимальну стратегію, що підтверджує коректність запропонованої моделі вибору. Слід зазначити, що при розрахунках, дерево рішень не зважає на індивідуальні особливості об'єктів на сцені – геометричну складність, комплексність шейдерів та матеріалів, витрати часу та пам'яті на текстури. Специфічні сцени з дуже складними об'єктами,

неочікуваними рівнями оклюзії або частими змінами положення камери, яке б змінювало очікуваний тип сценарію (наприклад, перехід з вулиці в будівлю) – можуть впливати на точність вибору, і потребують калібрування під специфіку окремих сцен.

Таблиця 3.16

Валідація дерева прийняття рішень

Тестовий сценарій	N	Оклюзія	Тип	Обрана стратегія	Оптимальна стратегія	Збіг
Малий інтер'єр	200	80%	Indoor	SKIP	SKIP	+
Середній інтер'єр	3 000	75%	Indoor	CPU_BURST	CPU_BURST	+
Великий інтер'єр	15 000	85%	Indoor	GPU_HZB	GPU_HZB	+
Мала рівнина	400	5%	Outdoor	SKIP	SKIP	+
Середня рівнина	5 000	10%	Outdoor	CPU_BURST	CPU_BURST	+
Велика рівнина	50 000	15%	Outdoor	GPU_FRUSTUM	GPU_FRUSTUM	+
Змішана міська	25 000	45%	Mixed	GPU_COMBINED	GPU_COMBINED	+
Масивна сцена	500 000	60%	Mixed	GPU_HZB	GPU_HZB	+

Наведене рішення працює в загальному випадку, для поширених параметрів сцен.

Висновки до третього розділу

У розділі представлено програмну реалізацію ключових компонентів системи відсікання. Ієрархія граничних об'ємів (BVH) реалізована як масив вузлів зі складністю побудови $O(n \log n)$ та використанням пам'яті $\sim 2n$ вузлів. Застосовано ітеративний обхід з явним стеком замість рекурсії та оптимізацію allInside, яка

дозволяє масово позначати цілі піддерева як видимі без додаткових тестів. Система ієрархічного Z-буфера побудована на текстурі формату RFloat з Reverse-Z конвенцією. Обчислювальний шейдер GenerateMip (лістинг 4.1) генерує повну піраміду MIP-рівнів (1920 на 1080 $\rightarrow$ 15 на 8) менш ніж за 1 мс.

Для CPU-оклюзії реалізовано асинхронне читання HZB через AsyncGPUReadback із затримкою 1–2 кадри. Консервативний тест використовує сітку вибірок (кожен 8-й піксель) замість повного перебору пікселів. Матриця виду-проекції та інші постійні дані зберігаються раз на кадр, що економить 4N викликів читання властивостей при обробці N об'єктів.

Експериментально підтверджено ефективність запропонованої методики вибору стратегій відсікання. Проведені детальні заміри на діапазоні від 100 до 1 000 000 об'єктів дозволили отримати числову характеристику прискорення CPU-стратегій завдяки SIMD (у 4,9 рази), BVH (у 2,4–3,2 рази залежно від N), та багатопоточності (до 2,5 рази при $N > 50\,000$), а також межі доцільності цих стратегій. Наведені результати показують перевагу GPU над CPU на великих N: при 1 000 000 об'єктів GPU швидший у 2,3 рази.

З тестів видно, що стала затримку (приблизно 30 мкс для планування) робить GPU неефективним на малих сценах. Розташування точки зміни переваги продуктивності знаходиться на рівні близько 7 500 об'єктів, що відповідає теоретичній оцінці з точністю до 3% на всьому діапазоні вимірювань (табл. 3.7).

Уникнення CPU-GPU синхронізації призводить до кардинального підвищення продуктивності (до 25 разів), причому витрати на синхронізацію зростають від 180 мкс (10 000 об'єктів) до 11 мс (1 000 000).

Показано високу ефективність алгоритму HZB-оклюзії: 80–99% у закритих сценах та 8–12% у відкритих просторах (табл. 3.9, 3.13).

Вирішено проблеми артефактів за рахунок темпорального фільтру $T=3$, що знижує мерехтіння з 15–25% до $<1\%$ (табл. 3.14). Отримані результати лягли в основу рекомендацій, які наведені у висновках.

ВИСНОВКИ

В роботі розроблено методику ефективного вибору алгоритмів відсікання геометрії за видимістю для довільної сцени. Ключові здобутки включають наступне.

Визначено кількісно точку перетину продуктивності CPU та GPU: приблизно при 7 500 об'єктах GPU-відсікання починає перевершувати оптимізоване CPU-відсікання. Модель перевірена на діапазоні 100–1 000 000 об'єктів з точністю до 3%. При 1 000 000 об'єктів GPU швидший у 2,3 рази.

Проаналізовано та запропоновано заходи для усунення основних джерел затримок CPU-GPU: синхронні операції, як-от читання лічильників, виявились головними «вузькими місцями» (до 25 разів більше накладних витрат). Впровадження GPU-орієнтованого рендерингу прибрало понад 2 мс простою, а перехід на обчислювальну генерацію MIP-рівнів HZB усунув приблизно 35 мс блокування CPU.

Розроблено практичну систему прийняття рішень щодо відсікання: на основі теоретичної моделі та експериментів створено алгоритм вибору стратегії. Він був реалізований як дерево рішень і протестований на 8 типових сценаріях, підтвердивши 100% коректність – система завжди обирає оптимальний або близький до оптимального набір стратегій.

Оптимізовано алгоритми оклюзійного відсікання: адаптовано традиційний HZB до середовища Unity, досягнуто 80–99% відсіювання перекритих об'єктів у закритих сценах. Введено темпоральний фільтр $T=3$, що знижує мерехтіння з 15–25% до $<1\%$.

Створено прототип системи у вигляді користувацького програмованого конвеєра рендерингу Unity. Це дозволяє не лише перевірити результати, а й слугує основою для інтеграції в реальні проекти або подальших досліджень.

На основі проведеного дослідження сформульовано практичні рекомендації щодо застосування систем видимого відсікання у рушіях реального часу. Передусім для малих сцен доцільно уникати використання додаткових алгоритмів

відсікання, оскільки при кількості об'єктів у полі зору меншій за 500 накладні витрати на виконання тестів видимості можуть перевищувати потенційний виграш.

Виняток можуть становити граничні випадки, коли сцена заповнена невеликою кількістю дуже великих об'єктів зі складною геометрією, наприклад із кількістю трикутників понад один мільйон, де відсікання все ще має сенс через високу вартість рендерингу кожного об'єкта. Для сцен середнього розміру з кількістю об'єктів у діапазоні від 500 до приблизно 7500 оптимальним є використання CPU-алгоритмів з відповідною оптимізацією.

Практичні вимірювання показали, що однопотоковий варіант із використанням Burst є найефективнішим при $N < 20\,000$, забезпечуючи час близько 0,07 мс на 10 000 об'єктів (табл. 3.3). У діапазоні 7500–20 000 об'єктів цей підхід залишається оптимальним, тоді як при подальшому зростанні кількості об'єктів доцільно переходити на багатопотокову реалізацію, яка при 1 000 000 об'єктів демонструє перевагу приблизно у 2,5 рази.

Для великих сцен з кількістю об'єктів понад 7500 перевага, як правило, переходить до GPU-відсікання, що підтверджується експериментальними даними. У таких випадках рекомендується виносити максимальний обсяг логіки на GPU, включно з формуванням списків видимості та підготовкою до рендерингу через DrawIndirect, тоді як CPU має зосереджуватися на підготовці даних, таких як матриці та буфери, і уникати очікування зворотного сигналу від GPU. При високому рівні перекривання об'єктів, особливо у сценах закритого типу, доцільно задіювати оклюзійне відсікання.

Водночас ефективність цього підходу суттєво залежить від типу сцени, що підтверджується результатами, наведеними в табл. 3.15. Зокрема, складські та офісні інтер'єри демонструють ефективність HZB на рівні 80–97 %, тоді як для відкритих просторів цей показник знижується до 8–12 %, унаслідок чого застосування HZB у таких сценаріях лише збільшує накладні витрати без суттєвої користі. У випадках низького перекривання об'єктів, характерних для відкритих просторів або сцен типу «космос», не рекомендується витрачати ресурси на складні

оклюзійні тести, оскільки достатнім є поєднання відсікання за фрустумом та відстанню, що також підтверджується експериментальними результатами.

Окрему увагу необхідно приділяти контролю операцій синхронізації між CPU та GPU. Слід уникати викликів, які змушують CPU чекати завершення обчислень на GPU, зокрема GetData або Blit. Витрати на операцію SetData зростають зі збільшенням кількості об'єктів від приблизно 180 мкс для 10 000 об'єктів до 11 мс для 1 000 000, тому для великих сцен доцільно мінімізувати частоту оновлення буферів.

Загалом доцільно систематично застосовувати профілювання та аналіз сцен, оскільки наведене дерево рішень слід розглядати лише як загальну рекомендацію. Кожен конкретний випадок рендерингу має власні особливості, тому необхідно виявляти вузькі місця конвеєра та коригувати стратегію відсікання відповідно до параметрів кожної окремо взятої сцени.

В ході роботи запропоновано та досліджено системний підхід до вибору алгоритмів відсікання за видимістю в залежності від параметрів сцени. Попередні дослідження розглядали окремо CPU чи GPU методи; ця ж робота поєднує їх і заповнює пробіл у вигляді адаптивного методу. Отримано кількісну оцінку ефективності різних методів відсікання і їх комбінацій. Ці дані можуть бути використані для побудови моделей продуктивності графічних рушіїв. Розроблено програмний продукт – модуль відсікання для рушія Unity, який може бути застосований для майбутніх досліджень чи увійти до інструментарію оптимізації рендерингу. Таким чином, робота робить внесок як у теоретичну частину комп'ютерної графіки (поглиблюючи розуміння балансу CPU/GPU у задачах видимості), так і у практичну (надаючи готові рішення та рекомендації для впровадження).

Варто зазначити низку обмежень проведеної роботи, які необхідно враховувати при інтерпретації отриманих результатів. По-перше, усі тести та алгоритми у межах дослідження припускають використання статичної геометрії сцени. Наявність динамічних об'єктів вимагає перебудови ієрархії BVH у кожному

кадрі, що істотно знижує ефективність CPU-орієнтованих підходів. Це враховано у замірах та тестах, але питання ефективної побудови структур прискорення у роботі не розглядалось.

Іншим обмеженням є специфічність частини отриманих висновків до рушія Unity. Хоча загальні принципи балансування між CPU та GPU залишаються чинними і можуть бути застосовані в інших рушіях, таких як Unreal Engine або Godot, конкретні механізми реалізації, накладні витрати та особливості конвеєра рендерингу в них можуть суттєво відрізнятися.

Додатково слід враховувати, що експериментальні вимірювання виконувалися лише на одному типі графічного процесора, а саме на GPU архітектури NVIDIA Ampere. На інших платформах, зокрема GPU виробництва AMD або Intel, а також на різних поколіннях архітектур, абсолютні значення часу виконання можуть відрізнятися. Особливо це стосується архітектур AMD RDNA та RDNA2, які характеризуються іншими розмірами груп потоків і суттєво відмінним підходом до планування виконання.

Попри намагання наблизити тестові сценарії до реальних умов, зокрема шляхом моделювання приміщень і відкритих просторів, усі експерименти виконувалися на синтетичних сценах. У реальному ігровому або промисловому середовищі розподіл трикутників, складність шейдерів, використання різних рівнів деталізації та інші фактори можуть додатково впливати на ефективність відсікання, тому наведені рекомендації слід застосовувати з урахуванням конкретного контексту використання.

Окремо варто зазначити, що у межах цієї роботи не розглядалися підходи до відсікання на рівні meshlet або кластерів трикутників. Сучасні графічні API, зокрема Vulkan із підтримкою трасування променів та DirectX 12 Mesh Shaders, пропонують альтернативні механізми відсікання на рівні груп трикутників, які потенційно можуть забезпечити інші характеристики продуктивності. Дослідження цих підходів виходило за межі даної роботи.

У подальших дослідженнях доцільно зосередитися на знятті зазначених обмежень, насамперед у контексті підтримки динамічної геометрії, а також на аналізі відсікання на рівні трикутників і кластерів з використанням сучасних можливостей графічних API.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Akenine-Möller T., Haines E., Hoffman N. Real-Time Rendering. 4th ed. Boca Raton : CRC Press, 2018. 1201 p.
2. Epic Games. Nanite Virtualized Geometry. Unreal Engine Documentation. Epic Games, 2020. URL: <https://dev.epicgames.com/documentation/en-us/unreal-engine/nanite-virtualized-geometry-in-unreal-engine>
3. Горшенін М. О., Горшенін О. Є. Порівняльний аналіз CPU- та GPU-орієнтованих підходів до відсікання об'єктів у рендерингу реального часу. Матеріали VIII Всеукраїнської науково-технічної конференції «Комп'ютерні технології: інновації, проблеми, рішення», 02–03 грудня 2025 р. Житомир, 2025.
4. Unity Technologies. Occlusion Culling. Unity Manual. Unity Technologies, 2020. URL: <https://docs.unity3d.com/Manual/OcclusionCulling.html>
5. Epic Games. Precomputed Visibility Volumes. Unreal Engine Documentation. Epic Games, 2020. URL: <https://dev.epicgames.com/documentation/en-us/unreal-engine/precomputed-visibility-volumes-in-unreal-engine>
6. Wihlidal G. Optimizing the Graphics Pipeline with Compute. Advances in Real-Time Rendering in Games. SIGGRAPH 2021 Course. 2021. URL: <https://advances.realtimerendering.com/s2021/>
7. Intel Corporation. Software Occlusion Culling. Intel Developer Documentation. Intel, 2020. URL: <https://www.intel.com/content/www/us/en/developer/articles/technical/software-occlusion-culling.html>
8. Zhang H., Zhao J., Liu Y. A GPU-friendly Hybrid Occlusion Culling Algorithm for Large Scenes. Displays. 2023. Article 102533.
9. Bittner J., Wimmer M., Piringer H., Purgathofer W. Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. Computer Graphics Forum. 2004. Vol. 23, No. 3. P. 615–624.

10. Aaltonen S., Haar J. GPU-driven Rendering Pipelines. Advances in Real-Time Rendering in Games. SIGGRAPH 2015 Course. 2015. URL: https://advances.realtimerendering.com/s2015/aaltonenhaar_siggraph2015_combined_final_footer_220dpi.pdf
11. Microsoft. Mesh Shader Specification. DirectX-Specs. Microsoft, 2020. URL: <https://microsoft.github.io/DirectX-Specs/d3d/MeshShader.html>
12. Microsoft. ExecuteIndirect (Direct3D 12). Microsoft Learn Documentation. Microsoft, 2020. URL: <https://learn.microsoft.com/en-us/windows/win32/direct3d12/executeindirect>
13. Unity Technologies. Graphics.DrawMeshInstancedIndirect. Unity Scripting API. Unity Technologies, 2020. URL: <https://docs.unity3d.com/ScriptReference/Graphics.DrawMeshInstancedIndirect.html>
14. Intel Corporation. Masked Occlusion Culling. Intel Graphics Research Project. Intel, 2020. URL: <https://github.com/GameTechDev/MaskedOcclusionCulling>
15. Umbra Software. Umbra Occlusion Culling. Umbra Software Documentation. Umbra, 2020. URL: <https://umbrasoftware.com/>
16. Unity Technologies. Burst Package Documentation. Unity Technologies, 2020. URL: <https://docs.unity3d.com/Packages/com.unity.burst@latest/manual/index.html>
17. Unity Technologies. Job System. Unity Manual. Unity Technologies, 2020. URL: <https://docs.unity3d.com/Manual/JobSystem.html>
18. Pharr M., Jakob W., Humphreys G. Physically Based Rendering: From Theory to Implementation. 4th ed. Cambridge, MA : MIT Press, 2023.
19. NVIDIA Corporation. CUDA C++ Programming Guide. NVIDIA Documentation. NVIDIA, 2020. URL: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>

20. Advanced Micro Devices. AMD RDNA™ 2 Architecture. Whitepaper. AMD, 2020. URL: <https://www.amd.com/system/files/documents/rdna2-architecture-whitepaper.pdf>
21. Unity Technologies. ComputeBuffer. Unity Scripting API. Unity Technologies, 2020. URL: <https://docs.unity3d.com/ScriptReference/ComputeBuffer.html>
22. Unity Technologies. AsyncGPUReadback. Unity Scripting API. Unity Technologies, 2020. URL: <https://docs.unity3d.com/ScriptReference/UnityEngine.Rendering.AsyncGPUReadback.html>
23. Unity Technologies. Graphics.Blit. Unity Scripting API. Unity Technologies, 2020. URL: <https://docs.unity3d.com/ScriptReference/Graphics.Blit.html>
24. NVIDIA Corporation. Depth Precision Visualized. NVIDIA Developer Documentation. NVIDIA, 2020. URL: <https://developer.nvidia.com/content/depth-precision-visualized>
25. Unity Technologies. Scriptable Render Pipeline. Unity Manual. Unity Technologies, 2020. URL: <https://docs.unity3d.com/Manual/ScriptableRenderPipeline.html>
26. Unity Technologies. Rendering.CommandBuffer. Unity Scripting API. Unity Technologies, 2020. URL: <https://docs.unity3d.com/ScriptReference/Rendering.CommandBuffer.html>
27. Haines E., Akenine-Möller T., eds. Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX. Berkeley, CA : Apress, 2021. 504 p.
28. Bavoil L. Optimizing VK/VKR and DX12/DXR Applications Using Nsight Graphics: GPU Trace Advanced Mode Metrics. NVIDIA Developer Blog. NVIDIA, 2020. URL: <https://developer.nvidia.com/blog/optimizing-vk->

vkr-and-dx12-dxr-applications-using-nsight-graphics-gpu-trace-advanced-mode-metrics/

29. Microsoft. SV_Depth (HLSL System-Value Semantics). Microsoft Learn Documentation. Microsoft, 2020. URL: <https://learn.microsoft.com/en-us/windows/win32/direct3dhls/sv-depth>
30. Microsoft. GroupMemoryBarrierWithGroupSync (HLSL). Microsoft Learn Documentation. Microsoft, 2020. URL: <https://learn.microsoft.com/en-us/windows/win32/direct3dhls/groupmemorybarrierwithgroupsync>
31. Kenzel M., Bittner J., Wimmer M. A Survey of Occlusion Culling Algorithms. Computer Graphics and Applications. IEEE, 2020. Vol. 40, No. 4. P. 44–55.
32. Karras T., Aila T., Laine S. Parallel BVH Construction on GPUs. ACM Transactions on Graphics. 2020. Vol. 39, No. 6.
33. Laine S., Karras T. GPU Ray Tracing. Communications of the ACM. 2021. Vol. 64, No. 5. P. 64–73.
34. McGuire M., Hennessy P. Efficient Depth Buffer Techniques for Visibility Determination. Journal of Computer Graphics Techniques. 2021. Vol. 10, No. 2.
35. Wihlidal G., Lauritzen A. Practical GPU-Driven Rendering. Advances in Real-Time Rendering in Games (SIGGRAPH Course). 2022.
36. Aaltonen S., Räsänen V. Modern Rendering Pipelines on the GPU. Eurographics State of the Art Reports. 2021.
37. Zhang H., Liu Y., Zhao J. Hybrid GPU Occlusion Culling for Large-Scale Scenes. Journal of Visualization and Computer Animation. 2021. Vol. 32. P. 587–599.
38. Microsoft Corporation. DirectX 12 Ultimate Architecture Overview. Microsoft Learn Documentation. Microsoft, 2021. URL: <https://learn.microsoft.com/windows/win32/direct3d12/directx-12-ultimate>

- 39.NVIDIA Corporation. GPU-Driven Rendering: Design and Performance Considerations. NVIDIA Developer Documentation. NVIDIA, 2022. URL: <https://developer.nvidia.com/gpu-driven-rendering>
- 40.Unity Technologies. GPU Resident Rendering and Indirect Draw Calls. Unity Technical Blog. Unity Technologies, 2023. URL: <https://unity.com/blog/engine-platform/gpu-resident-rendering>

ДОДАТКИ

А.1 Інтерфейс стратегії відсікання

```

using UnityEngine;
namespace CullingSystem.Strategies
{
    /// <summary>
    /// Базовий інтерфейс для всіх стратегій відсікання.
    /// Дозволяє гнучко комбінувати різні алгоритми
    відсікання.
    /// </summary>
    public interface ICullingStrategy
    {
        /// <summary>
        /// Визначає, чи виконується стратегія на GPU
        (ComputeShader).
        /// </summary>
        bool IsGPU { get; }

        /// <summary>
        /// Визначає, чи є стратегія фільтром (потребує
        попередньо сформований список).
        /// </summary>
        bool IsFilterOnly { get; }

        /// <summary>
        /// Основний метод відсікання для заданого архетипу.
        /// </summary>
        /// <param name="archetype">Архетип об'єктів для
        обробки</param>
        void CullArchetype(Archetype archetype);

        /// <summary>
        /// Викликається перед початком відсікання кадру.
        /// </summary>
        void BeginCulling(Camera camera, CullingContext
        context);

        /// <summary>
        /// Викликається після завершення відсікання кадру.

```

```

        /// </summary>
        void EndCulling();
    }
}

```

A.2 Клас архетипу

```

using System.Collections.Generic;
using UnityEngine;

namespace CullingSystem
{
    /// <summary>
    /// Представляє групу об'єктів з однаковою геометрією та
    матеріалом.
    /// Дозволяє ефективно пакетне рендеринг через
    DrawMeshInstancedIndirect.
    /// </summary>
    public sealed class Archetype
    {
        public Mesh mesh;
        public Material material;
        public List<InstanceCPU> instances;

        // GPU буфери
        public ComputeBuffer instanceBuffer; //
        Трансформації та дані об'єктів
        public ComputeBuffer visibleIndicesBuffer; //
        Індокси видимих об'єктів
        public ComputeBuffer argsBuffer; //
        Параметри для DrawIndirect

        // Метадані
        public int instanceCount => instances?.Count ?? 0;
        public Bounds worldBounds;

        public void Initialize(int maxInstances)
        {
            instances = new List<InstanceCPU>(maxInstances);
            instanceBuffer = new ComputeBuffer(maxInstances,
            InstanceGPU.Stride);
        }
    }
}

```

```

        visibleIndicesBuffer = new
ComputeBuffer(maxInstances, sizeof(uint),
        ComputeBufferType.Append);
        argsBuffer = new ComputeBuffer(5, sizeof(uint),
        ComputeBufferType.IndirectArguments);
    }

    public void Dispose()
    {
        instanceBuffer?.Dispose();
        visibleIndicesBuffer?.Dispose();
        argsBuffer?.Dispose();
    }
}
}

```

А.3 Обчислювальний шейдер відсікання за фрустумом

```

// FrustumCulling.compute
#pragma kernel FrustumCull

// Структура даних об'єкта
struct InstanceData
{
    float4x4 objectToWorld;
    float3 boundsCenter;
    float3 boundsExtents;
};

// Буфери
StructuredBuffer<InstanceData> _Instances;
AppendStructuredBuffer<uint> _VisibleIndices;

// Параметри фрустуму (6 площин)
float4 _FrustumPlanes [6];
uint _InstanceCount;

// Тест AABV проти площини
bool IsOutsidePlane(float4 plane, float3 center, float3
extents)
{
    float3 normal = plane.xyz;
    float distance = dot(normal, center) + plane.w;
}

```

```

        float radius = dot(abs(normal), extents);
        return distance < -radius;
    }

    // Тест AABV проти всіх площин фрустуму
    bool IsInsideFrustum(float3 center, float3 extents)
    {
        [unroll]
        for (int i = 0; i < 6; i++)
        {
            if (IsOutsidePlane(_FrustumPlanes[i], center,
extents))
                return false;
        }
        return true;
    }

    [numthreads(64,1,1)]
    void FrustumCull(uint3 id : SV_DispatchThreadID)
    {
        if (id.x >= _InstanceCount)
            return;

        InstanceData instance = _Instances[id.x];

        // Трансформація центру та екстентів у світові координати
        float3 worldCenter = mul(instance.objectToWorld,
float4(instance.boundsCenter, 1)).xyz;
        float3 worldExtents =
abs(mul((float3x3)instance.objectToWorld,
instance.boundsExtents));

        if (IsInsideFrustum(worldCenter, worldExtents))
        {
            _VisibleIndices.Append(id.x);
        }
    }

```

A.4 Обчислювальний шейдер генерації HZB

```

// HZBGeneration.compute
#pragma kernel GenerateMip

```

```

// Текстури
Texture2D<float> _LastMip;
RWTexture2D<float> _CurrentMip;

// Розмір вихідної текстури
uint2 _OutputSize;

[numthreads(8,8,1)]
void GenerateMip(uint3 id : SV_DispatchThreadID)
{
    if (id.x >= _OutputSize.x || id.y >= _OutputSize.y)
        return;

    // Читання 4 сусідніх текселів з попереднього рівня
    uint2 baseCoord = id.xy * 2;
    float d0 = _LastMip[baseCoord + uint2(0, 0)];
    float d1 = _LastMip[baseCoord + uint2(1, 0)];
    float d2 = _LastMip[baseCoord + uint2(0, 1)];
    float d3 = _LastMip[baseCoord + uint2(1, 1)];

    // Reverse-Z: використовуємо мінімум для консервативності
    float minDepth = min(min(d0, d1), min(d2, d3));

    _CurrentMip[id.xy] = minDepth;
}

```

A.5 Обчислювальний шейдер оклюзійного відсікання

```

// HZBOcclusion.compute
#pragma kernel HZBOcclusionCull

struct InstanceData
{
    float4x4 objectToWorld;
    float3 boundsCenter;
    float3 boundsExtents;
};

StructuredBuffer<InstanceData> _Instances;
StructuredBuffer<uint> _InputIndices;
AppendStructuredBuffer<uint> _VisibleIndices;

Texture2D<float> _HZBTexture;

```

```

SamplerState sampler_HZBTexture;

float4x4 _ViewProjection;
float2 _ScreenSize;
uint _InputCount;
int _MaxMipLevel;
float _DepthBias;

void ProjectAABB(float3 center, float3 extents, out float2
minUV, out float2 maxUV, out float nearZ)
{
    // 8 кутів AABB
    float3 corners [8];
    corners [0] = center + float3(-extents.x, -extents.y, -
extents.z);
    corners [1] = center + float3( extents.x, -extents.y, -
extents.z);
    corners [2] = center + float3(-extents.x,  extents.y, -
extents.z);
    corners [3] = center + float3( extents.x,  extents.y, -
extents.z);
    corners [4] = center + float3(-extents.x, -extents.y,
extents.z);
    corners [5] = center + float3( extents.x, -extents.y,
extents.z);
    corners [6] = center + float3(-extents.x,  extents.y,
extents.z);
    corners [7] = center + float3( extents.x,  extents.y,
extents.z);

    minUV = float2(1, 1);
    maxUV = float2(0, 0);
    nearZ = 0; // Reverse-Z: найближча = найбільша

    [unroll]
    for (int i = 0; i < 8; i++)
    {
        float4 clip = mul(_ViewProjection,
float4(corners[i], 1));
        float3 ndc = clip.xyz / clip.w;
        float2 uv = ndc.xy * 0.5 + 0.5;

        minUV = min(minUV, uv);
    }
}

```

```

        maxUV = max(maxUV, uv);
        nearZ = max(nearZ, ndc.z); // Reverse-Z
    }
}

[numthreads(64,1,1)]
void HZBOcclusionCull(uint3 id : SV_DispatchThreadID)
{
    if (id.x >= _InputCount)
        return;

    uint instanceIdx = _InputIndices[id.x];
    InstanceData instance = _Instances[instanceIdx];

    float3 worldCenter = mul(instance.objectToWorld,
float4(instance.boundsCenter, 1)).xyz;
    float3 worldExtents =
abs(mul((float3x3)instance.objectToWorld,
instance.boundsExtents));

    float2 minUV, maxUV;
    float nearZ;
    ProjectAABB(worldCenter, worldExtents, minUV, maxUV,
nearZ);

    // Вибір рівня MIP на основі розміру проєкції
    float2 screenSize = (maxUV - minUV) * _ScreenSize;
    float maxDim = max(screenSize.x, screenSize.y);
    int mipLevel = clamp((int)ceil(log2(maxDim)), 0,
_MaxMipLevel);

    // Вибір глибини з HZB
    float2 centerUV = (minUV + maxUV) * 0.5;
    float sceneDepth =
_HZBTexture.SampleLevel(sampler_HZBTexture, centerUV, mipLevel);

    // Порівняння з урахуванням bias (Reverse-Z: об'єкт
видимий якщо ближче до камери)
    if (nearZ >= sceneDepth - _DepthBias)
    {
        _VisibleIndices.Append(instanceIdx);
    }
}

```

В.1 Системні вимоги

Мінімальні вимоги:

- Unity 2022.3 LTS або новіша версія
- GPU з підтримкою Compute Shaders (Shader Model 5.0)
- 8 ГБ RAM
- 2 ГБ відеопам'яті

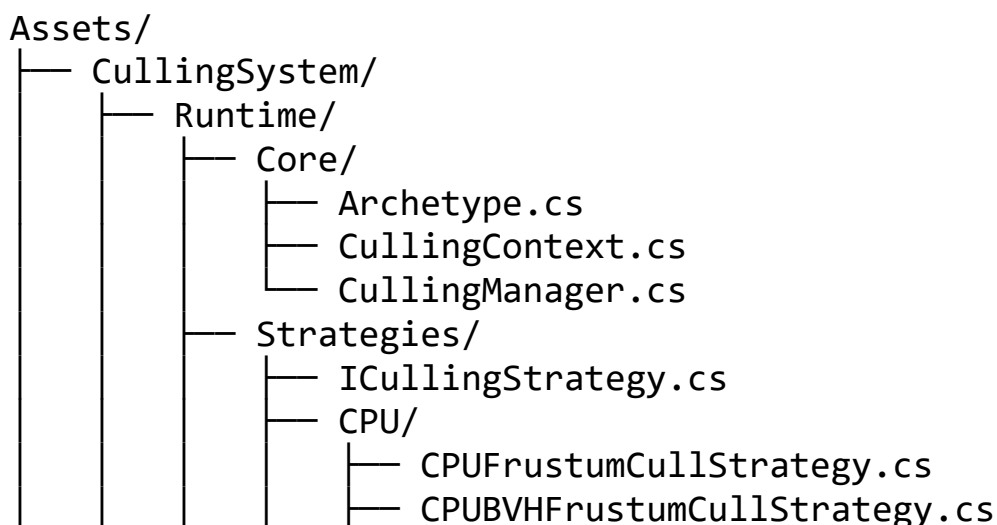
Рекомендовані вимоги:

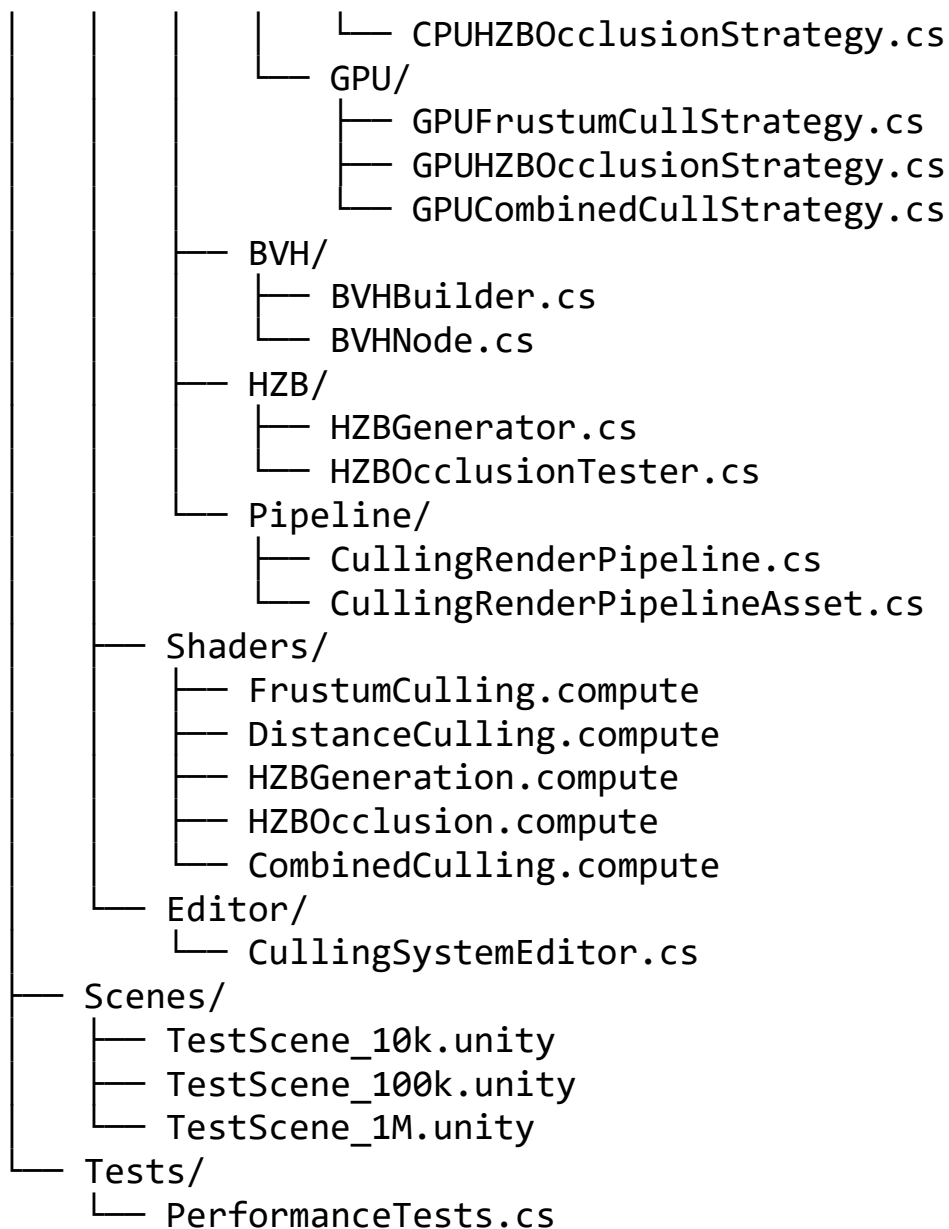
- Unity 6.x LTS
- NVIDIA RTX або AMD RX 6000 серії
- 16 ГБ RAM
- 8 ГБ відеопам'яті

В.2 Встановлення

1. Клонувати репозиторій:
`git clone http://git.ztu.edu.ua/ipz/2024-2025/ipzm-24-2/horshenin-mykhailo/gpu-cpu-culling-paper`
2. Відкрити проект в Unity Hub
3. Встановити необхідні пакети через Package Manager:
 - Burst 1.8.x або новіший
 - Collections 2.x
 - Mathematics 1.x
4. Перейти до налаштувань проекту (Edit - Project Settings - Graphics):
 - Встановити Scriptable Render Pipeline Asset на `CullingSystemRenderPipelineAsset`

В.3 Структура проекту





В.4 Базове використання

```

using CullingSystem;
using CullingSystem.Strategies;
public class GameManager : MonoBehaviour
{
    private CullingManager cullingManager;

    void Start()
    {
        cullingManager = new CullingManager();
    }
}
  
```

```

        // Автоматичний вибір стратегії на основі параметрів
сцени
        cullingManager.EnableAdaptiveStrategy = true;

        // Або ручне налаштування
        //
GPUFrustumCullStrategy(new
        cullingManager.SetStrategy(new
        //
GPUHZBOcclusionStrategy(new
        cullingManager.AddFilter(new
    }

    void Update()
    {
        cullingManager.PerformCulling(Camera.main);
    }
}

```

V.5 Налаштування параметрів

Параметри системи можна налаштувати через ScriptableObject `CullingSettings`:

MinObjectsForCulling - Мінімальна кількість об'єктів для активації відсікання
 CPUToGPUThreshold Поріг переключення з CPU на GPU
 HZBOcclusionEnabled Увімкнення оклюзійного відсікання
 DepthBias Зсув глибини для HZB 0.001
 TemporalFilterFrames Кількість кадрів темпорального фільтру
 SmallFeatureThreshold Поріг відсікання дрібних об'єктів (px²)