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 магістр
(бакалавр, спеціаліст, магістр)

Розробка інтерактивної комп'ютерної графіки реального часу, її особливості та принципи роботи

Development of real-time interactive computer graphics, its features and how it works

Науковий керівник:
кандидат технічних наук,
викладач кафедри
комп’ютерної інженерії та
електроніки
Котик М.В.

Рецензент:
кандидат технічних наук,
викладач
Петришин Михайло
Любомирович

Івано-Франківськ
2024

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 в своєму житті часто зустрічаються з результатами роботи комп’ютерної графіки в реальному часі. В цій науковій роботі буде розглянуто комп’ютерну графіку і принципи її роботи. Буде створено власне програмне забезпечення, що буде працювати з комп’ютерною графікою. Створення програмного забезпечення подібної рівня складності не тільки демонструє засвоєння матеріалу, але і успішно проведене дослідження галузі комп’ютерної графіки. Воно також буде використовуватись в цілях демонстрації описаних в роботі алгоритмів. З допомогою цією програми стає можливим опис складних тем в більш зрозумілий спосіб.

ABSTRACT

People during their day often encounter the results of real-time computer graphics. This research paper will take a look at computer graphics and explain the fundamentals. The creation of such a complex computer software not only demonstrates my success in obtaining knowledge but also in researching the entire industry of computer graphics. That being said the software itself can also be used as a mean to demonstrate mentioned algorithms. Using this software, it is now possible to explain complex subjects in a more 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
Розробка інтерактивної комп'ютерної графіки реального часу, її особливості та принципи роботи

					123.КІ(М)-24.02		
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) – це спосіб взаємодії однієї програми з іншою простіше кажучи набір компонентів для взаємодії [24].

RAM (Random Access Memory) – це оперативна пам’ять. Тобто вид пам’яті комп’ютера, який може надавати доступ до будь-якої комірки пам’яті з метою запису чи зчитування даних[23].

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

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

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

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

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

Сцена – набір різних об’єктів як геометрія, джерела світла, камери та решти всього необхідного для створення остаточного зображення[23].

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

Шейдер (Shader) –код, він є невеликою програмою яка працює на окремому від основної програми потоці виконання і вираховує реакцію поверхні на світло. Його принципи роботи сильно відрізняються в графіці реального часу і в традиційній графіці[3].

Зміст

Вступ

Майже кожен день ми можемо спостерігати результати роботи комп’ютерної графіки реального часу Розуміння принципів роботи чогось настільки поширеного дає людині немало перспектив. У цій роботі розглядаються не лише принципи комп’ютерної графіки та її функціонування, але й її зв’язок з апаратним забезпеченням. Особливу увагу приділено особливостям взаємодії між апаратним і програмним забезпеченням комп'ютера на прикладі комп’ютерної графіки в реальному часі. Дипломна робота спрямована на глибше розуміння процесів взаємодії програмного забезпечення з апаратним, що забезпечує ефективну роботу комп'ютерної графіки.

1.Комп’ютерна графіка статичного виду

Статична комп'ютерна графіка — це поширений метод створення зображень, який застосовується в проектах, що вимагають високої якості й деталізації фінального результату, як-от у кіновиробництві. Для досягнення максимальної якості зображення використовують традиційні методи, що базуються на складних математичних обчисленнях, які виконуються комп'ютерами, а інколи й суперкомп'ютерами. Такі розрахунки займають значний час, тому процес генерації зображень може тривати від кількох годин до днів.

Термін "комп’ютерна графіка" має багато значень і часто стосується всіх візуальних зображень, що відображаються на комп'ютері, окрім аудіо й тексту. Проте точніше його можна визначити як процес створення або маніпуляції зображеннями. Мотиви й цілі використання комп’ютерної графіки різноманітні: вона широко застосовується не лише в кіноіндустрії, а й у багатьох інших сферах. У кінематографі, наприклад, комп’ютерна графіка дає змогу створювати сцени, деталі чи спецефекти, що є занадто складними, дорогими або взагалі неможливими для реалізації за допомогою традиційних методів. Крім того, вона є потужним інструментом для візуалізації даних, відкриваючи безмежні можливості для її застосування.

Хоча комп’ютерна графіка суттєво відрізняється від живопису чи фотографії, вона здатна досягати схожої візуальної якості. У цієї технології є свої сильні й слабкі сторони, переваги та недоліки. Її принципи роботи еволюціонували з часом і можуть значно різнитися залежно від конкретної реалізації. Кожен розробник або компанія може мати унікальні програми та методи, що працюють по-різному. Втім, існують також загальні підходи й принципи, які є спільними для

багатьох реалізацій.

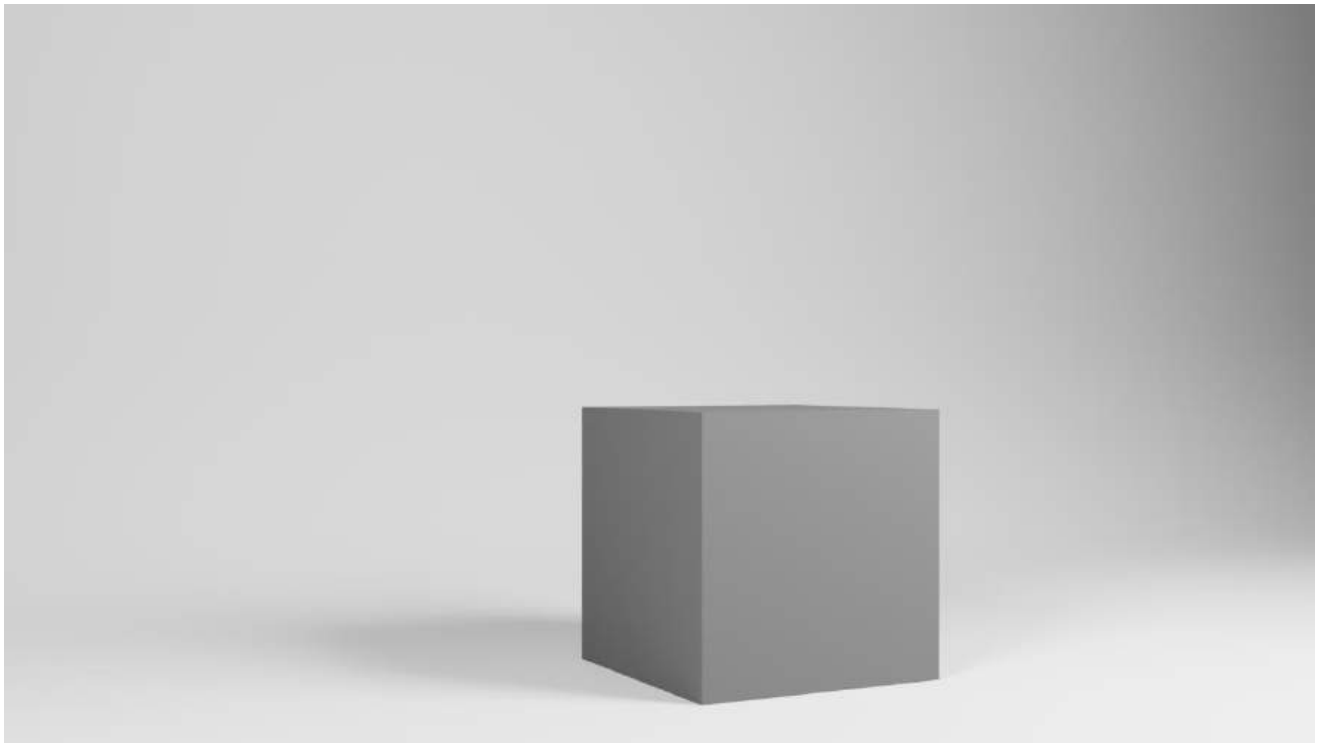


Рисунок 1.1 – Створений в комп’ютерній графіці куб.

1.1. Порівняння з видами графіки, що не використовують комп’ютер

Особливістю комп’ютерної графіки, яка відрізняє її від фотографії чи живопису, є застосування комп’ютера для точних розрахунків. Під час створення зображення комп’ютер виконує необхідні математичні обчислення для генерації кінцевого результату, імітуючи реальний світ із високою точністю. Ці обчислення потребують значного часу й обчислювальних ресурсів. Наприклад, якщо художник зображає тінь від дерева, він сам визначає її форму і розташування. Комп’ютер, виконуючи розрахунки в рамках 3D-графіки, також генерує тінь, однак може враховувати фізичні параметри освітлення й створювати зображення з високим рівнем реалістичності. Водночас це не робить роботу комп’ютера автоматично кращою за художню, проте спрощує процес створення реалістичного результату. Крім того, комп’ютерна графіка відкриває можливості для створення об’єктів, які неможливо відтворити в реальному світі: наприклад, дерево може мати фіолетове листя або "літати" у повітрі — такі зображення складно відтворити у фотографії.

1.2. Основи комп’ютерної графіки

Основний принцип роботи тривимірної комп’ютерної графіки базується на використанні тривимірних геометричних даних та інших збережених у комп’ютері інформаційних ресурсів для виконання обчислень і створення цифрових зображень. Для розрахунків у традиційній комп’ютерній графіці застосовуються центральний процесор або графічний процесор. Кожен з цих процесорів має свої переваги та відповідні сфери застосування. Існує можливість одночасного використання обох процесорів, що значно ускладнює розробку програмного забезпечення. Центральний процесор є більш універсальним і простішим для розробки програм, а графічний процесор, завдяки своїй архітектурі, дозволяє пришвидшити обчислення, якщо їх можна розбити на численні дрібні операції, для чого потрібні спеціалізовані алгоритми.

Дані, які необхідні для генерації фінального зображення, завантажуються у пам’ять. Залежно від типу процесора, ці дані зберігаються у відповідній пам’яті: оперативній або відеопам’яті. Для обчислення графіки часто використовують спеціалізоване апаратне забезпечення з великою кількістю пам’яті, оскільки для роботи потрібно завантажити всі ресурси у пам’ять системи. Таким чином, обсяг пам’яті є критично важливим для процесу обчислень.

Один з недоліків цього процесу — значна складність обчислень. Навіть для генерації простих зображень низької якості на потужних комп’ютерах може знадобитися кілька хвилин. У кінематографі стандартним є час обробки одного кадру, що триває близько 8 годин, при цьому частота кадрів у фільмах становить 24 і більше кадрів на секунду, тому для обчислень використовують суперкомп’ютери. Наприклад, для створення фільму "Аватар" Джеймса Кемерона 2009 року використовували суперкомп’ютер із площею у 972 м², оперативною пам'яттю 100 терабайт та дисковим сховищем у 2,5 петабайти. На обробку одного кадру цього фільму потрібно було близько семи з половиною годин.

1.3. 3D моделі

3D модель — це математична репрезентація об’єкта у тривимірному просторі на основі координатної системи [1]. Вона створюється за допомогою спеціалізованого програмного забезпечення і складається з точок, трикутників, ребер, полігонів або ліній. Однак у своїй основі 3D модель є сукупністю трикутників. Для того щоб модель стала видимою, вона має бути щонайменше двовимірною; найпростішим таким об’єктом є трикутник.

Одна точка в тривимірній графіці, що відповідає певним координатам, не має жодних розмірів: ні висоти, ні ширини, ні довжини, тому вона невидима. Дві точки, з'єднані між собою, утворюють лінію, яка має довжину й є одновимірною, але також залишається невидимою. Лише три точки, що утворюють трикутник, формують двовимірний об’єкт, який стає видимим для людського ока.

У ранній комп’ютерній графіці як основу використовували чотирикутники, проте трикутники виявилися більш ефективними. По-перше, чотирикутник можна розбити на два трикутники, зменшуючи потребу в їхньому використанні як базових елементів. По-друге, трикутники завжди утворюють плоску поверхню, оскільки три будь-які точки в просторі визначають площину. У випадку з чотирикутниками ця властивість не завжди виконується, що ускладнює обробку. Крім того, будь-який тривимірний об’єкт можна побудувати, використовуючи трикутники, що робить їх зручним та ефективним елементом для комп’ютерних обчислень.

Кожен трикутник має нормаль — перпендикулярну до нього лінію, яка визначає орієнтацію трикутника в просторі та використовується в розрахунках освітлення. Сукупність таких трикутників утворює 3D модель. Хоча модель може включати додаткові дані, такі як властивості матеріалу та інформацію для анімації.

Рисунок 1.2 – Візуалізація сітки трикутників 3D моделі.

1.4. Шейдери

На цьому етапі 3D модель представляє собою лише сукупність геометричних даних, яка не може самостійно взаємодіяти зі світлом. Для забезпечення такої взаємодії необхідно використовувати шейдери. Шейдер — це програмний код, який розраховує поведінку поверхні моделі під час взаємодії зі світлом під час рендерингух[3]. Шейдер можна розглядати як «матеріал» поверхні, оскільки він визначає фізичні властивості поверхні, подібно до матеріалів реальних об’єктів. Ранні версії шейдерів були відносно простими та мали обмежену взаємодію зі світлом, але з часом їхня складність зросла, що дало змогу значно наблизити візуальні ефекти до реальності. Шейдери визначають колір поверхні, відбивні властивості, прозорість та здатність поверхні випромінювати світло, надаючи об’єктам широкий спектр характеристик.

Код шейдерів можна писати різними способами, що робить їх реалізацію унікальною для кожного розробника. Навіть використовуючи однакові математичні формули, можливо отримати різний результат завдяки відмінностям у коді. Крім того, код може бути оптимізований шляхом виключення зайвих обчислень або заміни формул на власні, більш ефективні варіанти.

1.5 Текстури

Шейдери зберігають свої властивості у числовому форматі. Наприклад, колір задається комбінацією компонентів червоного, зеленого та синього (RGB), іноді з додаванням альфа-каналу, що відповідає за прозорість. Кожен із цих компонентів представлений трьома 8-бітними значеннями від 0 до 255. Застосування шейдера до об’єкта або набору трикутників забарвлює всю поверхню в один колір, що може не відповідати реальним потребам, оскільки природні поверхні зазвичай складаються з багатьох відтінків. Окреме зафарбовування кожного трикутника є складним процесом, що знижує продуктивність, особливо на моделях низької складності.

Для розв'язання цієї проблеми були створені текстури. Текстура — це двовимірне зображення, яке накладається на тривимірний об’єкт, щоб забезпечити різноманітні кольорові значення для обчислень[2]. Наприклад, текстури можуть надавати поверхні кольорові відтінки або деталі. Завантажені у пам’ять системи, текстури використовуються шейдерами під час рендерингу.

Оскільки текстура є двовимірною, її потрібно правильно «обгорнути» навколо тривимірного об’єкта. Для цього кожній точці моделі призначається положення в спеціальній проекції на двовимірній площині, що нагадує процес загортання подарунка у обгортковий папір. Цей процес здебільшого виконується вручну, проте спеціальне програмне забезпечення може автоматично спробувати здійснити обгортання.

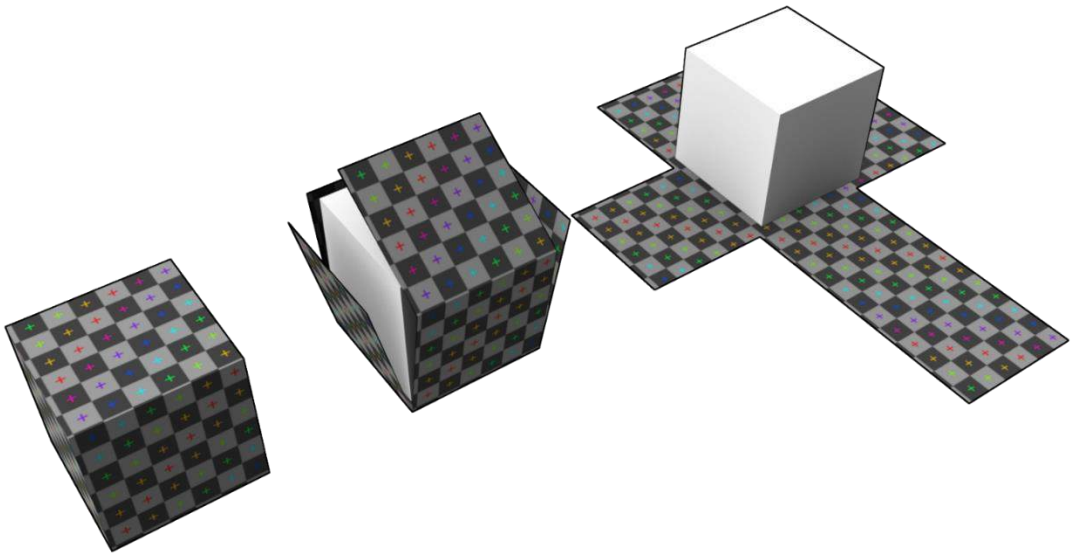


Рисунок 1.3 – Візуальна репрезентація обгортки.

1.6. Трасування шляху

У комп'ютерній графіці освітлення відіграє ключову роль, оскільки без нього зображення стають невидимими — як у реальному світі, так і в цифрових сценах. Існує декілька методів обробки світла, залежно від обраного алгоритму, проте одним з найпоширеніших є трасування шляху.

Трасування шляху (або трасування напрямку) — це метод рендерингу, який обчислює поведінку світла при взаємодії з поверхнями об’єктів[1]. Цей підхід визначає, як різні промені світла, що потрапляють на певну точку поверхні, взаємодіють з матеріалом та які компоненти освітлення врешті-решт передаються до віртуальної камери. Принцип роботи методу полягає у запуску променя в заданому напрямку, щоб відслідковувати його маршрут при взаємодії з об’єктами сцени. Ця ідея не є унікальною для графічної сфери: схожий принцип застосовується у фізиці, наприклад, для вимірювання радіаційного потоку. Промені можуть бути направлені в будь-які точки сцени для перевірки наявності об'єктів на їхньому шляху.

Ранні алгоритми використовували спрощені обчислення. Із віртуальної камери, яка лише симулювала реальну, випускалися промені, і піксель відображав перший об'єкт, з яким промінь зіткнувся. Додатково, від точки зіткнення до джерела світла проводився промінь, і на основі результатів розраховувалася освітленість піксель який метод був доволі неточним, адже не враховував складні взаємодії світла. Недоліком цього підходу було те, що кожен піксель мав або повну освітленість, або ж повну тінь, що створювало жорсткий контраст без плавного переходу між темнішими і світлішими тонами. Відсутність таких переходів робила тіні однорідно темними, надаючи зображенням нереалістичного вигляду.

Рисунок 1.4 – Ранній результат роботи трасування шляху.

1.7. Сучасний стан комп’ютерної графіки

У сучасних методах рендерингу при зіткненні променя зі поверхнею він не зупиняється, а відбивається, що імітує реальну фізичну поведінку світла. Крім того, джерело світла більше не вважається точкою в просторі, а має фізичний розмір, що наближає процес рендерингу до реальних умов. Фізичний розмір джерела світла дозволяє створювати більш природні тіні. В залежності від розмірів джерела та кута падіння променів на поверхню, можливе утворення м'яких тіней, які характеризуються плавним переходом від темних до світлих ділянок.

У процесі рендерингу кожен піксель отримує множину променів, що відбиваються в різних напрямках, при цьому кожен промінь може відбиватися від поверхонь декілька разів. Кожен з цих променів несе в собі інформацію про колір, залежно від того, на яку поверхню він потрапляв. Коли трасування для пікселя завершено, сумарний результат обчислень дає кінцевий колір цього пікселя. Зазвичай для точного рендерингу використовують до двадцяти мільйонів променів, кожен з яких може відбиватися до 32 разів, і це число застосовується до кожного пікселя зображення. Для кіноіндустрії ці показники можуть бути значно вищими.

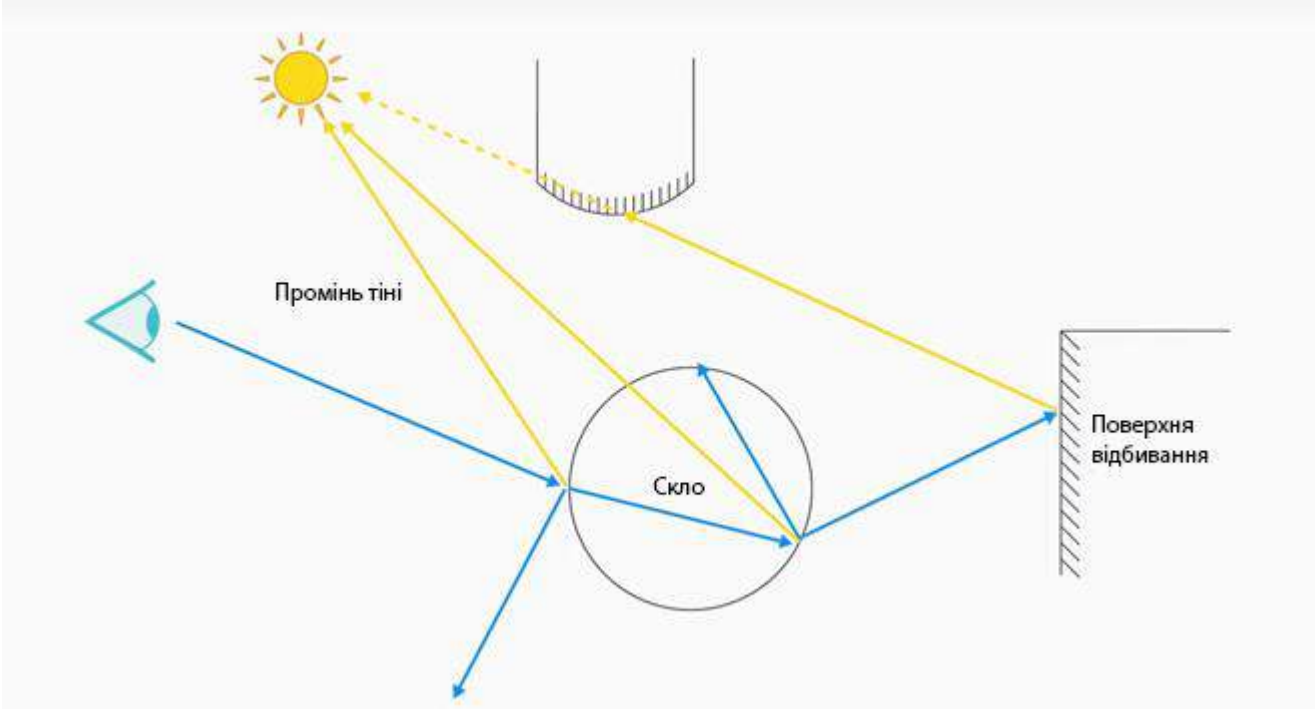


Рисунок 1.5 – Візуальна репрезентація трасування шляху.

У сучасних методах рендерингу яскравість освітлення залежить від кута падіння променя світла, а також від напрямку нормалі кожного окремого трикутника моделі, який потрапляє під його вплив. Це явище описується законом освітленості Ламберта, який був сформульований Йоганном Генріхом Ламбертом, видатним німецьким науковцем, у 1760 році.

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

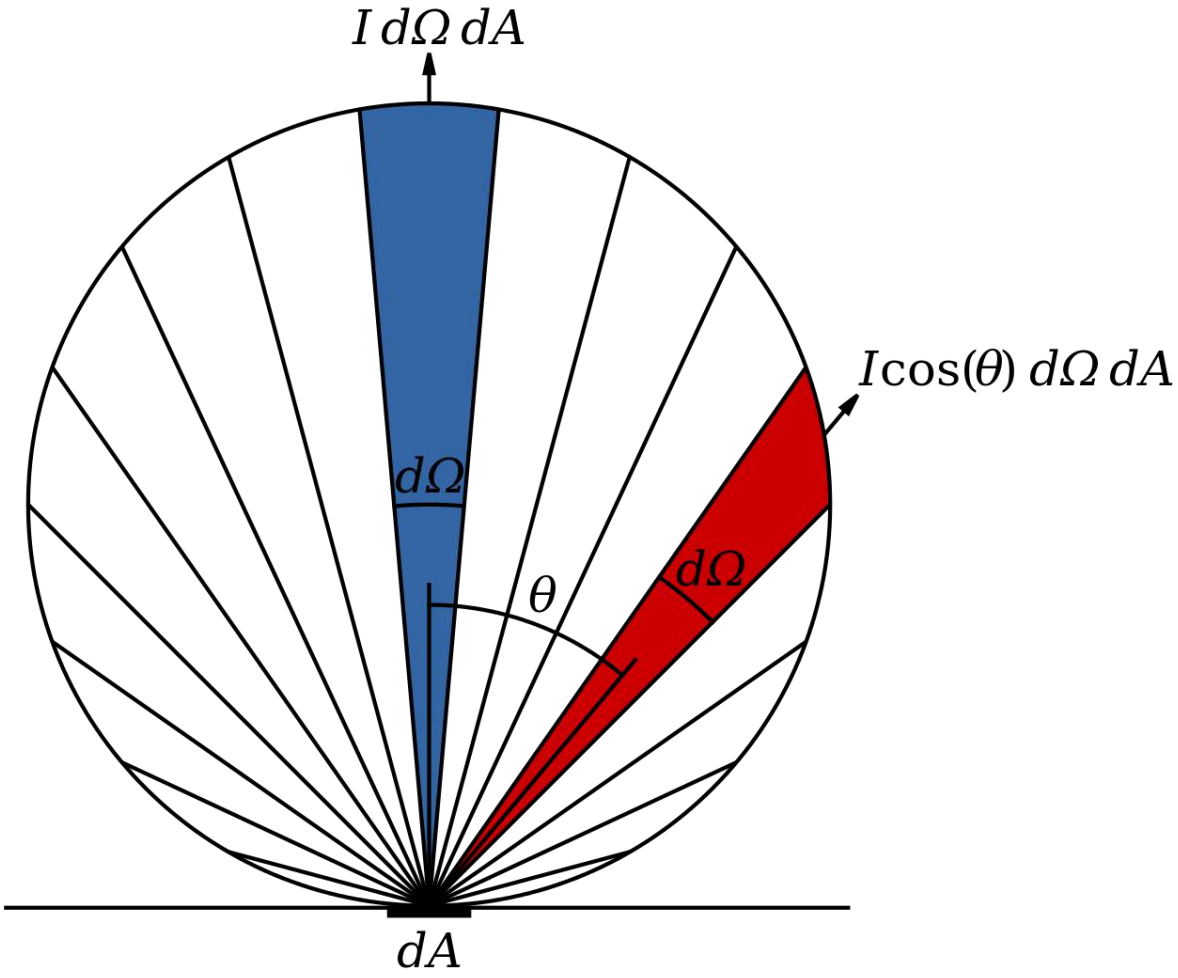


Рисунок 1.6 – Візуальна репрезентація формули Ламберта (1.1).

Згідно з цим законом (1.1), освітленість поверхні визначається залежно від кута між напрямком падіння світла і нормаллю поверхні. Якщо світло падає на поверхню під кутом, близьким до перпендикулярного, тобто практично по нормалі, то освітленість буде максимальною. Водночас, чим більше відхиляється кут падіння від нормалі, тим менше буде освітлення цієї поверхні.

Окрім того, сучасна комп’ютерна графіка часто застосовує штучний інтелект для покращення якості зображень. Однією з його функцій є усунення шумів —

графічних артефактів, таких як чорні пікселі, які з’являються через недостатню кількість обчислених променів, коли промінь не досягає джерела світла. Шум можна усунути, запустивши більшу кількість променів, але це вимагає значних додаткових обчислювальних ресурсів і часу. Для того, щоб прискорити процес і отримати чисте зображення без шуму, штучний інтелект застосовується для корекції артефактів, що дозволяє значно зменшити час рендерингу, одночасно зберігаючи високу якість кінцевого результату.

Рисунок 1.7 - Приклад роботи штучного інтелекту.

1.8. Історія розвитку комп’ютерної графіки

Ранні спроби та експериментальні варіанти реалізації комп’ютерної графіки можна знайти вже в 1950-х роках. Однак, значний прорив, що дозволив вивести комп’ютерну графіку на новий рівень реалізму, стався в 1970-х роках.

1.8.1. Чайник з Юти

Одним із найвідоміших об’єктів у галузі комп’ютерної графіки є чайник з Юти, який став своєрідним символом цієї сфери, подібно до того, як «Hello, World!» є стандартом у програмуванні. Цей чайник настільки популярний, що регулярно з'являється на задніх планах у різних комп'ютерних анімаціях, фільмах та інших мультимедійних продуктах. Зокрема, модель чайника була використана в усіх сценах відомого анімаційного фільму «Історія іграшок», який став першим повністю анімованим фільмом, створеним за 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

У 1980-х роках розпочалася активна модернізація та комерціалізація комп’ютерної графіки. Одним із ключових факторів стало широке поширення комп’ютерних систем для домашнього використання, що зробило технології комп’ютерної графіки доступними для більшої кількості користувачів. Зростання популярності персональних комп’ютерів відкривало нові можливості для роботи з графікою, дозволяючи аматорам і професіоналам створювати зображення.

Прогрес у розробці комп’ютерних технологій призвів до появи нових 16-бітових процесорів та перших графічних процесорів, що значно прискорило процеси рендерингу і сприяло створенню зображень більшої якості. Зниження

вартості деяких ключових компонентів комп'ютерів, зокрема оперативної та відеопам'яті, також зробило роботу в цій галузі більш доступною.

Окрім того, комп'ютерна графіка почала активно використовуватися у кінематографі. Фільми, такі як *Зоряні війни*, стали піонерами в інтеграції комп'ютерної графіки в розважальну індустрію, що значно підвищило популярність цієї технології серед широкої аудиторії.

У сфері створення реалістичних зображень важливим етапом стало розроблення загальної формули рендерингу зображень, запропонованої Давидом Іммелем і Джеймсом Каджі. Це відкриття стало значущим кроком до імплементації глобального освітлення в комп'ютерній графіці, що дозволило наблизити візуальні ефекти до рівня реалістичності, а також значно поліпшило якість зображень у кінематографічних і комп'ютерних анімаціях.

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.

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

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

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

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

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

1.8.4. 1990-ті роки

З початком 1990-х років значно зросла популярність створення тривимірних моделей, що раніше були доступні лише для дорогих систем вартістю десятки тисяч доларів. Однак, з розвитком технологій, ця можливість стала доступною і для звичайних персональних комп'ютерів.

У цей період згенеровані комп'ютерні зображення вперше почали наближатися до рівня фотореалізму. Для звичайного користувача робота комп'ютерних систем стала нагадувати відображення реального світу, що значно покращило сприйняття візуальних ефектів. Згодом комп'ютерна графіка знайшла своє місце в анімації, мультимедіа та відеоіграх, ставши важливим інструментом в цих сферах.

Важливим етапом у розвитку комп'ютерної графіки став випуск у 1995 році першого повністю комп'ютерно-анімованого фільму *Історія іграшок* студії Pixar. Цей фільм став не лише надзвичайно популярним, але й значно посприяв популяризації комп'ютерної графіки в анімаційній індустрії.

Зростання потужності комп'ютерних систем дозволило досягти ще вищої якості зображень. Приріст швидкодії процесорів не лише прискорив процеси рендерингу, а й надав змогу застосовувати більш складні обчислювальні алгоритми, що значно наблизило комп'ютерну графіку до рівня фотореалізму.

1.8.5. Сучасний стан комп'ютерної графіки

На сьогоднішній день, починаючи з 2010-х років, традиційна комп'ютерна графіка досягла майже повного фотореалізму. Формули обчислень значно наблизилися до реальних фізичних моделей, що дозволяє створювати зображення з високим рівнем правдоподібності.

Комп’ютерна графіка набула широкого застосування в численних галузях. Вона стала невід’ємною частиною майже всіх кінематографічних проектів, а також домінує в сучасних анімаційних фільмах. Крім того, комп’ютерна графіка активно використовується в рекламі, телебаченні та інших сферах діяльності.

Процес створення комп’ютерної графіки став значно доступнішим порівняно з минулими десятиліттями. Сучасні персональні комп’ютери здатні генерувати фотореалістичні зображення, що ще кілька років тому вимагали потужних суперкомп’ютерів для реалізації.

2. Комп’ютерна графіка в реальному часі

Комп'ютерна графіка в реальному часі — це галузь комп'ютерної графіки, яка орієнтована на створення зображень з високою швидкістю при забезпеченні інтерактивності[23]. Хоча графічні інтерфейси користувача також відносяться до цієї категорії, справжньою комп'ютерною графікою в реальному часі є, переважно, тривимірна інтерактивна графіка, яка для своєї обробки зазвичай використовує графічний процесор.

Фундаментально графіка в реальному часі визначається за наявністю інтерактивності та відповідним рівнем швидкодії. Швидкодія вимірюється кількістю кадрів, що генеруються програмою за секунду, або ж у термінах кадрів на секунду (FPS). У кінематографії стандартна швидкість складає 24 кадри на секунду. Для імітації руху достатньо 15 кадрів на секунду, проте частота кадрів нижча за 24 може викликати дискомфорт у більшості людей. В контексті реального часу мінімальною прийнятною швидкістю є 30 кадрів на секунду. Цей стандарт виник через обмеження старих телевізорів, де частота оновлення екрану була обмежена. Вища частота оновлення забезпечує менш помітні зміни в зображенні та більш комфортну взаємодію користувача з програмою. Однак, максимальна частота оновлення може бути обмежена апаратними можливостями дисплея. Таким чином, якщо програма генерує 30 кадрів на секунду або більше і володіє інтерактивністю, вона належить до графіки реального часу.

Окрім інтерактивності, важливим аспектом є висока швидкість обчислень, що потребує спрощення математичних моделей і алгоритмів, через що якість фінального зображення в графіці реального часу зазвичай є нижчою, порівняно з традиційною комп’ютерною графікою. Перехід від обчислень, що займають години на суперкомп'ютері, до виконання в межах 10-30 мілісекунд на звичайному

комп'ютері вимагає значних змін у самих алгоритмах.

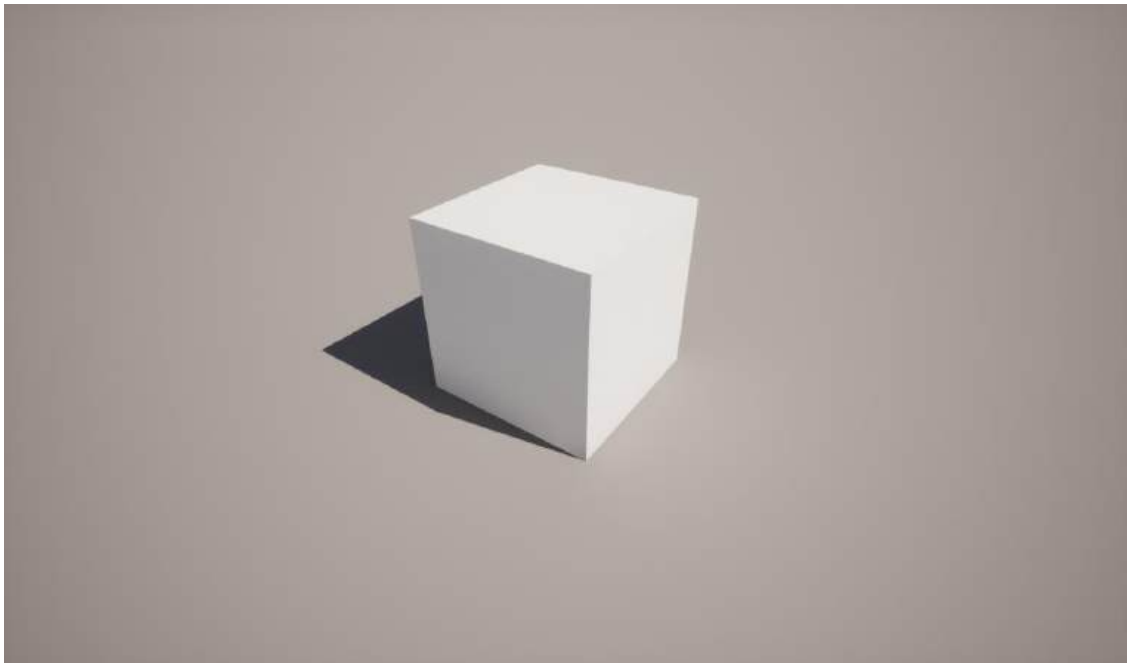


Рисунок 2.1 – Приклад комп’ютерної графіки в реальному часі.

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

Однією з ключових переваг графіки реального часу є її висока швидкість обробки зображень. У традиційному процесі створення фільмів кожен кадр може вимагати годинних обчислень, а для створення кількох секунд кінопродукту потрібні дні[32]. Проте для перевірки результатів роботи команди необхідний перегляд готового матеріалу, що також вимагає багато часу. На відміну від цього, графіка реального часу дозволяє отримати результат без затримок, що значно прискорює весь процес створення зображень та взаємодії з ними.

Графіка реального часу відрізняється від пре-рендерингу тим, що для отримання високоякісних зображень не обов’язково використовувати суперкомп’ютери — достатньо звичайного персонального комп’ютера. Цей низький поріг входження робить комп’ютерну графіку більш доступною для широкого кола користувачів, що відкриває нові можливості для самовираження та творчості в цій галузі.

Навіть для великих корпорацій графіка реального часу дозволяє досягти достатньо високої якості зображень, що можуть конкурувати з пре-рендерингом, але при цьому зберігають високу швидкість обробки. У ситуаціях, коли

максимальна якість зображення не є критичною, перевагу має використання графіки реального часу, оскільки вона є швидшою та економічнішою.

Не менш важливою є інтерактивність, яку забезпечує графіка реального часу. Висока швидкість обчислень дозволяє швидко реагувати на дії користувача, що відкриває можливості для застосування цієї технології в різних галузях. Наприклад, у системах додаткової реальності, де графіка виводить додаткову інформацію про навколишній світ безпосередньо в окуляри користувача, що особливо корисно для інженерів. Віртуальна реальність, що використовує графіку реального часу, знайшла застосування в освіті, зокрема для військових тренувань, навчання медичних працівників, пілотів та в автомобільній індустрії для симуляції поведінки автомобіля на дорозі.

2.2. Загальні принципи роботи

Програма, розроблена для генерації графіки реального часу, насамперед працює з даними, такими як тривимірні моделі та текстури, залежно від цілей і вимог конкретного застосування [23]. Алгоритм генерації зображень визначається особливостями програми та вибором розробників. Кінцевим результатом роботи програми є створення зображень, а також забезпечення реакції на взаємодію з користувачем.

На відміну від традиційної комп’ютерної графіки, графіка реального часу повинна генерувати зображення із забезпеченням певної швидкості обробки. Це вимагає оптимізації обчислювальних процесів для досягнення максимальної швидкості, що може призводити до певного зниження якості зображення, але дозволяє значно прискорити реакцію програми.

Для графіки реального часу критичними є високі вимоги до швидкодії та інтерактивності. Хоча алгоритми та формули, використовувані для рендерингу, відрізняються від тих, що застосовуються в пре-рендерингу, вони не визначають кардинальних меж між цими двома видами графіки. В перспективі, коли можливість застосування алгоритмів пре-рендерингу для графіки реального часу стане реальністю, і за умови збереження інтерактивності, така графіка буде вважатися графікою реального часу.

2.3. Растрова графіка

Протягом значної частини історії комп'ютерної графіки алгоритми рендерингу були обмежені можливостями обчислювальних систем, що призводило до використання простіших методів, зокрема методу растеризації. У порівнянні з трасуванням шляху, яке традиційно застосовується в комп'ютерній графіці і яке з недавніх пір стало доступним і для графіки реального часу, растеризація є значно швидшим процесом.

Растеризація є методом перетворення тривимірних об'єктів у двовимірне растрове зображення, яке складається з пікселів і відображається на екрані користувача [31]. Для ефективності цього процесу використовується буфер глибини, який зберігає дані про глибину об'єктів (їх координати Z) з певної перспективи [31]. Буфер глибини призначений для спрощення обчислень, що дозволяє програмі ігнорувати об'єкти, що знаходяться за іншими непрозорими об'єктами, таким чином знижуючи навантаження на систему. Буфер глибини є двовимірним масивом, який для кожного пікселя зберігає не колір, а його глибину, що дозволяє визначати порядок об'єктів на сцені. Без цього елементу, відсутність Z-буфера ускладнює або унеможливлює правильне визначення просторового розташування об'єктів і їх взаємодії в растровій графіці.

Процес растеризації полягає у визначенні для кожного пікселя зображення найближчого об'єкта з урахуванням глибини, використовуючи Z-буфер [31]. Для кожного пікселя обчислюється його колір, де поверхня вважається освітленою, якщо на неї потрапляє світло, і темною, якщо ні.

Растеризація зосереджена на роботі з екранним простором, тобто тим, що видно на зображенні в конкретний момент часу. У рамках цього методу все, що знаходиться поза межами «камери», наприклад, за нею, не враховується в процесі рендерингу.

Аналізуючи принципи роботи растрової графіки, можна відзначити її основні недоліки. У порівнянні з пре-рендерингом, якість зображення, отриманого методом растеризації, є значно нижчою, а функціональність цієї графіки є примітивною. Хоча для маскування цих недоліків було розроблено різні техніки та методи, самі обмеження растрового методу залишаються.

2.4. Трасування променів в реальному часі

Завдяки інноваціям у сфері графічних процесорів та відеокарт, стало можливим реалізувати трасування променів у реальному часі. На відміну від традиційних методів комп’ютерної графіки реального часу, які використовують спрощені алгоритми обчислень, трасування променів застосовує модифіковані алгоритми традиційного рендерингу [30]. У методах пре-рендерингу можливо використовувати велику кількість променів для кожного пікселя та численні відбиття променів від поверхонь. Однак у трасуванні променів в реальному часі застосовується один промінь на піксель, який відбивається лише один раз від поверхні. Незважаючи на ці спрощення, трасування променів залишається складним і вимагає потужних обчислювальних ресурсів, зокрема найкращих доступних персональних комп’ютерів [38].

Цей метод також усуває необхідність використання буфера глибини [31]. Як і в традиційній графіці, з кожного пікселя випускається промінь, при цьому не потрібно окремо зберігати інформацію про розташування об'єктів у сцені. У порівнянні з растеризацією, де алгоритм шукає найближчий об'єкт для кожного пікселя, трасування променів здійснюється шляхом визначення, який об'єкт найближчий до кожного пікселя через випромінювання променя з цього пікселя [31].

Одним з методів оптимізації трасування променів є обмеження простору дії променів [34]. У тривимірному просторі визначаються обмежені сфери, що зменшують максимальну відстань, на яку може поширюватися промінь, що дозволяє зекономити обчислювальні ресурси, запобігаючи надмірним обчисленням. Важливо зазначити, що взаємодія з віртуальним небом, як глобальним джерелом освітлення сцени, відбувається за допомогою рівномірного освітлення поверхонь, що падає під певним кутом і з визначеною потужністю в люменах. Люмен (lm), на відміну від одиниць вимірювання освітлення, таких як кандела (cd) або люкс (lx), має рівномірну потужність, незалежно від площі освітленої поверхні чи відстані до джерела. У випадку віртуального Сонця відсутня конкретна відстань до джерела світла; воно має напрямок і потужність у люменах, освітлюючи весь простір в межах певного кута без чіткого

розташування. Це створює ефект реалістичного освітлення, хоча й із значними спрощеннями у порівнянні з фізичними процесами, коли світло може відбиватися багаторазово. У трасуванні променів, проте, відбувається лише одне відбиття, що робить цей метод придатним для обчислень у реальному часі, на відміну від растрової графіки.

З точки зору апаратного забезпечення, метод трасування променів вимагає наявності відеокарти з підтримкою цієї технології. Сучасні відеокарти оснащені спеціальними ядрами для обробки трасування променів, і без таких ядер технологія не може бути реалізована в реальному часі [38]. Ці ядра призначені виключно для трасування променів і не можуть використовуватися для інших обчислювальних задач.

2.5. Трасування напрямку

Трасування напрямку є методикою рендерингу, яка активно використовується в традиційній комп'ютерній графіці та тепер доступна й у графіці реального часу [39]. Це метод, що можна вважати вдосконаленою та більш складною версією трасування променів. У трасуванні напрямку здійснюється кілька відбиттів променів для кожного пікселя, що є основною відмінністю від трасування променів, де кожному пікселю відповідає лише одне відбиття променя, після чого відстежується його шлях до джерела освітлення. Фінальний результат цього процесу визначає колір пікселя. Трасування напрямку використовує схожий підхід, але з кількома відбиттями променя – від одного до чотирьох. Завдяки таким удосконаленням цей метод дозволяє отримати зображення, яке максимально наближене до реалістичного. Однак очевидним недоліком є неймовірна складність обчислень, що вимагає використання найпотужнішого і найдорожчого доступного апаратного забезпечення.

Оскільки трасування напрямку є поліпшеною версією трасування променів, його апаратні вимоги є подібними до вимог для трасування променів [38]. Для реалізації цієї технології необхідні спеціалізовані компоненти відеокарт, розроблені саме для цієї мети, що робить її застосування неможливим без відповідного апаратного забезпечення.

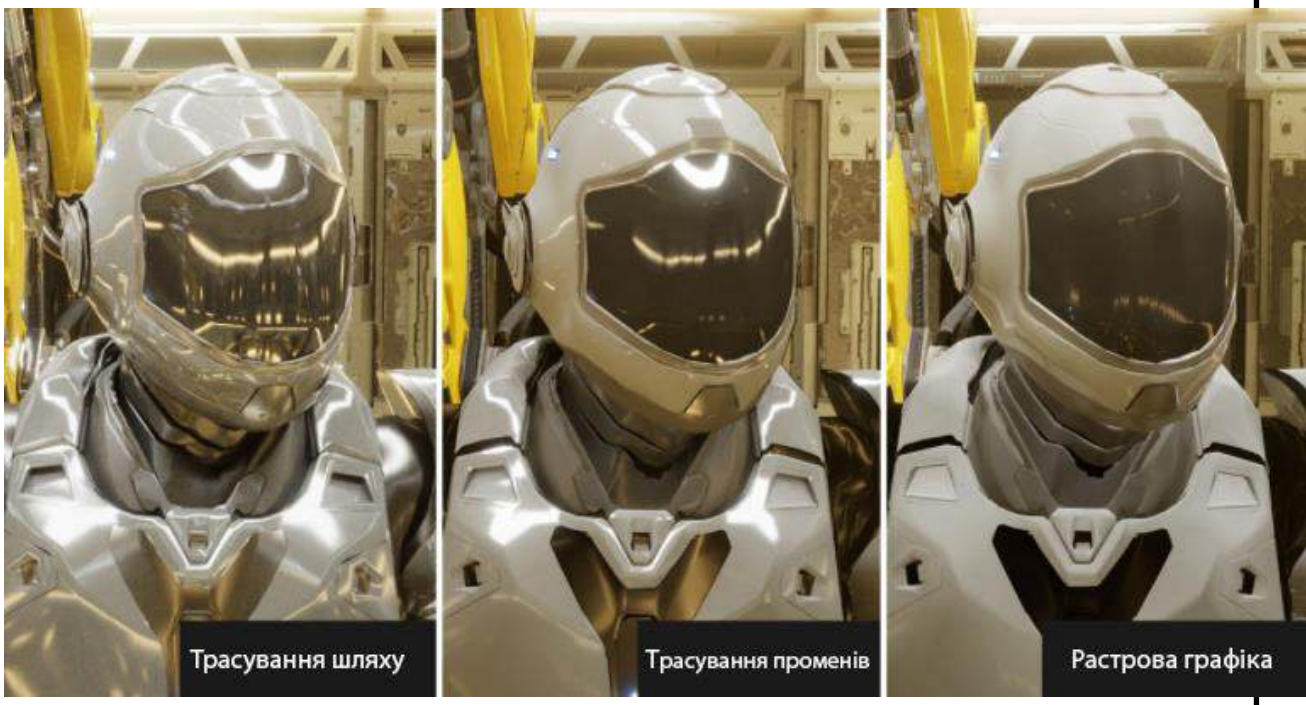


Рисунок 2.2 – Порівняння різних методів роботи світла.

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. Необхідність 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 основних алгоритмів обчислення.

2.6.1. Тіні

Однією з найбільш очевидних проблем графіки реального часу є відтворення тіней. У растровій графіці рівень освітленості поверхні зазвичай визначається за двома станами: наявність тіні або її відсутність [30]. Це пов’язано з тим, що в растровій графіці реального часу світло не враховує відбивання, тобто кількість відбиттів дорівнює нулю. Колір кожного пікселя визначається окремо: поверхня вважається освітленою, якщо на неї потрапляє світло, і темною, якщо ні. Для цього між джерелом світла і поверхнею проводиться пряма лінія; якщо вона не перетинає перешкод, поверхня отримує освітлення.

Подібний підхід, хоча і модифікований, застосовувався у ранніх алгоритмах статичної графіки. Основний недолік такого методу полягає в різкому розподілі зон на однаково освітлені та однаково затемнені області. Такі тіні називають

твердими. Вони є менш ресурсоємними для обчислення, але виглядають нереалістично. У реальному світі тверді тіні трапляються рідко, однак їх можна спостерігати в космосі, наприклад, астронавтами, які перебувають на відносно невеликій відстані від Сонця.

Для вирішення цієї проблеми почали застосовувати методи імітації освітлення від неба [16]. На відміну від реального світу, цей підхід накладає на всі поверхні певний рівень кольору з заданою яскравістю. Зазвичай використовується синій відтінок, що нагадує колір неба. Хоча цей метод здається досить простим, оскільки він рівномірно додає колір на всі поверхні, він значно покращує візуальну якість зображення.

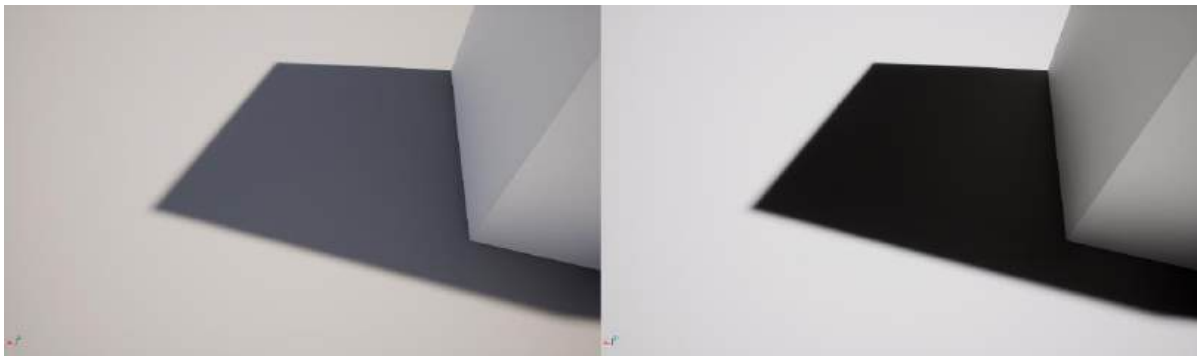


Рисунок 2.3 – Демонстрація накладання рвіномірного кольору на поверхні. Без накладання(справа) і з накладанням(зліва).

Накладання кольору на затемнені області дозволяє зробити їх світлішими та частково приховати певні візуальні недоліки, однак цей підхід не усуває жорсткість тіней і не додає їм реалістичності. Межа між освітленою та затемненою поверхнею залишається різкою, без плавного переходу, що робить зображення штучним.

Одним із методів пом’якшення жорсткості тіней є додавання плавних переходів на краях тіней [1]. Замість одноманітного затемнення краї тіней отримують градієнт, колір і розмір якого визначаються характеристиками джерела світла, такими як його інтенсивність і розміри. Це імітує реальну поведінку світла, хоча конкретні математичні моделі залежать від розробника. Попри покращення візуальної якості, такий метод є більш ресурсоємним.

Наступний метод для усунення основних проблем з тінями це є їх запікання. Запікання тіней є ще одним популярним підходом до усунення недоліків жорстких тіней. У цьому випадку тіні обчислюються заздалегідь для конкретного джерела освітлення з використанням алгоритмів попереднього рендерингу, після чого результати зберігаються у вигляді текстур [20]. Оскільки тінь можна виразити як зображення, її можна заздалегідь прорахувати та просто накладати на поверхню під час виконання програми.

Перевагами цього методу є низькі витрати на обчислення в реальному часі та можливість досягнення високої якості тіней [10]. Однак процес запікання є доволі тривалим і збільшує використання відеопам’яті, хоча і незначною мірою. Крім того, для реалізації методу об’єкти повинні мати спеціальні шари даних для текстур тіней, що ускладнює роботу з об’єктами нестандартної форми. Основний недолік запечених тіней полягає у відсутності динамічності: такі тіні не можуть змінюватися в реальному часі, що обмежує інтерактивність. Будь-які рухомі або трансформовані об’єкти не можуть мати запечені тіні [17]. Через ці обмеження метод запікання тіней дедалі рідше використовується, особливо у середовищах із високими вимогами до інтерактивності.

Наступний спосіб вирішення проблеми це трасування променів. Трасування променів є одним із найбільш прогресивних методів усунення проблем із тінями. Цей алгоритм працює в реальному часі та забезпечує значно вищу якість, ніж запечені тіні. Трасування променів імітує поведінку світла у реальному світі, дозволяючи отримати реалістичні переходи тіней та деталізацію. Недоліком цього підходу є висока обчислювальна складність, навіть якщо метод обмежується лише генерацією тіней [10]. Використання трасування променів вимагає спеціалізованого апаратного забезпечення, але цей метод забезпечує високоякісний фінальний результат і є важливим кроком у розвитку технологій обчислювальної графіки. Ще більш точним, хоча й складнішим, є метод трасування шляху, який є розширенням трасування променів. Завдяки моделюванню численних відбиттів світла цей метод створює зображення найвищої якості, максимально наближені до реального світу. Водночас, він є найвимогливішим до апаратних ресурсів і потребує значної обчислювальної

потужності. З розвитком технологій трасування шляху має потенціал для подальшого покращення продуктивності та візуальної якості.

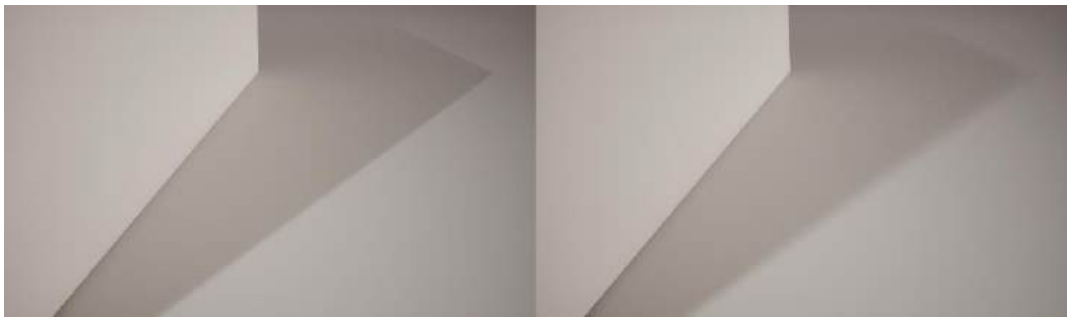


Рисунок 2.4 – Растрова тінь(зліва) і тінь від трасування променів(зправа).

2.6.2. Відображення

Обмеження стандартної растрової графіки в реальному часі, зокрема відсутність відбивань світла, унеможливлюють створення відображень [1]. Відображення виникають як результат відбивання променів світла від поверхонь. Тому за відсутності відбивань світла неможливі і самі відображення. Для подолання цього недоліку існують кілька підходів, кожен із яких має свої переваги та недоліки.

Перший метод по принципу роботи подібний до запікання тіней, але працює з відображеннями. За принципом своєї роботи метод доволі зрозумілий, з певної точки в тривимірному просторі створюються шість зображень, шість як сторін куба, тобто по одному на кожну сторону, з метою покриття в такий спосіб простору певного розміру [13]. Кожне з шести зображень це текстура певного розміру, а всі шість з них потім створюють куб чи, що також доволі часто, сферу, яка вже буде використовуватися для відображень у певній області. Звісно реалізація може відрізнятися у різних розробників, але основні принцип роботи залишається однаковим. Фінальний результат, а саме шість зображень, перетворених в куб чи сферу використовують для проекції на об’єкти в тривимірному просторі, поверхня яких дає відображення. Важлива перевага подібного методу це той факт, що він не використовує обрахунків під час безпосередньої роботи програми, проте, в залежності від заданих розмірів і кількості проведених запікань, він може вимагати для роботи великої кількості відеопам’яті [10]. З недоліків звісно подібний метод, як і в випадку з тінями, не

здатен працювати з динамічними об’єктами, а також за принципом своєї роботи він не спроможній на реалістичний результат. Запечені відображення майже ніколи не є коректними, вони тільки приблизні до того, що було би в реальності, крім цього об’єкт не може мати відображення самого себе в собі, що не є реалістичним [13].

Наступний підхід дозволяє розв'язати проблему динамічності, однак додає низку недоліків і обмежень — це метод відображень, заснованих на екранному просторі [15]. Цей метод функціонує у межах екранного простору. Використовуючи попередньо згенероване рендер-зображення сцени в реальному часі, відображення накладаються на відповідні поверхні [5]. Таким чином, метод здатний значною мірою створювати ілюзію реальних відображень, проте аналіз принципів його роботи дозволяє виявити певні недоліки.

Метод базується на опрацюванні фінального результату рендеру, тобто простору, що відображається на екрані. Його функціонування залежить від попереднього прорахунку пікселів, які повинні бути включені у відображення. Якщо ж певні елементи сцени виходять за межі екранного простору, вони не відображатимуться. У певних випадках така обмеженість може бути вкрай помітною. Наприклад, об’єкти, які фізично знаходяться поза межами видимого екрану, не відображатимуться на відбиваючих поверхнях.

Принцип роботи передбачає визначення розташування та напряму відображення для кожного пікселя. Використовуючи дані Z-буфера, визначається положення точки відображення у тривимірному просторі, а нормалі поверхонь дозволяють обчислити напрямок цього відображення [5]. Всі ці обчислення виконуються в межах екранного простору. Далі, за допомогою буфера глибини, перетини напряму відображення з геометрією сцени ідентифікують 2D-координати пікселів, які відповідають відображенню, а також їхні кольорові характеристики. Крім того, перед фінальним накладанням відображення може бути застосована додаткова обробка, що вплине на його якість.

З точки зору обчислювальної складності, метод відображень екранного простору є досить ресурсоємним, особливо в порівнянні з мінімальними обчислювальними витратами, властивими методу запікання текстур [15].

Наступний підхід, планарні відображення, є досить простим і прямолінійним рішенням для створення відображень. З концептуальної точки зору, відображення можна трактувати як відтворення існуючого середовища. Таким чином, якщо виконати обчислення всіх об'єктів сцени з урахуванням певної поверхні, тобто, умовно кажучи, створити її дзеркальну копію, це виглядатиме як відображення [11]. Саме за таким принципом функціонують планарні відображення.

Цей метод працює шляхом накладання відображень на двовимірну плоску поверхню, що водночас є його ключовим обмеженням, окрім високої складності обчислень. Він передбачає обрахунки всієї сцени з точки зору заданої поверхні, незалежно від того, чи об’єкти реально потрапляють у відображення, що може значно ускладнити процес розрахунків [11].

Серед основних недоліків методу є його низька ефективність. Обчислення сцени двічі для створення одного відображення не є оптимальним підходом, а у випадку, коли відображень кілька, цей процес необхідно повторювати для кожного з них. Щоб частково вирішити цю проблему, можна вручну визначати, які об’єкти мають враховуватися у відображеннях, а які — ні. Це дозволяє зменшити кількість обчислюваних об'єктів, але сам процес відбору може бути дуже тривалим, особливо у складних сценах з сотнями чи тисячами об'єктів. Крім того, якщо якийсь об'єкт не буде включено у список для обрахунку відображень, він не з’явиться у відображенні.

Обрахунок відображень для всієї сцени передбачає її повне дублювання, незалежно від того, чи всі об'єкти потрапляють у "дзеркальну" площину. Наприклад, дзеркало в кімнаті може намагатися врахувати не лише інтер'єр приміщення, але й зовнішнє середовище за його межами, що суттєво знижує ефективність.

Попри зазначені обмеження, цей метод має своє застосування. Він є першим серед розглянутих, який дозволяє створювати точні, динамічні відображення, що є його ключовою перевагою.

Останній підхід реалізується за допомогою методу трасування променів. Цей метод забезпечує найвищу якість відображень серед усіх розглянутих, хоча й характеризується найбільшою складністю обчислень. Водночас, у залежності від

складності сцени, обчислення можуть бути менш трудомісткими порівняно з планарними відображеннями. Основним недоліком методу є обмеження на кількість відбивань променів світла [21].

Ці обмеження унеможливлюють коректний розрахунок відображень дзеркальних поверхонь у інших дзеркалах. У таких випадках візуалізація обмежується контурами предметів у вигляді чорних областей, які виникають через неможливість подальших обчислень. Як вирішення, процес трасування може бути зупинений, а чорний колір замінений на заздалегідь визначений, щоб імітувати правильну роботу відображень [22].

Як альтернативу трасуванню променів можна використовувати метод трасування шляху. Він забезпечує ще вищу якість зображення, проте значно збільшує обчислювальну складність.

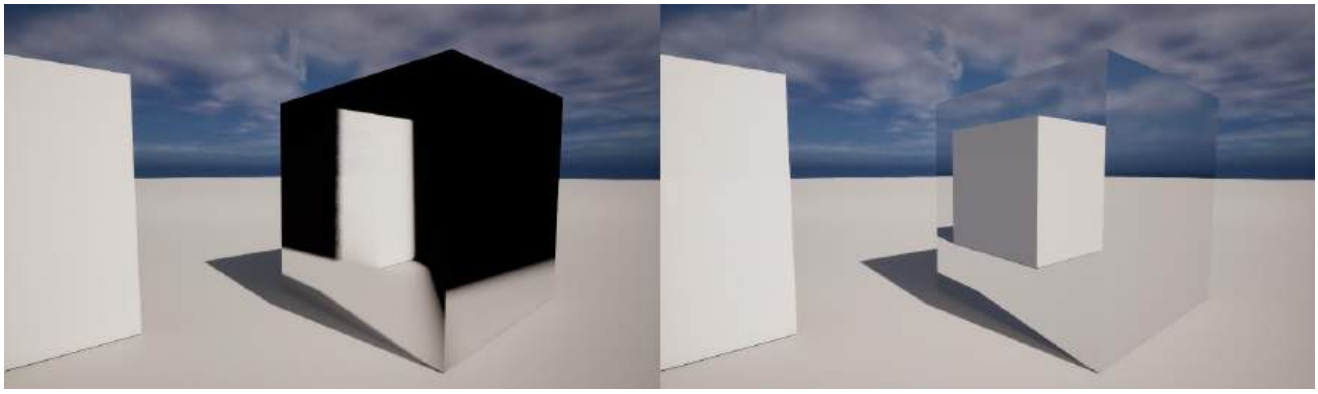


Рисунок 2.5 – Порівняння двох відображень, екранного простору (зліва) і трасування променів (справа)

2.7. Оптимізація комп’ютерної графіки в реальному часі

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

2.7.1 MIP-текстурування (MIP mapping)

У комп’ютерній графіці використовується метод міпмапінгу, або MIP-текстурування, де MIP є скороченням від латинського “multum in parvo”, що означає "багато в одному". Цей метод базується на послідовності оптимізованих і попередньо обчислених зображень, кожне з яких є меншою за розміром репрезентацією попереднього зображення [18].

Основна мета використання міпмапінгу — спрощення обчислень та зменшення графічних артефактів, які виникають через помилки, пов’язані із згладжуванням. У такій послідовності високоякісне зображення (міпмап) використовується для областей поблизу камери, де висока концентрація пікселів, тоді як зображення меншого розширення застосовується для віддалених областей. Такий підхід є більш ефективним з точки зору обчислень, ніж детальне опрацювання кожного пікселя текстури для визначення його впливу на пікселі екрану.

Методологічно міпмапінг реалізується як набір зображень, кожне з яких має зменшений розмір порівняно з попереднім. Процес створення міпмапів відбувається автоматично на програмному рівні: зображення високої роздільної здатності використовується для генерування послідовності менших зображень, аж до досягнення мінімального розміру — 2×2 пікселів. Однак для забезпечення рівномірного зменшення зображення його ширина та висота повинні бути степенем двійки (2^n) [18]. Наприклад, розміри можуть бути 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096 тощо. Кожен наступний міпмап має ширину та висоту, зменшені у два рази порівняно з попереднім. При цьому початкове зображення може бути не квадратним (наприклад, 1024×512), хоча квадратні текстури є більш зручними для використання. Якщо текстура відповідає цим вимогам, кожне наступне зображення буде вчетверо меншим за попереднє.

Використання міпмапів потребує додаткової відеопам’яті, оскільки кожна текстура займає приблизно на 33% більше пам’яті. Проте це виправдано через значний приріст швидкості обчислень і зменшення артефактів згладжування. Залежно від реалізації, у пам’яті можуть одночасно зберігатися всі можливі міпмапи або лише версії текстур максимальної якості та однієї нижчої. Наприклад, якщо текстура має розмір 4096×4096 і використовується у своєму максимальному розширенні, у пам’яті буде одночасно зберігатися текстура цього розміру та наступного нижчого — 2048×2048. Якщо ж текстура використовується у мінімальному розмірі (2×2), то в пам’яті також будуть усі текстури більшої роздільної здатності.

Цей метод активно використовується в комп’ютерній графіці, зокрема в задачах реального часу, і є важливим елементом сучасних графічних технологій.

Рисунок 2.6 – Приклад MIP-текстурування.

2.7.2. LOD

LOD (Level of Detail) у перекладі з англійської означає "рівень деталізації". Це підхід у комп’ютерній графіці, що дозволяє адаптувати складність тривимірної моделі залежно від відстані до камери [1].

Принцип роботи LOD полягає у заміні детальної моделі на менш складну, коли вона віддаляється від камери настільки, що подібна заміна стає непомітною. Моделі зі зниженою деталізацією (тобто з меншою кількістю трикутників) створюються разом із основними моделями за допомогою спеціалізованих програм. У цих програмах моделі збираються у набори LOD. Заміна моделі залежить від відстані до камери, а в сучасних реалізаціях — від відсотка площі екрана, яку займає модель. У програмному коді це зазвичай реалізовано як масив, де індекси відповідають рівням LOD, зазвичай у кількості від 0 до 3 або 4. Цей процес можна порівняти з міпмапами для текстур, хоча він є менш автоматизованим.

Дані LOD завантажуються в пам’ять системи та відображаються на екрані за потребою. У залежності від реалізації, всі рівні деталізації можуть зберігатися в пам’яті одночасно, або ж лише ті моделі, які використовуються в конкретний момент [1].

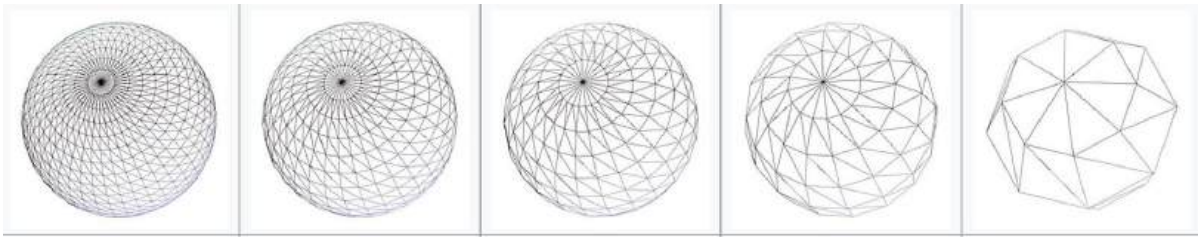


Рисунок 2.7 - Приклад LOD.

3. Реалізація програмного забезпечення комп’ютерної графіки реального часу

Основною метою дипломної роботи - це створення власного програмного забезпечення, що працює з комп’ютерною графікою в реальному часі. Ця програма є практичною реалізацією описаної в дипломній роботі теорії комп’ютерної графіки, та демонструє засвоєння матеріалу і проведене дослідження галузі комп’ютерної графіки.

3.1 Призначення програмного забезпечення

Створення програмного забезпечення подібної складності не тільки свідчить про засвоєння матеріалу та проведене дослідження в галузі комп’ютерної графіки, але й забезпечує можливість демонстрації описаних у роботі алгоритмів. Така програма може мати значну навчальну цінність. Використання інтерактивного підходу для пояснення складного матеріалу дозволяє значно покращити процес засвоєння знань, надаючи користувачам можливість глибше вивчати матеріал у більш доступній та динамічній формі, порівняно з традиційним підходом через підручники.

Основною метою програми є перетворення складної інформації в більш зрозумілу та захоплюючу форму для більш широкої аудиторії. Хоча зміст програми відображає основні аспекти, викладені в дипломній роботі, вона має більший акцент на заохочення користувача до активного навчання. Додавання елементів інтерактивності дозволяє зберігати увагу користувача на складних формулах і алгоритмах, що є значно більш ефективним, ніж просте сприйняття тексту на папері.

Програма надає користувачам можливість безпосередньо взаємодіяти з елементами комп’ютерної графіки, що ілюструють описані алгоритми на практиці. Мотивація є ключовим фактором у навчальному процесі, і зниження інтересу до навчальної діяльності може значно знизити ефективність засвоєння матеріалу. Завдяки інтерактивній природі програми, користувач заохочується до більш активного вивчення і засвоєння матеріалу через демонстрацію графічних елементів з подальшою перевіркою рівня засвоєння. Такий підхід, незважаючи на

технічну складність реалізації, забезпечує набагато вищий рівень засвоєння матеріалу.

Інформація, представлена в програмі, є стислим і лаконічним викладом змісту дипломної роботи, включаючи опис формул і алгоритмів, але в інтерактивному форматі, який забезпечує більш зручний доступ до даних. Для забезпечення максимальної доступності було обрано використання спрощених формул та технологій. Використання передових технологій могло б значно підвищити вимоги до обчислювальних потужностей, що обмежило б кількість користувачів, здатних запустити програму. Завдяки цьому рішенню більше користувачів може ознайомитися з основами комп'ютерної графіки та здобути нові знання.

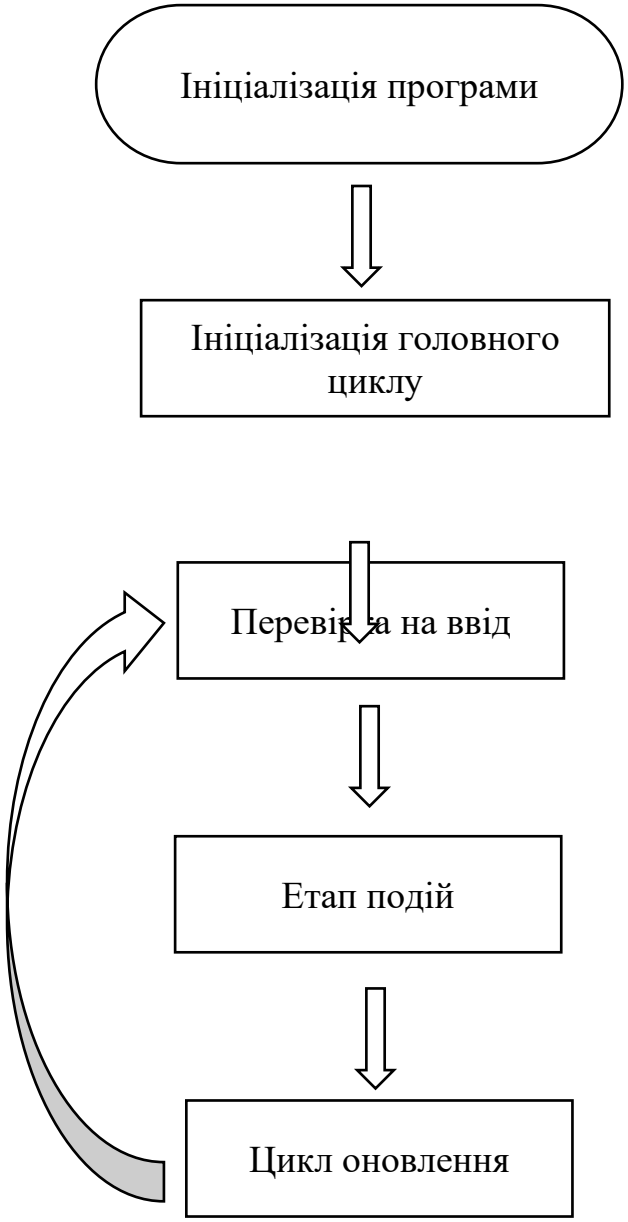
3.2. Основа програмного забезпечення

Програмне забезпечення працює за аналогією з іншими програмами, однак орієнтоване на роботу з комп'ютерною графікою. Запуск програми здійснюється стандартним чином через файл формату .exe, що знаходиться в папці програми. Ця папка містить всі необхідні для функціонування програми файли та дані.

3.3. Принципи роботи програмного забезпечення

В основі програми використовується рушій Unreal Engine 5, що є фреймворком і редактором для створення програм подібного типу. В момент запуску програми, рушій проводить ініціалізацію необхідних для запуску компонентів. Це в свою чергу завантаження різних модулів, які необхідні для функціонування програми, завантаження коду, що дозволяє програмі працювати на конкретній платформі, а також завантаження параметрів налаштування користувача. Також в момент ініціалізації створюються окремі потоки на програму і на її шейдери. Після початкового етапу ініціалізації створюється основний цикл програми, він буде виконуватись, поки програма не буде закрита користувачем. Основний цикл програми виконується один раз перед створенням кожного кадру. Весь код виконується в межах цього циклу. В основному циклі виконується базовий код, він необхідний для роботи програми, також проходить перевірка на ввід користувача і виконується наступний етап - етап подій. Подія (Event) – це в

цьому конкретному рушії виклик функцій, класів, тощо за певних умов. Наприклад, якщо, користувач захотів закрити програму і натиснув відповідну кнопку, буде викликана функція завершення роботи програми. З метою оптимізації цей код не виконується для кожного кадру, а тільки коли є на це команда, наявність якої перевіряє етап подій. Крім основного циклу, існує цикл update, він же цикл оновлення. В цикл update входить весь код, що повинен виконуватися кожен кадр. Не складно здогадатися, що зловживання циклом update може легко призвести до надмірного навантаження і неефективного використання ресурсів системи. Перш ніж буде завершено 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.

3.4. Робота програми з апаратним забезпеченням

Операційна система в значній мірі відповідає за взаємодію програми і комп'ютера. Код програми тісно працює з операційною системою, в цьому випадку це Microsoft Windows. Мінімальна необхідна для роботи версія - Windows 10, неповноцінна емуляція можлива на Windows 7-8, проте значна частина функціоналу DirectX 12 буде відсутня. Програма виконує виклики до системи для надання ресурсів і остаточного виведення зображення через API системи. Сама платформа Windows використовує для роботи програми і апаратного забезпечення свій набір драйверів DirectX, який є власністю компанії Microsoft і функціонує тільки на їхніх платформах.

З метою отримання доступу до відеокарти і обчислення комп'ютерної графіки програма використовує API платформи Windows. Далі платформа Windows використовуючи DirectX починає роботу з відеокартою. В свою чергу відеокарта починає виконання графічного коду програми.

Взаємодія програми з іншим апаратним забезпеченням відбувається в подібний до цього спосіб. Проте варто сказати, що основна робота виконується операційною системою і її драйверами.

Для своєї роботи програма використовує DirectX 12 – власність компанії Microsoft, потребує версію Windows 10 і вище для повноцінної роботи. За умови відсутності цих драйверів або, якщо, використовується застаріла версія програмне забезпечення не буде запущено.

3.5. Функціонал програми

Крім мінімальної основи, для повноцінної роботи програма мусить мати певний додатковий функціонал. Було створено код, що надає програмі різноманітних додаткових можливостей.

3.5.1. Підтримка додаткових пристроїв введення даних

Для взаємодії з програмою користувачем зазвичай використовуються стандартні для персональних комп'ютерів пристрої – комп'ютерна миша і

клавіатура. Звісно подібний функціонал є достатнім для реалізації взаємодії користувача з програмою, але задля забезпечення різноманітності способів керування програмою і з урахуванням зростаючої популярності альтернативних засобів управління і способів вводу інформації у програмному середовищі комп'ютера було прийнято рішення додати підтримку додаткових пристроїв введення.

У програму було інтегровано підтримку XInput API, яке є частиною архітектури операційної системи Microsoft Windows [27]. XInput API дозволяє програмам отримувати дані від контролерів, сумісних з цією технологією, найпоширенішими з яких є контролери серії Xbox. Варто зазначити, що XInput API доступний не лише на платформі Windows, а й на інших операційних системах, таких як macOS, Android, Linux та iOS. На версіях Windows 7 і новіших XInput API вже є встановленим за замовчуванням [26]. Така інтеграція дозволяє користувачам повною мірою взаємодіяти з програмою без необхідності використання клавіатури чи миші, що відкриває додаткові можливості для тих, хто віддає перевагу альтернативним методам керування комп'ютером.

Рисунок 3.1 – Умовне зображення контролера серії Xbox Controller.

Завдяки використанню XInput, програма може отримувати інформацію від контролера [27], зокрема, про стан кнопок (чи була натиснута певна кнопка) та

положення аналогових стіків. Аналоговий стік – це елемент контролера, який постійно передає дані про своє місцезнаходження на двовимірній площині з координатами в діапазоні від -1 до 1, де (0,0) є центром координатної системи [26]. Інформація про положення стіків може бути використана різними способами. У даному випадку ці дані застосовуються для навігації в межах програми.

3.5.2. Підтримка різних мов

Програма підтримує використання двох мов – англійської і української. Для досягнення цього була проведена додаткова робота з перекладом тексту, подібний функціонал в свою чергу дозволяє донести інформацію до більшої кількості людей. Програма дозволяє користувачам легко змінювати мову під час своєї роботи без необхідності перезапуску. Цей функціонал не залежить від сторонніх API і в своїй основі є простою заміною одного тексту на інший, з використанням вбудованої в рушій системи локалізації, яка дозволяє легко знаходити і перекладати необхідний текст.

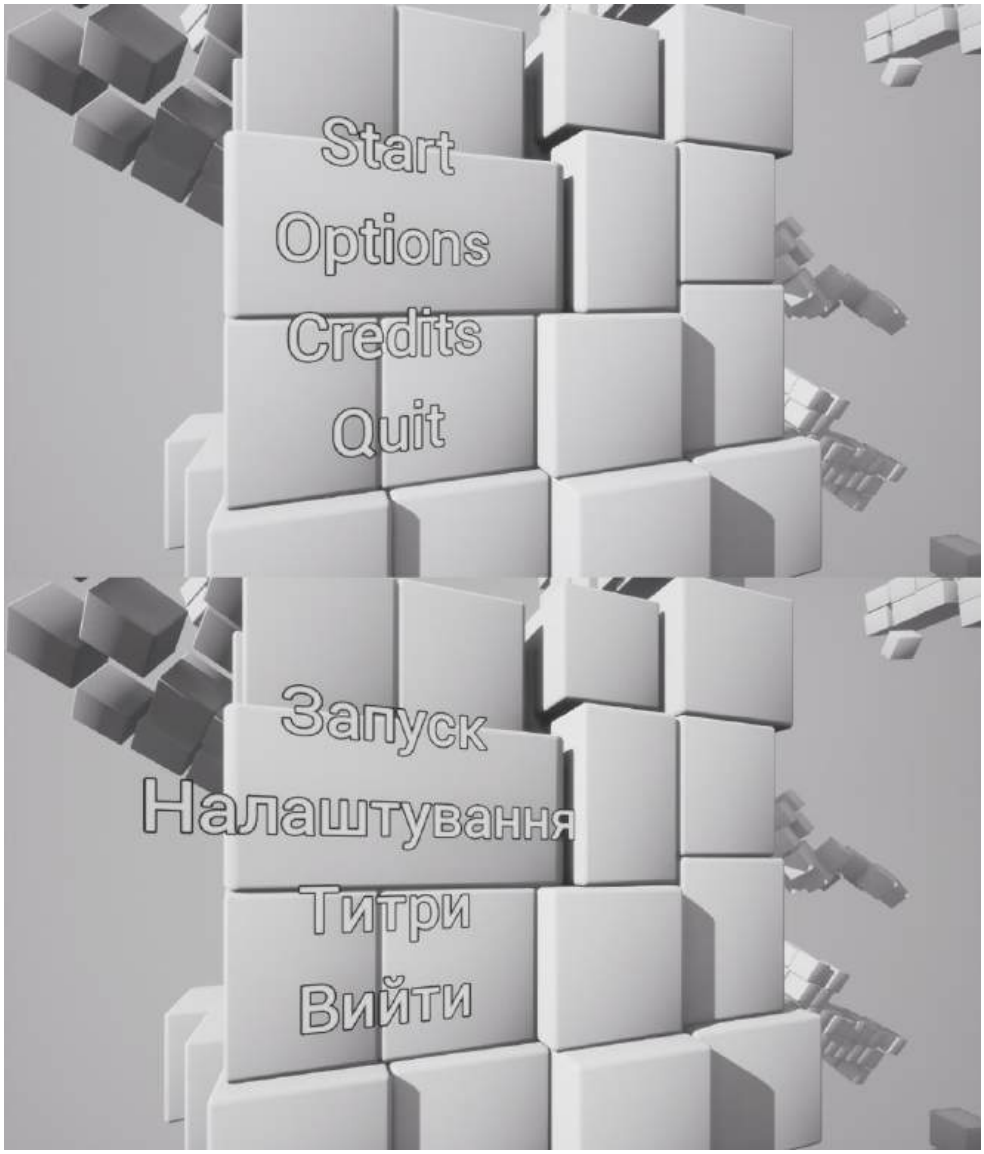


Рисунок 3.2 – Переклад головного меню програми.

3.5.3. Зміна графічних налаштувань програми

Програма надає можливість користувачам легко змінювати її графічні налаштування. Графічні налаштування мають великий вплив на різні параметри комп’ютерної графіки, та дозволяють користувачам додатково спростити різні обрахунки, що в свою чергу зменшує навантаження на комп’ютер і його графічний процесор. Зміни графічних налаштувань програми надають можливість слабшим

персональним комп’ютерам запускати програму, але з гіршою якістю зображення.

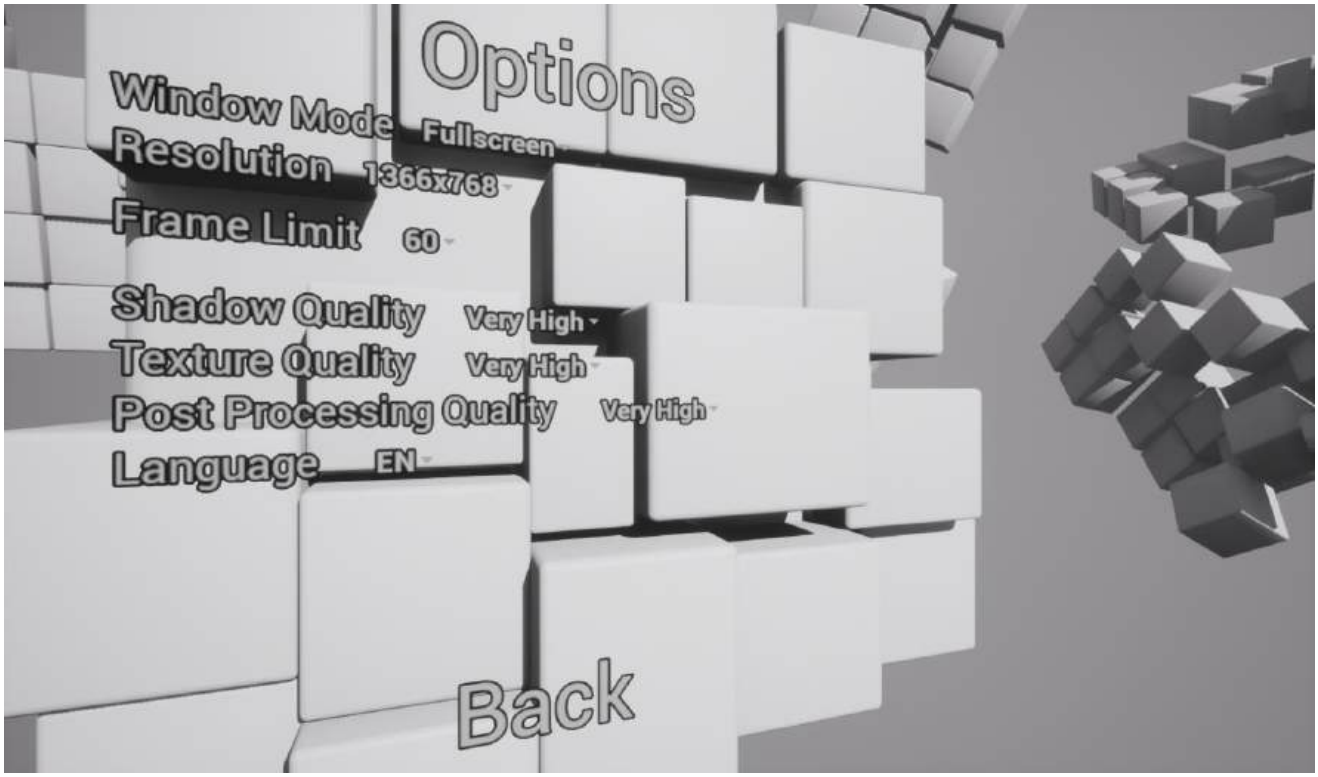


Рисунок 3.3 – Меню налаштувань програми.

Зменшення роздільної здатності зображення має значний вплив на продуктивність обчислень, оскільки з більшою кількістю пікселів зростає кількість необхідних обчислень.

Обмеження кадрів за секунду не підвищує швидкодію програми, а спрямоване на економію ресурсів комп’ютера. Якщо монітор підтримує частоту 60 Гц, тобто виводить 60 кадрів на секунду, то обчислення додаткових кадрів, що не відображаються на екрані, є надлишковими і непотрібними [10]. Користувач може змінювати ліміт частоти кадрів або повністю зняти обмеження для досягнення максимальної кількості кадрів у секунду.

Якість тіней визначає рівень деталізації динамічних тіней, які не були попередньо згенеровані. Кожна тінь має певний розмір 2^n, аналогічно до текстур. Зниження якості тіней веде до зменшення степеня n, наприклад, 4096x4096 стає 2048x2048. Залежно від кількості тіней на екрані, зменшення їхньої якості може суттєво вплинути на продуктивність програми, а також на візуальну якість тіней [10].

Якість текстур функціонує аналогічно як і тіні, але взаємодіє з уже створеними міпмапами. Зниження якості текстур призводить до переходу на нижчі рівні міпмапів, якщо такі існують. Хоча розмір текстур має незначний вплив на швидкодію, він дозволяє заощаджувати відеопам’ять, що може значно покращити роботу програми [10]. Візуально цей ефект буде помітним.

Якість постобробки впливає на візуальні ефекти, такі як корекція кольорів, затемнення по кутках, хроматична аберація та інші ефекти, що накладаються на зображення. Зменшення параметрів якості таких ефектів знижує точність обчислень, що може вплинути на їхню реалізацію та кінцевий результат [10]. Залежно від кількості активованих ефектів, це може суттєво вплинути на швидкодію.

Описані налаштування є базовими. Для складніших програм можуть бути додані додаткові опції, які дозволяють більш точно регулювати якість зображення та швидкодію програми.

3.5.4. Виведення інформації через взаємодію з об’єктами

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

З метою визначення погляду користувача використовуються сфери певного радіусу. Одна з таких сфер запускається кожен кадр роботи програми в напрямку погляду користувача на певну відстань. Якщо сфера зустрічає на своєму шляху певний об’єкт, то програма вважає, що користувач на нього дивиться. В програмі використовуються сфери з радіусом в 40см і максимальний їх шлях – 5 метрів. Коли сфера зустрічає перший об’єкт, то зупиняє свій подальший рух, що економить ресурси комп’ютера.

Для взаємодії з користувачем використовується створений під цього код. Якщо об’єкт, який зустріла сфера, має в собі імплементацію даного коду, тоді він виділяється певним кольором. Залежно від розширення зображення виділення може бути два і більше пікселі в ширину. Крім цього, після натискання конкретної кнопки користувач має можливість взаємодіяти з предметом, що, у більшості

випадків, виведе певну інформацію. Об’єкти взаємодії мають виглядати як таблички або різного виду кнопки, це робить процес взаємодії більш зрозумілим користувачеві.

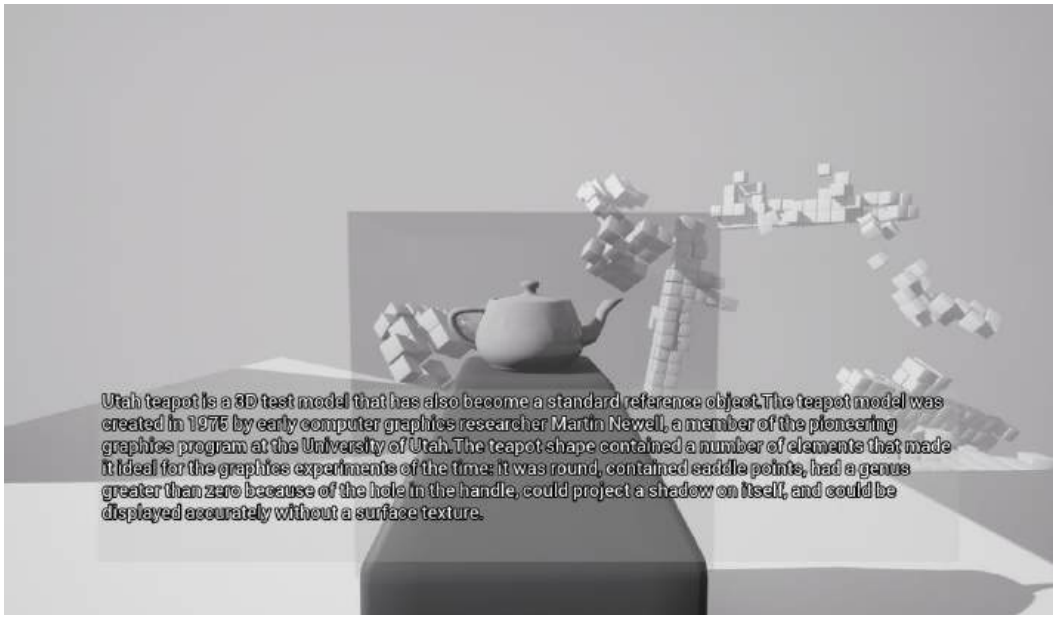
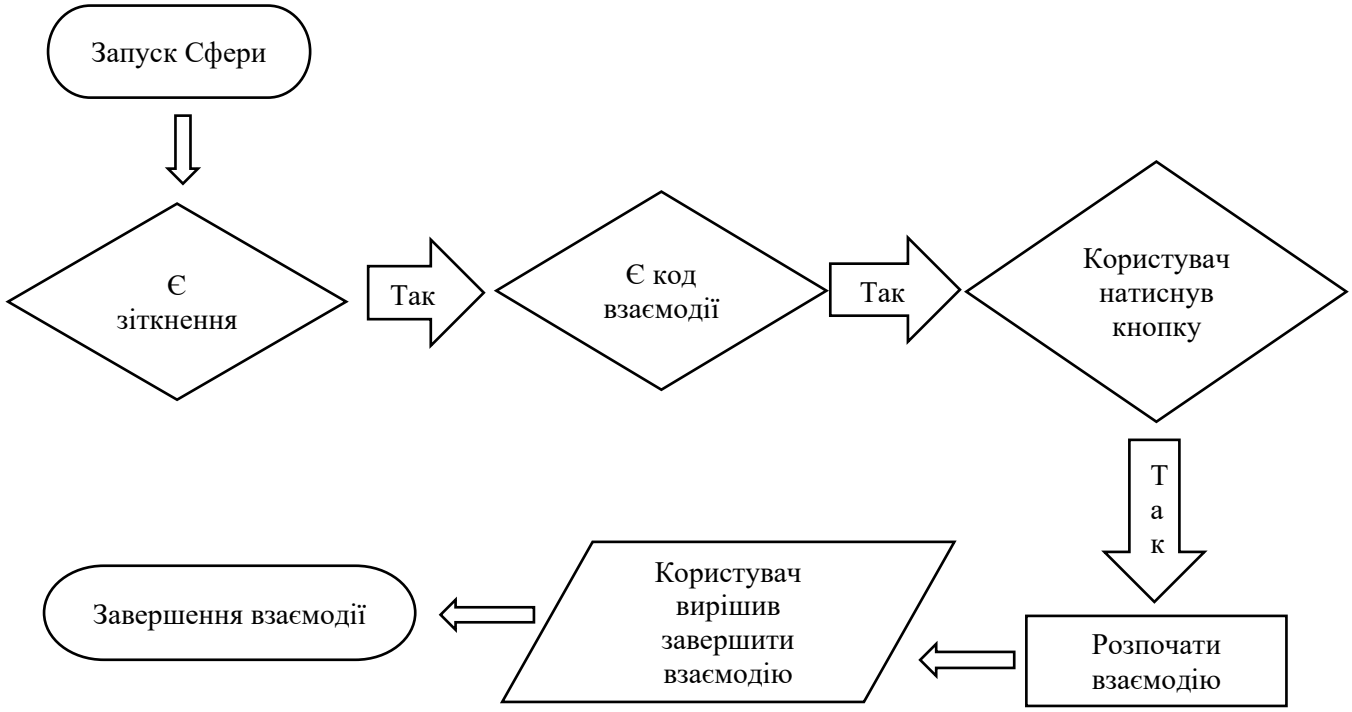


Рисунок 4.4 – Чайник з Юти і інформація про нього в англійському перекладі.

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



Блок-схема 2 – Робота коду взаємодії.

4. Економічна характеристика проектного виробу

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

4.1 Визначення собівартості і ціни створення програми

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

$$TC = FC + VC \tag{4.1}$$

TC – загальна вартість.

FC – фіксована вартість, а саме вартість незалежна від кількості виготовлених продуктів.

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

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

$$FC = DC + SC \tag{4.2}$$

DC – вартість розробки.

SC – вартість програмного забезпечення, яке необхідне для розробки.

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

$$DC = DN * DS * DT \tag{4.3}$$

DN – час розробки.

DS – зарплата одного робітника.

DT – кількість робітників.

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

$$SC = LC + SPC \quad (4.4)$$

LC – загальна вартість ліцензій на необхідний час.

SPC – загальна вартість програмного забезпечення, яке не потребує регулярних виплат.

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

Таблиця 4.1. – 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 для команди розробників. Це все звісно без урахування вартості необхідного апаратного забезпечення, а також без оренди приміщення і з мінімальними витратами.

Варто сказати, що створення простішого програмного забезпечення, яке наведене в дипломній роботі, звісно, вимагало менших витрат. Визначення його вартості буде відбуватися без урахування апаратного забезпечення і звісно без додавання вже придбаного програмного забезпечення, а також без необхідності платити зарплату. Час розробки – 9 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 абсолютної відсутності зарплати.

5. Охорона праці та безпека

5.1. Аналіз шкідливих дій при виготовленні охоронного пристрої.

В даному проекті було створено програмне забезпечення. Процес створення програмного забезпечення містить в собі 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. Створено програмне забезпечення, що працює з комп’ютерною графікою реального часу.
4. Створене програмне забезпечення було використано в навчальних цілях.

Використані джерела:

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

17. Epic Games Incorporated. Understanding Lightmapping in Unreal Engine https://docs.unrealengine.com/5.0/en-US/understanding-lightmapping-in-unreal-engine/
18. Epic Games Incorporated. Unreal Engine Hardware Ray Tracing https://docs.unrealengine.com/5.2/en-US/hardware-ray-tracing-in-unreal-engine/
19. Epic Games Incorporated. 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/
20. Learn OpenGL – Graphics Programming. Joey de Vries https://learnopengl.com/book/book_pdf.pdf
21. Microsoft Corporation. Direct3D 12 programming guide https://learn.microsoft.com/en-us/windows/win32/direct3d12/directx-12-programming-guide
22. Microsoft Corporation Getting started with DirectX Graphics https://learn.microsoft.com/en-us/windows/win32/getting-started-with-directx-graphics
23. Microsoft Corporation Getting Started With XInput in Windows applications https://learn.microsoft.com/en-us/windows/win32/xinput/getting-started-with-xinput
24. Microsoft Corporation Programming Guide (XInput Game Controller APIs) https://learn.microsoft.com/en-us/windows/win32/xinput/programming-guide
25. Nvidia Corporation DIRECTX 12 ULTIMATE https://developer.nvidia.com/directx
26. Nvidia Corporation NVIDIA RTX Path Tracing SDK https://developer.nvidia.com/rtx/path-tracing
27. Nvidia Corporation Ray Tracing Essentials 1: https://developer.nvidia.com/blog/ray-tracing-essentials-part-1-basics-of-ray-tracing/
28. Nvidia Corporation Ray Tracing Resources Page https://www.realtimerendering.com/raytracing.html
29. Nvidia Corporation What is Path Tracing? https://blogs.nvidia.com/blog/2022/03/23/what-is-path-tracing/
30. Physically Based Rendering: From Theory To Implementation https://www.pbr-book.org/3ed-2018/contents
31. Ryusuke Villemin , Pixar Animation Studios Art and Technology at Pixar https://graphics.pixar.com/library/SigAsia2018Course/paper.pdf
32. Per H. Christensen, Pixar Animation Studios Ray Tracing for the Movie ‘Cars’ https://graphics.pixar.com/library/RayTracingCars/paper.pdf
33. Dana Batali, Pixar Animation Studios RenderMan, Theory and Practice https://graphics.pixar.com/library/RMan2003/paper.pdf
34. Christine Waggoner ,Pixar Animation Studios Revamping the Cloth Tailoring Pipeline at Pixar https://graphics.pixar.com/library/C3d/paper.pdf
35. Robert L. Cook Pixar Animation Studios Stochastic Simplification of Aggregate Detail https://graphics.pixar.com/library/StochasticSimplification/paper.pdf

36. Pixar Animation Studios Volume Rendering for Pixar's Elemental https://graphics.pixar.com/library/ElementalVolume/paper.pdf
37. Real-Time Rendering Fourth Edition, Online chapter: Real-Time Ray Tracing https://www.realtimerendering.com/Real-Time_Rendering_4th-Real-Time_Ray_Tracing.pdf
38. Craig Schroeder, The Walt Disney Company A Material Point Method For Snow Simulation https://media.disneyanimation.com/uploads/production/publication_asset/94/asset/SSCTS13_2.pdf
39. The Walt Disney Company Denoising with Kernel Prediction and Asymmetric Loss Functions https://la.disneyresearch.com/publication/denoising-with-kpal/
40. Christian Eisenacher, 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
41. 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
42. 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
43. The Walt Disney Company Automatic Feature Selection for Denoising Volumetric Renderings https://studios.disneyresearch.com/2022/09/19/automatic-feature-selection-for-denoising-volumetric-renderings/
44. The Walt Disney Company Unifying Points, Beams, and Paths in Volumetric Light Transport Simulation https://studios.disneyresearch.com/2014/07/27/unifying-points-beams-and-paths-in-volumetric-light-transport-simulation/
45. The Walt Disney Company Image-Space Control Variates for Rendering https://studios.disneyresearch.com/2016/11/11/image-space-control-variates-for-rendering/
46. The Walt Disney Company Joint Importance Sampling of Low-Order Volumetric Scattering https://studios.disneyresearch.com/2013/11/01/joint-importance-sampling-of-low-order-volumetric-scattering/
47. The Walt Disney Company Physically-based Simulation of Rainbows https://studios.disneyresearch.com/2012/01/01/physically-based-simulation-of-rainbows/
48. Nvidia Corporation Recursive Control Variates for Inverse Rendering https://research.nvidia.com/publication/2023-05_recursive-control-variates-inverse-rendering

49. Nvidia Corporation VMF Diffuse: A unified rough diffuse BRDF https://research.nvidia.com/publication/2024-07_vmf-diffuse-unified-rough-diffuse-brdf
50. Nvidia Corporation ReSTIR GI: Path Resampling for Real-Time Path Tracing https://research.nvidia.com/publication/2021-06_restir-gi-path-resampling-real-time-path-tracing
51. Nvidia Corporation Scaling Probe-Based Real-Time Dynamic Global Illumination for Production https://research.nvidia.com/publication/2021-05_scaling-probe-based-real-time-dynamic-global-illumination-production
52. Nvidia Corporation Glossy Probe Reprojection for Interactive Global Illumination https://research.nvidia.com/publication/2020-11_glossy-probe-reprojection-interactive-global-illumination

Додаток:

Код камери програми, містить в собі функціонал взаємодії з тривимірним середовищем.

```
// Copyright Bohdan Basok

#pragma once

#include "CoreMinimal.h"
#include "CineCameraComponent.h"
#include "NyxCineCameraComponent.generated.h"

class ANyxPlayerController;
class UNyxInteractionInterface;
class UNyxGameUserSettings;
/**
 *
 */
UCLASS()
class NYX_API UNyxCineCameraComponent : public
UCineCameraComponent
{
	GENERATED_BODY()

public:
	UNyxCineCameraComponent();

	virtual void TickComponent(float DeltaTime, ELevelTick TickType,
FActorComponentTickFunction* ThisTickFunction) override;

	virtual void BeginPlay() override;

	//Interaction Interface
	void DoInteraction();

	//Camera Shake

	UPROPERTY(EditAnywhere, Category= "Nyx CineCamera")
	TSubclassOf<UCameraShakeBase> CameraShakeIdle;

	UPROPERTY(EditAnywhere, Category= "Nyx CineCamera")
	TSubclassOf<UCameraShakeBase> CameraShakeWalking;

	UPROPERTY(EditAnywhere, Category= "Nyx CineCamera")
	TSubclassOf<UCameraShakeBase> CameraShakeRunning;
```

```
        UPROPERTY(EditAnywhere, Category = "Debug")
        bool bDrawDebug;

protected:
        //DoF

        UPROPERTY(EditAnywhere, Category = "Nyx CineCamera|DoF")
        bool bIsDynamicDOFEnabled;

        UPROPERTY(EditAnywhere, Category = "Nyx CineCamera|DoF",
                meta
=(EditCondition="bIsDynamicDOFEnabled",EditConditionHides, Units
="Centimeters"))
        float MaximumDOFFocusLength;

        UPROPERTY(EditAnywhere, Category = "Nyx CineCamera|DoF",
                meta
=(EditCondition="bIsDynamicDOFEnabled",EditConditionHides,Units
="Centimeters"))
        float DefaultFocusDistance;

        //Camera Shake

        UPROPERTY(EditAnywhere, Category = "Nyx CineCamera|Camera
Shake")
        bool bIsCameraShakeEnabled;

        UPROPERTY(EditAnywhere, Category = "Nyx CineCamera|Camera
Shake",
                meta
=(EditCondition="bIsCameraShakeEnabled",EditConditionHides, Units
="CentimetersPerSecond"))
        float IdleVelocity;

        UPROPERTY(EditAnywhere, Category = "Nyx CineCamera|Camera
Shake",
                meta
=(EditCondition="bIsCameraShakeEnabled",EditConditionHides, Units
="CentimetersPerSecond"))
        float WalkingVelocity;

        UPROPERTY(EditAnywhere, Category = "Nyx CineCamera|Camera
Shake",
                meta
=(EditCondition="bIsCameraShakeEnabled",EditConditionHides, Units
="CentimetersPerSecond"))
```

```
        float RunningVelocity;

        void LineTrace();

        void CameraShake();

        void DynamicCameraDOF(float Distance);

        UFUNCTION()
        void UpdateSettings();

        UPROPERTY()
        ANyxPlayerController* MyController;

        UPROPERTY()
        APawn* MyOwner;

        UPROPERTY()
        FCameraFocusSettings DynamicDOF;

        UPROPERTY()
        TObjectPtr<UNyxGameUserSettings> CurrentGameUserSettings;

        //Interaction Interface

        UPROPERTY()
        TObjectPtr<AActor> HitActor;

        UPROPERTY()
        TObjectPtr<AActor> OldHitTarget;

        UPROPERTY(EditAnywhere, Category = "Nyx CineCamera|Interaction",
                meta =(Units ="Centimeters"))
        float SweepSphereRadius;

        UPROPERTY(EditAnywhere, Category = "Nyx CineCamera|Interaction",
                meta =(Units ="Centimeters"))
        float MaximumInteractionDistance;

};
```

```
// Copyright Bohdan Basok


#include "Graphics/NyxCineCameraComponent.h"


#include "Game/NyxGameUserSettings.h"
#include "Kismet/GameplayStatics.h"
#include "Player/NyxInteractionInterface.h"
#include "Player/NyxPlayerController.h"
#include "Game/NyxLog.h"

UNyxCineCameraComponent::UNyxCineCameraComponent()
{
	PrimaryComponentTick.bCanEverTick = true;

	MaximumInteractionDistance = 300.0f;

	bIsDynamicDOFEnabled = true;
	bIsCameraShakeEnabled = true;

	MaximumDOFFocusLength = 350.0f;
	DefaultFocusDistance = 100000.0f;

	IdleVelocity = 0.0f;
	WalkingVelocity = 300.0f;
	RunningVelocity = 500.0f;

	bDrawDebug = false;

	Filmback.SensorWidth = 36.0f;
	Filmback.SensorHeight = 24.0f;

	LensSettings.SqueezeFactor = 1.33f;
	LensSettings.DiaphragmBladeCount = 9;

	CurrentFocalLength = 32.0f;
	CurrentAperture = 2.0f;

	SweepSphereRadius = 20.0f;
}

void UNyxCineCameraComponent::TickComponent(float DeltaTime,
ELevelTick TickType,
	FActorComponentTickFunction* ThisTickFunction)
```

```
{
	Super::TickComponent(DeltaTime, TickType, ThisTickFunction);

	LineTrace();
	if(bIsCameraShakeEnabled)
	{
		CameraShake();
	}
}

void UNyxCineCameraComponent::BeginPlay()
{
	Super::BeginPlay();
	MyController =
Cast<ANyxPlayerController>(UGameplayStatics::GetPlayerController(GetWorl
d(),0));
	MyOwner = Cast<APawn>(GetOwner());

	CurrentGameUserSettings =
Cast<UNyxGameUserSettings>(GEngine->GetGameUserSettings());
	CurrentGameUserSettings->CameraSettingsChanged.AddUniqueDynamic
(this, &UNyxCineCameraComponent::UpdateSettings);
}

void UNyxCineCameraComponent::DoInteraction()
{
	FCollisionQueryParams CollisionQueryParams;
	CollisionQueryParams.AddIgnoredActor(MyOwner);

	FVector EyeLocation = GetComponentLocation();;
	FRotator EyeRotation = GetComponentRotation();;

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

	FHitResult HitResult;
	if(GetWorld()->LineTraceSingleByChannel(HitResult,EyeLocation, End,
ECC_WorldDynamic, CollisionQueryParams))
	{
		HitActor = HitResult.GetActor();
		if(HitActor != nullptr)
		{
			if(HitActor->Implements<UNyxInteractionInterface>())
			{
```

```
		INyxInteractionInterface::Execute_Interaction(HitActor, MyOwner,
HitResult);
				}

			}
		}
		if (bDrawDebug)
		{
			DrawDebugLine(GetWorld(), EyeLocation, End, FColor::Emerald,
false, 2.0f, 0, 2.0f);
		}
}

/**
 * Line Trace from camera. There are several scenarios:
 *	A. LastActor has no interface && ThisActor has no interface
 *		- Do nothing
 *	B. LastActor has no interface && ThisActor has an interface
 *		- Highlight ThisActor
 *	C. LastActor has an interface valid && ThisActor has no interface
 *		- Unhighlight LastActor
 *	D. Both have an interface, but LastActor != ThisActor
 *		- Unhighlight LastActor and Highlight ThisActor
 *	E. Both actor have an interface and both are the same actor
 *		- Do nothing
 *	F. LineTrace hit no target, therefore ThisActor is null
 *		- Unhighlight LastActor
 ***/

void UNyxCineCameraComponent::LineTrace()
{
		FCollisionQueryParams CollisionQueryParams;
		CollisionQueryParams.AddIgnoredActor(MyOwner);

		FVector EyeLocation = GetComponentLocation();;
		FRotator EyeRotation = GetComponentRotation();;

		//Sets maximum line trace length in cm
		FVector End = EyeLocation + (EyeRotation.Vector() *
MaximumInteractionDistance);

		FCollisionShape Shape;
		Shape.SetSphere(SweepSphereRadius);

		//Line Trace for Outlines
```

```
        FHitResult HitResult;
        bool bBlockingHit = GetWorld()->SweepSingleByChannel(HitResult,
EyeLocation,
End,FQuat::Identity,ECC_WorldDynamic,Shape,CollisionQueryParams);

        FColor LineColor = bBlockingHit ? FColor::Green : FColor::Magenta;


        OldHitTarget = HitActor;
        HitActor = HitResult.GetActor();
        if(bBlockingHit)
        {
                if(OldHitTarget)
                {
                        if(!OldHitTarget->Implements<UNyxInteractionInterface>())
                        {

        if(HitActor->Implements<UNyxInteractionInterface>())
                                {
                                        //Case B

        INyxInteractionInterface::Execute_OnLineTraceHit(HitActor,MyOwner,
HitResult);
                                } //Else Case A
                        }
                }
                 //LastActor is valid

                if(!HitActor->Implements<UNyxInteractionInterface>())
                {
                        //Case C
                        if(OldHitTarget)
                        {

        if(OldHitTarget->Implements<UNyxInteractionInterface>())
                                {

        INyxInteractionInterface::Execute_OnLineTraceEnd(OldHitTarget,MyOw
ner,HitResult);
                                }
                        }
                }
                else //Both actors are valid
                {
                        if(HitActor != OldHitTarget)
                        {
```

```
                        //Case D
                        if(OldHitTarget)
                        {

        if(OldHitTarget->Implements<UNyxInteractionInterface>())
                                {

        INyxInteractionInterface::Execute_OnLineTraceEnd(OldHitTarget,MyOw
ner,HitResult);
                                }
                        }


        INyxInteractionInterface::Execute_OnLineTraceHit(HitActor,MyOwner,
HitResult);
                } //Else Case E - do nothing
        }

        }
        else //Case F
        {
                if(OldHitTarget)
                {
                        if(OldHitTarget->Implements<UNyxInteractionInterface>())
                        {

        INyxInteractionInterface::Execute_OnLineTraceEnd(OldHitTarget,MyOw
ner,HitResult);
                        }
                }
        }


        if(bDrawDebug)
        {

        DrawDebugSphere(GetWorld(),HitResult.ImpactPoint,SweepSphereRadiu
s, 32,LineColor,false,2.0f );
        }

        //Line Trace for DoF
        End = EyeLocation + (EyeRotation.Vector() *
MaximumDOFFocusLength);
        if(GetWorld()->LineTraceSingleByChannel(HitResult,EyeLocation, End,
ECC_WorldDynamic, CollisionQueryParams))
        {
```

```
                if(bIsDynamicDOFEnabled)
                {
                        DynamicCameraDOF(HitResult.Distance);
                }
        }
        else
        {
                DynamicDOF.ManualFocusDistance = DefaultFocusDistance;
                SetFocusSettings(DynamicDOF);
                if(bDrawDebug)
                {
                        UE_LOGFMT(NyxLog,Log,"No Hit for Dof ");
                }

        }
}

void UNyxCineCameraComponent::CameraShake()
{
        if(!CameraShakeIdle||!CameraShakeWalking||!CameraShakeRunning)
        {
                return;
        }

        if(GetOwner()->GetVelocity().Length()<=IdleVelocity)
        {
                MyController->ClientStartCameraShake(CameraShakeIdle, 1.0f,
ECameraShakePlaySpace::CameraLocal, FRotator::ZeroRotator);
        }
        else if(GetOwner()->GetVelocity().Length()<=WalkingVelocity)
        {
                MyController->ClientStartCameraShake(CameraShakeWalking,
1.0f, ECameraShakePlaySpace::CameraLocal, FRotator::ZeroRotator);
        }
        else //if (GetOwner()->GetVelocity().Length()<=RunningVelocity)
        {
                MyController->ClientStartCameraShake(CameraShakeRunning,
1.0f, ECameraShakePlaySpace::CameraLocal, FRotator::ZeroRotator);
        }
}

void UNyxCineCameraComponent::DynamicCameraDOF(float Distance)
{
        DynamicDOF.ManualFocusDistance = Distance;
        SetFocusSettings(DynamicDOF);
}
```

```
void UNyxCineCameraComponent::UpdateSettings()
{
	bIsDynamicDOFEnabled =
CurrentGameUserSettings->GetIsDepthOfFieldEnabled();
	bIsCameraShakeEnabled =
CurrentGameUserSettings->GetIsCameraShakeEnabled();
}
```