ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інженерії програмного забезпечення

Пояснювальна записка

до бакалаврської роботи
на ступінь вищої освіти бакалавр

на тему: **«ВІЗУАЛІЗАЦІЯ 3D СЦЕН НА БАЗІ ПОСТ-ПРОЦЕСІВ ЗА ДОПОМОГОЮ DIRECT3D»**

Виконав: студент 5 курсу, групи ППЗ-51
Спеціальності:
121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

Гугля А.О.
(прізвище та ініціали)
Керівник Марченко В.В.
(прізвище та ініціали)
Рецензент_____
(прізвище та ініціали)
Нормоконтроль_____
(прізвище та ініціали)

Київ – 2021

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ
Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти - «Бакалавр»

Спеціальність - 121 Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ

Завідувач кафедри

Інженерії програмного забезпечення

_____ О.В. НЕГОДЕНКО

«____» _____2021 року

З А В Д А Н Н Я

НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТУ

_____Гуглі Артему Олексійовичу_____
(прізвище, ім'я, по батькові)

1.Тема роботи: «Візуалізація 3D сцен на базі пост-процесів за допомогою direct3d»

Керівник роботи: Марченко Віталій Вікторович_____,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від «12» березня 2021 року №65.

2. Строк подання студентом роботи «01» червня 2021 року

3. Вихідні дані до роботи:

3.1 Мова програмування C++

3.2 Рендер API DirectX 11

3.3 Мова програмування шейдерів HLSL

3.4 Науково-технічна література,

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):

4.1 Аналіз предметної області

4.2 Вибір засобів розробки

4.3 Імплементація візуалізатора

4.4 Аналіз результатів та висновки

5. Перелік графічного матеріалу:

5.1 Приклад набору вершин

5.2 Положення про матриці трансформацій

5.3 Математичні формули
5.4 Теоритичні пояснення роботи текстур
5.5 Зображення 3D сцени
5.6 Зображення світла на тіні
5.7 Зображення пост-процесів
5.8 Фінальне зображення візуалізатора

6. Дата видачі завдання: «19» квітня 2021 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури	19.04.21	Виконано
2	Аналіз прикладів створення візуалізаторів	20.04.21	Виконано
3	Налаштування системи генерації проекту	22.04.21	Виконано
4	Створення архітектури програми	25.04.21	Виконано
5	Розробка візуалізатора	27.04.21	Виконано
6	Імплементація пост-процесів	29.04.21	Виконано
7	Вступ, висновки, реферат, демонстраційні матеріали	30.04.21	Виконано
8	Попередній захист роботи	10.04.21	Виконано
9	Здача роботи в деканат	01.06.21	Виконано

Студент _____ Гугля А.О.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Марченко В.В.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Текстова частина бакалаврської роботи 44 с., 34 рис., 4 джерела.

Об`єкт дослідження — вплив пост-процесів на фінальне зображення в комп'ютерній графіці.

Предмет дослідження — візуалізатор 3Д сцени.

Мета роботи — імплементувати візуалізатор 3Д сцени та пост-процеси, проаналізувавши як саме буде змінюватися якість зображення.

Актуальність роботи полягає в аналізі зображення окремо за з пост-процесами.

Методи дослідження: за допомогою розробленого візуалізатора на базі DirectX 11, використовуючи мову програмування C++, а також реалізація найпопулярніших пост-процесів на мові програмування HLSL.

Стислий опис результатів дослідження: розроблено візуалізатор, та набір пост-процесів для 3Д сцени. Це дає нам змогу оцінити важливість застосування пост обробки як дуже відчутну для зображення.

Галузь використання — 3Д моделювання, відеоігри.

Ключові слова: ВІЗУАЛІЗАТОР, ПОСТ-ПРОЦЕСИ, РЕНДЕР.

ЗМІСТ

ВСТУП

Вже багато років поспіль комп’ютерна графіка стрімко розвивається. Будь то спеціальні ефекти в кіно та відеоіграх, проектування будівель, або використання 3D друку. Ми контактуємо с нею в багатьох сферах життя. Загалом для отримання необхідної для людини інформації з відео дисплею нам достатньо відтворити два елементи зображення:

- Розташування об'єктів з відчутною перспективною проекцією
- Примітивна світлотінь для розпізнавання граней фігур

Але це все було досягнуто в середині 20-го сторіччя. Що ж є рушієм розвитку комп'ютерної графіки в сьогоденні?

Всі наукові роботи зводиться до однієї цілі: показати зображення якомога схожим на наше життя. Якщо казати спрощено, то ми намагаємось зробити зображення на відео дисплеї фізично коректним. В комп’ютерній графіці це називається PBR(physically based rendering).

Що саме заважає нам просто відтворити всі закони фізики використовуючи необхідні обчислення? Обчислювальне обмеження комп’ютерів. Особливо якщо ми говоримо про рендеринг(відображення) в реальному часі, який має оновлювати зображення достатньо швидко. Тому всі напрямки досліджень зводяться до знаходження алгоритмів, які будуть розраховувати якнаймога схоже до реальності зображення, при цьому правильно та ефективно використовуючи пам’ять та обчислювальні можливості техніки.

Основний вплив на зображення, після того, як була намальована геометрія, мають пост-процеси(пост обробка зображення). На цьому етапі рендерингу ми вже не додаємо на сцену додаткових примітивів(світло, 3Д об’єкти та інші), а працюємо з готовим вихідним зображенням та деякою інформацією, яку ми накопичили, поки його малювали: карта нормалей вершин, карта позицій вершин, карта затемнення, матриці для перетворення координат вершин з локальної системи координат у систему дисплея та багато-багато іншого.

Ми дослідимо як саме можна вплинути на фінальну якість малюнку за допомогою пост-процесів.

У своїй роботі я буду імплементувати:

- Візуалізацію сцени з нуля. Хоча в наш час і існує багато різноманітних програм для рендерингу, я буду імплементувати свій рендеринг, бо це дає мені повний контроль над ресурсами програми, а також дає можливість модифікувати процес малювання сконцентрувавшись на пост-процесах.
- Було вирішено для аналізу скомбінувати такі пост-процеси: відбиття в екранному просторі SSR(screen space reflections), згладження MSAA(Multisample anti-aliasing), відблиск об'єктива LS(lens flare), світіння(Bloom), а також тональне відображення(tone mapping).

Оскільки все буде виконуватись в режимі реального часу прийнятою швидкістю оновлення малюнку будемо мати такий стандарт: 60 кадрів за секунду.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 3D Об’єкти. Матриці трансформації

Найголовнішим примітивом для малювання в комп’ютерній графіці є піксель, але етап, коли ми доходимо до його зафарбовування – майже останній.

Ядро візуалізації 3D сцени – моделі та їх матеріали. Саме вони є первинними вхідними даними для рендеру. Як саме їх задати?

В комп’ютерній графіці моделі з себе представляють набір вершин. Кожні три вершини задають трикутник. Цей самий трикутник – важливий примітив для замалювання пікселів моделі. Саме він, зазвичай, визначає як буде виглядати та чи інша поверхня. Тому дуже важливо при формуванні набору вершин, які пізніше будуть відправлені до відеокарти, тримати в голові що кожні наступні три вершини мають задавати трикутник.

Коли ми дійдемо до етапу зафарбування кожного пікселю, саме за допомогою набору з трьох вершин буде виконана інтерполяція в трикутнику, після чого ми отримаємо вершинну інформацію в кожному окремому пікселі.

Зазвичай, окрім позиції у локальному для моделі світі, ми також можемо мати наступну інформацію щодо кожної вершини:

- Вектор нормалі. Використовується для багатьох наступних апроксимацій: світла, відбиття, затемнення та інші. Він може бути вирахуваний, коли ми вже будемо знати усі три вершини трикутника. Їх достатньо щоб за допомогою векторного добутку знайти вектор нормалі для поверхні, а потім записати це значення до кожної вершини. Але результат буде дуже різким та не придатним для PBR. Тому є два розв'язання цієї проблеми: заздалегідь розрахувати нормаль для кожної вершини так, щоб зробити поверхню якнаймога гладкою, або використати карту нормалі. Другий варіант більш популярний, бо саме він дає максимально реалістичні значення. Використовуючи карту

нормалі, ми зможемо задавати унікальне значення нормалі для кожного пікселю. І тоді ми можемо уникнути цього компоненту в вершині.

- Двовимірна текстурна координата. Вона задає координати звідки саме нам потрібно зчитувати інформацію з текстур для окремої вершини. Оскільки текстури, зазвичай, задаються двовимірними зображеннями – вони можуть мати різну роздільну здатність. Наприклад, одна текстура буде розміром 1000x200, а інша – 500x500. Тому текстурні координати задаються в діапазоні [0; 1], що є відносним значенням. А потім вже, при читанні, ми переносимо це значення до розмірів кожного окремого зображення. Як вже було зазначено вище, за допомогою інтерполяції в трикутниках ми будемо визначати значення текстурної координати для кожного окремого пікселя. Цей компонент дуже важливий.
- Сама по собі позиція вершини у локальній системі координат.
- Інколи в цілях спрощення також вказують колір вершини, який потім буде градієнтом(знову ж таки, за допомогою інтерполяції) визначати колір поверхні.

Майже всі вищезазначені компоненти – вибіркові. Усі, крім позиції. Позиція кожної вершини зазвичай задається коли модель створюється в спеціальних програмах, таких як: Blender, Maya, 3DMax та інші. Тому нам важливо правильно імпортувати моделі, використовуючи документації окремих програм, щодо того, як вони містять в собі вершини та їх компоненти.

Також треба звертати увагу на базис системи координат, та де він знаходиться відносно моделі. Тому що нам належить переводити кожну позицію з її локальної системи координат в нашу, яку ми використовуємо у своїй програмі.

Оскільки майже кожен трикутник моделі буде мати дві спільних вершини з іншими трикутниками. То заведено використовувати ще додатково до масиву вершин ще й масив індексів. Це допомагає заощадити дуже велику кількість місця, так замість дублювання одних і тих же вершин для різних трикутників(а як ми знаємо в одній вершині може зберігатися дуже багато інформації), ми будемо лише

дублювати їх індекси в масиві. При цьому кожна вершина буде в одному екземплярі.

Нижче наведений приклад моделі в нашій роботі (рисунок 1.1): чайник, намальований зі спеціальним D3D11_FILL_WIREFRAME модом, щоб показати з яких примітивів він складається:

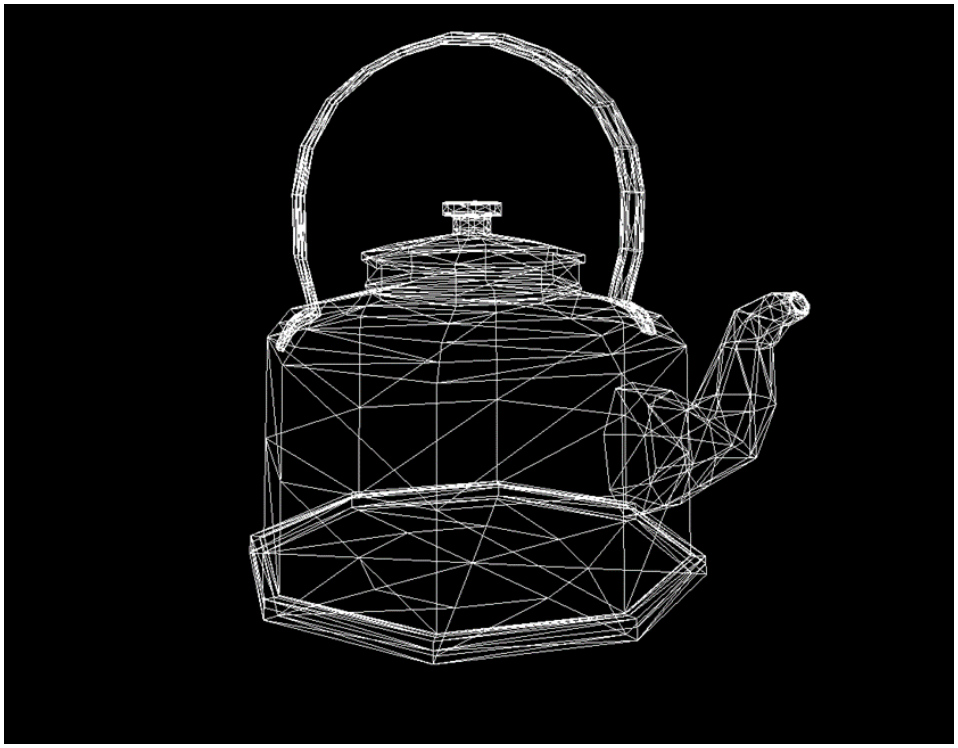


Рисунок 1.1 – Сітка моделі чайнику. Наочно видно вершини, та трикутники, які вони створюють.

Отримавши набір вершин та набір їх індексів, які утворюють трикутники, ми розпочинаємо готувати ланцюжок трансформацій, які потрібно зробити задля того, щоб перенести кожну вершину з її локальної системи координат у систему координат нашого дисплея.

Для цього ми будемо використовувати матриці трансформації [1]. Оскільки всі наші трансформації будуть проходити в тривимірному просторі, та будуть нести в собі лише афінне перетворення, то нам достатньо буде для кожної такої матриці.

Матриця повороту:

$$\begin{bmatrix} \cos\theta + R_x^2(1-\cos\theta) & R_xR_y(1-\cos\theta) - R_z\sin\theta & R_xR_z(1-\cos\theta) + R_y\sin\theta \\ R_yR_x(1-\cos\theta) + R_z\sin\theta & \cos\theta + R_y^2(1-\cos\theta) & R_yR_z(1-\cos\theta) - R_x\sin\theta \\ R_zR_x(1-\cos\theta) - R_y\sin\theta & R_zR_y(1-\cos\theta) + R_x\sin\theta & \cos\theta + R_z^2(1-\cos\theta) \end{bmatrix}$$

(1.1)

Компоненти матриці повороту згідно формули 1.1: $R(x, y, z)$ — довільна вісь обертання(де R — Rotation), θ — кут повороту навколо заданій вісі.

Матриця перенесення.

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.2)$$

Компоненти матриці перенесення згідно з формулою 1.2: $T(x, y, z)$ – вектор перенесення вершини(Де T — Translation).

Треба звернути увагу, що всупереч тому, що ми у 3D просторі, всі матриці є чотирьох вимірними. Це все із за впливу гомогенної координати w. Вона зручно дозволяє нам робити перенесення вектору, бо по правилам множення матриць ми не можемо інакше виконати логіку перенесення, тобто додавання векторів (формула 1.3):

$$V(x, y, z) = V_o(x, y, z) + T(x, y, z) \quad (1.3)$$

Де: V_o – розташування нашої вершини до трансформації, V – розташування нашої вершини після перенесення, T– значення перенесення для кожної вісі.

Матриця масштабування:

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.4)$$

Компоненти матриці масштабування згідно формули 1.4:
$S(x, y, z)$ – вектор масштабування для кожної з осей(де S – Scaling).

Скомбінувавши всі матриці разом ми отримаємо нашу матрицю трансформацію вершини у 3D просторі (формула 1.5):

$$M_{Transformation} = M_T \cdot M_R \cdot M_S \quad (1.5)$$

Де: $M_{Transformation}$ – фінальна матриця трансформації, M_S – матриця масштабування, M_R – матриця повороту, M_T – матриця перенесення.

Зверніть увагу! Порядок множення матриць інверсійний. Ми використовуємо в поясненні так звану систему матриць на основі рядків, тому вершини, на які ми будемо множити матрицю трансформації треба розглядати як наступну матрицю 4x1 (формула 1.6):

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \quad (1.6)$$

Зверніть увагу на гомогенну координату w. Вона буде мати значення 1 або 0 в залежності це вектор позиції чи вектор направлення. Поглянувши на матрицю перенесення стає зрозуміло яку роль грає цей компонент у добутку: для значення w — 1 ми будемо робити перенесення, а для значення 0 — ні(оскільки це не має сенс для вектора направлення).

Ми вирішили задачу створення матриці трансформацій, але скільки таких матриць нам треба, щоб перенести вершину з її локальної системи координат в систему дисплею.

Виділяють 3 основні:

- Матриця перетворення з локальної системи в систему світу нашої програми.
- Матриця перетворення з системи світу нашої програми у систему камери.
- Матриця проекції.

Для наочного прикладу які саме матриці трансформації взаємодіють с кожною вершиною, а також в якому порядку вони множаться дивіться на рисунок 1.2:

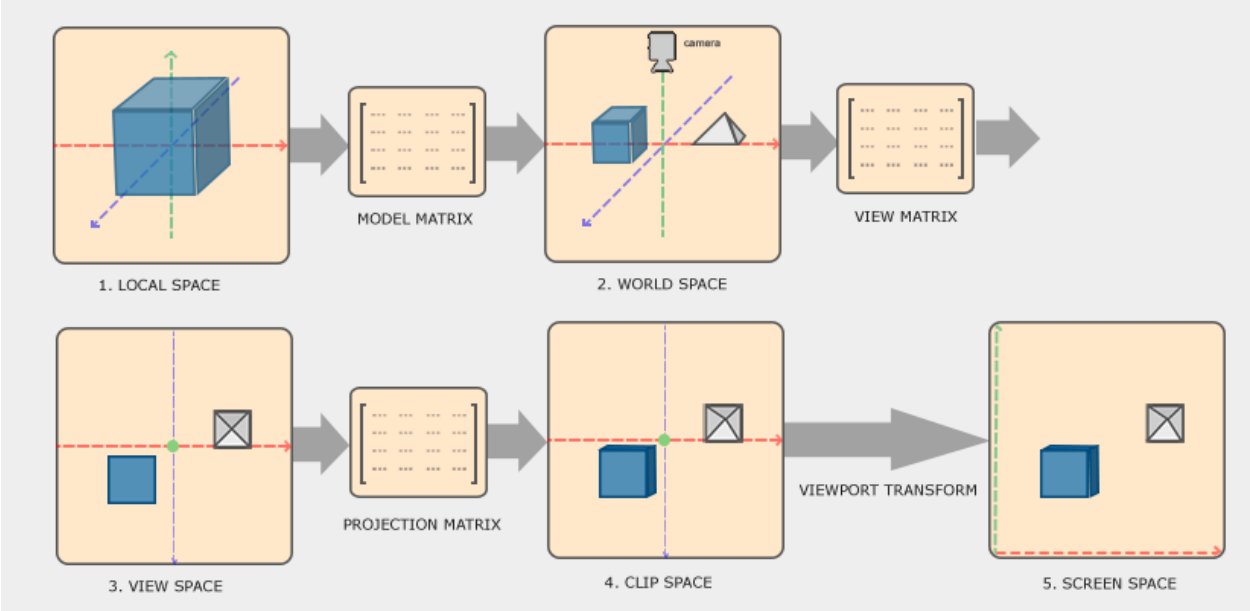


Рисунок 1.2 – Шлях вершин у візуалізаторі. Використано ресурс LearnOgl [1]

Пройдемось по кожній з трьох матриць більш детально:

- Матриця моделі: вона переносить координату с локальної системи у систему нашої програми(або систему світу). Це потрібно оскільки ми маємо змогу переносити, повертати та масштабувати модель в нашій програмі на наш розсуд. Тому модель буде мати своє розташування в нашому світі. Саме це переведення з оригінального розташування в наше робить матриця моделі.

- Матриця камери(також матриця зору): ми будемо мати змогу вільно рухатися камерою по нашому світу, тим самим ми маємо задавати значення розташування та повороту камери. Але у комп'ютерній графіці використовується наступна техніка щодо камери: оскільки проектувати кожен раз вершини на поверхню лінзи, яка буде мати довільну позицію та поворот у світі досить довго для вичислення. Ми не будемо рухати саму камеру, вона завжди буде у центрі світу $(0,0,0)$. Ми, навпаки, будемо рухати усі вершини інверсивно до трансформації камери. Тим самим матриця зору — інверсійна матриця трансформації камери у світі нашої програми.
- Матриця проекції. Вона має роль трансформувати наші вершини в так званий кліп простір. В реальному житті присутній ефект перспективної проекції: чим далі об'єкт, тим ближче він до центру горизонту. Ефект наочно видно на рисунку 1.3.

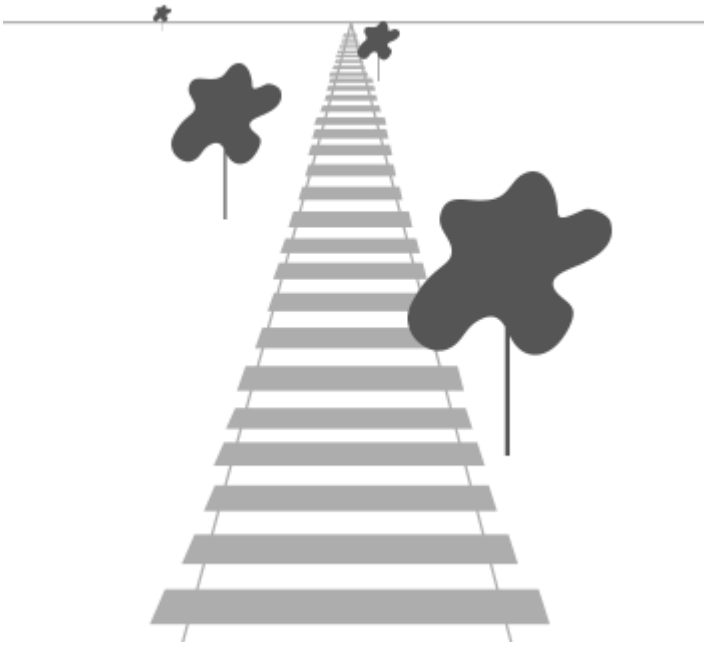


Рисунок 1.3 – Перспектива

Щоб зробити перспективну проекцію коректною, нам треба зробити низку обчислень. По-перше, ми застосовуємо до нашої вершини наступну матрицю:

$$\begin{bmatrix} \frac{1}{ar*\tan\frac{\alpha}{2}} & 0 & 0 & 0 \\ 0 & \frac{1}{ar*\tan\frac{\alpha}{2}} & 0 & 0 \\ 0 & 0 & \frac{-N-F}{N-F} & \frac{2FN}{N-F} \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad (1.7)$$

Компоненти формули 1.7 наступні:

- ar – aspect ratio або відношення сторін. Оскільки в системі координат дисплея кожна компонента знаходиться в діапазоні [0;1] ми повинні поділити x на значення відношення ширини до висоти, бо зазвичай відео девайси мають широкоформатний режим.
- α – вертикальний кут поля зору. Він визначає масштаб загального зображення.
- N – значення мінімальної дистанції с центру камери після якої ми починаємо малювати(де N – near plane)
- F – значення максимальної дистанції с центру камери після якої ми починаємо малювати(де F – far plane)

Оскільки для правильної формули ми ще повинні поділити кожен компонент на z, це буде зроблено вже самою відеокартою, бо це неможливо зробити за допомогою однієї матриці.

Тобто перспективна трансформація поділена на дві частини: за допомогою матриці проекції ми задаємо вершині такий стан, щоб потім, коли відеокарта зробить так зване ділення перспективи(perspective division), кожна компонента буде в діапазоні від [0;1], що дозволить нам знайти відповідний піксель для нашої вершини, а також відсіяти усі вершини, що не знаходяться в нашому діапазоні.

Закінчивши трансформацію вершин, ми передаємо їх по три на наступний етап, де вже будемо обробляти пікселі. Для цього спочатку растеризатор знаходить усі пікселі в трикутнику, а потів переносить їх у піксель шейдер.

1.2 Текстури

Текстури – це двовимірні масиви які використовуються задля надання поверхні властивостей з покращеною деталізацією. Як вже зазначалось вище, ми можемо передати які завгодно властивості поверхні через інформацію для кожної вершини, але оскільки після етапу растеризації ми маємо цілу купу пікселей в окремому трикутнику, їм залишається лише інтерполювати всю інформацію з трьох вершин. Потім володіючи нею задавати властивості, за допомогою яких ми отримуємо фінальний колір. Але щоб досягти якомога схожого до реальності зображення, дуже важливо мати можливість визначати властивості поверхні саме в окремому пікселі. Щоб до кінця зрозуміти роль текстур у візуалізації сцен, ми розглянемо найбільш поширені приклади:

- Карта базового кольору(Diffuse map) – найпримітивніша текстура яка визначає колір поверхні. Вона не несе у собі дифузну компоненту світла, тому не має в собі мати ніяких засвітів або затемнення. Більш детально буде розглянута у наступному розділі про світ та тінь.
- Карта відбиття(Specular map) – ще одна карта яка задає світлову властивість поверхні. А саме яка саме частина поверхні має властивість виблискувати у лінзу.
- Карта нормалі (Normal map) – за допомогою цієї текстури ми можемо деталізувати рельєф, задаючи нормалі для кожного пікселю поверхні.

Як вже було сказано, щоб зрозуміти звідки саме треба читати інформацію с текстур ми використовуємо текстурні координати, що є частиною вершин.

Це лише базові текстури які зазвичай використовуються у кожному візуалізаторі. Для фізично коректних властивостей поверхні вони можуть налічувати десятки.

Також треба відмітити, що за поняттям усіх рендер API, текстура – це двовимірний масив який зберігає кожен елемент за зазначеним форматом, та з якого ми плануємо провести операцію читання(лише читання, для написання існує інше поняття – рендер таргет). Текстури, що були перечислені вище, ми задаємо

вхідними параметрами для малювання моделі. Але ми також можемо передавати як вхідні параметри для окремого етапу візуалізації те, що ми намалювали у попередньому етапі. Це також буде вважатися текстурою. Більш детально ми будемо з цією властивістю працювати під час реалізації пост-процесів.

Тексель — примітив, піксель текстуры. Він несе в собі значення кольору. Зазвичай це значення є комбінацією чотирьох компонентів:

- R – червоний(red)
- G – зелений(green)
- B – синій(blue)
- A – значення напівпрозорості(alpha)

Треба зауважити, що розмір кожного компонента є варіаційним, ми можемо виділити по 8 байтів на кожен, а можемо й по 16. Від цього буде залежати діапазон можливих чисел для кожної компоненти кольору, тобто насиченість кольорів. Також часто роблять RGB компоненти більші за пам'яттю ніж A, аби заощадити пам’ять, бо напівпрозорість не обов’язково потребує великий діапазон чисел.

Але ми можемо використовувати цю пам’ять не лише для кольору. Як вже було зазначено вище, лише Diffuse map несе у собі поняття кольору.

Рисунок 1.4 – Карта базового кольору(зліва) та карта нормалі(справа)

Карта нормалі, як ви можете побачити на прикладі рисунку 1.4, несе в собі значення нормалі для кожного пікселя, синя там де нормаль ближче до (0, 0, 1), бо переводячи у формат RGB це буде компонента яка відповідає за синій колір — B. Також ми бачимо зелені частини там де нормаль наближається до значення (0, 1, 0), переводячи до RGB це зелений — G.

Також треба звернути увагу на одну із головні проблеми читання з текстури, та на методи їх рішення. Є вірогідність, що одна або всі поверхні моделі займатимуть невелику кількість екранного місця, бо знаходяться на далекій відстані від камери. Вони можуть займати навіть декілька пікселів екрану. Як же вирішити який саме колір читати з текстури, коли на один піксель на екрані, зразу можуть підійти більш ніж один тексель з текстури? Рішення складається з двох частин:

- Ми генеруємо заздалегідь набір тієї ж самої текстури, але меншої роздільної здатності. Для прикладу дивіться рисунок 1.5. Потім, в залежності від екранного місця поверхні, ми будемо читати с тієї текстури, яка сильніше всього буде близька до співвідношення: один тексель на один піксель. Це називається MIP-текстурування (MIP mapping) [2].

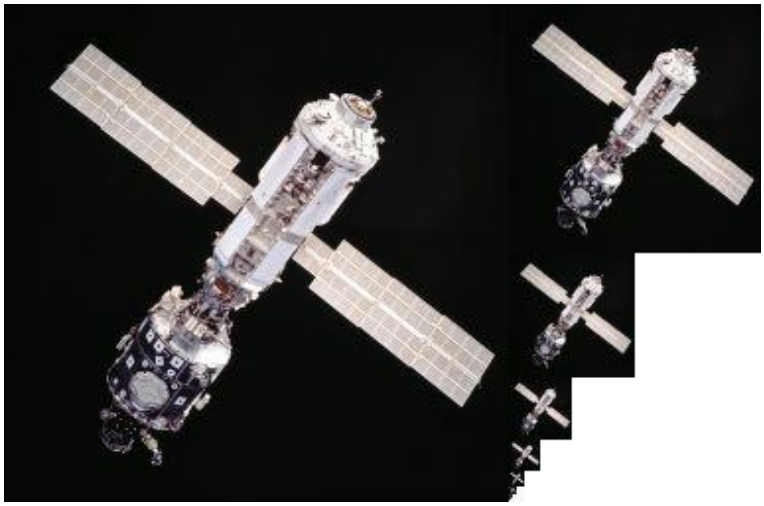


Рисунок 1.5 – Головна текстура та її зменшені копії.

- Навіть якщо ми використовуємо MIP-текстурування, все ще неможливо досягти ідеального співвідношення теклів текстури та пікселів екрану. Тому ми будемо вибирати колір використовуючи так зване фільтрування текстур: наприклад, ми будемо читати з того текселя, центр якого ближче всього до текстурної координати пікселя. Цей метод фільтрації називається nearest filtering(фільтрація за найближчим). Дивіться рисунок 1.6.

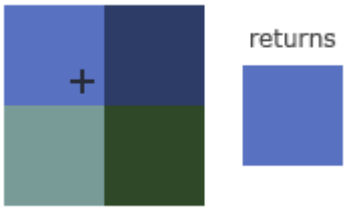


Рисунок 1.6 – Nearest filtering. Плюсом зазначена текстурная координата пікселю.

Або ми можемо інтерполювати колір, в залежності від близькості до центрів ближніх текселів. Цей метод називається linear filtering(лінійна фільтрація). Дивіться рисунок 1.7.

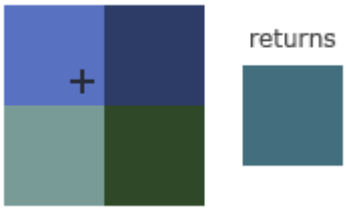


Рисунок 1.7 – Linear filtering.Плюсом зазначена текстурная координата пікселю.

Існує багато інших евристик, які можуть видавати якісніший результат. Але й виконувати більше обчислень. Тому потрібно зважити всі за та проти під час вибору метода фільтрування текстур.

1.3 Модель освітлення

Навіть намалювавши модель з правильною перспективною проекцією, а також наклавши на поверхні текстури, ми не отримуємо того 3D ефекту, який є хоча б наближеним до реальності.

Щоб досягти його, нам потрібно мати можливість відрізняти поверхні одна від одної. Саме це дасть нам світло та тінь. Світло є найважливішою частиною PBR рендерингу, воно є його базою.

Існує багато можливих моделей освітлення, вони визначають як саме буде впливати сума власного та відбитого випромінювання. Але для наших цілей я вибрав найбільш базову, але ефективну модель. Її розробниками є: в'єтнамський програміст Буі Туонг Фонг, а також Джим Блінн, який модернізував її, зробивши ще більш PBR направленою. Враховуючи авторів модель і дістала свою назву: модель відбиття світла Блінн-Фонг [3].

Значення освітлення задається наступними параметрами:

- Ambient factor – фактор навколишнього середовища. Оскільки в комп'ютерній графіці апроксимується поняття навколишнього середовища, ми вважаємо що повністю темних ділянок моделі не може бути, тому що світло відбивається від поверхонь та, хоча і с дуже малим впливом, але все ж освітлює навколишнє середовище. Тому це — мінімальний фактор, який ми множимо на колір поверхні. Зазвичай є константою не більше ніж 0.1.
- Diffuse factor – дифузний коефіцієнт. Він вже результатом потрапляння безпосередньо прямих променів світла на поверхню та їх дифузного (всебічного) поширення. Цей фактор вже більш вагомий та вираховується за допомогою нормалізованих векторів нормалі пікселя та вектора напрямку від пікселя до джерела світла. Скалярний добуток цих векторів і є дифузним коефіцієнтом. Формула для обчислення фактору може бути знайдена нижче (формула 1.8):

$$D = V_l \cdot N \qquad (1.8)$$

Де: D – дифузний фактор, V_l – нормалізований вектор направлення від координат пікселя до джерела світла, N – нормаль пікселя.

Треба зауважити, що як нормаль, так і вектор направлення повинні знаходитися в одній системі координат: світових нашої програми, або системи координат камери.

- Specular factor – коефіцієнт віддзеркалення. Найвагоміший фактор, бо він виражає в собі наскільки дзеркально відбиті щодо нормалі промені потрапляють у лінзу.

  Розраховується як скалярний добуток вектора, який знаходиться посеред вектора направлення від пікселя до нашої точки зору (лінзи) та вектором направлення від пікселя до джерела світла, а також вектором нормалі пікселя.

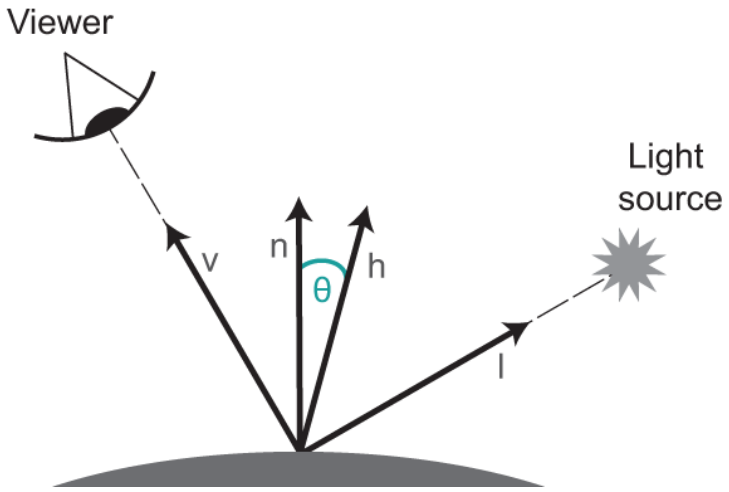


Рисунок 1.8 – Specular factor

Компоненти для обчислення фактору за рисунком 1.8: v – вектор направлення від пікселя до лінзи, n – вектор нормалі пікселю, l – вектор направлення від пікселя до джерела світла, h – вектор направлення посеред v та l, θ– кут між n та h.

Фінальний фактор віддзеркалення (формула 1.9):

$$S = v \cdot h \quad (1.9)$$

Де: S – фактор віддзеркалення.

Тому, скомбінувавши всі фактори разом результуючим кольором будемо мати наступну формулу 1.10:

$$C = T(A * D * S) \quad (1.10)$$

Де: C – фінальний колір, T – колір пікселя, заданий як трикомпонентний вектор RGB, A – фактор навколишнього середовища, D – дифузний коефіцієнт, S – фактор віддзеркалення.

1.4 Пост-процеси Bloom, Lens Flare

Bloom – це пост-процес який додає ореол світіння навколо яскравих ділянок зображення. Як саме це досягається?

По перше, ми повинні отримати інформацію які саме ділянки ми повинні обробити. Для цього ми створюємо окрему текстуру, в яку будемо записувати кожен піксель попереднього зображення, який має значення яскравості більше ніж задана константа.

Після цього ми повинні розмити результат, використовуючи, наприклад алгоритм Гаусового розмиття (рисунок 1.9): принцип його роботи в тому щоб спочатку розтягнути зображення горизонтально, а потім вертикально. Ці операції можуть повторюватися задля збільшення радіуса фінального ореолу сіяння.

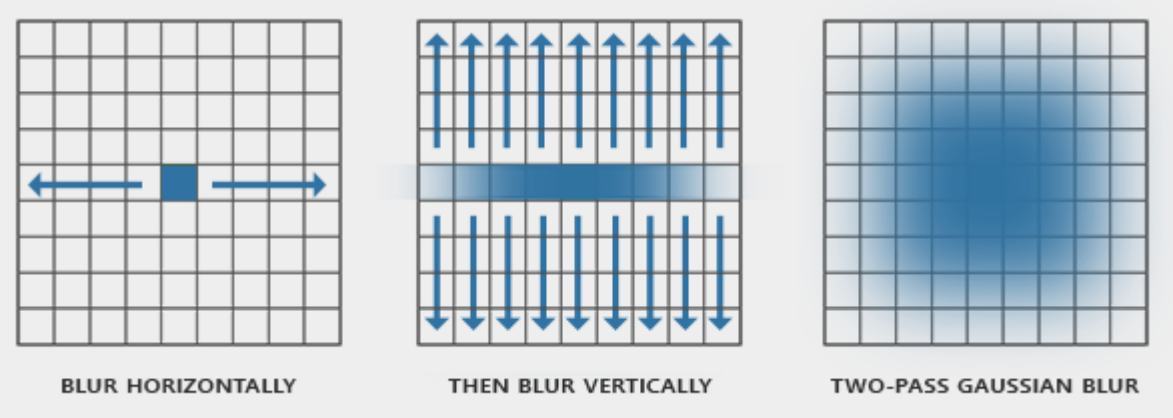


Рисунок 1.9 – Принцип Гаусового розмиття [1]

Після цього ми отримаємо результуючу текстуру, яку потрібно лише додаванням значень пікселів змішати з оригінальним зображенням [4]. Як результат будемо різницю, що можна побачити на рисунку 1.10.

Рисунок 1.10 – Приклад впливу ефекту світіння

Lens flare – ефект відблиску об’єктиву. Цей ефект з’являється на лінзах камери коли вони дивляться в сторону джерела світла.

Існують декілька способів реалізації цього пост-процесу. Один вираховує кожен візуальний артефакт динамічно, беручи до уваги позицію джерела світла, його дистанцію в світі, та багато іншого. Другий підхід простіший та примітивніший. Ми маємо заздалегідь сформовані текстури артефактів відблиску лінзи на рисунку 1.11.

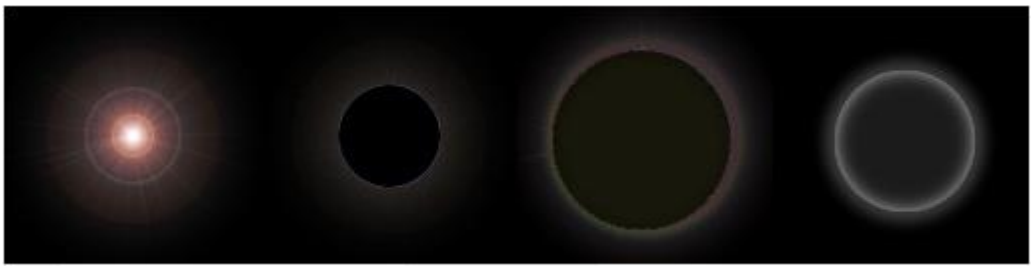


Рисунок 1.11 – Приклад набору текстур артефактів відблиску лінзи

Окрім цього нам потрібні будуть:

- Позиція світла у системі координат світу, або камери
- Константа центру екрана(може відрізнятися в різних рендер API)
- Матриця перспективної проекції

Використовуючи матрицю проекції ми трансформуємо положення нашого джерела світла до двовимірної системи координат екрану(не забуваючи про додаткове перспективне ділення). Потім розраховуємо направлення від джерела світла до центру екрану. Та через довільно задані проміжки малюємо текстури артефактів відблиску лінзи. Як саме це виглядає у просторі нашого екрану можна побачити на рисунку 1.12. Також, використовуючи магнітуду нашого вектора, ми можемо визначити додаткову яскравість з якою малювати текстури.

Також, за побажанням, попередньо можна використати Гаусове розмиття.

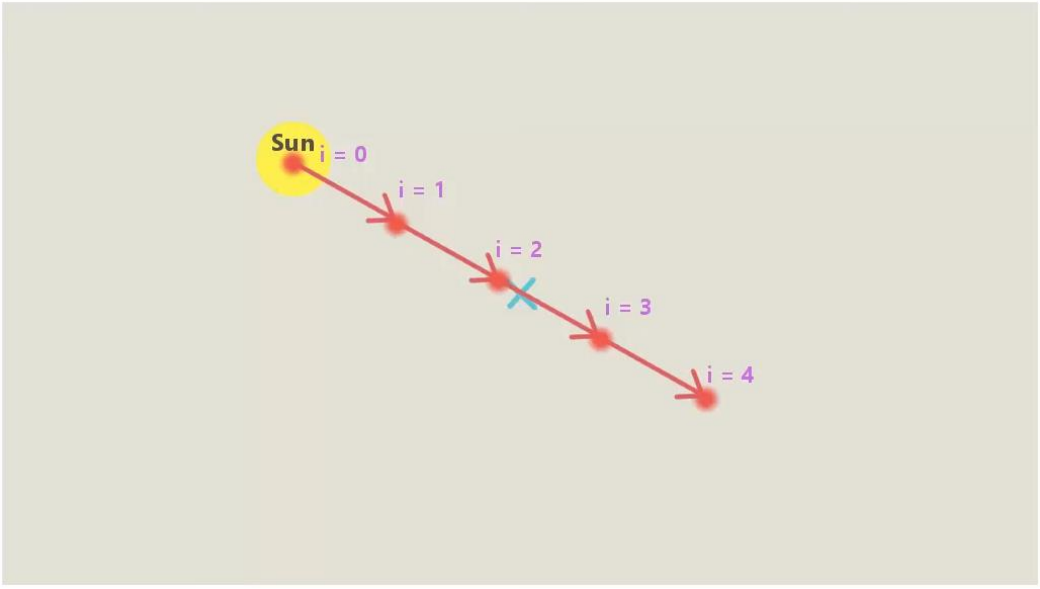


Рисунок 1.12 – Приклад кроку між кожною текстурою

1.5 Пост-процеси MSAA, Tone mapping

При візуалізації 3D сцен ми стикаємося з проблемою зубців на межах кривих ліній. Вони виникають, тому що примітив, який ми замальовуємо — прямокутний піксель. Тому під час малювання протяжної кривої лінії ми будемо бачити сходинки з прямокутників.

Задача ж Anti Aliasing методів прибрати цю різкість за допомогою згладжування цих зубців. На рисунку 1.13 видно цю різницю.

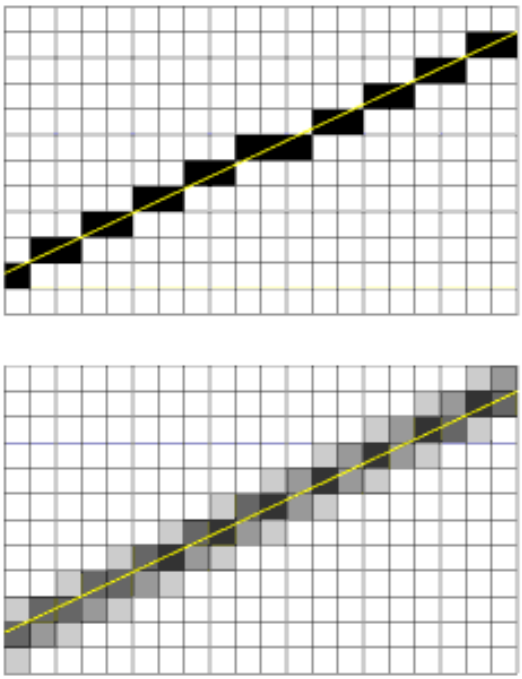


Рисунок 1.13 – Проблема зубців(зверху) та її рішення за допомогою згладжування(нижній малюнок)

То як саме виконати це згладження? Існує багато методів, які використовують евристику для знаходження границь об'єктів, та згладжують окремо їх(такий метод називається FXAA), або використовують буфери текстур, які зберігають історію попередніх декількох кадрів, та інтерполюють значення між ними(цей метод носить назву TAA). Але ми будемо розглядати більш прямолінійний спосіб.

Ми спеціально продублюємо наше намальоване зображення у роздільній здатності в два рази більшу за оригінальне. Потім, використовуючи лінійний фільтр текстур(див розділ про текстури), ми будемо утворювати наше зображення ще раз в оригінальному розмірі, але читаючи дані з зробленої текстури більшого розміру. Це автоматично згладить усі гострі криві лінії. Хоча цей метод не є найефективнішим, він все ще дає досить гарний результат та має назву MSAA(Multisample anti-aliasing).

Tone mapping — це процес перетворення вихідного діапазону кольору нашого зображення з формату RGB, розмірність якого ми задали у рендер API, в діапазон, який є прийнятним для нашого відео дисплею.

Зазвичай, це важливо, якщо дисплей підтримує так званий широкий діапазон кольорів(HDR), тобто на кожний компонент кольору RGB ми маємо досить багато виділеної пам’яті. Це дає нам змогу додати до зображення насиченості.

Вплив HDR на зображення можна побачити на рисунку 1.14.

Рисунок 1.14 – Зліва ми бачимо вплив на насиченість кольору з підтримкою HDR

1.6 Пост-процес SSR

SSR(screen space reflection) — відбивання в просторі екрану. Цей ефект надає нам можливість задати поверхні властивості дзеркальності. Саме вона робить поверхні схожими на гладкі та/або вологі.

Задля того, щоб визначити який саме піксель буде відзеркалено використовується алгоритм підбору Ray Marching. Ми рахуємо завдяки нормалі відбиття для кожного пікселю заданої поверхні. Потім, маючи відбитий промінь, ми переносимо його у систему координат камери та почитаємо ітерації по збільшенню його магнітуди, а саме: збільшуємо його довжину на задане константне значення, переводимо цей промінь назад до двовимірної системи координат екрану, перевіряємо використовуючи карту глибини, чи не перекривається кінець нашого променя якимось з пікселів. Якщо ні, то ми знов збільшуємо магнітуду, та знову перевіряємо перекриття, якщо ж так — ми знайшли потенціальний піксель, якій буде зображений на дзеркальній поверхні.

Враховуючи вищезазначений алгоритм, нам знадобиться наступна інформація:

- Карта позицій вершин у просторі камери. Щоб вирахувати стартове положення променя відбиття.
- Карта глибини. Для порівняння перекриття променю.
- Матриця трансформації з двовимірної системи координат екрану у систему координат камери.
- Матриця трансформації з системи координат камери в двовимірну систему координат екрану.

2 ВИБІР ЗАСОБІВ РОЗРОБКИ

2.1 Мови програмування

Мовою написання програми було обрано C++ 14-ого стандарту.

Це мова програмування високого рівня, яка зразу підтримує кілька парадигм програмування: об'єктно орієнтована, узагальнена та процедурна. Загалом, це всесвітньо прийнята мова для розробки швидкодіючого та ресурсо-економічного програмного забезпечення.

Також C++ має глибоку інтеграцію з усіма видами рендер API.

Середою розробки було обрано Visual Studio 2017 як найбільш зручну середу для роботи з C++.

Для програмування шейдерів використовується HLSL. Причина цього вибору — рендер API, яке ми використовуємо, а саме DirectX 11. Воно підтримує нативно лише HLSL.

Для налаштування зручної системи генерації проекту використовується мова скриптів Lua.

2.2 Система генерації проекту. Система контролю версій

Система генерації проекту потрібна задля того, щоб в залежності від системи, на якій буде запускатися програма, ми мали змогу генерувати відповідний файл проекту для Visual Studio 2017. Це скорочує час налаштування проекту для нових систем.

Для генерації проекту використовується Premake5. Ця система подібна до всесвітньо визнаної системи CMake, але менш завантажена та, що найголовніше, використовує мову скриптів Lua замість окремої розробленої мови CMake. Тому саме із за легкості користування та порівняно невеликого масштабу проекту мій вибір припав на Premake5. Також для можливості запуску написаного мною

генератора проекту однією кнопкою я написав примітивний BATCH скрипт ОС Windows.

Системою контролю версій обрано GitHub. Це безкоштовна відкрита система, яка допомагає зберігати кожну ревізію програми окремо в хмарі. Це дає можливість контролювати процес розробки, та мати можливість відкотитися до потрібної версії програми у разі потреби. А також дає можливість, за допомогою історії ревізій, побачити весь хід робот.

2.3 API для рендеру

API для рендеру — це набір функціоналу для:

- Виділення відео пам’яті(пам’яті, яка знаходиться біля відеоадаптера).
- Налаштування стану, за яким буде оброблятися інформація на кожному етапі створення зображення: налаштування растеризатору пікслів, налаштування шаблону читання інформації з вершин, налаштування шейдерних програм та багато іншого.
- Посилання команд на виконання з CPU на GPU.

Оскільки програма розробляється на ОС Windows, то найбільш зручним та швидкодійним API є DirectX 11. Вибір був з трьох кандидатів:

OpenGL, DirectX 11 та DirectX 12. Оскільки OpenGL вважається вже застарілим, а DirectX 12 потребує досконалого ручного контролю виділення ресурсів на відеопам'яті, а також прямого контролю відправлення команд на GPU, було вирішено використовувати DirectX 11.

Використовується найновіша(на час написання роботи) версія SDK DirectX.

2.4 Сторонні бібліотеки

Роль сторонніх бібліотек в тому, щоб зберегти час під час розробки, частково використовуючи вже готові рішення. Це зазвичай стосується окремих модулів

програми, та, зазвичай, не є частиною її архітектури, а лише способом її імплементації.

У C++ сторонній код може бути доданий у вигляді:

- Статичні бібліотеки — бінарний файл, який містить у собі імплементацію якогось функціоналу. При створенні файлу виконання, усі статично зв’язані с програмою бібліотеки будуть частиною цього файлу. .lib — розширення статичних бібліотек.
- Динамічні бібліотеки — те ж саме що й статичні, але вони не будуть частиною файлу виконання, натомість вони будуть знаходитися завжди в окремих файлах, та будуть вимагати читання з цих файлів під час виконання роботи програми. З одного боку це надає нам можливість зменшити розмір фінального файлу виконання, а з іншою — буде потребувати додаткові обчислення в режимі реального часу(бо нам потрібно буде взаємодіяти з окремими файлами). .dll — розширення динамічних бібліотек.
- Вихідний код — просто файлів коду до нашого проекту. Найпростіший спосіб, але вимагатиме кожний раз компілювати більше коду.

Для компіляції шейдерів, програм, які будуть виконуватися на GPU, використовується DirectXShaderCompiler. Код компілятору відкритий, та може бути знайдений на GitHub.

Використовуючи дебаг мод під час компіляції HLSL коду, ми будемо мати змогу бачити які саме помилки були допущені, та як їх можна виправити. Це прийнятий більшістю компілятор і він йде в базовому пакеті Microsoft.

Для обробки вікна, незалежно від системи, ми будемо використовувати SDL 2.0. Ця бібліотека бере на себе налаштування вікон, використовуючи вхідні параметри користувача. Вона зчитування інформації щодо подій введення(input) зі сторони користувача, та обробляє частину з них.

Для виконання математичних розрахунків використовується DirectXmath — також бібліотека зі стандартного пакета, яка дозволяє робити обчислення за допомогою SIMD операцій. Це спеціально адаптовані операції для математичного

обчислення, що використовується у візуалізаторі. Вони взаємодіють одночасно з великим обсягом вхідних дани. Навіть можуть робити операції над якнайбільш чотирма векторами одночасно. Бібліотека також несе у собі весь необхідний функціонал для будування матриць трансформації та матриць проекцій.

WICTextureLoader11 — бібліотека для налаштування та конвертації текстур з формату зображення у формат, придатний для користування нашим рендер API. Працює з такими форматами зображення як: BMP, JPEG, PNG.

DDSTextureLoader11 — також бібліотека для налаштування текстур, використовуючи вхідне зображення, але з підтримкою лише DDS(DirectDraw Surface) формату, який розроблений спеціально для роботи з DirectX.

Обидві ці бібліотеки допомагають вказувати нашому рендер API як саме ми повинні читати з текстури, яку кількість пам’яті займає один елемент текстури, та багато іншого.

tinyobjloader — бібліотека, яка дозволить нам імпортувати моделі формату OBJ, саме за допомогою неї ми будемо складати масиви з інформацією щодо вершин та масиви індексів вершин.

3 ІМПЛЕМЕНТАЦІЯ ВІЗУАЛІЗАТОРА 3Д СЦЕНЫ

3.1 Налаштування середовища

Спочатку ми створили відповідний репозиторій на GitHub. Саме тут буде зберігатися наш код та всі його ревізії з відповідною історією змін. Сторінка репозиторію зображенна на рисунку 3.1.

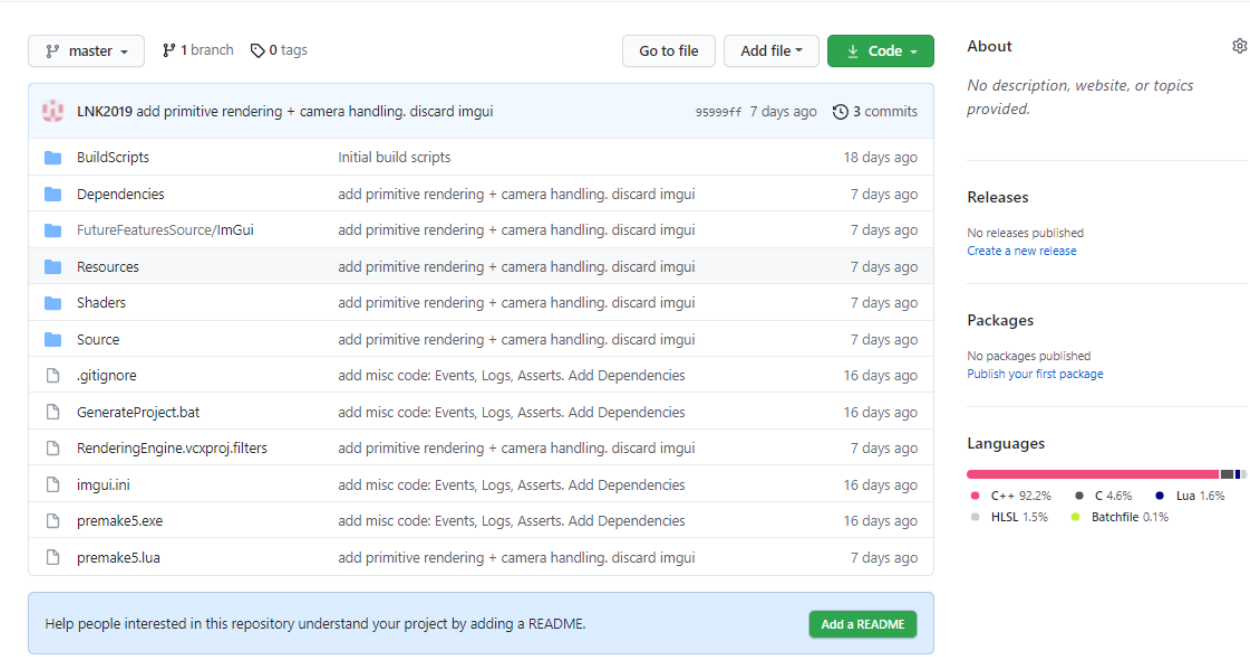


Рисунок 3.1 – Репозиторій з нашим кодом

Треба зауважити, що ми також додали .gitignore файл, який є фільтром того, що не треба додавати до репозиторію та які файли є тимчасовими для кожного користувача коду. Там знаходяться заборони на додавання: файлів Visual Studio(вони будуть автоматично генеруватися для кожного користувача коду), бінарні вихідні файли програми, а також проміжні файли які використовує C++ під час компіляції. Усе це зроблено, аби файли, що створюється для кожного користувача, не засмічували репозиторій.

Для автоматичної генерації проекту ми створили скрипт для Premake.

Він описує усі залежності проекту до стороннього коду, налаштовує шляхи для вихідних файлів нашого проекту, а також вказує якого типу проект створити для Visual Studio. Скрипт прикріплено у Додатку А.

Також додали BATCH скрипт, який буде запускати Premake. Скрипт зображенно на рисунку 3.2.

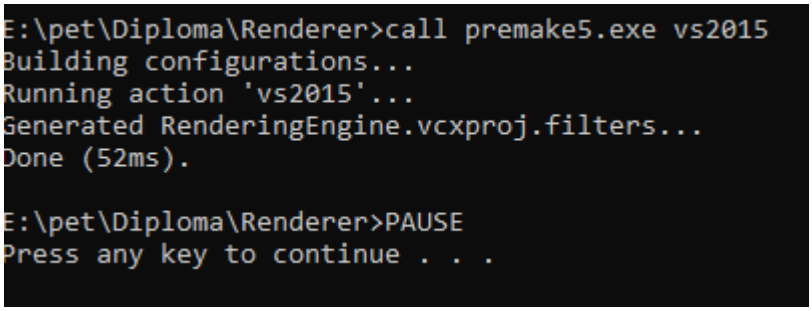


Рисунок 3.2 – Результат виконання нашого скрипту через BATCH файл

Після того, як ми маємо змогу у будь-який час та на любій системі отримати доступ до нашого репозиторію, та генерувати відповідний проект для кожного середовища розробки, ми розпочинаємо написання програмного коду.

3.2Архітектурне рішення

Для того, щоб мати уявлення про структуру програми, ми склали UML діаграму співвідношення основних модулів:

- Application — клас, у якому ми починаємо виконання програми. Його завдання налаштувати зв’язок дій користувача та реакції програми на ці дії. Також саме за допомогою цього класу ми задаємо параметри нашому вікну. Зазвичай, в цьому класі присутній код, який у нескінченному циклі оновлює вікно та перевіряє чи настав час закривати програму.
- Window — вікно, в якому ми збираємося малювати наше зображення. Саме цей клас відповідає за будь-які дії з вікном. Та є частиною обгорткою SDL функціоналу. Він володіє менеджером ренреду, що дає

нам змогу вказувати останньому в яке саме вікно треба наносити розраховане зображення.

- Renderer — основний клас, якій відповідає за збір усіх команд, що будуть відправленні на GPU. Його задача правильно створити стан рендер API, а потім по черзі збирати всю необхідну інформацію, та надсилати її відеоадаптеру, тим самим створюючи ланцюг, кожен етап якого буде виконувати необхідні махінації з зображенням.
- Input manager — функціонал, який дозволить нам встановити зв’язок реакції програми на дії користувача.
- Scene — більш структура, аніж клас(за поняттями C++), яка зберігає у себе всю інформацію 3D сцену: об’єкти, їх вершини, їх трансформації (розташування, поворот, масштабування), інформація про камеру, світло і багато іншого.
- Shader Manager — набір програм шейдерів які, в залежності від етапу ланцюгу малювання, обробляють вхідні дані вершин та матеріалів. Цей клас також відповідальний за компіляцію шейдерів в реальному часі.

Application, Window та Input Manager — це допоміжні класи, які не відповідають за сам процес створення зображення на екрані. Їх завдання бути обгорткою, яка має допомагати налаштовувати процес рендерингу. Так, Application і Window відповідають лише за стан та розмір вікна, а Input Manager за стан матриці трансформації камери.

Загальну архітектуру програми зображено на рисунку 3.3.

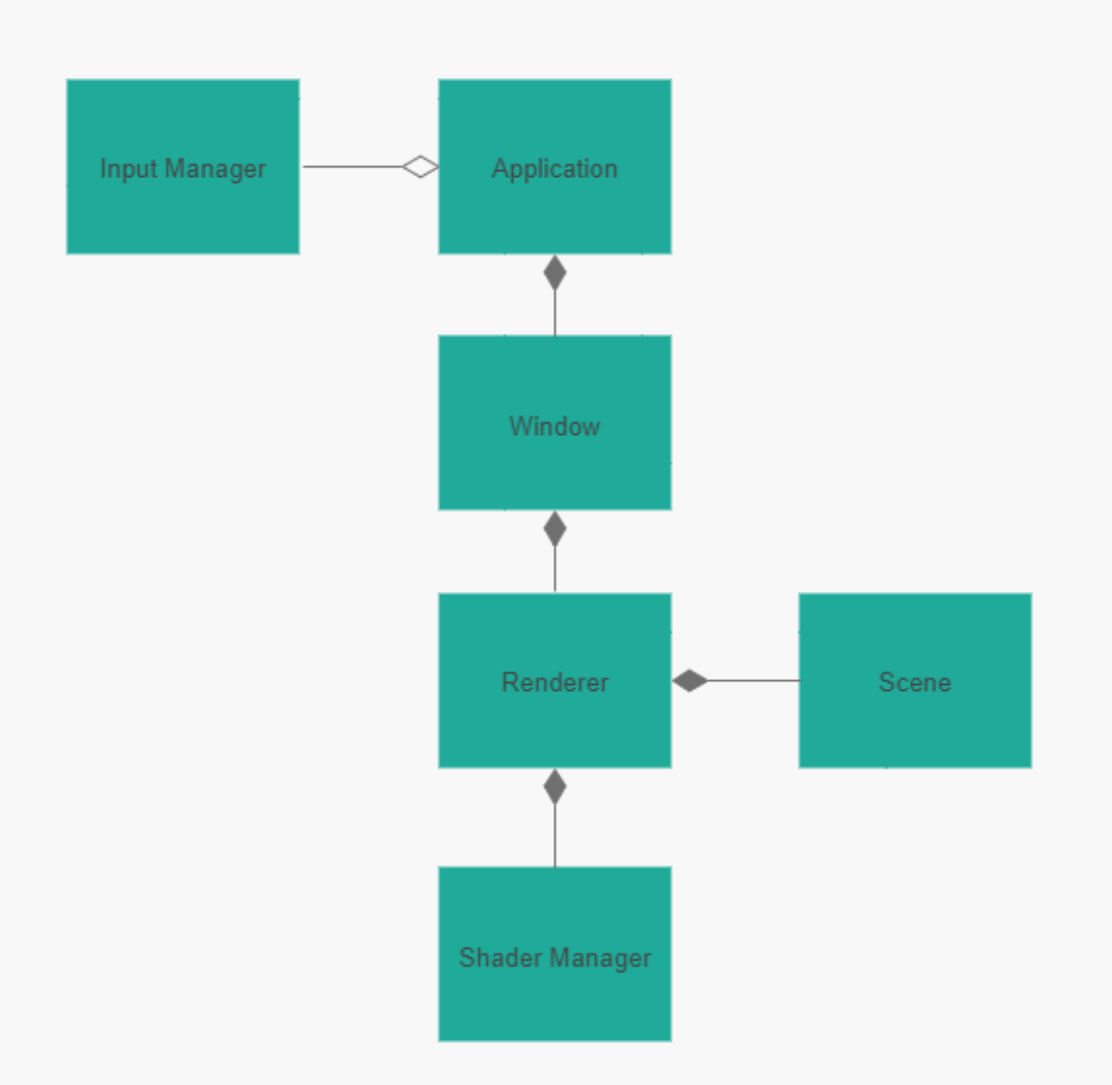


Рисунок 3.3 – UML діаграма основних мудулів нашої програми

Розглянемо більш детально три найголовніші частини архітектури: Renderer, Scene, Shader Manager.

Renderer. Цей клас є сполучною ланкою між набором даних та шейдерним кодом, який буде ці дані обробляти. Також саме він напряму взаємодіє з рендер API DirectX 11. В цьому класі ми збираємо до купи: усі наші моделі на сцені, шейдерний код для кожного з етапів рендеру, інформацію щодо налаштування вікна. Ми Використовуємо ці всі дані, щоб відправити їх та обробити на відеокарті. Також саме тут ми викликаємо процесс малювання нашого зображення. Декомпозуємо більш детально:

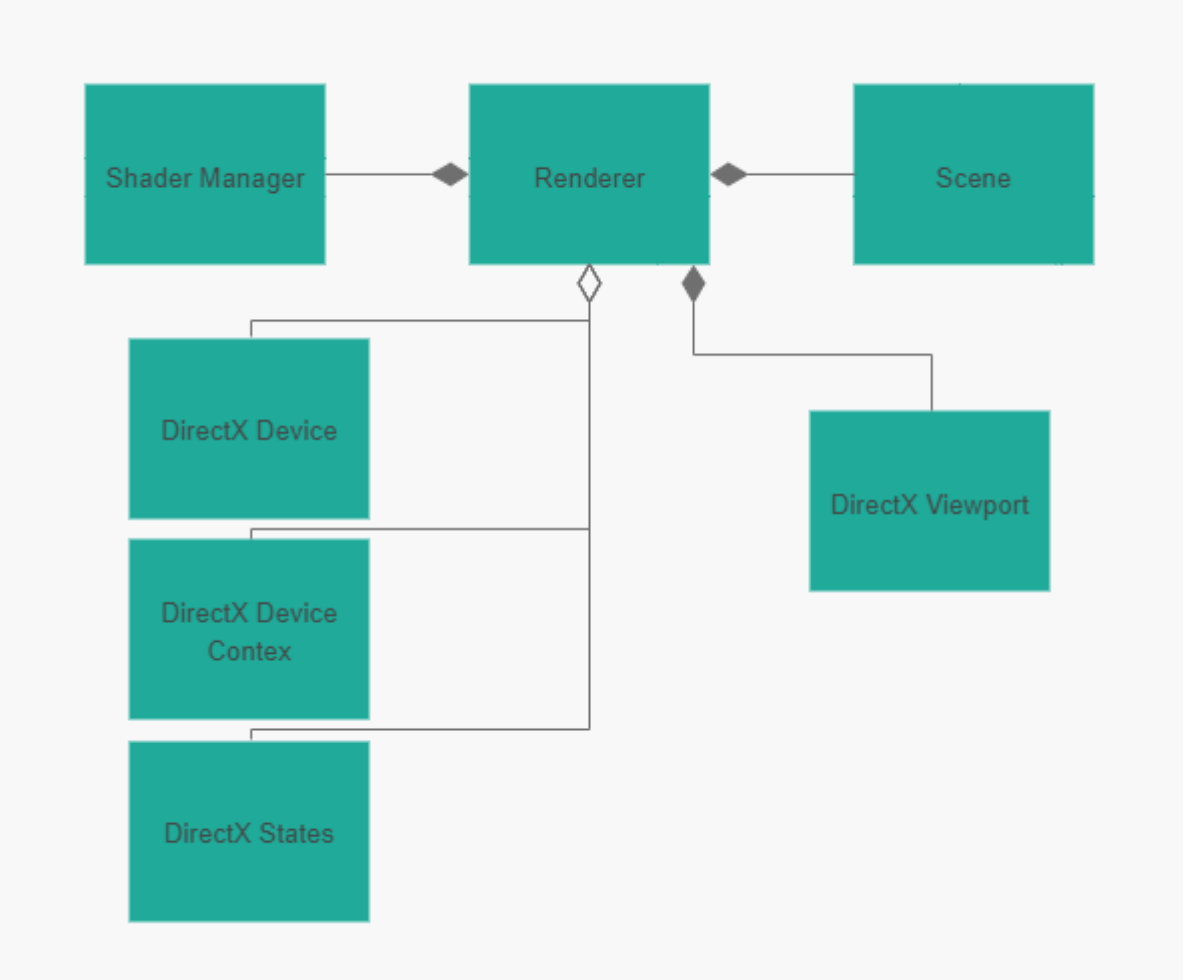


Рисунок 3.4 – UML діаграма класу Renderer

Як можна побачити на рисунку 3.4, крім класу сцени та менеджеру шейдерів, Renderer також володіє вказівниками на, створені при його ініціалізації, об’єкти DirectX 11. Пристрій та контекст пристрою(Device, Device context) — це саме ті об’єкти, які будуть відправляти команди на GPU та налаштовувати стан ланцюга малювання. Саме набір цих станів можна отримати за допомогою DirectX States. DirectX Viewport визначає розмір і положення намальованого зображення на екрані(майже завжди буде тим що й відповідні значення у вікна), а також визначає мінімальний та максимальний рівень глибини яку ми можемо задати в нашому зображенні(за рівень глибини відповідає z координата вершини у просторі екрану).

Scene. Завдання сцени зберігати та оновлювати дані щодо: моделей, їх позиції у світі, камери та її позиції у світі, світла, та його позиції у світі.

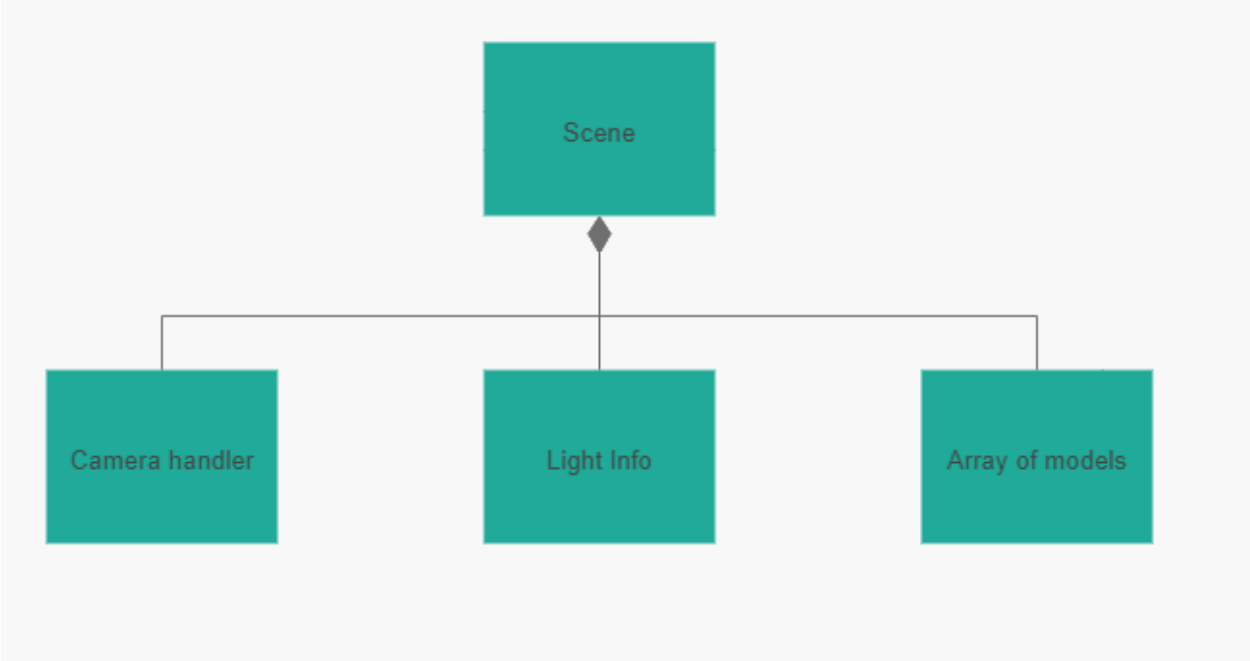


Рисунок 3.5 – UML діаграма класу сцени

Дивлячись на рисунок 3.5 ми маємо змогу оцінити, за що саме відповідає сцена. Camera handler — клас, який дає нам змогу змінювати положення камери: повертати її та змінювати позицію. А також цей клас відповідальний за формування матриці трансформації у систему координат камери(див. теоретичний розділ щодо систем координат).

Light info — клас, що визначає позицію світла у світі, а також його колір.

Array of models — масив моделей. Кожна модель зберігає в собі відповідні текстури(колір, нормаль тощо), та масив інформації про вершини цієї моделі.

Треба зауважити, що саме клас сцени буде відповідати за імпортування в нашу програму текстур та моделей.

Shader Manager. Цей клас займається наступним: компілює всі шейдера, що будуть використовуватися під час виконання програми, за заданим шаблоном встановлює зв’язок між вихідними даними зі сцени та вхідними даними кожного шейдеру(по запиту Renderer’а). Також він зберігає вказівники на скомпільований код шейдерів у пам’яті.

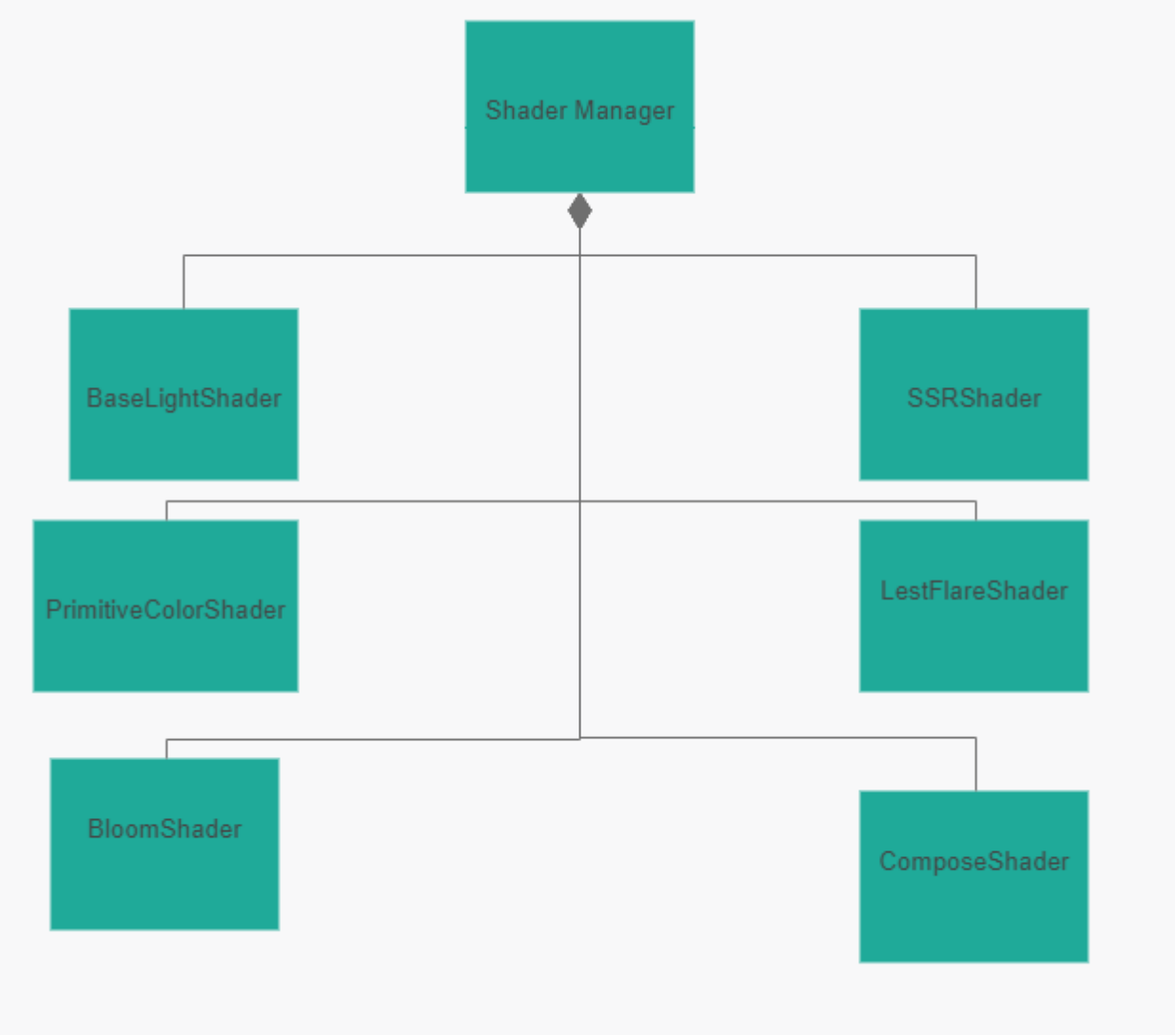


Рисунок 3.6 – UML діаграма менеджеру шейдерів

Як бачимо на рисунку 3.6, ми маємо доступ до бінарного коду всіх шейдерів, що будуть використані під час малювання.

Це був опис того, як загалом виглядає архітектура програми. Звичайно, ми ще додали багато допоміжного коду, але він не є таким вагомим. Це стосується логіки дебагу програми, логіки написання логів, деякої логіки математики.

3.3 Налаштування 3D об’єктів та відповідних матеріалів

Цей розділ покаже як саме, використовуючи архітектуру, задану вище працює процес налаштування та малювання об’єкту в нашій програмі. Для нашого прикладу було взято модель чайнику.

По-перше, імпортуємо наш obj файл: зчитуємо дані за допомогою tinyobjloader, та зберігаємо у зручному для нас форматі, в якому вони будуть передані на GPU. Часткове налаштування можна побачити за рисунком 3.7.

```
VertexDataType Vertex1;
Vertex1.Position.x = Position1.X;
Vertex1.Position.y = Position1.Y;
Vertex1.Position.z = Position1.Z;

Vertex1.Normal.x = Normal1.X;
Vertex1.Normal.y = Normal1.Y;
Vertex1.Normal.z = Normal1.Z;

Vertex1.UV = TextureCoord1;
```

Рисунок 3.7 – Model.cpp: ділянка коду. де ми створюємо одну з вершин, використовуючи раніше імпортовану інформацію

Потім, під час етапу, коли рендерер вже відправив на GPU інформацію щодо того, який шейдер повинен виконуватися, ми співвідносимо дані вершин, які ми маємо, та дані які отримує шейдер для обробки:

```
//Setup vertex layout
HRESULT Result;
D3D11_INPUT_ELEMENT_DESC VertexLayout[3];

VertexLayout[0].SemanticName = "POSITION";
VertexLayout[0].SemanticIndex = 0;
VertexLayout[0].Format = DXGI_FORMAT_R32G32B32_FLOAT;
VertexLayout[0].InputSlot = 0;
VertexLayout[0].AlignedByteOffset = 0;
VertexLayout[0].InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;
VertexLayout[0].InstanceDataStepRate = 0;

VertexLayout[1].SemanticName = "NORMAL";
VertexLayout[1].SemanticIndex = 0;
VertexLayout[1].Format = DXGI_FORMAT_R32G32B32_FLOAT;
VertexLayout[1].InputSlot = 0;
VertexLayout[1].AlignedByteOffset = D3D11_APPEND_ALIGNED_ELEMENT;
VertexLayout[1].InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;
VertexLayout[1].InstanceDataStepRate = 0;

VertexLayout[2].SemanticName = "TEXCOORD";
VertexLayout[2].SemanticIndex = 0;
VertexLayout[2].Format = DXGI_FORMAT_R32G32_FLOAT;
VertexLayout[2].InputSlot = 0;
VertexLayout[2].AlignedByteOffset = D3D11_APPEND_ALIGNED_ELEMENT;
VertexLayout[2].InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;
VertexLayout[2].InstanceDataStepRate = 0;
```

Рисунок 3.8 – Renderer.h: етап створення шаблону зчитування масиву даних, що буде відправлений на GPU, як вхідні дані для шейдеру

Далі, ми розраховуємо всі необхідні матриці трансформацій, й також відправляємо їх на GPU, щоб вони були використані для кожної вершини на етапі вершинного шейдеру, але перед цим створивши шаблон читання даних(рисунок 3.8).

Після усіх інших налаштувань, не пов’язаних з моделлю, ми використовуємо примітивний піксельний шейдер який просто замалює білим усі пікселі, які відносяться до нашої моделі. Як результат:

Рисунок 3.9 – Результуюче зображення, використовуючи примітивний піксельний шейдер. Позиція камери — (0, 0, -10)

Далі, ми повинні накласти текстуру кольору на наш чайник. Для цього ми завантажимо спеціальний малюнок, який відповідає текстурі кольору чайнику.

Імпортуємо його використовуючи WICTextureLoader11. Потім відправляємо на етап піксельного шейдера, щоб саме звідси, використовуючи текстурні координати, ми брали інформацію кольору пікселів: для цього, за поняттями DirectX 11 ми створюємо так званий Shader Resource View, або SRV. Це об’єкт,

який буде асоціюватися з нашим зображенням на GPU. Шейдер може тільки зчитувати з нього, написання — заборонено.

Рисунок 3.10 – Текстура кольору для нашого чайнику

Щоб зв’язати наш SRV з піксельним шейдером, ми для нього викликаємо наступний код: *D3DDeviceContext->PSSetShaderResources(0, 1, &SRV);*

Це зв’яже нашу текстуру SRV с піксель шейдером, який на даний момент часу є активним. Тепер, ми зчитуємо колір для пікселя з SRV. Зображення з впливом текстурування можна побачити на рисунку 3.11.

Рисунок 3.11 – Зображення, з використанням текстури кольору

3.4 Реалізація моделі освітлення

Як вже було зазначено в теоретичній часті, задля реалізації моделі Блінн-Фонга нам потрібно розрахувати 3 коефіцієнта, сума яких буде множитись на значення кольору пікселя. Тим самим чим вище буде значення суми — тим яскравішим буде піксель.

Обчислення робляться для кожного пікселю окремо в піксель шейдері.

Почнемо з першого коефіцієнту: ambient factor або фактор навколишнього середовища. Тут відсутні будь-які розрахунки, ми просто будемо вважати що по замовчуванню ми завжди бачимо кожен піксель моделі, хоч і з дуже невеличкою яскравістю. Наприклад, в нашому випадку задаємо значення цього фактору як 0.2 та множимо на колір. HLSL код наступний:

```
const float AmbientFactor = 0.2f;
const float3 AmbientColor = AmbientFactor * PixelColor;
```

Рисунок 3.12 – Ambient фактор освітелння

Вище на рисунку 3.12 можна побачити, як саме виглядає модель, освітлена лише фактором навколишнього середовища.

Наступний крок додати фактор дифузного розсіювання променів. Як вже було зазначено в теоретичній частині, рахується він за допомогою нормалі пікселя, та направлення до джерела світла. HLSL код виглядає наступним чином:

const float3 LightDir = normalize(LightPosition - PixelPosition);

const float3 Normal = normalize(input.normal);

const float DiffuseFactor = max(dot(LightDir, Normal), 0.0);

*const float3 DiffuseColor = DiffuseFactor * PixelColor;*

Нижче на рисунку 3.13 можна побачити вплив фактора навколишнього середовища та фактору дифузного розсіювання разом.

Рисунок 3.13 – Ambient та Diffuse фактори освітелння

Та фінальний фактор дзеркального відбиття світла — specular factor. Він відповідає за найяскравіші ділянки поверхні. Які саме це ділянки, розраховується за допомогою вектора, який знаходиться між направленням до джерела світла та направленням до точки зору і вектора нормалі пікселя. Важливо, що цей фактор множиться на білий колір, а не колір поверхні. HLSL код наступний:

const float3 WhiteColor = float3(1.f, 1.f, 1.f);

const float3 ViewDir = normalize(ViewPosition - input.position);

const float3 HalfwayDir = normalize(LightDir + ViewDir);

const float SpecularFactor = pow(max(dot(Normal, HalfwayDir), 0.f), 32.0);

*const float3 SpecularColor = WhiteColor * SpecularFactor*;

Рисунок 3.14 – Ambient, Diffuse та Specular фактори освітелння. Фінальний вигляд моделі освітлення Блінн-Фонг

На рисунку 3.14 можна побачити фінальний результат впливу моделі освітлення Блінн-Понг на нашу сцену. Результуючий колір для окремого пікселя розраховується як сума усіх факторів помножених на кольор нашого пікселя. Код наступний:

const float4 FinalColor = float4(AmbientColor + DiffuseColor + SpecularColor, 1.f);

Де кожен із кольорів – це значення кольору помноженого на відповідний фактор.

FinalColor – це фінальний колір нашого пікселю.

3.5 Реалізація пост-процесів

Спочатку додаємо підтримку згладжування MSAA. DirectX 11 підтримує автоматичне використання цього методу згладжування на етапі растеризатора, що не потребує імплементації з нашої сторони. Достатньо при ініціалізації стану растеризатору у Render.cpp вказати наступне налаштування:

RasterDesc.AntialiasedLineEnable = true;

RasterDesc.MultisampleEnable = true;

Де RasterDesc — це об’єкт стану растеризатору D3D11_RASTERIZER_DESC.

На рисунку 3.15 можна побачити яку саме проблему вирішує згладжування: зубці на кривих лініях.

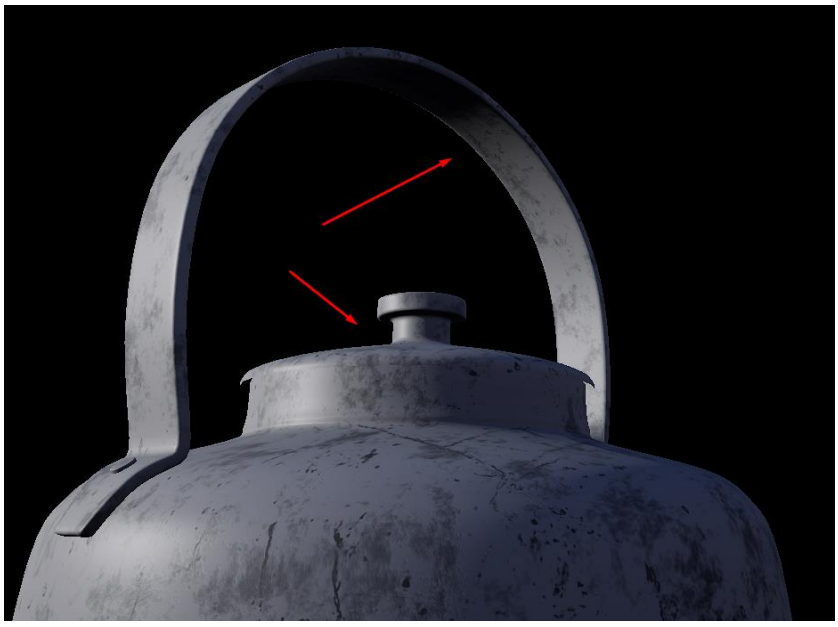


Рисунок 3.15 – Проблема Aliasing’у

На рисунку 3.16 результат виконання MSAA. Криві лінії тепер більш гладкі. Але також треба розуміти що це потребує значного використання пам’яті в реальному часі. Тому що ми зберігаємо текстури більшої роздільної здатності(більш детально про це розповідається в теоретичному розділі).

Рисунок 3.16 – Результат MSAA

Наступний пост-процес, який ми імплементуємо буде ореол світіння bloom.

Спочатку ми повинні створити текстуру, в яку запишемо тільки ті пікселі нашого малюнку, які перевищують задану константу яскравості *Threshold*. Для цього ми повинні створити окремий піксель шейдер, який має виконувати наступну перевірку:

const float IntensityTest = (float)(length(PixelColor.rgb) > Threshold);

*FinalColor = float4(IsntensityTest * PixelColor.rgb, 1.0);*

Таким чином значення *IntensityTest* буде мати стан 1 або 0, в залежності від чого піксель буде мати оригінальний колір, або значення (0, 0, 0, 1).

Далі, використовуючи алгоритм Гаусового розмиття(дивитися в теоретичних відомостях), ми отримуємо розмите зображення найяскравіших пікселів. Об’єднуємо його с оригінальним зображенням та отримуємо результат:

Рисунок 3.17 – Зображення без ефекту ореолу світіння

Рисунок 3.18 – Зображення з ефектом ореолу світіння

Наступним ми імплементуємо відблиски лінзи Lens Flare. Як вже було зазначено в теоретичній частині ми будемо мати набір текстур артефактів лінз, які будуть розташовуватись вздовж вектора напрямку з центру екрану до позиції джерела світла у двовимірній системі координат екрану:

XMVECTOR ScreenLightPosition = ConvertToScreenSpace(LightPosition);

XMVECTOR LightToCentreDirection = ScreenLightPosition - ScreenCentre;

Далі, маючи вектор направлення та позицію світла ми з визначеним шагом малюємо довільно текстури відблиску. Результуюче зображення додаємо до нашого:

```
const XMVECTOR StartPosition = ScreenLightPosition;
const XMFLOAT StepSize = 0.3;
for (int i = 0; i < NumberOfLestFlareTextures; ++i)
{
XMVECTOR DrawPosition = StartPosition + Normalize(LightToCentreDirection) * StepSize;
DrawTextureInPosition(DrawPosition, LensFlareTexture[i]);
}
```

В шейдері ми також можемо задати їм константне значення довільного кольору, та значення напівпрозорості, яке буде залежати від магнітуди нашого вектора направлення від джерела світла до центру *LightToCentreDirection.* Чим вона менша, тим більш чітко буде видно артефакти.

На рисунку 3.19 можна побачити приклад текстур, які можуть бути використані. Треба зауважити, що розмір намальованих артефактів слід підбирати не використовуючи розміри текстур, бо вони можуть бути замалими, або, навпаки, завеликими для роздільної здатності, яку ми обрали для нашого вікна.

На рисунку 3.20 результат впливу пост-процесу Lens Flare.

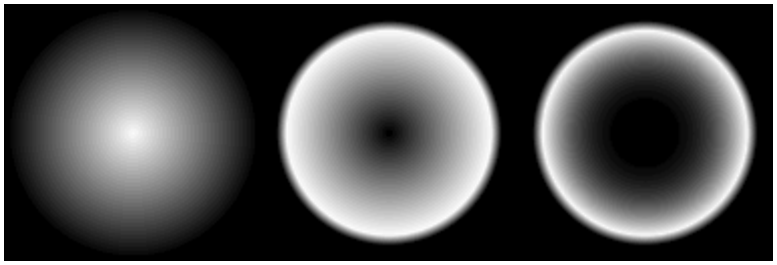


Рисунок 3.19 Текстури артефактів відблиску лінз

Рисунок 3.20 Вплив Lens Flare на зображення

У фінальну чергу імплементуємо SSR — дзеркальне відбиття від поверхонь.

Для цього, ми розраховуємо наступні текстури для використання:

- Карта нормалі пікселів у системі координат камери
- Карта позицій пікселів у системі координат камери
- Карта глибини сцени

Нашим завданням буде розрахувати вектор відбиття направлення від точки зору до пікселю(використовуючи задані вище текстури), та ітеративно робити кроки у напрямку цього вектора, кожен раз повертаючись у координати екрану:

float3 CameraToPixel = PixelsViewPos - ViewPos;

float3 ReflRay = normalize(reflect(CameraToPixel , PixelNormal));

*float3 TestPixel = PixelsViewPos + ReflRay * StepSize;*

Повертаючись до системи координат екрану ми перевіряємо, чи є в тій самій позиції в карті глибини значення, яке перекриває наш піксель, якщо є, то ми, потенційно, знайшли результуючий колір, який має бути відзеркалений.

Результат можна побачити на рисунку 3.21.

Рисунок 3.21 Вплив дзеркального відбиття на сцену

Закінчивши реалізацію всіх пост-процесів ми можемо побачити як саме перетворилося наше зображення після його обробки , використовуючи тільки інформацію, що ми отримали під час рендерингу 3D сцени (рисунок 3.22)

Як вже можна було побачити під час імплементації SSR та bloom, ми додали до сцени додатково дві моделі:

- Куб, що схематично вказує положення нашого джерела світла.
- Прямокутний паралелепіпед для наочного прикладу дзеркального відбиття SRR.

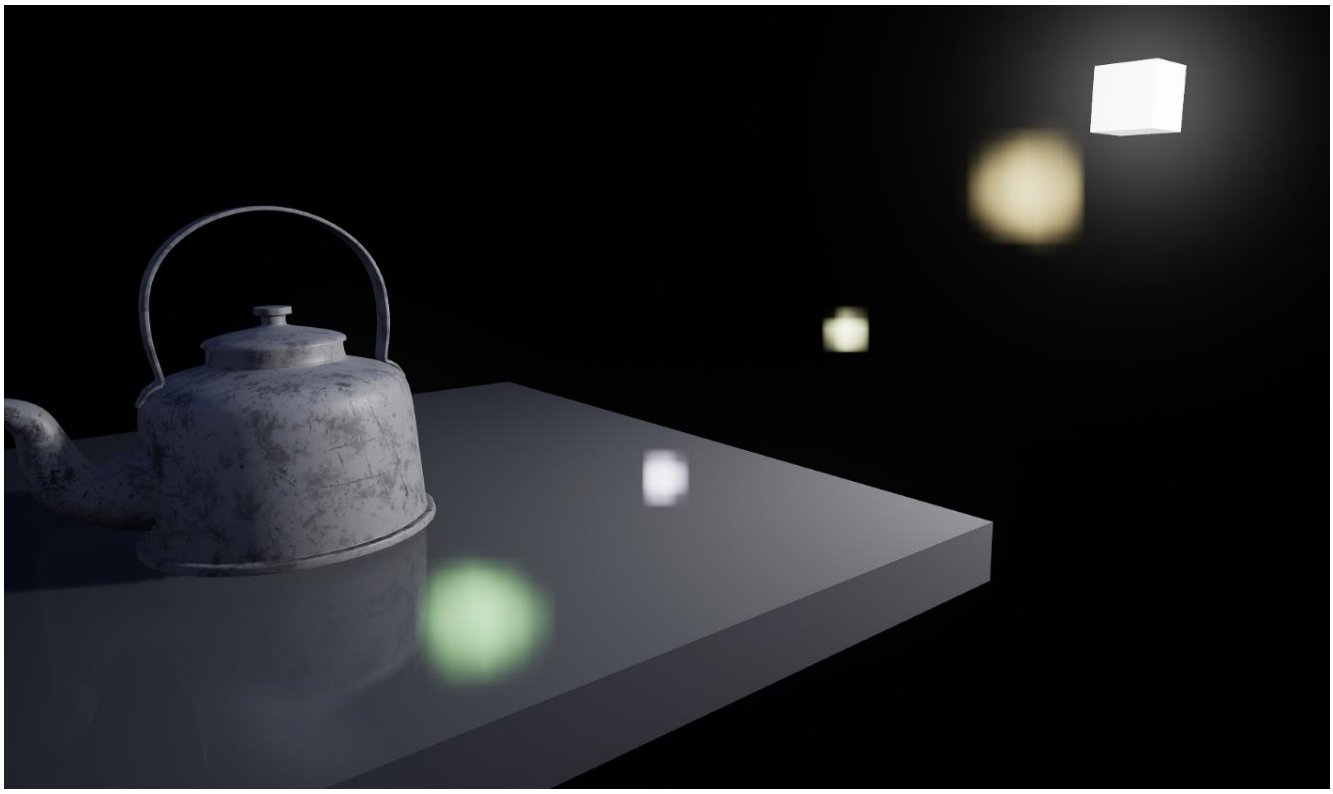


Рисунок 3.22 Результат впливу всіх пост-процесів

ВИСНОВКИ

Під час виконання дипломної роботи було проаналізовано, чого саме досягла комп’ютерна графіка як розділ комп’ютерних наук. Які техніки, методи та алгоритми використовуються для того, щоб апроксимувати фізичні закони нашого світу. Фізичні закони, що мають вплив на властивість навколишньої середи передавати інформацію до лінз нашого ока.

Для цього ми спочатку створили архітектуру, а потім імплементували з нуля програму яка візуалізує 3Д сцену з моделлю освітлення Блінн-Фонг. Програма імпортує файли моделі, текстури, налаштовує стан рендер API, виконує математичну логіку обчислення трансформацій вершин у різні системи координат, робить виклики команд з ЦПУ на ГПУ, обробляє введення даних користувачем з мишки та клавіатури, має систему для вільного польоту камери, систему дебагу, виведення логів та ще багато роботи, потрібної для візуалізатора та його розробки.

Реалізувавши різні пост-процеси, ми змогли обробити малюнок достатньо для того, щоб вважати його, в деякій мірі, схожим на реальність. Це вийшло зробити за допомогою: ефекту ореолу світіння bloom, артефактів відблиску від лінз lens flare та ефекту дзеркальної поверхні SSR(screen space reflection).

Всю рендер частину програми ми написали за допомогою DirectX 11 API: компіляція та виконання шейдерів HLSL, відправлення ресурсів з ЦПУ на відеокарту, ініціалізація стану відеокарти та багато іншого.

Використовуючи високорівневу мову C++ та сторонні бібліотеки, що направлені на прискорення роботи кожного з етапів рендеру, ми досягли високої швидкості програми та малого розміру файлу виконання(менш ніж 1.5 мегабайт). Це дозволило нам підтримувати частоту оновлення кадрів у розмірі 60 кадрів за секунду.

Як результат(рисунок 3.22), ми можемо сказати, що за допомогою пост-процесів можна досягти вагомого збільшення якості зображення. При тому, що пост обробка використовує лише ресурси, що ми отримали самі під час виконання програми: наше вихідне зображення, та ще деяку кількість текстур, які ми

рендеримо окремо. Використовуючи ці ресурси, ми обробили їх за допомогою HLSL шейдерів та отримали досить схожий на реальність результат.

ПЕРЕЛІК ПОСИЛАНЬ

1. Christopher Tremblay. Mathematics for Game Developers, 2004; pages 125-151
2. Ashoka Vanjare. GPU Gems 3: Programming Techniques, Tips and Tricks for Real-Time Graphics, 2019; pages 643-646
3. Joey de Vries. Learn OpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion, 2020; pages 271-275
4. Tomas Akenine-Moller, Eric Haines, Naty Hoffman. Real-Time Rendering, Fourth Edition , 2018; pages 524-527

ДОДАТОК А. Код програми

Увесь код програми налічує більш ніж декілька тисяч строк, він може бути знайдений у репозиторії за посиланням:

https://github.com/ArtemHuhlia/Renderer

ДОДАТОК Б. Демонстраційні матеріали

Презентація дипломної роботи на тему:

«Візуалізація 3D сцени на базі пост-процесів за допомогою Direct3D»

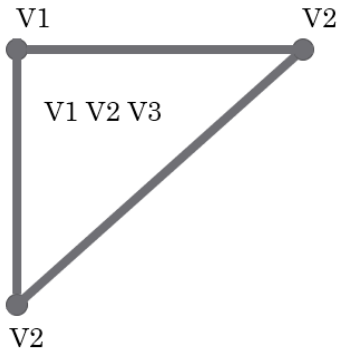
Виконав студент ППЗ – 51
Гугля А.О.
Керівник
Марченко В.В.

Київ 2021

3D Модель

Вершина (англ. Vertex) - це структура даних, яка описує певні атрибути, наприклад положення точки в 2D або 3D просторі.

Полігональна сітка (англ. Polygon mesh) — це набір вершин, ребер, та граней, що описують форму багатогранного об'єкта в тривимірній графіці та твердотілому моделюванні.

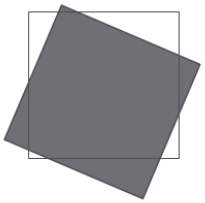


V1 = 0.5, 0.5, 1.0,-1.0, 0.0, 0.0, 0.5, 0.5

Position Normal UV

Матриця трансформації

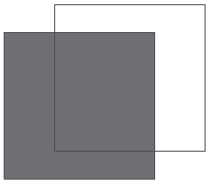
$$R = \begin{bmatrix} \cos\theta + R_x^2(1-\cos\theta) & R_xR_y(1-\cos\theta) - R_z\sin\theta & R_xR_z(1-\cos\theta) + R_y\sin\theta & 0 \\ R_yR_x(1-\cos\theta) + R_z\sin\theta & \cos\theta + R_y^2(1-\cos\theta) & R_yR_z(1-\cos\theta) - R_x\sin\theta & 0 \\ R_zR_x(1-\cos\theta) - R_y\sin\theta & R_zR_y(1-\cos\theta) + R_x\sin\theta & \cos\theta + R_z^2(1-\cos\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



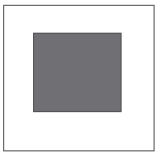
Transformation = T * S * R

1. S
2. R
3. T

$$T = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

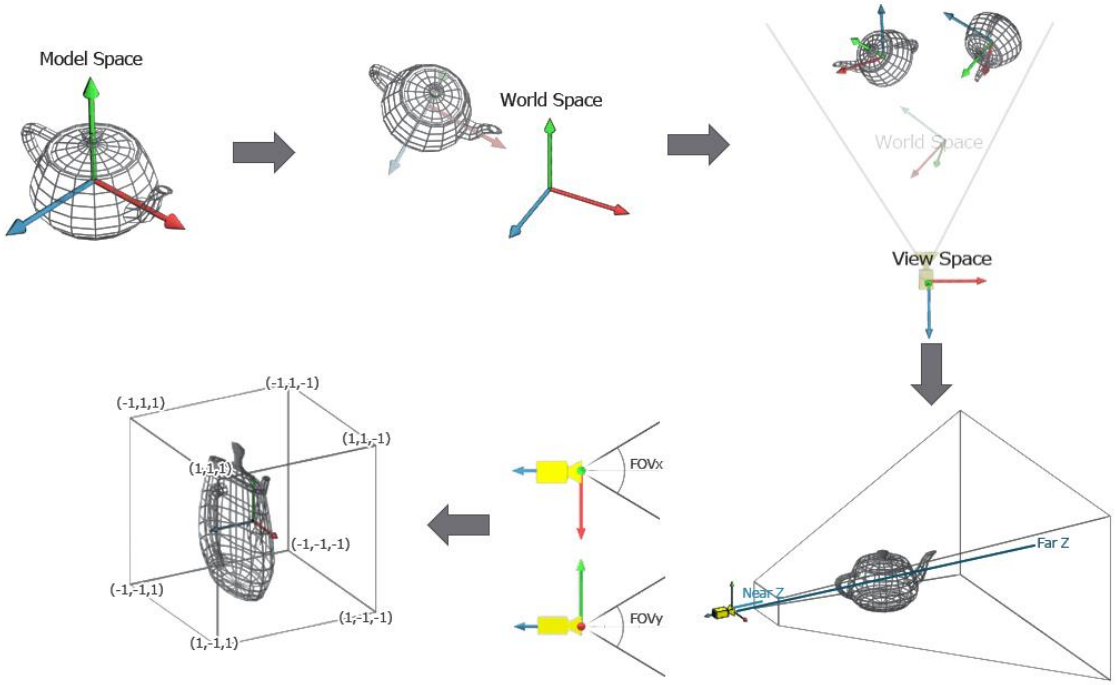


$$S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_Z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



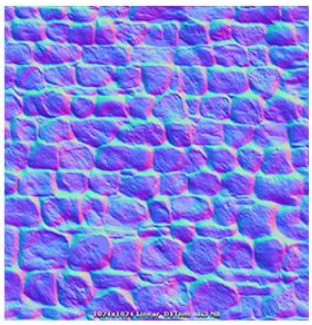
Зверніть увагу! Порядок множення матриць інверсійний. Ми використовуємо в поясненні так звану систему матриць на основі рядків, тому вершини, на які ми будемо множити матрицю трансформації треба розглядати як матрицю 4x1

Трансформації у комп'ютерній графіці



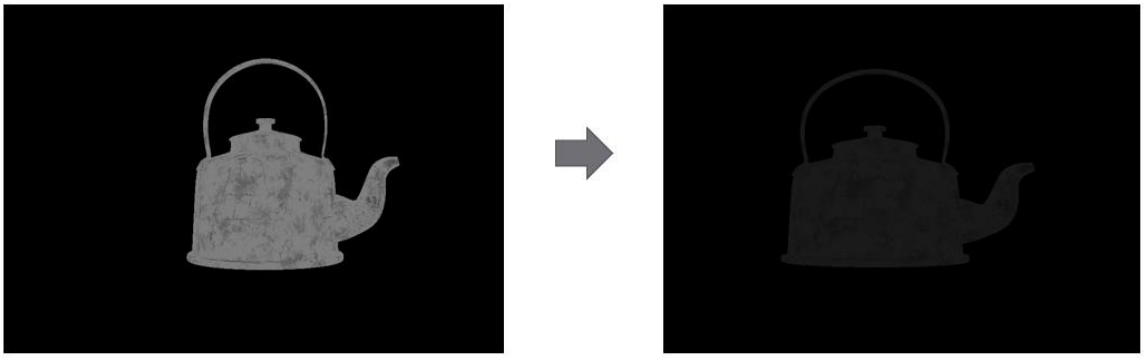
Текстури

Текстури – це двовимірні масиви які використовуються задля надання поверхні властивостей з покращеною деталізацією



Освітленя. Ambient

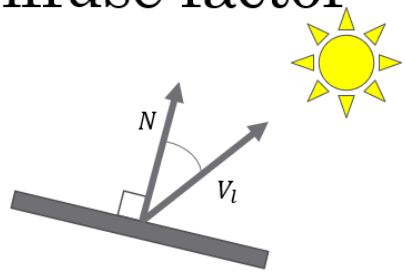
AmbientColor = *AmbientFactor* * *PixelColor*

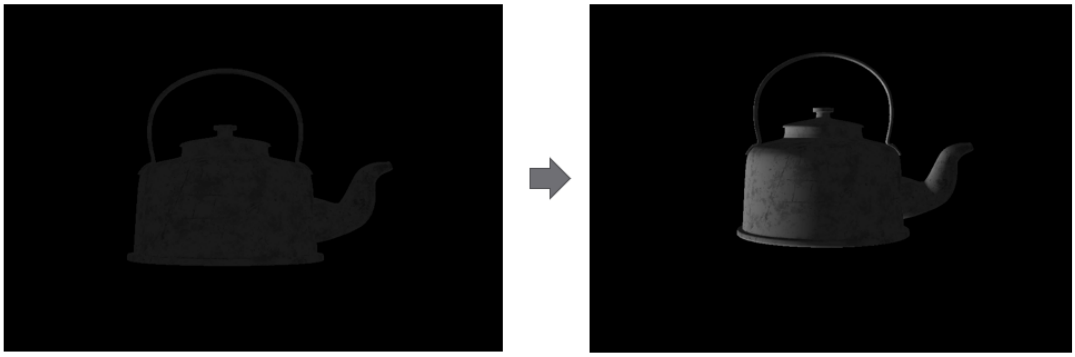


Освітлення. Diffuse factor

$$F_d = N \cdot V_l$$

F_d – дифузний фактор
V_l – нормалізований вектор направлення від координат пікселя до джерела світла
N – нормаль пікселя

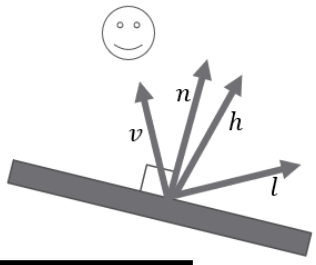




Освітлення. Specular

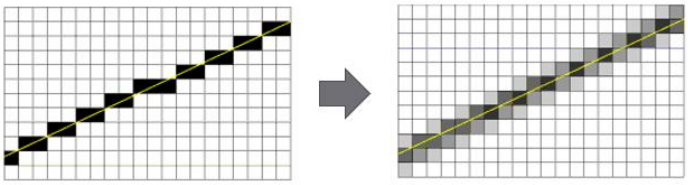
$$F_s = v \cdot h$$

v – вектор направлення від пікселя до лінзи
n – вектор нормалі пікселю
l – вектор направлення від пікселя до джерела світла
h – вектор направлення посеред v та l

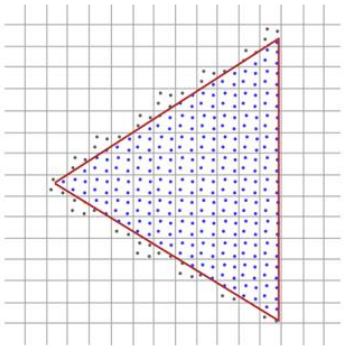


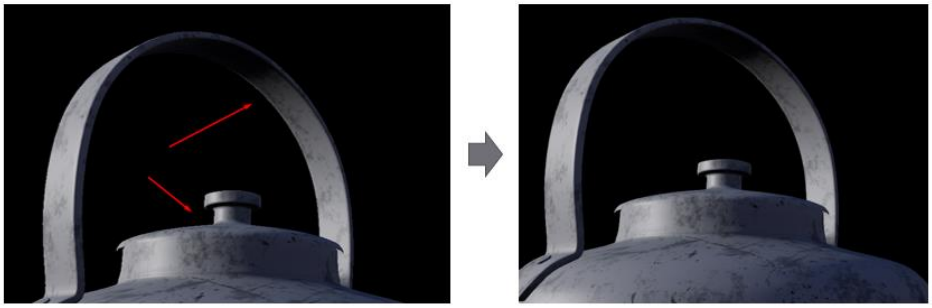
Пост-процеси.MSAA

Примітив, який ми замальовуємо — прямокутний піксель. Тому під час малювання протяжної кривої лінії ми будемо бачити сходинки з прямокутників.



Multisample anti-aliasing





Пост-процеси.Bloom

Спочатку ми повинні створити текстуру, в яку запишемо тільки ті пікселі нашого малюнку, які перевищують задану константу яскравості *Threshold*. Для цього ми повинні створити окремий піксель шейдер, який має виконувати наступну перевірку:

const float IntensityTest = (float)(length(PixelColor.rgb) > Threshold);

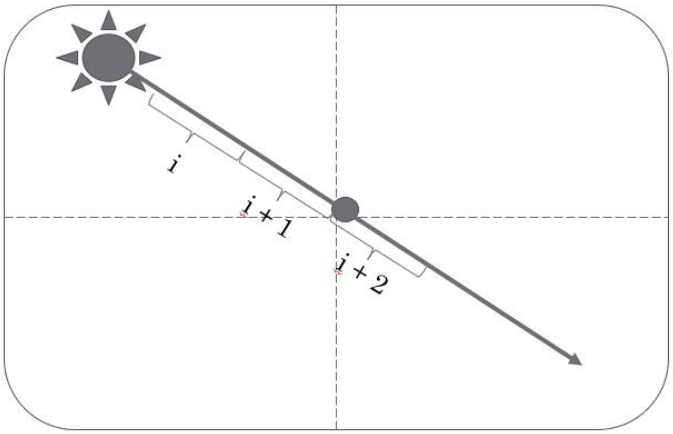
*FinalColor = float4(IsntensityTest * PixelColor.rgb, 1.0);*

Таким чином значення *IntensityTest* буде мати стан 1 або 0, в залежності від чого піксель буде мати оригінальний колір, або значення (0, 0, 0, 1).

Далі, використовуючи алгоритм Гаусового розмиття(дивитися в теоретичних відомостях), ми отримуємо розмите зображення найяскравіших пікселів. Об’єднуємо його с оригінальним зображенням та отримуємо результат:

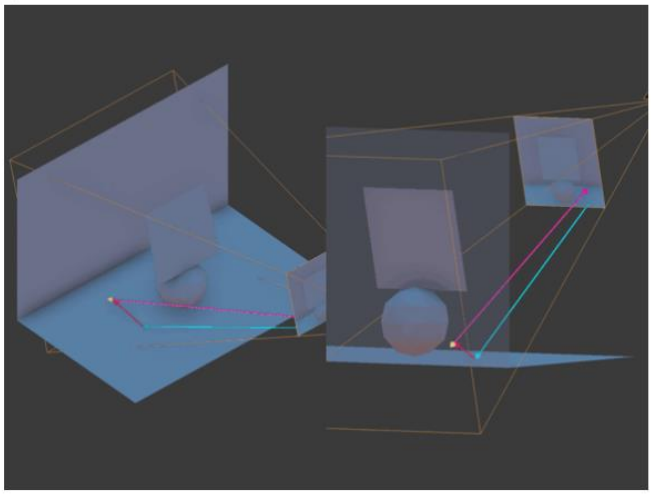
Пост-процеси. Lens Flare

Маючи вектор направлення та позицію світла ми з визначеним шагом малюємо довільно текстури відблиску. Результуюче зображення додаємо до нашого.



Пост-процеси.SSR

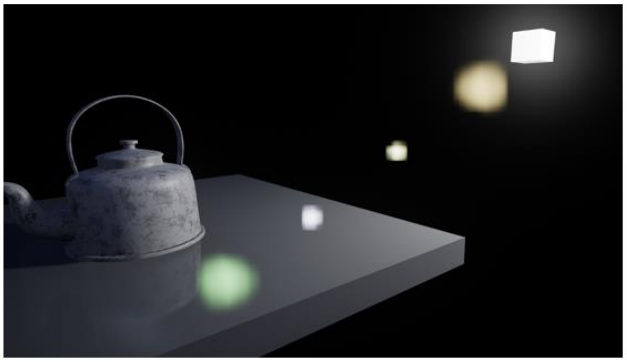
Нашим завданням буде розрахувати вектор відбиття направлення від точки зору до пікселю(використовуючи задані вище текстури), та ітеративно робити кроки у напрямку цього вектора, кожен раз повертаючись у координати екрану:



Повертаючисьдо системи координат екрану ми перевіряємо, чи є в тій самій позиції в карті глибини значення, яке перекриває наш піксель

Результат візуалізатора

До



Після

Висновки

- Під час виконання дипломної роботи було проаналізовано, чого саме досягла комп'ютерна графіка як розділ комп'ютерних наук. Які техніки, методи та алгоритми використовуються для того, щоб апроксимувати фізичні закони нашого світу. Фізичні закони, що мають вплив на властивість навколишньої середи передавати інформацію до лінз нашого ока.
- Реалізувавши різні пост-процеси, ми змогли обробити малюнок достатньо для того, щоб вважати його, в деякій мірі, схожим на реальність. Це вийшло зробити за допомогою: ефекту ореолу світіння bloom, артефактів відблиску від лінз lens flare та ефекту дзеркальної поверхні SSR(screen space reflection).