SECTION 10.
INFORMATION TECHNOLOGIES AND SYSTEMS

Гуменюк Вікторія Віталіївна
здобувачка третього (освітньо-наукового) рівня вищої освіти
кафедри комп'ютерних наук та інформаційних технологій
Житомирський державний університет імені Івана Франка, Україна

Науковий керівник: Іванов Дмитро Євгенійович
доктор технічних наук,
професор кафедри комп'ютерних наук та інформаційних технологій
Житомирський державний університет імені Івана Франка, Україна

СУЧАСНІ МЕТОДИ ОПТИМІЗАЦІЇ ОБЧИСЛЮВАЛЬНИХ РЕСУРСІВ У ГРАФІЧНИХ КОНВЕЄРАХ

Рендеринг тривимірних сцен є однією з ключових задач сучасної комп'ютерної графіки, що знаходить застосування в науковій візуалізації, системах автоматизованого проєктування (CAD/CAM/CAE), ігровій індустрії та системах віртуальної реальності. Безпосередній процес перетворення векторного опису 3D-сцени у двовимірне зображення (растр) виконується через складну послідовність етапів, відомий як конвеєр рендерингу. Технічно конвеєр рендерингу – це послідовність етапів обробки даних, необхідних для перетворення математичного опису тривимірної сцени (вершин, текстур, джерел світла) у готове двовимірне зображення на екрані монітора.

Зі зростанням складності об'єктів та вимог до графіки, розуміння ключових етапів рендерингу є критично важливим не лише для розробників графічних рушіїв, але й для прикладних програмістів, 3D-художників та технічних директорів. Ефективність фінальної візуалізації безпосередньо залежить від збалансованої роботи всіх етапів – від попередньої обробки на центральному процесорі (CPU) до фінального виведення зображення на екран. Оскільки етапи растеризації та обробки фрагментів є найбільш ресурсомісткими, відсутність оптимізації на ранніх стадіях призводить до нераціонального використання обчислювальних потужностей.

Відповідно, метою дослідження є аналіз сучасних методів оптимізації обчислювальних ресурсів у графічних конвеєрах для запобігання надмірному навантаженню на систему та підвищення швидкодії рендерингу.

Підготовчий етап графічного конвеєра повністю виконується на CPU і слугує фундаментальною базою для подальших обчислень. Головна мета цього етапу в контексті оптимізації — максимально розвантажити відеокарту (GPU), відфільтрувавши зайві дані та підготувавши геометрію таким чином, щоб ресурси графічного процесора використовувалися лише для обробки того, що дійсно формує фінальне зображення. [2]

Ефективність роботи всього конвеєра значною мірою залежить від застосування алгоритмів на стороні CPU, серед яких ключове місце займають методи відсікання та просторового структурування.

Завдання методів відсікання невидимих поверхонь — своєчасно ідентифікувати та запобігти відправленню на рендеринг об'єктів або їх частин, які користувач не побачить у фінальному кадрі.

Таблиця 1

Методи відсікання невидимих поверхонь

Назва методу	Суть методу
Відсікання за пірамідою видимості (Frustum Culling)	Базовий та найпоширеніший метод первинної оптимізації. Процесор перевіряє, чи потрапляють обмежувальні об'єми (Bounding Volumes) 3D-об'єктів у поле зору камери. Якщо об'єкт знаходиться повністю позаду камери або виходить за межі її кута огляду, він відкидається і не переходить на геометричний етап.
Відсікання перекриттів (Occlusion Culling)	Значно складніший метод, який є критично важливим для сцен із високою щільністю об'єктів (наприклад, міська забудова або густий ліс). Метод визначає, чи повністю перекритий один об'єкт іншим. Якщо, наприклад, стіна будівлі закриває меблі всередині кімнати, ці меблі виключаються з подальшої обробки, заощаджуючи колосальні ресурси.
Апаратні та програмні запити видимості	Оптимізація також може включати використання спеціальних структур даних (програмний Z-буфер на CPU) або застосування апаратних запитів, коли CPU запитує у GPU, чи було б відмальовано хоча б один піксель екранного простору, якби замість складного об'єкта відрендерили його найпростішу габаритну модель.
Відсікання задніх граней (Backface Culling)	Програмне відкидання цілих груп полігонів, що розвернуті до камери невидимою "спиною". Виконання частини цієї роботи заздалегідь на CPU дозволяє зменшити потік вершинних даних, що відправляються на відеокарту.

дані сформовано з [4]

Для забезпечення швидкої перевірки видимості в сценах, що налічують тисячі об'єктів, лінійний перебір елементів є надто повільним і неефективним. Тому для швидкого відсікання застосовуються ієрархічні структури даних: BVH (Bounding Volume Hierarchy), окто-дерева (Octrees), BSP-дерева (Binary Space Partitioning). [1]

Використовуючи структуру даних BVH об'єкти ієрархічно об'єднуються у групи, окреслені простими віртуальними фігурами (кубами або сферами). Головна перевага методу: якщо алгоритм бачить, що великий "батьківський" контейнер не потрапляє в кадр, він миттєво відкидає всі вкладені в нього "дочірні" об'єкти без необхідності перевіряти кожен із них окремо.

Під час застосування окто-дерева, будують математичну структуру, яка передбачає рекурсивний поділ тривимірного простору на вісім рівних кубів. Це дозволяє максимально швидко локалізувати об'єкти в конкретній просторовій зоні та відтинати порожні або невидимі області.

Для обробки статичної геометрії (наприклад, стін та архітектури) переважно застосовують методи BSP-дерева (Binary Space Partitioning). Простір поділяється площинами, що дозволяє системі швидко визначати точний порядок відображення полігонів від найвіддаленіших до найближчих.

Важливим інструментом оптимізації обчислювальних ресурсів на підготовчому етапі конвеєра є методи управління деталізацією (LOD — Level of Detail). Суть цих методів полягає у виборі відповідної версії тривимірної моделі залежно від того, на якій відстані від камери вона знаходиться.

Виокремлюють дискретний та безперервний LOD. Під час застосування дискретного LOD CPU вибирає одну із заздалегідь згенерованих версій 3D-моделі у залежності від відстані до камери. Для об'єктів на близьких планах обирається деталізована високополігональна сітка, а для фонових — максимального спрощена. Це радикально зменшує кількість вершин для обробки на наступних етапах.

Безперервний LOD передбачає динамічне спрощення сітки в реальному часі. І хоча в сучасних архітектурах безпосередня генерація нової топології (тесселяція) виконується на GPU, розрахунок критеріїв деталізації та первинне управління станами залишається завданням CPU. [3]

Після застосування вищезазначених методів оптимізації на центральному процесорі (відсікання, застосування просторових структур та управління деталізацією), підготовлений і значно зменшений масив геометричних даних передається через графічний API до пам'яті відеокарти. Саме на цьому етапі архітектура сучасних GPU продовжує оптимізаційний конвеєр, оскільки

навіть попередньо відфільтрована геометрія може містити елементи, які перекриваються в екранному просторі.

Після завершення підготовчого та геометричного етапів конвеєр переходить до растеризації. Цей етап вважається "серцем" графічного конвеєра, оскільки саме тут відбувається перетворення безперервної векторної математики (вершин та трикутників) у дискретні пікселі екрана, які називаються фрагментами.

Хоча основне завдання растеризації полягає у формуванні фрагментів, сучасні графічні архітектури використовують цей етап для додаткової, критично важливої оптимізації. Найбільш дієвим інструментом тут є алгоритм раннього тесту глибини (Early Depth Test).

У класичній моделі рендерингу перевірка глибини (порівняння фрагмента з даними Z-буфера) відбувається на фінальних стадіях конвеєра, вже після того, як для пікселя були розраховані освітлення, текстури та інші візуальні параметри. Проте етап обробки фрагментів є найбільш ресурсомістким, тому сучасні растеризатори намагаються відсіяти невидимі елементи ще до запуску "важкого" фрагментного шейдера.

Механізм раннього тесту глибини працює наступним чином: якщо графічний процесор (GPU) під час растеризації визначає, що розрахована глибина нових фрагментів трикутника гарантовано більша за ту, що вже зберігається в буфері глибини (тобто цей новий трикутник свідомо перекритий іншим об'єктом, наприклад, стіною), він миттєво відкидає ці фрагменти. Завдяки цьому відкиданню система уникає виконання складних математичних розрахунків PBR-матеріалів для тих пікселів, які користувач все одно не побачить у фінальному кадрі.

Впровадження раннього тесту глибини на етапі растеризації дозволяє уникнути марнотратного використання обчислювальних потужностей GPU, економлячи колосальну кількість ресурсів при рендерингу сцен із високим показником перекриття об'єктів (overdraw).

Висновки. Підбиваючи підсумки дослідження, можна стверджувати, що оптимізація графічного конвеєра є комплексним процесом, який вимагає збалансованого розподілу обчислювального навантаження між центральним та графічним процесорами. Ефективність фінальної візуалізації критично залежить від превентивного відсікання надлишкових даних ще на підготовчому етапі на стороні CPU. Застосування методів відсікання невидимих поверхонь, ієрархічних просторових структур (BVH, окто-дерев, BSP-дерев) та алгоритмів управління деталізацією (LOD) дозволяє

радикально зменшити обсяг геометрії, що обробляється на наступних етапах. Зі свого боку, на рівні відеокарти вирішальне значення для збереження продуктивності має алгоритм раннього тесту глибини (Early Depth Test). Його впровадження на етапі растеризації дозволяє апаратно відкидати перекриті фрагменти ще до виконання ресурсомістких інструкцій фрагментних шейдерів. Загалом, синергія методів програмного просторового відсікання та апаратних тестів глибини є необхідною умовою для рендерингу складних PBR-сцен у реальному часі без втрати швидкодії. Перспективи подальших досліджень у цьому напрямку полягають у розробці інтелектуальних систем динамічного балансування навантаження між CPU та GPU для автоматичної адаптації під мінливу складність тривимірних сцен.

Список використаних джерел:

1. Kessenich J., Baldwin D., Rost R. The OpenGL Shading Language, Version 4.60.8. The Khronos Group Inc. Aug-2023.
2. Pharr M., Jakob W., Humphreys G. Physically Based Rendering: From Theory to Implementation. 4 ed, 2023. URL: https://www.pbr-book.org/4ed/contents (дата звернення: 15.06.26)
3. Ray Tracing. URL: https://developer.nvidia.com/discover/ray-tracing (дата звернення: 18.06.26)
4. Rendering Pipeline Overview. URL: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview (дата звернення: 18.06.26).