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

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування і спеціалізованих комп‘ютерних систем

«На правах рукопису»

УДК 004.925.3

«До захисту допущено»

Завідувач кафедри СПСКС
_____ Віталій РОМАНКЕВИЧ
_____
(підпис) (ініціали, прізвище)
“___” грудень 2022 р.

Магістерська дисертація

на здобуття ступеня магістра

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

на тему: Спосіб апаратно-програмного трасування променів у задачах тривимірної симулаяції

Виконав :

студент II курсу, групи КВ-11мп
(шифр групи)

Бербега Володимир Олегович _____ _____
(прізвище, ім’я, по батькові) (підпис)

Керівник доц. каф. СПСКС, к.т.н. Петрашенко А.В. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

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

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

Київ – 2022 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування і спеціалізованих комп‘ютерних систем

Рівень вищої освіти – другий (магістерський)

Спеціальність 123 «Комп’ютерна інженерія»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Віталій РОМАНКЕВИЧ

(підпис) (ініціали, прізвище)

«___» _____2022 р.

ЗАВДАННЯ
на магістерську дисертацію студента
Бербега Володимир Олегович

1. Тема дисертації Спосіб апаратно-програмного трасування променів у задачах тривимірної симуляції _____,
керівник проєкту доц. каф. СПСКС, к.т.н. Петрашенко А.В. _____ ,
затверджені наказом по університету від «__» _____ 2022р. №_____

2. Термін подання студентом проєкту 13.12.2022

3. Об’єкт дослідження: процес трасування променів в реальному часі.

4. Предмет дослідження: способи трасування променів та використання графічного процесора в трасуванні променів.

5. Перелік завдань, які потрібно розробити : провести аналіз методів трасування променів; дослідити використання графічних процесорів;

запропонувати та обгрунтувати вибір методу трасування променів; розробити програмне забезпечення для реалізації способу апаратно-програмного трасування променів.
6. Орієнтований перелік графічного (ілюстрованого) матеріалу: презентація.

7. Дата видачі завдання «11» жовтня 2021р._____

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів	Примітка
1.	Вивчення літератури за тематикою дисертації	15.10.2021	
2.	Визначення структури магістерської дисертації	11.12.2021	
3.	Робота над першим розділом магістерської дисертації; проведення наукового дослідження	21.02.2022	
4.	Підготовка матеріалів першого розділу дипломного проекту	02.05.2022	
5.	Проведення наукового дослідження; робота над другим розділом магістерської дисертації; розроблення програмного забезпечення	04.06.2022	
6.	Проведення наукового дослідження; робота над статтею за результатами наукового дослідження	28.07.2022	
7.	Проведення наукового дослідження; робота над третім розділом магістерської дисертації	27.09.2022	
8.	Підготовка матеріалів четвертого розділу магістерської дисертації	11.09.2022	
9.	Завершення роботи над основною частиною магістерської дисертації; підготовка ілюстративного матеріалу; підготовка матеріалів доповіді на конференції ПМК-2018	27.10.2022	
10.	Оформлення текстової і графічної частини магістерської дисертації	30.11.2022	

Студент _____ Володимир БЕРБЕГА
(підпис) (ініціали, прізвище)

Науковий керівник дисертації _____ Андрій ПЕТРАШЕНКО
(підпис) (ініціали, прізвище)

РЕФЕРАТ

Актуальність теми. Однією із актуальних задач при відтворенні тривимірних комп'ютерних симуляцій є отримання реалістичного відображення сцени. Дана задача напряму перекликається з не менш актуальною задачею комп'ютерної графіки – отриманням реалістичного зображення. Реалістичне зображення характеризується такими ефектами як каустика, нечіткі відбиття, м'які тіні, блиск, напівпрозорість. Серед існуючих способів реалістичного відображення зображення одним із найбільш точних являється метод трасування променів, а із правильною оптимізацією даний спосіб здатний на візуалізацію в реальному часі.

Існує багато різноманітних способів трасування променів, отже з'являється необхідність у виборі найбільш ефективного з точки зору реалістичності та продуктивності.

Об'єктом дослідження є спосіб апаратно-програмного трасування променів у задачах тривимірної симуляції.

Предметом дослідження є спосіб апаратного-програмного рендерингу тривимірного зображення методом трасування променів у задачах тривимірної симуляції.

Мета роботи: покращення продуктивності апаратно-програмного трасування променів на основі аналіз існуючих способів трасування променів у задачах тривимірної симуляції; дослідження апаратно-програмних можливостей способів рендерингу тривимірного зображення методом трасування променів; створення програмного забезпечення для рендерингу зображення для задач тривимірної симуляції, що буде базуватися на способі апаратно-програмного трасування променів.

Наукова новизна полягає в наступному: запропоновано підхід апаратно-програмного трасування променів у задачах тривимірної симуляції що відрізняється від існуючих способів тим, що дає змогу ефективніше використовувати апаратну складову персонального комп'ютера і дозволяє підвищити швидкодію роботи вихідної тривимірної симуляції в реальному часі.

Практична цінність отриманих в роботі результатів полягає в тому, що запропонований спосіб апаратно-програмного трасування променів дає змогу візуалізувати реалістичне зображення у задачах тривимірної симуляції. Розроблений спосіб дає змогу візуалізовувати сцени з достатнім рівнем швидкодії для роботи візуалізації в реальному часі порівняно з іншими способами візуалізації на основі алгоритму трасування променів. Покращення продуктивності становить приблизно 15-20 відсотків.

Апробація роботи. Основні положення і результати роботи були представлені та обговорювались на науковій конференції магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2022 та на IX Міжнародній науково-технічній Internet-конференції «Сучасні методи, інформаційне, програмне та технічне забезпечення систем керування організаційно-технічними та технологічними комплексами».

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

У вступі подано загальну характеристику роботи, зроблено оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку досліджень, сформульовано мету і задачі досліджень, показано наукову новизну отриманих результатів і практичну цінність роботи.

У першому розділі розглянуто існуючі способи трасування променів в задачах тривимірної симуляції, а також проведений аналіз, який дає змогу визначити основні переваги та недоліки існуючих технологій візуалізації шляхом використання трасування променів.

У другому розділі наведено результати аналізу засобів реалізації для розробки програмного забезпечення, що реалізовує спосіб апаратно-програмного трасування променів. Розглянуто базові концепції трасування променів та апаратно-програмної реалізації трасування променів.

У третьому розділі описується реалізоване програмне забезпечення, підхід до розробки, технології та засоби розробки, що були використані під час розробки ПЗ.

У *четвертому розділі* описується тестування, аналіз роботи розробленого програмного забезпечення та порівняння з існуючими рішеннями.

У висновках представлені результати проведеної роботи.

Робота представлена на 91 аркушах, містить посилання на список використаних літературних джерел.

Ключові слова: візуалізація, рендеринг, трасування променів, тривимірна симуляція, апаратно-програмний спосіб.

ABSTRACT

Relevance of the topic. One of the urgent tasks in the reproduction of three-dimensional computer simulations is to obtain a realistic representation of the scene. This task is directly related to the no less relevant task of computer graphics - obtaining a realistic image. A realistic image is characterized by such effects as caustics, blurred reflections, soft shadows, gloss, translucency. Among the existing methods of realistic image display, one of the most accurate is the beam shaking method, and with proper optimization, this method is capable of real-time visualization.

There are many different ways to trace rays, so it becomes necessary to choose the most effective in terms of realism and performance.

The object of research is the method of hardware and software ray tracing in three-dimensional simulation tasks.

The subject of the research is the method of hardware and software rendering of a three-dimensional image by ray tracing in three-dimensional simulation tasks.

Objective: improving the performance of hardware and software ray tracing based on the analysis of existing ray tracing methods in three-dimensional simulation tasks; research of hardware and software capabilities of three-dimensional image rendering methods using ray tracing; creation of image rendering software for three-dimensional simulation tasks, which will be based on the method of hardware-software ray tracing.

The scientific novelty: a hardware-software ray tracing approach in three-dimensional simulation tasks is proposed, which differs from existing methods in

that it allows more efficient use of the hardware component of a personal computer and allows to increase the speed of the output three-dimensional simulation in real time.

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

Approbation of work. The main provisions and results of the work were presented and discussed at the scientific conference of master's and postgraduate students "Applied mathematics and computing" PMK-2022 and at the IX International scientific and technical Internet conference "Modern methods, information, software and technical support of management systems of organizational technical and technological complexes".

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

The introduction provides a general description of the work, makes an assessment of the current state of the problem, substantiates the relevance of the research direction, formulates the purpose and tasks of the research, shows the scientific novelty of the obtained results and the practical value of the work, provides information about the approbation of the results and their implementation.

In the *first section*, existing methods of ray tracing in three-dimensional simulation tasks are considered, as well as an analysis that allows determining the main advantages and disadvantages of existing visualization technologies by using ray tracing.

The *second section* presents the results of the analysis of means of implementation for the development of software that implements the method of hardware-software ray tracing. Basic concepts of ray tracing and hardware and software implementation of ray tracing are considered

The *third section* describes the implemented software, the development approach, technologies and development tools that were used during the development of the software.

The *fourth section* describes testing, performance analysis of the developed software and comparison with existing solutions.

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

The work is presented on 91 pages, contains links to the list of references used.

Keywords: visualization, rendering, ray tracing, three-dimensional simulation, hardware-software method.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

API – Прикладний програмний інтерфейс;

AS – Структура прискорення;

CPU –центральний процесор, або ж ЦЗ;

GPU –графічний процесор;

RGB – Модель, що описує спосіб синтезу кольору;

PSO – Об’єкт стану конвеєра;

SBT – Таблиця зв’язування шейдерів;

SSBO – Об'єкт буфера зберігання шейдерів;

ПЗ – Програмне забезбечення;

3D – Тривимірний опис об’єкта;

ВСТУП

В еру стрімкого розвитку комп’ютерних технологій та електроніки комп’ютерна графіка стає невід’ємною частиною в житті кожного з нас. Так візуалізація зображення є одним із провідних напрямків у дослідженнях комп’ютерних технологій. Станом на сьогодні вже існує безліч способів рендерингу зображення, як двовимірного так і тривимірного, на екран. Спеціалізованого програмного забезпечення, що спирається на відомі принципи візуалізації, існує не менше, а навіть більше ніж відомих методів рендеру. В силу специфіки задач, що ставив розробник ПЗ, чи направленості створеної прикладної програми можуть використовуватись як реалістичні так і не реалістичні методи візуалізації. До реалістичних способів можна віднести способи кидання променів, трасування променів, трасування шляху та метод фотонних карт.

Метою даної роботи є дослідження існуючих методів рендерингу тривимірних сцен та розробка апаратно-програмного способу візуалізації зображення для задач тривимірної симуляції шляхом використання алгоритму трасування променів.

1. АНАЛІЗ ІСНУЮЧИХ СПОСОБІВ ВІЗУАЛІЗАЦІЇ В ЗАДАЧАХ ТРИВИМІРНОЇ СИМУЛЯЦІЇ

1.1. Основні визначення

Комп'ютерна візуалізація (Рендеринг) [2] — це процес отримання зображення за заздалегідь заданою сценою чи моделлю за допомогою використання спеціалізованого програмного забезпечення. Тут модель — це опис тривимірних об'єктів сцени що візуалізується певною мовою програмування і у вигляді певної структури. Такий опис містить всі дані сцени такі як геометричні дані, положення точки спостерігача, інформацію про освітлення, тощо. Комп'ютерна візуалізація є одним із найважливіших розділів комп'ютерної графіки. Для візуалізації створюються самостійні програмні пакети — рендерери. Таке програмне забезпечення може інтегруватись до ПЗ, що направлене на тривимірне моделювання, анімацію, відеомонтаж, 2D малювання та фоторедагування.

Різновиди комп'ютерної візуалізації:

- фотореалістична візуалізація;
- нефотореалістична візуалізація;

Зображення — це результат візуалізації, що може бути описаний як набір певних візуальних особливостей, що відповідають фізичним явищам чи властивостям об'єкту. Дослідження та розробки у області комп'ютерної візуалізації шукають найкращі способи для більш коректної та ефективної їх симуляції.

Особливості рендерингу:

- текстурна карта (спосіб нанесення матеріалу на поверхню);
- шейдинг (визначення зміни кольору і яскравості поверхні в залежності від освітлення);
- відображення;
- глибина різкості (ефект відображення об'єктів при якому об'єкти, на яких не фокусується камера, здаються розмитими або не в фокусі);
- дифракція (вигин, поширення та інтерференція світла, що проходить поблизу границі об'єкта);
- заломлення (вигин світла, відносно коефіцієнту заломлення матеріалу);
- рельєфне текстурування;
- каустика (відбиття світла від блискучого об'єкта, або фокусування світла через прозорий об'єкт, для отримання яскравих відблисків на інший об'єкт);
- м'які тіні;
- непряме освітлення (світло, що було відбите від інших поверхонь, а не виходило безпосередньо від джерела світла);
- нефотореалістична візуалізація (рендеринг сцен в художньому стилі).
- прозорість або непрозорість;
- тінь;
- ефект туману;
- розмиття в русі.

Тривимірна симуляція – це комп’ютерна симуляція певних явищ, процесів, законів чи середовища як в реальному часі, так і шляхом пререндерингу за допомогою використання спеціалізованого програмного забезпечення.

1.2. Способи візуалізації в задачах тривимірної симуляції

Кидання променів (рис. 1.1.1) [3] – найпростіший спосіб візуалізації двовимірного чи тривимірного зображення з використанням променів. Принцип роботи даного алгоритму є таким, із точки сцени з якої зображення бачить глядач, виходять промені направлені вертикальними лініями на всі об’єкти тривимірної сцени, що знаходяться в полі зору глядача. Після того як промінь перетинає об'єкт сцени, він далі не. Так, рендерер, що базується на даному способі, розуміє як представити об’єкт сцени на екрані.

Рисунок 1.1.1 – Ілюстрація способу кидання променів

Трасування променів (рис. 1.1.2) [3] – спосіб візуалізації тривимірного зображення на екран, який базується на роботі з промінням. Даний спосіб рендеру є досить ресурсоємним. За суттю роботи суміжний із киданням променів. Але має досить суттєву відмінність. При застосуванні трасування променів, промені, що були випущені із точки зору глядача сцени, проходять через кожен піксель екрану. Після перетену з об’єктом тривимірної сцени

промені заломлюються, тобто відбиваються від поверхні, що має властивість відображення. Таким чином після перетену променя із усіма можливими об’єктами на своєму шляху, визначається колір та яскравість пікселя з якого був випущений даний промінь. Так, задопомогою даного способу візуалізації тривимірної сцени, користувач має змогу переглядати зображення, що симулює різні оптичні ефекти, серед яких відобреження, заломлення світла або його розсіювання. Трасування променів дозволяє створювати досить реалістичне освітлення в тривимірних сценах, що в свою чергу значною мірою впливає на реалістичність відображення вихідного зображення. Спочатку даний спосіб рендеру застосовували лише у пререндирунгу, але з розвитком апаратного забезпечення, даний спосіб відображення об’єктів тривимірної сцени отримав змогу працювати в режимі рендерингу в реальному часі.

Рисунок 1.1.2 – Візуалізація реалістичного зображення задопомогою трасування променів

Трасування шляху (рис. 1.1.3) [3] – спосіб візуалізації тривимірного зображення на екран, що в своїй основі має роботу з променями. Даний спосіб має найбільш точну симуляцію осітлення серед усіх інших способів візуалізації тривимірного зображення, що базуються на технології випускання променів. На відміну від інших способів візуалізації зображення з використанням променів, трасування шляху випускає всі промені не із точки зору глядача тривимірної сцени, а із точки, де знаходиться передбачуване джерело світла. Також, до основних особливостей даного способу відноситься те, що в цьому способі промінь приломляється до тих пір, поки не буде повністю розсіяний, або поки він не буде поглинутим об’єктом тривимірної сцени, що візуалізується. Так, даний спосіб поруч із своєю вагомою перевагою, у вигляді найбільш точної симуляції освітлення сцени тривимірного, має досить серйозний недолік. Недоліком є його ресурсозатратність, що є найбільшою серед усіх інших методів візуалізації тривимірного зображення, що в своїй основі мають технологію випускання променів.

Рисунок 1.1.3 – Візуалізація реалістичного зображення задопомогою трасування шляху

Метод фотонних карт (рис. 1.1.4) — один з найбільш універсальних та поширених алгоритмів рендерингу. Спосіб складається з трьох частин: трасування фотонів, побудови фотонної карти та збір освітленості.

Фотони в цьому методі — це умовні частинки, що переносять деяку дискретну дозу світла. Спочатку фотони випромінюються з умовного передбаченого джерела світла в тривимірній сцені відповідно до розподілу світла цього джерела. В процесі трасування фотони перетинаються з різними об'єктами сцени, що візуалізується. Залежно від особливостей матеріалу об'єкту, з ними можуть відбуватися різні події: фотон може відбитися у випадковому напрямку, тобто розсіяно, змінити напрямок при перетині поверхні, тобто дзеркально, або поглинутися. Рішення про те, яка з подій відбувається з фотоном, приймається на основі особливостей матеріалу об'єкту.

Рисунок 1.1.4 – Візуалізація реалістичного зображення за допомогою методу фотонних карт

1.3. Аналіз інсуючих способів трасування променів

У 3D-графіці, створеній комп’ютером, трасування променів є формою техніки візуалізації для розрахунку освітлення з метою створення фотореалістичного зображення.

В реальному сівті джерело світла випромінює промінь світла, що, потрапляє на поверхню й тим самим зупиняє його рух. Цей промінь можна розглядати як потік фотонів, що рухаються в одному напрямку. У ідеальному вакуумі цей промінь буде прямою лінією (без урахування релятивістських ефектів). З цим світловим променем може статися будь-яка комбінація

чотирьох речей: поглинання, відбиття, заломлення та флуоресценція. Поверхня може поглинати частину світлового променя, що призводить до втрати інтенсивності відбитого та/або заломленого світла. Вона також може відбивати весь/або частину світлового променя в одному чи кількох напрямках. Якщо поверхня має будь-які прозорі або напівпрозорі властивості, вона заломлює частину світлового променя в себе в іншому напрямку, одночасно поглинаючи частину (або весь) спектр і, можливо, змінюючи колір. Рідше поверхня може поглинати деяку частину світла та флуоресцентно повторно випромінювати світло з довшою довжиною хвилі кольору у випадковому напрямку, хоча це досить рідко, тому його можна скинути з більшості програм візуалізації. Між поглинанням, відображенням, заломленням і флуоресценцією необхідно враховувати все вхідне світло, і не більше. Поверхня не може, наприклад, відбивати 66% вхідного світлового променя і заломлювати 50%, оскільки ці два разом складатимуть 116%. Звідси відбиті та/або заломлені промені можуть потрапляти на інші поверхні, де їхні властивості поглинання, заломлення, відбиття та флуоресценції знову впливають на рух вхідних променів. Деякі з цих променів рухаються таким чином, що вони потрапляють на наше око, змушуючи нас бачити сцену, і таким чином сприяють остаточному відтвореному зображенню.

Ідея трасування променів, полягає в тому, щоб відстежувати промені від ока, по одному на піксель, і знаходити найближчий об’єкт, який блокує шлях цього променя. Використовуючи властивості матеріалу та ефект світла в сцені, алгоритм трасування променів може визначити затінення об’єкту сцени, що візуалізовується. Затінення поверхні обчислюється за допомогою традиційних моделей затінення. Однією з важливих переваг трасування променів є його здатність легко працювати з неплоскими поверхнями та твердими тілами,

такими як конуси та сфери. Якщо математичну поверхню можна перетнути променем, її можна відобразити за допомогою трасування променів.

У методі об’ємного кидання променів кожен промінь трасується, щоб колір і/або щільність можна було взяти вздовж променя, а потім об’єднати в остаточний колір пікселя. Це часто використовується, коли об’єкти не можуть бути легко представлені явними поверхнями (такими як трикутники), наприклад, під час візуалізації хмар.

У SDF похідних променів, або трасуванні сфери, кожен промінь трасується в декілька кроків, щоб наблизити точку перетину між променем і поверхнею, визначеною функцією відстані зі знаком. SDF оцінюється для кожної ітерації, щоб мати можливість робити якомога більші кроки, не пропускаючи жодної частини поверхні. Порогове значення використовується для скасування подальшої ітерації, коли досягнуто точки, яка знаходиться досить близько до поверхні. Цей метод часто використовується для тривимірної фрактальної візуалізації.

Попередні алгоритми відстежували промені від ока до сцени, доки вони не потрапляли на об’єкт, але визначали колір променя без рекурсивного відстеження додаткових променів. Рекурсивне трасування променів продовжує процес. Коли промінь потрапляє на поверхню та після його перетину із об'єктом можуть виникати додаткові промені через відображення, заломлення та тінь:

- Промінь відбиття простежується в напрямку дзеркального відбиття. Найближчий об'єкт, який він перетинає, це те, що буде видно у відображенні.

- Заломлюючий промінь, що проходить через прозорий матеріал, працює аналогічно, але заломлюючий промінь може входити або виходити з матеріалу.
- Промінь тіні простежується до кожного світла. Якщо між поверхнею та світлом знаходиться будь-який непрозорий предмет, поверхня знаходиться в тіні, і світло її не освітлює.

Ці рекурсивні промені додають реалістичності зображенням, що були відображені із застовуванням трасування променів.

Популярність візуалізації на основі трасування променів пояснюється його основою в реалістичній симуляції освітлення порівняно з іншими методами візуалізації, такими як растеризація, яка більше зосереджується на реалістичній симуляції геометрії. Такі ефекти, як відблиски та тіні, які важко змоделювати за допомогою інших алгоритмів, є природним результатом алгоритму трасування променів. Обчислювальна незалежність кожного променя робить трасування променів придатним для базового рівня розпаралелювання.

Серйозним недоліком трасування променів є продуктивність. До кінця 2010-х років трасування променів у реальному часі зазвичай вважалося неможливим на звичайному користувацькому обладнанні для нетривіальних завдань. Алгоритми Scanline та інші алгоритми використовують послідовність даних для обміну обчисленнями між пікселями, тоді як трасування променів зазвичай починає процес заново, обробляючи кожен промінь ока окремо. Однак це розділення пропонує інші переваги, такі як можливість знімати більше променів, якщо потрібно, щоб виконати просторове згладжування та покращити якість зображення, де це необхідно.

Незважаючи на те, що традиційне трасування променів не обов’язково фотореалістичне, воно справляється з взаємовідбиттям і оптичними ефектами, такими як заломлення. Справжній фотореалізм виникає, коли рівняння візуалізації є наближеним або повністю реалізованим. Реалізація рівняння візуалізації дає справжній фотореалізм, оскільки рівняння описує кожен фізичний ефект потоку світла. Однак це зазвичай неможливо з огляду на необхідні обчислювальні ресурси.

Реалістичність усіх методів візуалізації можна оцінити як наближення до рівняння. Трасування променів, не обов’язково є найреалістичнішим. Методи, які відстежують промені, але включають додаткові методи (фотонне відображення, відстеження шляху), дають більш точну симуляцію реального освітлення.

1.4. Аналіз технології Nvidia RTX

Nvidia GeForce RTX [9] — висококласна професійна візуальна обчислювальна платформа, створена Nvidia, яка в основному використовується для візуалізації складних, великомасштабних тривимірних моделей та сцен в архітектурі, дизайні продуктів, науковій візуалізації, дослідженні енергії, відеоіграх, а також виробництві фільмів і відео. Nvidia RTX підтримує трасування променів у реальному часі. Історично трасування променів було зарезервовано для програм, які не працюють у реальному часі, наприклад, CGI у візуальних ефектах для фільмів і фотореалістичних візуалізаціях, а відеоігри повинні були покладатися на пряме освітлення та попередньо розрахований непрямий внесок для їх рендерингу. RTX сприяє новому розвитку генерації інтерактивних зображень в комп’ютерній графіці, які реагують на освітлення, тіні та відображення. RTX працює на графічних

процесорах на основі Nvidia Volta, Turing, Ampere та Ada Lovelace, зокрема з використанням ядер Tensor і нових ядер RT на Turing і наступних архітектурах для апартного прискорення трасування променів.

1.5. Аналіз технології CUDA

CUDA [4] — це паралельна обчислювальна платформа та інтерфейс прикладного програмування (API), який дозволяє програмному забезпеченню використовувати певні типи графічних процесорів (GPU) для обробки загального призначення. Цей підхід називається обчисленням загального призначення на GPU (GPGPU). CUDA — це програмний інтерфейс, який надає прямий доступ до віртуального набору інструкцій GPU та паралельних обчислювальних елементів для виконання обчислень на обчислювальних ядрах.

Платформа CUDA доступна розробникам програмного забезпечення через прискорені бібліотеки CUDA, директиви компілятора, такі як OpenACC, і розширення стандартних мов програмування, включаючи C, C++ і Fortran. Програмісти C/C++ можуть використовувати «CUDA C/C++», скомпільований у PTX за допомогою nvcc, компілятора C/C++ Nvidia на основі LLVM або за допомогою самого clang. Програмісти Fortran можуть використовувати «CUDA Fortran», скомпільований за допомогою компілятора PGI CUDA Fortran від The Portland Group.

На додаток до бібліотек, директив компілятора, CUDA C/C++ і CUDA Fortran, платформа CUDA підтримує інші обчислювальні інтерфейси, включаючи OpenCL від Khronos Group, DirectCompute від Microsoft, OpenGL Compute Shader і C++ AMP. Оболонки сторонніх розробників також доступні

для Python, Perl, Fortran, Java, Ruby, Lua, Common Lisp, Haskell, R, MATLAB, IDL, Julia.

В індустрії комп'ютерних ігор графічні процесори використовуються для візуалізації графіки та для обчислень таких фізичних ефектів, таких як уламки, дим, вогонь, рідини. CUDA також використовувалася для прискорення неграфічних додатків у обчислювальній біології, криптографії та інших.

CUDA надає як API низького рівня (API драйвера CUDA), так і API вищого рівня (API середовища виконання CUDA). Початковий CUDA SDK був опублікований 15 лютого 2007 року для Microsoft Windows і Linux. Підтримка Mac OS X була пізніше додана у версії 2.0. CUDA працює з усіма графічними процесорами Nvidia, починаючи з серії G8x, включаючи GeForce, Quadro та лінійку Tesla. CUDA сумісна з більшістю стандартних операційних систем.

При порівнянні з традиційним підходом до обчислень загального призначення допомогою можливостей графічних API, у архітектури CUDA відзначають такі переваги:

- Інтерфейс програмування додатків CUDA (CUDA API) заснований на стандартній мові програмування C з деякими обмеженнями. Тобто, це повинно спростити і згладити процес вивчення архітектури CUDA;
- Колективна пам'ять між потоками (shared memory);
- Більш ефективна передача даних між пам'яттю центрального процесора і відеопам'яттю;
- Апаратна підтримка побітових та цілочисельних операцій;
- Підтримка компіляції GPU коду засобами LLVM.

Основним недоліком технології CUDA є її скутість в середині відеокарт від команії Nvidia. Тобто, використовуючи CUDA, розробник втрачає можливість розробки кросплатформеного програмного забезпечення.

1.6. Аналіз технології Vulkan

Vulkan [10] це фактично OpenGL наступного покоління. Початково розробка даного API відбувалась в рамках ініціативи вирішення наявних проблем API OpenGL. Якщо врахувати основні принципи, що були використані при розробці даного прикладного програмного інтерфейсу, то його застосування має принести значну перевагу продуктивності шляхом ефективнішого використання графічного процесору у порівнянні з OpenGL.

Основою метою створення API Vulkan є розумніше використання ресурсів та можливостей GPU та CPU. Цей програмний інтерфейс впроваджує більший контроль над роботою графічного процесора і зменшує навантаження на центральний процесор персонального комп'ютера. Що в свою чергу призводить до збільшення продуктивності при використанні API Vulkan.

До основних переваг API Vulkan відносяться:

- Використання проміжного бінарного формату представлення коду SPIR-V, що зменшує навантаження для розробників драйверів;
- Кросплатформеність;
- Підтримка систем з використанням принципів багатопоточності;
- Зменшення навантаження на центральний процесор;
- Збільшення пропускної здатності для обчислень на графічному процесорі та рендиренгу.

Висновки до розділу

Задачі тривимірної симуляції стають все більш актувальними в наш час. Відовідно способи реалізації рендерингу зображення в таких задачах стоять так само гостро. Тому, для коректної роботи симуляції просто необхідно обрати ефективний та продуктивний спосіб візуалізації зображення. Серед розмаїття різних способів, що дають на виході достатньо реалістичне зображення, по співвдношення реалістичність-швидкодія можна виділити трасування променів. Також найкращим по швидкодії способом трасування променів є апаратно-програмний спосіб. Це зумовлено архітектурою апаратного забезпечення, що використовується у даному способі. Серед апаратно-програмних способів особливо виділяється API Vulkan, що дозволяє створювати кросплатформене програмне забезпечення, що буде використовувати всі обчислювальні можливості графічних процесорів.

У розділі розглянуто основні визначення, способи та технології, які необхідні для візуалізації зображення у задачах тривимірної симуляції. Було представлено аналіз існуючих способів трасування променів.

2. АНАЛІЗ ЗАСОБІВ РЕАЛІЗАЦІЇ

2.1. Мова та набір засобів реалізації

Мова реалізації – це мова програмування, що використовується при розробці різноманітного програмного забезпечення. Мова програмування — це система позначень для написання комп’ютерних програм. Більшість мов програмування є текстовими формальними мовами, але вони також можуть бути графічними. Вони є різновидом комп'ютерної мови.

Опис мови програмування зазвичай розбивається на дві складові: синтаксис (форму) і семантику (значення), які зазвичай визначаються формальною мовою. Деякі мови визначені специфікаційним документом (наприклад, мова програмування C визначена стандартом ISO), тоді як інші мови мають домінуючу реалізацію, яка розглядається як еталон. Деякі мови мають обидва, причому базова мова, визначена стандартом, і розширення, взяті з домінуючої реалізації, є загальними.

C (Рис. 2.1.1)— це мова комп’ютерного програмування загального призначення. Вона була створений у 1970-х роках Деннісом Річі та залишається дуже широко використовуваною та впливовою. За дизайном, функції C чітко відображають можливості цільових ЦП. Дана мова широко використовується використання в операційних системах, драйверах пристроїв, стеках протоколів, хоча стає все менш популярною при розробці прикладного програмного забезпечення. C зазвичай використовується в комп’ютерних архітектурах, які варіюються від найбільших суперкомп’ютерів до найменших мікроконтролерів і вбудованих систем.

C — імперативна процедурна мова, яка підтримує структуроване програмування, область видимості лексичних змінних і рекурсію зі статичною системою типів. Вона була розроблена для забезпечення низькорівневого доступу до пам’яті, містить мовні конструкції, які ефективно відображаються на машинні інструкції, і все це з мінімальною підтримкою часу виконання. Незважаючи на свої низькорівневі можливості, мова була розроблена для заохочення кросплатформного програмування. Сумісну зі стандартами програму на C можна скомпільувати для широкого спектру комп’ютерних платформ і операційних систем з невеликими змінами у вихідному коді.

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

struct stats { int count; int sum; int sum_squares; };

void stats_update(struct stats * s, int x, bool reset) {
    if (s == NULL) return;
    if (reset) * s = (struct stats) { 0, 0, 0 };
    s->count += 1;
    s->sum += x;
    s->sum_squares += x * x;
}

double mean(int data[], size_t len) {
    struct stats s;
    for (int i = 0; i < len; ++i)
        stats_update(&s, data[i], i == 0);
    return ((double)s.sum) / ((double)s.count);
}

void main() {
    int data[] = { 1, 2, 3, 4, 5, 6 };
    printf("MEAN = %lf\n", mean(data, sizeof(data) / sizeof(data[0])));
}
```

Рисунок 2.1.1 – Приклад програмного коду, що був написаний мовою програмування C

C++ (Рис. 2.1.2) — мова програмування високого рівня загального призначення, створена датським комп'ютерним науковцем Б'ярне Страуструпом як розширення мови програмування C. Мова значно розширилася з часом, і сучасний C++ тепер має об'єктно-орієнтовані, загальні та функціональні функції на додаток до можливостей низькорівневого маніпулювання пам'яттю.

C++ було розроблено з урахуванням системного програмування та вбудованого програмного забезпечення з обмеженими ресурсами та великих систем, з продуктивністю, ефективністю та гнучкістю використання. C++ також виявився корисним у багатьох інших контекстах, головними перевагами якого є інфраструктура програмного забезпечення та програми з обмеженими ресурсами, включаючи настільні програми, відеоігри, сервери, критичні програми (наприклад, телефонні комутатори або космічні зонди).

```
#include <yarp/os/Network.h>
#include <yarp/os/Port.h>
#include <yarp/os/Bottle.h>
#include <yarp/os/Time.h>
#include <stdio.h>
using namespace yarp::os;
int main() {
    Network yarp;
    Port output;
    output.open("/sender");
    int top = 100;
    for (int i=1; i<=top; i++) {
        // prepare a message
        Bottle bot;
        bot.addString("testing");
        bot.addInt(i);
        bot.addString("of");
        bot.addInt(top);
        // send the message
        output.write(bot);
        printf("Sent message: %s\n", bot.toString().c_str());
        // wait a while
        Time::delay(1);
    }
    output.close();
    return 0;
}
```

Рисунок 2.1.2 – Приклад програмного коду, що був написаний мовою програмування C++

C# (Рис 2.1.3) — це багатопарадигмальна мова програмування загального призначення високого рівня. C# охоплює статичну типізацію, сильну типізацію, лексичну, імперативну, декларативну, функціональну, загальну, об’єктно-орієнтовану (на основі класу) та компонентно-орієнтовану дисципліни програмування.

Мова програмування C# була розроблена Андерсом Хейлсбергом із Microsoft у 2000 році та пізніше була затверджена як міжнародний стандарт Ecma (ECMA-334) у 2002 році та ISO/IEC (ISO/IEC 23270) у 2003 році. Microsoft представила C# разом із .NET Framework і Visual Studio, обидва з яких були закритими. У той час у Microsoft не було продуктів з відкритим кодом. Через чотири роки, у 2004 році, розпочався безкоштовний проект з відкритим вихідним кодом під назвою Mono, що надає кросплатформний компілятор і середовище виконання для мови програмування C#. Через десять років Microsoft випустила Visual Studio Code (редактор коду), Roslyn (компілятор) і уніфіковану платформу .NET (фреймворк програмного забезпечення), які підтримують C# і є безкоштовними, з відкритим кодом і кросплатформенними.

```
private void btnOpen_Click(object sender, EventArgs e)
{
    string file_name = "C:\\Users\\Owner\\Documents\\testData.txt";
    string tLine = "";

    System.IO.StreamWriter objWriter;
    objWriter = new System.IO.StreamWriter(file_name);

    string[] aryTest = new string[5];

    aryTest[0] = "Mary";
    aryTest[1] = "had";
    aryTest[2] = "a";
    aryTest[3] = "little";
    aryTest[4] = "one";

    for (int i = 0; i < 5; i++)
    {
        tLine = tLine + aryTest[i] + "\r\n";
        objWriter.WriteLine(aryTest[i]);
    }

    textBox1.Text = tLine;
    MessageBox.Show("Wrote File");
    objWriter.Close();
}
```

Рисунок 2.1.3 – Приклад програмного коду, що був написаний мовою програмування C#

Python (Рис. 2.1.4) — це мова програмування високого рівня загального призначення. На даний момент Python вважається однією з найпопулярніших мов програмування. Python динамічно типізується та збирає сміття. Вона підтримує кілька парадигм програмування, включаючи структуроване (зокрема процедурне), об’єктно-орієнтоване та функціональне програмування.

```
def log_message_delivery(func):
 def wrapper(*args, **kwargs):
   try:
     func(*args, **kwargs)
   except Exception as e:
     logger.exception(f"Failed to deliver message to: {e}")
   else:
     logger.info(f"Delivered message successful")
 return wrapper

@log_message_delivery
def send_email(email_addres):
 #logic related to email delivery
 print(email_addres)

@log_message_delivery
def send_sms(phone_number):
 #logic related to sms delivery
 print(phone_number)
```

Рисунок 2.1.4 – Приклад програмного коду, що був написаний мовою програмування Python

Засоби реалізації – це спеціалізоване програмне забезпечення, що призначене в першу чергу для спрощення процесу розробки цільового продукту розробниками. Засобом реалізації може виступати як і IDE так й інше вузько спеціалізоване програмне забезпечення, що направлене на створення та редагування додаткових чи допоміжних модулів чи об’єктів, що буде використовувати розроблений фінальний програмний продукт. До такого роду програмного забезпечення відносяться й засоби роботи з графічними файлами, тобто таке ПЗ, що створює чи редагує файли, що містить у собі опис

двовимірного векторного чи растрового зображення, опис тривимірних моделей, об’єктів чи параметри сцени. IDE в свою чергу це – інтегроване середовище розробки. Іншими словами це комплексне програмне рішення, що направлене на розробку та відлагодження програмного забезпечення.

Visual Studio (Рис. 2.1.5) — це інтегроване середовище розробки (IDE) від Microsoft. Він використовується для розробки прикладного програмного забезпечення, включаючи веб-сайти, веб-додатки, веб-сервіси та мобільні додатки. Visual Studio використовує платформи розробки програмного забезпечення Microsoft, такі як Windows API, Windows Forms, Windows Presentation Foundation, Windows Store і Microsoft Silverlight. Він може створити як рідний, так і керований код.

Visual Studio містить редактор коду, який підтримує IntelliSense (компонент завершення коду), а також рефакторинг коду. Вбудований налагоджувач працює як налагоджувач на рівні джерела, так і на рівні машини. Інші вбудовані інструменти включають профайлер коду, конструктор для створення додатків графічного інтерфейсу користувача, веб-дизайнер, конструктор класів і конструктор схем бази даних. Він приймає плагіни, які розширюють функціональні можливості майже на всіх рівнях, включаючи додавання підтримки систем контролю джерел (наприклад, Subversion і Git) і додавання нових наборів інструментів, таких як редактори та візуальні дизайнери для доменно-спеціальних мов або наборів інструментів для інших аспектів розробки програмного забезпечення. життєвий цикл (наприклад, клієнт Azure DevOps: Team Explorer).

Visual Studio підтримує 36 різних мов програмування та дозволяє редактору коду та налагоджувачу підтримувати майже будь-яку мову

програмування, якщо існує спеціальна служба для цієї мови. Вбудовані мови включають C, C++, C++/CLI, Visual Basic .NET, C#, F#, JavaScript, TypeScript, XML, XSLT, HTML і CSS. Підтримка інших мов, таких як Python, Ruby, Node.js і M, серед інших, доступна через плагіни.

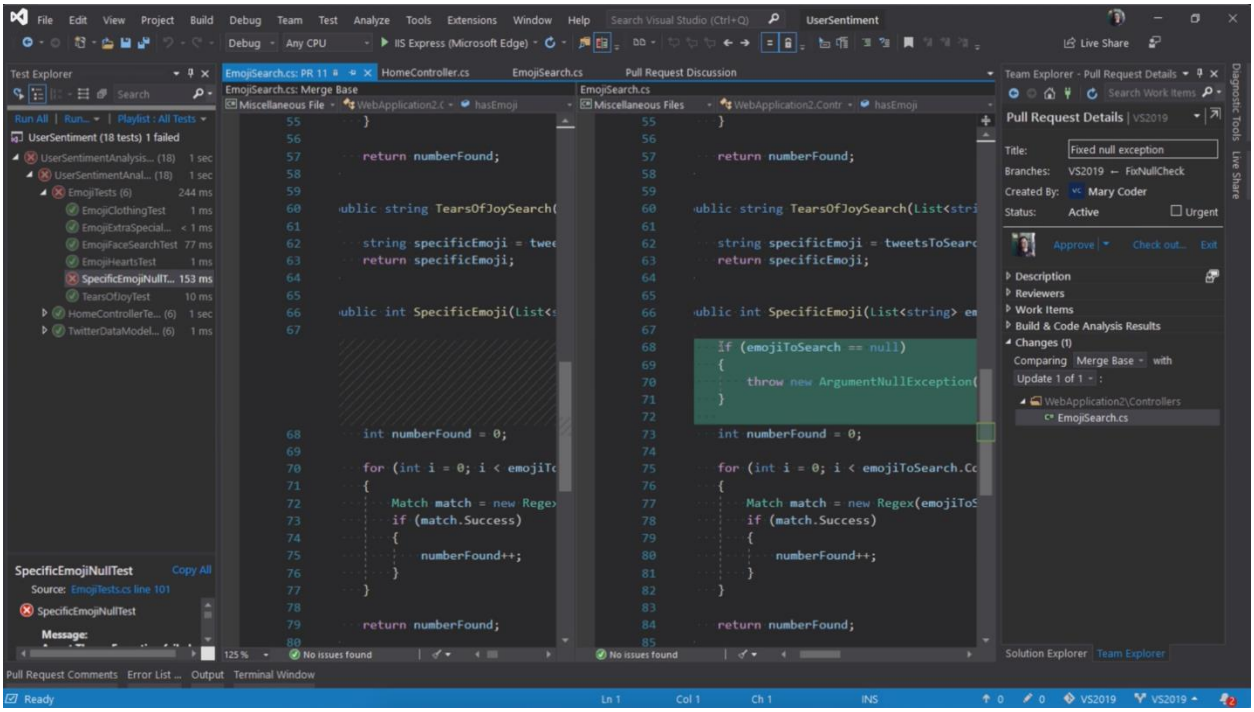


Рисунок 2.1.5 – Інтегроване середовище розробки Visual Studio

PyCharm (Рис. 2.1.6) — це інтегроване середовище розробки (IDE), яке використовується в комп'ютерному програмуванні, зокрема для мови програмування Python. Він розроблений чеською компанією JetBrains. Він забезпечує аналіз коду, графічний налагоджувач, вбудований блок-тестер, інтеграцію з системами контролю версій (VCS), а також підтримує веб-розробку за допомогою Django. PyCharm є кросплатформним із версіями для Windows, macOS і Linux. Версія для спільноти випущена за ліцензією apache, а також є професійна версія з додатковими функціями, випущена за ліцензією, що фінансується передплатою, а також освітня версія.

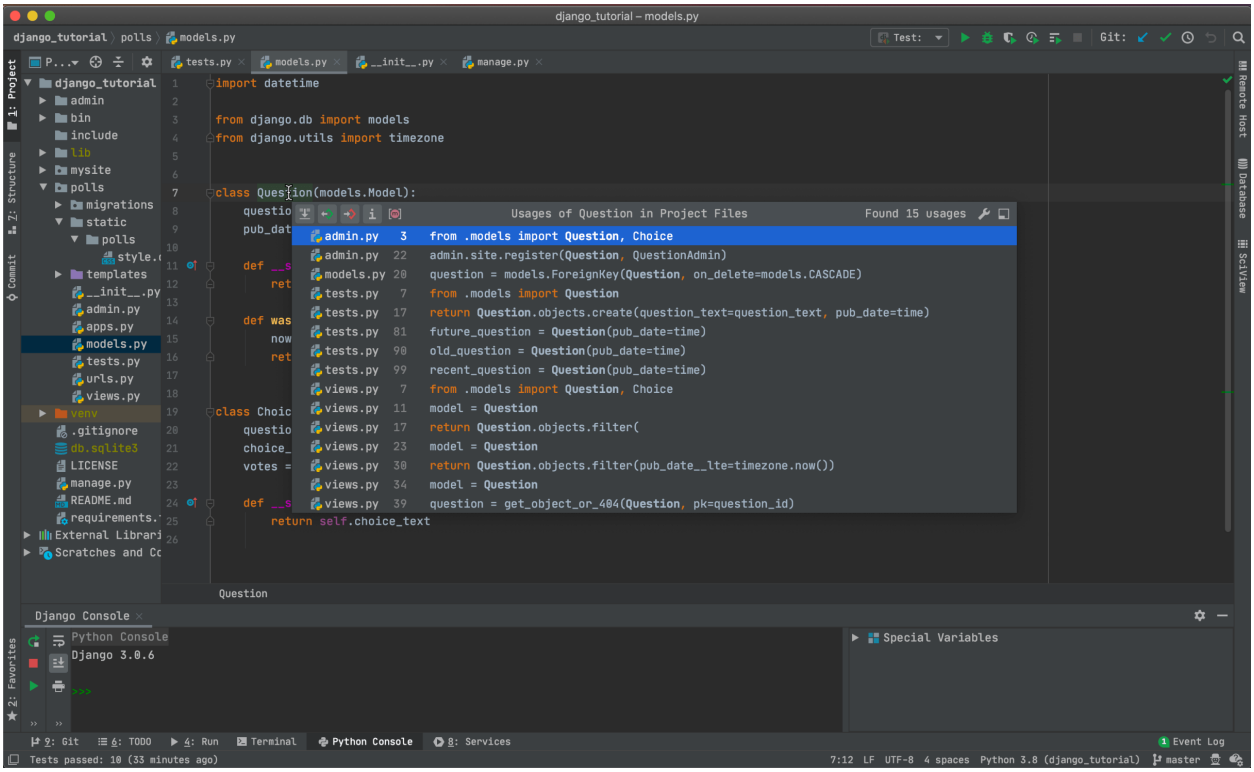


Рисунок 2.1.6 – Інтегроване середовище розробки PyCharm

Eclipse — це інтегроване середовище розробки (IDE). Він містить базову робочу область і розширювану систему плагінів для налаштування середовища. Це друга за популярністю IDE для розробки Java. Eclipse здебільшого написаний на Java, і його основне використання — це розробка додатків Java , але його також можна використовувати для розробки додатків на інших мовах програмування за допомогою плагінів, включаючи Ada, ABAP, C, C++, C#, Clojure, COBOL, D, Erlang, Fortran, Groovy, Haskell, JavaScript, Julia, Lasso, Lua, NATURAL, Perl, PHP, Prolog, Python, R, Ruby (включаючи структуру Ruby on Rails), Rust, Scala, і Схема. Eclipse також можна використовувати для розробки документів за допомогою LaTeX (через плагін TeXlipse) і пакетів для програмного забезпечення Mathematica.

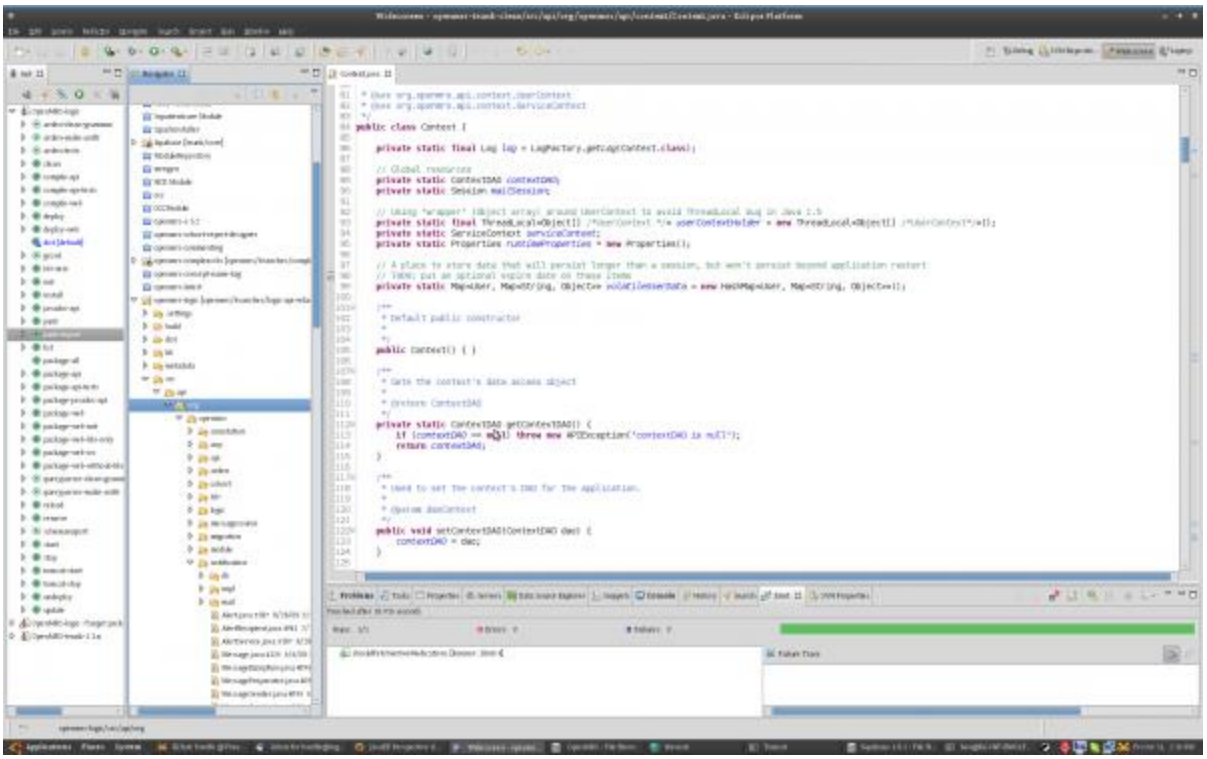


Рисунок 2.1.7 – Інтегроване середовище розробки Eclipse

Autodesk Maya (Рис. 2.1.8) — це тривимірна програма комп’ютерної графіки, яка працює в Windows, macOS і Linux, спочатку розроблена Alias, а зараз належить і розробляється компанією Autodesk. Даний редактор використовується для створення ресурсів для інтерактивних 3D-додатків (включаючи відеоігри), анімаційних фільмів, серіалів і візуальних ефектів.

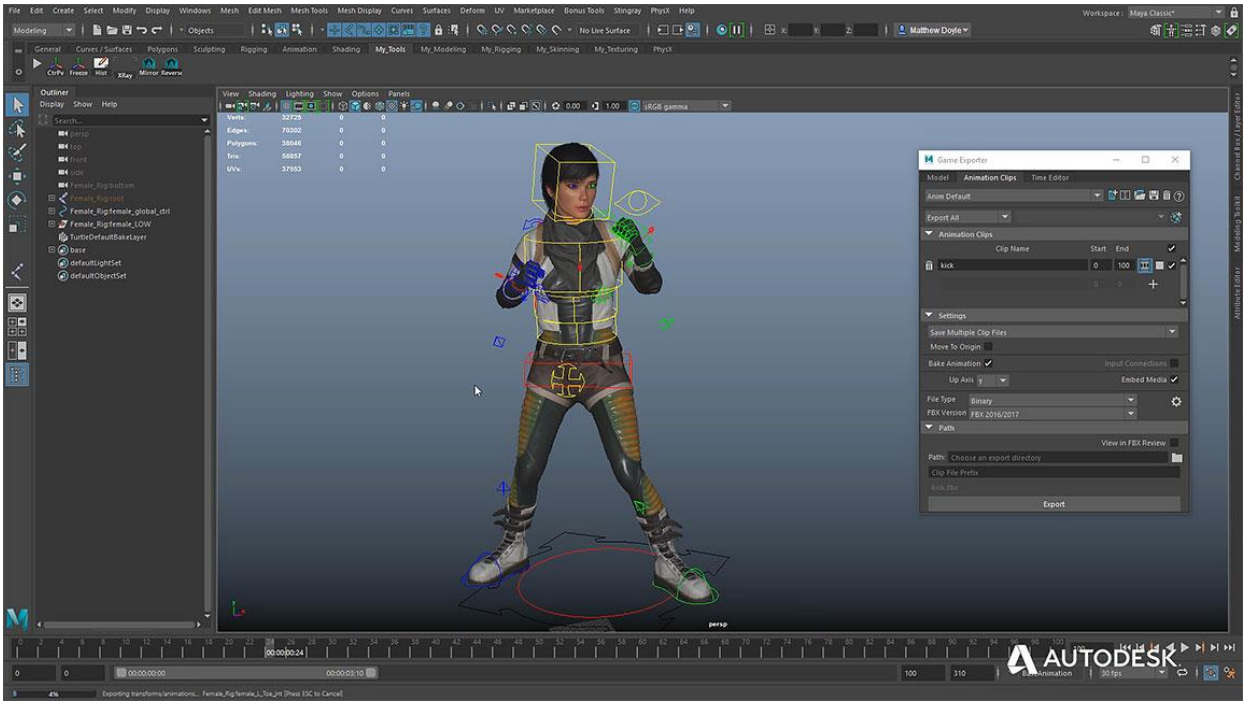


Рисунок 2.1.8 – Створення тривимірної моделі в Autodesk Maya

Autodesk 3ds Max (Рис. 2.1.9) — це професійна програма комп’ютерної 3D-графіки для створення 3D-анімацій, моделей і зображень. Вона була розроблене і створене Autodesk Media and Entertainment. Має можливості моделювання та гнучку архітектуру плагінів і використовуватися на платформі Microsoft Windows. Дане прикладне програмне забезпечення часто використовується розробниками відеоігор, багатьма телевізійними рекламними студіями та студіями архітектурної візуалізації. Також використовується для кіноефектів і попередньої візуалізації. 3ds Max містить шейдери (такі як оклюзія навколишнього середовища та підповерхневе розсіювання), динамічне моделювання, системи частинок, радіовипромінюваність, створення та візуалізація нормальної карти, глобальне освітлення, настроюваний інтерфейс користувача та власну мову сценаріїв.

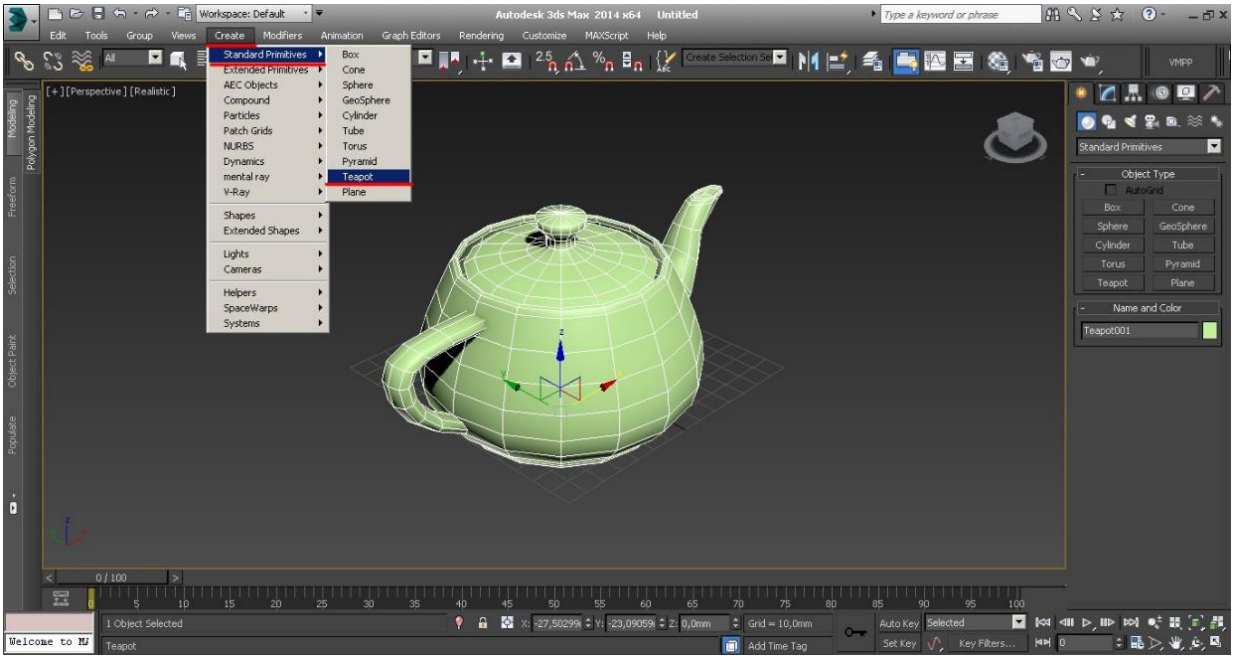


Рисунок 2.1.9 – Створення тривимірної моделі в Autodesk 3ds Max

Blender (Рис. 2.1109) — це безкоштовний набір інструментів 3D-комп'ютерної графіки з відкритим кодом, який використовується для створення анімаційних фільмів, візуальних ефектів, мистецтва, 3D-друкованих моделей, анімованої графіки, інтерактивних 3D-додатків, віртуальної реальності та відеоігор. Функції Blender включають 3D-моделювання, ультрафіолетове відображення, текстурування, цифрове малювання, редагування растрової графіки, такелаж і скінінг, симуляцію рідини та диму, моделювання частинок, моделювання м'якого тіла, скульптуру, анімацію, переміщення сірників, рендеринг, анімаційну графіку, редагування відео та компонування.

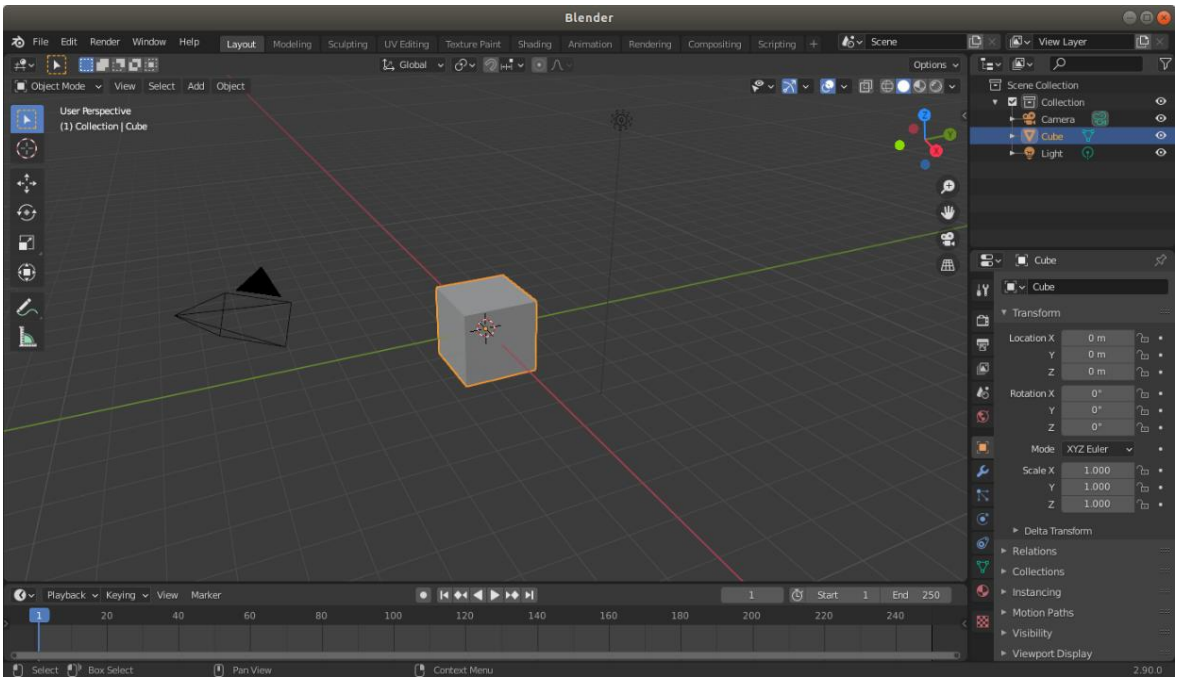


Рисунок 2.1.10 – Створення тривимірної моделі в Blender

Paint 3D (Рис. 2.1.11) — це не професійне програмне забезпечення, що призначене для створення та редагування растрової та тривимірної комп’ютерної графіки та є оновленою версією Microsoft Paint. Це одна з кількох програм для 3D-моделювання та друку що підтримує 3MF формат.

Paint 3D поєднує в собі функції програм Microsoft Paint і 3D Builder, щоб поєднати легкий гібридний досвід редагування 2D-3D, який дозволяє користувачам витягувати різноманітні форми з програми, свого персонального комп’ютера, і сервіс Microsoft OneDrive. Хоч Paint 3D і не є професійним програмним забезпеченням, але є дуже корисним коли справа вимагає швидкого редагування не значних аспектів тривимірної моделі, або ж створення простого 3D об’єкту сцени.

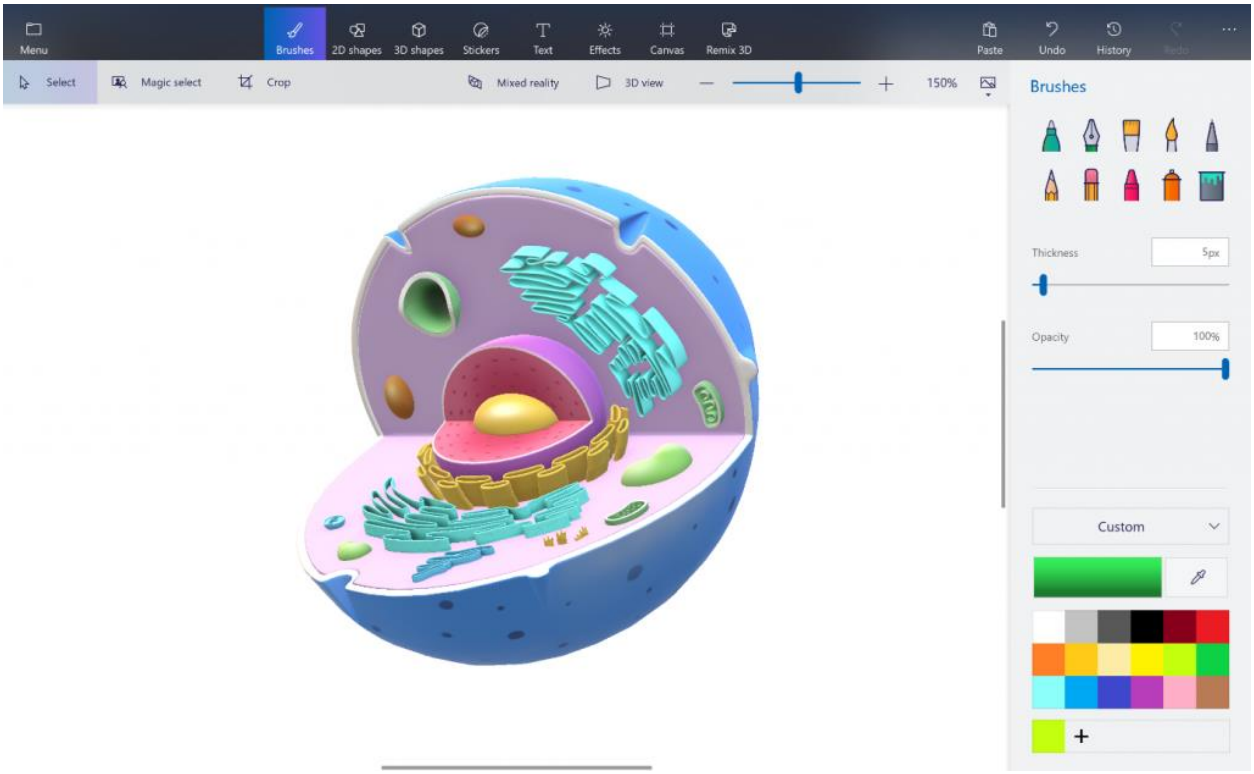


Рисунок 2.1.11 – Створення тривимірної моделі в Paint 3D

2.2. Огляд базового способу трасування променів

В основі алгоритму трасування променів лежить модель (Рис. 2.2.1), що випускатиме один промінь на піксель. До прикладу, якщо розмір зображення 1280 на 720 пікселів, то після завершення трасування всього буде знято 921600 променів. Кожен з цих променів починається біля умовного глядача сцени й закінчується на найближчому перетині випущеного променя з об’єктом тривимірної сцени. Розташування глядача переважно визначається у вхідному файлі, що задає опис та місце розташування всім іншим об’єктам сцени. Для того, аби визначити скільки світла падає на точку перетину та який колір буде отримано по закінченню трасування застосовується модель освітлення. Модель освітлення — це рівняння, що використовується для обчислення

інтенсивності світла, що умовний глядач має спостерігати в даній точці на поверхні об’єкта.

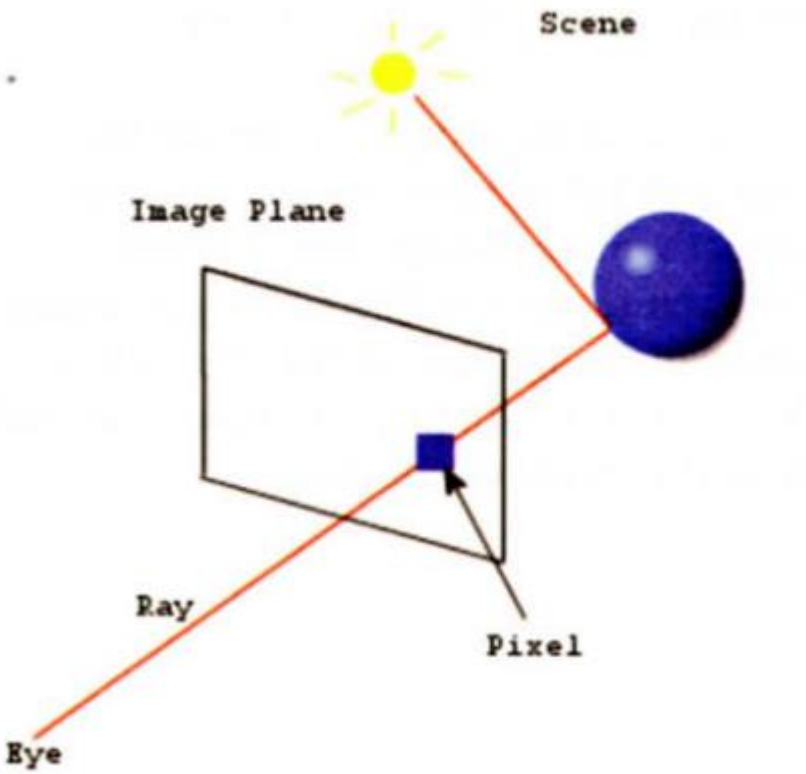


Рисунок 2.2.1 – Схема базової моделі трасування променів

Об’єкти у межах сцени мають набір різних властивостей, що визначають місце розташування об’єкта в сцені, його колір, здатність до світловідбивання та заломлення світла (Рис. 2.2.2). Об’єктами можуть різні геометричні фігури, такі як сфери, трикутники , кільця, циліндри тощо. Будь-який з цих об’єктів також може бути джерелом світла та випромінювати світло на всій своїй поверхності, або ж її частині. Для стандартної моделі освітлення використовуються точкові джерела світла, тобто світло, що надходить з однієї точки сцени.

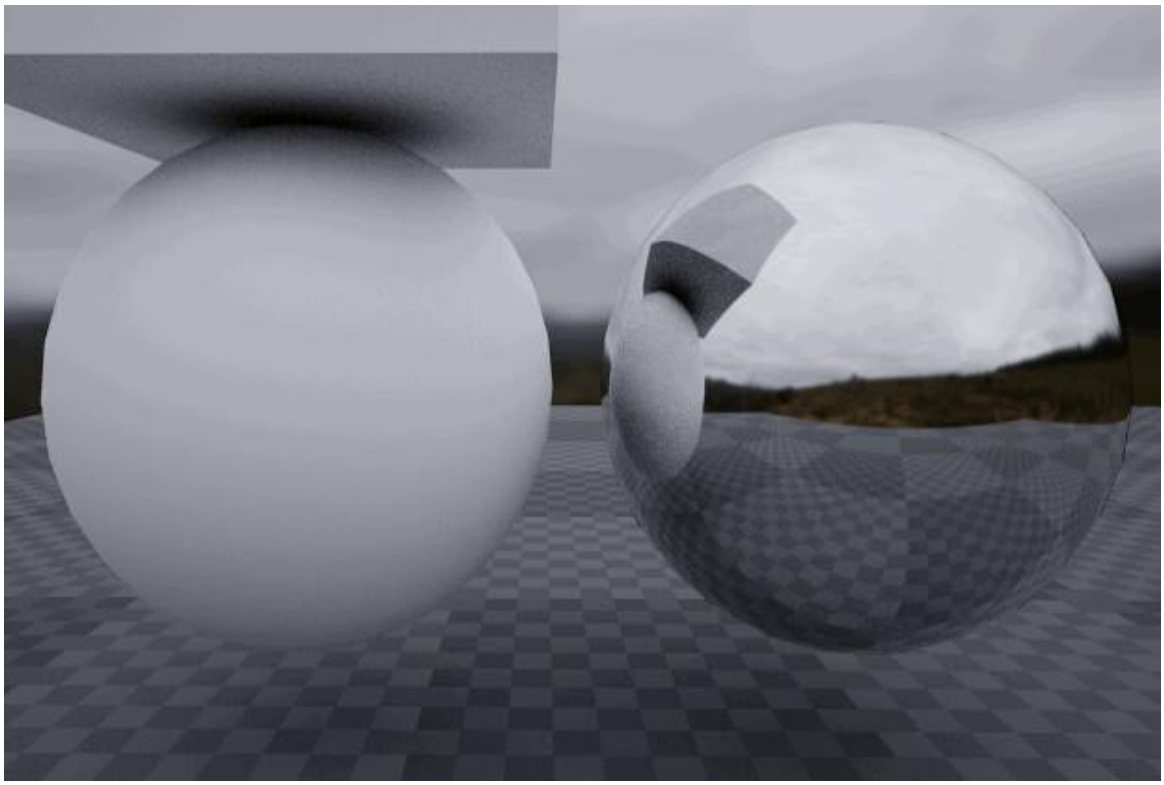


Рисунок 2.2.2 – Приклад властивостей об’єктів

Параметричні рівняння для лінії в тривимірному просторі використовуються для розрахунку близького відтінку об’єкта від умовного спостерігача.

P_0 (x_0, y_0, z_0) – це розташування точки глядача в початковій точці променя. P_1 (x_1, y_1, z_1) – це точка на площині зображення. P (x, y, z) – це будь-яка точка на прямій, визначеній P_0 та P_1. ($x_1 – x_0$) – це x компонент вектора. Так само і для y та z. Тож, промінь можна представити таким вектором: ($x_1 – x_0$), ($y_1 – y_0$), ($z_1 – z_0$).

$$x = x_0 + t * (x_1 – x_0)$$

$$y = y_0 + t * (y_1 – y_0)$$

$z = z_0 + t * (z_1 - z_0)$

Тепер, коли знайдено розрахунок того, що умовний глядач бачитиме в конкретному пікселі, застосовується режим освітлення. Використаний режим освітлення визначає колір пікселя.

Дифузний матеріал - це тьмяний матеріал, що нагадує крейду. У точці перетину з об’єктом створюється вектор від перетину до світла на сцені. Так утворюється світловий вектор L. N в свою чергу є нормаллю поверхні в цьому перетині. Нормаль перпендикулярна до поверхні. Формула для розрахунку розсіяної складової моделі локального освітлення має такий вигляд :

$Diffuse = K_{diffuse} * Color_{diffuse} * Cos\theta$

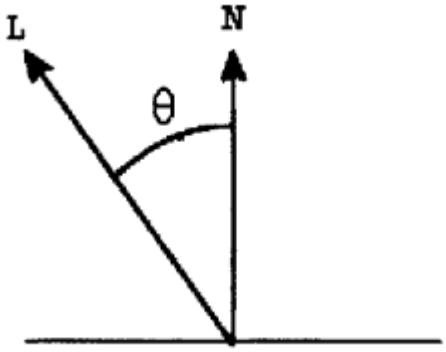


Рисунок 2.2.3 – Світловий вектор L та нормаль поверхі N

Колір відблиску (Рис. 2.2.4) залежить від умовного спостерігача сцени. Чим ближче вектор відбиття R спрямований до сопстерігача, тим яскравішим буде отриманий колір пікселя. Іншими словами, колір відблиску має більшу інтенсивність освітлення, якщо світло відбивається назад до глядача сцени. Формула для розрахунку дзеркальної складової моделі локального освітлення має такий вигляд:

$Specular = K_{Specular} * Color_{Specular} * Cos\varphi$

$K_{diffuse}$ та $Color_{diffuse}$ є попередньо визначеними вхідними даними трасувальника променів. Вони описують дифузні властивості конкретного об'єкта тривимірної сцени. Кут між L і N дорівнює θ, який розраховується та змінюється відповідно до розташування світла в сцені. Це надає об’єкту сцени затінений вигляд залежно від розташування джерела світла.

$K_{Specular}$, $Color_{Specular}$ і блискучість є вхідними даними, що описують об'єкт. Показник блиску впливає на дзеркальну пляму на об'єкті. Чим більший відблиск, тим концентрованіша пляма. φ — це кут між вектором нормалі N і вектором ока умовного спостерігача.

Пікселі будуть у кольоровому просторі «червоний-зелений-синій», відомому як (R, G, B). Кожен компонент RGB матиме діапазон [0,0 - 1,0]. Білий буде (1, 1, 1) і чорний (0,0,0). Формула моделі відблиску виглядає так:

$Pixel_{ColorR} = Diffused_R + Specular_R$

$Pixel_{ColorG} = Diffused_G + Specular_G$

$Pixel_{ColorB} = Diffused_B + Specular_B$

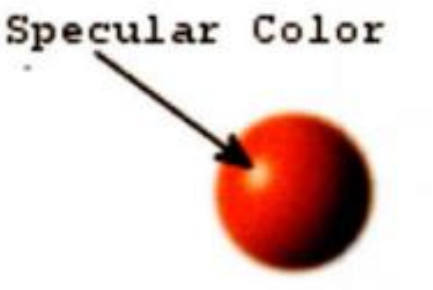


Рисунок 2.2.4 – Колір відблиску на сфері

Згладжування (Рис. 2.2.5) — це техніка, що використовується для згладжування зображення шляхом змішування кольору гострого краю об’єкта з кольорами навколо краю, роблячи об’єкт більш гладким. До прикладу, чорна поверхня, що перетинається з білою поверхнею, у точках перетину кольорів змішуватиметься й перетворюватиметься на сіруватий колір. Застосувати цю техніку для трасування променів просто. Для початку потрібно взяти піксель і розділити його на субпікселі та випустити промені із субпікселів. Додати всі субкольори та поділити їх на кількість субпікселів. Це в свою чергу дасть середнє значення кольору для всього пікселя. Так субпромені не потраплять на одну й ту ж саму точку, деякі із них потраплять на чорну поверхню, а інші — на білу. Тоді усереднення кольорів дасть сірий колір, середній між краєм об’єкту та кольором навколо краю.

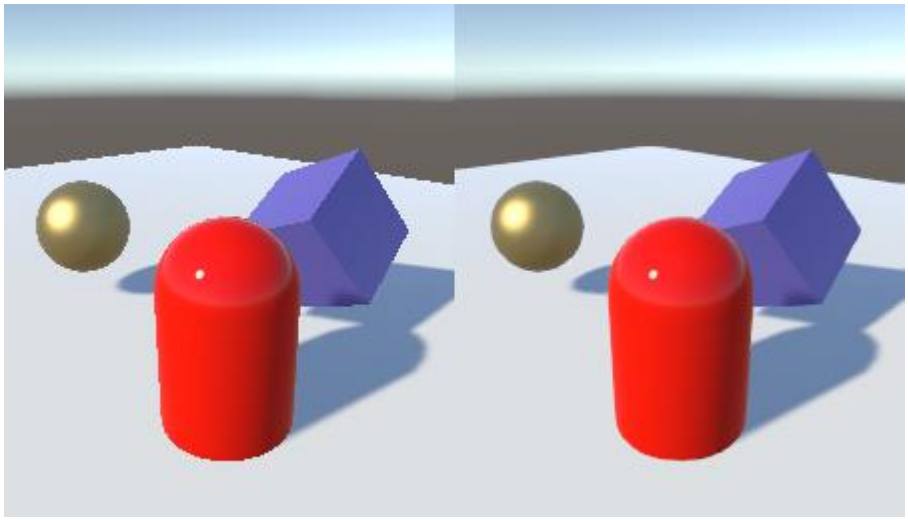


Рисунок 2.2.5 – Приклад використання згладжування об’єктів

Нормаль – перпендикуляр до поверхні в певній точці цієї поверхні. Якщо трикутник має лише одну нормаль для всіх точок трикутника, то цей трикутник буде ідеально плоским. З однією нормаллю на трикутник об’єкт, що складається з трикутників, стане розбитим на плоскі частини (Рис. 2.2.6). Мета

інтерполяції полягає в тому, щоб мати дещо іншу нормаль для кожної точки трикутника, що робить об’єкт викривленим.

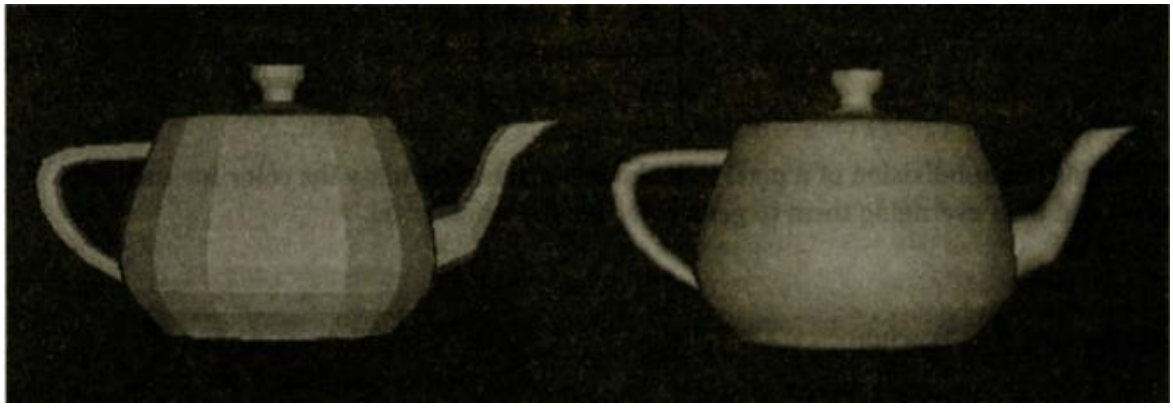


Рисунок 2.2.6 – Об’єкт візуалізований без інтерполяції (зліва) та з інтерполяцією (зправа)

Ця нова нормаль, N, буде обчислюватись за допомогоб трьох інших нормалей, Na, Nb і Nc, що представляють нормалі трьох точок трикутника A, B і C відповідно. Нормалі трикутника є попередньо визначеними для трасувальника променів.

Глобальне освітлення дасть більш реалістичне зображення. Воно буде враховувати все світло наявне в сцені, пряме та непряме, для того, аби сформувати кращу модель освітлення на поверхні. При локальному освітленні один промінь спрямовується на кожне світло, щоб обчислити, скільки світла падатиме на цю поверхню. При глобальному освітленні після обчислення перетину променя з об’єктом викидаються інші промені в різних напрямках, щоб створити світло, що падає на цю точку. Для того, аби створити максимально реалістичне зображення, слід протестувати всі можливі напрямки світла. Це фактично неможливо, оскільки на будь-яку окрему точку

падає нескінченна кількість напрямків світла. Замість цього пробні промені кидаються від об’єкту для створення моделі освітлення .

Промінь може випадковим чином відбиватися від об’єкта, поки не досягне світла. Якщо він ніколи не досягає максимально дозволеного відскоку світла, він викидається з остаточного обчислення кольору пікселя. Якщо зразок потрапляє на світло безпосередньо, у розрахунок береться повна інтенсивність світла. Якщо він не потрапляє на світло прямо, після кожного відбиття інтенсивність світла зменшується на коефіцієнт розсіяності компонента об’єкта, від якого воно відбивається. У глобальному освітленні вибірковий промінь займає місце вектора світла L у розрахунках кольору пікселя. Формула кольору пікселя тепер змінюється на наступну:

$$Pixel_{color} = \frac{1}{N} * \sum_{i=1}^{\#samples} (Diffused_i + Specular_i) * gl(samples)$$

Створення променів є важливою частиною алгоритму. Зразкові промені вироблятимуть світло. Тому промені поганої вибірки призведуть до поганого освітлення. Якщо пробний промінь ніколи не потрапляє на світло, він викидається з розрахунку. Промені повинні бути спрямовані вбік від поверхні, з якої вони виходять. Пробний промінь повинен мати кут з нормаллю між 0 і 90 градусами та мати можливість досягати 360 градусів навколо нормалі. Це зображує півкулю, нормаль поверхні якої проходить прямо через її верхній «Північний полюс» як зображено на рисунку 2.2.7.

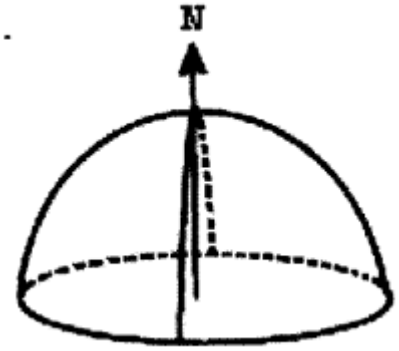


Рисунок 2.2.7 – Півкуля для пробного променя

Сферичну систему координат можна використовувати для створення випадкових вибіркових променів. Точка P на півкулі повинна бути випадковою. Це можна зробити шляхом рандомізації кутів φ і θ. Радіус r можна підтримувати постійним на рівні 1. P потрібно перетворити на його компоненти x, y і z за такими формулами:

$$x = r \sin\varphi \cos\theta$$

$$y = r \sin\varphi \cos\theta$$

$$z = r \cos\varphi$$

P орієнтується на джерело. Отже, P потрібно трансформувати, щоб воно було орієнтоване відносно поверхні та її нормалі. Для цього робиться ортонормований базис з нормаллю поверхні. Ортонормований базис (ОНБ) — це набір взаємно перпендикулярних векторів одиничної довжини. Найвідомішим ОНБ є система координат xyz, природний базис. Після створення ортонормального базису з нормаллю поверхні P можна перетворити на P'. P' тепер орієнтовано на нову основу. Щоб завершити цей процес, створюється вектор від точки перетину на поверхні до P' і формується новий

промінь вибірки. Новий вибірковий промінь буде використано замість світлового променя у формулі освітлення.

2.3. Апаратно-програмні способи реалізації

Першою відомою спробою роботи з трасування променів на програмованому графічному обладнанні є «Ray Engine» Натана Карра, що був представлений у 2002 році. У Ray Engine центральний процесор виконує обхід променя на структурі прискорення та генерує список трикутників-кандидатів для обчислення перетину променів із трикутниками. Потім графічний процесор обчислює перетини променів і трикутників для цих трикутників шляхом передачі даних трикутника з пам’яті ЦП. Оскільки для процесу перетину променів і трикутників на графічному апаратному забезпеченні не потрібно знати базове представлення структури даних прискорення, обчислення на графічному апаратному забезпеченні стає відносно простим. Основна причина такої конструкції системи полягає в тому, що раннє програмоване графічне обладнання мало серйозні обмеження, такі як кількість інструкцій, точність обчислень і обсяг пам’яті, що ускладнювало реалізацію обходу променів на графічному обладнанні. На рисунку 2.3.1 показана діаграма алгоритму Ray Engine.

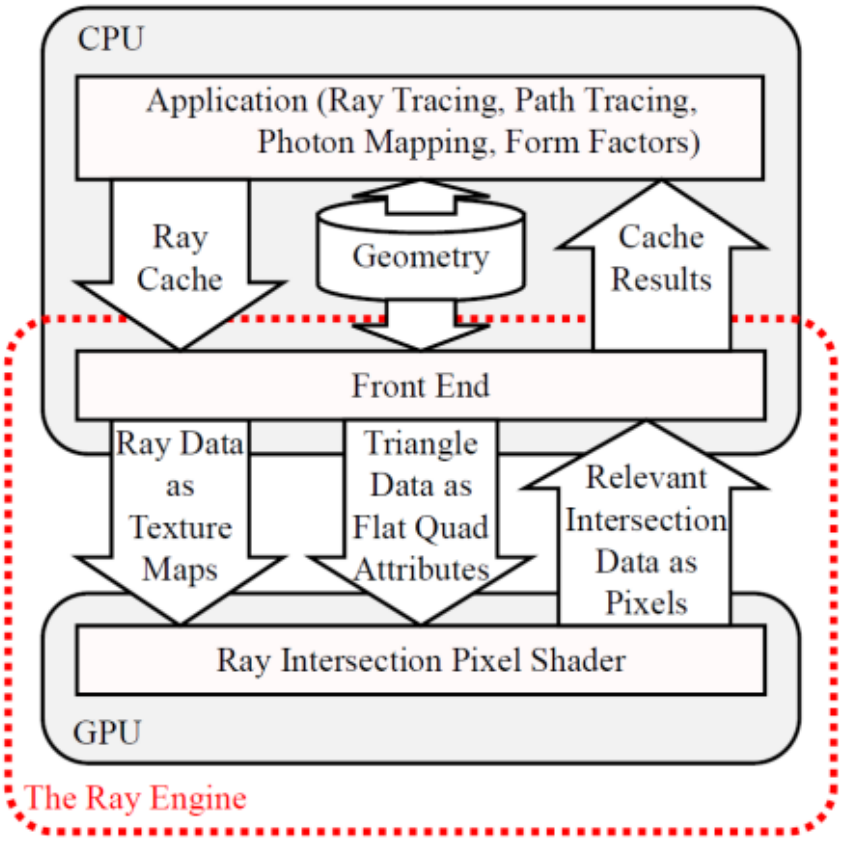


Рисунок 2.3.1 – діаграма Ray Engine

Основний недолік цього підходу полягає в тому, що існує високий обсяг передачі даних між центральними процесорами та графічним обладнанням, що є відносно не ефективним. Наприклад, цей підхід вимагає передачі даних трикутників від центрального процесору до графічного процесору та даних про перетини від графічного процесору назад до центрального. Так, через такий спосіб роботи алгоритму Натан Карр дійшов висновоку, що продуктивність змішаного типу трасувальника променів, тобто такого, що одночасно використовує потужності графічного та центрального процесорів, є не достатньо ефективною реалізацією трасувальника променів порівняно з

реалізацією на ЦП, навіть якщо ми припустимо нескінченно швидкі обчислення на графічному обладнанні.

На відміну від загальної продуктивності трасування променів, продуктивність перетинів променів і трикутників на графічному обладнанні значно вища, ніж на процесорах. Графічна апаратна реалізація перетину променів із трикутником досягла 114 млн перетинів за секунду на ATI Radeon R200 (тактова частота ядра становить 275 МГц), тоді як найшвидший на той час перетин променів із трикутником ЦП досягнуто лише від 20 млн до 40 млн перетинів за секунду на Pentium III (тактова частота ядра – 800MHz). Так відбувається головним чином тому, що ATI Radeon R200 має чотири незалежні процесори для пікселів. Це дає можливість паралельно обробляти чотири перетини променів і трикутників.

Оскільки ефективне використання структури прискорення є ключем до покращення продуктивності трасування променів, більшість робіт, що пов’язана із дослідженням трасування променів на графічному апаратному забезпеченні, зосереджено на обрахуванні променів на графічному апаратному забезпеченні.

Перший повний трасувальник променів, що включає обрахування променів на графічному апаратному забезпечені, використовував рівномірну сітку. Рівномірна сітка розділяє обмежувальну рамку всієї сцени на вокселі однакового розміру, які є прямокутними підпросторами обмежувальної рамки. Кожен воксель містить список трикутників, які накладаються на нього самого. В якості ініціалізації обходу променя обчислюється перетин між променем і обмежувальною рамкою сцени. На основі перетину можна знайти воксель, де промінь входить у рівномірну сітку. Після того як був знайдений початковий

воксель, промінь переходить до наступного вокселя залежно від напрямку променя. Якщо воксель містить будь-які трикутники, виконується перетин променів із трикутником, щоб отримати фактичні точки перетину. Оскільки перетини променів і трикутників виконуються лише для вокселів, які пронизують промінь, рівномірна сітка зменшує кількість перетинів променів і трикутників.

Так Тімоті Перселл запропонував перший алгоритм трасування променів, який повністю працює на графічному обладнанні. Він розглядає програмоване графічне обладнання як потоковий процесор. Потоковий процесор — це модель процесора, яка використовується в мовах потокового програмування, яка використовує потік елементів як вхідні та вихідні дані. Ці елементи обробляються ядрами паралельно. Оскільки ядро не відрізняється між різними елементами в одному потоці, паралельна обробка в потоці стає достатньо легкою. Програмоване графічне апаратне забезпечення можна розглядати як потоковий процесор, оскільки воно виконує ту ж саму програму вершин/фрагментів на наборі вершин/фрагментів. Пізніше ця точка зору була перетворена в мови потокового програмування на графічному обладнанні.

Обхід променя на рівномірній сітці можна легко відобразити на потоковому процесорі. Для обходу променя рівномірної сітки потрібно знати лише поточний воксельний індекс, початок променя та напрямок променя, щоб обчислити наступний воксель. Вхідними потоками є поточні воксельні індекси, джерела променів і напрямки променів. Ядро, що виконує один крок обходу променя в рівномірній сітці, виводить наступні воксельні індекси як вихідний потік. На рисунку 2.3.2 зображено схематичне представлення алгоритму потокового трасування променів з використанням рівномірної сітки.

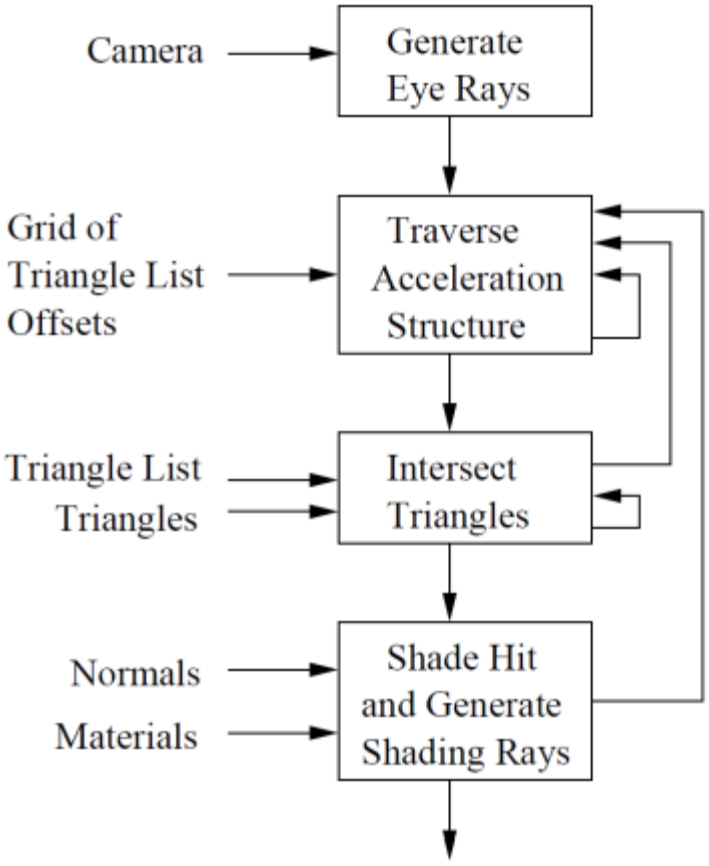


Рисунок 2.3.2 – Схематичне представлення алгоритму потокового трасування променів з використанням рівномірної сітки.

Завдяки цій моделі потокової обробки весь процес трасування променів може бути безпосередньо відображено на програмованому графічному обладнанні без частої передачі даних між центральним процесором і графічним процесором. Хоча побудова рівномірної сітки все ще виконується на центральному процесорі. Цей алгоритм, тобто побудова структури прискорення на процесорі та виконання обходу променів на графічному обладнанні, широко використовувався в багатьох інших трасувальниках променів на графічному апаратному забезпечені. Проте уніфікована структура даних сітки ефективна лише для рівномірного розподілу трикутників, тому

вона погано працює з нерегулярним розподілом трикутників. Ця проблема відома як проблема чайника на стадіоні. Це стосується того факту, що один воксель міститиме всі трикутники чайника, якщо ми створимо однорідну сітку над стадіоном. Оскільки ми обчислюємо перетини променів і трикутників у кожному вокселі, кількість перетинів променів і трикутників для чайника не зменшується.

KD-дерево – структура прискорення, яка може уникнути проблеми рівномірної сітки. За допомогою kD-дерева сцена розбивається на невеликі області за ієрархією площин. Кожен нелистовий вузол kD-дерева зберігає площину, яка розділяє простір на дві половини. Цей розділ продовжується, доки не буде виконано певний критерій (наприклад, кількість трикутників на листковий вузол). Нарешті, кожен листовий вузол зберігає список трикутників, які перекриваються з ним самим.

Обхід променя kD-дерева може бути ефективно реалізований за допомогою локального стека. На жаль, напряму перенести цей алгоритм на графічне обладнання неможливо, оскільки графічне обладнання не має локального стеку для кожного елемента (вершини чи пікселя). Різні розробники експериментували з емуляцією локального стека на графічному обладнанні за допомогою програмованих функцій, але це виявилось неефективно для трасування променів через витрати обчислювальних можливостей на емуляцію. Щоб уникнути використання стека в обході променя було запропоновано два нові безстекові методи обходу kD-дерева, kD-restart і kD-backtrack. В обох методах ми підтримуємо інтервал променя під час обходу променя. Метод kD-restart просто перезапускає обхід променя з кореня, коли промінь виходить із поточного листкового вузла. Промінь не відвідує той самий вузол, тому що інтервал променя зменшено, коли промінь

виходить із вузла. Якщо n є кількістю вузлів, найгірша вартість стає O(n log(n)), порівняно з O(n) при обході kD-дерева на основі стека. Додатковий O(log(n)) походить від того факту, що ми можемо очікувати, що кількість листкових вузлів, відвіданих променем, дорівнює O(log(n)).

Спосіб kD-backtrack зменшує цю додаткову вартість, використовуючи зворотне відстеження до початкового вузла з додатковим батьківським покажчиком на кожен вузол. Вузли, що надсилаються в стек, завжди є дочірніми вузлами вузлів-предків у стековому обході kD-дерева. Таким чином, повертаючись до батьківських вузлів, так можна отримати всі вузли в стеку без явного збереження їх у стеку. Метод kD-backtrack підтримує вартість найгіршого випадку до O(n). Однак, зворотне відстеження до батьківських вузлів спричиняє додаткові витрати. Крім того, кожен вузол повинен зберігати покажчик на свій батьківський вузол, що збільшує споживання пам'яті.

Незалежно від додаткових накладних витрат від стандартного обходу kD-дерева, кінцева продуктивність у 8 разів швидша, ніж метод рівномірної сітки зі сценами, як-от чайник на стадіоні.

Висновки до розділу

У розділі розглянуто основні мови та засоби реалізації, що використовуються при розробці прикладного програмного забезпечення пов’язаного на роботу із комп’ютерною графікою. Також було представлено аналіз базового алгоритму трасування променів та основних базових способів апаратно-програмної реалізації трасування променів з використанням графічного апаратного забезпечення. Серед усіх представлених в розділі мов реалізації слід виділити C++. Дана мова реалізації є швидкою та досить універсальною, що зумовлює високий рівень продуктивності прикладного

програмного забезпечення, яке було розроблене з використанням даної мови. Задля зручнішої роботи з дизайном об’єктів симуляції, розробнику слід використовувати й спеціалізовані програмні засоби. Найкращим програмним засобом такого характеру є Blender. Blender має достатньо можливостей для комфортного створення та редагування об’єктів тривимірної сцени. Також окремо слід виділити Paint3D. І хоч цей програмний засіб поступається за своїми можливості Blender, але має куди більш вагому перевагу для не професіоналів – він є стандартною утилітою операційної системи Microsoft Windows 10. Тому, коли розробнику потрібно створити відносно простий об’єкт або відредагувати який незначний аспект тривимірної моделі і розробник використовує Microsoft Windows, то простіше буде використати стандартний програмний засіб операційної системи.

При будь-якому досліджені чи покращені існуючої технології слід відштовхуватись від уже існуючих напрацювань. Так в розділі представлені в базовий принцип трасування променів та способи апаратно-програмної реалізації трасування променів є необхідними.

3. ОПИС РЕАЛІЗОВАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Вибір програмних засобів та технологій розробки

Задля реалізації поставленої задачі використовувалась мова програмування C++. Дана мова реалізації є достатньо швидкою та, за рахунок своєї популярності, має широкий спектр різноманітних бібліотек, підтримуваних прикладних програмних інтерфейсів та інших інструментів, що дозволяють працювати з комп’ютерною графікою.

Серед представлених інтегрованих середовищ розробки, що були представлені в пункті 2.1 було використане таке інтегроване середовище розробки як Microsoft Visual Studio. Visual Studio має ряд переваг над іншими IDE. Серед переваг можна виділити редактор коду, що підтримує компонент завершення коду IntelliSense, рефакторинг коду, вбудований налагоджувач та широкий вибір плагінів, що також розширюють функціонал інтегрованого середовища розробки.

В якості технології, на основі якої було розроблено ПЗ, був оправний програмний інтерфейс Vulkan. Даний програмний інтерфейс має вагому перевагу над іншими, що були представлені в розділі 1 магістерської дисертації. Він є кросплатформеним. Як і з програмної точки зору, тобто може працювати на більшості популярних операційних систем, так і апаратної точки зору, так як не прив’язаний лише до одного вендору, як наприклад технологія CUDA, що має змогу працювати лише на графічному обладнанні компанії NVIDIA.

3.2. Підхід до розроблення трасувальника променів

Випуск графічного процесору Turing компанією NVIDIA у 2018 році вперше запровадив трасування променів у реальному часі на користувацькому графічному процесорі та дав поштовх до активного розвитку даного напрямку трасування променів. Так багато розробників вирішили перейти на більш відкритий підхід з використанням низькорівневих API. Тому, в даній роботі також було прийнято рішення використовувати низькорівневий API Vulkan, що дозволяє орієнтуватися на багато різних платформ, включаючи Windows і Linux, що дозволяє ширше розповсюджувати прикладне програмне забезпечення, яке спирається на апаратно-програмну реалізацію тривимірної графіки.

Vulkan — це API, що не залежить від апаратного забезпечення, і тому реалізація трасування променів з використанням даного прикладного програмного інтерфейсу також не залежить від апаратного забезпечення. Весь API можна реалізувати поверх існуючої обчислювальної функціональності Vulkan. Ми також доклали чимало зусиль, щоб переконатися, що розширення добре відповідає існуючим концепціям API Vulkan. Розподіл пам’яті, обробка ресурсів і мова/байт-код шейдерів обробляються так само, як це передбачено основним API Vulkan.

На відміну від растеризації, кількість «одиниць» (променів) виконаної роботи залежить від результату попередніх робочих одиниць. Це означає, що нова робота може бути породжена програмованими етапами та подана назад у конвеєр.

API трасування променів складається з чотирьох ключових компонентів:

- Структури прискорення
- Нові домени шейдерів для трасування променів
- Таблиця зв'язування шейдерів
- Трасування променів конвеєрних об'єктів

Традиційна растеризація включає обробку кожного геометричного примітиву незалежно. Трасування променів навпаки вимагає перевірки кожного променя на всі примітиви сцени, що може бути надзвичайно ресурсо та часозатратним.

Більшість трасувальників променів реалізують певну форму структури прискорення, щоб забезпечити швидке відхилення примітивів. Структура прискорення (AS) — це об’єкт, який містить геометричну інформацію для примітивів у сцені, попередньо оброблений таким чином, щоб уможливити тривіальне відхилення потенційних перетинів променів і примітивів.

Структура прискорення представлена як непрозора, визначена реалізацією структура даних, яка не залежить від будь-якого основного алгоритму чи методу відсіву. Вона моделюється як дворівнева структура: вузли нижнього рівня структури прискорення містять геометричні дані, тоді як вузли верхнього рівня структури прискорення містять список посилань на вузли нижнього рівня разом із пов’язаною інформацією про перетворення та затінення. На рисунку 3.2.1 показано зв’язки між цими структурами.

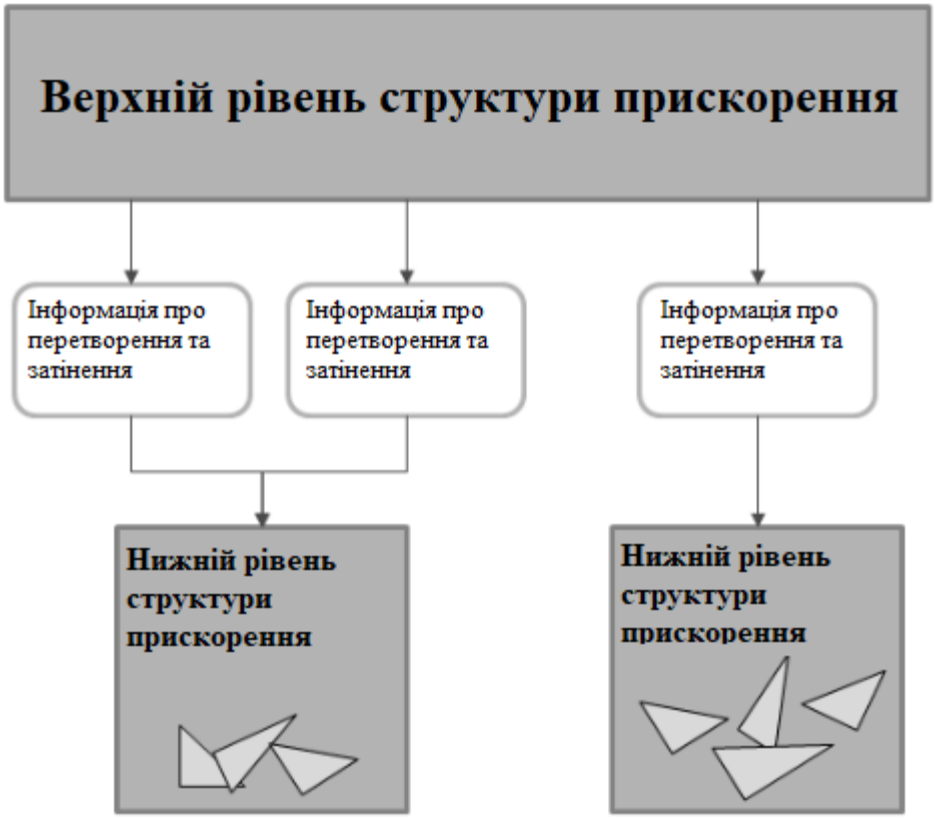


Рисунок 3.2.1 – Зв’язки між верхнім та нижнім рівнем структури прискорення

Структури прискорення можуть бути побудовані з існуючої геометрії в об’єктах VkBuffer. Створення структури прискорення — це двоетапний процес: спочатку створюють вузли нижнього рівня, а потім генерують вузол верхнього рівня.

Усі операції збирання та оновлення відбуваються на графічному процесорі. У цьому контексті «збирання» означає створення нового об’єкта з нуля, наприклад під час початкового налаштування сцени, тоді як «оновлення» означає, що ми оновлюємо наявний об’єкт деякими новими даними.

На рівні програмного інтерфейсу визначається новий тип об’єкта Vulkan: VkAccelerationStructureNVX. Кожен екземпляр цього об’єкта інкапсулює вузол AS верхнього або нижнього рівня. Ми також визначаємо новий тип дескриптора ресурсу, VK_DESCRIPTOR_TYPE_ACCELERATION_ _STRUCTURE_NVX, що використовується для прив’язки об’єктів структури прискорення до шейдерів.

Створення та знищення об’єктів відбувається за звичайною парадигмою Vulkan: вузол структури прискорення створюється за допомогою виклику vkCreateAccelerationStructureNVX, який повертає непрозорий дескриптор. Потім цей дескриптор можна використовувати з vkGetAccelerationStructureMemoryRequirementsNVX для отримання інформації про те, який тип і скільки пам’яті знадобиться цій структурі прискорення. Потім пам’ять можна виділити за допомогою vkAllocateMemory і зв’язати з об’єктом, викликавши vkBindAccelerationStructureMemoryNVX.

Операції збирання/оновлення структури прискорення вимагають деякої кількості тимчасової «нульової» пам’яті для роботи. Цю пам’ять можна запитати, викликавши:

vkGetAccelerationStructureScratchMemoryRequirementsNVX.

Початкова пам’ять має форму звичайного об’єкта VkBuffer, виділеного на основі вимог до пам’яті, які повертає реалізація. Це передається як аргумент для команд збирання та оновлення.

Об’єм пам’яті, який потребує об’єкт AS, залежить від конкретних геометричних даних, які передаються. Таким чином, дані, які повертає vkGetAccelerationStructureMemoryRequirementsNVX, є верхньою межею

обсягу пам’яті, який буде вимагатися певним об’єктом. Після того, як структуру було створено, можна використати команду vkCmdWriteAccelerationStructurePropertiesNVX, щоб змусити графічне обладнання записати стислий розмір даного об’єкта структури прискорення в об’єкт запиту Vulkan, який потім можна прочитати на центральному процесорі. Після це можна використати для виділення окремого об’єкта AS із точним об’ємом пам’яті, який потрібен, а vkCmdCopyAccelerationStructureNVX можна використати для стиснення вихідного об’єкта структури прискорення у новий об’єкт структури прискорення.

Команди збирання/оновлення AS виконуються на графічному процесорі та можуть надсилатися в черги графіки чи обчислення. API дозволяє розпаралелювати послідовні команди збирання та оновлення, щоб максимізувати використання графічного процесору.

Слід бути обережним під час повторного використання тимчасових буферів у командах збирання/оновлення, оскільки їхнє виконання може збігатися. Синхронізація бар’єрів необхідна для коректності, використовуючи нові біти прапорів доступу до пам’яті VK_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_NVX і VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_NVX: їх слід використовувати в бар’єрах буферної пам’яті на тимчасовому буфері перед повторним використанням тієї самої області буфера для іншої операції та на глобальних бар’єрах пам’яті, щоб переконатись, що збирання/оновлення об’єкту структури прискорення завершилося до того, як об’єкт AS буде використано для трасування променів.

Щоб максимізувати перекриття, рекомендується виділяти достатньо пам’яті початкового буфера для кількох операцій збирання/оновлення та призначати кожній послідовній операції окремі області пам’яті. Скільки пам’яті виділити, залежить від того, наскільки програмне забезпечення чутливе до продуктивності побудови структури прискорення, а також від того, скільки структур прискорення буде використано.

Щоб надати API Vulkan функції трасування променів, визначається новий набір доменів шейдерів разом із примітивами для зв’язку між шейдерами.

Шейдери генерації променів («raygen») розпочинають ввесь процес трасування променів. Шейдер raygen працює на двовимірній сітці потоків, подібно до обчислювального шейдера, і є відправною точкою для трасування променів у сцені. Він також відповідає за запис остаточного результату алгоритму трасування променів у пам’ять.

Шейдери перетину реалізують довільні алгоритми перетину примітивних променів. Вони корисні, коли потрібно дозволити програмному забезпеченню перетинати промені різними видами примітивів, які не мають вбудованої підтримки, наприклад сфера. Трикутні примітиви мають вбудовану підтримку і не вимагають шейдера перетину.

Хіт-шейдери викликаються, коли знайдено перетин примітивів променя. Вони відповідають за обчислення взаємодій, які відбуваються в точці перетину, наприклад, взаємодії світла та матеріалу для графічних додатків, і можуть породжувати нові промені за потреби.

Міс-шейдери викликаються, коли не знайдено перетину для даного променя.

Під час виконання виклику трасування променів різні етапи шейдерів повинні передавати дані один одному. Шейдерам перетину потрібно виводити дані перетину, хіт-шейдерам потрібно отримати дані перетину та змінювати деякі довільні дані результату, а шейдерам «raygen» потрібно споживати кінцеві дані результату та виводити їх у пам’ять.

Корисне навантаження променя — це довільна структура даних, визначена програмою, яка зберігає інформацію, що накопичується на шляху променя. Він ініціалізується шейдером, який ініціює операцію запиту променя (зазвичай шейдером raygen), і може бути прочитаний і змінений хіт-шейдерами. Зазвичай він використовується для виведення властивостей матеріалу, накопичених уздовж шляху променя, які потім raygen-шейдер записує в пам’ять.

Атрибути променя містять набір значень, що передаються назад від шейдера перетину до хіт-шейдерів, що містить будь-які дані, які програма повинна виключити з тесту перетину. Вони інкапсулюються в визначену програмою структуру, яка записується шейдером перетину та зчитується хіт-шейдерами, викликаними на цьому перетині. Для вбудованого шейдера перетину променів і трикутника атрибути променя є вектором, що містить барицентричні координати точки перетину вздовж трикутника.

Щоб увімкнути трасування променів у Vulkan, потрібно внести відносно мало змін у мову затінення.

Програмний інтерфейс визначає п’ять нових етапів шейдерів для побудови шейдерів GLSL: rgen для шейдерів raygen, rint для шейдерів перетину, rchit для хіт-шейдерів з найближчим ударом, rahit для хіт-шейдерів із будь-яким ударом і rmiss для шейдерів із пропуском.

Тепер існує кілька нових вбудованих змінних, залежно від стадії шейдера, які містять інформацію про різні параметри. До них входять ідентифікатор поточного потоку raygen, розмір сітки запуску raygen, значення ідентифікатора примітиву та екземпляра, поточне походження та напрямок променя у світі та просторі об’єктів, а також значення T для поточного перетину, що обробляється вздовж променя, серед іншого.

Визначено новий непрозорий тип ресурсу для зв’язків ресурсів структури прискорення, accelerationStructureNVX. Структури прискорення прив’язані до конвеєра шейдерів так само, як і інші типи ресурсів. Так в одному шейдері можна використовувати кілька структур прискорення, що забезпечує ієрархічний обхід різних наборів геометричних даних у шейдері.

Також визначається кілька нових кваліфікаторів сховища для визначених користувачем типів: rayPayloadNVX, rayPayloadInNVX і hitAttributeNVX, які оголошують корисне навантаження променя, визначають якому етапу шейдера належить сховище для даного корисного навантаження і оголошують типи атрибутів звернень, які будуть використовуватися. Додано новий кваліфікатор макета shaderRecordNVX, який оголошує об’єкт буферу зберігання шейдерів прив’язаним до таблиці зв’язування шейдерів.

Нарешті, додано декілька нових вбудованих функцій. До них належать traceNVX(), який запускає промінь у сцену, ignoreIntersectionNVX(), щоб

відкинути задану точку перетину, і terminateRayNVX(), щоб зупинити обробку променя на заданому попаданні.

API шейдера трасування променів дозволяє використовувати будь-яку структуру, визначену програмою, як корисне навантаження променів. Це корисне навантаження оголошується на етапі шейдера, який може породжувати промені (зазвичай raygen), і передається за посиланням на етапи шейдера, які його змінюють.

Оскільки GLSL не має поліморфізму типів, traceNVX() не можна визначити як поліморфний тип. Це означає, що довільні типи не будуть передаватись як аргументи. Щоб обійти це, було знайдено рішення, яке відповідає типам даних на основі кваліфікатора макета розташування.

Кожна структура корисного навантаження ray оголошується з кваліфікатором rayPayloadNVX плюс кваліфікатором макета розташування, який пов’язує її з цілим значенням. Це ціле значення по суті стає числовим ідентифікатором для змінної корисного навантаження, яка потім передається до виклику traceNVX() замість посилання на фактичну змінну. Хіт-шейдери повинні оголосити той самий тип даних як змінну з кваліфікатором rayPayloadInNVX, дозволяючи інфраструктурі трасування променів розглядати цю змінну як посилання на вхідні дані корисного навантаження. Фрагмент коду простого шейдера raygen ілюструє рисунок 3.2.2:

```
layout (location = 1) rayPayloadNVX primaryPayload {

    vec4 color;

};

void main()
{
   color = vec4(0.0, 0.0, 0.0, 0.0);

   traceNVX(scene,     // acceleration structure
            ...
            1);        // payload location index

   imageStore(framebufferImage, gl_GlobalInvocationIDNV.xy,
              color);
}
```

Рисунок 3.2.2 – Фрагмент коду простого шейдера raygen

primaryPayload — це корисне навантаження променів. При декларації йому призначається індекс location 1, і цей індекс передається як останній параметр у traceNVX(). Приклад відповідного хіт-шейдеру з найближчим ударом зображено на рисунку 3.2.3.

```
rayPayloadInNVX primaryPayload {
    vec4 color;
}

layout(set = 3, binding = 0) sampler3D solidMaterialSampler;

void main()
{
    vec3 pos = gl_WorldRayOriginNVX +
               gl_HitTNVX * gl_WorldRayDirectionNVX;
    color = texture(solidMaterialSampler, pos);
}
```

Рисунок 3.2.3 – Приклад хіт-шейдеру з найближчим ударом

Така ж сама структура корисного навантаження оголошується за допомогою ключового слова rayPayloadInNVX. У шейдері може існувати лише

одна змінна rayPayloadInNVX . Також слід зазначити, що не існує статичної перевірки для відповідних типів корисного навантаження, а невідповідність типів призведе до невизначеної поведінки. Значення корисного навантаження у виклику traceNVX() має бути негайним значенням під час компіляції.

Останньою новою концепцією, представленою в програмному інтерфейсі, є таблиця зв'язування шейдерів (SBT). Це VkBuffer, що містить набір записів однакового розміру та складається з маркерів шейдерів, за якими йдуть дані, визначені програмою. Маркери шейдерів визначають, які шейдери запускати для даного запису SBT, тоді як визначені програмою дані в записі стають доступними для шейдерів як SSBO.

Кожен екземпляр таблиці зв'язування шейдерів має фіксований розмір запису. У кожному записі низка правил індексації визначає, як інфраструктура трасування променів отримуватиме маркери та дані SSBO для виконання наступного шейдера.

Дані SSBO в SBT призначені для визначення того, які ресурси (текстури, уніфіковані буфери, тощо) використовуватиме кожен шейдер. Передбачуваний шаблон використання полягає в тому, щоб усі потенційно доступні ресурси прив'язувалися до конвеєра через набори дескрипторів, використовуючи масиви ресурсів. Тоді таблиця зв'язування шейдерів вставлятиме індекси в масиви, що вказують, які конкретні ресурси слід використовувати для даного набору шейдерів. Тим не менш, SBT — це «просто дані» з точки зору шейдера.

Останнім необхідним елементом є об'єкт стану конвеєра трасування променів. Конвеєри трасування променів складаються лише з колекції рейгенів, перетинів, хіт та міс шейдерів разом із парою специфічних

параметрів трасування променів (таких як максимальна глибина рекурсії променя).

Під час створення PSO новий прапорець VK_PIPELINE_CREATE_DEFER_COMPILE_BIT_NVX наказує драйверу негайно уникати компіляції шейдерів. Потім програма повинна викликати vkCompileDeferredNVX для кожного шейдера, щоб запустити роботу компіляції. Це можна розпаралелювати між потоками, щоб мінімізувати час компіляції.

Після створення конвеєрні об’єкти трасування променів можна прив’язати до графіки чи черги обчислень за допомогою стандартних викликів Vulkan із новою точкою прив’язки конвеєра

VK_PIPELINE_BIND_POINT_RAYTRACING_NVX.

Коли структуру прискорення створено, таблицю зв’язування шейдерів створено, а об’єкт стану конвеєра трасування променів створено та зв’язано, настав час розпочати трасування променів.

Vulkan ініціює роботу трасування променів за допомогою команди vkCmdTraceRaysNVX. Аргументи цієї команди вказують розміри сітки двовимірного потоку, а також VkBuffer, який містить таблицю зв’язування шейдерів, що буде використовуватися, і зміщення в межах цієї таблиці для різних необхідних елементів даних (дескриптор шейдера raygen і зміщення даних SSBO, дескриптор шейдера початкового попадання та Зміщення даних SSBO, а також зміщення для маркера пропущеного шейдера та відповідних даних SSBO).

3.3. Опис модулів розробленого ПЗ

Розроблене програмне забезпечення поділене на модулі, що в свою чергу відповідають за свою область роботи з графікою, інтерфейсом користувача та шейдери.

Main.cpp (Рис. 3.3.1) – це своєрідна точка входу програмного забезпечення. В даному файлі присутні налаштування рендеру графічного інтерфейсу користувача (Рис. 3.3.2 та Рис. 3.3.3). Також тут відбувається підключення потрібних розширень для API Vulkan та ініціалізація трасування променів шляхом виклику методу initRayTracing() класу VulkanClass, (Рис. 3.3.4). В main.cpp відбувається створення неявної геометрії(Рис. 3.3.5) та об'єктів сцени(Рис. 3.3.6). Відбуваються виклики відповідних методів, що відповідають за відображення сцени на екрані (Рис. 3.3.7).

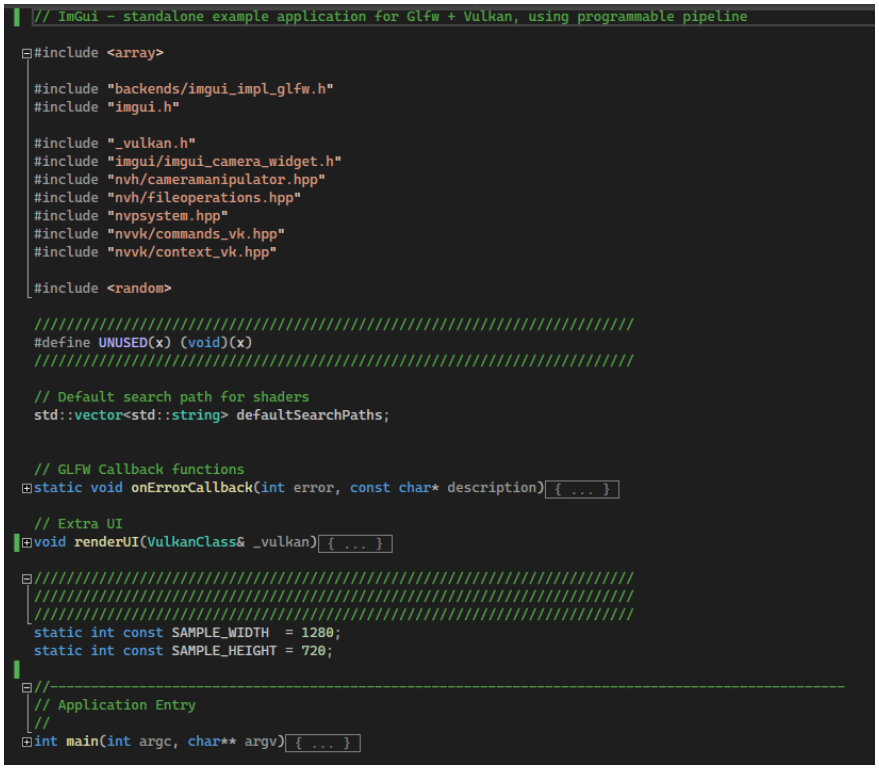


Рисунок 3.3.1 – Файл main.cpp

```
// Extra UI
void renderUI(VulkanClass& _vulkan)
{
  bool changed = false;

  if(ImGui::CollapsingHeader("Light"))
  {
    auto& pc = _vulkan.m_pcRaster;

    changed |= ImGui::RadioButton("Point", &pc.lightType, 0);
    ImGui::SameLine();
    changed |= ImGui::RadioButton("Spot", &pc.lightType, 1);
    ImGui::SameLine();
    changed |= ImGui::RadioButton("Infinite", &pc.lightType, 2);


    if(pc.lightType < 2)
    {
      changed |= ImGui::SliderFloat3("Light Position", &pc.lightPosition.x, -20.f, 20.f);
    }
    if(pc.lightType > 0)
    {
      changed |= ImGui::SliderFloat3("Light Direction", &pc.lightDirection.x, -1.f, 1.f);
    }
    if(pc.lightType < 2)
    {
      changed |= ImGui::SliderFloat("Light Intensity", &pc.lightIntensity, 0.f, 500.f);
    }
```

Рисунок 3.3.2 – Налаштування рендеру інтерфейсу користувача

```
// Show UI window.
if(_vulkan.showGui())
{
  ImGuiH::Panel::Begin();
  bool changed = false;
  // Edit 3 floats representing a color
  changed |= ImGui::ColorEdit3("Clear color", reinterpret_cast<float*>(&clearColor));
  // Switch between raster and ray tracing
  changed |= ImGui::Checkbox("Ray Tracer mode", &useRaytracer);
  if(changed)
    _vulkan.resetFrame();

  renderUI(_vulkan);
  ImGui::Text("Application average %.3f ms/frame (%.1f FPS)", 1000.0f / ImGui::GetIO().Framerate, ImGui::GetIO().Framerate);
  ImGuiH::Panel::End();
}
```

Рисунок 3.3.3 – Налаштування та рендер інтерфейсу користувача

```
// #VKRay
_vulkan.initRayTracing();
```

Рисунок 3.3.4 – Виклик методу, що відповідає за ініціалізацію трасування променів

```
// Creation of implicit geometry
MaterialObj mat;
// Reflective
mat.diffuse   = nvmath::vec3f(0, 0, 0);
mat.specular  = nvmath::vec3f(1.f);
mat.shininess = 0.0;
mat.illum     = 3;
_vulkan.addImplMaterial(mat);
// Transparent
mat.diffuse  = nvmath::vec3f(0.4, 0.4, 1);
mat.illum    = 4;
mat.dissolve = 0.5;
_vulkan.addImplMaterial(mat);
_vulkan.addImplCube({-6.1, 0, -6}, {-6, 10, 6}, 0);
_vulkan.addImplSphere({1, 2, 4}, 1.f, 1);
```

Рисунок 3.3.5 – Створення неявної геометрії

```
// Creation of the example
_vulkan.loadModel(nvh::findFile("media/scenes/Medieval_building.obj", defaultSearchPaths, true));
_vulkan.loadModel(nvh::findFile("media/scenes/plane.obj", defaultSearchPaths, true));
_vulkan.loadModel(nvh::findFile("media/scenes/wuson.obj", defaultSearchPaths, true),
                  nvmath::scale_mat4(nvmath::vec3f(0.5f)) * nvmath::translation_mat4(nvmath::vec3f(0.0f, 0.0f, 6.0f)));
```

Рисунок 3.3.6 – Приклад ініціалізації об’єктів сцени

```
// Offscreen render pass
{
  VkRenderPassBeginInfo offscreenRenderPassBeginInfo{VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO};
  offscreenRenderPassBeginInfo.clearValueCount = 2;
  offscreenRenderPassBeginInfo.pClearValues    = clearValues.data();
  offscreenRenderPassBeginInfo.renderPass      = offscreen.renderPass();
  offscreenRenderPassBeginInfo.framebuffer     = offscreen.frameBuffer();
  offscreenRenderPassBeginInfo.renderArea      = {{0, 0}, _vulkan.getSize()};

  // Rendering Scene
  if(useRaytracer)
  {
    _vulkan.raytrace(cmdBuf, clearColor);
  }
  else
  {
    vkCmdBeginRenderPass(cmdBuf, &offscreenRenderPassBeginInfo, VK_SUBPASS_CONTENTS_INLINE);
    _vulkan.rasterize(cmdBuf);
    vkCmdEndRenderPass(cmdBuf);
  }
}
```

Рисунок 3.3.7 – Виклик методів, що відповідають за відображення сцени на екрані

Файл _vulkan.cpp (Рис. 3.3.8) містить в собі основну роботу із програмним інтерфейсом Vulkan. Так у даному файлі представлені основні методи, що призначені для роботи з графікою та викликались у файлі main.cpp. У функції updateUniformBuffer (Рис. 3.3.9) зберігаються та оновлюються матриці камери, що потрібні для трасування променів. Шейдери трасування променів мають доступ до опису сцени, тому потрібно розширити прапорці доступу відповідних буферів, це відбуваєьбся в createDescriptorSetLayout(Рис. 3.3.10). RayGen має отримати доступ до матриць камери для обчислення напрямків променів, а ClosestHit потребує доступу до матеріалів, екземплярів сцени, текстур, буферів вершин та буферів індексів. Незважаючи на те, що буфери вершин та індексів використовуватимуться лише шейдерами трасування променів, вони додаються до набору дескрипторів, оскільки семантично відповідають набору дескрипторів сцени. Оновлення набору дескрипторів трасування променів, наприклад при зміні розміру вікна,

виконується в методі updateDescriptorSet(Рис. 3.3.11). У методі loadModel(Рис. 3.3.12) відбувається завантаження об’єкту сцени та налаштування всіх необхідних об’єкту буферів. Параметрами методу є назва об’єкту, що включає шлях до нього, та позиція). Створенням всіх знайдених текстур займається метод createTextureImages(Рис. 3.3.13). За растеризацію сцени відповідає метод rasterize(Рис. 3.3.14). За трасування променів в даному класі відповідає метод raytrace(Рис. 3.3.15). Даний метод оновлює кадр та викликає метод класу Raytracer. У файлі _vulkan.h (Рис. 3.3.16) міститься оголошення класу VulkanClass.

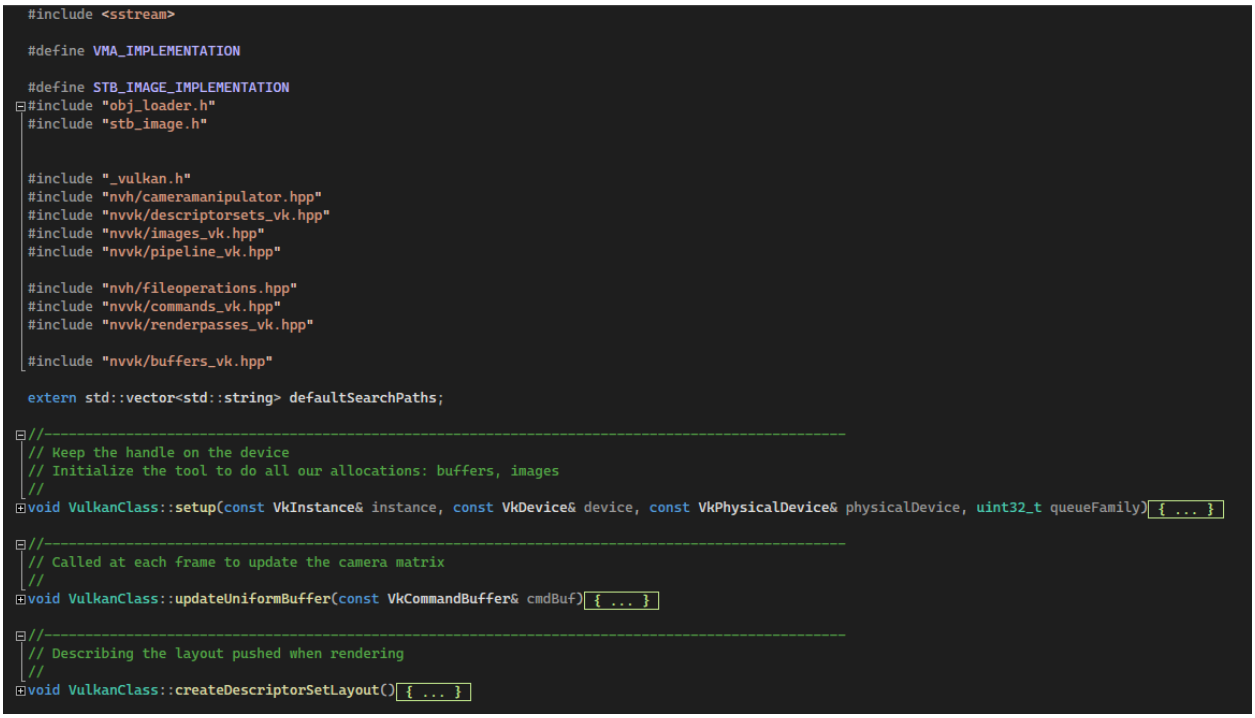


Рисунок 3.3.8 – Файл _vulkan.cpp

```cpp
void VulkanClass::updateUniformBuffer(const VkCommandBuffer& cmdBuf)
{
  // Prepare new UBO contents on host.
  const float    aspectRatio = m_size.width / static_cast<float>(m_size.height);
  GlobalUniforms hostUBO     = {};
  const auto&    view        = CameraManip.getMatrix();
  const auto&    proj        = nvmath::perspectiveVK(CameraManip.getFov(), aspectRatio, 0.1f, 1000.0f);
  // proj[1][1] *= -1;  // Inverting Y for Vulkan (not needed with perspectiveVK).

  hostUBO.viewProj    = proj * view;
  hostUBO.viewInverse = nvmath::invert(view);
  hostUBO.projInverse = nvmath::invert(proj);

  // UBO on the device, and what stages access it.
  VkBuffer deviceUBO      = m_bGlobals.buffer;
  auto     uboUsageStages = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT | VK_PIPELINE_STAGE_RAY_TRACING_SHADER_BIT_KHR;

  // Ensure that the modified UBO is not visible to previous frames.
  VkBufferMemoryBarrier beforeBarrier{VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER};
  beforeBarrier.srcAccessMask = VK_ACCESS_SHADER_READ_BIT;
  beforeBarrier.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
  beforeBarrier.buffer        = deviceUBO;
  beforeBarrier.offset        = 0;
  beforeBarrier.size          = sizeof(hostUBO);
  vkCmdPipelineBarrier(cmdBuf, uboUsageStages, VK_PIPELINE_STAGE_TRANSFER_BIT, VK_DEPENDENCY_DEVICE_GROUP_BIT, 0,
                       nullptr, 1, &beforeBarrier, 0, nullptr);
```

Рисунок 3.3.9 – Метод updateUniformBuffer

```cpp
//--------------------------------------------------------------------------------------------------
// Describing the layout pushed when rendering
//
void VulkanClass::createDescriptorSetLayout()
{
  auto nbTxt = static_cast<uint32_t>(m_textures.size());

  // Camera matrices
  m_descSetLayoutBind.addBinding(SceneBindings::eGlobals, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, 1,
                                 VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_RAYGEN_BIT_KHR);
  // Obj descriptions
  m_descSetLayoutBind.addBinding(SceneBindings::eObjDescs, VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, 1,
                                 VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT
                                     | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR | VK_SHADER_STAGE_ANY_HIT_BIT_KHR);
  // Textures
  m_descSetLayoutBind.addBinding(SceneBindings::eTextures, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, nbTxt,
                                 VK_SHADER_STAGE_FRAGMENT_BIT | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR | VK_SHADER_STAGE_ANY_HIT_BIT_KHR);

  // Storing implicit obj (binding = 3)
  m_descSetLayoutBind.addBinding(eImplicits, VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, 1,
                                 VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR | VK_SHADER_STAGE_INTERSECTION_BIT_KHR
                                     | VK_SHADER_STAGE_ANY_HIT_BIT_KHR);


  m_descSetLayout = m_descSetLayoutBind.createLayout(m_device);
  m_descPool      = m_descSetLayoutBind.createPool(m_device, 1);
  m_descSet       = nvvk::allocateDescriptorSet(m_device, m_descPool, m_descSetLayout);
}
```

Рисунок 3.3.10 – Метод createDescriptorSetLayout

```cpp
//--------------------------------------------------------------------------------------------------
// Setting up the buffers in the descriptor set
//
void VulkanClass::updateDescriptorSet()
{
  std::vector<VkWriteDescriptorSet> writes;

  // Camera matrices and scene description
  VkDescriptorBufferInfo dbiUnif{m_bGlobals.buffer, 0, VK_WHOLE_SIZE};
  writes.emplace_back(m_descSetLayoutBind.makeWrite(m_descSet, SceneBindings::eGlobals, &dbiUnif));

  VkDescriptorBufferInfo dbiSceneDesc{m_bObjDesc.buffer, 0, VK_WHOLE_SIZE};
  writes.emplace_back(m_descSetLayoutBind.makeWrite(m_descSet, SceneBindings::eObjDescs, &dbiSceneDesc));

  // All texture samplers
  std::vector<VkDescriptorImageInfo> diit;
  for(auto& texture : m_textures)
  {
    diit.emplace_back(texture.descriptor);
  }
  writes.emplace_back(m_descSetLayoutBind.makeWriteArray(m_descSet, SceneBindings::eTextures, diit.data()));

  VkDescriptorBufferInfo dbiImplDesc{m_implObjects.implBuf.buffer, 0, VK_WHOLE_SIZE};
  writes.emplace_back(m_descSetLayoutBind.makeWrite(m_descSet, 3, &dbiImplDesc));

  // Writing the information
  vkUpdateDescriptorSets(m_device, static_cast<uint32_t>(writes.size()), writes.data(), 0, nullptr);
}
```

Рисунок 3.3.11 – Метод updateDescriptorSet

```cpp
void VulkanClass::loadModel(const std::string& filename, nvmath::mat4f transform)
{
  LOGI("Loading File:  %s \n", filename.c_str());
  ObjLoader loader;
  loader.loadModel(filename);

  // Converting from Srgb to linear
  for(auto& m : loader.m_materials)
  {
    m.ambient  = nvmath::pow(m.ambient, 2.2f);
    m.diffuse  = nvmath::pow(m.diffuse, 2.2f);
    m.specular = nvmath::pow(m.specular, 2.2f);
  }

  ObjModel model;
  model.nbIndices  = static_cast<uint32_t>(loader.m_indices.size());
  model.nbVertices = static_cast<uint32_t>(loader.m_vertices.size());

  // Create the buffers on Device and copy vertices, indices and materials
  nvvk::CommandPool  cmdBufGet(m_device, m_graphicsQueueIndex);
  VkCommandBuffer    cmdBuf          = cmdBufGet.createCommandBuffer();
  VkBufferUsageFlags flag            = VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT;
  VkBufferUsageFlags rayTracingFlags =  // used also for building acceleration structures
      flag | VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY_BIT_KHR | VK_BUFFER_USAGE_STORAGE_BUFFER_BIT;
  model.vertexBuffer = m_alloc.createBuffer(cmdBuf, loader.m_vertices, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT | rayTracingFlags);
  model.indexBuffer = m_alloc.createBuffer(cmdBuf, loader.m_indices, VK_BUFFER_USAGE_INDEX_BUFFER_BIT | rayTracingFlags);
  model.matColorBuffer = m_alloc.createBuffer(cmdBuf, loader.m_materials, VK_BUFFER_USAGE_STORAGE_BUFFER_BIT | flag);
  model.matIndexBuffer = m_alloc.createBuffer(cmdBuf, loader.m_matIndx, VK_BUFFER_USAGE_STORAGE_BUFFER_BIT | flag);
  // Creates all textures found and find the offset for this model
  auto txtOffset = static_cast<uint32_t>(m_textures.size());
  createTextureImages(cmdBuf, loader.m_textures);
  cmdBufGet.submitAndWait(cmdBuf);
  m_alloc.finalizeAndReleaseStaging();
```

Рисунок 3.3.12 – Метод loadModel

```cpp
//--------------------------------------------------------------------------------------------------
// Creating all textures and samplers
//
void VulkanClass::createTextureImages(const VkCommandBuffer& cmdBuf, const std::vector<std::string>& textures)
{
  VkSamplerCreateInfo samplerCreateInfo{VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO};
  samplerCreateInfo.minFilter  = VK_FILTER_LINEAR;
  samplerCreateInfo.magFilter  = VK_FILTER_LINEAR;
  samplerCreateInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
  samplerCreateInfo.maxLod     = FLT_MAX;

  VkFormat format = VK_FORMAT_R8G8B8A8_SRGB;

  // If no textures are present, create a dummy one to accommodate the pipeline layout
  if(textures.empty() && m_textures.empty())
  {
    nvvk::Texture texture;

    std::array<uint8_t, 4> color{255u, 255u, 255u, 255u};
    VkDeviceSize           bufferSize      = sizeof(color);
    auto                   imgSize         = VkExtent2D{1, 1};
    auto                   imageCreateInfo = nvvk::makeImage2DCreateInfo(imgSize, format);

    // Creating the dummy texture
    nvvk::Image           image  = m_alloc.createImage(cmdBuf, bufferSize, color.data(), imageCreateInfo);
    VkImageViewCreateInfo ivInfo = nvvk::makeImageViewCreateInfo(image.image, imageCreateInfo);
    texture                      = m_alloc.createTexture(image, ivInfo, samplerCreateInfo);

    // The image format must be in VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
    nvvk::cmdBarrierImageLayout(cmdBuf, texture.image, VK_IMAGE_LAYOUT_UNDEFINED, VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL);
    m_textures.push_back(texture);
  }
```

Рисунок 3.3.13 – Метод createTextureImages

```cpp
//--------------------------------------------------------------------------------------------------
// Drawing the scene in raster mode
//
void VulkanClass::rasterize(const VkCommandBuffer& cmdBuf)
{
  VkDeviceSize offset{0};

  m_debug.beginLabel(cmdBuf, "Rasterize");

  // Dynamic Viewport
  setViewport(cmdBuf);

  // Drawing all triangles
  vkCmdBindPipeline(cmdBuf, VK_PIPELINE_BIND_POINT_GRAPHICS, m_graphicsPipeline);
  vkCmdBindDescriptorSets(cmdBuf, VK_PIPELINE_BIND_POINT_GRAPHICS, m_pipelineLayout, 0, 1, &m_descSet, 0, nullptr);

  auto nbInst = static_cast<uint32_t>(m_instances.size() - 1);  // Remove the implicit object
  for(uint32_t i = 0; i < nbInst; ++i)
  {
    auto& inst             = m_instances[i];
    auto& model            = m_objModel[inst.objIndex];
    m_pcRaster.objIndex    = inst.objIndex;  // Telling which object is drawn
    m_pcRaster.modelMatrix = inst.transform;

    vkCmdPushConstants(cmdBuf, m_pipelineLayout, VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT, 0,
                       sizeof(PushConstantRaster), &m_pcRaster);
    vkCmdBindVertexBuffers(cmdBuf, 0, 1, &model.vertexBuffer.buffer, &offset);
    vkCmdBindIndexBuffer(cmdBuf, model.indexBuffer.buffer, 0, VK_INDEX_TYPE_UINT32);
    vkCmdDrawIndexed(cmdBuf, model.nbIndices, 1, 0, 0, 0);
  }
```

Рисунок 3.3.14 – Метод rasterize

```
void VulkanClass::raytrace(const VkCommandBuffer& cmdBuf, const nvmath::vec4f& clearColor)
{
  updateFrame();
  if(m_pcRaster.frame >= m_maxFrames)
    return;

  m_raytrace.raytrace(cmdBuf, clearColor, m_descSet, m_size, m_pcRaster);
}
```

Рисунок 3.3.15 – Метод raytrace

```
#pragma once

#include "nvvk/appbase_vk.hpp"
#include "nvvk/debug_util_vk.hpp"
#include "nvvk/resourceallocator_vk.hpp"
#include "shaders/host_device.h"

// #VKRay
#include "nvvk/raytraceKHR_vk.hpp"
#include "offscreen.hpp"

#include "obj.hpp"
#include "raytrace.hpp"

#include <memory>

// Choosing the allocator to use
#define ALLOC_DMA
//#define ALLOC_DEDICATED
//#define ALLOC_VMA
#include <nvvk/resourceallocator_vk.hpp>

#if defined(ALLOC_DMA) Active Preprocessor Block
#elif defined(ALLOC_VMA) Inactive Preprocessor Block
#endif


//--------------------------------------------------------------------------------------------------
// Simple rasterizer of OBJ objects
// - Each OBJ loaded are stored in an `ObjModel` and referenced by a `ObjInstance`
// - It is possible to have many `ObjInstance` referencing the same `ObjModel`
// - Rendering is done in an offscreen framebuffer
// - The image of the framebuffer is displayed in post-process in a full-screen quad
//
class VulkanClass { ... };
```

Рисунок 3.3.16 – Файл _vulkan.h

raytraysce.cpp (Рис. 3.3.17) – це файл який відповідає за більшість методів, що працюють саме для реалізації трасування променів. В даному файлі присутня конвертація геометричних даних завантаженої моделі в кілька структур, що потім використовуються при створенні структури прискорення(Рис. 3.3.18). Це перетворення є першим кроком створення

нижнього рівня структури прискорення. Так, метод objectToVkGeometryKHR заповнить три структури, що згодом будуть передані конструктору структури прискорення. За створення нижнього рівня структури прискорення відповідає метод createBottomLevelAs(Рис. 3.3.19). Він виконується для кожного завантаженого об’єкту. Отримані структури прискорення зберігатимуться в помічнику в порядку побудови, щоб на них можна було безпосередньо посилатися за індексом пізніше.

```
#include "raytrace.hpp"
#include "nvh/fileoperations.hpp"
#include "nvvk/descriptorsets_vk.hpp"

#include "nvh/alignment.hpp"
#include "nvvk/shaders_vk.hpp"
#include "obj_loader.h"
#include "nvvk/buffers_vk.hpp"

extern std::vector<std::string> defaultSearchPaths;


void Raytracer::setup(const VkDevice& device, const VkPhysicalDevice& physicalDevice, nvvk::ResourceAllocator* allocator, uint32_t queueFamily) { ... }


void Raytracer::destroy() { ... }

//--------------------------------------------------------------------------------------------------
// Converting a OBJ primitive to the ray tracing geometry used for the BLAS
//
auto Raytracer::objectToVkGeometryKHR(const ObjModel& model) { ... }


//--------------------------------------------------------------------------------------------------
// Returning the ray tracing geometry used for the BLAS, containing all spheres
//
auto Raytracer::implicitToVkGeometryKHR(const ImplInst& implicitObj) { ... }


void Raytracer::createBottomLevelAS(std::vector<ObjModel>& models, ImplInst& implicitObj) { ... }

void Raytracer::createTopLevelAS(std::vector<ObjInstance>& instances, ImplInst& implicitObj) { ... }

//--------------------------------------------------------------------------------------------------
// This descriptor set holds the Acceleration structure and the output image
//
void Raytracer::createRtDescriptorSet(const VkImageView& outputImage) { ... }
```

Рисунок 3.3.17 – Файл raytrace.cpp

```
//--------------------------------------------------------------------------------------------------
// Converting a OBJ primitive to the ray tracing geometry used for the BLAS
//
auto Raytracer::objectToVkGeometryKHR(const ObjModel& model)
{
  // Building part
  VkDeviceAddress vertexAddress = nvvk::getBufferDeviceAddress(m_device, model.vertexBuffer.buffer);
  VkDeviceAddress indexAddress  = nvvk::getBufferDeviceAddress(m_device, model.indexBuffer.buffer);

  VkAccelerationStructureGeometryTrianglesDataKHR triangles{VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_TRIANGLES_DATA_KHR};
  triangles.vertexFormat             = VK_FORMAT_R32G32B32_SFLOAT;
  triangles.vertexData.deviceAddress = vertexAddress;
  triangles.vertexStride             = sizeof(VertexObj);
  triangles.indexType                = VK_INDEX_TYPE_UINT32;
  triangles.indexData.deviceAddress  = indexAddress;
  triangles.transformData            = {};
  triangles.maxVertex                = model.nbVertices;

  // Setting up the build info of the acceleration
  VkAccelerationStructureGeometryKHR asGeom{VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_KHR};
  asGeom.geometryType       = VK_GEOMETRY_TYPE_TRIANGLES_KHR;
  asGeom.flags              = VK_GEOMETRY_NO_DUPLICATE_ANY_HIT_INVOCATION_BIT_KHR;  // For AnyHit
  asGeom.geometry.triangles = triangles;
```

Рисунок 3.3.18 – Конвертація геометричних даних моделі

```
void Raytracer::createBottomLevelAS(std::vector<ObjModel>& models, ImplInst& implicitObj)
{
  // BLAS - Storing each primitive in a geometry
  std::vector<nvvk::RaytracingBuilderKHR::BlasInput> allBlas;
  allBlas.reserve(models.size());
  for(const auto& obj : models)
  {
    auto blas = objectToVkGeometryKHR(obj);

    allBlas.emplace_back(blas);
  }

  // Adding implicit
  if(!implicitObj.objImpl.empty())
  {
    auto blas = implicitToVkGeometryKHR(implicitObj);
    allBlas.emplace_back(blas);
    implicitObj.blasId = static_cast<int>(allBlas.size() - 1);  // remember blas ID for tlas
  }


  m_rtBuilder.buildBlas(allBlas, VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_BUILD_BIT_KHR
                                     | VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_COMPACTION_BIT_KHR);
}
```

Рисунок 3.3.19 – Метод createBottomLevelAs

Шейдери трасування променів, як і шейдери растеризації, використовують зовнішні ресурси, на які посилається набір дескрипторів. За допомогою графічного конвеєра растеризації під час візуалізації сцени з використанням різних матеріалів об’єкти групуються за матеріалом. Конвеєр

матеріалу та дескриптори потрібно прив’язувати лише під час візуалізації об’єктів цього матеріалу. Щоб підтримувати сумісність між растеризацією та трасуванням променів використовується метод createRtDescriptorSet(Рис. 3.3.20), в якому містяться набір дескрипторів, що містять інформацію про сцену, набір дескрипторів, що посилається на TLAS і буфер, у якому зберігається вихідне зображення.

```
void Raytracer::createRtDescriptorSet(const VkImageView& outputImage)
{
  using vkDSLB = VkDescriptorSetLayoutBinding;

  m_rtDescSetLayoutBind.addBinding(RtxBindings::eTlas, VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR, 1,
                                   VK_SHADER_STAGE_RAYGEN_BIT_KHR | VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR);  // TLAS
  m_rtDescSetLayoutBind.addBinding(RtxBindings::eOutImage, VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, 1, VK_SHADER_STAGE_RAYGEN_BIT_KHR);  // Output image

  m_rtDescPool      = m_rtDescSetLayoutBind.createPool(m_device);
  m_rtDescSetLayout = m_rtDescSetLayoutBind.createLayout(m_device);

  VkDescriptorSetAllocateInfo allocateInfo{VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO};
  allocateInfo.descriptorPool     = m_rtDescPool;
  allocateInfo.descriptorSetCount = 1;
  allocateInfo.pSetLayouts        = &m_rtDescSetLayout;
  vkAllocateDescriptorSets(m_device, &allocateInfo, &m_rtDescSet);

  VkAccelerationStructureKHR                   tlas = m_rtBuilder.getAccelerationStructure();
  VkWriteDescriptorSetAccelerationStructureKHR descASInfo{VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_ACCELERATION_STRUCTURE_KHR};
  descASInfo.accelerationStructureCount = 1;
  descASInfo.pAccelerationStructures    = &tlas;
  VkDescriptorImageInfo imageInfo{{}, outputImage, VK_IMAGE_LAYOUT_GENERAL};
```

Рисунок 3.3.20 – Метод createRtDescriptorSet

Як і у випадку з набором дескрипторів растеризації, набір дескрипторів трасування променів потрібно оновити, якщо його вміст змінюється. Зазвичай це відбувається під час зміни розміру вікна, оскільки вихідне зображення створюється заново та потребує повторного зв’язування з набором дескрипторів. Оновлення виконується у методі updateRtDescriptorSet(Рис. 3.3.21).

```
void Raytracer::updateRtDescriptorSet(const VkImageView& outputImage)
{
  VkDescriptorImageInfo imageInfo{{}, outputImage, VK_IMAGE_LAYOUT_GENERAL};
  VkWriteDescriptorSet  wds = m_rtDescSetLayoutBind.makeWrite(m_rtDescSet, RtxBindings::eOutImage, &imageInfo);
  vkUpdateDescriptorSets(m_device, 1, &wds, 0, nullptr);
}
```

Рисунок 3.3.21 – Метод updateRtDescriptorSet

Генерація конвеєра трасування променів відбувається в методі createRtPipeline(Рисунок 3.3.22) і починається з додавання етапів генерації променів і міс шейдера, після чого йде найближчий шейдер.

```
void Raytracer::createRtPipeline(VkDescriptorSetLayout& sceneDescLayout)
{

  enum StageIndices
  {
    eRaygen,
    eMiss,
    eMiss2,
    eClosestHit,
    eAnyHit,
    eClosestHit1,
    eAnyHit1,
    eIntersect,
    eCall0,
    eCall1,
    eCall2,
    eShaderGroupCount
  };

  // All stages
  std::array<VkPipelineShaderStageCreateInfo, eShaderGroupCount> stages{};
  VkPipelineShaderStageCreateInfo stage{VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO};
  stage.pName = "main";  // All the same entry point
  // Raygen
  stage.module = nvvk::createShaderModule(m_device, nvh::loadFile("spv/raytrace.rgen.spv", true, defaultSearchPaths, true));
  stage.stage     = VK_SHADER_STAGE_RAYGEN_BIT_KHR;
  stages[eRaygen] = stage;
  // Miss
  stage.module = nvvk::createShaderModule(m_device, nvh::loadFile("spv/raytrace.rmiss.spv", true, defaultSearchPaths, true));
  stage.stage   = VK_SHADER_STAGE_MISS_BIT_KHR;
  stages[eMiss] = stage;
  // The second miss shader is invoked when a shadow ray misses the geometry. It simply indicates that no occlusion has been found
  stage.module =
      nvvk::createShaderModule(m_device, nvh::loadFile("spv/raytraceShadow.rmiss.spv", true, defaultSearchPaths, true));
  stage.stage    = VK_SHADER_STAGE_MISS_BIT_KHR;
  stages[eMiss2] = stage;
```

Рисунок 3.3.22 – Метод createRtPipeline

Усі етапи зберігаються у std::vector об’єктів VkPipelineShaderStageCreateInfo. На цьому кроці індекси в даному векторі будуть використовуватися як унікальні ідентифікатори для шейдерів. Потім створюється vkCreateShaderModule із попередньо скомпільованого шейдера та визначається якому етапу він відповідає. Ці ідентифікатори зберігаються в структурі VkRayTracingShaderGroupCreateInfoKHR (Рис. 3.3.23). Ця структура спочатку визначає тип, який представляє тип групи шейдерів, представленої в структурі.

```
// Shader groups
VkRayTracingShaderGroupCreateInfoKHR group{VK_STRUCTURE_TYPE_RAY_TRACING_SHADER_GROUP_CREATE_INFO_KHR};
group.anyHitShader       = VK_SHADER_UNUSED_KHR;
group.closestHitShader   = VK_SHADER_UNUSED_KHR;
group.generalShader      = VK_SHADER_UNUSED_KHR;
group.intersectionShader = VK_SHADER_UNUSED_KHR;

// Raygen
group.type          = VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_KHR;
group.generalShader = eRaygen;
m_rtShaderGroups.push_back(group);

// Miss
group.type          = VK_RAY_TRACING_SHADER_GROUP_TYPE_GENERAL_KHR;
group.generalShader = eMiss;
m_rtShaderGroups.push_back(group);
```

Рисунок 3.3.23 – Збереження ідентифікаторів в структурі

Перетини керуються трьома типами шейдерів: шейдер перетину обчислює перетини геометрії променів, шейдер будь-якого влучання (один із типів хіт шейдерів) запускається для кожного потенційного перетину, а шейдер найближчого влучання застосовується до найближчої точки влучання вздовж променя.

```
#include "raytrace.hpp"
#include "nvh/fileoperations.hpp"
#include "nvvk/descriptorsets_vk.hpp"

#include "nvh/alignment.hpp"
#include "nvvk/shaders_vk.hpp"
#include "obj_loader.h"
#include "nvvk/buffers_vk.hpp"

extern std::vector<std::string> defaultSearchPaths;


void Raytracer::setup(const VkDevice& device, const VkPhysicalDevice& physicalDevice, nvvk::ResourceAllocator* allocator, uint32_t queueFamily) { ... }


void Raytracer::destroy() { ... }

//--------------------------------------------------------------------------------------------------
// Converting a OBJ primitive to the ray tracing geometry used for the BLAS
//
auto Raytracer::objectToVkGeometryKHR(const ObjModel& model) { ... }


//--------------------------------------------------------------------------------------------------
// Returning the ray tracing geometry used for the BLAS, containing all spheres
//
auto Raytracer::implicitToVkGeometryKHR(const ImplInst& implicitObj) { ... }


void Raytracer::createBottomLevelAS(std::vector<ObjModel>& models, ImplInst& implicitObj) { ... }

void Raytracer::createTopLevelAS(std::vector<ObjInstance>& instances, ImplInst& implicitObj) { ... }

//--------------------------------------------------------------------------------------------------
// This descriptor set holds the Acceleration structure and the output image
//
void Raytracer::createRtDescriptorSet(const VkImageView& outputImage) { ... }
```

3.4. Опис графічного інтерфейсу розробленого ПЗ

В якості графічного інтерфейсу розробленого програмного забезпечення використовувалась бібліотека Dear ImGui

Dear ImGui (Рис. 3.4.1) — це проста бібліотека графічного інтерфейсу користувача для мови програмування C++. Ця бібліотека є досить швидкою, портативною та не залежить від засобу візуалізації. Dear ImGui особливо підходить для інтеграції в ігрові двигуни, 3D програмного забезпечення в реальному часі, повноекранних програми.

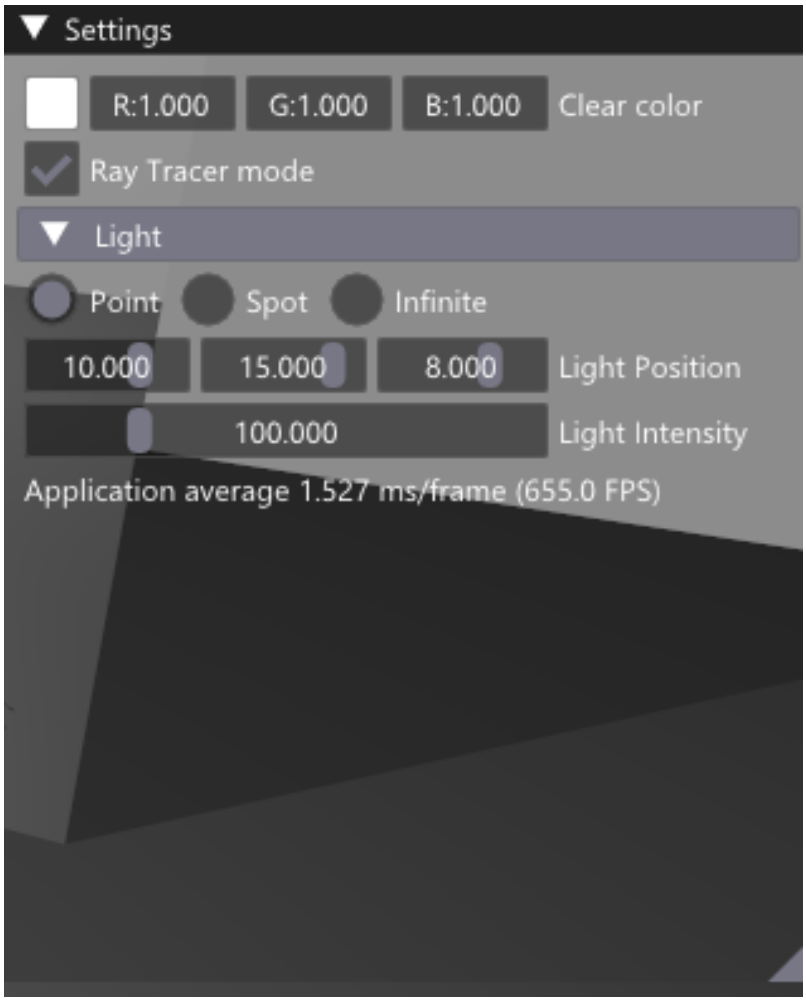


Рисунок 3.4.1 – Приклад інтерфейсу користувача imGui

Так в розробленому програмному забезпеченні присутня можливість перемикання режиму роботи візуалізатора між режимом растеризації та режимом трасування променів та зміна типу, розташування та інтенсивності джерела світла на сцені. А також присутній лічильник кадрів.

Висновки до розділу

У даному розділі була описаний підхід до розробки апаратно-програмного трасувальника променів. Було описане розроблене програмне забезпечення. Також були описані обрані програмні засоби та технологія розробки та їх основні переваги.

Так обрані технології розробки найкраще відповідають підходу до розробки програмного забезпечення, що працює з комп’ютерною графікою.

4. ТЕСТУВАННЯ РОЗРОБЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

4.1. Тестування роботи розробленого трасувальника променів

Перед тестуванням розробленого програмного забезпечення слід зазначити, що тестування відбувалось на наступній конфігурації:

- AMD Ryzen 5 4600h
- NVIDIA RTX2060 laptop gpu
- Відеопам’ять графічного процесору об’ємом 6 гігабайт
- Оперативна пам’ять об’ємом 16 гігабайт
- Операційна система Microsoft Windows 10

Робота розробленого програмного забезпечення тестувалась в різних за складністю сценах.

В найпростішій сцені (Рис. 4.1.1) присутній куб на пластині. Продуктивність розробленого програмного забезпечення становить близько 963 кадрів за секунду при роздільній здатності в 1280 на 720 пікселей.

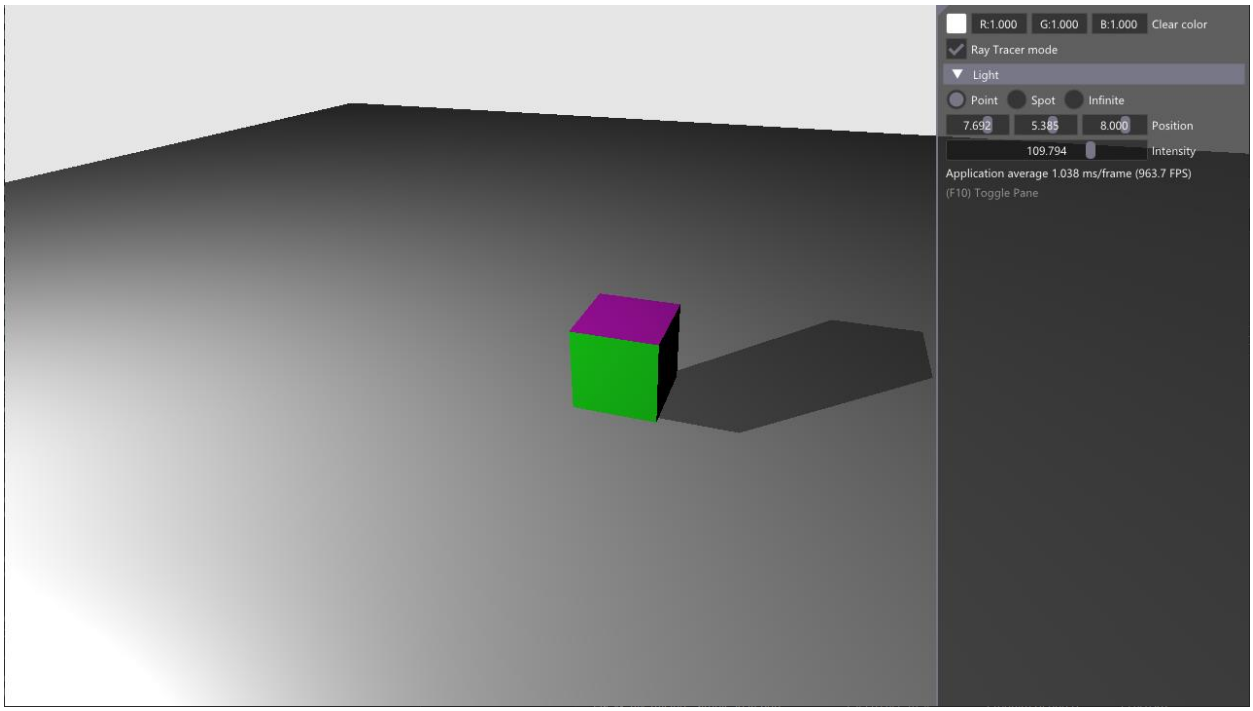


Рисунок 4.1.1 – Найпростіша сцена з матовим кубом

В наступній за складністю сцені (Рис. 4.1.2) додається лише один об’єкт – сфера. Як можна побачити з рисунку продуктивність змінюється не сильно та становить близько 942 кадрів за секунду. Так можна дійти до проміжного висновку, що при додаванні простих об’єктів без складних візуальних властивостей продуктивність, при використанні сучасного графічного обладнання, майже не змінюється.

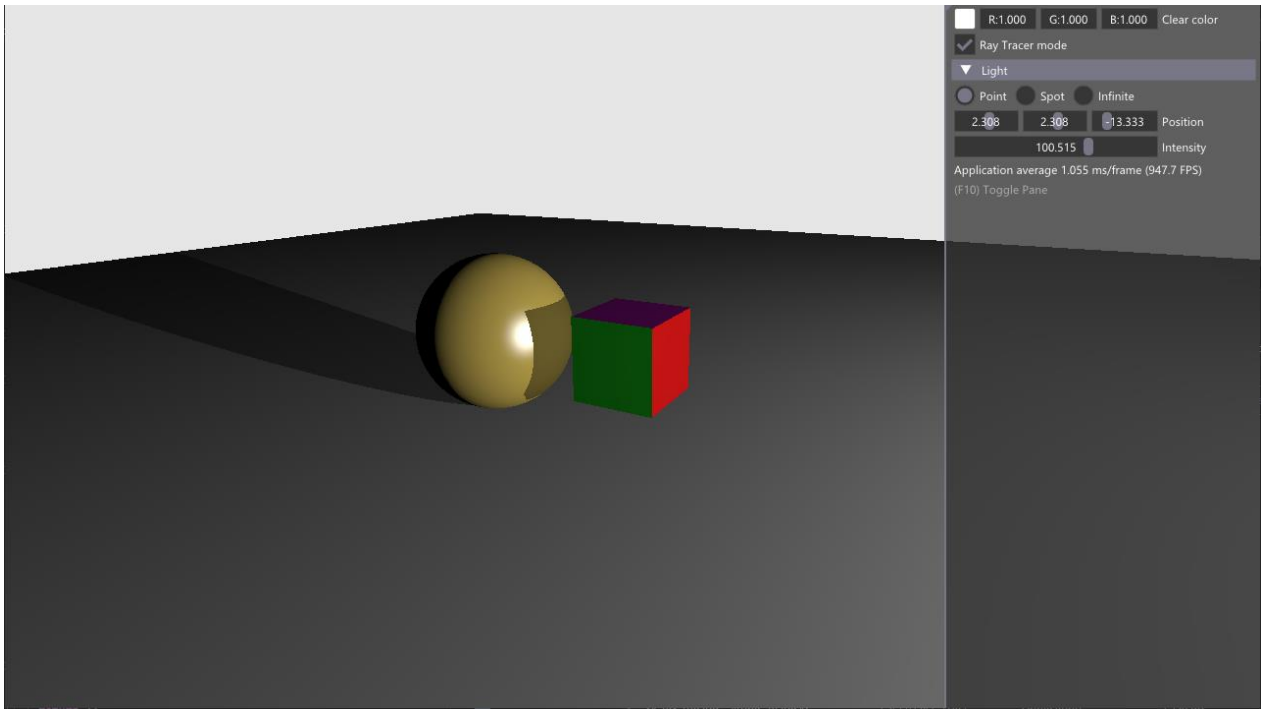


Рисунок 4.1.2 – Проста сцена з матовим кубом та не матовою сферою

Наступною сценою що тестувалась стала не статична сцена з переміщенням об’єктів (Рис. 4.1.3)

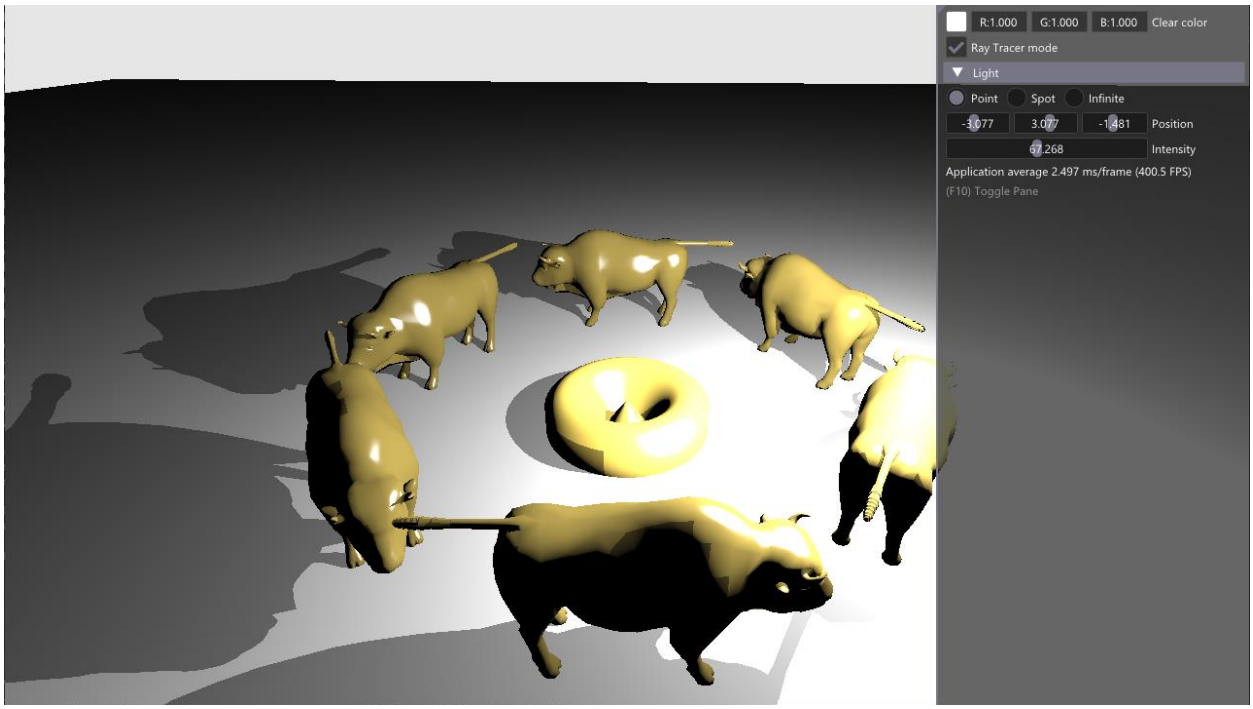


Рисунок 4.1.3 – Не статична сцена з переміщенням об’єктів

З даного тесту можна помітити, що навіть невелика кількість об’єктів, що мають властивість відблискування, при не статичній симуляції досить суттєво зменшують продуктивність розробленої системи. 400 кадрів проти 942 кадрів за секунду при повністю статичній тривимірній симуляції. Зменшення продуктивності зумовлюється тим, що об’єкти симуляції не є статичними і при зміні свого розташування в сцені потрібно вираховувати більше варіантів хіт-променів. Але, все ще, при роздільній здатності в 1280 на 720 пікселей значення продуктивності в 400 кадрів за секунду є більш ніж задовільним.

Далі розроблене програмне забезпечення тестувалось в статичній сцені, але більш складній ніж раніше(Рис. 4.1.4).

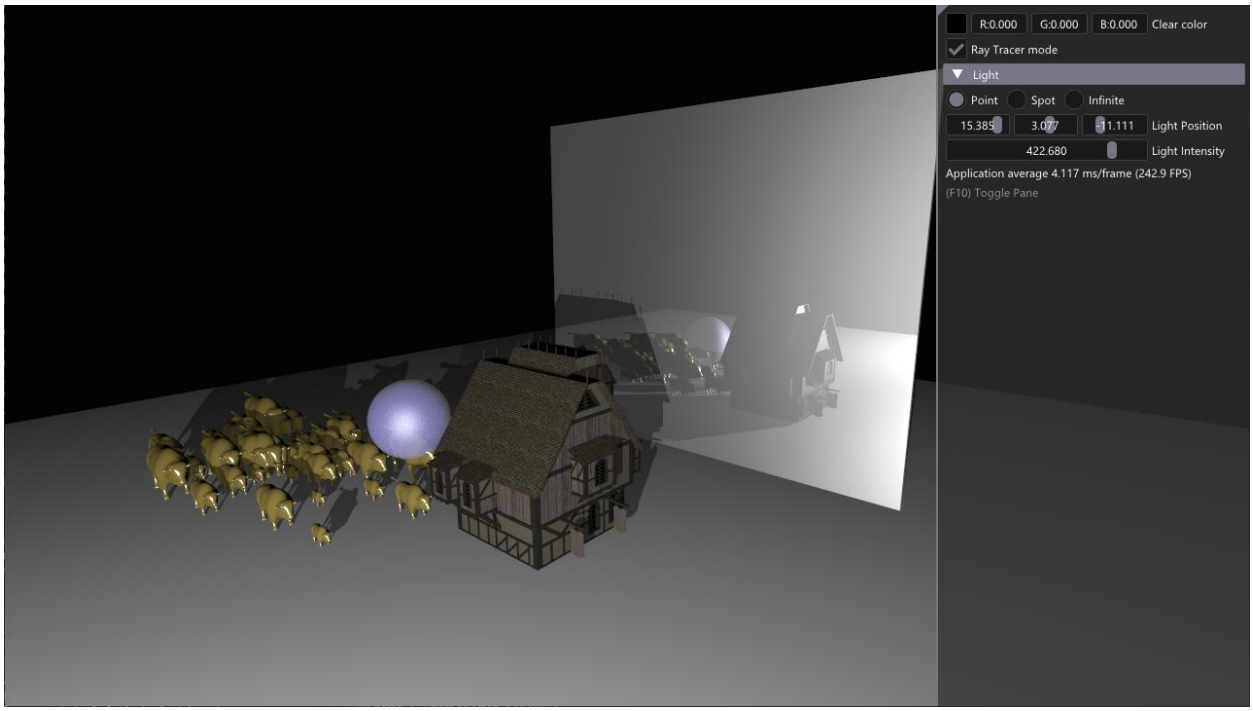


Рисунок 4.1.4 – Статична сцена з великою кількістю різних за властивостями об’єктів

В даній сцені присутні різні за властивостями об’єкти. Серед яких п’ятдесят не матових об’єктів, прозора сфера, середньовічна будівля з матовою текстурою та об’єкт, що має дзеркальні властивості. В сцені можна побачити як програмне забезпечення справляється з візуалізацією тіней багатьох об’єктів та візуалізацією відзеркалення всіх об’єктів сцени. Продуктивність при рендері даної сцени становить близько 242 кадрів за секунду.

Також дана сцена тестувалась в режимі зонованого світла (Рис. 4.1.5). На продуктивність це в загальному не вплинуло. Кількість кадрів зменшилась до 234 кадрів за секунду.

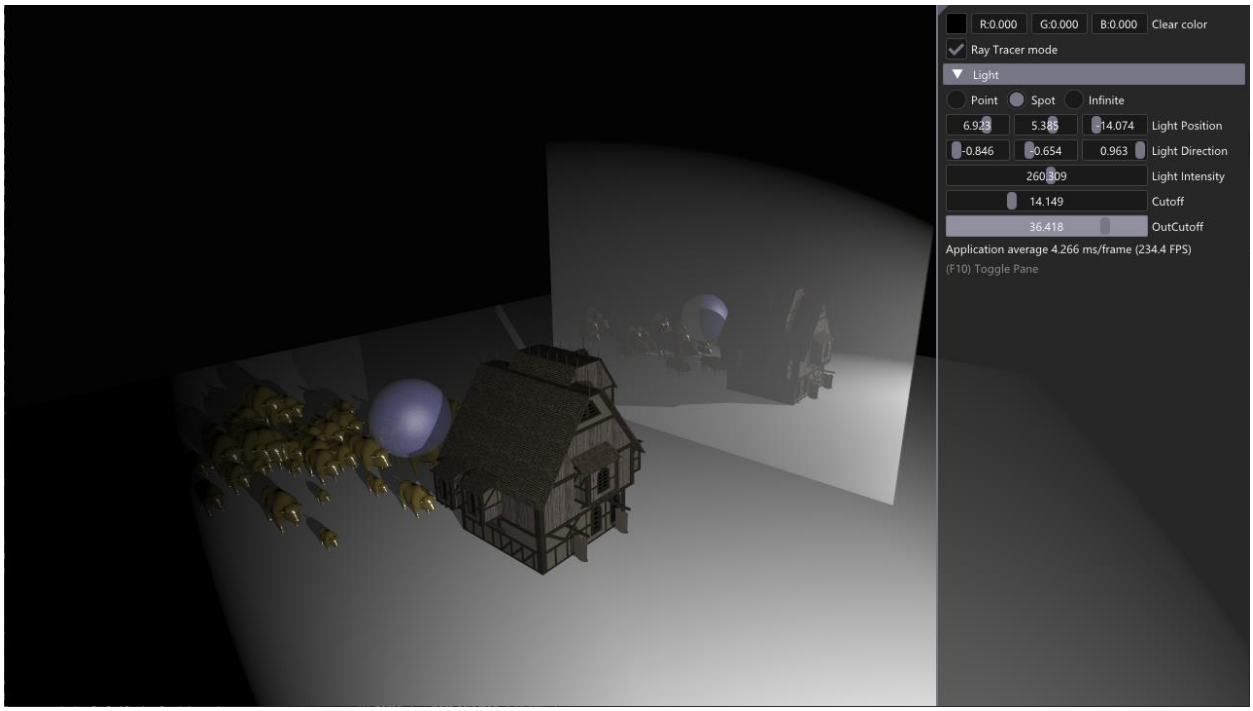


Рисунок 4.1.5 – Статична сцена з великою кількістю різних за властивостями об’єктів в режимі зонованого світла

Останньою статичною сценою що тестувалась стала сцена з непрямим освітленням (Рис. 4.1.6). В даній сцені присутній об’єкт, що є середньовічною будівлею з матовою текстурою, та сім сфер, що виступають в ролі ліхтарів. В даній симуляції можна спостерігати тіні, що були утворені від різного джерела світла, а також розсіювання світла ліхтарів на об’єктах сцени.

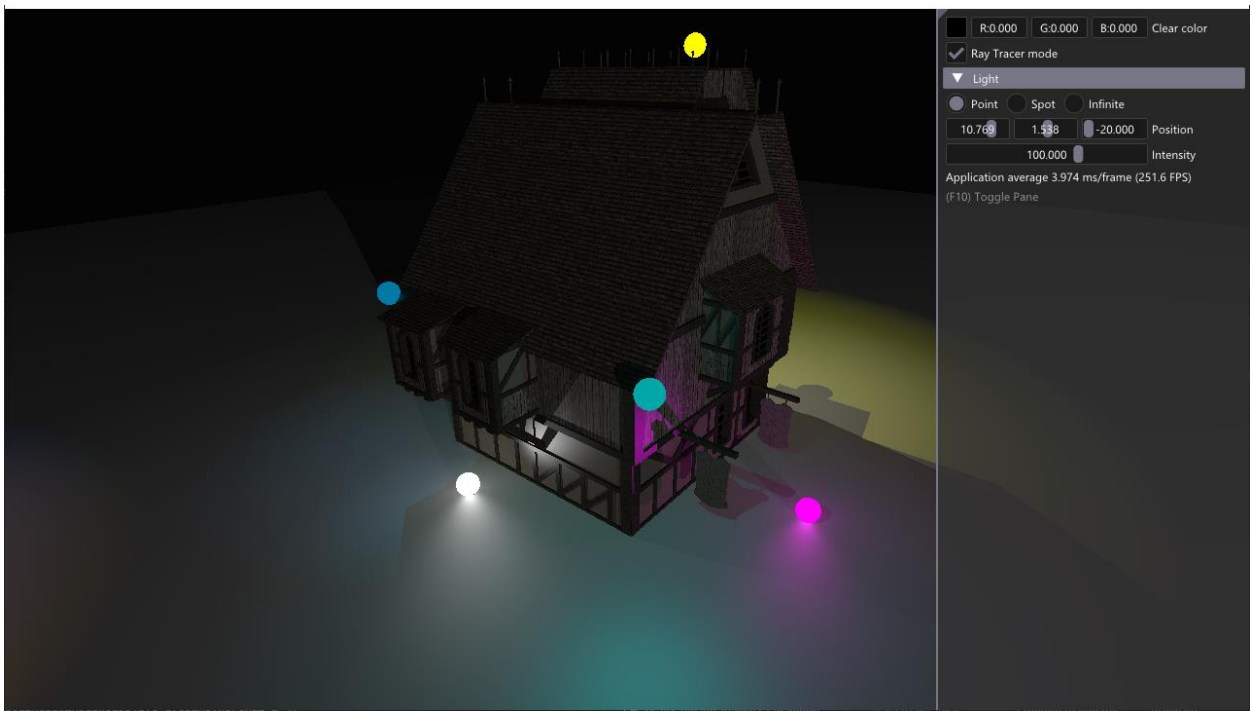


Рисунок 4.1.6 – Сцена з непрямим освітленням

Продуктивність реалізованого апаратного-програмного способу трасування променів залежить не лише від складності тривимірної симуляції, що візуалізується, та характеристик машини, на якій відбувався рендер, а й від роздільної здатності вихідної симуляції. Тому, є сенс в дослідженні впливу вихідної роздільної здатності на продуктивність фінального програмного забезпечення. Тест проводився в найбільш складній предсталеній статичній сцені.

На рисунку 4.1.7 зображено візуалізовану тривимірну сцену в роздільній здатності 640 на 480 пікселей. При даному роздільному здатності отримуємо вражаючу продуктивність, близько 931 кадру за секунду. Проте в сучасному світі така роздільна здатність не є популярною, оскільки дає не надто якісне вихідне зображення. Тому тестування проводилось в притул до сучасного користувацького стандарту в 1920 на 1080 пікселей.

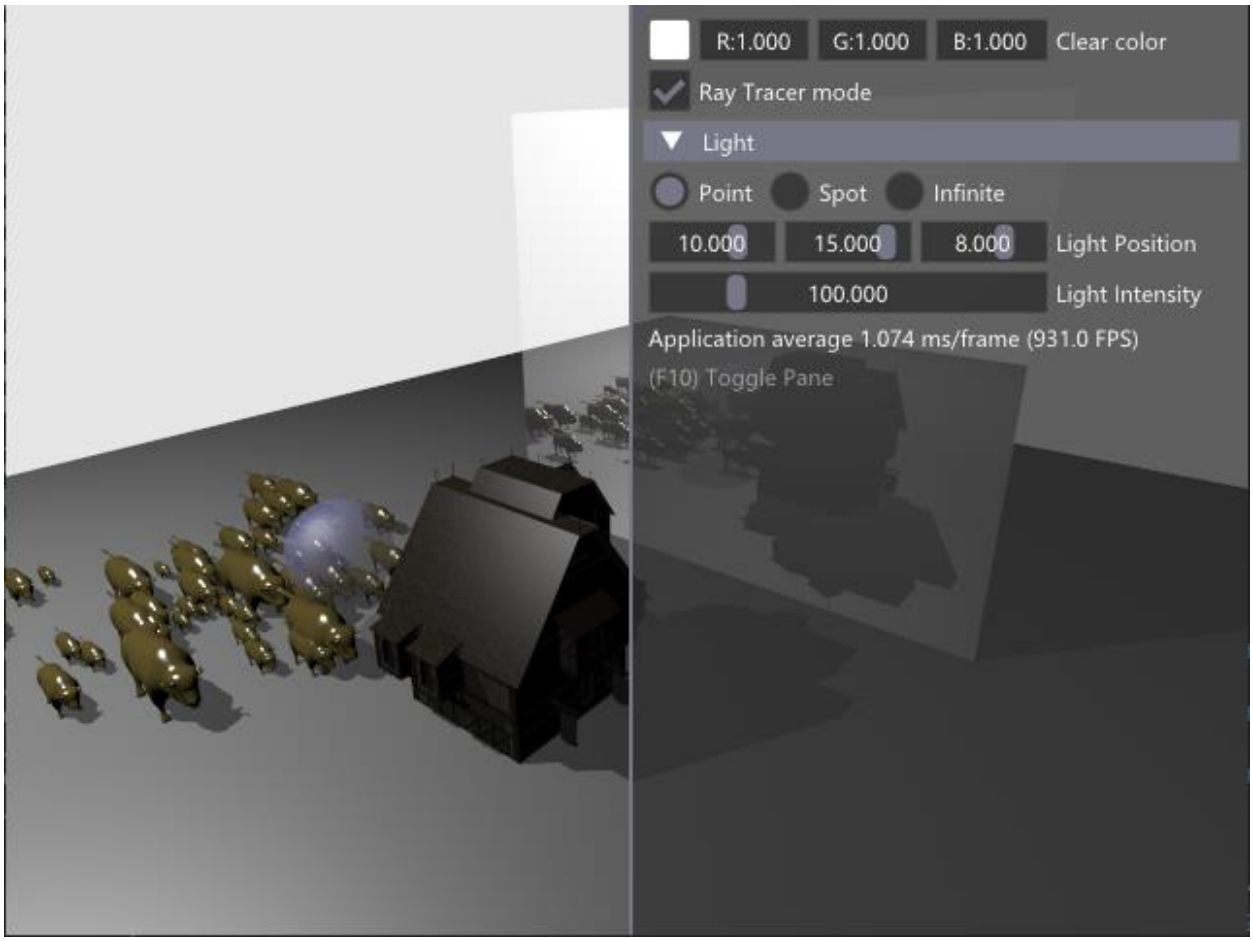
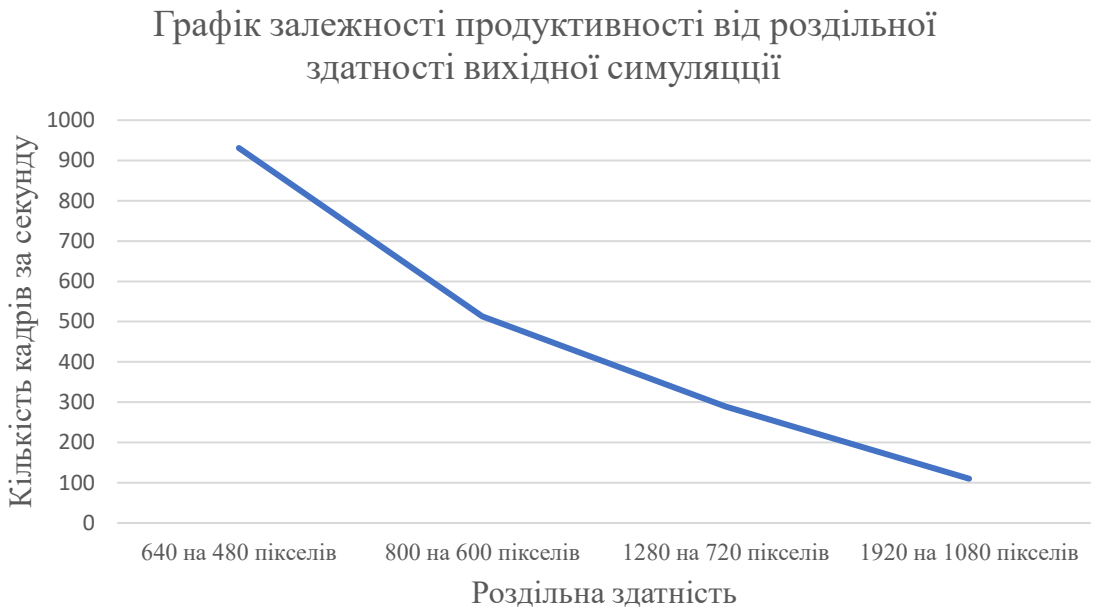


Рисунок 4.1.7 – Тривимірна сцена в роздільній здатності 640 на 480 пікселей

Таблиця 1 – Тестування впливу роздільної здатності на продуктивність

Роздільна здатність	Кількість кадрів за секунду
640 на 480 пікселів	931
800 на 600 пікселів	513
1280 на 720 пікселів	289
1920 на 1080 пікселів	110



З таблиці 1 та графіку залежності продуктивності від роздільної здатності вихідної симуляції видно, що при збільшенні роздільної здатності сцени кількість кадрів за секунду зменшується. Хоча при роздільній здатності 1920 на 1080 пікселей показник продуктивності розробленого програмного забезпечення становить 110 кадрів за секунду, далі збільшувати роздільну здатність не варто. При більш складних динамічних сценах показник продуктивності буде зменшуватись. В такому випадку при збільшенні роздільної здатності показник кількості кадрів буде опускатись до не комфортних для людського ока значень. Проте, використання більш потужного графічного обладнання може покращити продуктивність розробленого програмного забезпечення при візуалізації тривимірної сцени в більшій роздільній здатності.

4.2. Порівняння результатів роботи візуалізації з різними способами рендерингу

При порівнянні з трасувальником променів, що працює виключно на потужностях центрального процесора можемо спостерігати таку картину (Рис. 4.2.1). На рисунку 4.2.2 зображена сцена, що використовувалась в даному програмному забезпеченні.

```
==============RAY TRACER V1.0===============

 average FPS : 44.868599
```

Рисунок 4.2.1 – Продуктивність трасування променів з використанням потужностей лише центрального процесору

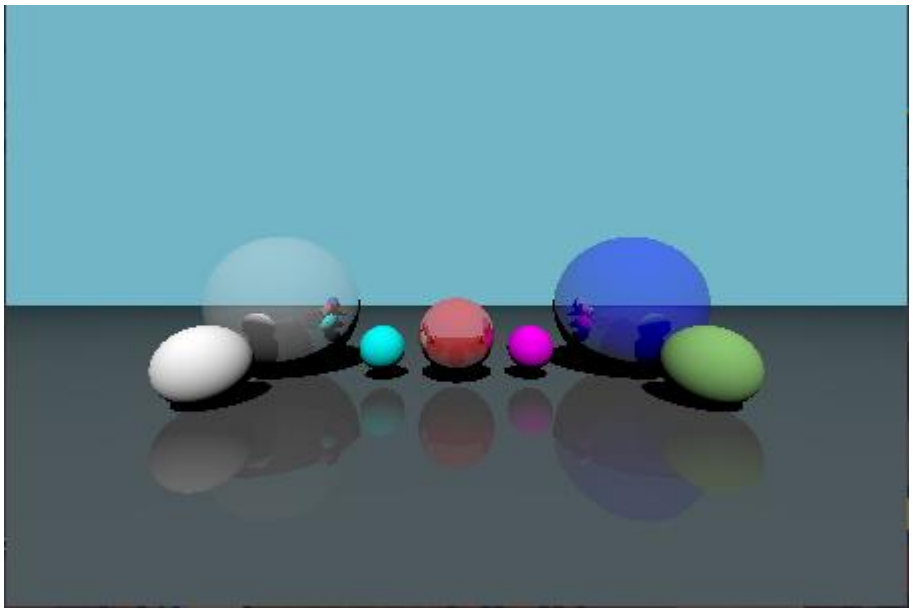


Рисунок 4.2.2 - сцена, що використовувалась в трасувальнику променів з використанням потужностей лише центрального процесору

Кількість кадрів є дуже малою, в рази меншою за кількість кадрів, що рендерить розроблене програмне забезпечення. Навіть зважаючи на те, що в

трасувальнику променів, який використовує виключно потужності CPU, тривимірна симуляція відображалась в меншій роздільній здатності (450 на 300 пікслей) та простішою сценою продуктивність даного трасувальника є досить низькою, близько 45 кадрів за секунду, тому доцільніше використовувати спосіб трасування променів, що спирається на GPU. Тому, в подальшому, порівняння розробленого програмного забезпечення буде відбуватись з трасувальниками променів, які якось опираються на потужності графічного обладнання. Тестування буде відбуватись в класичній для досліджень такого характеру сцені – корнелівська коробка.

Для порівняння використовувався трасувальник променів, що працює на основі можливостей технології CUDA.

При переглядів результатів візуалізації трасувальника променів на основі технології CUDA (Рис. 4.2.3) та розробленого програмного забезпечення, що спирається на API Vulkan (Рис. 4.2.4) отримуємо такі результати. CUDA трасувальник має продуктивність на рівні 337 кадрів за секунду, в той час як запропонований спосіб апаратного-програмного трасування променів працює з продуктивністю на рівні 473 кадрів за секунду. Тож, різниця продкутивності розробленого програмного забезпечення та трасувальника променів в реальному часі на основі CUDA становить 136 кадрів за секунду. Це свідчить про те, що в даній сцені розроблений трасуальник променів є значно продуктивнішим. Так ці результати вказують на те, що й загальна продуктивність трасувальника на основі технології CUDA є нижчою за представлений спосіб трасування променів з використанням API Vulkan.

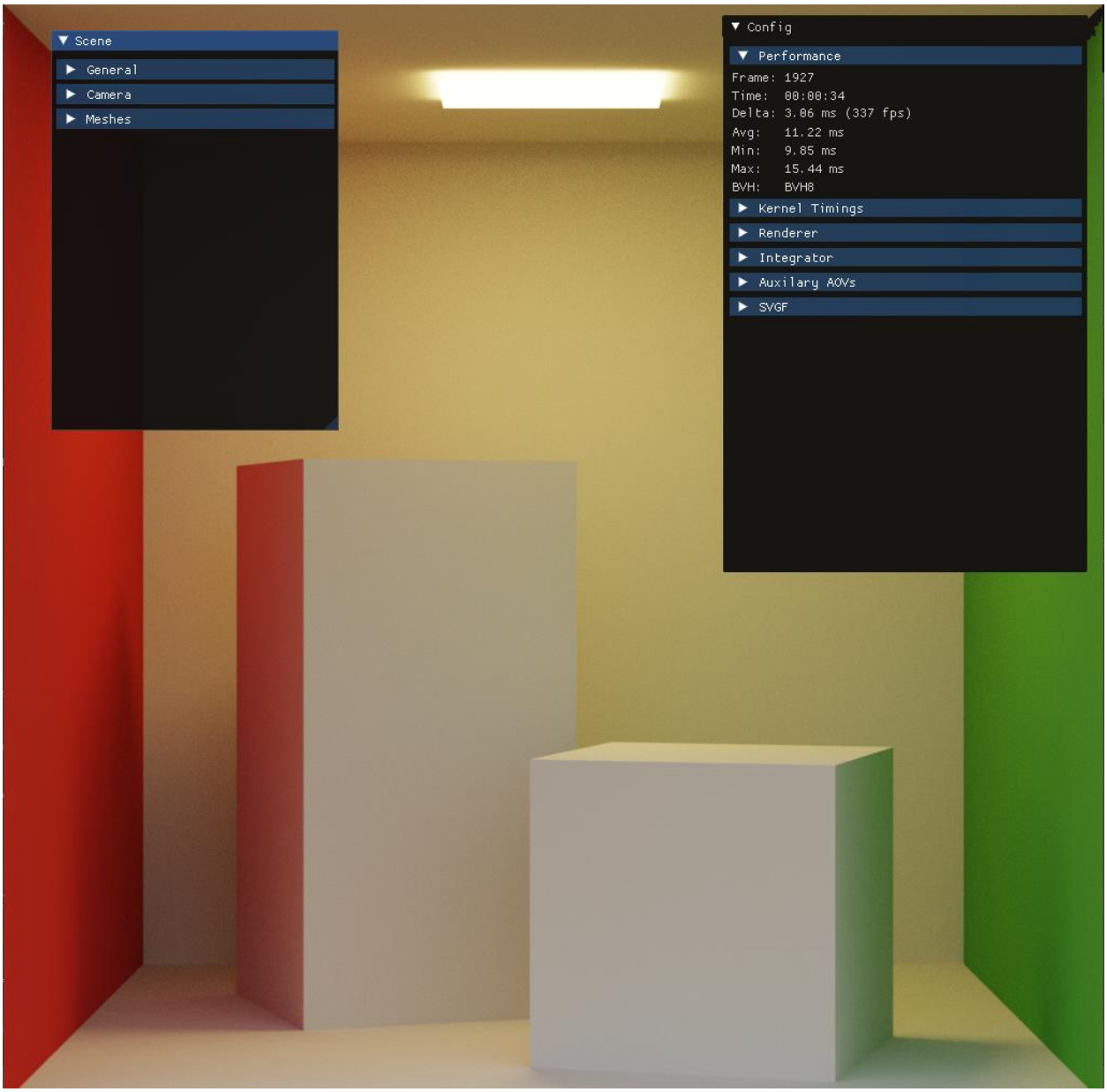


Рисунок 4.2.3 – Продуктивність трасування променів з використанням потужностей лише центрального процесору

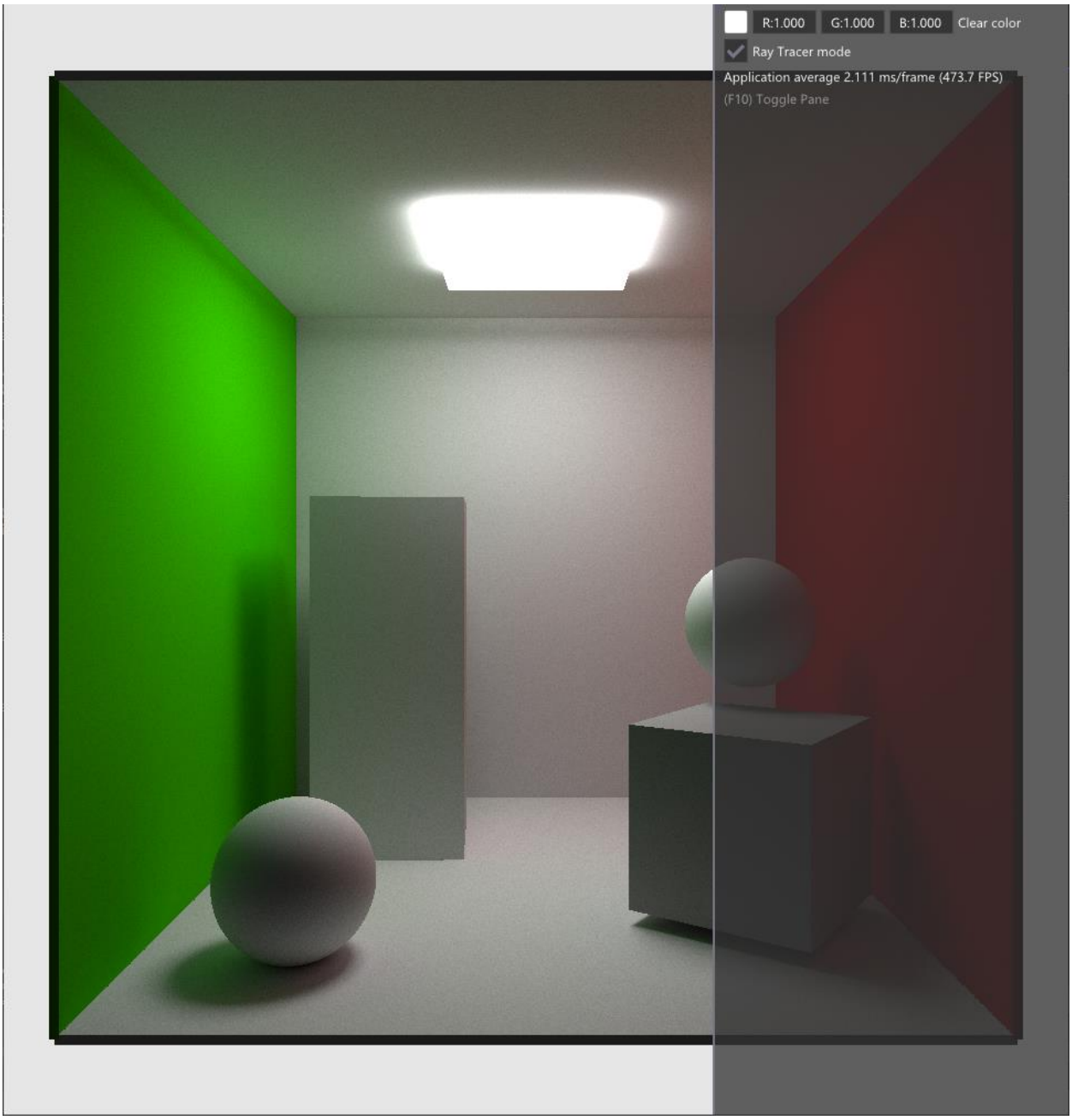


Рисунок 4.2.4 – Продуктивність трасування променів з використанням потужностей лише центрального процесору

Висновки до розділу

У даному розділі було представлене тестування реалізованого способу апаратно-програмного трасування променів. Тестування продуктивності розробленого програмного забезпечення відбувалось на різних за складністю тривимірних сцен. Також була протестована залежність продуктивності представленого ПЗ від роздільної здатності вихідної тривимірної симуляції. На основі цих тестів видно, що найбільш оптимальною роздільною здатністю є 1280 на 720 пікселів, що поєднує досить високу продуктивність та в цілому хорошу якість вихідної візуалізації. Так, роздільна здатність 1920 на 1080 пікселів, як видно з таблиці 1, теж показує достатньо високі результати, а саме 110 кадрів за секунду, та в більш складних та більш динамічних сценах показник кількості кадрів може зменшуватись надто сильно. Це стосується й більшої роздільної здатності. Також слід зазначити, що на загальну продуктивність візуалізації впливає й потужність апаратного забезпечення. Так при більш потужному графічному обладнанню показник кількості кадрів буде більшим ніж той, що був на конфігурації, що відповідала за тестування розробленого програмного забезпечення.

Також в цьому розділі присутнє порівняння продуктивності запропонованого способу візуалізації з іншими підходами до рендеру з використанням трасування променів. З результатів тестування можна зробити висновок, що використання виключно центрального процесору при роботі з алгоритмом трасування променів є не доцільною в силу низької продуктивності. В порівнянні з трасувальником, що опирається на технологію CUDA маємо різницю в продуктивності з розробленим програмним забезпеченням що становить 136 кадрів за секунду на користь запропонованого способу.

ВИСНОВКИ

В даній магістерській дисертації проводилось дослідження в галузі комп’ютерної графіки та був запропонований та реалізований спосіб апаратно-програмного трасування променів в задачах тривимірної симуляції.

Так був представлений апаратно-програмний спосіб трасування променів, що спирається на низькорівневі можливості прикладного програмного інтерфейсу Vulkan. Так, для реалізації представленого способу використовувалась структура прискорення, що поділяється на дві структури – нижнього та верхнього рівнів. Також використовувалась таблиця зв'язування шейдерів.

Прикладне програмне забезпечення було розроблене мовою програмування C++, що допомогло досягнути хорошої пордуктивності в роботі. В якості API для роботи з графікою використовувалось API Vulkan. Цей програмний інтерфейс значною мірою спростив взаємодію з графічним процесором.

Під час тестувань було визначено продуктивність розробленого трасувальника в залежності від симуляції, що візуалізувалась. Також провелось порівняння з існуючими способами трасування променів, а саме з трасувальником променів, що спирається на можливості технології CUDA та трасувальником, що працює на потужностях виключно центрального процесору. Так запропонований спосіб виявився ефективнішим за обидва способи, з якими велося порівняння.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Shirley P. Realistic Ray Tracing. A K Peters, Natick, Massachusetts, 2000. 196 с.
2. Baker, M. Pauline. Hearn Donald. Computer Graphics C Version. Prentice Hall, Upper Saddle River, New Jersey, 1986. 662 с.
3. Haines E., Akenine-Möller T. Ray Tracing Gems, 2019. 651 с.
4. Kandrot, E., Sanders, J., CUDA by Example: An Introduction to General-Purpose GPU Programming, 2010. 320 с.
5. Marrs A, Shirley P, Wald I. Ray Tracing Gems II, 2021. 914 с.
6. Мейерс С. Эффективный и современный C++. Київ, 2016. 306 с.
7. Документація з мови програмування C++: URL: https://en.cppreference.com/w/ (дата звернення: 05.11.2022).
8. Документація з ImGui: URL: https://github.com/ocornut/imgui (дата звернення: 08.11.2022).
9. Nvidia RTX: URL: https://www.hardwarezone.com.sg/featurewhat-you-need-know-about-ray-tracing-and-nvidias-turing-architecture/rt-cores-andtensor-cores (дата звернення: 19.08.2022).
10. Документація API Vulkan: URL: https://registry.khronos.org/vulkan/specs/1.3/styleguide.html (дата звернення: 22.08.2022)
11. Budge, B. C., Anderson, J. C., Grath, C., I., K. 2008. A hybrid cpu-gpu implementation for interactive ray-tracing of dynamic scenes. Tech. Rep. CSE2008-9, University of California, Davis Computer Science.