Міністерство освіти і науки України
Прикарпатський національний університет імені Василя Стефаника
Кафедра комп'ютерної інженерії та електроніки

Басок Богдан Олегович
Basok Bohdan

УДК 004:681.5

Спеціальність 123 «Комп'ютерна інженерія»
(шифр та назва спеціальності)

Кваліфікаційна робота
на здобуття освітньо-кваліфікаційного рівня бакалавр
(бакалавр, спеціаліст, магістр)

Особливості взаємодії апаратного та програмного забезпечення комп'ютера на прикладі комп'ютерної графіки в реальному часі

Features of computer hardware and software interactions while using real-time computer graphics

Науковий керівник:
кандидат фізико-математичних наук,
проректор з науково-педагогічної роботи
Запухляк Р.І.

Рецензент:
кандидат технічних наук,
доцент кафедри
Комп'ютерних наук та інформаційних систем
Горєлов В.О.

Івано-Франківськ
2023

Форм.	Зона	Поз.	Позначення	Найменування	К-ть	Прим.
			123. КІ-41	Пояснювальна записка	1	

Змн.	Арк.	№ докум.	Підп.	Дата	123. УДК 004:681.5			
Разробив		Ісайович Б.А.			Специфікація	Літ.	Арк.	Аркушів
Перевірив		Когут І.Т.					2	

АНОТАЦІЯ

Люди часто зустрічаються з результатами роботи комп’ютерної графіки в реальному часі. В роботі буде розглянуто не тільки комп’ютерну графіку і принципи її роботи, але і її зв'язок з апаратним забезпеченням. Буде створено власне програмне забезпечення, яке працює з комп’ютерною графікою. Створення програмного забезпечення подібної складності не тільки демонструє засвоєння матеріалу і проведене дослідження галузі комп’ютерної графіки, але й буде використовуватись для демонстрації описаних в роботі алгоритмів. З допомогою цією програми є можливим опис складних тем в зрозумілий спосіб.

ABSTRACT

People often encounter the results of real-time computer graphics. This paper will not only take a look at computer graphics and explain the fundamentals, it will also explain the communication between computer graphics and computer hardware. The creation of computer software of such complexity not only demonstrates my success in obtaining knowledge and researching and entire industry of computer graphics, the software itself can also be used as a mean to demonstrate mentioned algorithms. Using this software, it is possible to explain complex subjects in a comprehensible manner.

Державний вищий навчальний заклад
«Прикарпатський національний університет імені Василя Стефаника»
Фізико-технічний факультет
Кафедра комп’ютерної інженерії та електроніки

Пояснювальна записка
до кваліфікаційної роботи на тему
Особливості взаємодії апаратного та програмного забезпечення комп’ютера на прикладі комп’ютерної графіки в реальному часі

					123.КІ-41		
Змн.	*Арк.*	*№ докум.*	*Підпис*	*Дата*			
Розробив		Ісайович Б.А.			Пояснювальна записка	*Літ.*	*Арк.* 3 / *Аркушів* 1
Перевірив		Когут І.Т.					
Н. Контр.		.					
Затвердив							

ПЕРЕЛІК ОСНОВНИХ СКОРОЧЕНЬ

API (Application Programming Interface) – спосіб взаємодії однієї програми з іншою, набір компонентів для взаємодії [24].

RAM (Random Access Memory) – оперативна пам’ять. Вид пам’яті комп’ютера, що може надавати доступ до будь-якої комірки пам’яті для запису чи зчитування даних[23].

Текстура (Texture) – в комп’ютерній графіці це 2D зображення, що використовуватись для різних завдань і використовуватись як основа для певних обрахунків в комп’ютерній графіці[2].

Рендеринг (Rendering) – в перекладі з англійської проявлення. В сфері комп’ютерної графіки це процес створення остаточного зображення за допомогою комп’ютерної програми. Його фінальний результат називають рендер(Render)[43].

Пре-рендеринг (Pre-render) – повільний процес створення остаточного зображення комп’ютером. Використовується в традиційні комп’ютерній графіці. Префікс “пре” від того що зображення вже пройшло процес попереднього прорахунку і ніяких обрахунків вже не проходить[41].

3D модель – репрезентація поверхні чи об’єкта на основі математики і системи координат, множину можуть називати як і 3D моделі чи меші так і геометрією[23].

Меш (Mesh) – в перекладі з англійської сітка, більш поширене слово для 3D моделі[323].

Сцена – це набір таких об’єктів як геометрія, джерела світла, камери і всього необхідного для створення остаточного зображення[23].

Полігон – багатокутник. Використовується для побудови тривимірних об’єктів[1].

Шейдер (Shader) –код, що є невеликою програмою на окремому від основної програми потоці виконання і відповідає за реакцію поверхні на світло. Його принципи роботи значно відрізняються в графіці реального часу і в традиційній графіці[3].

Зміст

Вступ

Кожен день ми зустрічаємо результати роботи комп’ютерної графіки в реальному часі. Розуміння роботи чогось настільки поширеного є однозначно корисним для нас. В цій роботі буде розглянути не тільки комп’ютерну графіку і принципи її роботи, але і її зв'язок з апаратним забезпеченням. Будуть пояснені особливості взаємодії апаратного і програмного забезпечення комп’ютера на прикладі комп’ютерної графіки в реальному часі. Ця дипломна робота дозволить краще зрозуміти взаємодію програми і апаратного забезпечення.

1.Комп’ютерна графіка статичного виду

Комп’ютерна графіка статичного виду – це поширений підхід до створення комп’ютерної графіки для цілей, що потребують високого рівня якості і деталізації фінального результату, наприклад створення фільму. Для подібних цілей використовується комп’ютерна графіка традиційного виду, тобто результат величезної кількості математичних обрахунків комп’ютером чи навіть суперкомп’ютером з метою досягнення максимальної якості зображення. Обрахунки такої складності займають величезну кількість часу і процес генерації зображення може тривати години чи дні. Термін “комп’ютерна графіка” може мати велику кількість значень і його навіть використовують для опису всього зображеного на комп’ютері, що не є аудіо чи текстом. Проте точнішим визначенням терміну буде генерація чи маніпуляція з зображеннями.

Причини і цілі використання подібного можуть бути різними. Один з прикладів використання – це індустрія кіно, хоча, звісно, на цій галузі воно не завершується. Комп’ютерна графіка використовується в великій кількості різних сфер. Створення зображень комп’ютером можливо використовувати для створення деталей, сцен чи спецефектів у фільмі, що можуть бути надто складними чи дорогими, а то й узагалі неможливими для створення в реальності за допомогою альтернативних методів. Крім цього воно може бути чудовим способом для візуалізації даних. Можливі застосування комп’ютерної графіки фактично не мають меж.

Комп’ютерна графіка сильно відрізняється від картини художника і від фотографії, але може виглядати ідентично до них. Комп’ютерна графіка має свої сильні і слабкі сторони, переваги і недоліки.

Принципи роботи подібної технології не тільки змінювались з часом, але і сильно залежать від конкретної реалізації технології, тобто різні компанії чи розробники будуть мати різні програми, що працюють в різний спосіб. Звісно це не означає, що вони не мають нічого спільного, що немає поширених методів і

принципів роботи, які можна навести у приклад.

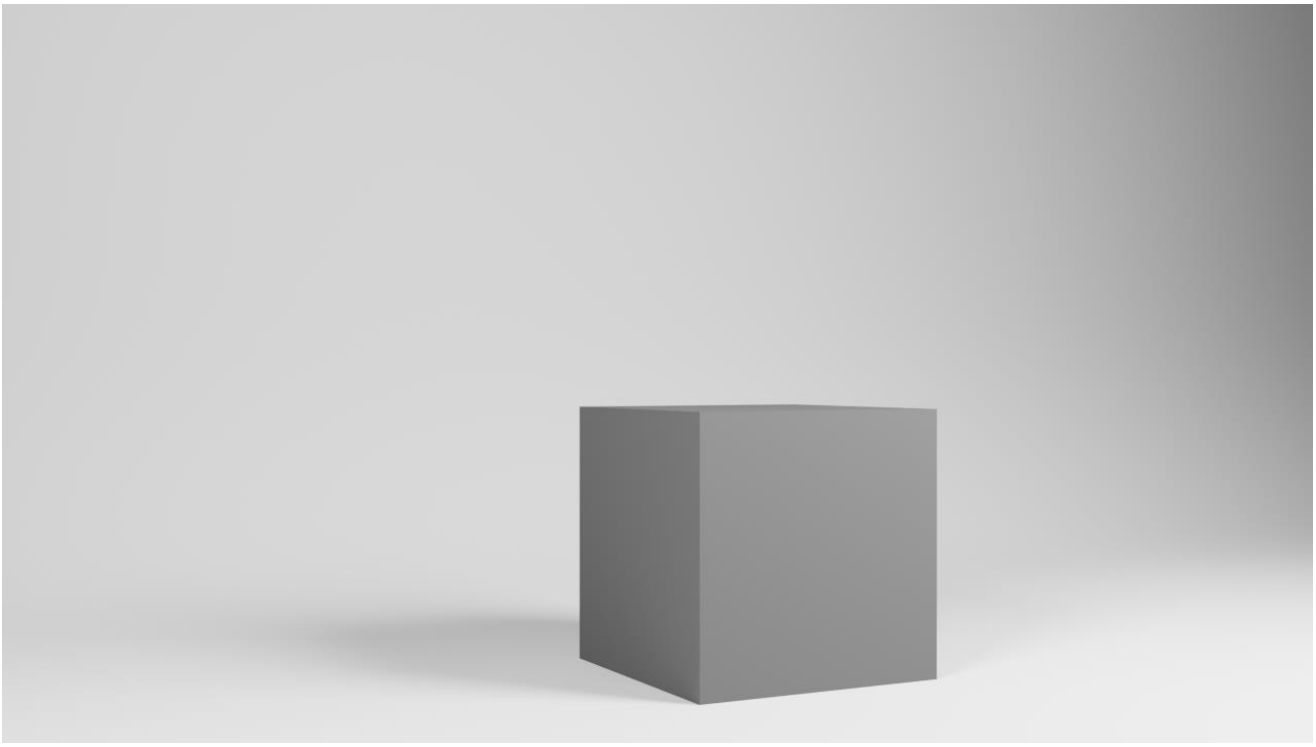


Рисунок 1.1 – Створений в комп’ютерній графіці куб.

1.1. Порівняння з іншими некомп’ютерними видами графіки

Особливість комп’ютерної графіки, що виділяє її від фотографій чи картин фарбами – це використання комп’ютера для точних обрахунків. В процесі створення зображення комп’ютер виконує необхідні математичні обрахунки для генерації кінцевого результату, тобто зображення. Математичні обрахунки комп’ютера створені з метою імітації реального світу, вони мають високу точність і потребують великої кількості часу. Коли художник зображає тінь від Сонця, що відкидає дерево, він повинен сам визначити де і як її зображати. Комп’ютер в роботі 3D графіки також визначає як має виглядати тінь від дерева, але завдяки своїм перевагам над людиною в розрахункових потужностях він може імітувати реальну роботу сонячного світла з дуже великою точністю. Звісно, це не робить роботу комп’ютера автоматично кращою і не означає, що робота художника буде неприємна оку, але це дозволяє простіше отримати реалістичний результат. Разом з цим, комп’ютерна графіка залишає за собою можливість створення неможливого, тобто в прикладі з деревом його листя можна зробити фіолетовими, а саме дерево можна зобразити в повітрі, що було б складно здійснити фотографу.

1.2. Основа комп’ютерної графіки

Основний принцип роботи тривимірної комп’ютерної графіки – це використання тривимірних геометричних даних і будь-яких додаткових даних, що збережені в комп’ютері з метою проведення обрахунків і створення цифрових зображень.

Традиційна комп’ютерна графіка може використовувати центральний процесор або графічний процесор для своїх обрахунків [1]. Кожен з процесорів має свої переваги і причини для використання. Можливе одночасне використання двох процесорів в обрахунках, але створення подібної програми є значно складнішим. В основі своїй обидва процесори створені для різного роду інструкцій, включно з математичними обрахунками. Програма на основі центрального процесора є простішою для розробки, а графічний процесор має перевагу в швидкості, але тільки у випадку, якщо обрахунки можливо розбити на мільйони менших, що, звісно, потребує спеціально зроблених під це алгоритмів, реалізація яких може бути складною.

Дані, що в собі містить програма і які необхідні для створення фінального результату завантажуються в пам’ять. Залежно від типу використаного процесору, дані завантажуються у відповідну пам’ять: оперативну чи відеопам’ять. Для обчислення графіки часто використовують спеціально створені під це системи і апаратне забезпечення, особливістю яких є великий обсяг пам’яті. Для роботи необхідно завантажити всі ресурси у пам’ять системи , а отже доступна кількість пам’яті є критичною для роботи.

Одним із недоліків цього процесу є висока складність обрахунків. Навіть найпростіші зображення доволі низької якості можуть займати хвилини часу і навіть довше на потужних комп’ютерах. В індустрії кіно для створення фільмів стандартним є час у 8 годин на проявлення одного зображення, частота кадрів у фільмах 24 і більше кадри в секунду, а тому для обрахунків використовують суперкомп’ютери. Для прикладу, фільм “Аватар” Джеймса Кемерона 2009-го року використовував суперкомп’ютер площею в 972м^2 і з оперативною пам’яттю в 100 терабайт, а також розміром диску в 2.5 петабайти. Незважаючи на це, йому потрібно було приблизно сім з половиною годин на один кадр.

1.3. 3D моделі

3D модель – це репрезентація об’єкту з допомогою математики на основі системи координат в тривимірному просторі [1]. Вони створюються, використовуючи спеціалізоване програмне забезпечення і складаються з точок, трикутників, країв, полігонів або ліній. Незважаючи на це, в своїй основі 3D модель – це, в першу чергу, набір трикутників. Для видимості 3D модель повинна бути як мінімум двовимірною, а найпростіший двовимірний об’єкт це трикутник.

Одна точка в тривимірній графіці, яка відповідає певним координатам, не має ніяких розмірів, вона не має висоти, ширини, довжини і відповідно не є видимою. Дві точки, де кожна з них відповідає певним координатам в тривимірному просторі і які є поєднанні між собою формують лінію. Ця лінія вже має довжину і є одновимірною. Проте подібна лінія досі не є видимою. Коли є три точки в тривимірному просторі, то ці три точки формують трикутник. Трикутник вже є двовимірним об’єктом і є видимим для людського ока.

Раніше в комп’ютерній графіці використовували чотирикутники як основу, але трикутники мали певну кількість переваг перед ними. Одна з переваг в тому, що чотирикутник можна створити з двох трикутників, що зменшувало необхідність використовувати чотирикутники як основу всього. Друга значна перевага трикутників – те, що вони гарантовано плоскі. Тобто, якщо взяти три точки, розташовані де завгодно в тривимірному просторі, об’єднати їх в трикутник, результатом буде плоска поверхня. Подібне не можна гарантувати з чотирикутниками, що в свою чергу ускладнювало роботу. Також, всі тривимірні об’єкти можна створити з трикутників. Сам трикутник – це ще й проста геометрична форма, що полегшує роботу комп’ютера.

Крім цього, кожен трикутник має в собі нормаль. Нормаль – це перпендикулярна до трикутника лінія, що використовується для визначення його орієнтації в просторі і в розрахунках освітлення. Комбінації подібних трикутників складають собою 3D модель. Хоча сама модель може в собі містити більшу кількість даних, таких як дані матеріалу чи анімаційні дані.

Рисунок 1.2 – Візуалізація сітки трикутників 3D моделі.

1.4. Шейдери

На даному етапі 3D модель є лишень сукупністю геометричних даних. Відповідно, набір геометричних даних просто так не може взаємодіяти зі світлом. Для взаємодії зі світлом необхідно виконати додаткову роботу, а саме створити шейдери.

Шейдер – це код, що прораховує взаємодію поверхні зі світлом під час рендеру. Шейдер можна вважати матеріалом поверхні, від нього залежать фізичні властивості поверхні, що схоже на матеріали поверхонь в реальності. Ранні шейдери були простішими і мали обмежену взаємодію зі світлом. Проте, з часом їх складність зростала і поступово наближалась до імітації реального світу. Шейдер відповідає за роботу поверхні зі світлом, за всю роботу конкретної поверхні, тобто від кольору поверхні до кількості світла, яку вона відбиває. Він вирішує такі властивості як прозорість поверхні і те, чи сама поверхня виділяє світло. Властивості шейдерів можуть надавати поверхні великої кількості властивостей.

Код можливо писати в різний спосіб, що робить код шейдерів унікальним від розробника до розробника. Навіть використовуючи одинакові формули,

можливо мати різний код і різний результат. Можливо не тільки міняти формули чи створювати власні, можна навіть відкидати зайві частини обрахунків з метою оптимізації.

1.5 Текстури

Свої властивості шейдери зберігають в чисельному виді. Наприклад колір – це комбінація червоного, зеленого і синього, тобто RGB, а іноді й альфа-каналу, що відповідає за прозорість.

За колір відповідають три беззнакові 8-ми бітні числа від 0 до 255. Шейдери застосовуються до всього об’єкта або певного набору трикутників, що в свою чергу робить цілу поверхню одного кольору. Це не завжди те, що потрібно і в реальності колір поверхні може складатись не з одного кольору, але окремо зафарбовувати кожен трикутник довго, ускладнює обрахунки і погано працює на моделях низької складності.

Для виправлення цієї проблеми було створено текстури. Текстура – це двовимірне зображення, що накладається на тривимірний об’єкт з метою використання кольорових значень зображення для різного виду обрахунків. Один з прикладів використання – надання поверхні кольору. Самі текстури завантажується в пам’ять системи і використовуються звідти для обрахунків.

Текстура є двовимірною і просто так не буде накладатись на тривимірний об’єкт. Для покривання об’єкту текстурою використовують метод створення обгортки. Кожній точці надається окреме розташування в спеціальній проекції на двовимірному просторі. Для простоти можна уявити це як загортання коробки з подарунком в красиву обгортку. Цей процес зазвичай робиться вручну, проте спеціальне програмне забезпечення може спробувати виконати процес автоматично.

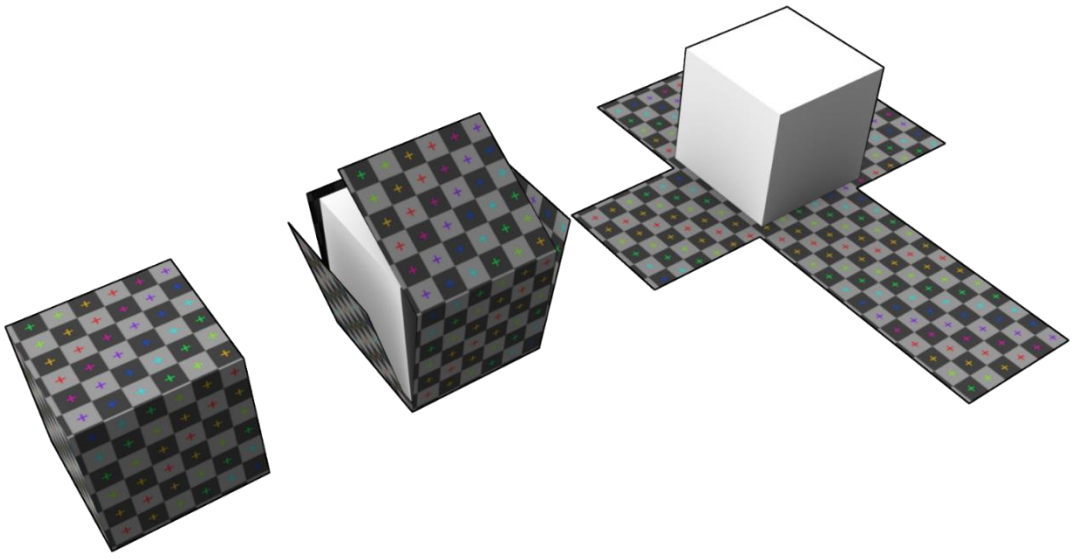


Рисунок 1.3 – Візуальна репрезентація обгортки.

1.6. Трасування шляху

Найважливішою частиною в комп’ютерній графіці є світло. В реальності, а також і в комп’ютерній графіці без світла неможливо бачити світ навколо. Залежно від обраного алгоритму обчислення, існує декілька різних методів роботи світла з комп’ютерною графікою.

Для обрахунків світла традиційна графіка використовує трасування напрямку. Трасування напрямку або ж трасування шляху – це метод рендеру зображень, який працює зі всім освітленням, яке падає на одну конкретну точку поверхні об’єкта і визначає, яка частина цього буде передана камері. Це запуск променю в певному напрямку з метою визначення його шляху. В своїй основі це не є щось унікальне для сфери графіки. Подібну ідею можна використовувати за межами обрахунків зображення, вона може використовуватись навіть при вимірюванні радіації. Промені також можна направляти у певні точки з інших, щоб перевірити, чи між цими двома точками щось знаходиться.

В ранніх алгоритмах використовували прості обрахунки. З умовної камери, яка не мала нічого спільного з реальною камерою, випускали промені і те, у що ці

промені першими попали, було на пікселі. Також додатково від точки зіткнення з поверхнею до точки джерела освітлення проводиться промінь, і залежно від результату міняється освітленість пікселя [30]. Подібний метод обрахунків був неточний, в реальності світло працює інакше. Одним з серйозних недоліків було те, що кожен піксель міг бути або освітлений, або ні. Через це результат виходив дуже контрастним і тіні не мали ніякого переходу з темніших тонів до світліших. Відсутність переходу тонів робило тіні рівномірно темними.

Рисунок 1.4 – Ранній результат роботи трасування шляху.

1.7. Сучасний стан комп’ютерної графіки

В сучасному виконанні, при зіткненні з поверхнею промінь не зупиняється, а відбивається від поверхні, як і справжній промінь світла [30]. Саме джерело освітлення має фізичний розмір, а не є точкою у просторі, що є більш наближено

до реального світу. Фізичний розмір дає джерелу світла можливість створювати кращі тіні. Залежно від розмірів самого джерела світла і кута падіння променів світла на об’єкт стають можливими м’які тіні, тобто ті, що мають плавний перехід з тіні до світла. Кожен піксель отримує певну кількість променів, що відбиваються в різних напрямках певну кількість раз і кожен з цих променів має в собі дані кольору залежно від об’єкту, в який він попадав. Коли певна кількість променів була відправлена і процес трасування для пікселя завершено, сума цих результатів дає фінальний колір пікселя. Типовим є використання двадцяти мільйонів променів, кожен з яких відбивається 32 рази [43], і так на кожен піксель зображення. Варто додати, що ці числа можуть бути значно більшими, особливо у сфері кіно.

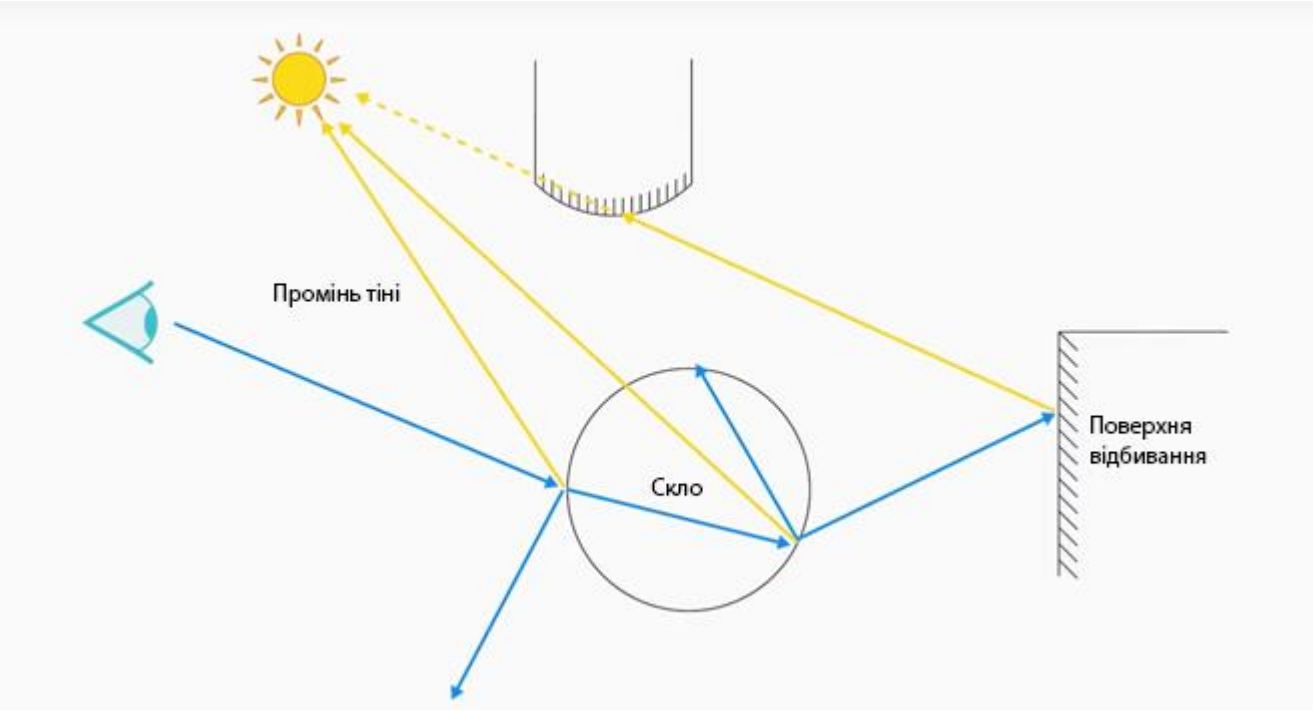


Рисунок 1.5 – Візуальна репрезентація трасування шляху.

В сучасному виконанні яскравість освітлення залежить від кута падіння променів Сонця і напрямку нормалі кожного окремого трикутника моделі, що потрапляє під його дію. Це називається законом освітленості Ламберта [35]. Закон названий в честь його автора, Йоганна Генріха Ламберта, видатного німецького науковця і він був опублікований ним в 1760 році.

$$I = I_0 \cos \theta \quad (1.1)$$

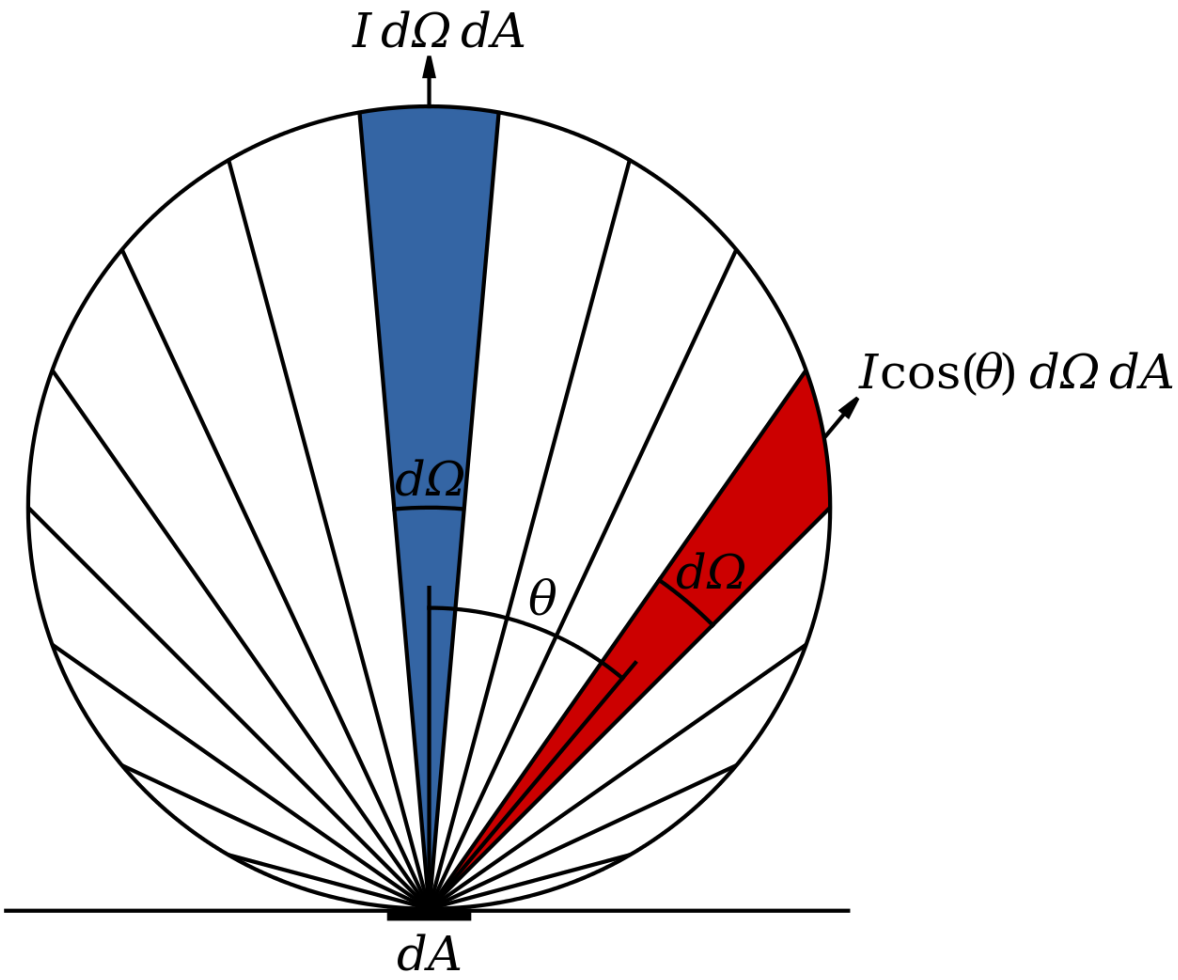


Рисунок 1.6 – Візуальна репрезентація формули Ламберта (1.1).

Формула (1.1) вираховує освітленість поверхні залежно від кута, під яким на нього падає світло. Загальний принцип полягає в тому, що якщо світло падає на поверхню під кутом, близьким до напрямку її нормалі, тобто перпендикулярно, то, відповідно, освітленість буде максимальною. Чим далі від нуля градусів відходить кут падіння світла, тим меншою буде освітленість поверхні.

Крім цього, в сучасній комп'ютерній графіці можливе використання штучного інтелекту [49]. Його використовують з метою позбавлення зображення від шуму. Шум це різного виду графічні артефакти, неточності у вигляді чорних пікселів, що появились, коли промінь не зміг дійти до джерела світла. Шум також можна виправити запуском більшої кількості променів, але це потребує додаткового часу на обрахунки. Зображення з шумом не варто таким залишати і тому зображення додатково може доробити штучний інтелект. Без використання подібного зображення буде мати різного виду неточності. В такий спосіб можливо

отримати результат за меншу кількість часу на обрахунки.

Рисунок 1.7 - Приклад роботи штучного інтелекту.

1.8. Історія розвитку комп’ютерної графіки

Різні ідеї і грубі варіанти реалізації комп’ютерної графіки можна знайти, починаючи з 1950-х років, проте справді значний прорив, що перевів її в русло реалізму і почав наближати до того, що ми знаємо сьогодні відбувся в 1970-х роках.

1.8.1. Чайник з Юти

Чайник з Юти – це нині стандартний об’єкт і дуже відомий чайник в сфері комп’ютерної графіки, настільки відомий, що його можна назвати аналогом “Hello, World!” [1]. Він є настільки популярним, що його регулярно вставляють на задні плани різного виду комп’ютерних анімацій, фільмів і не тільки. Цей чайник присутній в кожній частині відомого мультфільму “Історія іграшок”, який сам є першим фільмом, зробленим повністю з допомогою комп’ютерної графіки.

Комп’ютерна модель чайника була створена 1975 року Мартіном Ньюеллом, дослідником у сфері комп’ютерної графіки та учасником програми досліджень комп’ютерної графіки університету Юти, що знаходиться у штаті Юта, США.

Причина вибору чайника і причини його популярності доволі прості. Чайник краще за все підходив для експериментів того часу. Він має отвір у вигляді ручки, сідлову точку, тобто точку на поверхні графіка функції, де всі нахили в ортогональних напрямках дорівнюють нулю, але вона не є локальним

екстремумом функції, візуально вона може нагадувати сідло, звідки і походить назва. Чайник також може відкидати на себе тінь і може бути точно зображеним без текстур на його поверхні.

Сам чайник складається з кривих Безьє, що були винайдені французьким інженером П’єром Етьєном Безьє. Самі криві – це параметрично задані криві, вони представлені математичними функціями і піддаються контролю через зміну їх параметрів [1]. Вони також використовуються у векторній графіці для створення гладких кривих, і, що характерно для векторної графіки, їх можна масштабувати вічно, бо вони в основі своїй є просто математичними функціями.

Завдяки праці університету було винайдено багато новинок у сфері анімації, деформації предметів і було створено технологію текстур і їх накладання на об’єкт. З допомогою текстур стало можливо надавати предмету 3D графіки додаткових властивостей, що могло означати колір, блиск чи інші властивості. І все за допомогою накладання на цей предмет двовимірного зображення, що потім використовувалось для різного виду обрахунків на основі його даних. Студенти цього університету пізніше також стали засновниками таких фірм як Pixar, Adobe Inc і Silicon Graphics.

Рисунок 1.8 - чайника з Юти, сучасний рендеринг.

1.8.2. 1980-ті роки

В 1980-х роках почалась модернізація і комерціалізація комп'ютерної графіки. Почали набувати поширення комп'ютерні системи для домашнього використання. Ріст популярності персональних комп'ютерів надав більшій кількості людей можливість працювати з комп'ютерною графікою.

Відкриття у побудові комп'ютерів призвели до поширення нових 16-бітових процесорів і появи перших графічних процесорів. Це прискорило розробки в комп'ютерній графіці, а також дозволило створення зображень вищої якості.

Ціни на деякі з необхідних елементів комп'ютерів почали дешевшати. Значно дешевшою стала пам'ять системи, тобто оперативна і відеопам'ять, шо зробило роботу в галузі більш доступною.

Почали використовувати комп'ютерну графіку в фільмах. Завдяки популярним фільмам, таким як "Зоряні Війни" використання комп'ютерної графіки почалось і в сфері розваг.

В галузі створення реалістичних зображень було винайдено загальну формулу рендерингу зображень Давида Іммеля і Джеймса Каджія, що була важливим кроком до імплементації глобального освітлення, а також великим кроком в сторону реалізму.

1.8.3. Формула рендерингу

Формула рендерингу це одна із найважливіших формул в комп'ютерній графіці, на основі якої створено безліч методів обчислення зображення [35].

Сучасна реалізація формули виглядає так:

$$L_0 = (X, \widehat{\omega}_o) = L_e(X, \widehat{\omega}_o) + \int_{S^2} L_i(X, \widehat{\omega}_i) f_x(\widehat{\omega}_i, \widehat{\omega}_o) |\widehat{\omega}_i * \hat{n}| d\,\widehat{\omega}_i \qquad (1.2)$$

Де X – це точка в сцені.

$\widehat{\omega}_o$ – це вихідний напрямок світла.

$\widehat{\omega}_i$ – вхідний напрямок світла.

$\hat{n}$ – нормаль поверхні.

S^2 – це всі вхідні напрямки, що йдуть по сфері у всіх можливих напрямках.

$L_0 = (X, \widehat{\omega}_o)$ – це вихідне світло. Обравши точку і напрям, ми шукаємо вихідне світло.

$L_e(X, \widehat{\omega}_o)$ – це випромінюване світло. Обравши точку і напрям, ми шукаємо, яке світло з неї випромінюється.

$L_i(X, \widehat{\omega}_i)$ – вхідне світло. Обравши точку і напрям, ми шукаємо, яке освітлення надходить з того напряму.

$f_x(\widehat{\omega}_i, \widehat{\omega}_o)$ – матеріал. Взявши вхідне і вихідне освітлення, вираховуємо, яке освітлення іде у вихідному напрямку. Для прикладу, в дзеркала буде високий рівень вихідного освітлення.

$|\widehat{\omega}_i * \hat{n}|$ – Формула (1.1), закон Ламберта. Залежно від кута падіння світла на поверхню, його ефективність міняється. Тобто вплив джерела світла, що є перпендикулярним до поверхні, буде максимально сильним.

1.8.4. 1990-ті роки

З початком 1990-х років виросла популярність створення тривимірних моделей. Раніше це робити могли тільки дорогі системи, що коштували десятки тисяч доларів, але тоді це стало можливим і для звичайних комп'ютерів.

В цей час згенеровані зображення почали вперше наближатися до фотореалізму. Для звичайної людини робота комп'ютера почала нагадувати реальний світ. Разом з цим, комп'ютерна графіка почала використовуватись в анімації, мультимедіа і відеоіграх.

В 1995 році було випущено перший фільм, створений повністю за допомогою комп'ютерної графіки – "Історія іграшок" від Pixar. Він був неймовірно популярним і не тільки приніс компанії великі прибутки, але і популяризував використання комп'ютерної графіки в мультфільмах.

Розвиток комп'ютерних систем призвів до ще вищої якості зображень. Приріст швидкодії комп'ютерів не тільки прискорив процес обрахунків, але і дав можливість використовувати складніші алгоритми обчислення, що наблизило комп'ютерну графіку до фотореалізму.

1.8.5. Сучасний стан комп'ютерної графіки

Зараз, а насправді починаючи з 2010 років традиційна комп'ютерна графіка досягла майже повного реалізму. Формули обрахунків стали ближчими до реальних формул фізики.

Використання комп’ютерної графіки почалось в більшій кількості галузей. Майже всі фільми тепер використовують комп’ютерну графіку. Майже всі мультиплікаційні вироби тепер використовують комп’ютерну графіку. Комп’ютерна графіка є всюди навколо нас: в рекламі, у телебаченні та роботі.

Сам процес створення комп’ютерної графіки має більшу доступність ніж раніше. Звичайні системи користувачів можуть створювати фотореалістичні зображення, що раніше без суперкомп’ютерів було неможливо.

2. Комп’ютерна графіка в реальному часі

Комп’ютерна графіка в реальному часі – це частина комп’ютерної графіки, що намагається створювати зображення з високою швидкістю і має інтерактивність. Комп’ютерною графікою в реальному часі можуть вважати графічний інтерфейс користувача, але насправді це інтерактивна тривимірна комп’ютерна графіка, що зазвичай для своєї роботи використовує графічний процесор.

Фундаментально комп’ютерна графіка вважається графікою в реальному часі, якщо в неї є певна інтерактивність, а також певний рівень швидкодії [23]. Швидкодія виміряється в кадрах в секунду, тобто скільки зображень генерується програмою в секунду. В сфері кіно використовують стандарт в 24 кадри за секунду, в теорії необхідно 15 і вище кадрів для імітації руху зображення, але було виявлено, що частота кадрів нижча ніж 24 кадри за секунду виглядає неприємно для більшості людей [41]. В сфері графіки реального часу мінімальним стандартом є 30 кадрів в секунду. Цей стандарт походить від апаратних обмежень старих телевізорів. Вища частота оновлення робить зміну зображень менш помітною, що забезпечує зручнішу взаємодію з програмою для користувача. Звісно, частота оновлення також може бути обмежена частотою оновлення монітору, або будь-якого іншого дисплею. Якщо частота оновлення програми відповідає стандарту в 30 кадрів і вище, а також має інтерактивність, то вона використовує графіку реального часу.

Крім інтерактивності, необхідна і висока швидкість обрахунків, що призводить до спрощення загальних формул обрахунків, а якість фінального результату є значно нижчою, ніж у традиційній графіці. Прискорення від 7 годин роботи суперкомп’ютера до 10-30 мілісекунд звичайного комп’ютера вимагає

великої кількості змін в алгоритмах обчислення.

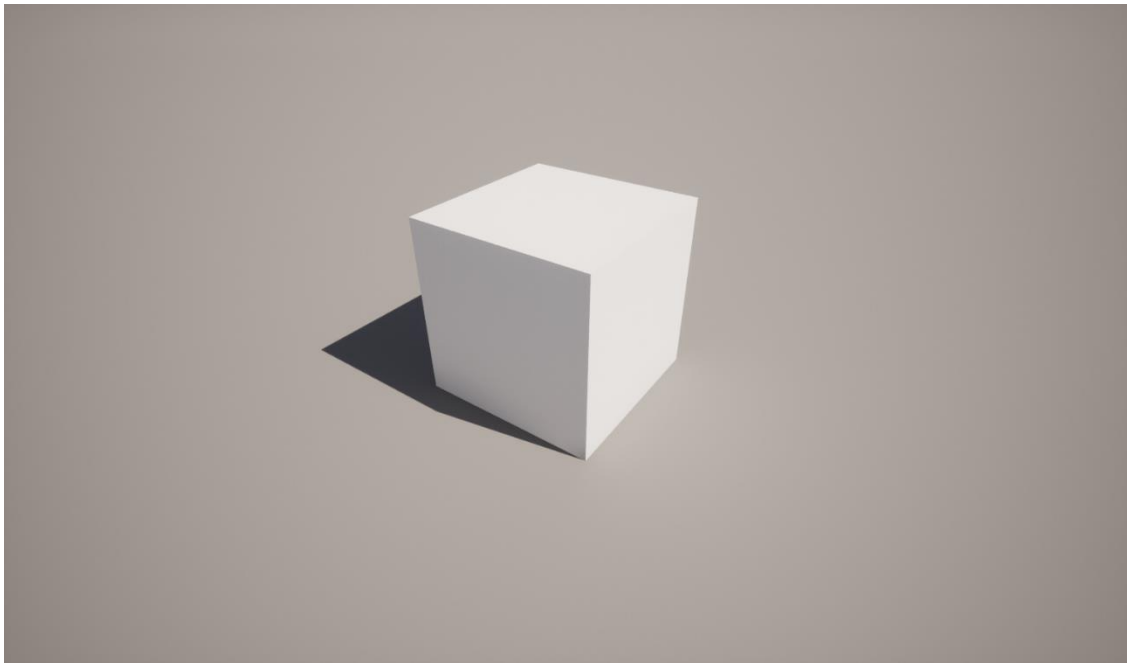


Рисунок 2.1 – Приклад комп’ютерної графіки в реальному часі.

2.1. Переваги комп’ютерної графіки реального часу і причини використання

Важливою перевагою графіки в реальному часі є швидкість. При створенні фільмів, для одного кадру необхідні години, а на створення секунд фільму необхідні дні. Для створення декількох секунд фільму необхідно багато часу , а перегляд готового продукту необхідний для перевірки результату роботи команди і внесення змін. На відміну від традиційної графіки, графіка реального часу може надати результат зразу і без довгих очікувань, що значно прискорює роботу з нею.

На відміну від пре-рендерингу, що для створення короткого, але повноцінного фільму використовує суперкомп’ютери, для графіки реального часу може вистачити і звичайного персонального комп’ютера. Низький поріг входження в свою чергу відкриває доступ до галузі для більшої кількості людей. Більша доступність галузі дає більшій кількості людей можливість виразити себе в унікальний спосіб з використанням комп’ютерної графіки.

Навіть для великих корпорацій графіка реального часу дає можливість створювати зображення достатньо високої якості, що можуть навіть змагатись з пре-рендерингом, але зберігають за собою високу швидкість обрахунків. Традиційна графіка дає високу якість зображення, але коли якість не є критичною,

то розумно буде скористатись графікою реального часу, яка є швидшою і дешевшою.

Важливою є і інтерактивність. Висока швидкість роботи дозволяє взаємодіяти з користувачем і швидко реагувати на його взаємодію з програмою. Подібне можливо застосовувати у великій кількості галузей. Для прикладу, це можуть бути окуляри додаткової реальності, що виводять додаткову інформацію про реальний світ в окуляри і допомагають в роботі інженерам. Віртуальна реальність зазвичай використовується у сфері освіти, для воєнних тренувань, для навчання медиків, для навчання пілотів і в автомобільній індустрії з метою симуляції поведінки автомобіля на дорозі.

2.2. Загальні принципи роботи

Програма, що створена з метою генерації графіки реального часу, працює в першу чергу з даними [23]. Залежно від цілей, це можуть бути тривимірні моделі і текстури. Алгоритм генерації зображення залежить від самої програми і її розробників. Кінцевим результатом вона повинна створювати зображення і реагувати на дії користувача.

На відміну від традиційної комп'ютерної графіки, графіка реального часу повинна прораховуватись з певною швидкістю, отже є необхідність оптимізації процесу обрахунків для максимальної швидкості. Цей процес погіршує можливу якість зображення, але надає програмі вищої швидкості.

Для комп'ютерної графіки в реальному часі необхідна швидкодія і інтерактивність. Формули і алгоритми, хоч і відрізняються від пре-рендерингу, але використання конкретних формул не поділяє їх на два різні види. Коли формули пре-рендерингу буде можливо використовувати в реальному часі, що однозначно колись наступить, і за умови інтерактивності, подібна графіка буде вважатись графікою в реальному часі.

2.3. Растрова графіка

Значну кількість часу алгоритми були обмежені можливостями систем, тому використовували простіший метод рендерингу, метод растеризації. На відміну від трасування шляху, що використовується в традиційній комп'ютерній графіці, але

який недавно став можливим і в графіці реального часу, растеризація є дуже швидким процесом. Сама растеризація – це спосіб перетворення тривимірних об’єктів в растрове зображення, тобто в двовимірне зображення, що складається з пікселів, що далі буде показано на моніторі користувача [31]. Растровий метод використовує буфер глибини. Буфер глибини – це просто тип даних, що є інформацією про глибину об’єктів, тобто їх значення Z, з певної перспективи [31]. Його було створено для спрощення обрахунків кожного пікселя зображення, таким чином, якщо один піксель попадає на один непрозорий предмет, то програма не буде дивитися і рахувати, що знаходиться за цим предметом з метою економії ресурсів. Сам буфер є двовимірним масивом, як і зображення, але на відміну від зображення, для кожного пікселя він зберігає не колір, а глибину. Без збереження цих даних і, відповідно, без Z-буфера, зрозуміти розташування об’єктів, та, відповідно, знати, який об’єкт за яким іде не є можливим в растровій графіці.

Загальний процес растеризації полягає в проходженні по кожному об’єкту і визначенню того, який об’єкт є ближчим до якого пікселя, використовуючи Z-буфер [31].

На кожен піксель зображення повинен бути обрахований його колір і в растровій графіці використовується примітивний метод обрахунку. Поверхня вважається освітленою за умови, що на неї попадає світло і вважається темною, якщо ні.

Сама растрова графіка має справу в першу чергу з екранним простором. Екранний простір – це все те, що в конкретний момент часу видно на зображенні. Відповідно згідно з цими принципами і за умови використання растрового методу генерації зображення, все те, що не попадає прямо в умовну камеру, наприклад, все те, що є за нею, не обраховується.

Розуміючи принципи роботи, можна легко вивести і недоліки цього методу. Порівнюючи з пре-рендером, якість зображення значно нижча, функціонал растрової графіки також є примітивним. Було створено велику кількість різних способів для маскування цих проблем і недоліків, але самі недоліки залишаються.

2.4. Трасування променів в реальному часі

З інноваціями в сфері графічних процесорів і відеокарт тепер можливо використовувати трасування променів в реальному часі. На відміну від традиційної графіки в реальному часі, де використовують спрощені методи обчислення, трасування променів використовує спрощені алгоритми традиційної графіки [30]. В пре-рендерингу можливо використовувати велику кількість променів на кожен піксель і величезну кількість відбивань променів від поверхні, але в трасуванні променів в реальному часі існує один промінь на піксель, а сам промінь відбивається від поверхні один раз. Незважаючи на всі ці спрощення обрахунків, трасування променів є неймовірно складним і потребує одних із найкращих доступних персональних комп'ютерів [38].

Подібний метод роботи також забирає необхідність у використанні буфера глибини [31]. Подібно до традиційної комп'ютерної графіки, з кожного пікселя випускається промінь, необхідності окремо зберігати розташування об'єктів з таким методом немає.

На відміну від растеризації, де алгоритм проходить по кожному об'єкту і шукає який з них є ближчий до якого пікселя, в трасуванні променів відбувається проходження по кожному пікселю і визначення об'єкту, ближчого до кожного пікселя [31]. Тобто випускання променів з кожного пікселя і визначення того, який об'єкт є ближчим до променю.

З додаткових методів оптимізації, трасування променів має обмеження по простору своєї дії [34]. Тобто, в тривимірному просторі існують сфери, в яких є ще менші сфери. Кожна з цих сфер обмежує максимальну дистанцію, яку буде проходити промінь. Це створено для того, щоб промені не виходили за розумні межі дії і не витрачали надто великих ресурсів на непотрібні обрахунки. Варто додати, що взаємодія з віртуальною імітацією неба, як глобального джерела освітлення в сцені, яке, з точки зору логіки, має знаходиться значно за межами дії променів, відбувається в відмінний від реальності спосіб. Реалізація неба і його освітлення відбувається за рахунок рівномірного покриття поверхонь світлом. Воно падає під певним кутом і з певною потужністю в люменах. Люмен(lm), що на відміну від кандела(cd) чи люкса(lx), які є двома іншими одиницями

вимірювання освітлення, має рівномірну силу, тобто потужність люменів буде однакова, незалежно від площі поверхні чи відстані до джерела. Навіть більше, в реалізації віртуальних джерел освітлення, таких як Сонце, немає ніякої відстані до віртуального Сонця, є напрям і потужність в люменах, воно освітлює весь простір, який попадає під його дію з певного кута і не має чіткого розташування. Незважаючи на все, результат є подібний до Сонця в реальності. Є певні спрощеннями в загальних обрахунках, світло від Сонця може відбиватись неймовірну кількість разів, навіть до півмільйона, що, звісно, погано підходить для прорахунків в реальному часі і не відбувається в растровій графіці, а в трасуванні променів є лишень одне відбивання.

З точки зору апаратного забезпечення, подібний метод вимагає відеокарту з підтримкою цієї технології. Сучасні відеокарти мають спеціально виділені на обрахунки цих променів ядра, без наявності яких подібна технологія не є доступна в реальному часі [38]. Самі ядра створені виключно для трасування променів і іншого використання не мають.

2.5. Трасування напрямку

Трасування напрямку чи шляху – це метод проявлення зображень, що має активне використання в традиційній графіці і тепер є доступним у графіці реального часу [39]. Трасування напрямку можна вважати продовженням трасування променів, його більш сучасним і складним методом. В трасуванні напрямку присутні декілька відбивань променів на кожен піксель. В трасуванні променів відбувається одне відбивання на піксель і потім його трасування в напрямку джерел освітлення, а фінальний результат цього всього дає колір пікселя. Трасування напрямку використовує схожий метод, але з більшою кількістю відбивань, від одного до чотирьох. В результаті подібного методу можливо отримати зображення, максимально наближене до реального світу. Одним з дуже очевидних недоліків подібного методу є його неймовірна складність обрахунків, що вимагає найкраще і найдорожче доступне апаратне забезпечення.

Оскільки за принципом роботи трасування напрямку – це краща версія трасування променів, то вона має апаратні вимоги як і трасування променів [38].

Без наявності конкретних частин відеокарти, створених спеціально під цю технологію її використання не є можливим.

Рисунок 2.2 – Порівняння різних методів роботи світла.

2.6. Недоліки комп’ютерної графіки реального часу і їх вирішення

Незважаючи на розвиток технологій, комп’ютерна графіка в реальному часі мала, і досі має, недоліки. Досягнення необхідної швидкодії завдало шкоди точності і якості зображення. Незважаючи на це все, було створено різні методи усунення або маскування проблем, що появились через спрощення алгоритмів обчислення.

2.6.1. Тіні

Одною з більш видимих проблем графіки реального часу є тінь. В растровій графіці рівень освітленості може бути двох видів: тінь або відсутність тіні [30]. Це є результатом того, що світло відбивається буквально нуль разів в растровій графіці реального часу, тобто ні одного відбивання. Кожен піксель зображення окремо визначає свій колір, а поверхня є освітленою, якщо на неї потрапляє світло і вважається темною, якщо ні. Тобто, проводиться лінія між поверхнею і джерелом освітлення, якщо лінія проходить без перешкод, то поверхня є освітленою. Подібний, але трохи відмінний підхід використовувався в ранніх алгоритмах для статичної графіки. Недоліки – це, в першу чергу, сильний

поділ на однаково освітлені і однаково темні області. Подібні тіні називають твердими, вони хоч і дешевші в розрахунках, але не є реалістичними. Тверді тіні нечасто можна побачити в реальності, але їх можуть бачити астронавти в космосі через їх відносно близьку відстань до Сонця. Для усунення подібної проблеми почали використовувати імітацію освітлення від неба [16]. На відміну від реального світу цей метод на все просто накладає певний колір певної яскравості. Зазвичай колір є синім, як і саме небо. Подібний метод звучить доволі просто, бо це рівномірне накладання певного кольору на всі поверхні. Незважаючи на це, візуально він допомагає створювати краще зображення.

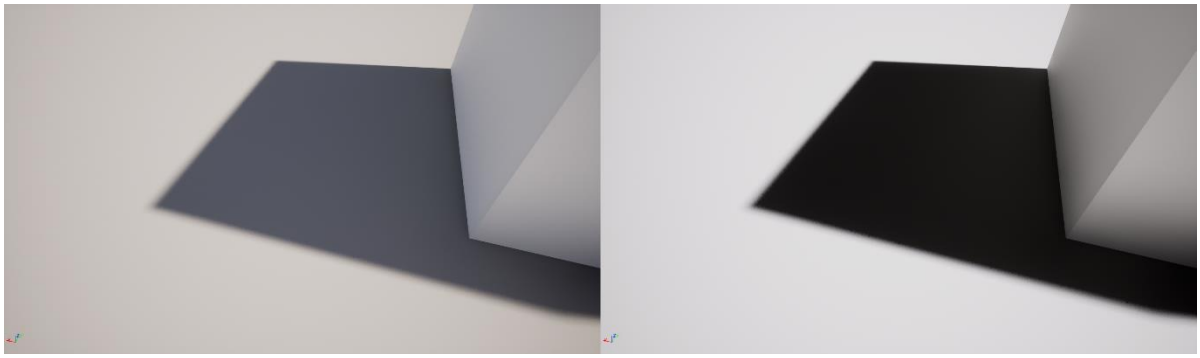


Рисунок 2.3 – Демонстрація накладання рвіномірного кольору на поверхні. Без накладання(справа) і з накладанням(зліва).

Накладання кольору робить затемнену обласить світлішою і маскує певні проблеми, але це не виправляє твердості тіней і не надає їм реалістичного вигляду. Все одно існує чітка межа між тінью і освітленою поверхнею, немає плавного переходу в тінь, вона починається зразу і одним тоном.

Один з методів усунення твердих тіней – це додавання плавного переходу на краї тіней[1]. Замість одного тону темноти краї будуть мати плавніший перехід, розмір і колір якого залежить від конкретних даних джерела освітлення, в першу чергу його яскравості і розмірів, що працює подібно до реальності, але які конкретні формули використовуються залежить від розробника. Варто зауважити, що подібні тіні є дорожчими в обрахунках.

Наступний метод усунення основних проблем з тінями є їх запікання. Запікання – це процес обрахунку тіней від конкретного джерела освітлення, використовуючи формули пре-рендерингу і зберігання результату в текстуру [20].

Оскільки тінь можна виразити як зображення, що накладається на поверхню, її можна вирахувати наперед і просто накладати як окрему текстуру. Процес обрахунків може займати певний час, в залежності від загальної складності, а саме від кількості джерел освітлення. Після цього додаткових обрахунків проводити немає необхідності. Це найдешевший спосіб роботи з тінями в реальному часі, що має потенційно велику якість [10]. З мінусів, процес запікання є доволі довгим, також він збільшує використання відеопам'яті, проте не в значній мірі. Об'єкти, на які падає тінь, повинні бути зроблені з окремим шаром даних для зберігання текстури з тінями. Крім цього, об'єкт повинен бути створений з дотриманням певних правил, що робить запечення тіней на об'єктах, які не є близькими до квадратних, дуже складним. Найбільший недолік зрозумуло з самого принципу роботи цього методу, він не є динамічним. Оскільки тіні для об'єктів вже були прораховані, то самі об'єкти не можуть рухатись в реальному часі, що в свою чергу шкодить інтерактивності програми. Будь-який об'єкт, що рухається, або має змогу рухатись, чи видозмінюватись у будь-який спосіб, не може мати запечену тінь [17]. В сучасному середовищі подібний метод використовується мало, а у деяких випадках вже повністю не використовується, бо є складнішим для розробників і має значні обмеження в інтерактивності.

Наступний спосіб вирішення проблеми є в трасуванні променів. Алгоритм залежить від його конкретних реалізацій, але в основі своїй є близьким до традиційної графіки. Він має вищу якість і також працює в реальному часі без обмежень, що не можна сказати про запечені тіні. З недоліків, подібний процес, навіть обмежений виключно до тіней, має найбільшу складність обрахунків [10]. Використовуючи подібний метод, можливо досягнути високої якості фінального зображення і отримати значно вищої якості тіні. Подібний метод також є більш наближеним до роботи світла в реальності, відповідно, він без сумніву є наступним кроком в розвитку технології і може потенційно досягнути ще кращих результатів з розвитком потужностей. Можливо використовувати і трасування шляху, що є більш точним, але подібним до трасування променів методом з значно вищою якістю і складністю обрахунків. Методи трасування потребують спеціального апаратного забезпечення і є найскладнішими для обрахунків.

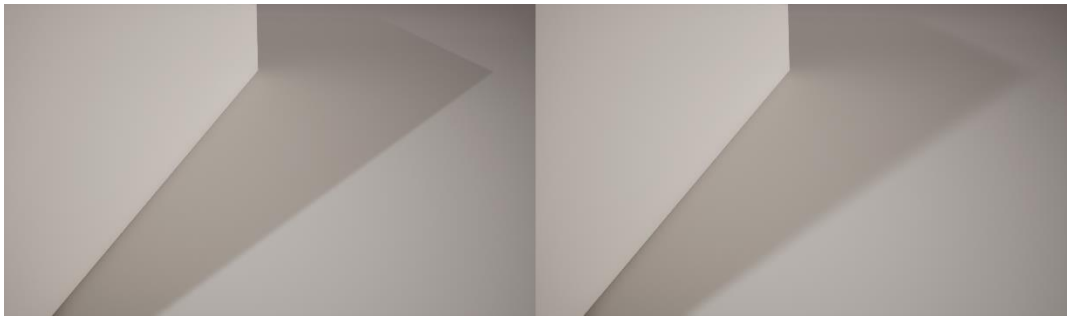


Рисунок 2.4 – Растрова тінь(зліва) і тінь від трасування променів(зправа).

2.6.2. Відображення

Розуміючи обмеження стандартної комп’ютерної графіки в реальному часі, можна легко усвідомити, що, крім тіней, стандартна растрова графіка не може створювати відображення. Стандартний спосіб обрахунків з відсутнім відбиванням світла не може давати ніяких відображень [1]. Відображення – це, в своїй основі, результат відбивання променів світла від поверхні, отже, якщо немає ніякого відбивання світла, то немає і ніяких відображень. Є різні методи маскування і вирішення цієї проблеми.

Перший метод подібний до запікання тіней, але вже з відображеннями. За принципом своєї роботи, з певної точки простору створюються шість зображень, як сторін куба, по одному на кожну сторону, з метою покриття простору певного розміру [13]. Кожне з шести зображень є текстурою певного розміру, всі шість з них потім формують куб чи, доволі часто, сферу, яка використовується для відображень у певній області. Хоча реалізація відрізняється у різних розробників, але принцип роботи залишається однаковим. Фінальний результат, тобто шість зображень, перетворених в куб або сферу і використовують для проекції на об’єкти, поверхня яких дає відображення. З переваг, подібний метод не використовує обрахунків під час роботи програми, хоча, в залежності від заданих розмірів і кількості подібних запікань, він може вимагати великої кількості відеопам’яті [10]. З недоліків, подібний метод, як і в випадку з тінями, не працює з динамічними об’єктами, також за принципом своєї роботи він не здатний на дуже якісний результат. Його відображення майже ніколи не є точними, вони приблизні, крім того об’єкт не може мати відображення себе в собі, що не є реалістичним [13].

Наступний метод вирішує проблему з динамічністю, але вносить немало своїх проблем і недоліків – це відображення, що базуються на екранному просторі [15]. Відображення екранного простору працюють з екранним простором. Використовуючи вже готовий рендер реального часу, він накладається сам на себе, на відповідні поверхні [5]. В такий спосіб він, в значній мірі, може підтримувати ілюзію наявності справжніх відображень, але, якщо подумати над принципами роботи, то можна легко зрозуміти і недоліки. Подібний метод працює з фінальним результатом, а саме з простором перед нами. Він працює за рахунок того, що певні пікселі, що повинні бути відображені, вже є прораховані. Якщо відображення не входить безпосередньо в цей простір, то його видно не буде, в залежності від ситуацій подібне може бути неймовірно помітним. Об’єкти, що не є видимим буквально на екрані не буде видно у відображенні. Принцип роботи – це знаходження розташування і напряму відображення, яке має бути в конкретному пікселі. Z-буфер дає розташування точки відображення в тривимірному просторі, а використовуючи дані нормалей, прораховується напрям відображення [5]. Це все відбувається в межах екранного простору. Далі, використовуючи буфер глибини, будь-які пересікання з напрямом відображення і реальною геометрією передають 2D-координатам конкретного пікселя дані про колір об’єкта, що попадає на відображення. Також, перед накладанням відображення, його можна обробити в певний спосіб, що впливатиме на фінальний вигляд відображення. З точки зору складності обрахунків, подібний метод має доволі високу складність, особливо порівняно з майже повною відсутністю обрахунків у попередньому методі [15].

Наступний метод, метод планарних відображень, є доволі логічним і прямолінійним вирішенням проблеми. З точки зору логіки, відображення можна вважати повторенням вже існуючого. Відповідно, якщо прорахувати всі об’єкти з точки зору певної поверхні, тобто умовно кажучи створити копію світу під певною поверхнею, тоді це буде виглядати як відображення [11]. Так працюють планарні відображення. Планарні відображення накладаються на двовимірну рівну поверхню, що є головним недоліком, крім складності подібних обрахунків. Подібний метод прораховує все відносно певної поверхні, незалежно від того, чи воно попадає у відображення, що може потенційно в десятки разів ускладнити

обрахунки [11]. Недоліки методу очевидні. Прораховування всього двічі для одного відображення не є ефективним методом, а якщо відображень більше ніж одне, тоді прийдеться ще раз повторяти цей складний процес. Можливо вручну обирати, які об’єкти повинні входити в обрахунок, а які не повинні, що може призвести до того, що тільки деякі об’єкти будуть обраховуватись двічі, але сам процес вибору може бути довгим, коли різних об’єктів – сотні і більше. Якщо якийсь елемент не буде додано в список на відображення, тоді його не буде й у відображенні. Якщо прораховувати відображення для всього, то буде відбуватись прорахунок всього двічі, незалежно від того, чи об’єкти попадають в умовне дзеркало, тобто, для прикладу, дзеркало в квартирі буде пробувати прорахувати не просто квартиру, але і ціле місто за нею, що не є ідеальним. Цей метод має своє використання і є першим з наведених, що може давати точні, динамічні відображення.

Останній метод полягає у використанні трасування променів. Цей метод дає найкращий можливий результат, хоча і має найвищу складність обрахунків, але, залежно від складності сцени, може бути простішим, ніж планарні відображення. Основний недолік – це обмежена кількість відбивань променів світла [21]. Через подібні обмеження відображення одних дзеркальних поверхонь в інших дзеркальних поверхнях не буде можливо прорахувати. Неможливість обрахунків веде до появи у відображенні одного предмету контурів іншого, повністю чорного кольору. Виходом з подібної ситуації може бути припинення обрахунків і заміна чорного кольору на будь-який інший, попередньо встановлений задля імітації коректної роботи відображень [22]. Замість трасування променів можна використовувати трасування шляху, що дасть вищу якість зображення, але, відповідно, підвищить складність обрахунків.

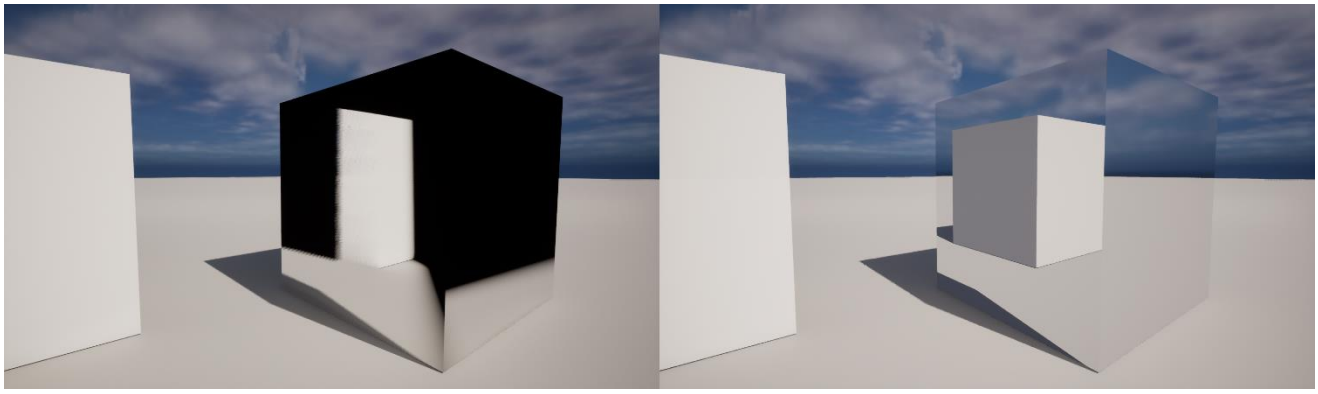


Рисунок 2.5 – Порівняння відображень екранного простору (зліва) і трасування променів (справа)

2.7. Оптимізація комп’ютерної графіки в реальному часі

Комп’ютерна графіка реального часу потребує додаткової оптимізації для своєї роботи. Це можуть бути як і зміни у формулах обрахунків , так і використання спеціальних методів для оптимізації.

2.7.1 MIP-текстурування (MIP mapping)

В комп’ютерній графіці є міпмапінг, або ж MIP-текстурування, де MIP походить від латинського “multum in parvo”, тобто “багато в одному”. Це послідовність оптимізованих і прорахованих зображень, де кожне зображення – це репрезентація попереднього, але меншого розширення в пікселях [18].

Призначення подібного методу в тому, щоб спростити прорахунки і зменшити графічні артефакти, тобто різного виду помилки, що появляються при роботі згладжування. Міпмап, тобто одне зображення з послідовності, високого розширення використовується для областей високої насиченості пікселями, ближче до умовної камери, а міпмап нижчого розміру буде далі від неї. Подібний метод є більш швидким методом прорахунку відображення текстури, ніж прорахунок кожного пікселя текстури і визначення того, які з них будуть впливати на пікселі екрана.

В своїй основі це набір зображень, кожне з яких менше за попереднє. Вони створюються автоматично, на програмному рівні, тобто зображення великих розмірів використовується програмою для створення з його основи менших, поки воно не дійде до мінімального розміру – 2 на 2 пікселя. Сам процес створення хоч і автоматичний, але має певні вимоги до зображення. Для того, щоб рівномірно

зменшити зображення його висота і ширина повинна бути двійкою в степені, 2^n [18]. Тобто 2, 4, 8, 16, 32, 62, 128, 256, 512, 1024, 2048, 4096 і так далі. Відповідно, кожен наступний міпмап є двійкою в меншій на один степені за попередній по ширині і висоті. Саме зображення не обов'язково має бути квадратним, розмір 1024 на 512 підходить для створення міпмапів, але більшість будуть квадратними через зручність їх використання. Математично, якщо використовувати текстуру, яка підходить під вимоги, тобто з розміром двійкою в степені, то кожна наступна текстура буде в 4 рази менша за попередню.

Оскільки міпмапи за своїм визначенням є набором зображень, то вони потребують додаткової відеопам'яті для свого використання. В такий спосіб кожна текстура потребує на 33% більше відеопам'яті, проте приріст в швидкості обрахунків і позбавлення артефактів згладжування однозначно вартує використання міпмапів. Залежно від реалізації технології, одночасно в пам'яті можуть бути завантажені як і всі можливі міпмапи, так і всі версії текстури вищої якості і одна нижчої, за умови наявності такої. Якщо є текстура розміру 4096 на 4096 і зараз вона на екрані перед нами саме в своєму максимальному розширенні, то в пам'яті буде текстура розміру 4096 і на один порядок нижчого, тобто 2048 на 2048, якщо завантажена текстура розміру 2 на 2 пікселя, то одночасно з нею в пам'яті будуть всі текстури вищого розширення.

Ця технологія має активне використання в сфері комп’ютерної графіки і має гарантоване використання в комп’ютерній графіці реального часу.

Рисунок 2.6 – Приклад MIP-текстурування.

2.7.2. LOD

LOD (Level of Detail) в перекладі з англійської означає рівень деталізації. LOD – це спосіб представлення складності тривимірної моделі в комп’ютерній графіці з можливістю зменшення її рівня складності залежно від відстані до умовної камери [1].

За принципом роботи це заміна одної моделі на простішу, коли вона відходить на певну відстань від камери і коли подібна заміна не буде помітна. Моделі нижчої якості, тобто меншої кількості трикутників, створюються разом з самими моделями в спеціальних програмах, там вони і збираються в LOD. Залежно від відстані, а в більш сучасних реалізаціях залежно від відсотку пікселів екрану, які займає модель, її можна замінити на простішу. З точки зору коду це масив від 0 до кількості LOD моделей, зазвичай до 3 чи 4. Сам процес подібний до міпмапів в текстурах, але не є настільки автоматичним.

LOD дані завантажені в пам’ять системи і за необхідністю виводяться на екран. Залежно від конкретної імплементації вони всі можуть бути в пам’яті одночасно, а в деяких імплементаціях тільки необхідні в даний момент моделі будуть в пам’яті [1].

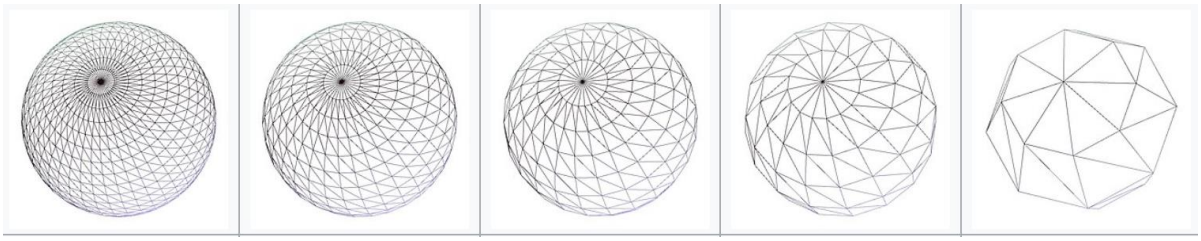


Рисунок 2.7 - Приклад LOD.

3. Взаємодія апаратного і програмного забезпечення комп’ютера в графіці реального часу

Комп’ютерна графіка в реальному часі на базовому рівні є кодом. Тобто, для створення комп’ютерної графіки необхідна програма, що генерує зображення.

В своїй основі програма зчитує дані з диска і виконує з ними певні операції, що в кінцевому результаті призведуть до створення зображення. В випадку з інтерактивною тривимірною комп’ютерною графікою в реальному часі зрозуміло, що програма працює з даними, необхідними для роботи тривимірної графіки, а також має можливість отримувати вхідні дані від користувача і реагувати на них. Хоча подібна програма і використовує велику кількість компонентів і систем комп’ютера, в своїй основі це зчитування даних з пам’яті. Програма не створює дані, всі необхідні дані в неї вже були занесені.

Через код виконуються виклики для надання ресурсів, налаштування, остаточного виведення зображення через API. Сам API стежить за станом програми, перевіряє наявність помилок в цих викликах і потім передає їх графічному драйверу, що буде працювати в режимі роботи користувача.

3.1. User-Mode драйвера

User-Mode драйвера – це частина набору драйверів системи Windows, що працює разом з програмою [25]. Цей набір драйверів створений для спрощення роботи розробників. Він надає більшої стабільності і вищий захист, бо має в собі обмеження доступу до даних користувача, що створено для запобігання використання драйверів з метою викрадення даних. Для оптимізації цей драйвер завжди працює на окремому потоці комп’ютера, що полегшує процес перенесення навантаження програми на різні ядра. Оскільки сам режим був створений для полегшення роботи розробника, то, відповідно, він має значно простіше середовище роботи, ніж звичайні драйвера, що спрощує його використання.

3.2. D3D

Вищевказаний драйвер імплементує API нижчого рівня, що називається D3D. D3D, або Direct3D - це графічний API для платформи Microsoft Windows, що є, в свою чергу, частиною DirectX (або Microsoft DirectX), значно більшої

бібліотеки API, створених компанією Microsoft для своїй систем, що працюють в першу чергу з комп'ютерною графікою [25]. Сам Direct3D є найбільш поширеним компонентом бібліотеки, настільки поширеним, що доволі часто DirectX путають з Direct3D. D3D містить в собі значну кількість команд для роботи з 3D графікою і апаратними компонентами. З його допомогою надається доступ до сучасних технік рендерингу зображень і зв'язок з апаратним забезпеченням. D3D часто використовується у сфері комп'ютерної графіки, наприклад, в програмному забезпеченні різних напрямків з застосуванням комп'ютерної графіки на платформах Microsoft.

Доступ до коду відеокарти і графічного процесора не надається. В цілях захисту інтелектуальної власності його зберігають в секреті. Для роботи з графічним процесором необхідний графічний API, як вище вказаний Direct3D, хоча існують і інші: Vulkan, OpenGL, Metal. Сам API в своїй основі - це набір C++ специфікацій, тобто програмний код, що описує їх функції, назви і параметри [25]. Взаємодія між API і графічним процесором відбувається, бо виробник відеокарт імплементував його підтримку, зазвичай у вигляді драйверів. Отже, якщо подібної імплементації немає, ніякої роботи з графічним процесором не буде. З наведених прикладів, Direct3D працює виключно на платформах Microsoft, таких як Windows і Xbox. Metal компанії Apple працює з їх виробами, а саме IOS і macOS, а OpenGL та його новіша версія Vulkan, мають повністю відкритий код і підтримку вищої кількості систем, але загалом нижчу швидкодію, ніж API створені під конкретні системи.

Отримавши API, який підтримується графічним процесором і системою, на якій ведеться розробка, а також системою, на якій планується запуск програми, можна приступати до роботи. Як вже зрозуміло, API певної системи доступний тільки на ній, відповідно, як мінімум процес фінальної компіляції програми під конкретну систему повинен відбуватись на тій самій системі. Для спрощення процесу компіляції під конкретну систему, за умови що стоїть ціль створення програми і її підтримки на різних платформах, використовується абстракція певної частини коду. Писати графічний код в одному API під конкретну систему і переписувати все це все під інший API – неймовірна трата часу і додатковий ризик

допущення помилок. Замість переписування коду, його частина переходить в більш абстрактний вид, створюється основа графічного коду, його принципи роботи, але окремо від конкретних систем. Наступний крок – це створення системи, яка буде переводити цей абстрактний код під конкретну платформу. Сам процес, безумовно, складний і потребує немалих знань, проте є корисним, якщо є ціль підтримувати більш ніж одну платформу і є однозначно необхідним при більш ніж двох платформах.

D3D також має справу з застарілими версіями драйверів і забезпечує їх підтримку, якщо це можливо [1]. Він також завантажує і розвантажує дані з відеопам’яті.

3.3. Шейдери

Через D3D відбувається компіляція шейдерів. Вони створюються з коду на HLSL і зазвичай мають певні оптимізації [25]. Апаратне забезпечення може без проблем працювати з D3D. Також сам компілятор має в собі певні оптимізації, однак оптимізація шейдерів – це завдання розробника конкретної програми [1].

Шейдер – це програма, яку можна вважати одним із ступенів процесу графіки, що використовується в тривимірній графіці для визначення остаточних параметрів об'єкта або зображення [3].

Теоретично, виробник відеокарт, компонента на якому відбувається значна робота комп’ютерної графіки, може вручну переписати чужий шейдерний код, якщо Ви і Ваша програма достатньо відома, та відправити його користувачам у вигляді драйверів. User-Mode драйвер це сам помітить і автоматично замінить код при роботі програми [23].

3.4. Графічний процесор

Основна робота 3D графіки відбувається на графічному процесорі. Сам графічний процесор – це частина значно більшого компонента комп’ютера, а саме відеокарти.

Як можна зрозуміти, графічні процесори було створено для роботи з комп’ютерною графікою. Вони існують для перенесення складного процесу обрахунків графіки на спеціально створений для цього компонент системи. Раніше

для цього використовували центральний процесор, але його швидкодії було недостатньо, що заставило створити компонент спеціально для роботи з графікою. Простіше кажучи, графічний процес має вищу швидкодію при роботі з графікою.

Різниця між графічними і центральними процесорами полягає в тому, що графічні процесори мають швидший обрахунок векторної математики, а також чисел з рухомою комою, тобто математичної основи самої графіки. В загальному, вони уступають центральним процесорам, але можуть значно швидше виконувати паралельні процеси. Коли операція успішно поділена на десятки мільйонів і більше малих операцій, обрахунки будуть швидше виконуватись на графічному процесорі, ніж на центральному. Навіть вивід зображення, залежно від розширення монітору може містити приблизно 2 мільйони пікселів, для кожного з яких необхідно визначити колір. Для подібних завдань використовують графічні процесори.

Графічні процесори зазвичай використовуються для створення зображень і для відображення їх на певному дисплеї. Вони можуть бути різної складності і потужності, все залежить від очікуваного способу застосування процесору. В персональних комп’ютерах графічні процесори є частиною відеокарти, яка є окремим компонентом комп’ютера і містить в собі необхідні для роботи з зображеннями компоненти. Незважаючи на це, графічні процесори різних видів можуть використовуватись і за межами персональних комп’ютерів. Зазвичай більшість видів техніки, що мають справу з виведенням зображень, як от дисплей холодильника, кавоварки чи телефона, містять в собі графічний процесор певного виду. Певні системи, скажімо, телефони, хоч і мають більш обмежене апаратне забезпечення, в основі своїй функціонують подібно до персональних комп’ютерів. На відміну від комп’ютера, обмеженості в потужності будуть вищі, а також API буде створений під конкретну систему, наприклад Android чи IOS. Крім цих нюансів, робота залишається подібною, міняється API і міняється середовище розробки, але не її принципи.

Варто сказати, що графічний процесор, незважаючи на це все, є доволі обмеженим і роботу за межами своєї спеціалізації зазвичай не виконує. Замінити ним центральний процесор – ідея не надто можлива і не надто корисна. Набір

команд графічного процесора є дуже обмеженим порівняно з центральним, оскільки він був створений для роботи конкретно з комп'ютерною графікою. Ця обмеженість команд не дозволяє використовувати його як заміну вже існуючим центральним процесорам.

В більш примітивному вигляді, але все ж можна використовувати центральний процесор для генерації зображень комп'ютерної графіки в реальному часі. Сучасні, потужні центральні процесори здатні на подібне, але вони будуть слабші за їх графічні версії і більш обмежені в загальній швидкості. Вони виконуватимуть свою роботу значно повільніше. Крім цього, графічні процесори мають апаратну підтримку певних технік, відсутніх в центральних процесорах, що може сповільнити їх роботу. Проте відсутність певних апаратних компонентів може, як в випадку з трасуванням променів в реальному часі, не дозволити використовувати певні алгоритми. Також центральні процесори використовують оперативну пам'ять системи, яка в середньому в 5-10 повільніша за відеопам'ять і у кращому випадку досягає 80 гігабайт в секунду.

3.4.1. Система пам'яті графічного процесора

Завантаження і розвантаження даних з пам'яті графічного процесора потребує окремих пояснень. Вона відрізняється від стандартної пам'яті за своїм призначенням і принципами роботи.

В першу чергу, пам'ять графічного процесора швидка. Вона може досягати 500-1000 гігабайт в секунду [2]. Така неймовірна швидкість необхідна для роботі з графікою реального часу, бо основна частина даних, що необхідні для обрахунку кожного кадру, надходять з відеокарти. Коли потрібно прораховувати один кадр кожні 10 чи 30 мілісекунд, подібна швидкість стає необхідною. Графіка реального часу працює за рахунок того, що тримає в пам'яті відеокарти всі об'єкти, що можуть бути на екрані, а також всі пов'язані з ними дані. Внаслідок цього важливо ретельно стежити за використанням пам'яті і не допускати перевищення меж використання доступної пам'яті. Коли відеопам'яті перестає вистачати, система починає зберігати дані в оперативній пам'яті, а якщо її не вистачає, то на накопичувачах. У найгіршому випадку, коли і цього не вистачає, варто очікувати

припинення процесу, тобто закривання програми. Варто зазначити, що перехід з відеопам’яті до оперативної пам’яті вже є значним ударом по швидкодії і буде призводити до зависання приблизно на пів секунди чи більше, залежно від рівня нестачі пам’яті. На противагу цьому, додатковий обсяг відеопам’яті, як і будь-якої іншої пам’яті, за умови, що вона не використовується, не призведе ні до якого приросту в швидкодії і не прискорить обрахунків.

Пам’ять, з якою працює графічний процесор, розташована в відеокарті і по суті є її частиною. Можливості її замінити у випадку несправності чи за бажанням немає. Причин на це декілька, наприклад те, що відеокарта – це середовище пов’язаних між собою компонентів, заміна одного з них на інший надто складна для некваліфікованих людей. Також відеокарта була створена з конкретними компонентами і заміна їх може привести до несправності. Проте найбільша і основна причина – це ціна. Створення пам’яті, яку можливо заміняти є просто надто дорогим процесом і у деяких випадках навіть неможливим. Відеокарти потребують величезної швидкості роботи пам’яті, значно вищої, ніж в самій системі комп’ютера, що є основною причиною швидкодії графічних процесорів. Створення змінної пам’яті потребуватиме складнішої роботи для забезпечення надійного зв’язку між нею і рештою відеокарти, що є потенційно як мінімум в три рази дорожчим процесом.

3.5. Оперативна пам’ять

В роботі програми, що працює з комп’ютерною графікою, як і в роботі будь-якої програми, буде використовуватись оперативна пам’ять. На відміну від традиційного виконання комп’ютерної графіки, коли результат необхідно отримувати в реальному часі, оперативна пам’ять є надто повільною для зберігання геометрії чи інших графічних даних. Проте для речей, що не використовуються в обрахунках графіки оперативна пам’ять знаходить своє використання, так само як і в інших програмах.

В оперативній пам’яті можуть зберігатись дані, що необхідні для різних неграфічних обрахунків, для прикладу це можуть бути обрахунки фізики. Процес передачі даних між графічним процесором і центральним процесором є надто

довгим для регулярного використання, відповідно деякі дані, що можуть знадобитись для неграфічних обрахунків, будуть зберігатись в оперативній пам’яті.

Один з видів потенційно об’ємних даних, які зберігаються в оперативній пам’яті – це аудіо дані. За умови наявності десятків і більше різних аудіофайлів, що можуть почати своє відтворення у будь-який момент часу та за умови відсутності компресії аудіофайлів, що в свою чергу значно збільшує їх розмір, можливе значне апаратне навантаження програми і збільшити необхідну кількість оперативної пам’яті.

3.6. Диск

Програма, що працює з комп’ютерною графікою реального часу знаходиться на диску комп’ютера і взаємодіє з диском. Програма для створення комп’ютерної графіки – це просто програма, яка на основі існуючих на диску даних створює зображення. З подібної точки зору можна вважати, що вона не відрізняється від інших програм, вони також по суті працюють з даними на диску.

Незважаючи на цю схожість, інтерактивна тривимірна комп’ютерна графіка в реальному часі має цікавішу взаємодію з диском системи. Хоча в своїй основі диски зберігають дані, зчитують їх і записують, але програма має певні особливості, що існують для покращення її роботи. Комп’ютерна графіка реального часу повинна мати оптимізацію під певну швидкість зчитування даних з диску.

Припустимо, що існує певна тривимірна сцена в програмі. Ця сцена може бути призначена для створення графічної частини фільму, або для проведення якоїсь певної симуляції. В ході роботи програми нам дуже часто потрібно розвантажувати вже непотрібні дані і завантажувати нові. Для користувача це може виглядати як зміна локації, наприклад перехід з сонячного пляжу на засніжені гори, ми покидаємо попередню сцену і повинні завантажити нову з новими даними. Ці дані можуть бути 3D моделями, шейдерами, текстурами, різного виду кодом, аудіо файлами. Це необхідно щоб звільнити відео і оперативну

пам’ять. Операція може здаватись простою, але подібну роботу з комп’ютерною графікою реального часу і з програмою можна виконати трьома різними методами.

Перший і найбільш зрозумілий метод – це завантаження всієї програми зразу при запуску в пам’ять комп’ютера, тобто оперативну пам’ять і відеопам’ять [23]. Подібний метод є буквально найпростішим, проте обмеженим загальним розміром програми. Завантажити 1 гігабайт в оперативну пам’ять і 500мб у відеопам’ять не є надто складною задачею. Проте, якщо загальний розмір програми починає переходити за десятки і сотні гігабайт, то починаються очевидні проблеми з таким методом. Наразі він майже не використовується, його можна застосовувати хіба що у найпростіших програмах.

Другий метод полягає в тому, що робиться пауза в роботі програми, щоб дати їй час завантажити дані [23]. Чудовий метод, простий і зрозумілий, а головне здатний працювати з програмами великих розмірів. Подібний процес можна ще і прикрасити красивим екраном завантаження, для надання користувачеві кращих вражень від використання програми і потенційно переводячи його увагу з самого факту завантаження. Тривалість подібного процесу залежить від кількості даних, що необхідно завантажити і від швидкості накопичувача. Подібний метод є наразі найпопулярнішим.

Третій, найновіший і найскладніший метод полягає у динамічному завантаженні і розвантаженні даних з системи [23]. Подібний метод створено з метою усунення зупинок в програмі для завантаження даних, оскільки вони можуть негативно впливати на враження користувача. В теорії тут нічого надто складного, завантаження даних і розвантаження даних працює як і в другому методі, але без паузи. Складність в тому, щоб уникнути самої паузи і вчасно все завантажити. Для прикладу необхідно завантажити 10 гігабайт і розвантажити 10 гігабайт даних. Максимальна швидкість жорсткого диску приблизно 160мб/с, приблизна максимальна швидкість ССД-накопичувача – 500мб/с і у випадку з M.2 ССД-накопичувачем ця швидкість може сягати 8гб/с і вище. Тобто у випадку звичайного жорсткого диску це займе більше хвилини часу, а саме 62 з половиною секунди, 20 секунди потрібно буде з використанням ССД-накопичувача і трохи більше однієї секунди на M.2 ССД-накопичувачі. Якщо необхідно завантажувати

такий великий обсяг даних, то є сенс відкинути підтримку жорстких дисків і робити програму з рекомендацією використання ССД дисків, бо у подібному прикладі досягнути коректної роботи з настільки повільним апаратним забезпеченням є неможливим. Важлива задача – зробити процес завантаження непомітним для користувача. Для прикладу, якщо користувач керує віртуальним автомобілем, використовується комп’ютерна графіка реального часу з метою симуляції. Користувач і його цифровий автомобіль зараз знаходиться в віртуальному місті, але він їде з міста в сторону пляжу. З точки зору програми, автомобіль і різні його дані як і саме місто зараз знаходяться в пам’яті, пляж до якого ще потрібно добратись не є в пам’яті системи. В такому випадку, використання третього методу буде через поступове розвантаження з пам’яті найбільш далеких ділянок міста, які користувач не бачить і бачити не може і поступове завантаження наступних даних по напрямку переміщення користувача, тобто в даному прикладі – пляжу. Коли користувач покине місто і буде на пляжі, ніяких даних міста в пам’яті його системи давно вже не буде завантажено, за умови, що шлях займає більший час ніж завантаження даних. Наведений приклад є одним з найпростіших, у випадку з великою віртуальною дистанцією і достатнім часом на зміну локацій є можливість завантажити і розвантажити необхідні дані, але на практиці подібний метод потребує більш точного розрахунку шляху і даних загрузки. В сучасності цей метод має активне використання і його популярність внаслідок росту потужності апаратних можливостей комп’ютерів тільки росте.

4. Реалізація програмного забезпечення комп’ютерної графіки реального часу

Основною метою дипломної роботи було створення власного програмного забезпечення, яке працює з комп’ютерною графікою в реальному часі. Ця програма є практичною реалізацією описаної в дипломній роботі теорії комп’ютерної графіки.

4.1 Призначення програмного забезпечення

Створення програмного забезпечення подібної складності не тільки демонструє засвоєння матеріалу і проведене дослідження галузі комп’ютерної графіки, але й може використовуватись для демонстрації описаних в роботі алгоритмів. Програма подібного типу може мати і навчальне застосування. Пояснення складного матеріалу в такий спосіб надає можливість іншим людям поглибити свої знання. Людині, особливо молодого віку, буде цікавіше вивчати подібні речі у вигляді інтерактивної програми, що робить засвоєння інформації динамічнішим і більш наглядним, ніж при використанні підручників.

Ціль програми – це перетворення складної інформації в цікавішу для сприйняття більшої кількості людей форму. Хоча зміст програми відповідає суті описаного в тексті дипломної роботи, але, на відміну від дипломної роботи, більше заохочує свого користувача. Додавання інтерактивності в програму дозволяє краще зберігати фокус користувача на різних формулах і алгоритмах, що є доволі нетривіальною задачею у випадку з простим текстом на папері.

Програма дає користувачам можливість особисто взаємодіяти з елементами роботи комп’ютерної графіки на наочних прикладах. Мотивація людини є найважливішим фактором в процесі навчання, відповідно, утрата інтересу до пізнавального виду діяльності веде до значного зниження ефективності процесу навчання та засвоєння нової інформації загалом. Ця програма створена з метою усунення цієї проблеми, вона призначена мотивувати користувача вивчати матеріал в процесі інтерактивної взаємодії та засвоєння матеріалу з конкретною демонстрацією описаних елементів комп’ютерної графіки з подальшою перевіркою рівня засвоєння поданого матеріалу. Подібний метод, незважаючи на

свою технічну складність виконання має значно вищий рівень засвоєння матеріалу.

Подана в програмі інформація відповідає суті дипломної роботи, проте в більш короткій, лаконічній формі. Вона містить в собі опис формул і алгоритмів з дипломної роботі, але відтворений у програмному середовищі.

З метою максимальної доступності було вирішено використовувати спрощені формули і технології. Використання найновіших технологій значно б збільшило вимоги до потужності комп'ютерів, які могли б її запустити. Таким чином, більша кількість людей зможе використовувати програму, а отже більша кількість людей може дізнатись щось нове про комп'ютерну графіку.

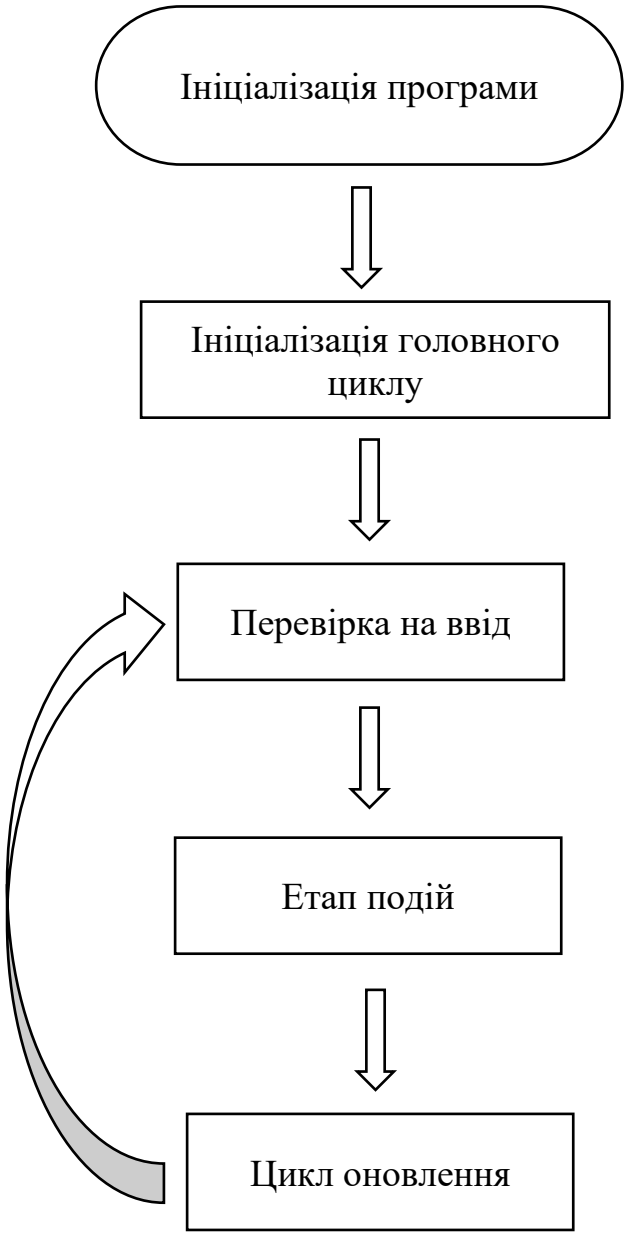
4.2. Основа програмного забезпечення

Програмне забезпечення функціонує подібно до більшості інших програм, але працює з комп'ютерною графікою. Запуск програми відбувається в стандартний для програм спосіб – через файл формату .exe в папці програми. Сама папка містить в собі всі дані, що необхідні для роботи програми.

4.3. Принципи роботи програмного забезпечення

Використовується для основи програми рушій Unreal Engine 5, що є фреймворком і редактором для створення програм подібного типу. При запуску програми, рушій ініціалізує необхідні для запуску компоненти. Це завантаження різних модулів, що необхідні для функціоналу програми, завантаження коду, що дозволяє програмі працювати на конкретній платформі, також це завантаження параметрів налаштування користувача [8]. При ініціалізації створюються окремі потоки на програму і на її шейдери. Після етапу ініціалізації створюється основний цикл програми, що буде виконуватись, поки програма не буде закрита [8]. Основний цикл виконується один раз перед створенням кожного кадру. Весь код програми виконується в межах цього циклу. В основному циклі виконується базовий код, що необхідний для роботи програми, проходить перевірка на ввід користувача і виконується етап подій. Подія (Event) – це виклик конкретних функцій, класів, тощо за певних умов. Наприклад, користувач захотів закрити програму і натиснув відповідну кнопку, це викликає функцію завершення роботи

програми. З метою оптимізації подібний код не виконується для кожного кадру, а тільки за окремою командою, наявність якої перевіряє етап подій. Крім основного циклу, існує цикл update або оновлення. В цикл update входить додатковий код, що також повинен виконуватися кожен кадр [8]. Зловживання циклом update може легко призвести до надмірного навантаження на центральний процесор. Перш ніж завершується графічне створення зображення, центральний процесор повинен виконати весь код в основному циклі. Загальна блок-схема має такий вигляд:



Блок-схема 1 – Основний цикл рушія Unreal Engine.

4.4. Робота програми з апаратним забезпеченням

За взаємодію програми і комп’ютера в значній мірі відповідає операційна система. Код програми тісно взаємодіє з операційною системою, в цьому випадку це Microsoft Windows. Програма виконує виклики для надання ресурсів і остаточного виведення зображення через API системи [25]. Платформа Windows використовує для роботи програми і апаратного забезпечення набір драйверів DirectX.

Для доступу до відеокарти і обчислення комп’ютерної графіки програма використовує API платформи Windows. Далі платформа Windows через свій API починає роботу з драйверами відеокарти [25]. В свою чергу драйвера відеокарти взаємодіють з самою відеокартою і починають виконання графічного коду програми. Блок-схема роботи програми і відеокарти має наступний вигляд:

Блок-схема 2 – Взаємодія програми і відеокарти.

Взаємодія програми з іншим апаратним забезпеченням відбувається в подібний спосіб. Основна робота виконується операційною системою і її драйверами.

Для своєї роботи програма використовує DirectX11 – старішу версію бібліотеки DirectX. DirectX11 було вперше випущено в 2009 році, що дозволяє старішим версіям системи Windows запускати програму. Для роботи з DirectX11 необхідно використовувати як мінімум Windows 7 або новіші версії.

4.5. Функціонал програми

Крім базової основи, для повноцінної роботи програма повинна мати певний додатковий функціонал. Було створено код, що надає програмі різного виду додаткові можливості.

4.5.1. Підтримка додаткових пристроїв введення даних

Для взаємодії з програмою користувач зазвичай використовує стандартні для персональних комп'ютерів пристрої – комп'ютерну мишу і клавіатуру. Хоча подібний функціонал є достатнім для реалізації взаємодії користувача з програмою, задля забезпечення різноманітності способів керування програмою і враховуючи зростаючу популярність альтернативних засобів управління і способів вводу інформації у програмному середовищі комп'ютера було вирішено додати підтримку додаткових пристроїв введення.

Була додана підтримка XInput API, що є частиною архітектури Microsoft Windows [27]. XInput API дозволяє програмі отримувати дані від контролерів, створених з підтримкою XInput API. Найбільш поширеним прикладом подібних контролерів є серія Xbox Controller. Варто додати, що сам API є доступним за межами платформи Windows, його також можливо використовувати на таких платформах, як macOS, Android, Linux та IOS. На Windows 7 і вище XInput API є встановленим за замовчуванням [26]. В такий спосіб користувач може повністю взаємодіяти з програмою без використання клавіатури чи комп'ютерної миші, що служить додатковим фактором, спрямованим на охоплення тої частини цільової аудиторії даного програмного забезпечення, що надає перевагу використанню альтернативних джерел керування комп'ютером.

Рисунок 4.1 – Умовне зображення контролера серії Xbox Controller.

Використовуючи XInput програма може сприймати інформацію, що надходить від контролера [27]. Серед можливої інформації є стан кнопок, тобто чи є якась з них нажатою, стан аналогових стіків. Аналоговий стік – це частина контролера, що через постійний потік інформації надає дані про своє розташування. Працюють вони по двовимірному графіку XY, де максимальне значення є 1 і мінімальне -1, а центр є точкою (0,0) [26]. Інформацію про розташування аналогових стіків в просторі можливо використовувати в різні способи. В даному випадку їх використовують для навігації.

4.5.2. Підтримка різних мов

Створена програма має підтримку двох мов – англійської і української. Була проведена додаткова робота з перекладом тексту, що в свою чергу дозволяє донести інформацію до більшої кількості людей. Користувачі можуть легко змінювати мову під час роботи програми. Цей процес не залежить від сторонніх API і є простою заміною одного тексту на інший, з допомогою вбудованої в рушій системи локалізації, що дозволяє легко знаходити і перекладати необхідний текст.

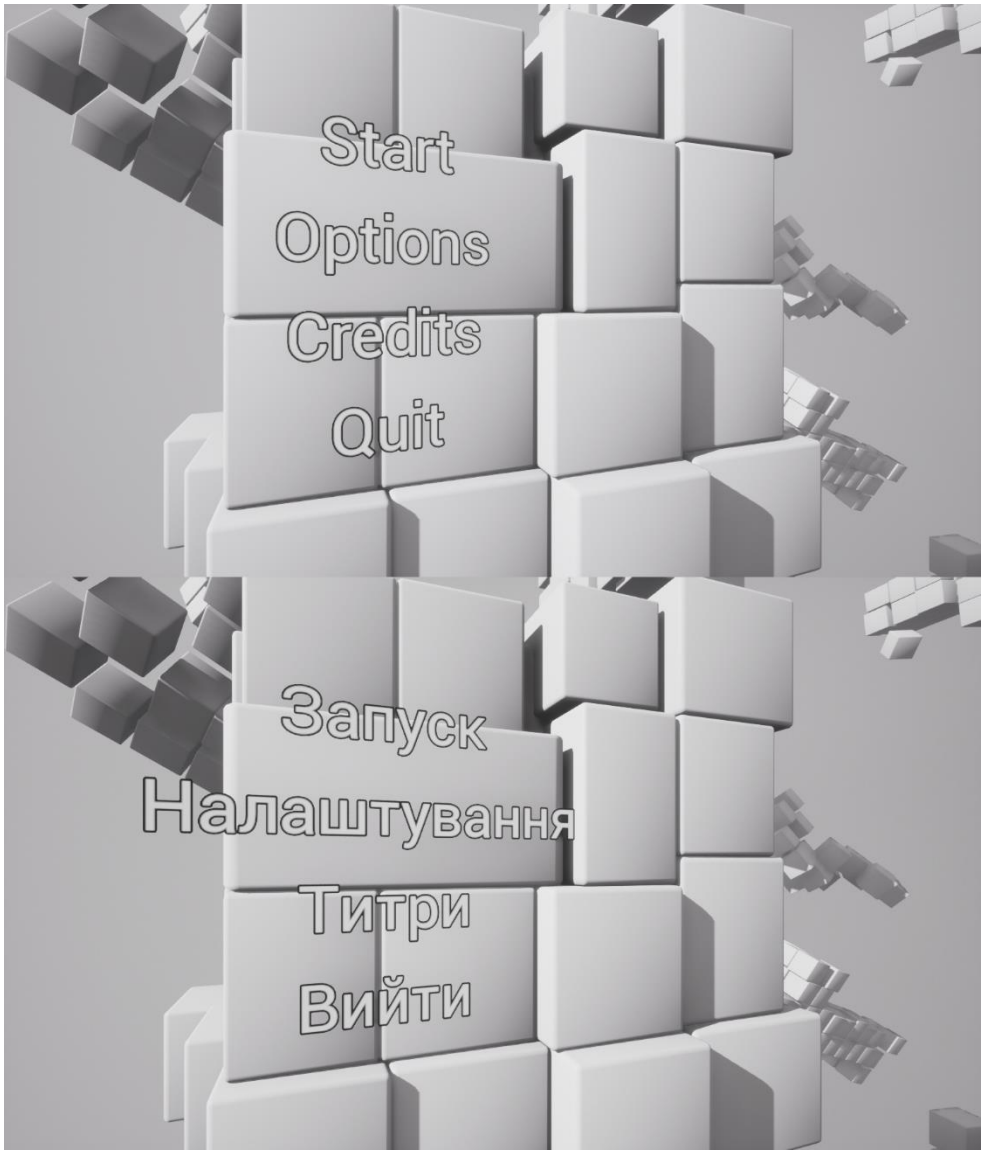


Рисунок 4.2 – Переклад головного меню програми.

4.5.3. Зміна графічних налаштувань програми

Користувач програми може змінювати її графічні налаштування. Графічні налаштування впливають на різні параметри комп’ютерної графіки і дають змогу додатково спростити різні обрахунки, що в свою чергу зменшує навантаження на графічний процесор. Подібні зміни можуть допомогти слабшим персональним

комп’ютерам запускати програму, хоч і з гіршою якістю зображення.

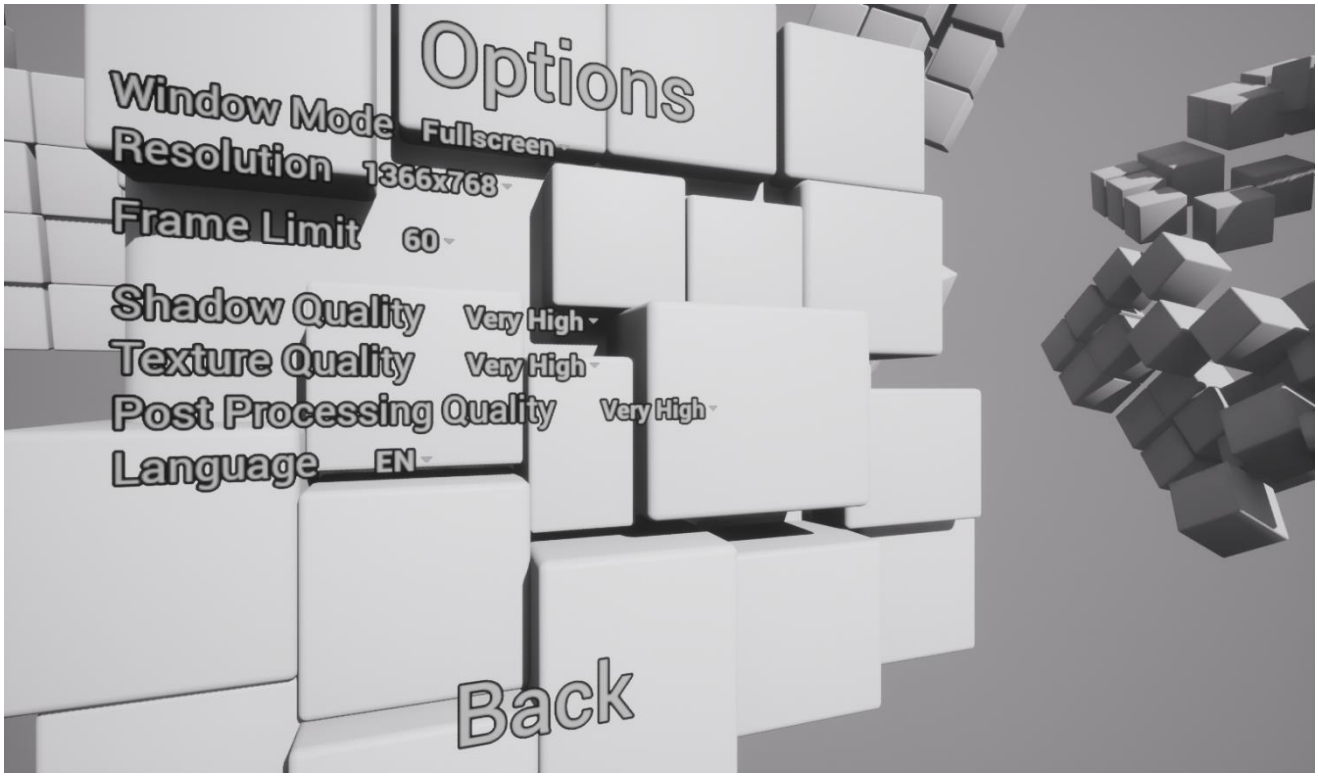


Рисунок 4.3 – Меню налаштувань програми.

Зменшення розширення зображення має значний вплив на швидкість обрахунків, оскільки з більшою кількістю пікселів потрібна більша кількість обрахунків.

Обмеження кадрів в секунду не надає приросту в швидкодії. Воно існує для економії ресурсів комп’ютера. У випадку, коли монітор має частоту в 60Hz, тобто 60 кадрів за секунду, немає необхідності програмі вираховувати зайві кадри, якщо їх не буде видно на екрані [10]. Користувач може виставляти різні обмеження на частоту кадрів, а може і зняти обмеження та генерувати максимально можливу кількість кадрів в секунду.

Якість тіней відповідає за максимальну якість динамічних тіней, тобто тих тіней, які не запікали. Кожна тінь має певні розміри 2^n , як і текстури. Зменшення якості зменшує степінь n на одиницю, тобто 4096 на 4096 стає 2048 на 2048. Залежно від кількості тіней на екрані, зменшення їх якості може мати значний вплив на швидкодію програми і на якість тіней в зображенні [10].

Якість текстур працює подібно до тіней, але взаємодіє з вже створеними міпмапами. Зменшення рівня якості заставить текстури перейти на нижчий рівень

міпмапів за умови, що подібний є. Розмір текстур має невеликий вплив на швидкодію, але звільняє значну кількість відеопам’яті, що може мати значний вплив на роботу програми [10]. Візуально вплив на зображення буде помітним.

Якість постобробки впливає на якість візуальних ефектів, що можуть додатково накладатись на зображення. Це може бути корекція кольорів, затемнення по кутам зображення, хроматична аберація та інші ефекти. Їх параметр якості впливає на складність обрахунків, що впливає на їх точність і фінальний результат [10]. Залежності від кількості ефектів це може мати значний влив на швидкодію.

Наведені в програмі налаштування є дуже базовими. Залежно від складності програми, можливо використовувати додаткові налаштування, що надає більший контроль над якістю зображення і швидкодією програми.

4.5.4. Виведення інформації через взаємодію з об’єктами

Інформація користувачеві подається через окремі об’єкти в тривимірному середовищі. Вони створені для взаємодії з користувачем і реагують на його погляд і натискання певних кнопок.

Для визначення погляду користувача використовуються сфери певного радіусу. Одна така сфера запускається кожен кадр роботи програми в напрямку погляду користувача на певну відстань. Якщо сфера зустрічає на своєму шляху певний об’єкт, то вважається, що користувач на нього дивиться. В програмі використовуються сфери радіусом в 40см і максимальний їх напрям – 5 метрів. Коли сфера зустрічає перший об’єкт, то перестає рухатись далі, що економить ресурси комп’ютера.

Для взаємодії з користувачем використовується спеціально створений для цього код. Якщо об’єкт, який зустріла сфера, має в собі імплементацію цього коду, тоді він виділяється певним кольором. Залежно від розширення виділення може бути два і більше пікселі в ширину. Крім цього, при натисканні конкретної кнопки користувач може взаємодіяти з предметом, що, у більшості випадків, виведе певну інформацію. Об’єкти взаємодії можуть виглядати як таблички або різного виду кнопки, що робить процес взаємодії більш зрозумілим користувачеві.

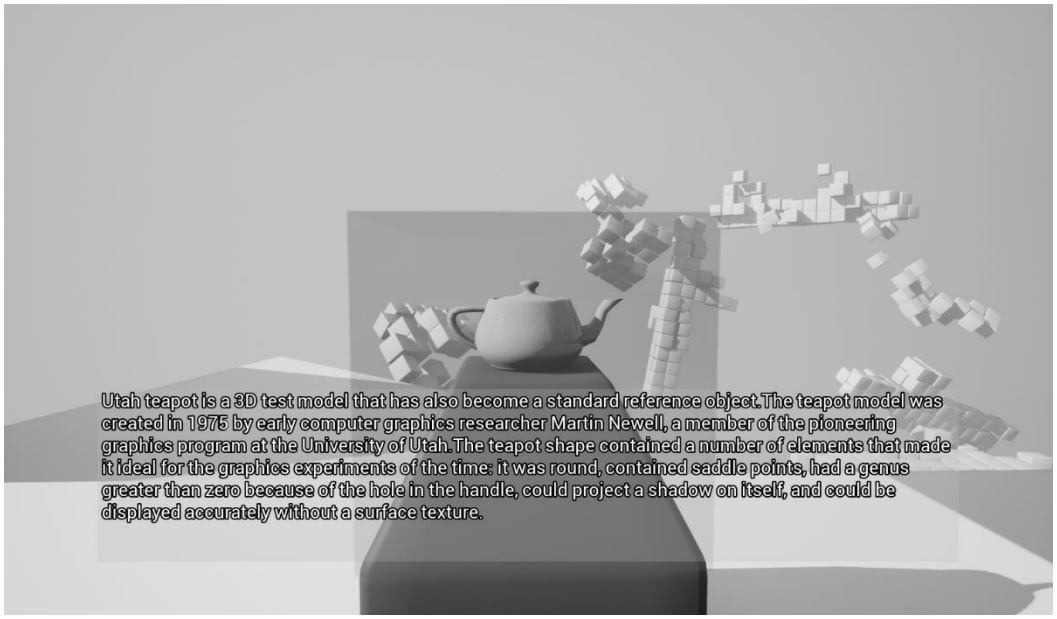
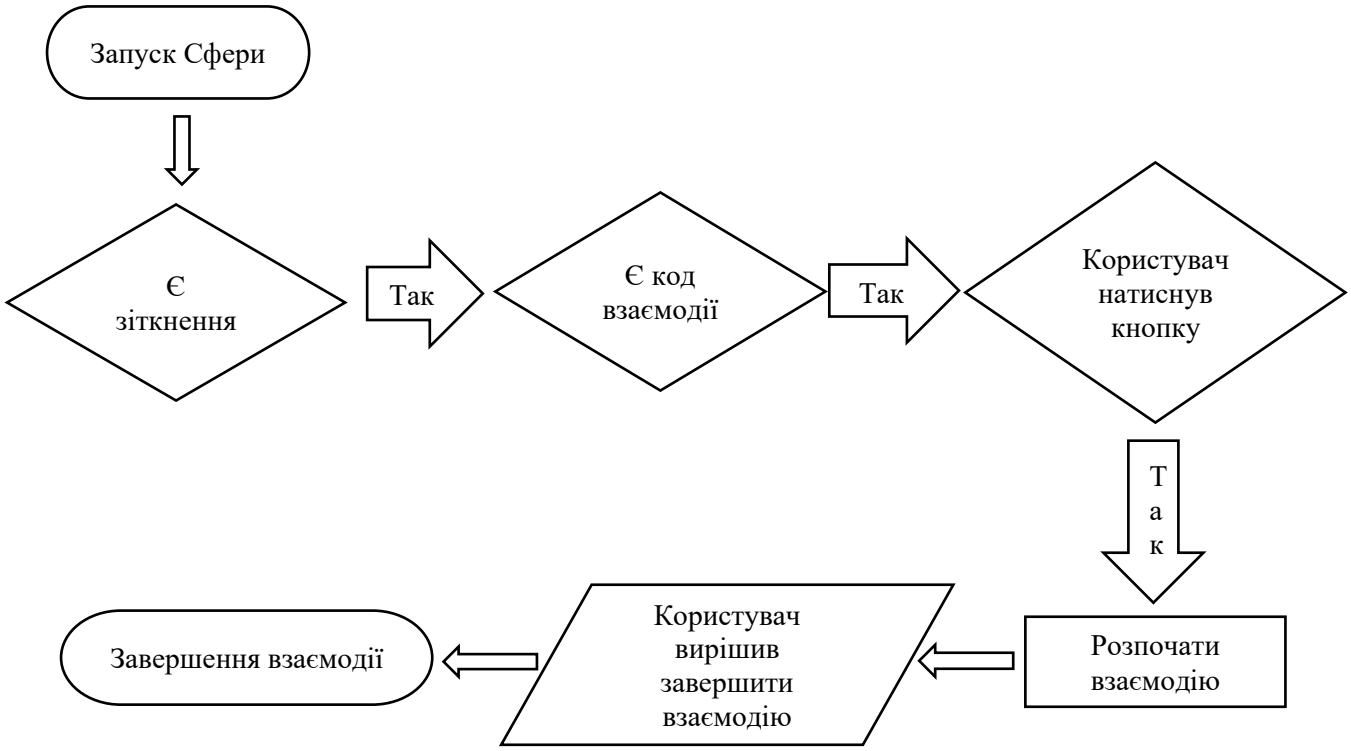


Рисунок 4.4 – Чайник з Юти і інформація про нього в англійському перекладі.

Подібна взаємодія також використовується в тестуванні користувача. В процесі тестування користувач повинен вірно відповісти на серію запитань, що перевірить засвоєний ним матеріал і дозволить перейти до вивчення наступних тем. Блок-схема коду взаємодії має наступний вигляд в циклі оновлення:



Блок-схема 3 – Робота коду взаємодії.

5. Економічна характеристика проектного виробу

Програма подібного рівня повинна мати високий рівень надійності, якості, а також доступну для державних закладів вартість. Для розрахунків буде використовуватись більш довершена і функціональна версія програми, що вимагатиме більшої кількості людей і ресурсів.

5.1 Визначення собівартості і ціни створення програми

Собівартість повноцінної програми є сумою витрат на зарплату команди розробників і на різного виду програмні ліцензії. Розмір команди і кількість необхідних під це програм залежить від запланованого функціоналу. Для спрощення розрахунків вважається, що апаратне забезпечення, наприклад персональні комп'ютери для розробки програми, вже є в команди і закупівлі не потребують.

$$TC = FC + VC \tag{5.1}$$

TC – загальна вартість.

FC – фіксована вартість, тобто вартість незалежна від кількості виготовлених продуктів.

VC – змінна вартість, тобто вартість, що залежить від кількості виготовлених продуктів.

В наведеному прикладі VC буде надто незначною або навіть відсутньою, що дозволяє нам її не враховувати. Загальна вартість буде повністю залежати від фіксованої вартості.

$$FC = DC + SC \tag{5.2}$$

DC – вартість розробки.

SC – вартість програмного забезпечення, що необхідне для розробки.

Для спрощення обрахунків команда розробників не буде поділена на посади і всі будуть мати однакову заробітну плату. Відповідно, формула визначення вартості розробки буде представлена у спрощеному вигляді.

$$DC = DN * DS * DT \tag{5.3}$$

DN – час розробки.

DS – зарплата одного робітника.

DT – кількість робітників.

Вартість необхідного програмного забезпечення може відрізнятись залежно від великої кількості факторів. Деякі з наведених програм можуть мати різні аналоги, для обрахунків будуть використовуватись найбільш поширені програми. Вартість програмного забезпечення – це вартість всіх програм або ліцензій тривалістю в необхідний для завершення розробки час.

$$SC = LC + SPC \tag{5.4}$$

LC – загальна вартість ліцензій на необхідний час.

SPC – загальна вартість програмного забезпечення, що не потребують регулярних виплат.

Конкретні програми та їх вартість буде наведена у таблиці. Загальний час розробки – один рік.

Таблиця 5.1. – Вартість створення повноцінної програми.

Назва витрати	Кількість, шт.	Вартість за одиницю, грн.	Сума, грн.
Зарплата робітникам	10	10,000/місяць	1,200,000
Ліцензія Adobe Photoshop	2	775/місяць	18,600
Ліцензія Autodesk 3dsMax	2	11,250/рік	22,500
Ліцензія Microsoft Windows	4	7,900	31,600
Всього			1,272,700

Як можна зрозуміти, на створення повноцінної програми більшість витрат іде на зарплати команді розробників. Це все без урахування вартості необхідного апаратного забезпечення, без оренди приміщення і з мінімальними витратами.

Створення простішого програмного забезпечення, яке наведене в дипломній роботі, звісно, вимагало менших витрат. Визначення його вартості відбувається без урахування апаратного забезпечення і без додавання вже придбаного програмного забезпечення, а також без необхідності платити зарплату. Час розробки – 6 місяців.

Таблиця 5.2. – Вартість створення дипломної програми.

Назва витрати	Кількість, шт.	Вартість за одиницю, грн.	Сума, грн.
Ліцензія Adobe	1	885/місяць	5,310
Ліцензія Autodesk	1	11,250/рік	5,625
Всього			10,935

Вищевказана таблиця не враховує більшість витрат і є лишень приблизною мінімальною вартістю, за умови повної відсутності зарплати.

5. Охорона праці та безпека

5.1. Аналіз шкідливих дій при виготовленні охоронного пристрої.

В даному проекті було розроблено програмне забезпечення. Створення програмного забезпечення містить мінімальну загрозу для здоров’я людини. Подібне програмне забезпечення не здатне завдати людині шкоди.

Висновки

1. Описано принципи роботи традиційної комп’ютерної графіки і її розвиток.
2. Описано принципи роботи комп’ютерної графіки в реальному часі і її розвиток.
3. Пояснені особливості взаємодії апаратного і програмного забезпечення при роботі з комп’ютерною графікою в реальному часі.
4. Створено програмне забезпечення, що працює з комп’ютерною графікою реального часу.
5. Створене програмне забезпечення використовувалось в навчальних цілях.

Використані джерела:

1. Apress, Inc Ray Tracing Gems https://www.realtimerendering.com/raytracinggems/rtg/index.html
2. Apress, Inc Ray Tracing Gems II https://www.realtimerendering.com/raytracinggems/rtg2/index.html
3. David Lettier 3D Game Shaders For Beginners https://github.com/lettier/3d-game-shaders-for-beginners
4. David Lettier 3D Game Shaders For Beginners: Lighting https://github.com/lettier/3d-game-shaders-for-beginners/blob/master/sections/lighting.md
5. David Lettier 3D Game Shaders For Beginners: SSR https://github.com/lettier/3d-game-shaders-for-beginners/blob/master/sections/screen-space-reflection.md
6. David Lettier 3D Game Shaders For Beginners: Texturing https://github.com/lettier/3d-game-shaders-for-beginners/blob/master/sections/texturing.md
7. Epic Games, Inc. Directional Lights https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/LightingAndShadows/LightTypes/Directional/
8. Epic Games, Inc. Game Flow Overview https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Framework/GameFlow/
9. Epic Games, Inc. Graphics Programming Overview https://docs.unrealengine.com/5.2/en-US/graphics-programming-overview-for-unreal-engine/
10. Epic Games, Inc. Guidelines for Optimizing Rendering for Real-Time https://docs.unrealengine.com/5.2/en-US/guidelines-for-optimizing-rendering-for-real-time-in-unreal-engine/
11. Epic Games, Inc. Planar Reflections https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/LightingAndShadows/PlanarReflections/
12. Epic Games, Inc. Reflection Environment https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/LightingAndShadows/ReflectionEnvironment/
13. Epic Games, Inc. Reflections Captures https://docs.unrealengine.com/5.0/en-US/reflections-captures-in-unreal-engine/
14. Epic Games, Inc. Scalability Reference https://docs.unrealengine.com/4.26/en-US/TestingAndOptimization/PerformanceAndProfiling/Scalability/
15. Epic Games, Inc. Screen Space Reflections https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/PostProcessEffects/ScreenSpaceReflection/
16. Epic Games, Inc. Sky Light https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/LightingAndShadows/LightTypes/SkyLight/
17. Epic Games, Inc. Static Lights https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/LightingAndShadows/LightMobility/StaticLights/

18. Epic Games, Inc. Texture Streaming Overview https://docs.unrealengine.com/5.2/en-US/texture-streaming-overview-for-unreal-engine/
19. Epic Games, Inc. Threaded Rendering https://docs.unrealengine.com/5.2/en-US/threaded-rendering-in-unreal-engine/
20. Epic Games, Inc. Understanding Lightmapping in Unreal Engine https://docs.unrealengine.com/5.0/en-US/understanding-lightmapping-in-unreal-engine/
21. Epic Games, Inc. Unreal Engine Hardware Ray Tracing https://docs.unrealengine.com/5.2/en-US/hardware-ray-tracing-in-unreal-engine/
22. Epic Games, Inc. Unreal Engine Hardware Ray Tracing Tips and Tricks https://docs.unrealengine.com/5.2/en-US/hardware-ray-tracing-tips-and-tricks-in-unreal-engine/
23. Joey de Vries Learn OpenGL – Graphics Programming https://learnopengl.com/book/book_pdf.pdf
24. Microsoft Corporation Direct3D 12 programming guide https://learn.microsoft.com/en-us/windows/win32/direct3d12/directx-12-programming-guide
25. Microsoft Corporation Getting started with DirectX Graphics https://learn.microsoft.com/en-us/windows/win32/getting-started-with-directx-graphics
26. Microsoft Corporation Getting Started With XInput in Windows applications https://learn.microsoft.com/en-us/windows/win32/xinput/getting-started-with-xinput
27. Microsoft Corporation Programming Guide (XInput Game Controller APIs) https://learn.microsoft.com/en-us/windows/win32/xinput/programming-guide
28. Nvidia Corporation DIRECTX 12 ULTIMATE https://developer.nvidia.com/directx
29. Nvidia Corporation NVIDIA RTX Path Tracing SDK https://developer.nvidia.com/rtx/path-tracing
30. Nvidia Corporation Ray Tracing Essentials 1:https://developer.nvidia.com/blog/ray-tracing-essentials-part-1-basics-of-ray-tracing/
31. Nvidia Corporation Ray Tracing Essentials 2:https://developer.nvidia.com/blog/ray-tracing-essentials-part-2-rasterization-versus-ray-tracing/
32. Nvidia Corporation Ray Tracing Essentials 3:https://developer.nvidia.com/blog/ray-tracing-essentials-part-3-ray-tracing-hardware/
33. Nvidia Corporation Ray Tracing Essentials 4:https://developer.nvidia.com/blog/ray-tracing-essentials-part-4-the-ray-tracing-pipeline/
34. Nvidia Corporation Ray Tracing Essentials 5:https://developer.nvidia.com/blog/ray-tracing-essentials-part-5-ray-tracing-effects/

35. Nvidia Corporation Ray Tracing Essentials 6:https://developer.nvidia.com/blog/ray-tracing-essentials-part-6-the-rendering-equation/
36. Nvidia Corporation Ray Tracing Essentials 7:https://developer.nvidia.com/blog/ray-tracing-essentials-part-7-denoising-for-ray-tracing/
37. Nvidia Corporation Ray Tracing Resources Page https://www.realtimerendering.com/raytracing.html
38. Nvidia Corporation RTX Technology https://developer.nvidia.com/rtx/ray-tracing
39. Nvidia Corporation What is Path Tracing? https://blogs.nvidia.com/blog/2022/03/23/what-is-path-tracing/
40. Physically Based Rendering: From Theory To Implementation https://www.pbr-book.org/3ed-2018/contents
41. Pixar Animation Studios Art and Technology at Pixar https://graphics.pixar.com/library/SigAsia2018Course/paper.pdf
42. Pixar Animation Studios Ray Tracing for the Movie 'Cars' https://graphics.pixar.com/library/RayTracingCars/paper.pdf
43. Pixar Animation Studios RenderMan, Theory and Practice https://graphics.pixar.com/library/RMan2003/paper.pdf
44. Pixar Animation Studios Revamping the Cloth Tailoring Pipeline at Pixar https://graphics.pixar.com/library/C3d/paper.pdf
45. Pixar Animation Studios Stochastic Simplification of Aggregate Detail https://graphics.pixar.com/library/StochasticSimplification/paper.pdf
46. Pixar Animation Studios Volume Rendering for Pixar's Elemental https://graphics.pixar.com/library/ElementalVolume/paper.pdf
47. Real-Time Rendering Fourth Edition, Online chapter: Real-Time Ray Tracing https://www.realtimerendering.com/Real-Time_Rendering_4th-Real-Time_Ray_Tracing.pdf
48. The Walt Disney Company A Material Point Method For Snow Simulation https://media.disneyanimation.com/uploads/production/publication_asset/94/asset/SSCTS13_2.pdf
49. The Walt Disney Company Denoising with Kernel Prediction and Asymmetric Loss Functions https://la.disneyresearch.com/publication/denoising-with-kpal/
50. The Walt Disney Company Sorted Deferred Shading for Production Path Tracing https://media.disneyanimation.com/uploads/production/publication_asset/70/asset/Sorted_Deferred_Shading_For_Production_Path_Tracing.pdf
51. The Walt Disney Company Subdivision Next-Event Estimation for Path-Traced Subsurface Scattering https://media.disneyanimation.com/uploads/production/publication_asset/179/asset/SNEE.pdf
52. The Walt Disney Company The Design and Evolution of Disney's Hyperion Renderer https://media.disneyanimation.com/uploads/production/publication_asset/177/asset/a.pdf

Додаток:

Компонент взаємодії.

```
// Copyright 2023 Bohdan Basok. All rights reserved

#pragma once

#include "CoreMinimal.h"
#include "Components/ActorComponent.h"
#include "RInteractionComponent.generated.h"


UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
class ROBOACADEMY_API URInteractionComponent : public
UActorComponent
{
	GENERATED_BODY()

public:
	void DoInteraction();
	void OutlineCast();
	UPROPERTY(EditAnywhere, Category = "LineCast Stats")
		float Radius = 30.0f;
public:
	// Sets default values for this component's properties
	URInteractionComponent();

	//Used to store previous hit to toggle stencil value back to 0
	AActor* OldHitTarget = GetOwner();
protected:
	// Called when the game starts
	virtual void BeginPlay() override;

public:
	// Called every frame
	virtual void TickComponent(float DeltaTime, ELevelTick TickType,
FActorComponentTickFunction* ThisTickFunction) override;


};

// Copyright 2023 Bohdan Basok. All rights reserved


#include "RInteractionComponent.h"
```

```
#include "RInteractionInterface.h"

// Sets default values for this component's properties
URInteractionComponent::URInteractionComponent()
{
	// Set this component to be initialized when the game starts, and to be
ticked every frame.  You can turn these features
	// off to improve performance if you don't need them.
	PrimaryComponentTick.bCanEverTick = true;

	// ...
}


// Called when the game starts
void URInteractionComponent::BeginPlay()
{
	Super::BeginPlay();

	// ...

}


// Called every frame
void URInteractionComponent::TickComponent(float DeltaTime, ELevelTick
TickType, FActorComponentTickFunction* ThisTickFunction)
{
	Super::TickComponent(DeltaTime, TickType, ThisTickFunction);

	OutlineCast();
}

//Linecast, changes stencil value to 1 on hit if hit actor implements
RInteractionInterface
//if not change value back to 0
void URInteractionComponent::OutlineCast()
{

	FCollisionObjectQueryParams ObjectQueryParams;
	ObjectQueryParams.AddObjectTypesToQuery(ECC_Pawn);

	AActor* MyOwner = GetOwner();
```

```
	FVector EyeLocation;
	FRotator EyeRotation;
	MyOwner->GetActorEyesViewPoint(EyeLocation, EyeRotation);

	//Sets maximum linecast length in cm
	FVector End = EyeLocation + (EyeRotation.Vector() * 400);

	TArray<FHitResult> Hits;

	FCollisionShape Shape;
	Shape.SetSphere(Radius);

	bool bBlockingHit = GetWorld()->SweepMultiByObjectType(Hits,
EyeLocation, End, FQuat::Identity, ObjectQueryParams, Shape);

	for (FHitResult Hit : Hits)
	{
		AActor* HitActor = Hit.GetActor();

		if (HitActor)
		{
			if (HitActor->Implements<URInteractionInterface>())
			{
				TArray<USkeletalMeshComponent*> Components;
				Components.Empty();

	HitActor->GetComponents<USkeletalMeshComponent>(Components);

				USkeletalMeshComponent* HitActorMesh =
Components[0];
				HitActorMesh->SetCustomDepthStencilValue(1);

				OldHitTarget = HitActor;

				break;
			}
			else
			{

					TArray<USkeletalMeshComponent*>
Components;
					Components.Empty();

	OldHitTarget->GetComponents<USkeletalMeshComponent>(Component
s);
```

```
                                        USkeletalMeshComponent* HitActorMesh =
Components[0];

        HitActorMesh->SetCustomDepthStencilValue(0);


                }
            }
        }
}

void URInteractionComponent::DoInteraction()
{
        FCollisionObjectQueryParams ObjectQueryParams;
        ObjectQueryParams.AddObjectTypesToQuery(ECC_Pawn);

        AActor* MyOwner = GetOwner();

        FVector EyeLocation;
        FRotator EyeRotation;
        MyOwner->GetActorEyesViewPoint(EyeLocation, EyeRotation);

        //Sets maximum linecast length in cm
        FVector End = EyeLocation + (EyeRotation.Vector() * 400);

        TArray<FHitResult> Hits;

        FCollisionShape Shape;
        Shape.SetSphere(Radius);

        bool bBlockingHit = GetWorld()->SweepMultiByObjectType(Hits,
EyeLocation, End, FQuat::Identity, ObjectQueryParams, Shape);

        for (FHitResult Hit : Hits)
        {
            AActor* HitActor = Hit.GetActor();
            if (HitActor)
            {
                if (HitActor->Implements<URInteractionInterface>())
                {
                    //Cast to pawn
                    APawn* MyPawn = Cast<APawn>(MyOwner);
                    IRInteractionInterface::Execute_Interaction(HitActor,
MyPawn);
                    break;
                }
```

```
            }
        }
}
```

Інтерфейс взаємодії.

```
// Copyright 2023 Bohdan Basok. All rights reserved

#pragma once

#include "CoreMinimal.h"
#include "UObject/Interface.h"
#include "RInteractionInterface.generated.h"

// This class does not need to be modified.
UINTERFACE(MinimalAPI)
class URInteractionInterface : public UInterface
{
        GENERATED_BODY()
};

/**
 *
 */
class ROBOACADEMY_API IRInteractionInterface
{
        GENERATED_BODY()

        // Add interface functions to this class. This is the class that will be
inherited to implement this interface.
public:
        UFUNCTION(BlueprintNativeEvent, BlueprintCallable)
                void Interaction(APawn* InstigatorPawn);
};

// Copyright 2023 Bohdan Basok. All rights reserved


#include "RInteractionInterface.h"

// Add default functionality here for any IRInteractionInterface functions that are
not pure virtual.
```

```
// Copyright 2023 Bohdan Basok. All rights reserved

#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Character.h"
#include "InputActionValue.h"
#include "RoboAcademyCharacterFP.generated.h"

class UInputComponent;

class USceneComponent;
class UCameraComponent;
class UAnimMontage;
class USoundBase;

class URInteractionComponent;

UCLASS()
class ROBOACADEMY_API ARoboAcademyCharacterFP : public ACharacter
{
	GENERATED_BODY()


	/** First person camera */
	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category =
Camera, meta = (AllowPrivateAccess = "true"))
		UCameraComponent* FirstPersonCameraComponent;

	/** MappingContext */
	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input,
meta = (AllowPrivateAccess = "true"))
		class UInputMappingContext* DefaultMappingContext;

	/** Jump Input Action */
	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input,
meta = (AllowPrivateAccess = "true"))
		class UInputAction* JumpAction;

	/** Move Input Action */
	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input,
meta = (AllowPrivateAccess = "true"))
```

```
		class UInputAction* MoveAction;

	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input,
meta = (AllowPrivateAccess = "true"))
		class UInputAction* Interaction;

	UPROPERTY(VisibleAnywhere)
	URInteractionComponent* InteractionComp;
public:
	// Sets default values for this character's properties
	ARoboAcademyCharacterFP();

protected:
	// Called when the game starts or when spawned
	virtual void BeginPlay() override;

public:

	/** Look Input Action */
	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Input,
meta = (AllowPrivateAccess = "true"))
	class UInputAction* LookAction;

	// Called every frame
	virtual void Tick(float DeltaTime) override;

protected:
	/** Called for movement input */
	void Move(const FInputActionValue& Value);

	/** Called for looking input */
	void Look(const FInputActionValue& Value);

	void Interact();

protected:
	// APawn interface
	virtual void SetupPlayerInputComponent(UInputComponent*
InputComponent) override;
	// End of APawn interface

public:

	/** Returns FirstPersonCameraComponent subobject **/
	UCameraComponent* GetFirstPersonCameraComponent() const { return
FirstPersonCameraComponent; }
```

```
};

// Copyright 2023 Bohdan Basok. All rights reserved


#include "RoboAcademyCharacterFP.h"
#include "Animation/AnimInstance.h"
#include "Camera/CameraComponent.h"
#include "Components/CapsuleComponent.h"
#include "EnhancedInputComponent.h"
#include "EnhancedInputSubsystems.h"
#include "RInteractionComponent.h"

// Sets default values
ARoboAcademyCharacterFP::ARoboAcademyCharacterFP()
{
	// Set this character to call Tick() every frame.  You can turn this off to
improve performance if you don't need it.
	PrimaryActorTick.bCanEverTick = true;
	// Set size for collision capsule
	GetCapsuleComponent()->InitCapsuleSize(55.f, 96.0f);

	// Create a CameraComponent
	FirstPersonCameraComponent =
CreateDefaultSubobject<UCameraComponent>(TEXT("FirstPersonCamera"));
	FirstPersonCameraComponent->SetupAttachment(GetCapsuleComponent
());
	FirstPersonCameraComponent->SetRelativeLocation(FVector(-10.f, 0.f,
60.f)); // Position the camera
	FirstPersonCameraComponent->bUsePawnControlRotation = true;

	InteractionComp =
CreateDefaultSubobject<URInteractionComponent>("Interaction Component");
}

// Called when the game starts or when spawned
void ARoboAcademyCharacterFP::BeginPlay()
{
	Super::BeginPlay();
	//Add Input Mapping Context
	if (APlayerController* PlayerController =
Cast<APlayerController>(Controller))
	{
```

```
		if (UEnhancedInputLocalPlayerSubsystem* Subsystem =
ULocalPlayer::GetSubsystem<UEnhancedInputLocalPlayerSubsystem>(PlayerC
ontroller->GetLocalPlayer()))
		{
			Subsystem->AddMappingContext(DefaultMappingContext,
0);
		}
	}
}

// Called every frame
void ARoboAcademyCharacterFP::Tick(float DeltaTime)
{
	Super::Tick(DeltaTime);

}

// Called to bind functionality to input
void
ARoboAcademyCharacterFP::SetupPlayerInputComponent(UInputComponent*
PlayerInputComponent)
{
	Super::SetupPlayerInputComponent(PlayerInputComponent);
	// Set up action bindings
	if (UEnhancedInputComponent* EnhancedInputComponent =
CastChecked<UEnhancedInputComponent>(PlayerInputComponent))
	{
		//Jumping
		EnhancedInputComponent->BindAction(JumpAction,
ETriggerEvent::Triggered, this, &ACharacter::Jump);
		EnhancedInputComponent->BindAction(JumpAction,
ETriggerEvent::Completed, this, &ACharacter::StopJumping);

		//Moving
		EnhancedInputComponent->BindAction(MoveAction,
ETriggerEvent::Triggered, this, &ARoboAcademyCharacterFP::Move);

		//Looking
		EnhancedInputComponent->BindAction(LookAction,
ETriggerEvent::Triggered, this, &ARoboAcademyCharacterFP::Look);
		//Interaction
		EnhancedInputComponent->BindAction(Interaction,
ETriggerEvent::Triggered, this, &ARoboAcademyCharacterFP::Interact);
	}
}
void ARoboAcademyCharacterFP::Move(const FInputActionValue& Value)
```

```
{
	// input is a Vector2D
	FVector2D MovementVector = Value.Get<FVector2D>();

	if (Controller != nullptr)
	{
		// add movement
		AddMovementInput(GetActorForwardVector(),
MovementVector.Y);
		AddMovementInput(GetActorRightVector(), MovementVector.X);
	}
}

void ARoboAcademyCharacterFP::Look(const FInputActionValue& Value)
{
	// input is a Vector2D
	FVector2D LookAxisVector = Value.Get<FVector2D>();

	if (Controller != nullptr)
	{
		// add yaw and pitch input to controller
		AddControllerYawInput(LookAxisVector.X);
		AddControllerPitchInput(LookAxisVector.Y);
	}
}
void ARoboAcademyCharacterFP::Interact()
{
	InteractionComp->DoInteraction();
}
```