ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
АВТОМАТИЗОВАНИХ СИСТЕМ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка рушію візуалізації з утилізацією ресурсів графічного процесору»

на здобуття освітнього ступеня магістра
зі спеціальності 126 Інформаційні системи та технології
(код, найменування спеціальності)
освітньо-професійної програми Інформаційні системи та технології
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Микола ЦЕЦЮРСЬКИЙ
(підпис) *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав:
здобувач вищої освіти Микола ЦЕЦЮРСЬКИЙ
група ІСДМ-62

Керівник: Юлія ГЛУЩЕНКО
науковий ступінь, вчене звання к.ф.-м.н., доцент

Рецензент:
науковий ступінь, вчене звання
_____
Ім'я, ПРІЗВИЩЕ

Київ 2023

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення автоматизованих систем

Ступінь вищої освіти Магістр

Спеціальність Інформаційні системи та технології

Освітньо-професійна програма Інформаційні системи та технології

ЗАТВЕРДЖУЮ
Завідувач кафедрою ІПЗАС

_____ Каміла СТОРЧАК
«_____» _____ 2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Цецюрському Миколі Валентиновичу
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: Розробка рушію візуалізації з утилізацією ресурсів графічного процесору

керівник кваліфікаційної роботи Юлія ГЛУЩЕНКО к.ф.-м.н., доцент
(Ім'я, ПРІЗВИЩЕ науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «19» 10.2023р. №145

2. Строк подання кваліфікаційної роботи «29» грудня 2023р.

3. Вихідні дані до кваліфікаційної роботи: вимоги до кваліфікаційної роботи магістра, науково-технічна література, документація щодо використання CUDA API.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
   4.1 Причини винекнення потреби у графічному процесорі
   4.2 Особливості створення програми для графічного процесору
   4.3 Дослідження швидкодії програми
5. Перелік графічного матеріалу:
   5.1 Алгоритм візуалізації методом трасування шляхів

5.2 Вихідне зображення програми
5.3 Презентація
6. Дата видачі завдання«19» жовтня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	19.10-10.11.23	
2	Дослідження доступних API для залучення ресурсів графічного процесору	11.11-17.11.23	
3	Написання теоретичного розділу звіту	18.11-24.11.23	
4	Розробка програмного модулю візуалізації	25.11-08.12.23	
5	Написання другого розділу звіту	09.12-12.12.23	
6	Проведення експериментальних досліджень розробленого модулю	13.12-14.12.23	
7	Написання третього розділу звіту	15.12-22.12.23	
8	Оформлення звіту відповідно до вимог постанови про дипломні роботи	23.12-25.12.23	
9	Розробка демонстраційних матеріалів	26.12-29.12.23	

Здобувач вищої освіти _____ Микола ЦЕЦЮРСЬКИЙ
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Юлія ГЛУЩЕНК
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 43 стор., 1 табл., 21 рис., 30 джерел.

Мета роботи – пришвидшити виконання методу трасування шляхів за допомогою програмного модулю, що виконуватиме обчислення за допомогою графічного процесору.

Об'єкт дослідження – програмне забезпечення візуалізації тривимірних сцен методом трасування променів.

Предмет дослідження – програмне забезпечення, що виконує обрахунки на графічному процесорі для прискорення процесу візуалізації.

Методи дослідження – аналітичний, емпіричний.

В роботі проведено аналіз особливостей та складнощів, пов'язаних з обрахунком методу трасування шляхів для візуалізації комп'ютерної графіки, досліджено, як графічний процесор допомагає вирішити поставлені проблеми та які особливості несе за собою програмування для графічного процесору. Було розроблено програмний модуль, що дозволив пришвидшити процес візуалізації в 24 рази у порівнянні з проведенням обчислень на ЦП. Проведено дослідження факторів, що впливають на швидкодію розробленої програми.

КЛЮЧОВІ СЛОВА: РЕНДЕР, ВІЗУАЛІЗАЦІЯ, ТРАСУВАННЯ ШЛЯХІВ, ГРАФІЧНИЙ ПРОЦЕСОР, ВІДЕОКАРТА, БАГАТОПОТОЧНІСТЬ, CUDA, ОПТИМІЗАЦІЯ.

ЗМІСТ

ВСТУП

Актуальність теми. За останнє десятиліття комп'ютерні технології візуалізації та апаратне забезпечення зробили великий стрибок у розвитку, змінивши можливості візуалізації, та відкривши до недавнього не уявні перспективи розвитку технології.

Останні досягнення, показали можливість інтеграції трасування променів у графічні процесори. Апаратне прискорення трасування променів, наприклад технологія NVIDIA RTX і DirectX Raytracing (DXR), революціонізували рендеринг, зробивши можливим реалістичне освітлення, віддзеркалення та тіні в програмах реального часу, таких як ігри.

Перехід до можливостей програмування, що підтримується такими архітектурами, як ядра CUDA NVIDIA та потокові процесори AMD, дозволив графічним процесорам виконувати й інші складні обчислення, окрім графіки, що призвело до їх використання в так званих суперкомп'ютерах, ШІ тощо.

Обчислення GPU загального призначення набрали обертів, використовуючи GPU для неграфічних завдань, таких як наукові симуляції, машинне навчання, криптографія тощо. Масовий паралелізм графічних процесорів зробив їх високоефективними для інтенсивних обчислювальних навантажень.

Такі API, як DirectX, OpenGL, Vulkan і CUDA, розвинулися для підтримки нових функцій, кращої продуктивності та оптимізації апаратного забезпечення. Ці API дозволяють розробникам ефективно використовувати можливості сучасного графічного обладнання.

Мета і завдання дослідження. Метою цього проєкту є прискорення процесу візуалізації комп'ютерної графіки методом трасування шляху за допомогою залучення графічного процесору для обчислень алгоритму та порівняння роботи програми з аналогом, що виконується виключно на центральному процесорі.

В процесі дослідження вирішувалися завдання вибору засобів залучення ресурсів графічного процесору до виконання обчислень та пошуку шляхів оптимізації програми для виконання на графічному процесорі.

Об'єкт дослідження: програмне забезпечення візуалізації тривимірних сцен методом трасування променів.

Предмет дослідження: програмне забезпечення, що виконує обрахунки на графічному процесорі для прискорення процесу візуалізації.

Методи дослідження. У процесі роботи над проєктом було проведено теоретичні дослідження на основі існуючих наукових робот за темою, результати яких викладені у першому та другому розділах звіту, практичні дослідження та порівняльний аналіз на основі розробленого експериментального програмного модулю.

Наукова новизна та практична значущість отриманих результатів. Розроблено програмний додаток, що утилізує ресурси графічного процесору для візуалізації тривимірної сцени та визначено параметри, що впливають на швидкодію процесу візуалізації.

Апробація результатів. Результати досліджень було апробовано на Всеукраїнській науково-технічній конференції «Технологічні горизонти: дослідження та застосування інформаційних технологій для технологічного прогресу України і світу».

Теоретична, методична та практична значущість отриманих результатів. За рахунок використання ресурсів графічного процесору вдалося досягти 24-кратного прискорення процесу візуалізації, та визначено фактори, що впливають на швидкість виконання програми на графічному процесорі.

Розроблений модуль можна інтегрувати у програмне забезпечення 3D моделювання або продовжувати оптимізувати швидкодію програми з метою досягнення візуалізації сцени у реальному часі.

РОЗДІЛ 1. ПРИЧИНИ ВИНЕКНЕННЯ ПОТРЕБИ У ГРАФІЧНОМУ ПРОЦЕСОРІ. СПЕЦИФІКА РОБОТИ З ГРАФІЧНИМ ПРОЦЕСОРОМ

1.1 Ознайомлення з поняттям рендерингу та методом трасування шляхів

Ключовим поняттям у темі комп'ютерної графіки є рендеринг – процес синтезу зображення з тривимірної сцени. Більшість алгоритмів рендерингу припадають до однієї з двох категорій: растеризація та трасування променів (рисунки 1.1, 1.2). На даний момент більшість графічних процесорів генерують зображення методом растеризації. Цей метод є дуже ефективним для апаратного прискорення, однак, якість зображення, отриманого цим методом обмежена та поступається в реалізмі методу трасування променів. Трасування променів, натомість, дозволяє генерувати фотореалістичні рендери. Зображення отримується шляхом симуляції оптичних властивостей світла, що дозволяє відтворити фізично точні тіні, віддзеркалення, рефракцію тощо.

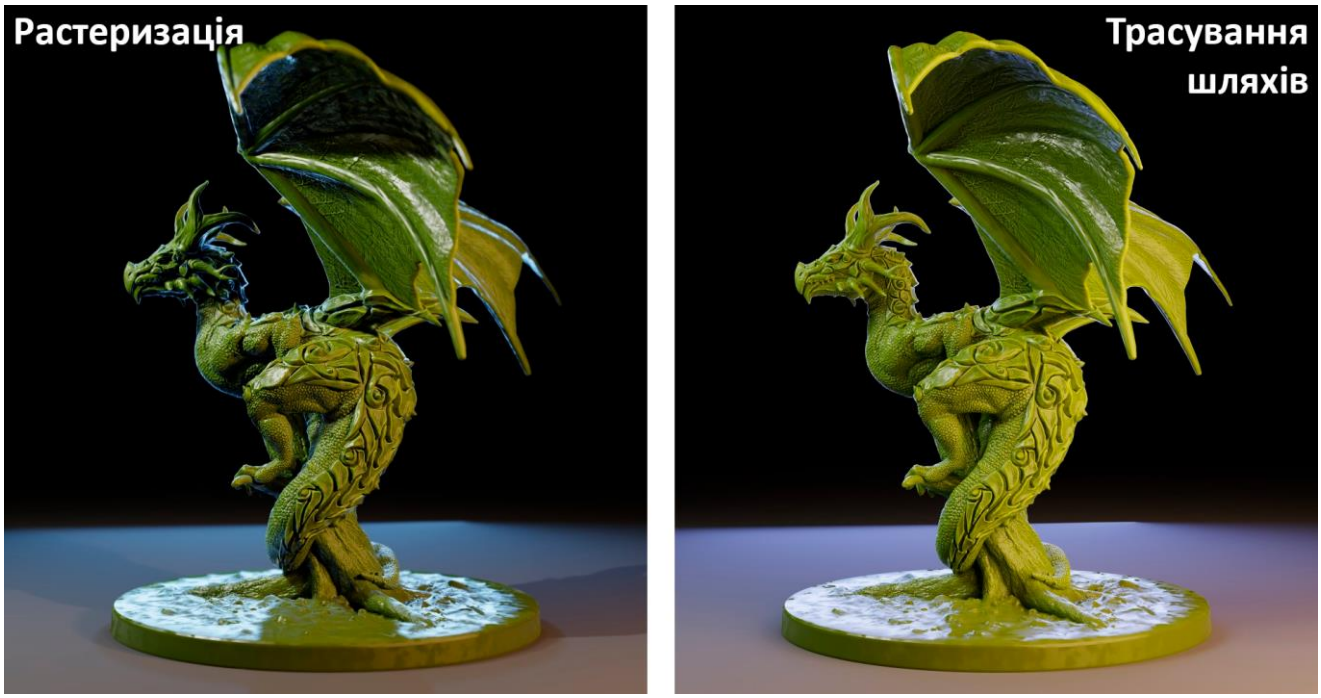


Рисунок 1.1 Візуалізація сцени зі студійним освітленням двома методами рендерингу (джерело: https://blog.render.st/cycles-vs-eevee-rendering-speed-comparison/)

Рисунок 1.2 Візуалізація сцени у закритому просторі двома методами рендерингу (джерело: https://blog.render.st/cycles-vs-eevee-rendering-speed-comparison/)

Трасування шляхів – це один з варінтів методу трасування променів, вперше запропонований Джеймсом Каджією. У його науковій статті 1986 року було вперше запропоновано рівняння рендерингу, яке є основою методу трасування шляхів. Це інтегральне рівняння, яке визначає кількість світлового випромінювання від заданої точки у певному напрямку як суму власного та відбитого випромінювань. Складність вирішення рівняння рендерингу полягає у нескінченній рекурсії функції, що унеможливлює вирішення задачі аналітично. Для вирішення даної проблеми Каджія запропонував використовувати метод Монте-Карла. Суть методу полягає в апроксимації результату за рахунок обчислення функції для певної скінченної добірки випадків та усереднення результату.

Метод трасування шляхів заснований на алгоритмах, які посилають промені з уявної камери (рис. 1.3). Коли промінь досягає поверхні якогось об’єкту в сцені, обраховуєть траєкторія відбиття (або заломлення в прозорих матеріалах) променя. При цьому також береться до уваги фізичні якості матеріалу: колір, Якість поверхні (чи вона є гладкою чи шороховатою), чи є матеріал металом чи діелектриком тощо.

Відбитий промінь подорожує далі, відбиваючись від об’єктів в сцені доки не досягне джерела світла або не вичерпає ліміт, на максимальну кількість відбиттів, який запроваджений для уникнення нескінченних циклів або прорахунку невигідно довгих траєкторій.

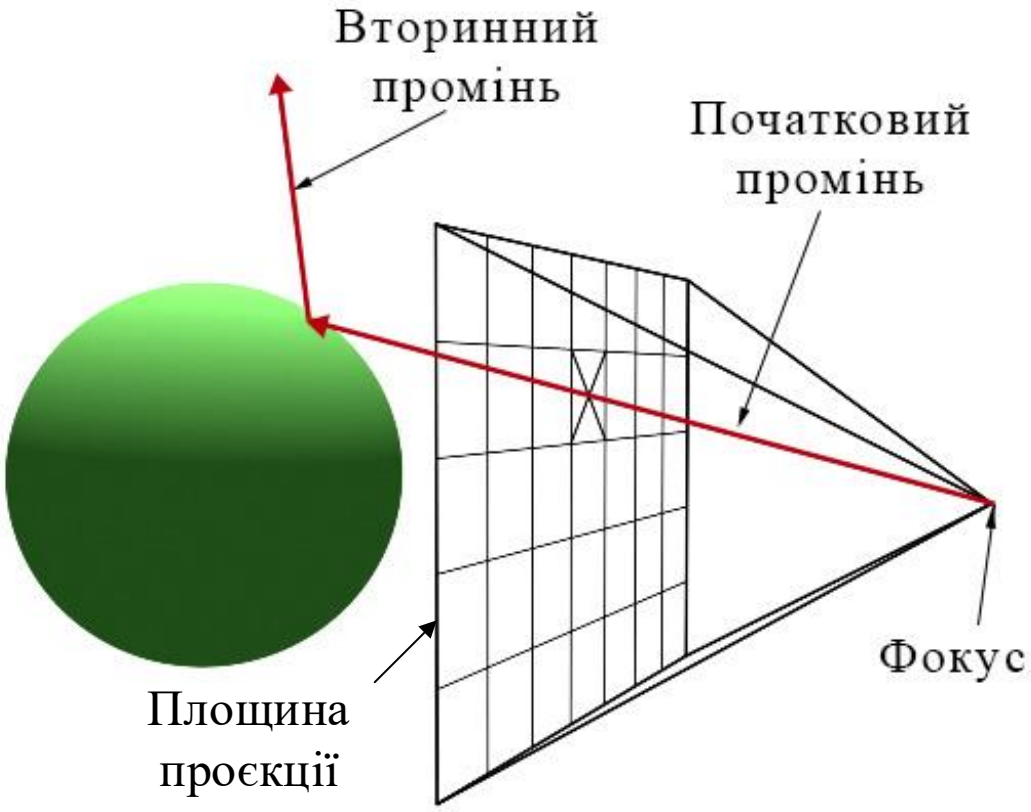


Рисунок 1.3 Віртуальна камера рушію візуалізації

Слід нагадати, що розв’язок одного шляху на один піксель зображення не достатньо для досягнення точної апроксимації – результуюче зображення матиме велику кількість шумів, низьку освітленість та різкість. Вирішенням цієї проблеми є обрахунок більшої кількості променів випущеної з різних точок на пікселі та подальше усереднення результату. Використання більшої вибірки дозволяє отримати більш точну апроксимацію рівняння рендерингу, що веде за собою більш чисте зображення без видимих артефактів.

Очевидно, що збільшення вибірки потребує пропорційного збільшення обрахунків. Це викликає необхідність пошуку компромісу між якістю зображення та часом обрахунку. Різні сцени можуть потребувати більшо вибірки аніж інші,

тому неможливо знайти ідеальне значення для застосування в усіх ситуацій. Часто, це рішення буде приймати художник виходячи з вимог до кінцвого зображення та особливостей сцени (наприклад, кількість та розмір джерел світла, наявність прозорих об’єктів, таких як скло або вода, складність геометрії в сцені тощо).

Блок-схема алгоритму трасування шляхів зображен на рисунку 1.4.

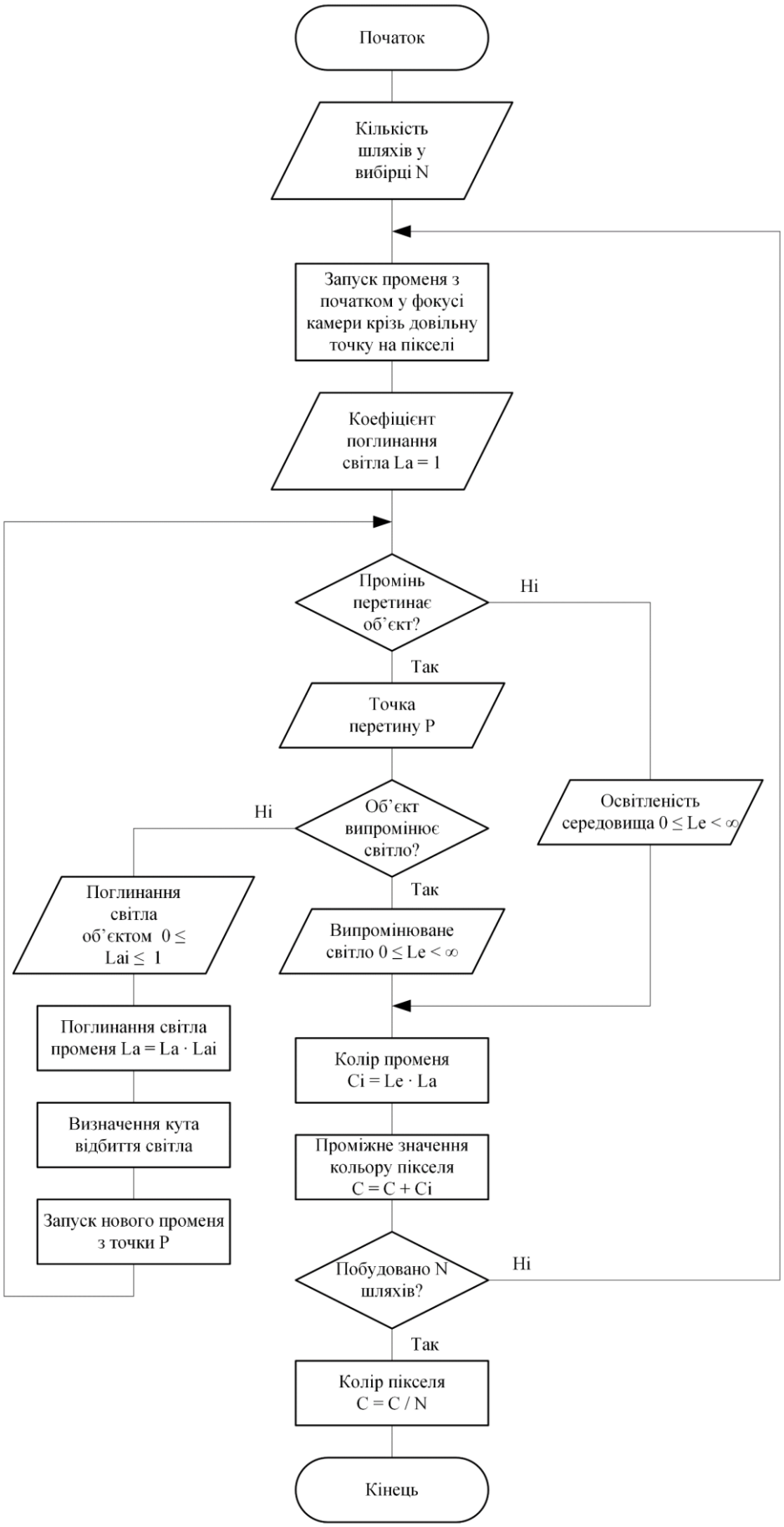


Рисунок 1.4 Блок-схема алгоритму трасування шляхів

1.2 Технічні виклики методу трасування шляхів

Одна з головних проблем методів трасування променів (включаючи метод трасування шляхів) полягає у необхідності зберігання у пам’яті усієї геометрії сцени, поки триває процес візуалізації. Подібної проблеми не виникає при застосуванні растеризації – будь-які предмети, що знаходяться за межами кадру ефективно не існують. Під час трасування променів, об’єкти, що знаходяться поза полем зору камери все ще можуть впливати на рендер, так як все ще можуть впливати на траєкторію розповсюдження світла у сцені (рисунок 1.5).

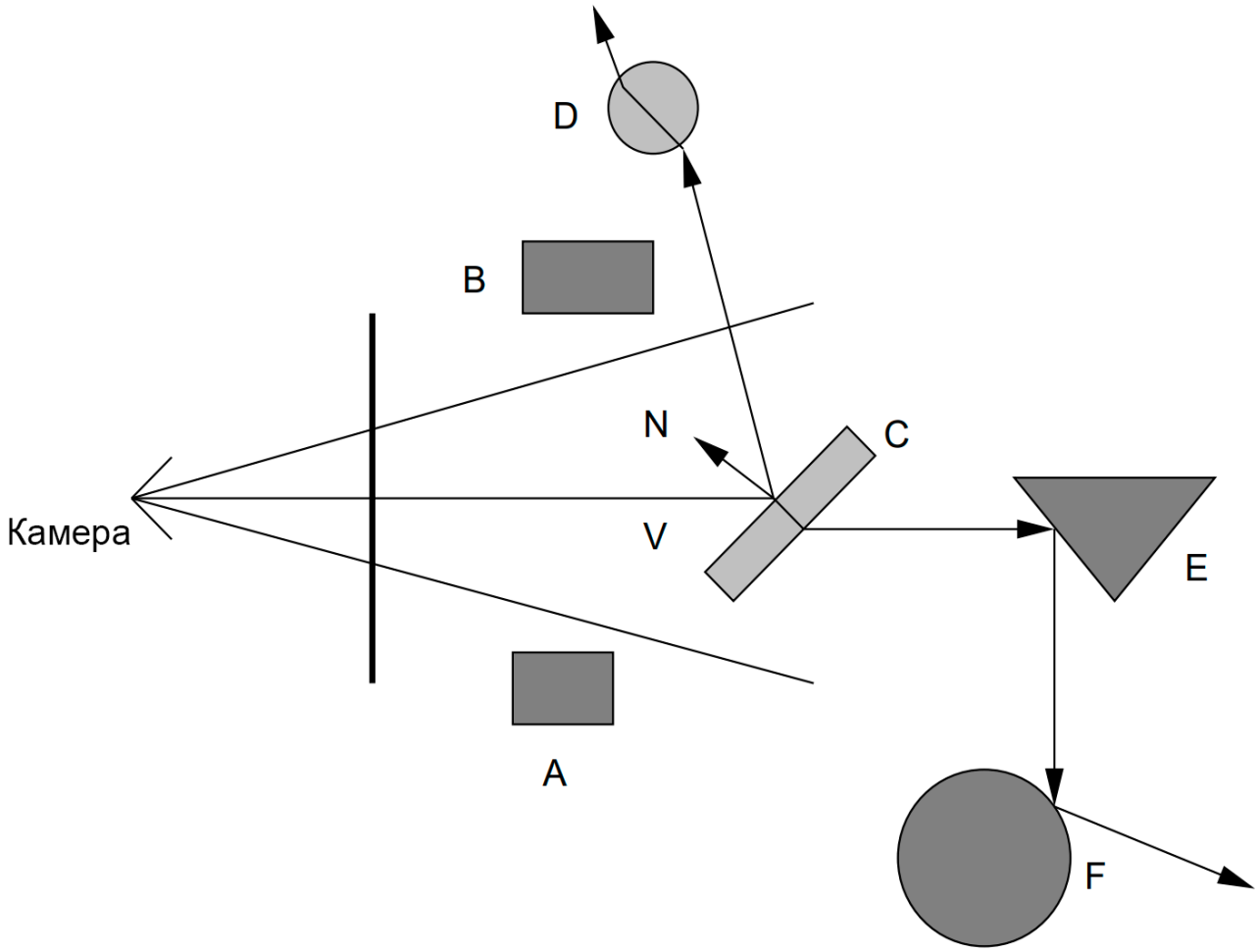


Рисунок 1.5 Схема розповсюдження променів поза межі кадру (джерело: https://www.doc.ic.ac.uk/~dfg/graphics/graphics2008/GraphicsLecture09.pdf)

До того ж проблема ускладнена тим, як відбувається процес пошуку найближчої точки перетину променя з об’єктами в сцені. Загальна ідея вирішення

цієї задачі полягає в тому, щоб прирівняти рівняння променя до рівняння об’єкта і визначити чи існує рішення. Якщо існує реальне рішення, то перетин знайдено. Процес повторюється для кожного об’єкта, результати порівнюються та знаходиться найближча точка перетину. Ця частина обчислень є найбільшим обмежувальним фактором оптимізації алгоритму трасування шляхів та становить приблизно 70-80% загального часу розрахунку.

Найвживаніше рішення цієї проблеми полягає в тому, щоб упорядкувати графічні примітиви в ієрархічні просторові структури (Bounding Volume Hierarchy, BVH) (рисунок 1.6). Таким чином, перевірка променів і сцен може швидко усунути непов’язані між собою простори та визначити найближчі до променів примітиви. Широко використовуваною структурою прискорення є деревовидна структура. Листові вузли дерева містять усі примітиви сцени, а внутрішні вузли дерева використовуються для поділу великого просторового представлення на кілька менших просторових областей або розкладання об’єктів у сцені на менші набори об’єктів. На основі просторових структур трасування променів поділяється на два етапи: перехід променя зі структурою прискорення та тест перетину між променем і примітивами.

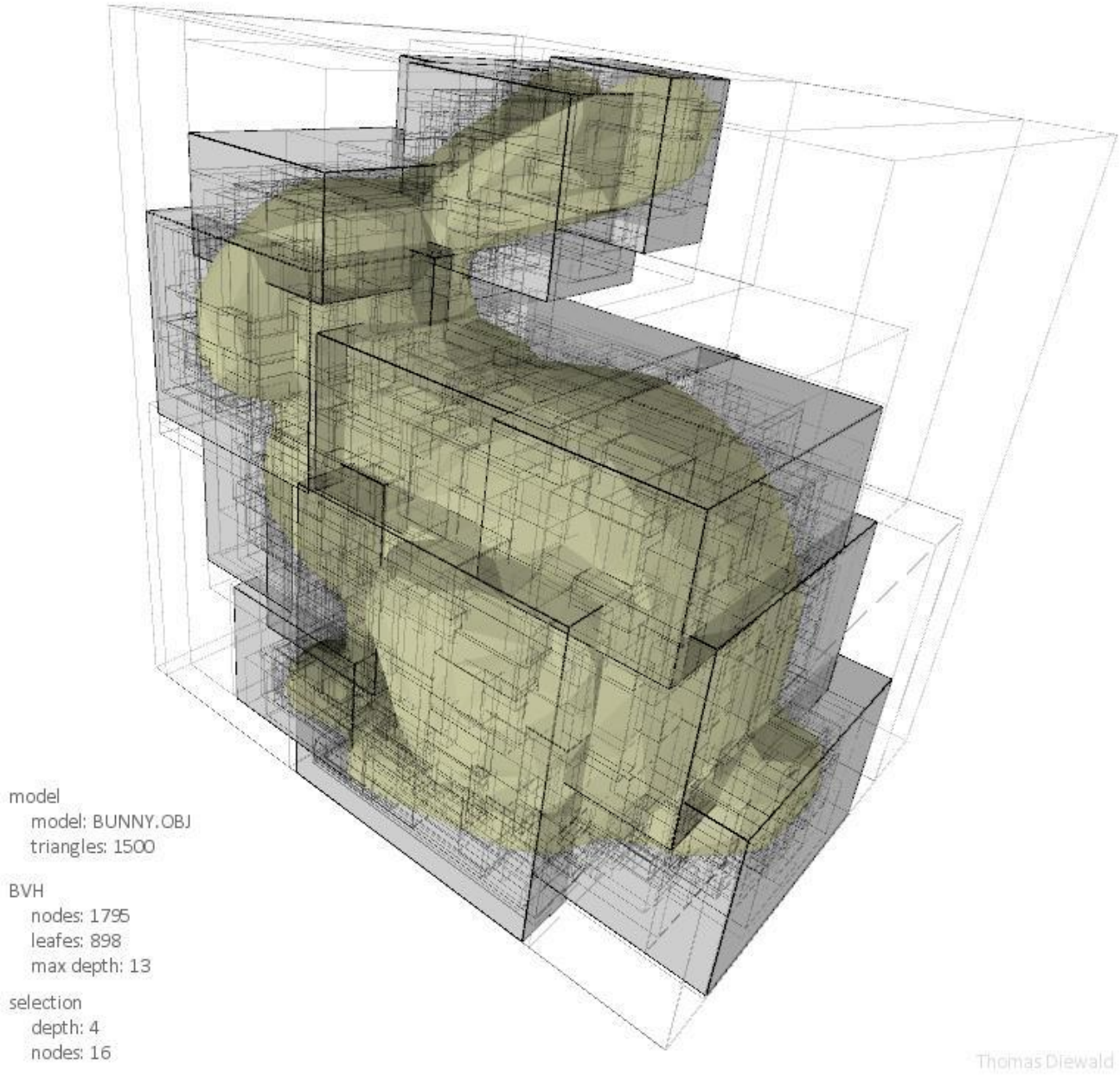


Рисунок 1.6 Візуалізація ієрархічної просторової структури 3D моделі (джерело оригінального зображення: https://cseweb.ucsd.edu/~sht005/ray_tracing_info.html)

1.3 Багатопоточність у контексті методу трасування променів

Метод трасування променів отримує значну вигоду від застосування паралельних обчислень, оскільки великий обсяг розрахнків, яких він потребує можуть бути дуже легко розділені на окремі потоки. Існує два основні методи розділення роботи алгоритму трасування променів: за попитом та за даними.

Оскільки кожен піксель зображення *m*n* обчислюється незалежно, найбільш очевидним способом розкладання є поділ зображення на *p* частин, де *p* — кількість доступних процесорів, і кожен процесор обчислював би *m*n/p* пікселів. В ідеальних умовах це пришвидшило б обчислення у *p* разів. Цей підхід називається паралельним трасуванням променів за запитом. Створюється кілька потоків, кожен з яких містить певну підмножину пікселів зображення, і ці потоки призначаються процесорам. Вхідна сцена копіюється в локальну пам'ять кожного процесора. Процесори рендерять свої частини, повертають обчислені пікселі, отримують інше завдання, якщо воно є, і врешті-решт остаточне зображення складається з цих частин. Основні переваги цього підходу: легке розділення роботи на підзадачі та хороше масштабування. Основним недоліком є те, що вхідну сцену потрібно копіювати в локальну пам'ять кожного процесора, що створює проблему, якщо сцена дуже велика.

Розпаралелення алгоритму трасування променів за даними розбиває сцену на окремі області і призначає ці області процесорам. Кожен процесор відповідає за обчислення, пов’язані з об’єктами в межах своєї секції сцени. Якщо промінь, створений на одному процесорі, потребує даних з іншого процесора, вони передаються цьому процесору. Визначення кількості променів, які проходитимуть через ділянку сцени, щоб оцінити ділянки, які вимагають найбільшої обробки, є однією з найскладніших проблем для ефективної реалізації цього методу розпаралелення алгоритму трасування променів. Основна перевага цього підходу полягає в тому, що вхідну сцену не потрібно повністю копіювати на кожен процесор. Основним недоліком є значні витрати на зв’язок завдань і передачу променів від одного ядра до іншого.

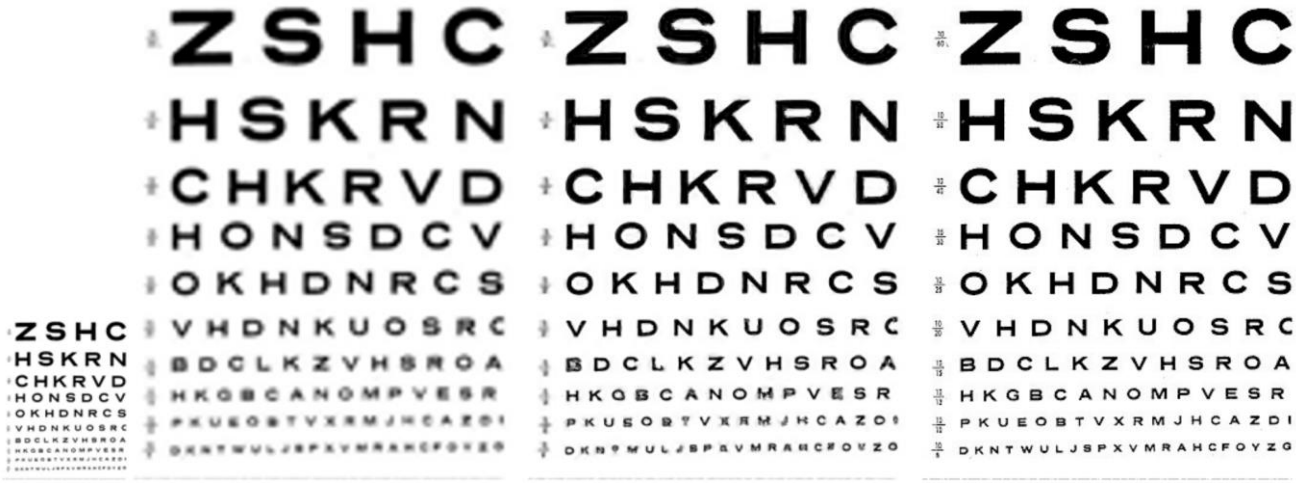
Завдяки особливостям дизайну, графічні процесори мають значну перевагу над центральним процесором у виконанні паралельних задач. Довгий час трасування променів було важко реалізувати на графічних процесорах, оскільки ранні графічні процесори не підтримували обчислення загального призначення. Однак за останні роки було досягнуто великого прогресу щодо оптимізації графічних процесорів для трасування променів. Комерційні графічні процесори

також представили прискорену архітектуру для трасування променів. У 2018 році NVIDIA випустила перший графічний процесор в архітектурі Turing з ядрами трасування променів. У 2021 році була створена архітектура Ampere з RT ядрами другого покоління. Ядра трасування променів замінюють програмну емуляцію, виконуючи обхід дерева BVH та тести перетину. Запит на промінь надсилається від потокового мультипроцесора до ядра трасування променів. Ядро виконує пошук перетину променя з геометрією сцени. Коли визначено відповідну точку перетину, результат повертається до потокового мультипроцесора для подальшої обробки. Виконання даних операцій на апаратному рівні замість програмної емуляції підвищила швидкодію методів трасування променів в рази та навіть зробила можливим візуалізацію цим методом можливою у реальному часі.

1.5 Інші методи прискорення візуалізації методом трасування шляхів

Окрім безпосередньо пришвидшення виконання самого алгоритму трасування променів, сучасні графічні процесори мають й інші інструменти, що дозволяють підвищити швидкодію рушія візуалізації. Поява апаратного прискорення нейронних мереж в сучасних графічних процесорах посприяла розвитку технологій апскейлінгу та знешумлення.

Ідея апскейлінгу полягає у тому, щоб візуалізувати зображення меншої роздільної здатності, після чого, за рахунок технік машинного навчання, збільшити зображення до бажаної роздільної здатності (рисунок 1.7). Зменшення кількості пікселів, що необхідно обрахувати дозволяє пропорційно зменшити час та обсяг обрахунків, необхідних для рендерингу.



Вхідне зображення | Бікубічна інтерполяція | Нейромережа | Оригінал

Рисунок 1.7 Апскейлінгу зображення нейронною мережею (джерело: https://www.wisdom.weizmann.ac.il/~vision/single_image_SR/files/single_image_SR.pdf)

Іншим шляхом зменшити кількість розрахунків є зменшення розміру вибірки для обрахунку пікселя. Однак, як зазначено у підрозділі 1.1, пошук оптимального розміру вибірки є балансом між швидкодією та якістю зображення. Малі вибірки призводять до зернистого зображення з великою кількістю шумів. Алгоритми знешумлення ставлять на меті мінімізувати цю проблему (рисунки 1.7 та 1.8).

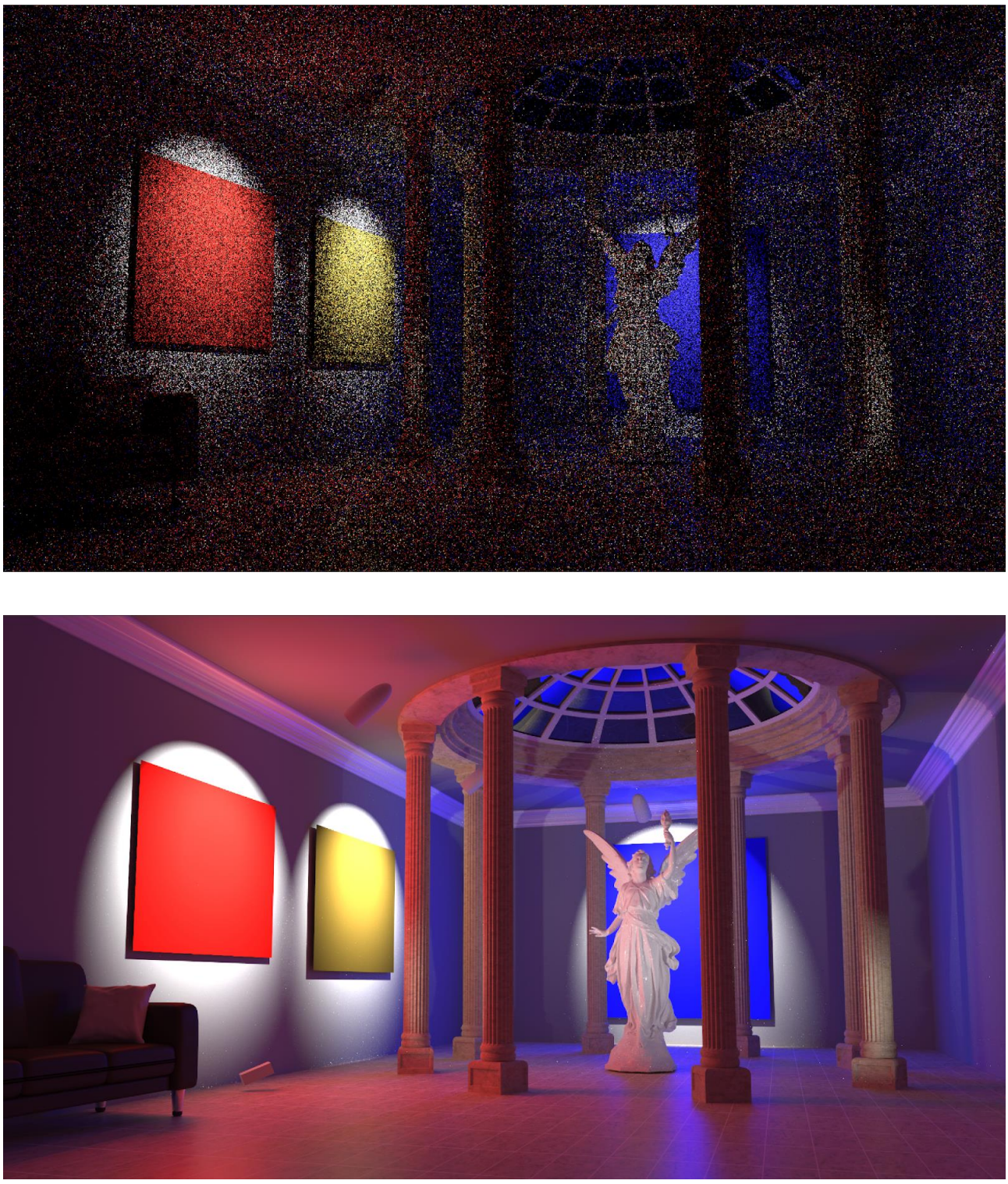


Рисунок 1.8 Зображення з малою вибіркою променів до та після застосування алгоритму зменшення шуму (джерело: https://tom94.net/data/publications/mueller20neural/interactive-viewer/)

Усунення шумів на зображенні зазвичай базується на одному з трьох методів: просторова фільтрація, часове накопичення та машинне навчання.

Просторова фільтрація вибірково змінює частини зображення шляхом повторного використання подібних сусідніх пікселів. Перевага просторової фільтрації полягає в її швидкодії, однак призводить до відносно розмитого зображення яке поступається результатам роботи інших методів. Окрім цього даний метод не підходить для графіки у реальному часі через помітне мерехтіння та артефакти між кадрами анімації.

Часове накопичення повторно використовує дані з попереднього кадру, щоб визначити, чи є в поточному кадрі артефакти або візуальні аномалії, які можна виправити. Цей метод не створює розмитість та часові нестабільності, що присутні у випадку просторової фільтрації. Однак цей метод потребує збору інформації впродовж кількох кадрів анімації, що унеможливлює його застосування для статичного зображення, а також привносить затримку у додатки з візуалізацією у реальному часі.

Техніка машинного навчання використовує нейронну мережу для реконструкції сигналу. Мережа тренується з використанням великої кількості природних зображень, і навчається доповнювати або виправляти зображення, щоб ті виглядати як сегменти навчальних зображень.

Комбінація технік зменшення шуму та апсемплінгу дозволяє істотно пришвидшити процес рендерингу і навіть виконувати візуалізацію методом трасування шляхів в реальному часі.

Апаратне прискорення операцій нейронних мереж виконується за допомогою так званих тензорних ядер. Тензорні ядра — це спеціалізовані ядра, які забезпечують навчання змішаної точності.

Обчислення зі змішаною точністю названо так тому, що хоча введені матриці можуть бути низькоточними FP16, остаточний результат буде FP32 лише з мінімальною втратою точності на виході. По суті, це швидко прискорює обчислення з мінімальним негативним впливом на кінцеву ефективність моделі.

Перше покоління тензорних ядер прийшло з мікроархітектурою Volta. Ці ядра дозволили навчання змішаної точності з форматом чисел FP16. Це збільшило потенційну пропускну здатність графічних процесорів до 12 разів.

Друге покоління тензорних ядер з’явилося разом із випуском графічних процесорів Turing. Підтримувані точності тензорних ядер були розширені з FP16, щоб також включити Int8, Int4 та Int1. Це дозволило прискорити продуктивність графічного процесора до 32 разів у порівнянні з графічним процесором Pascal.

У лінійці графічних процесорів Ampere представлено третє покоління тензорних ядер.

У графічному процесорі Ampere архітектура базується на попередніх інноваціях мікроархітектур Volta і Turing, розширюючи обчислювальні можливості до FP64, TF32 і bfloat16. Ці додаткові формати ще більше прискорюють операції машинного навчання. Формат TF32, наприклад, працює подібно до FP32, одночасно забезпечуючи до 20-кратного прискорення без зміни коду.

РОЗДІЛ 2. РОЗРОБКА РУШІЮ ВІЗУАЛІЗАЦІЇ З УТИЛІЗАЦІЄЮ РЕСУРСІВ ГРАФІЧНОГО ПРОЦЕСОРУ

2.1 Особливості розробки додатку, що залучатиме ресурси графічного процесору

Написання коду, який може виконуватися на графічному процесорі несе за собою деякі обмеження. Зі стрімким розвитком технології, архітектура графічних процесорів істотно змінюється від покоління до покоління, що має безпосередній вплив на сумісність розроблюваної програми з апаратним забезпеченням комп'ютера. Хоч у загальному випадку спрацьовує правило зворотної сумісності і розробка програми з розрахунком на більш старі моделі графічних процесорів дозволяє запускати програму і на більш сучасних відеокартах, розділ найпоширеніших графічних процесорів на два конкуруючі бренди — AMD та NVIDIA — інколи змушує розробників робити вибір між платформами. Інколи, певний функціонал може бути повністю відсутнім на графічному процесорі одного з виробників або мати іншу імплементацію. У таких випадках поширена практика розроблювати окремий функціонал програми спеціально під потреби певного графічного процесору або повністю його оминати на одній з платформ.

Існують також і мультиплатформенні рішення, що забезпечують запуск та успішне виконання програми на широкому спектрі відеокарт. Нажаль, програми, написані за допомогою мультиплатформених API зазвичай істотно поступаються у швидкодії рішенням, розробленим для специфічних відеокарт та, як правило, не можуть розкрити потенціал апаратного забезпечення у повній мірі.

2.2 Дослідження існуючих підходів до залучення ресурсів графічного процесору для виконання розрахунків

Існує велика кількість API для розробки програм, код яких буде виконуватися на графічному процесорі. Головними факторами при виборі API є спектр відеокарт та

операційних систем, на яких він працює, рівень доступу до апаратного забезпечення та набір функціоналу.

За переважний період часу, що графічні процесори існували, растеризація лишалася єдиним широковживаним методом візуалізації й у зв’язку з цим, популярні API довго лишалися орієнтованими на операціях, затребуваних у методі растеризації. Натомість трасування променів і трасування шляхів використовує цілковито інакший підхід до створення зображення і тому потребує специфічного функціоналу. На щастя, усі найпоширеніші графічні API за останні роки додали функціонал для підтримки методів трасування променів. У процесі виконання дипломного проєкту було розглянуто 4 API, які є стандартом індустрії комп’ютерної графіки:

- OpenGL
- DirectX
- Vulkan
- CUDA

При виборі інструментарію для реалізації програми було прийнято до уваги доступні мови програмування, функціонал та підтримувані графічні процесори.

2.2.1 Короткий огляд DirectX та його доцільність для реалізації рушію візуалізації методом трасування шляхів

DirectX є колекцією API від Майкрософт для роботи з мультимедіа на платформі Windows. Спочатку, назви API починалися з Direct, наприклад Direct3D, DirectDraw, DirectMusic, DirectPlay, DirectSound тощо. Назва DirectX була придумана як скорочення для всіх цих API (X заміняє конкретну назву API) і незабаром стала назвою всієї колекції. Direct3D є модулем для роботи з тривимірною графікою. Будучи найбільш вживаним компонентом пакету, назва Direct3D стала синонімічна з DirectX та назви стали взаємозамінні. Даний API широко використовується для програм візуалізації у реальному часі, таких як ігри. Весь час свого існування, DirectX був орієнтований на метод растеризації, однак, в останній версії API — DirectX 12, Майкрософт додала новий модуль DXR — набір інструментів спеціально для задач

трасування променів, таких як генерація променю та перевірка на найближчий перетин.

DirectX підтримує усі сучасні відеокарти обох виробників — NVIDIA та AMD, однак функції апаратного прискорення трасування променів доступне тільки для графічних процесорів NVIDIA. Єдиною підтримуваною мовою є C++. Загалом, DirectX є дуже потужним інструментом для залучення ресурсів графічного процесору для виконання задач візуалізації у тому числі методом трасування променів. Обмеження щодо сумісності виключно з ОС Windows може бути проблемою залежно від вимог та потреб до програми, однак не грає ролі в контексті даного проєкту.

2.2.2 Короткий огляд OpenGL та його доцільність для реалізації рушію візуалізації методом трасування шляхів

Аналогічно до DirectX, OpenGL є програмним інтерфейсом, що дозволяє програмісту комунікувати з графічним процесором комп’ютера. OpenGL розшифровується як Open Graphics Library (відкрита графічна бібліотека). На відміну від DiectX, OpenGL є мультиплатформенною і дозволяє створювати додатки для всіх поширених операційних систем та різного апаратного забезпечення включаючи мобільні телефони.

Але, можливо, найбільшою перевагою, яку OpenGL надає розробникам, є підтримка розширень. Якщо специфікація OpenGL не передбачає підтримки певної функціональності, постачальник апаратного чи програмного забезпечення може вирішити самостійно додати цю функціональність за допомогою розширень. Розширення можуть надавати потужні функціональні можливості, але зазвичай лишаються специфічними для конкретної реалізації OpenGL від конкретного постачальника.

OpenGL доступний для використання на більшості мов програмування, включаючи C, C++, C#, Java, Python тощо.

Незважаючи на мультиплатформенні можливості та гнучкий функціонал, даний API починає втрачати свою актуальність та підтримку розробників сучасних відеокарт. Інші API, розглянуті у цьому розділі надають можливості для більш

ефективного використання ресурсів графічного процесору та є більш надійними для забезпечення сумісності на роки вперед.

2.2.3 Короткий огляд Vulkan та його доцільність для реалізації рушію візуалізації методом трасування шляхів

Фактично, Vulkan є продовженням OpenGL. Створений тією ж самою компанією, він є новим поколінням мультиплатформенного API для доступу до графічного процесору з покращеними можливостями оптимізації.

API Vulkan зменшує робоче навантаження на процесори за допомогою пакетної обробки й інших оптимізацій низького рівня. Це зменшення робочого навантаження ЦП дозволяє ЦП виконувати більше обчислень або візуалізації, ніж було б можливо в іншому випадку.

API Vulkan приймає шейдери, уже переведені в проміжний бінарний формат під назвою Standard Portable Intermediate Representation (SPIR-V). SPIR-V є аналогом бінарного формату High-Level Shading Language (HLSL) що використовується у DirectX. Попередня компіляція шейдерів дозволяє покращити швидкість ініціалізації програми і для кожної сцени можна використовувати більшу різноманітність шейдерів.

Vulkan має численні переваги перед іншими API для відеокарт, забезпечуючи найкращу міжплатформенну підтримку, кращу підтримку багатопоточних процесорів, нижче навантаження на ЦП та широку підтримку ОС.

2.2.4 Короткий огляд CUDA та його доцільність для реалізації рушію візуалізації методом трасування шляхів

CUDA розшифровується як Compute Unified Device Architecture та є набором інструментів, які дозволяють розробникам комунікувати з ядрами загального призначення графічних процесорів NVIDIA. До нього входить однойменний API та компілятор nvcc.

CUDA SDK націлений на клас програм, контрольна частина яких виконується як процес на центральному процесорі, і які використовують один або кілька

графічних процесорів NVIDIA як співпроцесори для прискорення паралельних задач за моделлю «одна програма, багато потоків даних» (SPMD). Такі задачі є самодостатніми в тому сенсі, що вони можуть бути виконані та завершені групою потоків графічного процесора повністю без втручання центрального процесору.

Код для графічного процесору пишеться як набір функцій на мові, яка по суті є C++ з деякими синтаксичними доповненнями, а також анотаціями для розрізнення різних типів пам’яті, яка існує у графічному процесорі. Функції можуть мати параметри, і їх можна викликати за допомогою синтаксису, який дуже схожий на звичайний виклик функцій C++, але трохи розширений для можливості вказати матрицю потоків GPU, які повинні виконувати викликану функцію.

Обмеження щодо підтримки виключно відеокарт компанії NVIDIA є суттєвим недоліком, однак також надає перевагу оптимізації під специфічне апаратне забезпечення і надає можливість досягти швидкодію, порівняну з Vulkan, не вдаючись у низькорівневе програмування.

2.3 Вибір засобів для розробки програмного модулю

Найпершим фактором при виборі засобів розробки для реалізації даного проєкту є необхідність визначитися з API, оскільки він визначає мову програмування, яка буде використана, доступний функціонал, та сумісність з апаратним забезпеченням.

Тестові комп’ютери, на яких буде проводитися дослідження програми обладнані графічними процесорами RTX 3050 Ti та RTX 3070 Ti виробництва NVIDIA. Оскільки OpenGl та Vulkan є мультиплатформенними, а DirectX та CUDA мають повну підтримку графічних процесорів NVIDIA, обмежень стосовно сумісності з апаратним забезпеченням комп’ютера немає і всі чотири розглянуті API потенційно можуть бути використані.

Нажаль, OpenGL добігає кінця свого життєвого циклу. Цей API безпосередньо полягає на підтримку від виробників відеокарт та сторонніх розробників, і ця підтримка згасає на фоні появи нових альтернатив. Тому використання OpenGL є небажаним для цього проєкту з точки зору перспектив подальшого розвитку.

Vulkan API надає широкий функціонал та потенціал досягти максимальну швидкодію, однак високий поріг входження для опанування та відсутність необхідності у головній перевазі даного API роблять вибір цього засобу не бажаним у рамках даного проєкту.

DirectX можна оцінити як найпривабливіший варіант для реалізації рушію візуалізації методом трасування шляхів завдяки можливості залучати спеціалізовані ядра RTX, що вбудовані в обидві моделі доступних в рамках проєкту графічних процесорів. Однак, маючи на меті порівняти швидкодію програми, яка виконує обрахунки на графічному процесорі з аналогічною програмою, виконуваною на центральному процесорі, бажано ізолювати будь-які інші відмінності у процесі візуалізації. Архітектура CUDA призначена для виконання обчислень загального призначення, що дозволяє дуже близько відтворити алгоритм візуалізації, що був використаний для створення рушія візуалізації, який використовував виключно центральний процесор. Таким чином можна буде відокремити вплив архітектури графічного процесору від загальної ефективності певного алгоритму візуалізації.

Вибір CUDA у якості API для комунікації з графічним процесором вирішує питання вибору мови програмування, оскільки розрахований на використання C++.

Середовищем розробки було обрано Visual Studio 2019.

Результатом роботи програми буде зображення у форматі .ppm, який не можна переглянути стандартними засобами Windows, тому, для перегляду результатів рендерингу буде використано веб ресурс, що конвертує зображення .ppm у загальновживаний формат без втрат .png. Доступ до конвертору здійснюється за посиланням: https://www.cs.rhodes.edu/welshc/COMP141_F16/

ppmReader.html.

2.4 Застосування CUDA для розробки програмного модулю

У контексті розробки програми, що залучає ресурси відеокарти центральний процесор прийнято називати хостом, а графічний процесор — девайсом. Для позначення компілятору, який код виконуватиметься на хості, а який на девайсі,

використовуються відповідні префікси __host__ та __device__. При відсутності префікса вважається, що код призначений для хоста.

Приклад оголошення перегрузки оператора з використанням префіксів зображено на рисунку 2.1.

```
__host__ __device__ inline vec3& vec3::operator+=(const vec3& v) {
    e[0] += v.e[0];
    e[1] += v.e[1];
    e[2] += v.e[2];
    return *this;
}
```

Рисунок 2.1 Приклад застосування префіксів __host__ та __device__ при оголошені функції

Ці префікси використовуються під час оголошення змінних та функцій та можуть поєднуватися. У такому випадку буде скомпільовано як функцію для хоста так і для девайсу. Однак, хоч їх функціонал буде ідентичний, хост та девайс не можуть обмінюватися даними за звичайних обставин і спроба комунікації хоста з девайсом призведе до помилки.

Єдиним способом запуску функції для девайсу з процесу хосту є так звані kernel функції. Дані функції оголошується з префіксом __global__. При виклику, функція виконується N разів паралельно на N-ій кількості потоків CUDA. Кількість потоків, вказується за допомогою нового синтаксису <<<...>>>. Кожному потоку, надається унікальний ідентифікатор threadIdx, який доступний у функції через вбудовані змінні.

Приклад оголошення kernel функції наведено на рис. 2.2.

```
__global__ void render_init(int max_x, int max_y, curandState* rand_state) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;

    if ((i >= max_x) || (j >= max_y)) return;

    int pixel_index = j * max_x + i;
    curand_init(pixel_index, 0, 0, &rand_state[pixel_index]);
}
```

Рисунок 2.2 Приклад оголошення kernel

Приклад виклику kernel наведено на рис. 2.3

```
render_init<<<blocks, threads>>>(nx, ny, d_rand_state);
```

Рисунок 2.3 Приклад виклику kernel

Для зручності threadIdx є 3-компонентним вектором, тому потоки можна ідентифікувати за допомогою одновимірного, двовимірного або тривимірного індексу потоку, утворюючи одновимірний, двовимірний або тривимірний блок потоки, які називають блоком потоків.

Кількість потоків на блок є обмеженою апаратною частиною, оскільки всі потоки блоку розташовані на одному потоковому багатопроцесорному ядрі та мають спільно використовувати обмежені ресурси пам’яті цього ядра. Максимальна кількість потоків на ядро залежить від конкретної моделі графічного процесору та може досягати у сучасних відеокартах до 1024 потоків.

Потоки CUDA можуть отримувати доступ до даних із кількох просторів пам’яті під час їх виконання. Кожен потік має приватну локальну пам’ять, а також спільну пам'ять, видиму для всіх потоків блоку (рис. 2.4). Усі потоки мають доступ до однієї глобальної пам’яті.

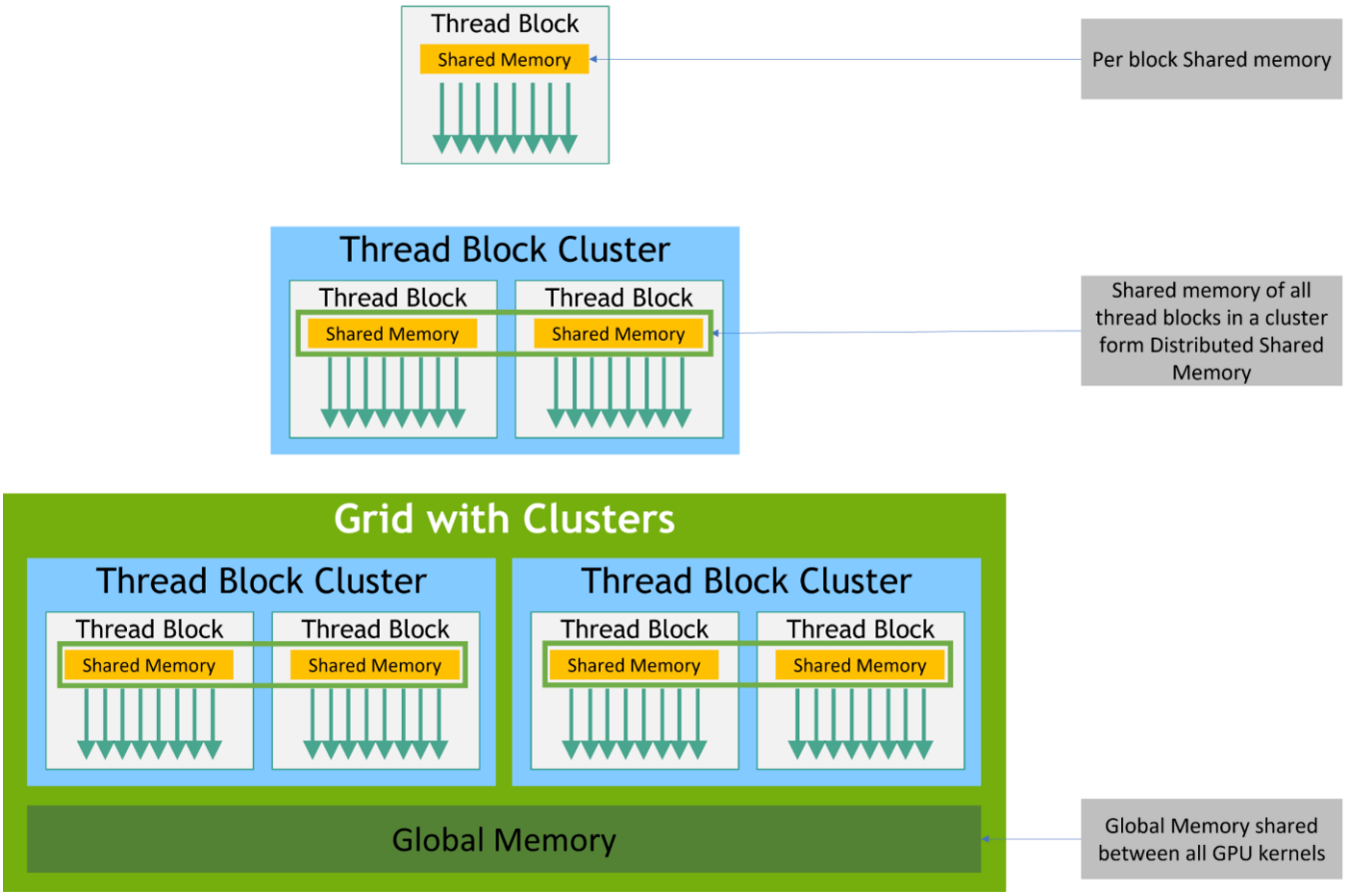


Рисунок 2.4 Схема розподілу пам’яті між потоками (джерело: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#id100)

Для компіляції програми, написаної за допомогою CUDA API потрібен спеціальний компілятор, що входить до встановлюваного SDK.

Вихідні файли, скомпільовані за допомогою nvcc, можуть містити комбінацію коду хоста (тобто коду, який виконується на хості) та коду пристрою (тобто коду, який виконується на пристрої). Основний робочий процес nvcc полягає у відділенні коду пристрою від коду хоста, після чого відбувається компіляція коду пристрою у форму складання (код PTX) та/або двійкову форму (об’єкт cubin), і модифікація коду хоста шляхом заміни синтаксису <<<...>>>, представленого в kernel, на необхідні виклики функцій CUDA для завантаження та запуску кожного скомпільованого kernel з коду PTX і /або об'єкт cubin.

Змінений код хоста виводиться як код C++, який залишається скомпілювати за допомогою стороннього компілятору.

Будь-який код PTX, завантажений програмою під час виконання, потім компілюється у двійковий код драйвером пристрою. Це називається «своєчасною компіляцією». «Своєчасна компіляція» збільшує час завантаження програми, але дозволяє програмі отримати вигоду від будь-яких нових удосконалень компілятора, які надходять із кожним оновленням драйвером пристрою. Це також єдиний спосіб запуску програм на пристроях, які не існували на момент компіляції програми.

Двійковий код залежить від архітектури. Об’єкт cubin генерується за допомогою параметра компілятора `-code`, який визначає цільову архітектуру: наприклад, компіляція з `-code=sm_80` створює двійковий код для пристроїв з обчислювальними здатностями 8.0. Бінарна сумісність гарантується від однієї незначної редакції до наступної, але не від однієї незначної версії до попередньої або між основними версіями. Іншими словами, об’єкт cubin, створений для обчислювальних здатностей X.y, виконуватиметься лише на пристроях з обчислювальними здатностями X.z, де z≥y.

У наведеній нижче таблиці (таблиця. 2.1) наведено назви поточних архітектур графічних процесорів та обчислювальні здатності, які вони надають. Існують інші відмінності, такі як кількість кластерів реєстрів і процесорів, які впливають лише на продуктивність виконання.

Код обчислювальної здатності	Назва архітектури
sm_50, sm_52, sm_53	Maxwell
sm_60, sm_61, sm_62	Pascal
sm_70, sm_72	Volta
sm_75	Turing
sm_80, sm_86, sm_87	Ampere
sm_89	Ada
sm_90, sm_90a	Hopper

Таблиця. 2.1. Перелік актуальних архітектур відеокарт NVIDIA

Обчислювальні здатності пристрою представлені номером версії, який також іноді називають «версією SM». Цей номер версії визначає функції, які підтримуються

апаратним забезпеченням графічного процесора, і використовується програмами під час виконання, щоб визначити, які апаратні функції та/або інструкції доступні на поточному графічному процесорі.

Щоб виконати код на пристроях із певними обчислювальними здатностями, програма повинна завантажити двійковий або PTX-код, який сумісний із цією обчислювальною здатністю. Зокрема, щоб мати можливість виконувати код на майбутніх архітектурах з вищими обчислювальними можливостями (для яких ще не можна створити двійковий код), програма повинна завантажити код PTX, який буде скомпільовано під час запуску.

РОЗДІЛ 3. ДОСЛІДЖЕННЯ РЕЗУЛЬТАТІВ РОБОТИ ПРОГРАМИ

3.1 Огляд розробленого програмного модулю

Розроблена програма є консольним додатком та не має способу взаємодії з користувачем, оскільки увага проєкту зосереджена саме на процесі виконання процесу візуалізації за рахунок графічного процесору. Задання вхідних даних виконується всередині коду програми за рахунок середовища програмування.

Вікно програми продемонстровано на рис. 3.1.

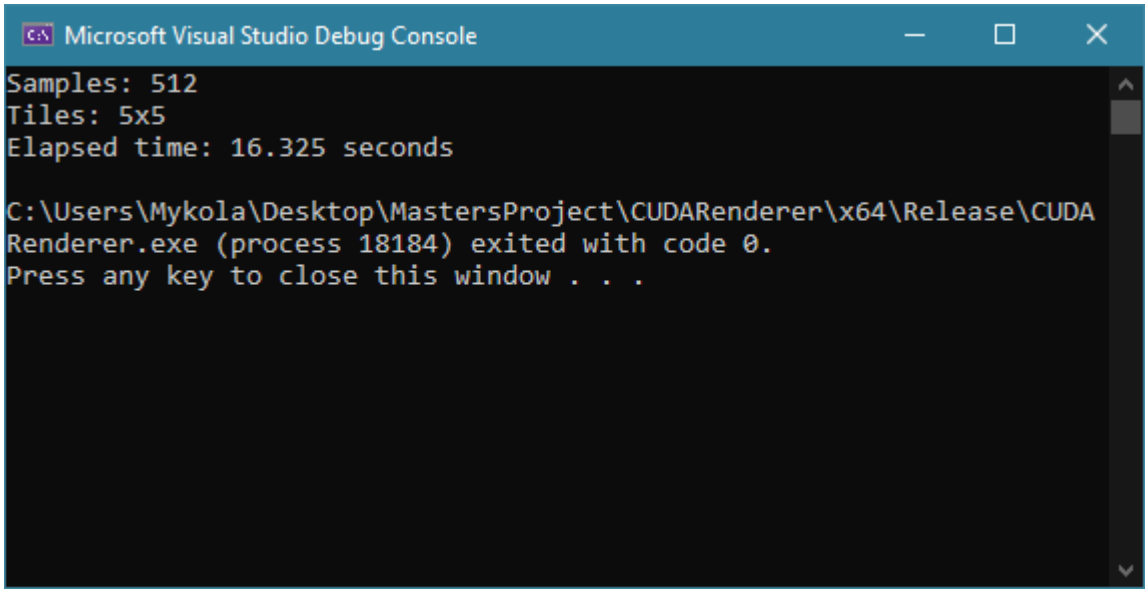


Рисунок 3.1 Вікно програми

Як зазначено у пункті 2.3, результатом роботи програми є зображення у форматі PPM. Для його перегляду використовується веб сервіс.

Зображення, згенероване, за допомогою розробленого рушію представлене на рис. 3.2.

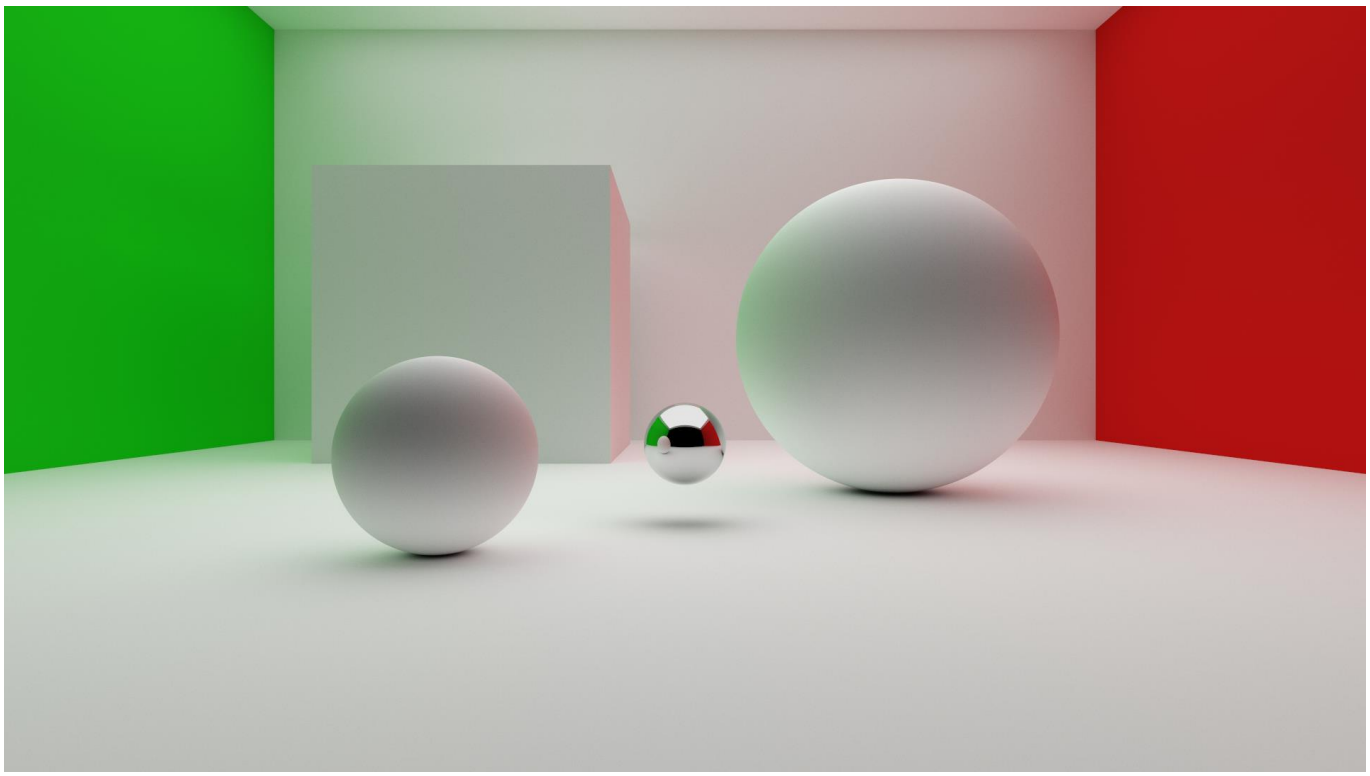


Рисунок 3.2 Результат роботи програми

3.2 Огляд умов тестування

Сцена задається у тілі програми у вигляді списку геометричних примітивів — сфер та прямокутників. На меті було відтворити умови, подібні до освітлення в кімнаті з одним джерелом світла на стелі. Біла кімната забезпечує нейтральний фон, на якому, у поєднанні з кольоровими стінами, можна оцінити пряме та не пряме освітлення. Матові та глянцеві поверхні додають різноманіття та створюють різні умови для прорахунку траєкторії відбитого світла. Опис наведеної вище сцени у коді програми представлено на рис. 3.3.

```
// Materials
auto material_grey = new lambertian(vec3(0.99, 0.99, 0.99));
auto material_emissive = new diffuse_light(vec3(1.0, 1.0, 1.0));
auto material_green = new lambertian(vec3(0.01, 0.8, 0.01));
auto material_red = new lambertian(vec3(0.8, 0.01, 0.01));
auto material_chrome = new metal(vec3(0.8, 0.8, 0.8), 0.0);

int i = 0;

// Light source
d_list[i++] = new xz_rect(-7, 7, -15, -1, 6.999, material_emissive);

// Walls of the room
d_list[i++] = new xz_rect(-8, 8, -16, 0, 7, material_grey);
d_list[i++] = new xz_rect(-8, 8, -16, 0, -1, material_grey);
d_list[i++] = new yz_rect(-1, 7, -16, 0, -8, material_green);
d_list[i++] = new yz_rect(-1, 7, -16, 0, 8, material_red);
d_list[i++] = new xy_rect(-8, 8, -1, 7, -16, material_grey);

// Cube
d_list[i++] = new xz_rect(-5, -1, -15, -11, 3, material_grey);
d_list[i++] = new yz_rect(-1, 3, -15, -11, -5, material_grey);
d_list[i++] = new yz_rect(-1, 3, -15, -11, -1, material_grey);
d_list[i++] = new xy_rect(-5, -1, -1, 3, -11, material_grey);
d_list[i++] = new xy_rect(-5, -1, -1, 3, -15, material_grey);

// Spheres
d_list[i++] = new sphere(vec3(2.0, 0.5, -8), 1.5, material_grey);
d_list[i++] = new sphere(vec3(-1.5, -0.4, -5.0), 0.6, material_grey);
d_list[i++] = new sphere(vec3(0.0, -0.4, -6.0), 0.3, material_chrome);
```

Рисунок 3.3 Задання сцени у коді програми

Сцени, в яких присутнє лише одне, відносно невелике, джерело світла, створюють для рушіїв візуалізації методом трасування шляхів одні з найскладніших умов освітлення, оскільки вірогідність того, що промінь, випущений з камери так і не досягне джерела світла є досить великою. Такі сцени схильні до генерації високого рівня шумів. Щоб це компенсувати необхідно мати досить високий рівень ліміта для кількості відбиттів променю та великий розмір вибірки (sample rate). Відповідно, збільшення обсягу обрахунків спричиняє пропорційне зростання часу візуалізації.

Складність умов даної сцени робить її хорошими тестовими умовами для дослідження швидкодіїї розробленої програми та порівняння показників з аналогом, що виконує усі розрахунки виключно на центральному процесорі комп’ютера.

Зображення, приведене на рис. 3.2 було візуалізовано з 4096 променями на піксель та обмеженням у максимум 12 відбиттів від поверхонь. Розмір зображення — 1920 на 1080 пікселів. Візуалізація з такими параметрами зайняла 138 секунд.

Щоб осягнути швидкодію нового рушію візуалізації зазначу, що, для рендерингу подібної сцени програмою, розробленою у ході виконання дипломного проєкту бакалавра, знадобилося 4 години 11 хвилин та 58 секунд, при тому, що налаштування візуалізації мали 2048 променів на піксель та максимум 7 відбиттів.

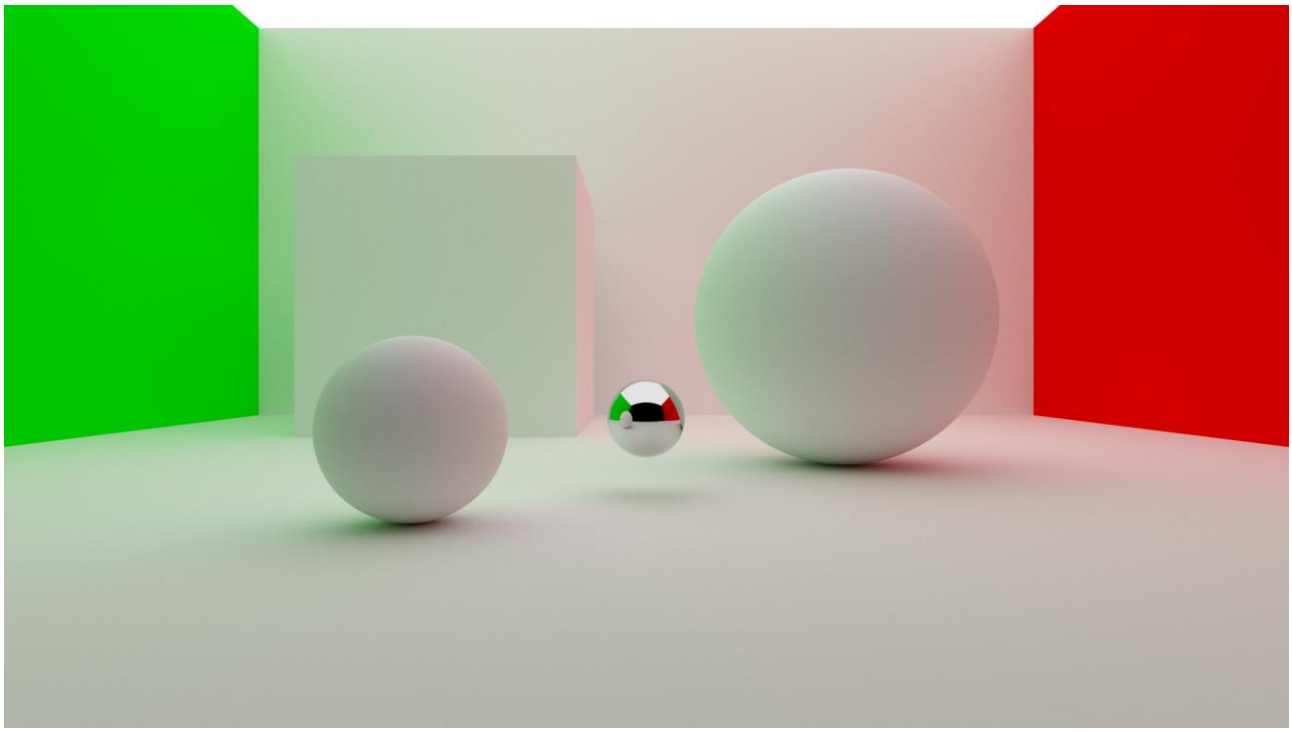


Рисунок 3.4 Рендер, отриманий під час виконання дипломної роботи бакалавра

Більш детальне порівняння швидкодії програм буде проведено у розділі 3.3 цього звіту. Перед цим важливо дослідити фактори, що впливають на швидкість виконання обрахунків на графічному процесорі та оптимізувати налаштування для досягнення максимальної швидкодії. Для стандартизації досліджень, усі експерименти будуть проводитися за допомогою сцени, зображеної на рис. 3.3. Однак, використання високих значень кількості променів на піксель призводить до

непомірно високого часу виконання за допомогою процесора. Тому, з точки зору практичності та економії часу, усі подальші дослідження будуть виконуватися з наступними параметрами візуалізації (якщо в самому експерименті не зазначено інакше):

Розмір зображення: 1920 × 1080.

Кількість відбиттів: 12.

Кількість променів на піксель зображення: 512.

Оскільки час обрахунку прямопропорційно збільшується зі збільшенням обрахованих променей, результати цих експериментів можна екстраполювати для будь-якої кількості променів з погрішністю близько 1-2 секунд.

3.3 Дослідження впливу розподілу потоків на швидкодію графічного процесора

Як зазначалося у пункті 2.4 цього звіту, потоки відеокарти розділяються на блоки, які поділяють між собою пам’ять. Кількість та розмір блоків може істотно впливати на швидкість виконання задачі.

У розробленій програмі, зображення розбивається на секції по $n \times m$ пікселів, кожна з яких виконується індивідуальним блоком. Хоч загальна кількість потоків залишається незмінною та залежить від кількості пікселів у зображенні, кількість блоків контролюється розміром секцій зображення і може бути зміненою у налаштуваннях програми.

Розміри секторів можуть бути довільними, з будь-яким співвідношення сторін та, в ідеалі, вміщуються у зображення цілу кількість разів (розділяють зображення на рівну кількість частин, без секторів, що виходять за межі зображення). Однак, потоки для пікселей, що знаходяться за межами зображення, миттєво зупиняються та не несуть жодного додаткового навантаження на програму, тому для проведення експериментів було використано квадратні сектори розміром 2x2 пікселі, 3x3, 4x4, 5x5, 6x6, 8x8, 16x16 та 32x32. Графічний процесор даної моделі не підтримує більше 1024 потоків в одному блоці, що зумовлено архітектурою процесору, тому, 32x32 пікселі є максимальним розміром сектору. Дана вибірка експериментів дозволить оцінити динаміку зміни швидкодії залежно від розміру блоків потоків та визначити,

який розмір сектору є оптимальним для графічного процесору, на якому виконуються експерименти.

Результати дослідження:

Розмір секції 2x2 (4 потоків на блок): 34 секунди.

Розмір секції 3x3 (9 потоків на блок): 24 секунди.

Розмір секції 4x4 (16 потоків на блок): 18 секунд.

Розмір секції 5x5 (25 потоків на блок): 16 секунд.

Розмір секції 6x6 (36 потоків на блок): 26 секунди.

Розмір секції 8x8 (64 потоків на блок): 28 секунди.

Розмір секції 16x16 (256 потоків на блок): 32 секунди.

Розмір секції 32x32 (1024 потоків на блок): 27 секунди.

Експеримент був проведений не менше 3 разів для кожного з випадків, після чого було обраховане середнє арифметичне значення. Результати дослідження представлені у вигляді графіку на рис. 3.5:

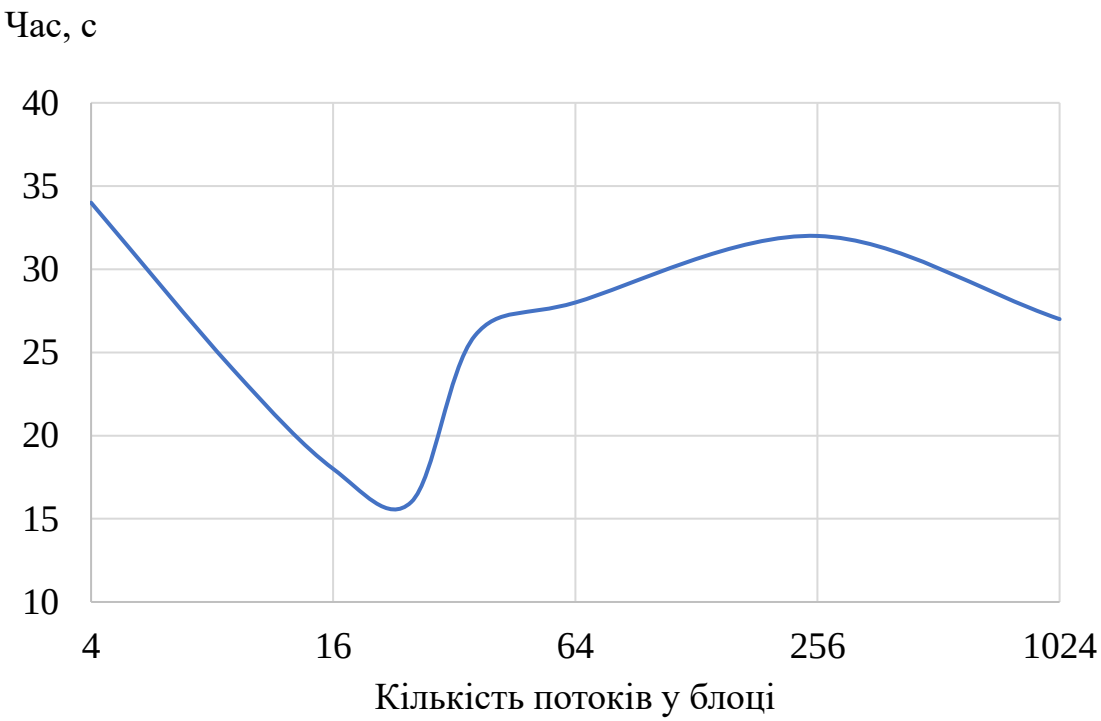


Рис. 3.5 Графік залежності часу рендерингу від кількості потоків у блоці

Як очікувалося, кількість потоків у блоці має великий вплив на швидкість виконання обчислень. Залежно від налаштувань, можна досягти збільшення

швидкодії програми майже в два рази. Експериментальним шляхом було визначено, що розділення зображення на сектори 5x5 пікселів є найбільш оптимальним для графічного процесору, на якому виконувалися тести.

3.4 Порівняння швидкодії програми при виконанні на графічних процесорах різних моделей

Швидкодія програми істотно відрізняється залежно від характеристик апаратного забезпечення. Найбільшими факторами для цього будуть архітектура процесору та кількість ядер. Також великим фактором може бути обсяг та швидкість відеопам’яті. Нажаль, випробувати параметри графічних процесорів, що пов’язані з пам’яттю не вийде, оскільки тестова сцена не має у собі такого обсягу даних, щоб повністю заповнити доступний обсяг пам’яті — візуалізація обраної тестової сцени займає близько 300 мегабайт відеопам’яті, у той час як обидва графічних процесори, доступні для проведення тестів мають 8 гігабайт вбудованої відеопам’яті. Під час візуалізації найбільше пам’яті займають текстури та високополігональні моделі, які даний програмний модуль не здатний візуалізувати.

Для проведення цього дослідження було використано два графічних процесори: RTX 3070 Ti та RTX 3050 Ti компанії NVIDIA. Оскільки дані відеокарти належать до одного покоління архітектури, швидкодія їх ядер, в теорії, має бути однаковою, однак, різниця, яка нас цікавить полягає у кількості CUDA ядер.

На сайті виробника зазначені такі характеристики:

RTX 3070 Ti:

6144 CUDA ядер.

Максимальна частота процесору: 1,77 ГГц.

Об’єм відеопам’яті: 8 Гб.

RTX 3050 Ti:

2560 CUDA ядер.

Максимальна частота процесору: 1,78 ГГц.

Об’єм відеопам’яті: 8 Гб.

Виходячи з даних характеристик можна очікувати, що виконання програми буде приблизно у два з половиною довшим на RTX 3050 Ti, оскільки цей графічний процесор має у 2,4 рази менше ядер, які використовуються даною програмою.

Теоретично, оптимальний розподіл потоків на блоки може відрізнятися у цих двох моделей. Щоб порівняти швидкодію відеокарт за оптимальних для них умов, важливо провести тест з різними параметрами розподілу зображення на сектори, аналогічно до досліджень, проведених для RTX 3070 Ti у пункті 3.3 цього звіту, та порівнювати кращі результати.

Результати дослідження:

Розмір секції 2x2 (4 потоків на блок): 82 секунди.

Розмір секції 3x3 (9 потоків на блок): 54 секунди.

Розмір секції 4x4 (16 потоків на блок): 38 секунд.

Розмір секції 5x5 (25 потоків на блок): 37 секунд.

Розмір секції 6x6 (36 потоків на блок): 63 секунди.

Розмір секції 8x8 (64 потоків на блок): 71 секунди.

Розмір секції 16x16 (256 потоків на блок): 89 секунди.

Порівняння швидкодії графічних процесорів у вигляді зіставлення графіків часу візуалізації наведено на рис. 3.6.

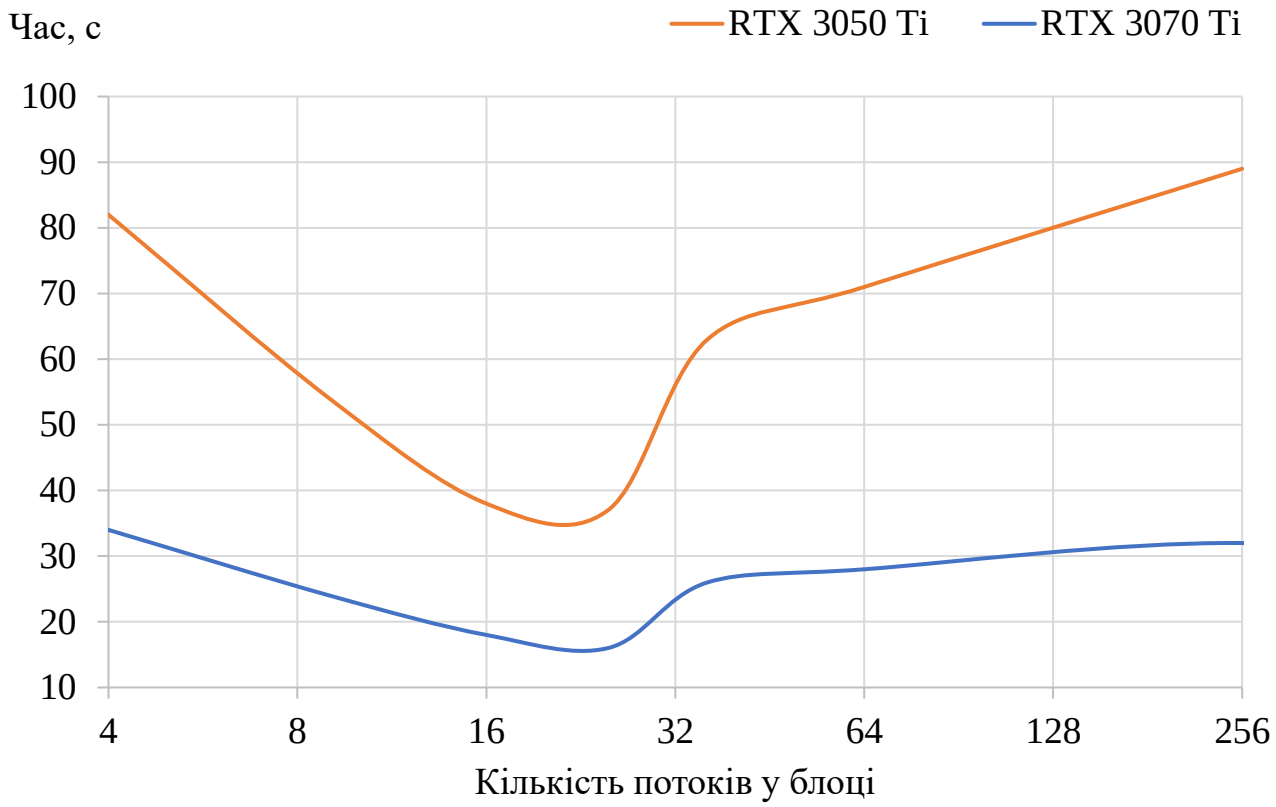


Рис. 3.6 Графік порівняння швидкодії графічних процесорів

Теорія щодо різниці у швидкості справдилася — найкращі результати графічних процесорів відрізняються у 2,3 рази. Дещо несподіваним стало те, що обидві відеокарти мають однакові розміри оптимальних блоків потоків та повторюють графік залежності часу візуалізації від розміру блоків. Це може бути пов’язане з тим, що графічні процесори належать до одного покоління архітектури, або з тим, що вони мають однаковий об’єм відеопам’яті, оскільки кількість блоків безпосередньо впливає на розподіл пам’яті, а потоки в середині блоку поділяють пам’ять між собою.

Дані відеокарти належать до архітектури Ampere, яка є передостанньою на даний момент. Найбільш сучасною архітектурою процесорів NVIDIA є Ada Lovelace, відеокарти якої вміщають в себе більше ядер та з більшими обчислювальними потужностями.

Згідно з документацією на сайті виробника, один мультипроцесор відеокарти останнього покоління архітектури містить:

128 ядер для розрахунків з плаваючою комою одинарної точності,

64 ядра для розрахунків з плаваючою комою подвійної точності,

64 ядра для цілочисельних операцій,

256 Кбайт пам’яті

та інші ядра, що не мають застосування в межах теми даного проєкту.

Для порівняння, попередні два покоління мають такі характеристики:

64 ядер для розрахунків з плаваючою комою одинарної точності,

32 ядра для розрахунків з плаваючою комою подвійної точності,

64 ядра для цілочисельних операцій

Та 192 Кбайти пам’яті.

Через це, показники швидкодії та оптимального розміру блоку потоків можуть варіюватися залежно від покоління архітектури.

Для проведення більш детального порівняльного аналізу графічних процесорів та впливу їх характеристик на швидкість виконання програми було б бажано провести аналогічні тести на інших графічних процесорах цього та інших сімейств архітектури. Однак, під час виконання даного проєкту не вдалося отримати змогу провести дослідження на будь-яких інших графічних процесорах.

3.4 Порівняння швидкодії програми при виконанні обчислень на графічному процесорі та центральному процесорі

У ході виконання дипломного проєкту бакалавру, мною було розроблено програмний модуль візуалізації, що використовує той самий алгоритм візуалізації комп’ютерної графіки, що і в програмі, розробленій у цьому проєкті. Програми мають не значні відмінності у функціоналі, такі, як поява можливості змінювати положення камери у просторі, глибина різкості, та деякі менші покращення у новій програмі. Головна їх відмінність — це те, що попередня програма виконувала всі обчислення виключно за рахунок центрального процесору.

Для прискорення процесу рендерингу, версія програми для центрального процесору також використовує багатопоточність. На відміну від версії програми для графічного процесору, у цій програми кожен потік займається обчисленням не одного пікселя, а цілого рядка. Оскільки центральний процесор має на кілька порядків менше

ядер аніж відеокарта, тільки кілька рядків пікселей обраховується одночасно. Комп'ютер, на якому проводилося тестування, обладнаний процесором Intel core i5 6600K, який має 4 ядра та підтримує максимум 4 потоки одночасно. Блок-схему алгоритм розподілу обчислень на потоки зображено на рис. 3.7.

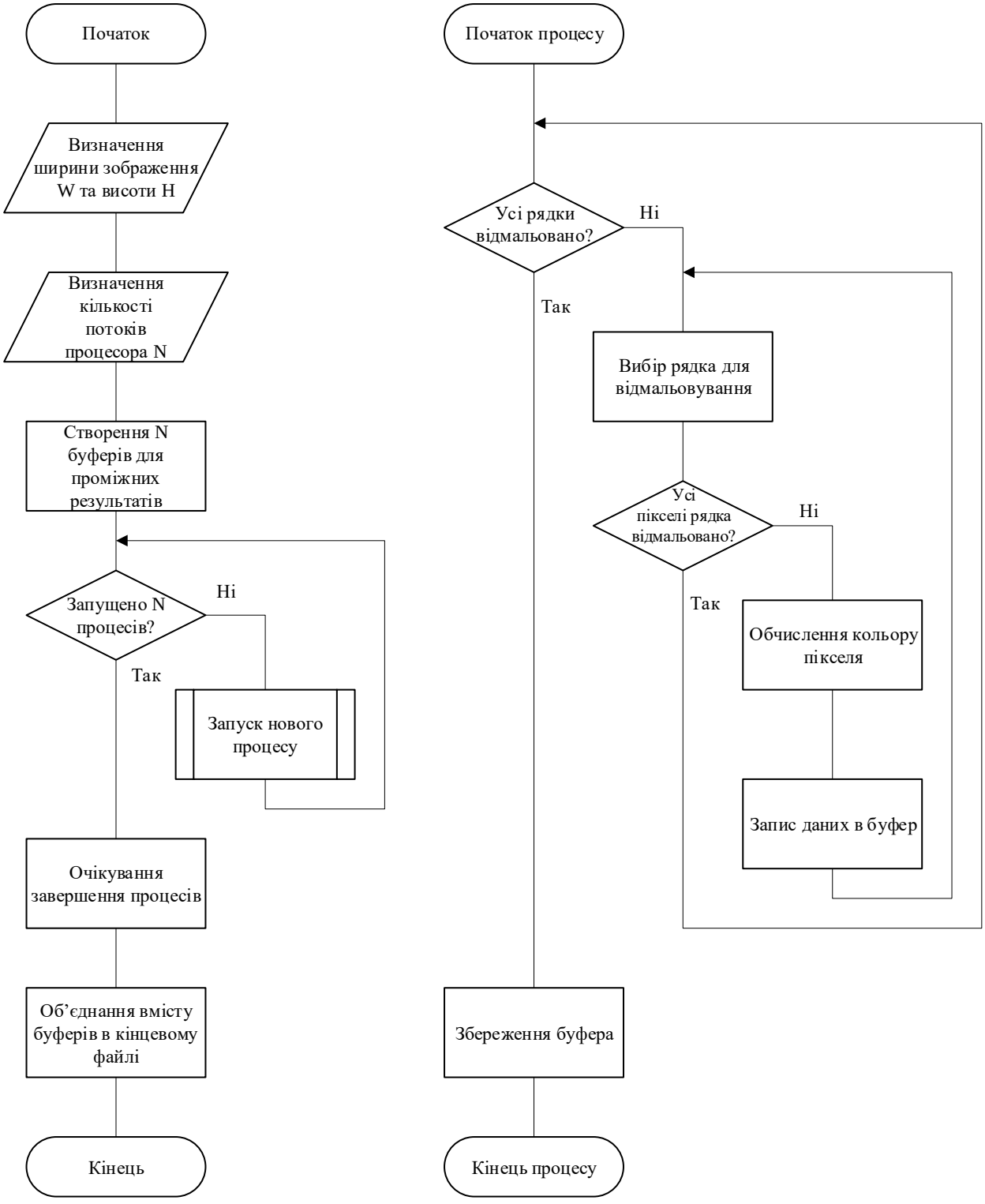


Рис. 3.7 Блок-схема алгоритму розподілу роботи на потоки

Для ізоляції експерименту від усіх можливих факторів впливу на швидкість рендерингу важливо переконатися, що програми створюють ідентичні зображення. Для цього слід відтворити тестову сцену у програмі для обрахунку на центральному процесорі. Задання сцени у програмі для центрального процесору наведено на рис. 3.7.

```
// Scene
objects_list scene;
auto material_grey = make_shared<lambertian>(color(0.99, 0.99, 0.99));
auto material_emissive= make_shared<diffuse_light>(color(1, 1, 1));
auto material_green = make_shared<lambertian>(color(0.01, 0.8, 0.01));
auto material_red = make_shared<lambertian>(color(0.8, 0.01, 0.01));
auto material_chrome = make_shared<metal>(color(0.8, 0.8, 0.8));

scene.add(make_shared<xz_rect>(-7, 7, -15, -1, 6.999, material_emissive));
scene.add(make_shared<xz_rect>(-8, 8, -16, 0, 7, material_grey));
scene.add(make_shared<xz_rect>(-8, 8, -16, 0, -1, material_grey));
scene.add(make_shared<yz_rect>(-1, 7, -16, 0, -8, material_green));
scene.add(make_shared<yz_rect>(-1, 7, -16, 0, 8, material_red));
scene.add(make_shared<xy_rect>(-8, 8, -1, 7, -16, material_grey));

scene.add(make_shared<xz_rect>(-5, -1, -15, -11, 3, material_grey));
scene.add(make_shared<yz_rect>(-1, 3, -15, -11, -5, material_grey));
scene.add(make_shared<yz_rect>(-1, 3, -15, -11, -1, material_grey));
scene.add(make_shared<xy_rect>(-5, -1, -1, 3, -11, material_grey));
scene.add(make_shared<xy_rect>(-5, -1, -1, 3, -15, material_grey));

scene.add(make_shared<sphere>(point3(2.0, 0.5, -8), 1.5, material_grey));
scene.add(make_shared<sphere>(point3(-1.5, -0.4, -5.0), 0.6, material_grey));
scene.add(make_shared<sphere>(point3(0.0, -0.4, -6.0), 0.3, material_chrome));
```

Рис. 3.8 Опис сцени всередині програми для ЦП

Порівняємо вихідне зображення обох програм (рис. 3.8).

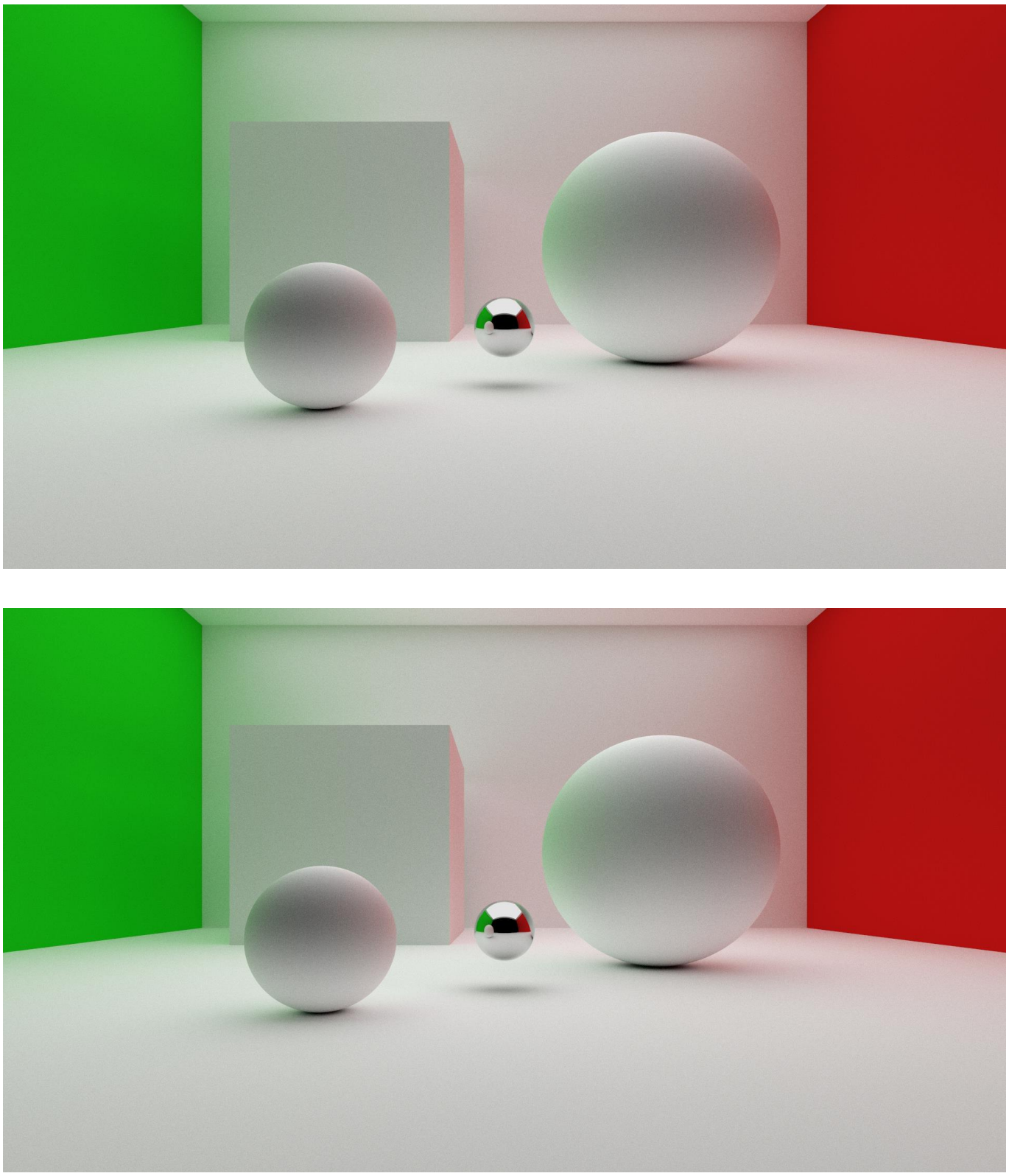


Рис. 3.9 Зображення, отримане програмою для ЦП (зверху), та зображення, отримане програмою для відеокарти (знизу)

Результати роботи програм є повністю ідентичними, що означає, що зміна імплементації методу візуалізації не призвела до спотворень в алгоритмі та будь-які відмінності в роботі програми зумовлені швидкістю роботи процесорів.

Дані зображення були візуалізовані так само як і в попередніх дослідах природмірах зображення 1920 на 1080 пікселей, 512 променів на піксель та максимум 12 відбиттів. При таких налаштуваннях, рендер за допомогою ресурсів центрального процесору було завершено за 6 хвилин 24 секунди. Нагадаю, що при таких самих налаштуваннях, на графічному процесорі вдалося досягти часу рендерингу у 16 секунд, що складає прискорення рівно у 24 рази.

ВИСНОВКИ

Результатом виконання цієї дипломної роботи є програмний модуль візуалізації, який, за допомогою використання апаратного забезпечення графічного процесору, дозволяє скоротити час візуалізації у 24 рази. Проведені експерименти дозволили визначити, які фактори впливають на швидкодію програми та дозволили виконати оптимізацію модулю для досягнення високої ефективності та короткого часу візуалізації.

З метою розробки високоефективної програми було виявлено головні складнощі методу трасування шляхів та визначено спосіб вирішення проблеми за рахунок перенесення обрахунків на графічний процесор. Проведено дослідження специфіки апаратного забезпечення графічних процесорів та, способи використання унікальних можливостей, що вони надають для пришвидшення розрахунків. Визначено перспективні технології, такі як зниження шуму та апскейлінг, які стають можливими завдяки відеокартам.

Ознайомившись з доступними засобами залучення ресурсів графічного процесору та визначившись з необхідним функціоналом для створення програмного модулю було обрано CUDA SDK. Після вивчення особливостей роботи з даним SDK було розроблено експериментальний модуль візуалізації для проведення практичних дослідів та подальшої оптимізації рушію.

Функціонал програми включає в себе візуалізацію простих геометричних фігур — куль та прямокутників. Доступні матеріали включають метал та діелектрик з можливістю регуляції шорсткості поверхні та колір, а також матеріал, що випромінює світло. Рушій візуалізації має кілька параметрів, які впливають на швидкість рендерингу та якість вихідного зображення: розмір зображення, кількість обрахованих променей для кожного пікселя, максимальна кількість відбиттів пікселя, розмір блоків візуалізації.

За допомогою реалізованого функціоналу було створено тестову сцену та проведено ряд експериментів для визначення швидкодії програми, виявлення

оптимальних налаштувань візуалізації та порівняно швидкість роботи з різним апаратним забезпеченням.

Розроблений програмний модуль має високі показники швидкодії та має високий потенціал для подальшого розвитку. Застосування спеціалізованих ядер трасування променів може ще більше прискорити виконання візуалізації та в поєднані з методами зменшення шумів та апскейлінгу може дозволити виконувати візуалізацію у реальному часі. Також, можливість завантаження 3D моделей та імплементація BVH може дозволити розвивати програму у напрямку фотореалістичного рушію візуалізації для таких цілей як рендеринг інтер’єрів, архітектури або спецефектів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. James T. Kajya The rendering equation. *California Institute of Technology. Pasadena.* 1986. 8 p.
2. Eric Veach Robust monte carlo methods for light transport simulation. Stanford University. 1997. 406 p.
3. Jacopo Pantaleoni Online path sampling control with progressive spatio-temporal filtering. *SN Computer Science*. 2020. 12p.
4. Benedikt Bitterli, Chris Wyman, Matt Pharr, Peter Shirley, Aaron Lefohn, Wojciech Jarosz Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. *ACM Transactions on Graphics*. 2020. 17 p.
5. Frederik W. Jansen, Arjan J. F. Kok, Theo Verelst Hardware Challenges for Ray Tracing and Radiosity Algorithms. *Eurographics Workshop on Graphics Hardware 1992.* 1992. P. 123 – 134.
6. Hangyu Zhang, Beibei Wang World-Space Spatiotemporal Path Resampling for Path Tracing. *Computer Graphics Forum*. 2023. 4 p.
7. Chenxi Qu Research and analysis of ray tracing methods. *International Conference on Software Engineering and Machine Learning*. 2023. 9 p.
8. Yan R., Huang L., Guo H., Lü Y., Yang L., Xiao N., Wang Y., Shen L., Lan M. RT Engine: An Efficient Hardware Architecture for Ray Tracing. *Applied Science*. 2022. 14 p.
9. An Overview of the Ray-Tracing Rendering Technique. URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview/ray-tracing-rendering-technique-overview.html
10. Daniel Rueckert Lecture 11 and 12: Ray tracing. 2002. URL: https://www.doc.ic.ac.uk/~dfg/graphics/graphics2008/GraphicsLecture09.pdf
11. Acceleration of ray tracing method using predictive evaluation and GPGPU technology. *Central European Journal of Computer Science*. 2014. P 118 – 126.

12. Ray tracing overview. URL: https://cseweb.ucsd.edu/~sht005/ray_tracing_info.html
13. David J. Eck Introduction to Computer Graphics. 2015. P. 331 – 339.
14. Path Tracing: What is it and How Does it Work? URL: https://history-computer.com/path-tracing-what-is-it-and-how-does-it-work/
15. Matt Pharr, Wenzel Jakob, and Greg Humphreys Physically Based Rendering: From Theory To Implementation. 2021. 1270 p.
16. Wenming Yang, Xuechen Zhang, Yapeng Tian, Wei Wang, Jing-Hao Xue, Qingmin Liao Deep Learning for Single Image Super-Resolution: A Brief Review. *IEEE Transactions on Multimedia Vol. 21, No12*. 2019. P 3106 – 3121.
17. Daniel Glasner, Shai Bagon, Michal Irani Super-Resolution from a Single Image. *IEEE International Conference on Computer Vision*. 2009. 8 p.
18. Christian Ledig, Lucas Theis, Ferenc Huszar, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, Wenzhe Shi Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network. *2017 IEEE Conference on Computer Vision and Pattern Recognition*. 2017. P. 105 – 114.
19. Chao D., Chen L., Kaiming He, Xiaoou T. Image Super-Resolution Using Deep Convolutional Networks. *Transactions on Pattern Analysis and Machine Intelligence Vol. 38, No 2*. 14 p.
20. Chakravarty R. Alla Chaitanya, Anton Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, Timo Aila Interactive Reconstruction of Monte Carlo Image Sequences using a Recurrent Denoising Autoencoder. *SIGGRAPH 2017*. 2017. 12 p.
21. Alain Galvan Ray Tracing Denoising. 2020. URL: https://alain.xyz/blog/ray-tracing-denoising. 2020.
22. Jon Hasselgren, Jacob Munkberg, Marco Salvi, Anjul Patney, Aaron LefohnNeural Temporal Adaptive Sampling and Denoising. *Computer Graphics Forum*. 2020. 9 p.

23. Tim Foley Next-Generation Graphics APIs: Similarities and Differences. SIGGRAPH 2015. 2015. 59 p.
24. Alain Galvan A Comparison of Modern Graphics APIs. 2021. URL: https://alain.xyz/blog/comparison-of-modern-graphics-apis
25. Suvoparno Banerjee DirectX 12 and Direct3D 11.3: An overview. 2015. URL: https://techarx.com/directx-12-and-direct3d-11-3-an-overview/
26. Eddy Luten OpenGL Book. 2014. URL: https://openglbook.com/
27. Khronos Group Vulkan Overview. 2016. URL: https://www.khronos.org/assets/uploads/developers/library/overview/vulkan-overview.pdf
28. Nermin Hajdarbegovic A Brief Overview of the Vulkan API. 2015. URL: https://www.toptal.com/api-developers/a-brief-overview-of-vulkan-api
29. CUDA C++ Programming Guide URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
30. NVIDIA CUDA Compiler Driver NVCC. URL: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html

ДОДАТОК А

Лістинг коду файлу main.cu

```
#include <iostream>
#include <time.h>
#include <float.h>
#include <fstream>

#include <curand_kernel.h>
#include <device_launch_parameters.h>

#include "vec3.h"
#include "ray.h"
#include "sphere.h"
#include "plain.h"
#include "hitable.h"
#include "hitable_list.h"
#include "camera.h"
#include "material.h"

__device__ vec3 color(const ray& r, hitable** scene,
curandState* local_rand_state) {
    ray cur_ray = r;
    vec3 cur_attenuation = vec3(1.0, 1.0, 1.0);
    vec3 emition = vec3(0.0, 0.0, 0.0);
    for (int i = 0; i < 12; i++) {
        hit_record rec;

        if ((*scene)->hit(cur_ray, 0.001f, FLT_MAX,
rec)) {
            ray scattered;
            vec3 attenuation;

            if (!rec.mat_ptr->scatter(r, rec,
attenuation, scattered, local_rand_state)) {
                emition = rec.mat_ptr->emitted();
                return emition * cur_attenuation;
            }
            cur_attenuation *= attenuation;
            cur_ray = scattered;
        }
        else {
            return vec3(0.0, 0.0, 0.0);
```

```
        }
    }
    return vec3(0.0, 0.0, 0.0);
}

__global__ void render_init(int max_x, int max_y,
curandState* rand_state) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;

    if ((i >= max_x) || (j >= max_y)) return;

    int pixel_index = j * max_x + i;
    curand_init(pixel_index, 0, 0,
&rand_state[pixel_index]);
}

__global__ void render(vec3* fb, int max_x, int max_y,
int ns, camera** cam, hitable** scene, curandState*
rand_state) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;

    if ((i >= max_x) || (j >= max_y)) return;

    int pixel_index = j * max_x + i;
    curandState local_rand_state =
rand_state[pixel_index];

    vec3 col(0, 0, 0);
    for (int s = 0; s < ns; s++) {
        float u = float(i +
curand_uniform(&local_rand_state)) / float(max_x);
        float v = float(j +
curand_uniform(&local_rand_state)) / float(max_y);
        ray r = (*cam)->get_ray(u, v,
&local_rand_state);
        col += color(r, scene, &local_rand_state);
    }
    rand_state[pixel_index] = local_rand_state;
    col /= float(ns);
    col[0] = sqrt(col[0]);
    col[1] = sqrt(col[1]);
    col[2] = sqrt(col[2]);
    fb[pixel_index] = col;
```

```
}

__global__ void create_scene(hitable** d_list, hitable**
d_scene, camera** d_camera, int nx, int ny) {
    if (threadIdx.x == 0 && blockIdx.x == 0) {

        // Materials
        auto material_grey = new lambertian(vec3(0.99,
0.99, 0.99));
        auto material_emissive = new
diffuse_light(vec3(1.0, 1.0, 1.0));
        auto material_green = new lambertian(vec3(0.01,
0.8, 0.01));
        auto material_red = new lambertian(vec3(0.8,
0.01, 0.01));
        auto material_chrome = new metal(vec3(0.8, 0.8,
0.8), 0.0);

        int i = 0;

        // Light source
        d_list[i++] = new xz_rect(-7, 7, -15, -1, 6.999,
material_emissive);

        // Walls of the room
        d_list[i++] = new xz_rect(-8, 8, -16, 0, 7,
material_grey);
        d_list[i++] = new xz_rect(-8, 8, -16, 0, -1,
material_grey);
        d_list[i++] = new yz_rect(-1, 7, -16, 0, -8,
material_green);
        d_list[i++] = new yz_rect(-1, 7, -16, 0, 8,
material_red);
        d_list[i++] = new xy_rect(-8, 8, -1, 7, -16,
material_grey);

        // Cube
        d_list[i++] = new xz_rect(-5, -1, -15, -11, 3,
material_grey);
        d_list[i++] = new yz_rect(-1, 3, -15, -11, -5,
material_grey);
        d_list[i++] = new yz_rect(-1, 3, -15, -11, -1,
material_grey);
        d_list[i++] = new xy_rect(-5, -1, -1, 3, -11,
material_grey);
```

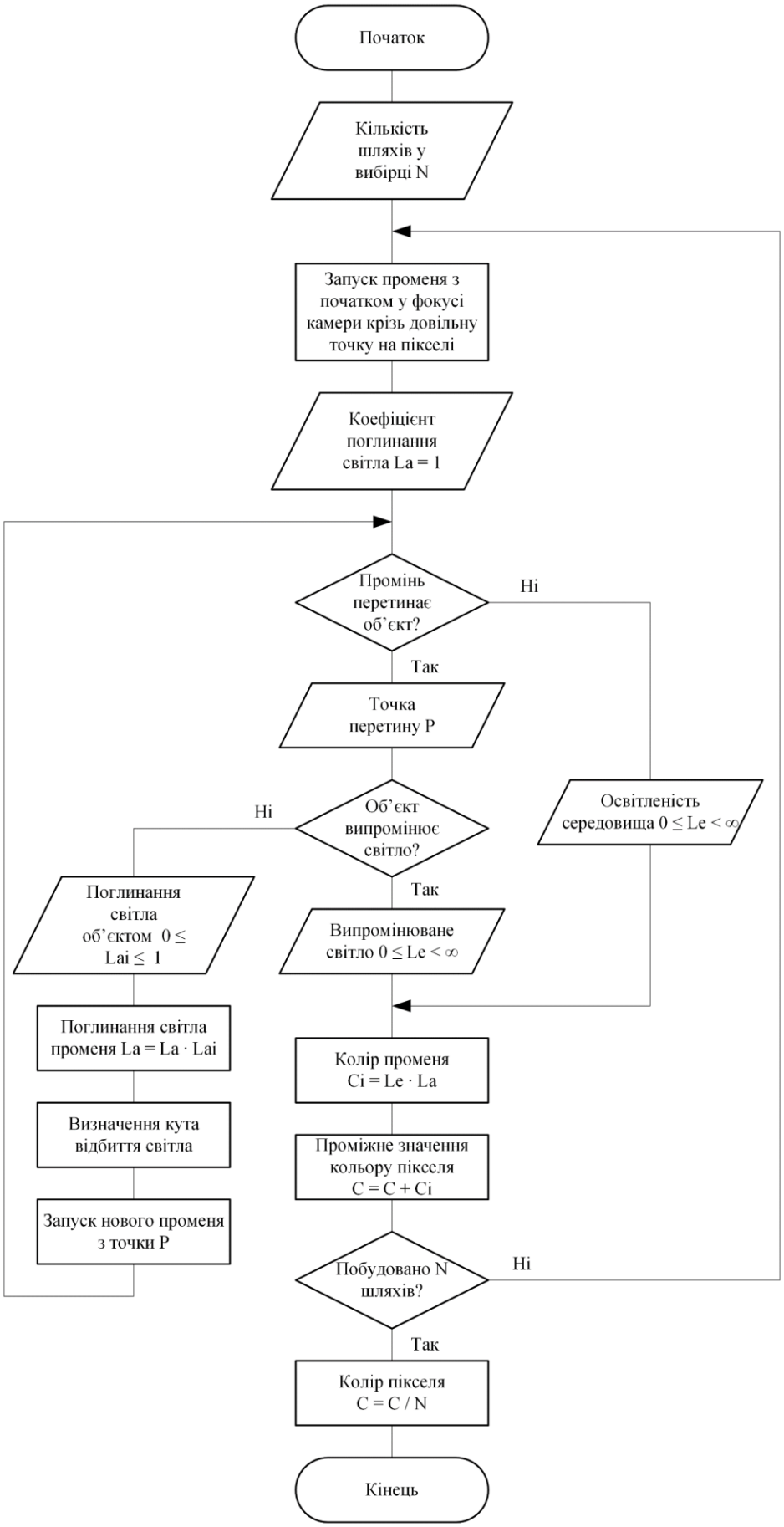
```
        d_list[i++] = new xy_rect(-5, -1, -1, 3, -15,
material_grey);

        // Spheres
        d_list[i++] = new sphere(vec3(2.0, 0.5, -8),
1.5, material_grey);
        d_list[i++] = new sphere(vec3(-1.5, -0.4, -5.0),
0.6, material_grey);
        d_list[i++] = new sphere(vec3(0.0, -0.4, -6.0),
0.3, material_chrome);

        *d_scene = new hitable_list(d_list, i);

        vec3 lookfrom(0.0, 0.0, 0.0);
        vec3 lookat(0.0, 0.0, -1.0);

        float dist_to_focus = (lookfrom -
lookat).length();
        float aperture = 0.1;
        *d_camera = new camera(lookfrom,
            lookat,
            vec3(0, 1, 0),
            50,
            float(nx) / float(ny),
            aperture,
            dist_to_focus);
    }
}

int main() {
    int nx = 1920;
    int ny = 1080;
    int ns = 512;
    int tx = 5;
    int ty = 5;

    std::cout << "Image size: " << nx << "x" << ny <<
std::endl;
    std::cout << "Samples: " << ns << std::endl;
    std::cout << "Tiles: " << tx << "x" << ty <<
std::endl;

    int num_pixels = nx * ny;
    size_t fb_size = num_pixels * sizeof(vec3);
```

```
    vec3* fb;
    cudaMallocManaged((void**)&fb, fb_size);

    curandState* d_rand_state;
    cudaMalloc((void**)&d_rand_state, num_pixels *
sizeof(curandState));

    cudaDeviceSynchronize();

    hitable** d_list;
    int num_hitables = 14;
    cudaMalloc((void**)&d_list, num_hitables *
sizeof(hitable*));

    hitable** d_scene;
    cudaMalloc((void**)&d_scene, sizeof(hitable*));

    camera** d_camera;
    cudaMalloc((void**)&d_camera, sizeof(camera*));

    create_scene<<<1, 1>>>(d_list, d_scene, d_camera,
nx, ny);
    cudaDeviceSynchronize();

    clock_t start, stop;
    start = clock();

    dim3 blocks(nx / tx + 1, ny / ty + 1);
    dim3 threads(tx, ty);

    render_init<<<blocks, threads>>>(nx, ny,
d_rand_state);
    cudaDeviceSynchronize();
    render<<<blocks, threads>>>(fb, nx, ny, ns,
d_camera, d_scene, d_rand_state);
    cudaDeviceSynchronize();
    stop = clock();
    double timer_seconds = ((double)(stop - start)) /
CLOCKS_PER_SEC;
    std::cout << "Elapsed time: " << timer_seconds << "
seconds\n";

    // Output FB as Image
    std::ofstream output("image.ppm");
    output << "P3\n" << nx << " " << ny << "\n255\n";
```

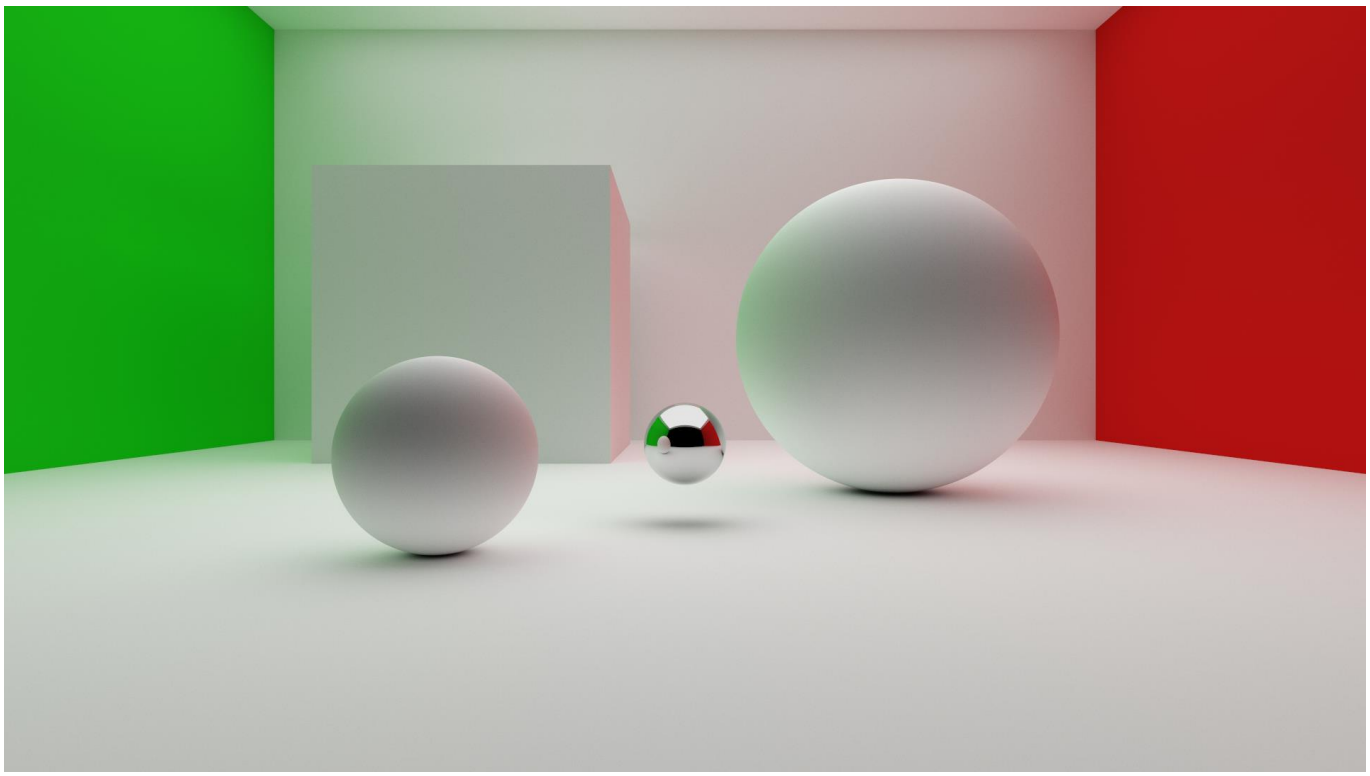
```
    for (int j = ny - 1; j >= 0; j--) {
        for (int i = 0; i < nx; i++) {
            size_t pixel_index = j * nx + i;
            int ir = int(255.99 * fb[pixel_index].r());
            int ig = int(255.99 * fb[pixel_index].g());
            int ib = int(255.99 * fb[pixel_index].b());
            output << ir << " " << ig << " " << ib <<
"\n";
        }
    }
    output.close();

    cudaDeviceSynchronize();
    cudaDeviceReset();
}
```

ДОДАТОК Б

Графічні матеріали



Алгоритм візуалізації методом трасування шляхів



Вихідне зображення програми

Презентація

Державний університет інформаційно-комунікаційних технологій

Кафедра Інженерії програмного забезпечення автоматизованих систем

КВАЛІФАКЦІЙНА РОБОТА

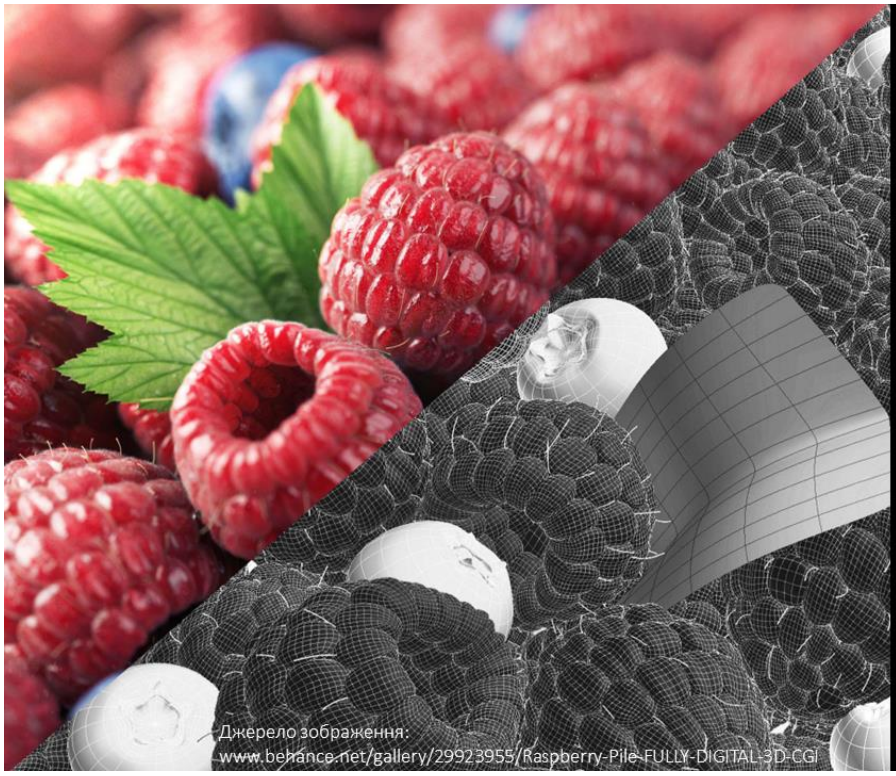
на тему:

Розробка рушію візуалізації з утилізацією ресурсів графічного процесору

Виконавець: Цецюрський Микола Валентинович

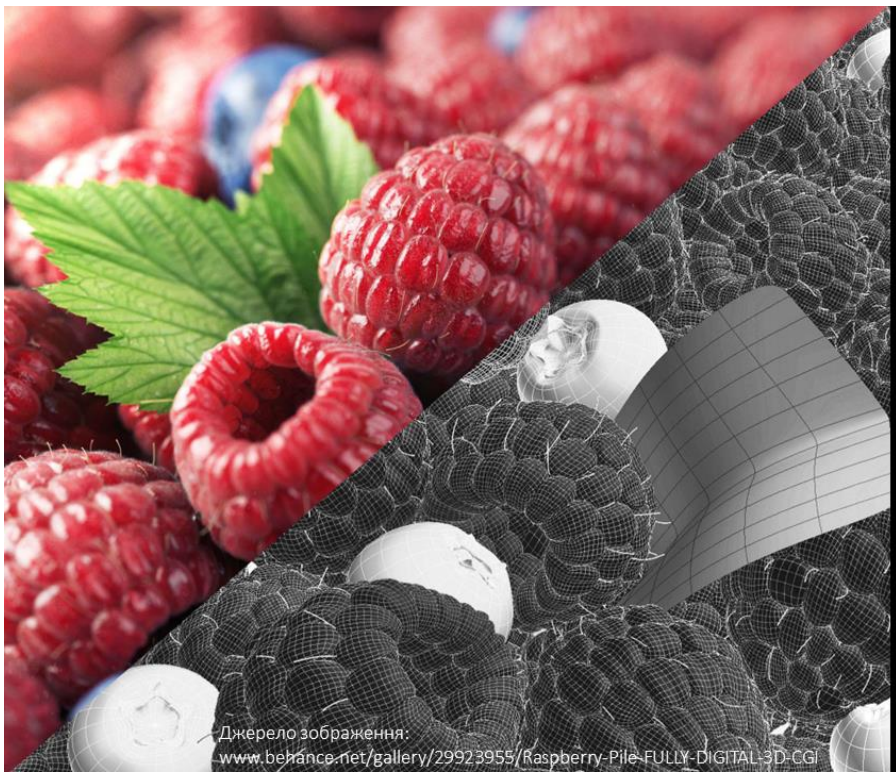
Керівник: Глущенко Юлія Анатоліївна

Київ 2023



Мета проєкту

- Дослідити переваги застосування графічного процесору для візуалізації методом трасування шляхів
- Розробити експериментальний модуль візуалізації із застосуванням ресурсів графічного процесору
- Провести емпіричні дослідження швидкодії розробленого модуля та порівняти з аналогічним рушієм, що проводить обчислення виключно за рахунок центрального процесору



Об’єкт дослідження:

Програмне забезпечення візуалізації тривимірних сцен методом трасування променів.

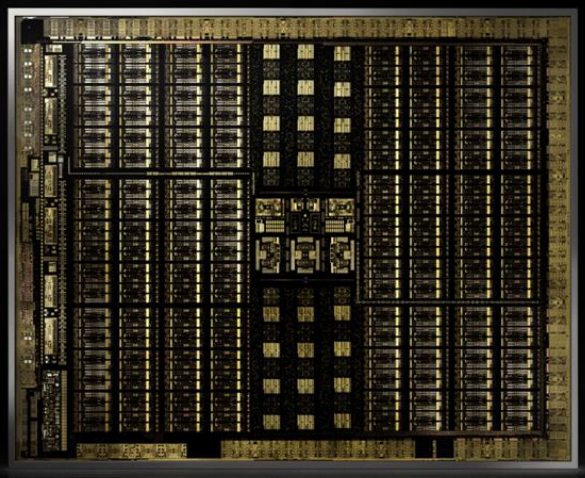
Предмет дослідження:

програмне забезпечення, що виконує обрахунки на графічному процесорі для прискорення процесу візуалізації.

Паралельні обчислення

Візуалізація зображень передбачає багато повторюваних обчислень для кожного пікселя, і завдяки своїй архітектурі, яка складається з тисяч ядер, оптимізованих для паралельної обробки, графічні процесори значно швидше справляються порівняно з послідовним характером обчислень ЦП.

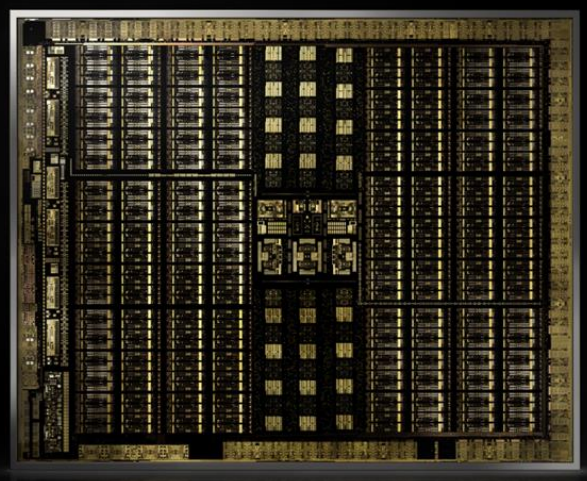
Джерело зображення:
https://www.vsgi.com/2021/09/13/weighing-the-merits-of-lcd-vs-led-video-walls/



Спеціалізована архітектура

Графічні процесори мають велику кількість спеціалізованих ядер для виконання специфічних задач візуалізації.

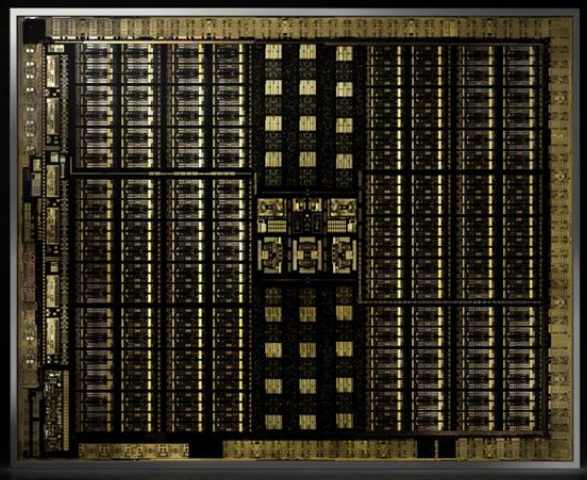
Джерело зображення: https://www.nvidia.com/en-us/design-visualization/technologies/turing-architecture/



Ядра трасування променів

Останні роки почала розвиватися технологія трасування променів та з нею почали інтегрувати спеціалізовані ядра для виконання таких затребуваних операцій як перевірку перетину.

Джерело зображення: https://www.nvidia.com/en-us/design-visualization/technologies/turing-architecture/



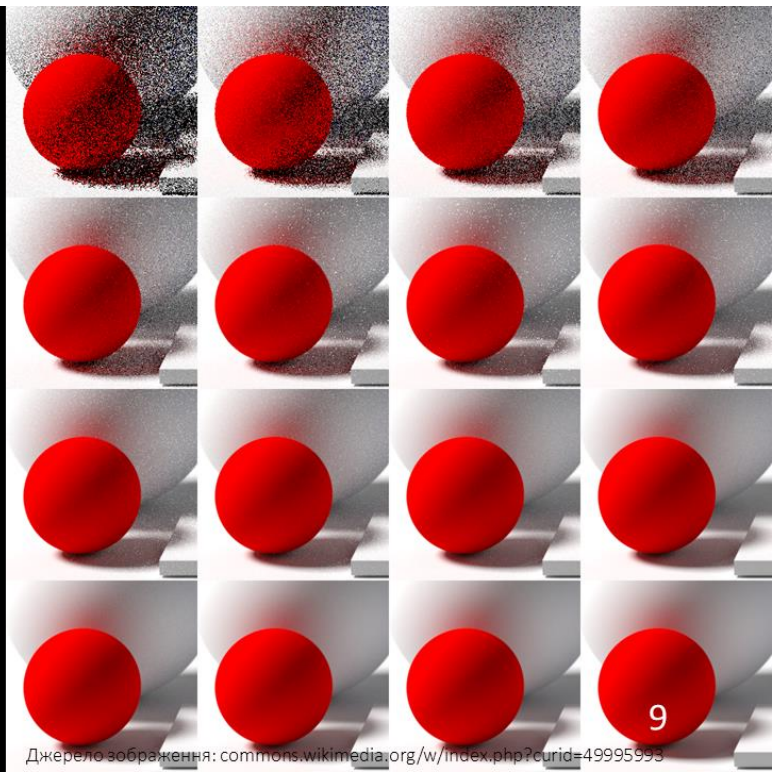
Ядра прискорення ШІ

Тензорні ядра розроблені для прискорення матричних обчислень, особливо у високопродуктивних обчисленнях (HPC) і робочих навантаженнях, пов’язаних зі штучним інтелектом.

Джерело зображення: https://www.nvidia.com/en-us/design-visualization/technologies/turing-architecture/

Зменшення шуму

Один з найголовніших способів застосування ШІ у комп'ютерній графіці полягає у зменшенні шумів та апскейлінгу зображення.

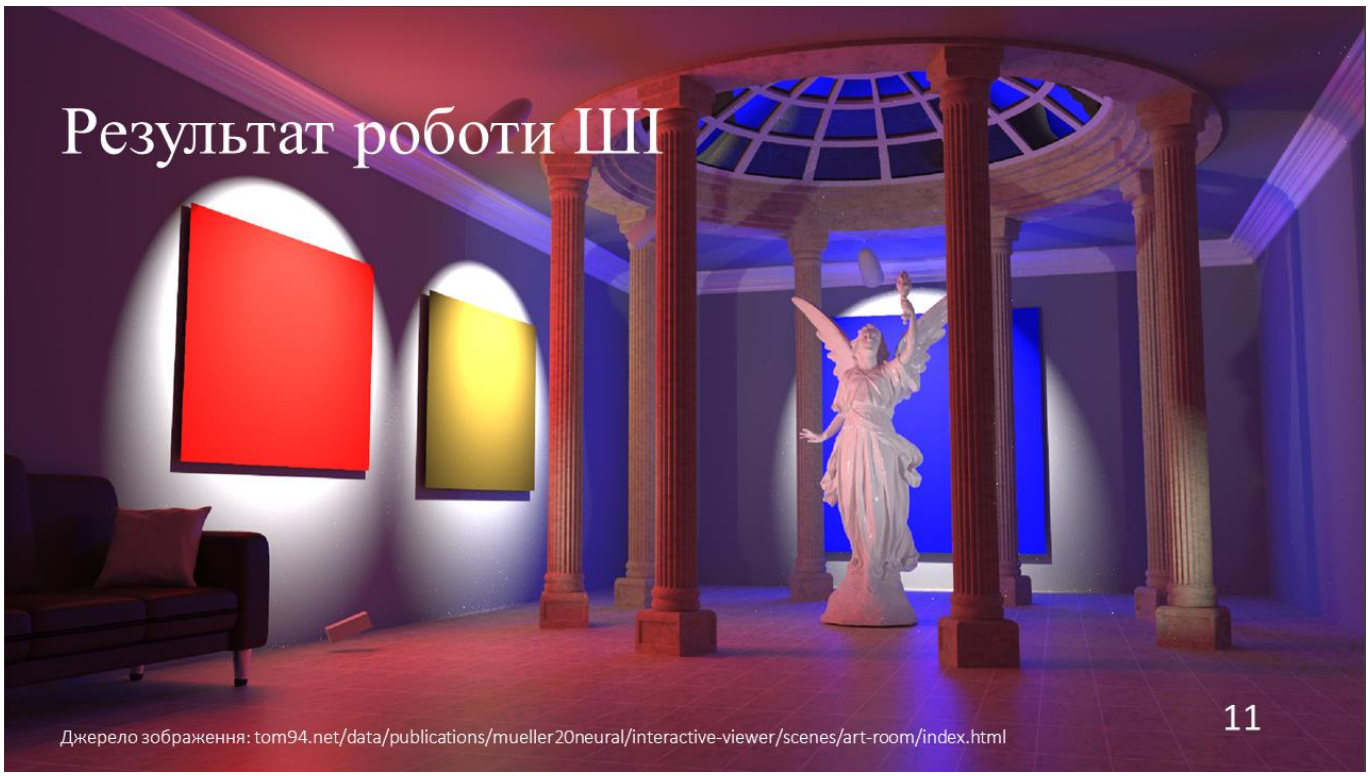


Джерело зображення: commons.wikimedia.org/w/index.php?curid=49995993

Рендер з дуже високим рівнем шуму

Джерело зображення: tom94.net/data/publications/mueller20neural/interactive-viewer/scenes/art-room/index.html

Результат роботи ШІ

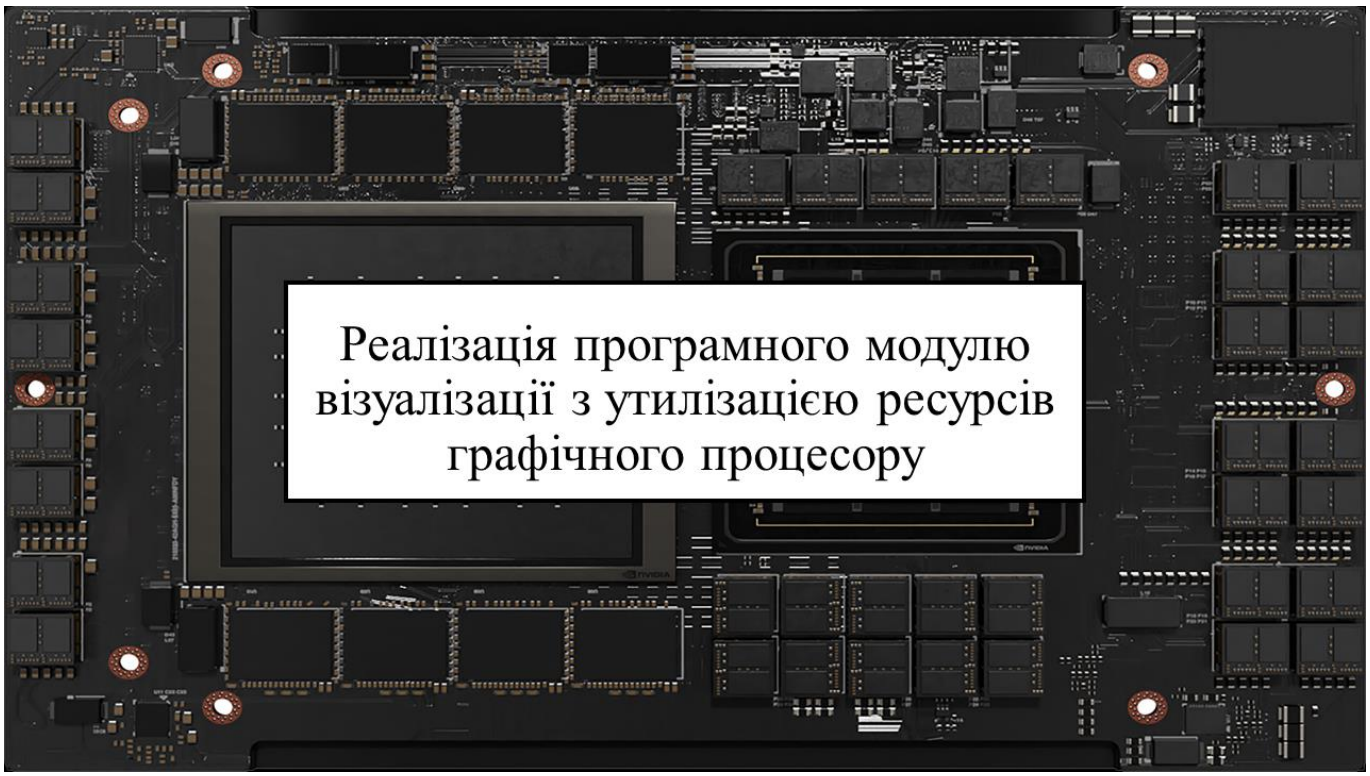


Джерело зображення: tom94.net/data/publications/mueller20neural/interactive-viewer/scenes/art-room/index.html

Складнощі пов’язані з використанням графічного процесору

Через стрімкий розвиток технології, сильну залежність від архітектури, доступних API та можливостей конкретної моделі відеокарти, створення програмного забезпечення, може сильно обмежувати вибір платформи, на якій програма може працювати або примушувати розробляти окремий код для різних відеокарт.

Джерело зображення:
https://www.nvidia.com/en-us/geforce/news/geforce-rtx-4070-ti/



NVIDIA

CUDA

CUDA Toolkit включає компілятори, бібліотеки, засоби налагодження та інші ресурси, необхідні для програмування CUDA ядер, якими обладнані графічні процесори виробника NVIDIA.

```
__device__ vec3 color(const ray& r, hitable** world, curandState* local_rand_state) { ... }
__host__ __device__ inline vec3 operator+(const vec3& v1, const vec3& v2) { ... }
```

У контексті програмування з використанням CUDA центральний процесор називають хостом, а відеокарту — девайсом. Щоб позначити компілятору, для якого пристрою призначено код функціям та змінним надаються відповідні префікси — __host__ та __device__. Обидва префікси можуть використовуватися одночасно, що дозволить використовувати функцію/змінну як на ЦП так і на графічному процесорі. Якщо префікс явно не вказано, вважається що код має префікс __host__.


```
__global__ void render_init(int max_x, int max_y, curandState* rand_state) { ... }

render_init<<<blocks, threads>>>(nx, ny, d_rand_state);
```

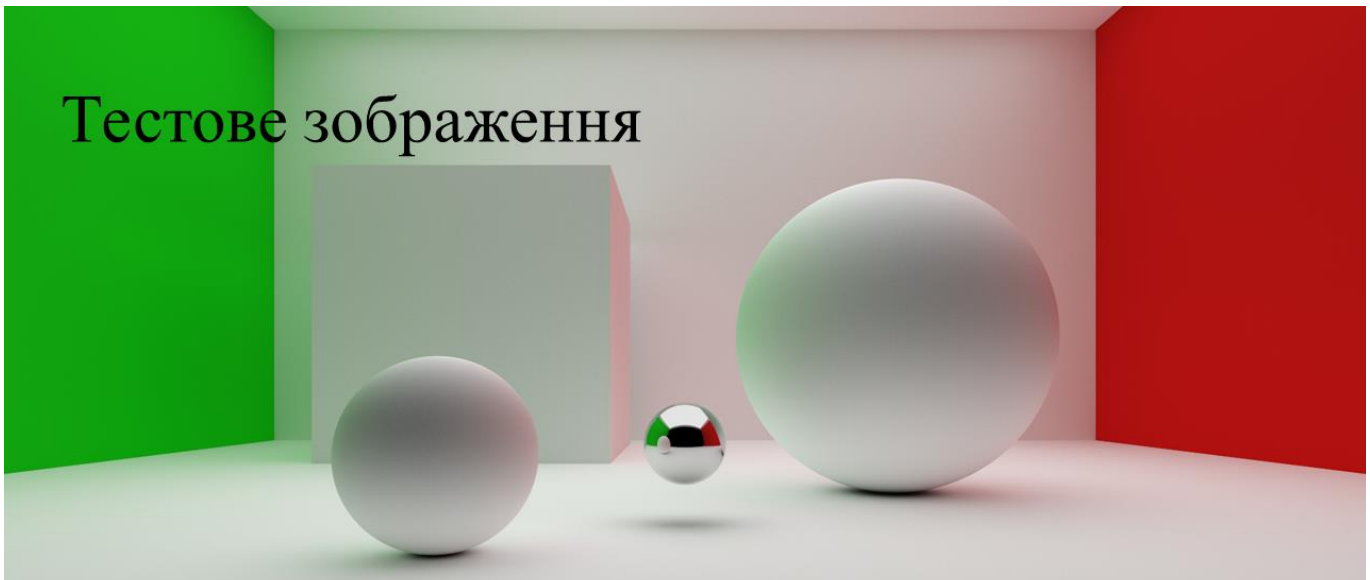
Однак викликати код для девайсу з хоста або навпаки звичайним чином не можливо. Єдиний випадок, у якому можна викликати код девайсу з хоста є “запуск ядра” для якого використвується префікс __global__.

```
render<<<blocks, threads>>>(fb, nx, ny, ns, d_camera, d_world, d_rand_state);
```

У програмуванні CUDA блоки та потоки є основними компонентами, які використовуються для організації та виконання паралельних обчислень на GPU. Вони є частиною моделі виконання CUDA і відіграють вирішальну роль у використанні можливостей паралельної обробки графічних процесорів NVIDIA.

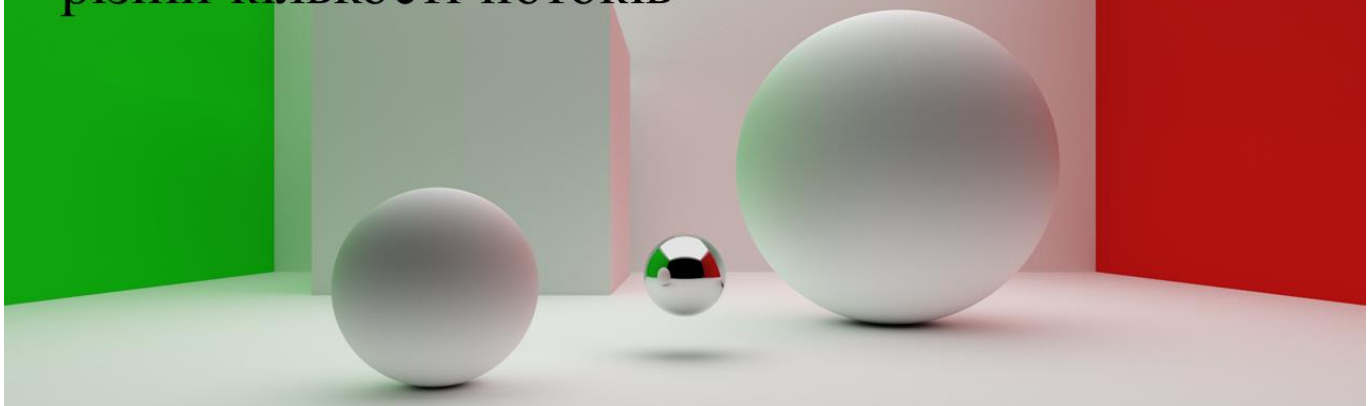
Потоки є найменшою одиницею виконання в CUDA. Вони виконують той самий код, але працюють з різними даними. Блоки — це групи потоків, які разом виконуються на GPU.

Тестове зображення



Для проведення дослідження роботи розробленого рушія візуалізації важливо створити сцену, що буде відтворювати умови повсякденних задач візуалізації. Данна сцена представляє собою замкнений простір з одним джерелом світла на стелі та об’єкти з різними фізичними характеристиками.

Порівняння швидкодії при різній кількості потоків



Під час візуалізації, зображення розбивається на секції по $n \times m$ пікселів, які обраховуються незалежними потоками одночасно. Дослідимо, як змінюється час рендерингу залежно від зміни розміру секцій і відповідно кількості потоків.

Відеокарта: NVIDIA RTX 3070 Ti

Параметри рендерингу:

Розмір зображення: 1920x1080

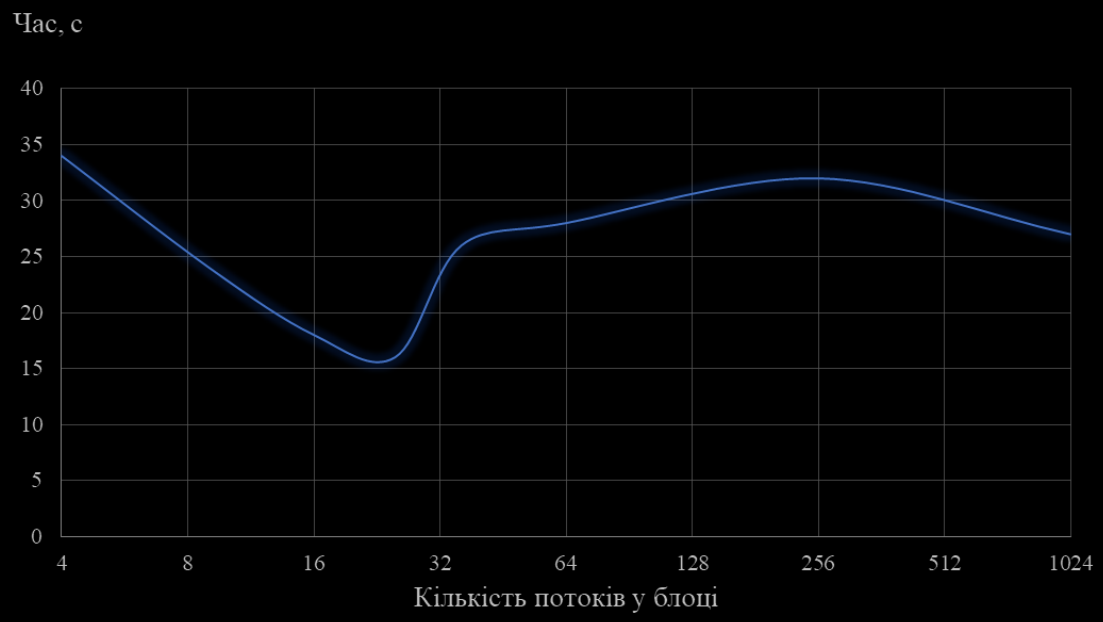
Максимальна кількість відбиттів променя: 12

Кількість семплів: 512 на піксель.

Результати:

2x2 пікселі	3x3 пікселі	4x4 пікселі	5x5 пікселів	6x6 пікселі	8x8 пікселі	16x16 пікселів	32x32 пікселі
34 с	24 с	18 с	16 с	26 с	28 с	32 с	27 с

Бачимо, що найшвидшого часу рендерингу було досягнуто при поділі зображення на блоки 5 × 5 пікселі. Дані показники можуть суттєво відрізнятися для різних графічних процесорів, тому імовірно, що більші або менші розміри блоків можуть показати кращі результати на інших відеокартах.



Відеокарта: NVIDIA RTX 3050 Ti

Параметри рендерингу:

Розмір зображення: 1920x1080

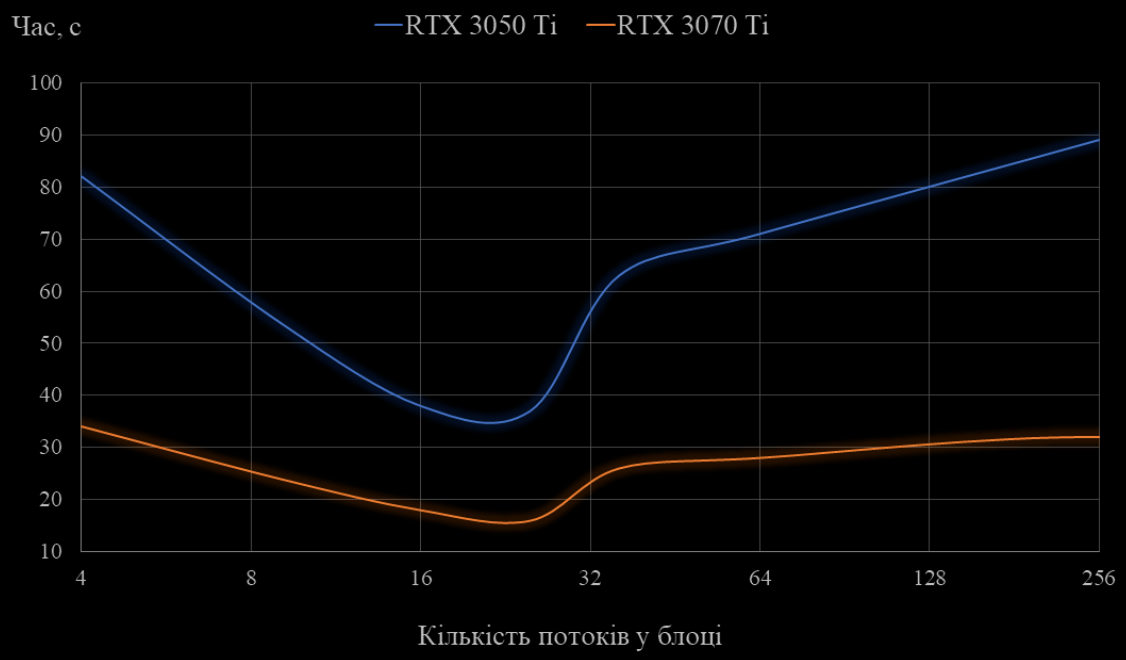
Максимальна кількість відбиттів променя: 12

Кількість семплів: 512 на піксель.

Результати:

2x2 пікселі	3x3 пікселі	4x4 пікселі	5x5 пікселів	6x6 пікселі	8x8 пікселі	16x16 пікселів
82 с	54 с	38 с	37 с	63 с	71 с	89 с

Найбільш оптимальний розмір блоків для даного графічного процесору є так само 5 × 5 пікселі (25 потоків відповідно). Це може бути спричинено тим, що процесори належать до одного покоління архітектури, або з тим, що вони мають однаковий об’єм відеопам’яті.



Порівняння швидкодії з обчисленнями на ЦП

Порівняємо швидкодію розробленого рушію візуалізації з аналогічною програмою, що веде обрахунки виключно на процесорі. Алгоритм візуалізації в обох рушіїв є ідентичним та відрізняється виключно імплементацією.

Відеокарта: NVIDIA RTX 3070 Ti
Параметри рендерингу:
Розмір зображення: 1920x1080
Максимальна кількість відбиттів променя: 12
Кількість семплів: 512 на піксель
Розмір блоків: 5x5

Час: 16 с

Процесор: Intel Core i5-6600K
Параметри рендерингу:
Розмір зображення: 1920x1080
Максимальна кількість відбиттів променя: 12
Кількість семплів: 512 на піксель
Кількість потоків: 4

Час: 384 с

Отже за рахунок утилізації ресурсів відеокарти вдалося досягти пришвидшення процесу візуалізації у 24 рази!

Висновки

У кваліфакаційній магістерській роботі досліджено технології, пов’язані з графічними процесорами.

Вивчено особливості створення програми для виконання на відеокарті.

Розроблено рушій візуалізації методом трасування шляху з утилізацією ресурсів графічного процесору.

Проведено емпіричні дослідження швидкодії розробленого модуля та порівняно з реалізацією цього ж алгоритму виключно на центральному процесорі.

Роботу було апробовано на Всеукраїнській науково-технічній конференції «Технологічні горизонти: дослідження та застосування інформаційних технологій для технологічного прогресу України і світу»