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

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка застосунку для рендерінгу та редагування 3D сцени мовою С++»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

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

________________ Богдан ЧЕРКАС
(підпис)

Виконав: здобувач вищої освіти групи ПД-41

Богдан ЧЕРКАС

Керівник: Олег ІЛЬЇН
д.т.н., професор

Рецензент: ________________

Київ 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_______________ Ірина ЗАМРІЙ
« ____ » ___________ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Черкасу Богдану Васильовичу

1. Тема кваліфікаційної роботи: «Розробка застосунку для рендерінгу та редагування 3D сцени мовою С++»
керівник кваліфікаційної роботи д.т.н., професор Олег ІЛЬЇН,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «27» лютого 2024 р. № 36.

2. Строк подання кваліфікаційної роботи «28» травня 2024 р.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про методи рендерінгу, технічна документація з описом бібліотек для обчислень на відеокарті.

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

   1. Огляд та аналіз існуючих застосунків для рендерінгу та редагування тривимірної сцени.
   2. Проектування застосунку для рендерінгу та редагування тривимірної сцени
   3. Розробка застосунку.

4. Тестування застосунку.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Діаграма варіантів використання.
5. Діаграма послідовності.
6. Діаграма класів
7. Алгоритм рендерінгу.
8. Екранні форми
9. Демонстрація роботи програми
10. Апробація результатів дослідження

6. Дата видачі завдання «28» лютого 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	28.02.-06.03.2024	
2	Аналіз та дослідження існуючих аналогів	07.03.-13.03.2024	
3	Огляд існуючих тривимірних графічних редакторів та аналіз проекту	14.03.-20.03.2024	
4	Проектування застосунку для рендерінгу та редагування тривимірної сцени	21.03.-29.03.2024	
5	Розробка застосунку	30.03.-24.04.2024	
6	Тестування застосунку	25.04.-28.04.2025	
7	Оформлення роботи: вступ, висновки, реферат	29.04.-05.05.2024	
8	Розробка демонстраційних матеріалів	06.05.-12.05.2024	
9	Попередній захист роботи	13.05.-31.05.2024	

Здобувач вищої освіти ________________ Богдан ЧЕРКАС
(підпис)

Керівник
кваліфікаційної роботи ________________ Олег ІЛЬЇН
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 43 стор., 1 табл., 16 рис., 10 джерел.

Мета роботи – спрощення рендерінгу та редагування 3D сцен за рахунок використання рейтрейсінгу.

Об'єкт дослідження – процес рендерінгу та редагування 3D сцен.

Предмет дослідження – програмні засоби рендерінгу та редагування 3D сцен.

Короткий зміст роботи: В роботі проаналізовано застосунки для рендерінгу та редагування тривимірної сцени, підкреслено їх сильні та слабкі сторони. Проаналізовано технології для проведення операцій рендерінгу на відеокарті. Розроблено алгоритм роботи застосунку для рендерінгу та редагування тривимірних сцен. Проведено мануальне тестування застосунку. Для розробки було використано мову програмування C++, технологію OpenCL та графічну бібліотеку SDL.

Сфера використання застосунку - Наукові дослідження і освіта

КЛЮЧОВІ СЛОВА: РЕНДЕРІНГ, РЕЙТРЕЙСІНГ, 3D-ОБ'ЄКТ, СЦЕНА, ЗАСТОСУНОК.

ЗМІСТ

ВСТУП.. 9
1 ОГЛЯД ТА АНАЛІЗ ІСНУЮЧИХ ЗАСТОСУНКІВ ДЛЯ РЕНДЕРІНГУ ТА РЕДАГУВАННЯ ТРИВИМІРНОЇ СЦЕНИ 11
1.1 Загальна характеристика тривимірних графічних редакторів........................... 11
1.2 Аналіз функціональних можливостей тривимірних графічних редакторів з точки зору задачі рендерінгу.. 13
1.3 Аналіз та обґрунтування вибору засобів розробки проекту 17
Висновки до першого розділу 19
2 ПРОЕКТУВАННЯ ЗАСТОСУНКУ ДЛЯ РЕНДЕРІНГУ ТА РЕДАГУВАННЯ ТРИВИМІРНОЇ СЦЕНИ.. 21
2.1 Моделювання вимог до програмного забезпечення.. 21
2.2 Проектування архітектури 22
2.3 Проектування інтерфейсу користувача 25
2.3.1 Вимоги до інтерфейсу користувача 25
2.3.2 Компоненти інтерфейсу... 25
2.3.3 Макет інтерфейсу 26
Висновки до другого розділу 28
3. РОЗРОБКА ЗАСТОСУНКУ 29
3.1 Створення вікна застосунку 29
3.2 Завантаження 3D-моделей та підготовка даних... 30
3.3 Передача даних в пам’ять відеокарти 31
3.4 Алгоритм рендерінгу 32
3.4.1 Створення напрямлених променів 33

3.4.2 Перевірка променю на перетин з об'єктами 33

3.4.3 Обчислення кольору пікселя 33

3.4.4 Обрахування кольору об’єктів, що мають дзеркальність 34

3.4.5 Генерація фінального зображення 34

3.5 Розробка візуального інтерфейсу 35

3.6 Відображення фінального результату 37

3.7 Діаграма класів 37

Висновки до третього розділу 44

4 ТЕСТУВАННЯ ЗАСТОСУНКУ 45

Висновки до четвертого розділу 50

ВИСНОВКИ 51

ПЕРЕЛІК ПОСИЛАНЬ 53

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ 55

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ 64

ВСТУП

Актуальність розробки застосунку для рендерінгу та редагування тривимірних сцен мовою C++ зумовлена зростаючими потребами в ефективних та високоякісних інструментах для роботи з 3D-графікою. Сучасні галузі, такі як архітектурна візуалізація, розробка відеоігор, анімація та віртуальна реальність, вимагають потужних рішень для створення реалістичних зображень. Використання рейтрейсінгу дозволяє значно підвищити якість візуалізації за рахунок точного моделювання взаємодії світла з об'єктами сцени.

Розробка такого застосунку є також важливою з точки зору навчання та досліджень, оскільки надає студентам і дослідникам можливість поглибленого вивчення алгоритмів рендерінгу, оптимізації використання ресурсів GPU та створення інтерактивних інтерфейсів. Це сприяє розвитку навичок програмування, розуміння архітектури програмних систем та застосування сучасних технологій у практичних проектах.

Об’єктом дослідження є процес рендерінгу та редагування 3D сцен.

Предметом дослідження є програмні засоби рендерінгу та редагування 3D сцен.

Мета роботи – спрощення рендерінгу та редагування 3D сцен за рахунок використання рейтрейсінгу.

Для досягнення поставленої мети в рамках дипломної роботи було поставлено та виконано наступні завдання:

1. Провести аналіз методів застосунків для рендерінгу та редагування 3D сцен, в тому числі, особливості використання рейтрейсінгу.
2. Провести аналіз та обґрунтування вибору засобів розробки застосунку для рендерінгу та редагування 3D сцени.
3. Виконати моделювання вимог до застосунку для рендерінгу та редагування 3D сцени.
4. Провести проектування архітектури та інтерфейсу користувача.

5. Розробити застосунок для рендерінгу та редагування 3D-сцени.
6. Провести тестування застосунку.

В рамках дипломного проекту проаналізовано застосунки для рендерінгу та редагування тривимірної сцени, підкреслено їх сильні та слабкі сторони. Проаналізовано технології для проведення операцій рендерінгу на відеокарті. Розроблено алгоритм роботи застосунку для рендерінгу та редагування тривимірних сцен. Проведено мануальне тестування застосунку.

Для розробки застосунку було використано мову програмування C++, бібліотеку SDL та технологію OpenCL.

Робота пройшла апробацію на Всеукраїнській науково-технічній конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях” (24 квітня 2024р., м. Київ, Державний університет інформаційно-телекомунікаційних технологій). За результатами апробації опубліковано тезу доповіді [1].

1 ОГЛЯД ТА АНАЛІЗ ІСНУЮЧИХ ЗАСТОСУНКІВ ДЛЯ РЕНДЕРІНГУ ТА РЕДАГУВАННЯ ТРИВИМІРНОЇ СЦЕНИ

1.1 Загальна характеристика тривимірних графічних редакторів

Тривимірні графічні редактори — це програми, що дозволяють користувачам створювати, модифікувати та зберігати 3D-графіку на комп'ютері для подальшого використання.

Процес роботи в таких редакторах зазвичай відбувається у три основні етапи. Найбільш часомісткий і складний етап – це моделювання, де з базових 3D об'єктів складається повноцінна сцена. Ці елементи можна модифікувати за допомогою різноманітних інструментів та комбінувати в більш складні структури. Після моделювання сцена може бути переглянута з різних кутів за допомогою обертання та переміщення камери. Також встановлюються джерела освітлення і камери, для збереження цікавих точок для огляду сцени. Для кращого орієнтування в просторі екран може бути поділений на декілька вікон: вид згори, збоку, спереду і з будь-якої іншої точки. Коли сцена готова, настає час її "оживлення" за допомогою накладання текстур.

Під час фінального етапу, рендерінгу, відбувається створення зображення, з урахуванням всіх текстур, світлових відблисків, тіней та інших візуальних ефектів. Використання 3D-редакторів не обмежується створенням статичних зображень; можливе також анімування об'єктів сцени для створення анімаційних фільмів. Окрім того, 3D-редактори використовуються у створенні персонажів для відеоігор.

Рендерінг - це процес перетворення 3D-моделі, сцени або об'єкта у двовимірне зображення на екрані, з урахуванням освітлення, текстур, тіней та інших візуальних ефектів, що забезпечує реалістичне або художнє відтворення сцени на двовимірному зображенні

Етап рендерінгу у тривимірній графіці є ключовим кроком у процесі візуалізації 3D-моделей і сцен. Цей процес включає перетворення тривимірних

моделей у двовимірне зображення, яке можна переглядати на екрані. Існує два основні види рендерінгу: рейтрейсінг (Ray Tracing) та растерізація (Rasterization).

Рейтрейсінг (Ray Tracing): Цей метод забезпечує високу якість зображень за рахунок точного моделювання взаємодії світла і матерії. Рейтрейсінг обчислює шляхи світлових променів, які відбиваються від об'єктів або проходять через них, що дозволяє точно відтворювати тіні, відблиски та рефракції.

Растеризація (Rasterization): Перетворює 3D моделі у пікселі на екрані, застосовуючи текстури та інші візуальні ефекти. Хоча растеризація менш ресурсомістка ніж рейтрейсінг, вона може бути менш точною у відтворенні взаємодій світла.

Дослідження популярних і успішних тривимірних графічних редакторів є важливим етапом для розуміння ключових функцій, механік та інструментів, які визначають цю категорію програмного забезпечення. Розглянемо декілька основних аспектів, які часто зустрічаються в таких редакторах:

Моделювання: Базовий компонент тривимірних редакторів, який дозволяє користувачам створювати складні 3D об'єкти з примітивів. Це може включати маніпуляції з вершинами, ребрами та полігонами для формування детальних моделей.

Текстурування: Процес накладення текстур на 3D моделі для додання реалістичності та деталізації поверхонь. Текстури можуть включати кольори, візерунки та картки нормалей, які імітують складність реальних матеріалів.

Освітлення: Важливий аспект, що впливає на зовнішній вигляд сцени у тривимірних редакторах. Правильне освітлення може значно підсилити реалізм 3D сцени, відтворюючи різні атмосферні ефекти, тіні та відблиски.

Анімація: Багато тривимірних графічних редакторів мають інструменти для анімації об'єктів та персонажів, дозволяючи їм рухатися та взаємодіяти відповідно до заданих траєкторій руху або скелетних анімацій.

Рендерінг: Процес перетворення тривимірної сцени у фінальне двовимірне зображення. Редактори пропонують різні методи рендерінгу, включаючи рейтрейсінг та растеризацію, для досягнення бажаного візуального ефекту.

Продовжуючи аналіз стандартних інструментів та можливостей, важливо зазначити, що редактор здатний адаптуватися до потреб користувача та пропонує гнучкість у персоналізації робочого процесу. Основні характеристики, такі як інтерфейс користувача, підтримка плагінів та інтеграція з іншими інструментами, повинні бути оптимізовані для ефективної роботи. Аналіз популярних редакторів допоможе визначити, які функції найбільше цінуються професіоналами галузі та як вони можуть бути вдосконалені або розширені в майбутньому.

1.2 Аналіз функціональних можливостей тривимірних графічних редакторів з точки зору задачі рендерінгу

Autodesk 3ds Max (рисунок 1.1) є одним з відомих продуктів у сфері тривимірного моделювання, анімації та рендерінгу, який вперше з'явився у 1996 році. Його первісні версії надали архітекторам, розробникам ігор та візуалізаторам потужний набір інструментів для створення детальних 3D моделей і складних анімацій. Ранній 3ds Max був особливо відомий своїм інтуїтивним інтерфейсом та широкими можливостями кастомізації, що забезпечувало користувачам гнучкість у реалізації творчих проектів. З розвитком технологій і потреб ринку, 3ds Max значно розширив свої можливості. Сучасні версії програми включають розширені функції для симуляції фізики, такі як динаміка рідин і тканин, покращені інструменти для рендерінгу, такі як Arnold, який забезпечує фотореалістичну якість зображень. 3ds Max також інтегрується з іншими продуктами Autodesk, що робить його незамінним інструментом у багатьох областях, від анімації та відеоігор до архітектурної візуалізації.

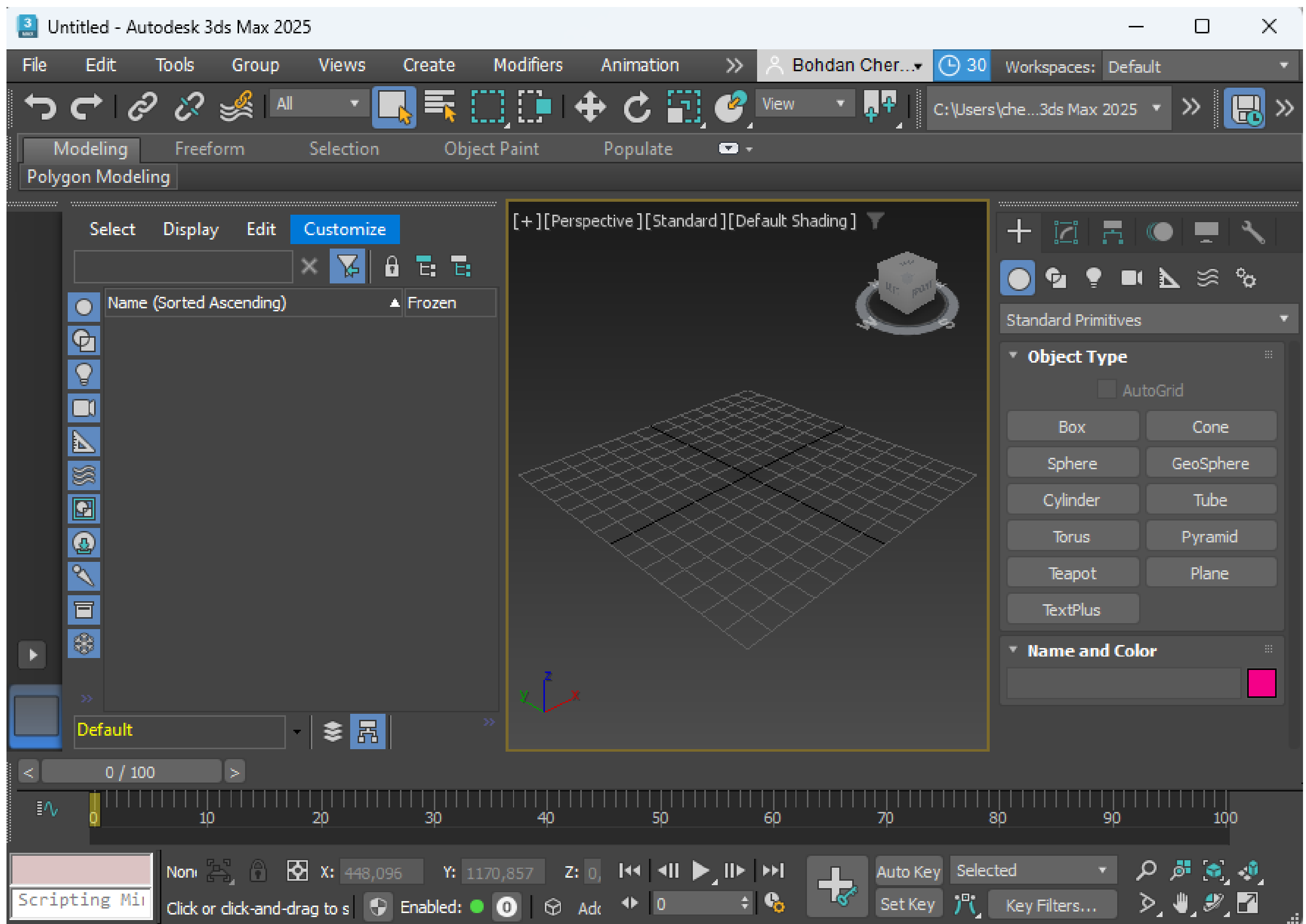


Рис.1.1 Знімок екрану застосунку «3ds Max»

Blender (рисунок 1.2) є відкритим програмним забезпеченням для 3D моделювання, анімації, рендерінгу, пост-продакшну та створення інтерактивних 3D застосунків, яке вперше було випущено в 1998 році. Від свого заснування Blender став особливо популярним у спільноті незалежних розробників та художників завдяки своїй відкритості, гнучкості та безкоштовності. Завдяки активному розвитку спільнотою, Blender постійно оновлюється та вдосконалюється, включаючи інновації, які роблять його конкурентоспроможним порівняно з комерційними аналогами. Особливістю Blender є його здатність об'єднувати різноманітні функції 3D графіки в одному пакеті, що охоплює всі аспекти 3D пайплайну — від моделювання, скульптурінгу та ретопології до складної анімації та симуляцій.

З появою версії Blender 2.8 та її значних оновлень у інтерфейсі користувача, введенні реальної підтримки PBR (Physically Based Rendering) та покращеному воркфлоу, Blender став ще більш доступним і зручним для

користувачів усіх рівнів. Він забезпечує підтримку різних рендерів, включаючи власний рендерер Cycles, який дозволяє створювати фотореалістичні зображення. Зростання популярності Blender у професійному сегменті підкріплюється його використанням у великих кіно- та анімаційних проектах, що демонструє його спроможність виконувати складні завдання на рівні з індустріальними стандартами. Його постійне вдосконалення та адаптація до сучасних технологій роблять Blender одним із ключових інструментів у світі цифрового дизайну та анімації.

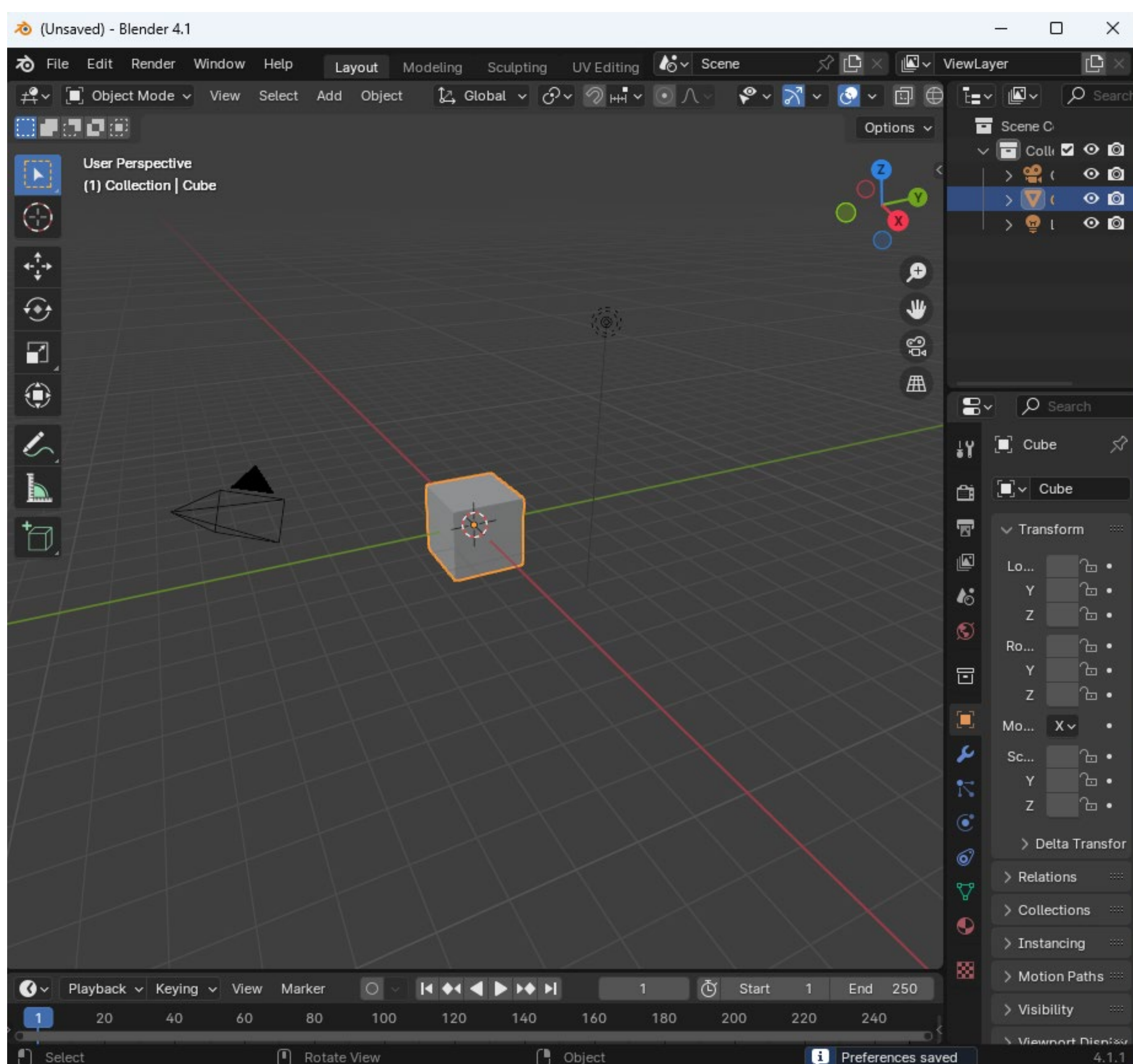


Рис. 1.2 Знімок екрану застосунку «Blender

LightWave 3D, розроблений компанією NewTek, є одним з класичних інструментів у сфері 3D моделювання, анімації та рендерінгу, який вперше з'явився на ринку у 1990 році. Спочатку LightWave 3D використовувався переважно для телевізійної продукції, але з часом знайшов застосування у кіноіндустрії, рекламі, архітектурі, та відеоіграх, завдяки своїм багатим можливостям та гнучкості.

З початку свого існування, LightWave здобув популярність завдяки інноваційному підходу до обробки відео та створення спецефектів. Його надзвичайна потужність в області анімації і рендерінгу дозволила створювати вражаючі візуальні ефекти, що часто виходили за рамки можливостей конкурентних програм у ті часи.

З розвитком програми, LightWave 3D ввів значні оновлення, такі як покращений інтерфейс користувача, що став більш інтуїтивним та доступним для новачків, а також інтеграцію новітніх технологій рендерінгу, що забезпечили ще більш реалістичні зображення. Програма також розширила свої можливості у сфері динамічних симуляцій, забезпечуючи користувачам інструменти для створення складних анімаційних сцен.

LightWave 3D продовжує бути важливим інструментом для професіоналів у сфері візуальних ефектів, особливо завдяки своїй здатності об'єднувати потужні інструменти моделювання та анімації з ефективними рішеннями для рендерінгу. Його використання у великомасштабних проектах та незмінна популярність у спільноті художників свідчить про стабільне становище LightWave як про одного з лідерів у сфері 3D графіки.

Проаналізувавши дані застосунки можна дійти висновку що сучасні тривимірні редактори багатофункціональні: з їх допомогою можна не лише редагувати сцени, а і створювати моделі, анімації, іноді навіть ефекти.

Проте така підтримка багатьох функцій використовує потужність платформи на якій запущено застосунок, що не дозволяє виділити абсолютно всі ресурси для самого рендерінгу та зайве ускладнює його алгоритм для

підтримки інших функцій. В таблиці 1.1 наведено зведені результати аналізу розглянутих тривимірних редакторів.

Таблиця 1.1

Порівняння існуючих тривимірних редакторів

Властивості	3Ds Max	Blender	LightWave 3D	RTracer
Редагування сцени	+	+	+	+
Підтримка різних видів освітлення	+	-	+	+
Проста у використанні	-	-	+	+
Рендерінг з реалістичним освітленням за допомогою рейтрейсінгу	+	-	+	+
Висока швидкість рендерінгу в реальному часі при обмежених ресурсах	-	-	-	+
Платформа	Windows	Windows OS X Linux	Windows OS X	Windows Linux

1.3 Аналіз та обґрунтування вибору засобів розробки проекту

Для розробки проекту, що використовує ресурси GPU для обчислення графіки, необхідно вибрати технологію, яка дозволить виконувати програмний

код на відеокарті та передавати туди дані. Зазвичай для растрової графіки використовують API OpenGL, DirectX або Vulkan. Однак у випадку рейтрейсінгу необхідні інші засоби, які дозволяють виконувати більш різноманітний код на відеокарті.

CUDA (Compute Unified Device Architecture) — паралельна обчислювальна платформа та програмна модель від NVIDIA, що використовує потужність графічних процесорів (GPU). Вона підтримує мови C++, Python, Java та інші, забезпечуючи масивно-паралельні обчислення.

Переваги:

- Висока продуктивність і ефективність.
- Оптимізація під архітектуру GPU NVIDIA.
- Широкий набір інструментів для розробки.
- Підтримка машинного навчання та штучного інтелекту.

Недоліки:

- Обмежена сумісність лише з GPU від NVIDIA.
- Залежність від одного вендора.
- Висока складність розробки.

OpenCL (Open Computing Language) — відкритий стандарт для програмування на гетерогенних платформах, включаючи CPU, GPU, DSP та FPGA. Розроблений під егідою Khronos Group, він забезпечує паралельне виконання задач.

Переваги:

Крос-платформенність.

Гнучкість і відкритість.

Підтримка широкого спектру обчислювальних задач.

Недоліки:

Непостійна продуктивність між платформами.

Складність розробки і оптимізації.

Відсутність стандартизації підтримки обладнання.

DirectX Compute Shader — частина DirectX API від Microsoft для програмування загальнопризначених обчислень на GPU в рамках DirectX 11 і новіших версій.

Переваги:

Інтеграція з екосистемою DirectX.

Підтримка з боку Microsoft.

Оптимізація під апаратну архітектуру Windows.

Недоліки:

Платформна залежність.

Обмежена гнучкість.

Залежність від версій DirectX.

Vulkan Compute Shader — сучасний графічний та обчислювальний API від Khronos Group, що надає високий рівень контролю над ресурсами.

Переваги:

Крос-платформенність.

Низькорівневий контроль та висока продуктивність.

Модернізація та підтримка з боку спільноти.

Недоліки:

Висока складність розробки.

Більші вимоги до ресурсів для навчання.

Залежність від якості реалізації драйверів.

Отже, для написання коду на GPU було вибрано технологію OpenCL, оскільки вона працює на багатьох платформах та з різними виробниками відеокарт (Nvidia та AMD), а також є відносно простою для опанування.

Висновки до першого розділу

Перший розділ дослідження підкреслив фундаментальні аспекти та етапи роботи з тривимірними графічними редакторами, що включають моделювання, текстурування, освітлення, анімацію та рендерінг. Аналіз сучасних

тривимірних редакторів, таких як 3ds Max, Blender та LightWave 3D, показав їхню спроможність адаптуватися до різноманітних вимог користувачів, надаючи потужні інструменти для реалізації складних візуальних проектів, що при цьому і створює недолік: недостатньо ефективну роботу рендерінгу сцени через необхідність підтримки інших можливостей редактору.

З'ясувалося, що необхідність у високоякісному рендерінгу, який здатен ефективно використовувати ресурси сучасної обчислювальної техніки, є критичною. Це відкриває потенціал для розробки нових засобів у сфері рендерінгу, які могли б покращити швидкість та якість візуалізації 3D сцен. Особливе значення має розвиток та оптимізація рейтрейсінгу, який є предметом даної роботи.

Розглянувши існуючі рішення, можна стверджувати, що індустрія 3D графіки знаходиться в стані постійного розвитку, і є значний простір для інновацій. Такі інновації можуть включати розробку спеціалізованого програмного забезпечення, зорієнтованого на оптимізацію рендерінгу, що стане предметом подальшого дослідження у наступних розділах цієї роботи.

Також для виконання коду на GPU було вибрано технологію OpenCL у зв'язку з багатоплатформенністю та відносною простотою роботи.

2 ПРОЕКТУВАННЯ ЗАСТОСУНКУ ДЛЯ РЕНДЕРІНГУ ТА РЕДАГУВАННЯ ТРИВИМІРНОЇ СЦЕНИ

2.1 Моделювання вимог до програмного забезпечення

Редактор тривимірних сцен має бути досить швидким та давати можливість користувачу взаємодіяти зі сценою. Зважаючи на це було розроблено вимоги до програмного забезпечення:

Функціональні вимоги описують конкретні функції, які система повинна виконувати:

1. Можливість рендерінгу 3D сцен за допомогою рейтрейсінгу.
2. Додавання та видалення об’єктів.
3. Управління камерою.
4. Підтримка імпорту 3D-моделей та експорту створених сцен.
5. Налаштування освітлення.

Нефункціональні вимоги визначають атрибути якості, які система повинна мати, щоб бути ефективною, надійною та зручною у використанні:

1. Висока швидкість рендерінгу в реальному часі - не менше ніж 30 FPS.
2. Можливість роботи під управлінням операційних систем Windows та Linux.
3. Сумісність з відеокартами від різних виробників та підтримку різних API, зокрема Nvidia та AMD.
4. Оптимізована робота з пам’яттю, зокрема пам’яттю відеокарти.

Діаграма варіантів використання ілюструє основні взаємодії між користувачами та системою, підкреслюючи функціональні вимоги. Також вона використовується як довідник для зрозуміння основного потоку операцій у системі. Загальний вигляд діаграми варіантів представлено на рисунку 2.1.

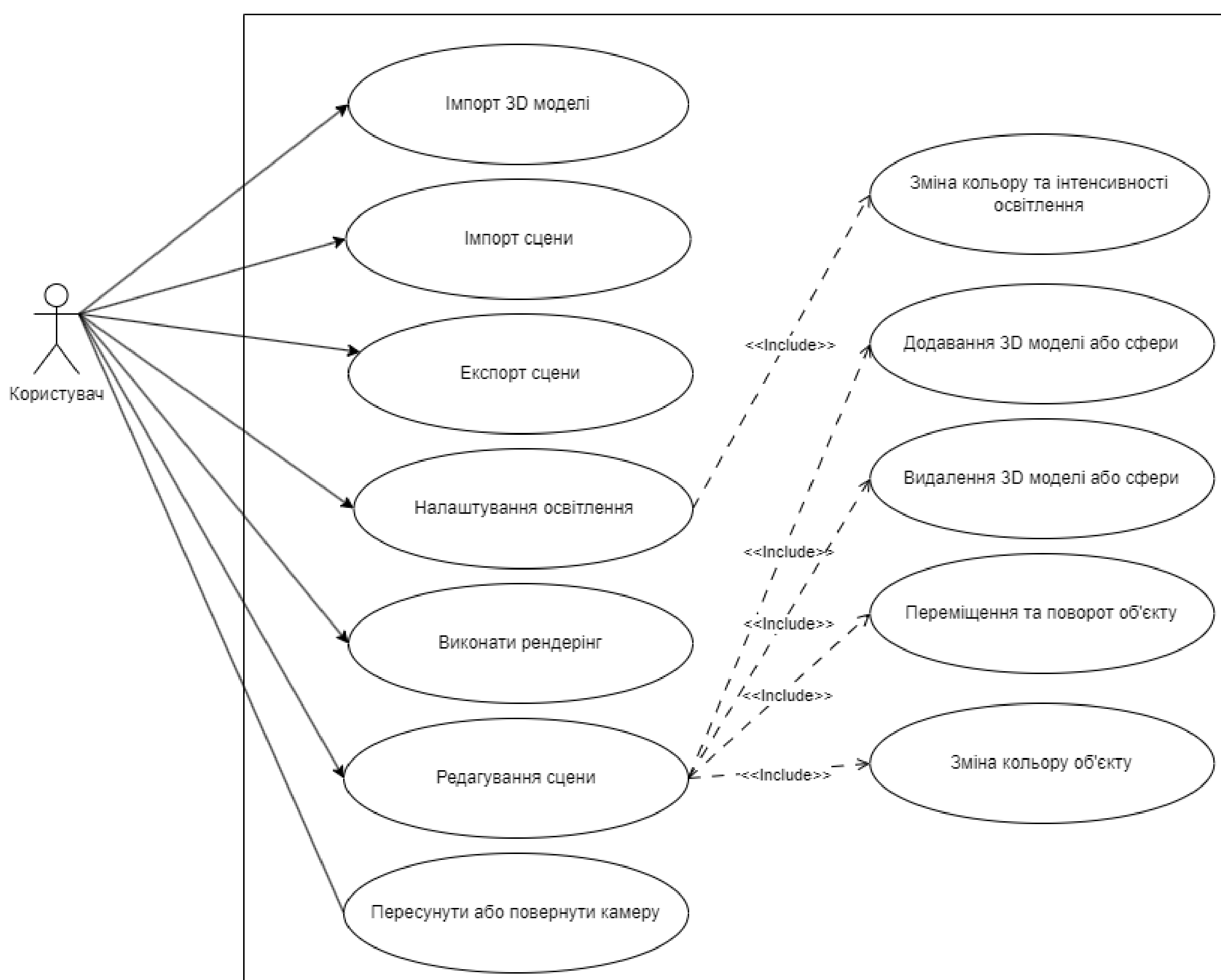


Рис. 2.1 Діаграма варіантів використання

2.2 Проектування архітектури

Архітектура рендеринг-пайплайну (Rendering Pipeline) є структурованою послідовністю етапів, через які проходять дані для створення зображення з тривимірної сцени. Вона є основою більшості графічних движків і використовується для перетворення 3D-моделей у 2D-зображення, які ми бачимо на екрані. Вона забезпечує послідовне або паралельне виконання ряду етапів (стадій), де кожен етап виконує свою специфічну задачу. Цей підхід дозволяє підвищити продуктивність та ефективність обробки, розподіляючи навантаження між різними компонентами системи.

Архітектура Pipeline складається з кількох ключових етапів, кожен з яких відповідає за обробку різних аспектів застосунку. В контексті рендерінгу за рахунок рейтрейсінгу можна описати такі етапи архітектури Pipeline:

1. Завантаження даних.
2. Обробка сцени, оновлення місцезнаходження та повороту об’єктів.
3. Передача даних сцени у пам’ять відеокарти.
4. Трасування променів.
5. Шейдінг.
6. Виведення на екран.

Переваги архітектури Pipeline:

- Кожен етап Pipeline можна розробляти та оптимізувати як окремий модуль, що полегшує обслуговування та розширення системи.
- Деякі етапи можуть виконуватися паралельно, що може підвищити продуктивність.
- Архітектура Pipeline дозволяє легко додавати нові етапи або змінювати існуючі без значних змін у всій системі.
- Висока продуктивність: Завдяки розподілу обробки на кілька етапів, система може ефективно використовувати ресурси, такі як багатоядерні процесори та GPU.

Архітектура Pipeline є ефективним підходом до рендерінгу 3D-сцени, забезпечуючи високу продуктивність та гнучкість системи. Вона дозволяє розподіляти навантаження між різними компонентами та спрощує розробку та обслуговування системи. Діаграма послідовності, що відображає роботу системи у відповідності до архітектури Pipeline, наведена на рисунку 2.2

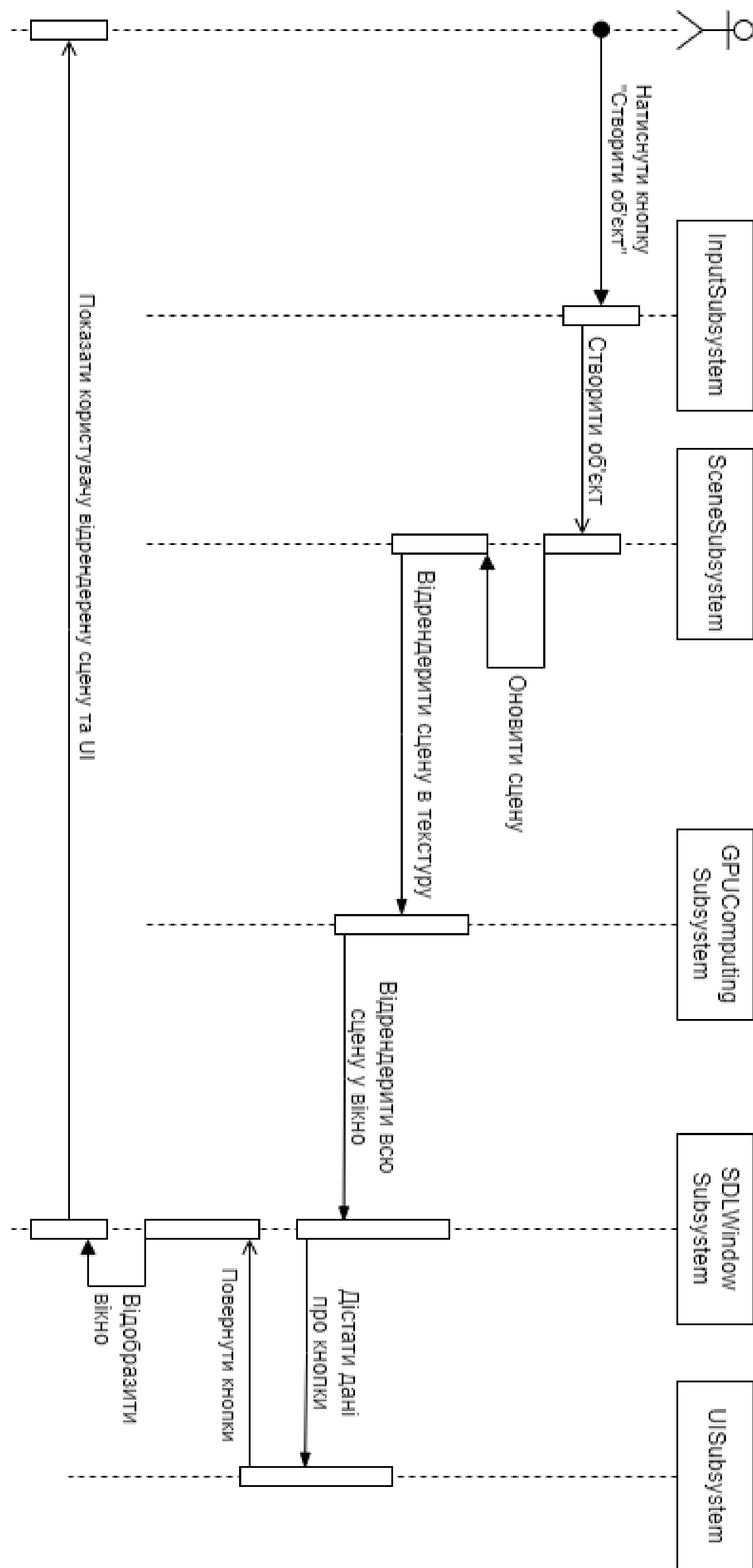


Рис. 2.2 Діаграма послідовності роботи застосунку при створенні об'єкту на сцені

2.3 Проектування інтерфейсу користувача

2.3.1 Вимоги до інтерфейсу користувача

Перед початком проектування необхідно визначити вимоги до інтерфейсу, які враховують потреби цільової аудиторії та специфіку проекту. Основні вимоги включають:

- Інтерфейс має бути зрозумілим для користувача з першого погляду, без потреби у тривалому навчанні аналогічно до інтерфейсів Blender і LightWave 3D, які відомі своєю зручністю для початківців.
- Усі елементи інтерфейсу повинні розташовуватися таким чином, щоб користувач легко міг знайти потрібні функції, як у 3ds Max, де інтерфейс дозволяє швидко орієнтуватися завдяки зрозумілому розташуванню інструментів.
- Основні функції та інструменти повинні бути легко доступними, щоб мінімізувати час, необхідний на їх пошук.
- Інтерфейс повинен мати інтерактивні елементи, текстові поля і кнопки, які дозволяють легко налаштовувати параметри, як це реалізовано в Blender.
- Інтерфейс повинен підтримувати використання гарячих клавіш для швидкого доступу до функцій, як це реалізовано в Blender, де гарячі клавіші значно прискорюють роботу.
- Інтерфейс повинен бути візуально привабливим і консистентним, використовуючи однакові стилі, кольори і шрифти для всіх елементів, як це реалізовано в LightWave 3D.

2.3.2 Компоненти інтерфейсу

Інтерфейс користувача повинен включати такі основні компоненти:

- Набір елементів інтерфейсу повинен містити інструменти для редагування та налаштування 3D сцен. Інструменти для переміщення об'єктів повинні бути розташовані в окремій панелі,

доступній користувачу, подібно до Blender, де ці інструменти знаходяться з лівого боку інтерфейсу. Інтерактивні кнопки для виконання операцій, таких як імпорт моделей, додавання/видалення об'єктів, повинні бути інтуїтивно зрозумілими та легко доступними.

- Центральна частина інтерфейсу призначена для відображення 3D сцени. Вона повинна забезпечувати можливість взаємодії з сценою (повертання, переміщення) за допомогою миші та клавіатури, аналогічно до LightWave 3D.
- Панель властивостей повинна відображати всі параметри вибраного об'єкта (позиція, матеріали) і дозволяти їх змінювати, як у Blender. Інтерактивні текстові поля забезпечують точне налаштування параметрів об'єктів, що є особливо зручним у 3ds Max.

2.3.3 Макет інтерфейсу

Макет інтерфейсу користувача повинен бути організований таким чином, щоб забезпечити логічний і зручний доступ до всіх необхідних функцій. Основні принципи проектування макету включають групування схожих елементів разом для зручності використання, відсутність зайвих елементів і складних конструкцій, щоб не перевантажувати інтерфейс, а також використання однакових стилів, кольорів і шрифтів для всіх елементів.

Інтерфейс розроблений з урахуванням потреб і досвіду користувачів, які працюють з 3D моделюванням і рендерингом. UX (user experience) користувача включає декілька ключових аспектів:

- Користувачі можуть легко орієнтуватися в інтерфейсі завдяки логічному розташуванню інструментів та функцій. Новачки можуть швидко освоїти базові операції без необхідності тривалого навчання, що робить інтерфейс доступним для широкої аудиторії.
- Основні інструменти для редагування та налаштування 3D сцени розташовані у видимій та доступній зоні інтерфейсу. Це дозволяє

професіоналам швидко виконувати свої завдання, мінімізуючи час на пошук необхідних функцій.

- Центральна частина інтерфейсу відведена під відображення та взаємодію з 3D сценою. Користувачі можуть легко маніпулювати об'єктами (обертати, масштабувати, переміщувати) за допомогою миші та клавіатури, що забезпечує високий рівень контролю та точності.
- Інтерактивні елементи, такі як слайдери, текстові поля та кнопки, дозволяють легко налаштовувати параметри. Використання гарячих клавіш для швидкого доступу до функцій значно прискорює робочий процес, що є особливо корисним для професіоналів.

На рисунку 2.3 можна побачити макет інтерфейсу застосунку з урахуванням вимог до інтерфейсу користувача.

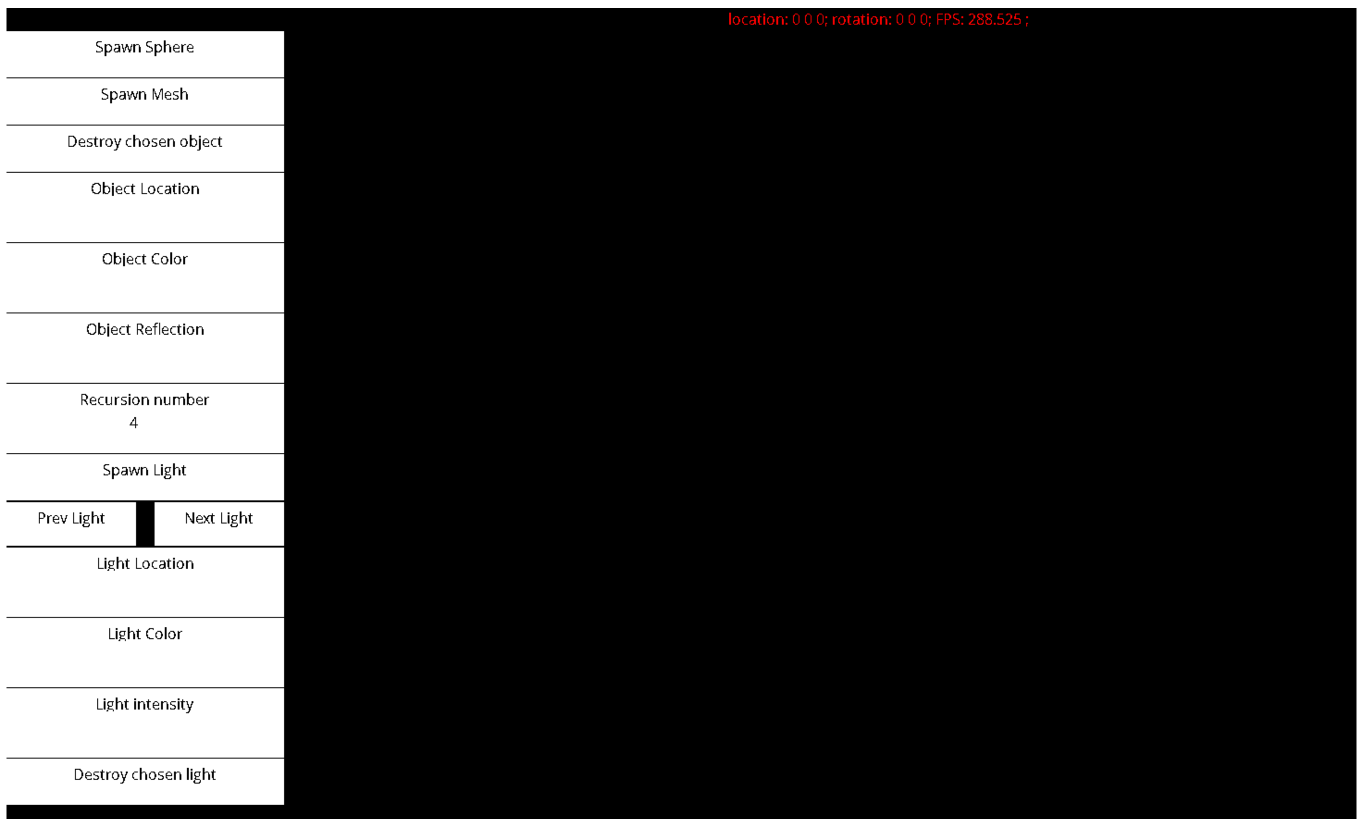


Рис. 2.3 Макет інтерфейсу застосунку

Висновки до другого розділу

У другому розділі роботи було розглянуто проектування застосунку для рендерінгу та редагування тривимірних сцен. В першу чергу, визначено вимоги до програмного забезпечення, зокрема функціональні та нефункціональні вимоги. Було встановлено, що для успішного виконання завдань застосунку необхідно забезпечити високу швидкість рендерінгу в реальному часі та сумісність з різними операційними системами та відеокартами.

Проектування архітектури застосунку базується на концепції рендерінг-пайплайну (Rendering Pipeline), що дозволяє ефективно розподіляти навантаження між різними етапами обробки даних. Цей підхід забезпечує високу продуктивність та гнучкість системи, дозволяючи легко додавати нові етапи або змінювати існуючі без значних змін у всій системі.

Особлива увага приділена проектуванню інтерфейсу користувача, де було враховано потреби цільової аудиторії та специфіку проекту. Інтерфейс користувача включає кілька основних компонентів: інструменти для редагування та налаштування 3D сцен, центральну частину для відображення 3D сцени, та панель властивостей для налаштування параметрів об'єктів. Вимоги до інтерфейсу включають інтуїтивну зрозумілість, логічну структуру розташування елементів, легкодоступність основних функцій, інтерактивність, підтримку гарячих клавіш та візуальну привабливість.

Завдяки детальному опрацюванню компонентів інтерфейсу вдалося створити макет, що забезпечує логічний і зручний доступ до всіх необхідних функцій. Основні принципи проектування макету включають групування схожих елементів, відсутність зайвих елементів та складних конструкцій, а також використання однакових стилів, кольорів і шрифтів.

3. РОЗРОБКА ЗАСТОСУНКУ

3.1 Створення вікна застосунку

Створення вікна застосунку є першим кроком у розробці графічного інтерфейсу. Для цього створено клас **USDLWindowSubsystem**.

Для створення вікна використано бібліотеку SDL (Simple DirectMedia Layer), котра підтримує багато платформ та дозволяє створювати мультиплатформенні застосунки. SDL надає прості та ефективні засоби для створення вікна та додавання в нього елементів.

Для створення вікна потрібно ініціалізувати SDL за допомогою функції **SDL_Init**. Це включає ініціалізацію основних підсистем SDL, таких як відео та події. У разі успішної ініціалізації потрібно створити вікно, задати його розмір, позицію та параметри, наприклад повноекранний режим, з чим допоможе функція **SDL_CreateWindow**

Для того щоб мати можливість відображати графічні елементи у вікні, необхідно створити контекст рендерінгу. Це здійснюється за допомогою функції **SDL_CreateRenderer**, яка створює об'єкт, що буде використовуватись для малювання елементів у вікні.

Після створення вікна та контексту рендерінгу важливо налаштувати обробку подій. Це включає обробку подій закриття вікна, натискання клавіш та інших подій, які забезпечують взаємодію користувача із застосунком. SDL надає механізм для обробки подій через чергу подій, що дозволяє легко керувати різними типами подій. Щоб дістати подію з черги використовується функція **SDL_PollEvent.**

Під час рендерінгу сцени застосунок буде заповнювати кожен піксель створеного вікна обрахованим кольором і таким чином створювати зображення.

3.2 Завантаження 3D-моделей та підготовка даних

Процес завантаження 3D-моделей є критично важливим етапом підготовки даних для рендерінгу. 3D-моделі містять геометрію об'єктів сцени, яку необхідно обробляти за допомогою рейтрейсінгу. В застосунку за зберігання 3D-моделі відповідає клас **FMeshObject**.

Процес імпорту моделей включає відкриття файлу, зчитування даних та створення внутрішніх структур для зберігання геометрії. Геометрія моделей зазвичай зберігається у вигляді вершин, нормалей, текстурних координат та індексів, які визначають полігони (в основному трикутники, але це можуть бути інші багатокутники). Після завантаження моделі дані можуть потребувати нормалізації для забезпечення коректного рендерінгу. Це включає центрування моделей відносно їх середини та масштабування до стандартного розміру.

Перевірка цілісності завантажених даних є важливим етапом, що включає перевірку на наявність всіх необхідних атрибутів, таких як вершини, нормалі та текстурні координати. Це допомагає уникнути помилок під час рендерінгу.

Перевірка цілісності дуже проста: якщо у моделі не вистачає вершин, нормалей, тощо, то модель не буде завантажено.

Оскільки у даному випадку не використовується анімація та матеріали, процес завантаження значно спрощується. Немає потреби обробляти ключові кадри для анімації або налаштовувати параметри матеріалів, такі як колір, прозорість та відбиття. Це дозволяє зосередитися виключно на геометрії моделей та їх оптимізації для рендерінгу.

За імпорт 3D-моделей, їх валідацію та збереження в об’єкти класу FMeshObject відповідає клас MeshObjectParser.

Після імпорту 3D-модель зберігається у пам’яті сцени, в об’єкті класу USceneSubsystem, після чого застосунок може передати дані в пам’ять GPU та відрендерити її.

3.3 Передача даних в пам’ять відеокарти

У збереженні 3D-моделі у об’єкті сцени є особливість: для ефективного використання пам'яті GPU всі полігони, вершини та нормалі 3D-моделі копіюються у масиви даних, які потім будуть передані у пам’ять GPU. Це робиться для того, щоб забезпечити швидкий доступ до даних під час рендерінгу та мінімізувати затримки, пов'язані з передачею даних між CPU та GPU. Кожна 3D-модель представлена у вигляді трьох основних масивів: масиву полігонів, масиву вершин та масиву нормалей. Ці масиви повинні бути організовані таким чином, щоб забезпечити ефективну адресацію та доступ до даних.

Для того щоб під час рендерінгу розуміти, які полігони та вершини до якого об’єкту належать, створюється об’єкт класу FGPUMeshObject. Цей об’єкт зберігає у собі інформацію про кількість полігонів, індекс, який вказує, де в масивах даних знаходяться полігони, вершини та нормалі даної 3D-моделі, а також інші метадані, необхідні для правильної обробки 3D-моделі на відеокарті. FGPUMeshObject також може містити інформацію про матеріали та текстури, які використовуються для рендерінгу конкретної моделі, що дозволяє забезпечити більш реалістичне відображення сцен.

Для обрахунку даних і рендерінгу на відеокарті використовується технологія OpenCL, яка надає інтерфейс для надсилання команд відеокарті та взаємодії з її пам’яттю. Щоб передати всю інформацію про сцену, а саме масиви об’єктів, всіх полігонів, вершин, нормалей, джерел світла та налаштування сцени, потрібно ініціалізувати OpenCL і використати об’єкт cl::Buffer.

Ініціалізація OpenCL в C++ починається з включення заголовочного файлу CL.hpp і створення контексту, який визначає середовище, в якому виконуються обчислення. Спочатку потрібно отримати доступні платформи OpenCL за допомогою функції clGetPlatformIDs, а потім вибрати одну з них. Далі, використовуючи платформу, отримують пристрої (відеокарти або

центральні процесори) за допомогою clGetDeviceIDs. Після цього створюють контекст за допомогою clCreateContext, передаючи вибрані платформи та пристрої. Завершальним кроком є створення черги команд для вибраного пристрою за допомогою clCreateCommandQueue, що дозволяє надсилати завдання для виконання. Після цього можна копіювати дані в cl:Buffer.

cl:Buffer копіює дані у спеціальний внутрішній масив у пам’яті відеокарти, що потім напряму використовується відеокартою. Він забезпечує ефективне управління пам’яттю та швидкий доступ до даних під час виконання обчислень на відеокарті.

Таким чином, перед початком рендерінгу, на пам’ять відеокарти копіюються масиви з інформацією про всі 3D-моделі на сцені.

3.4 Алгоритм рендерінгу

Алгоритм рендерінгу є ключовим компонентом у процесі створення зображень з 3D-сцен. Він визначає, як дані, що зберігаються в пам'яті GPU, обробляються для отримання кінцевого зображення.

Для побудування алгоритму рендерінгу було використано рейтрейсінг та модель відображення Фонга.

Модель відображення Фонга в комп'ютерній графіці створює реалістичне освітлення поверхонь, враховуючи три компоненти: фонове, дифузне та дзеркальне освітлення. Фонове освітлення симулює рівномірне розсіяне світло, яке є всюди в сцені незалежно від джерел світла чи глядача. Дифузне освітлення імітує матове відображення світла від поверхні, залежно від кута падіння світла. Дзеркальне освітлення відображає блискуче світло, враховуючи як кут падіння, так і положення глядача.

Алгоритм можна розбити на кілька частин:

3.4.1 Створення напрямлених променів

Алгоритм рендерінгу починається зі створення напрямлених променів для кожного пікселя екрану. Ці промені визначають напрямок, у якому буде "дивитися" кожен піксель для обрахування кольору.

Точкою початку променю являється позиція камери в просторі. Напрямок обраховується за рахунок перетворення двувимірних координат пікселів на екрані на трьовимірні координати простору на сцені.

3.4.2 Перевірка променю на перетин з об'єктами

Для кожного променю перевіряється перетин з об'єктами сцени. Це досягається за допомогою використання математичних функцій перетину променю з різними типами об’єктів, такими як сфери, площини або 3D-моделі. Промені послідовно перевіряються на перетин з кожним об'єктом у сцені. Для кожного об'єкта обчислюється точка перетину, якщо така існує. В результаті обирається найближчий об'єкт у напрямку променю.

3.4.3 Обчислення кольору пікселя

Після визначення точки перетину для кожного променю обчислюється колір відповідного пікселя. Це включає врахування фонового, дифузного та дзеркального освітлення, а також прозорості та відбиття.

Дифузне освітлення обчислюється на основі кута падіння світла на поверхню. Якщо світло падає прямо на поверхню, вона виглядає яскравішою, а якщо під гострим кутом – тьмянішою. Це імітує матове відображення світла, де світло розсіюється рівномірно у всіх напрямках від точки падіння.

Дзеркальне освітлення враховує кут відбиття світла від поверхні. Це освітлення створює блискучі відблиски на поверхнях, подібні до тих, які ми бачимо на полірованих або металевих поверхнях. Важливу роль тут грає позиція камери: відбитий промінь світла повинен потрапити в камеру, щоб створити ефект блиску.

Комбінуючи ці компоненти, алгоритм визначає остаточний колір кожного пікселя, що дозволяє отримати реалістичне зображення з врахуванням різних типів освітлення та взаємодії світла з поверхнями.

3.4.4 Обрахування кольору об’єктів, що мають дзеркальність

Алгоритм також враховує дзеркальність об'єктів, що дозволяє досягти більш реалістичних результатів. Не варто плутати дзеркальність та дзеркальний елемент освітлення: дзеркальне освітлення відповідає за відблиски на глянцевому об’єкті, а дзеркальність відповідає за відбивання світла як дзеркало.

Для обрахування дзеркальності алгоритм використовує додаткові промені що створюються враховуючи кут відбиття світла від поверхні. Додаткові промені шукають найближчий інший об’єкт, дістають його колір в точці перетину та накладають його колір на поверхню дзеркального об’єкту для симуляції дзеркала.

3.4.5 Генерація фінального зображення

Фінальне зображення генерується шляхом виконання алгоритму рендерінгу для кожного пікселя екрану. Результати зберігаються у вихідний масив пікселів, окремний cl::Buffer, з якого вихідні дані копіюються далі.

Таким чином, алгоритм рендерінгу забезпечує доволі реалістичне відображення 3D-сцени.

Діаграма активності, що відображає алгоритм рендерінгу одного пікселю, представлена на рисунку 3.1

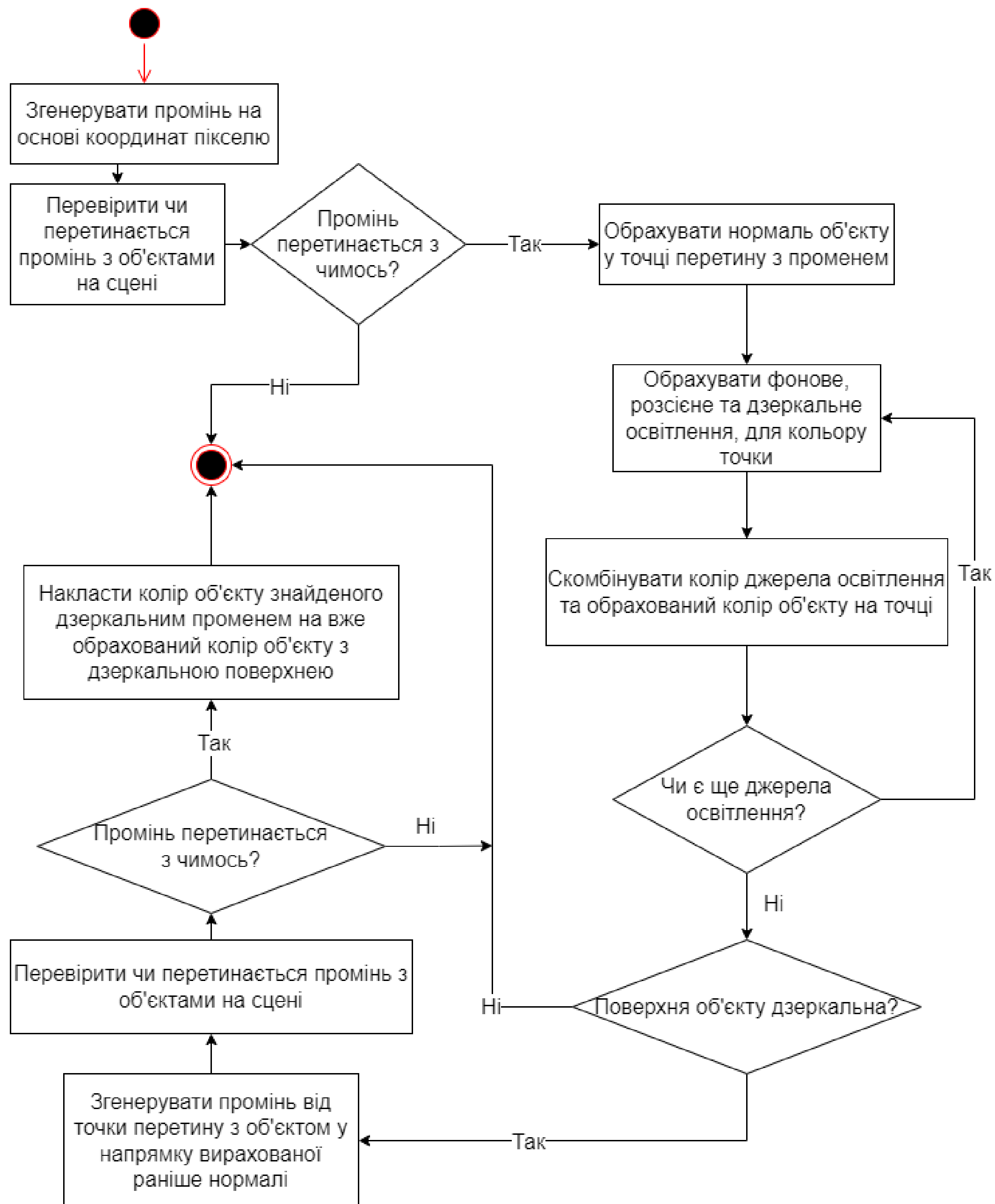


Рис. 3.1 Діаграма активності, що відображає алгоритм рендерінгу одного пікселю

3.5 Розробка візуального інтерфейсу

Оскільки вікно застосунку було створено з допомогою бібліотеки SDL, було вирішено розробити інтерфейс користувача з її ж допомогою. SDL графічна бібліотека, проте вона не надає вже готових рішень для створення таких UI елементів як кнопки, текстові поля, тощо. Натомість SDL дає можливість малювати у вікні прямокутники заповнені кольором функцією

SDL_FillRect, пусті прямокутники функцією DrawRect та рендерити текст з допомогою з допомогою функції SDL TTF_RenderText_Solid.

Використовуючи наявний функціонал було розроблено декілька основних елементів інтерфейсу.

Кнопки необхідні щоб виконувати прив'язаний до них функціонал, наприклад імпорт нової 3D-моделі.

Кнопки було створено використовуючи три елементи SDL: заповнений заданим кольором прямокутник, поверх якого накладається пустий прямокутник з обводкою та текст.

Для того щоб зафіксувати натискання на кнопку, при натисканні на ліву клавішу миші дістається її позиція та порівнюється з позицією та розміром всіх кнопок. Перша знайдена кнопка на яку наведена миша вважається натиснутою і викликається функціонал закріплений до цієї кнопки.

Приклад кнопок можна побачити на рисунку 3.2.

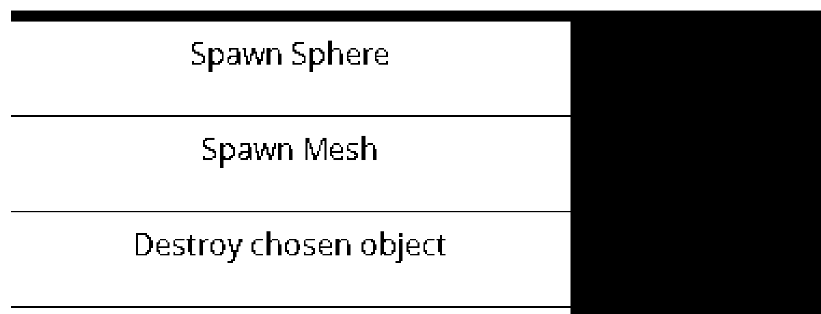


Рис. 3.2 Вигляд кнопок у застосунку

Інтерактивне текстове поле необхідне для відображення певних параметрів сцени або вибраного об'єкту на сцені, та зміни цих значень користувачем.

Інтерактивні текстові поля створено таким же чином як кнопки, проте при натисканні замість виклику закріпленої функції, дозволяється введення тексту та зміна на основі введеного значення прикріпленого до поля елементу. Наприклад можна прикріпити до поля позицію камери та змінювати її при оновленні текстового поля.

На рисунку 3.3 можна побачити приклад інтерактивних текстових полів.

Light Location
24/ 2.00 0.00

Light Color
255 255 255

Light intensity
1.00

Рис. 3.3 Інтерактивні текстові поля

3.6 Відображення фінального результату

Після обробки даних, передачі їх в пам’ять GPU та обрахунку кольорів пікселів необхідно це все вивести у вікні, щоб користувач бачив результат.

Спочатку вікно екрану повністю очищається та робиться чорним. Далі ліва частина екрану замальовується сірим кольором і в тій зоні відображаються всі елементи UI. Згодом з пам’яті відеокарти напряму в текстуру SDL копіюються результати обчислень і текстура копіюється у вікно. В кінці результат рендерінгу відображається на екрані користувача.

3.7 Діаграма класів

Під час розробки застосунку було використано архітектуру “Pipeline”, що розбиває роботу застосунку на послідовні розділені етапи.

Процес роботи застосунку було розділено на основні частини

- Читання введених користувачем даних
- Оновлення об'єктів на сцені (якщо щось додали або змінили)
- Передача даних в пам'ять відеокарти та обрахування рендерінгу
- Рендерінг UI

- Відображення у вікні

Опис класів застосунку:

FObject – відповідає за відображення всіх видів об’єктів для відеокарти. може представляти з себе як сферу, так і 3D-об'єкт. У випадку якщо це сфера – зберігає в собі радіус, якщо це 3D-об’єкт, то зберігає індекс об’єкту.

Атрибути:

- Position – позиція об’єкту
- Rotation – поворот
- Color – колір
- Reflection – дзеркальність, від 0 до 1
- Type – тип об’єкту
- MeshIdx – індекс 3D-моделі
- Radius – радіус сфери

FFace – представляє з себе полігон, трикутник. Зберігає в собі індекси вершин та нормалі.

Атрибути:

- Vertices – масив вершин, звичайних векторів

FMeshObject – представляє з себе імпортовану 3D-модель. Зберігається лише в основній пам'яті, не передається у пам'ять відеокарти.

Атрибути:

- Faces – масив полігонів FFace
- Vertices – масив вершин
- Normals – масив нормалей
- UniqueID – індекс 3D-моделі
- BoundingBox – представляє з себе 2 точки, які указують межі об’єкту

FGPUMeshObject – оптимізована для пам'яті відеокарти репрезентація 3D-моделі.

Атрибути:

- BoundingBox - представляє з себе 2 точки, які указують межі об'єкту
- StartingFaceIdx – індекс початку всіх полігонів цього об'єкту в масиві всіх полігонів
- StartingVertexIdx– індекс початку всіх вершин цього об'єкту в масиві всіх вершин
- StartingNormalIdx– індекс початку всіх нормалей цього об'єкту в масиві всіх нормалей
- NumFaces – кількість полігонів
- UniqueID - індекс 3D-моделі

UMeshObjectParserSubsystem – відповідає за імпорт 3D-моделей.

Методи:

- ParseMeshObjectFromFile - зчитує файл в переданому у функцію шляху, та повертає імпортовану 3D-модель як об'єкт класу FMeshObject
- ParseMeshObjectFromDialogWindow - відкриває вікно Windows, в якому вибирається файл для читання, повертає імпортовану 3D-модель як об'єкт класу FMeshObject

FCamera – віртуальна камера користувача, точка огляду сцени

Атрибути:

- Origin – позиція камери
- Rotation – поворот камери

USceneSubsystem – відповідає за збереження всіх даних сцени, додавання та видалення об'єктів.

Методи:

- SpawnSphere – створює сферу та додає її у список всіх об'єктів для рендерінгу
- SpawnMesh – за допомогою UMeshObjectParserSubsystem імпортує 3D-модель та додає її на сцену. Одразу створює її оптимізовану версію – FGPUMeshObject
- AddLightSource – додає джерело світла
- RemoveObject – прибирає обрану сферу або 3D-модель зі сцени
- RemoveLightSource - видаляє обране джерело світла
- Update – оновлює інформацію про кількість об'єктів, вершин, полігонів та їх позицію

Атрибути:

- Meshes – масив 3D-об'єктів, FMeshObject
- OptimizedMeshes - масив оптимізованих об'єктів, FGPUMeshObject
- Primitives – масив FObject
- Lights – масив джерел світла, FLightSource
- AllSceneFaces – масив полігонів всіх об'єктів, FFace
- AllSceneVertices – масив вершин всіх об'єктів
- AllSceneNormals – масив нормалей всіх об'єктів
- Camera – камера якою управляє користувач

USDLWindowSubsystem – відповідає за вікно і відображення на екрані.

Методи:

- GetScreenHeight – повертає висоту вікна в пікселях
- GetScreenWidth – повертає ширину вікна в пікселях
- Update – відображає на екрані відрендерену сцену

Атрибути:

- Window – вікно застосунку

- Texture – текстура куди записуються результати рендерінгу перед відображенням на екрані

UIButton – представляє з себе кнопку UI

Методи:

- OnButtonClicked – викликається при натисканні кнопки

Атрибути:

- Position – позиція кнопки
- Size – розміри кнопки
- Color – колір кнопки
- Text – текст кнопки

UIInputText – інтерактивне текстове поле

Атрибути:

- Position – позиція поля
- Size – розміри поля
- Color – колір поля
- Text – текст поля

UUISubsystem – зберігає кнопки та текстові поля, управляє ними

Методи:

- Update – надсилає вікну інформацію про наявні кнопки та інтерактивні текстові поля для їх рендерінгу.
- Initialize – створює всі кнопки та текстові поля

Атрибути:

- Buttons – масив кнопок
- InputTextFields – масив інтерактивних текстових полів

UInputSubsystem – відповідає за інпут користувача.

Методи:

- Update – відловлює інформацію про інпут користувача з бібліотеки SDL
- ProcessInputDown – виконує функціонал прив'язаний до натискання клавіші клавіатури.
- ProcessUIInput – виконує функціонал прив'язаний до натискання кнопки на екрані.

UGPUComputingSubsystem – передача даних і запуск обрахування на відеокарті.

Методи:

- Update – копіює дані зі сцени у пам’ять відеокарти та запускає програму на відеокарті.

Атрибути:

- ActiveDevice – посилання на відеокарту
- Context – являється мостом для відправки команд на відеокарту

На рисунку 3.4 представлено діаграму класів.

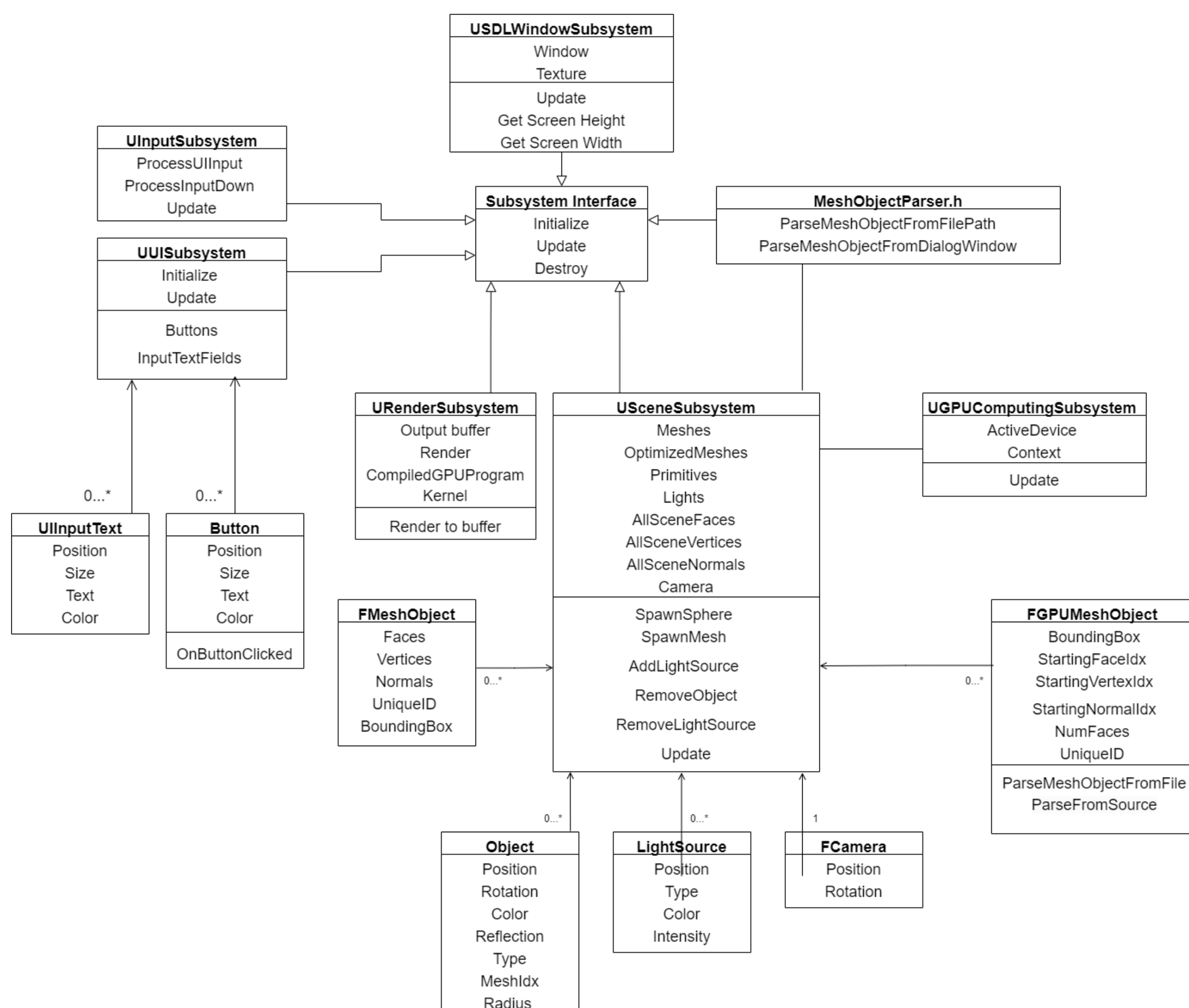


Рис. 3.4 Діаграма класів

Висновки до третього розділу

У третьому розділі було детально описано процес розробки застосунку для рендерінгу та редагування тривимірної сцени. Основні етапи розробки включали створення вікна застосунку, завантаження та підготовку даних, передачу даних в пам'ять відеокарти, реалізацію алгоритму рендерінгу, розробку візуального інтерфейсу та відображення фінального результату.

Використано бібліотеку SDL (Simple DirectMedia Layer), яка дозволяє створювати мультиплатформенні застосунки. Створено клас USDLWindowSubsystem для ініціалізації та налаштування вікна і обробки подій.

Процес завантаження 3D-моделей включав імпорт файлів, перевірку цілісності даних та нормалізацію моделей. Дані зберігалися у внутрішніх структурах для ефективної обробки під час рендерінгу.

Для обробки даних на GPU використовувалася технологія OpenCL. Дані передавалися у пам'ять відеокарти за допомогою об'єктів cl::Buffer, що забезпечувало швидкий доступ до даних під час рендерінгу.

Алгоритм рендерінгу був побудований на основі рейтрейсінгу та моделі відображення Фонга, що дозволяло створювати реалістичні зображення з урахуванням освітлення, тіней та відбиттів.

Інтерфейс користувача створений за допомогою SDL, включає кнопки та інтерактивні текстові поля для забезпечення взаємодії користувача із застосунком.

Результати рендерінгу відображалися у вікні застосунку, дозволяючи користувачеві бачити та взаємодіяти з 3D-сценою в реальному часі.

Розробка застосунку здійснювалася за архітектурним підходом "Pipeline", що дозволяло розділити процес на послідовні етапи, забезпечуючи високу продуктивність і гнучкість системи.

4 ТЕСТУВАННЯ ЗАСТОСУНКУ

Мануальне тестування є важливим етапом розробки програмного забезпечення, оскільки дозволяє перевірити функціональність застосунку у реальних умовах. В рамках мануального тестування було проведено перевірку всіх сценаріїв використання застосунку, які були заявлені при визначені функціональних вимог.

Перший прецедент для тестування – чи працює рендерінг та чи виводить результати на екран. Для цього тесту створено просту сцену щоб провести її огляд. Першу тестову сцену зображено на рисунку 4.1.

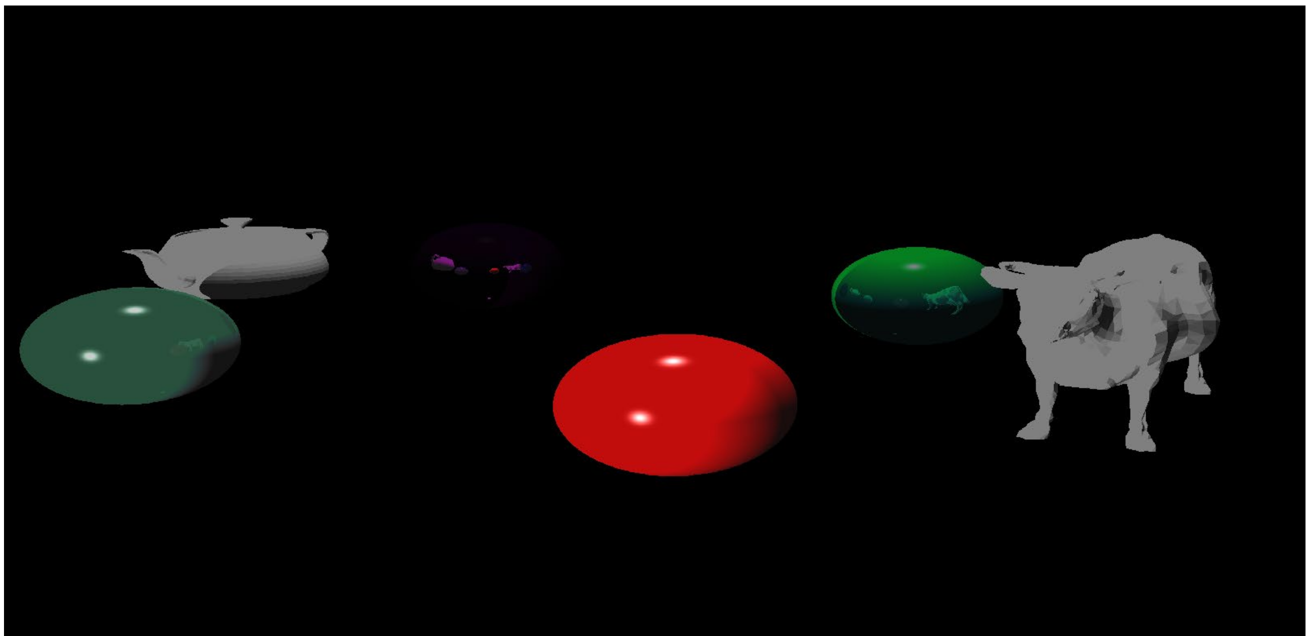


Рис. 4.1 Тестова сцена з двома 3D-моделями та кількома сферами

Другий прецедент – це можливість додавати об'єкти під час роботи застосунку иа змінювати їх позицію на сцені. Було створено ще одну сферу та переміщено її поміж інших об'єктів. Першу тестову сцену з новим об'єктом зображено на рисунку 4.2.

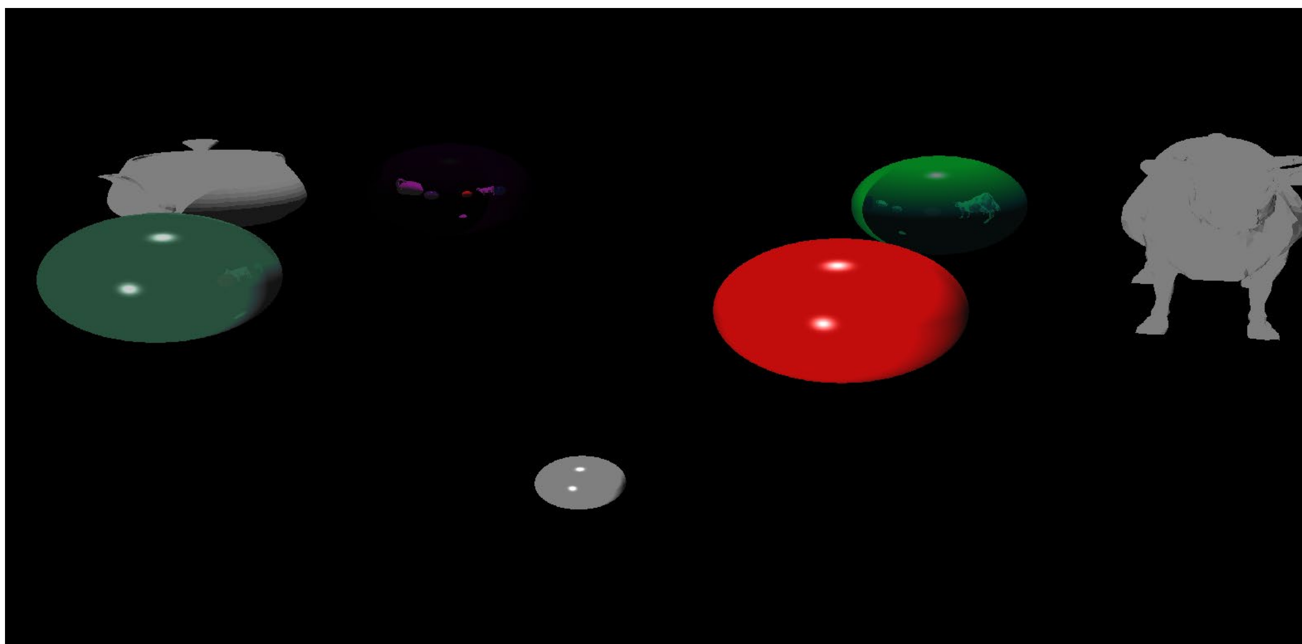


Рис. 4.2 Перша тестова сцена з новою доданою сферою

Третій прецедент – можливість управління камерою. Для тестування було проведено огляд сцени з кількох різних координат. Зображено на рисунках 4.3 та 4.4.

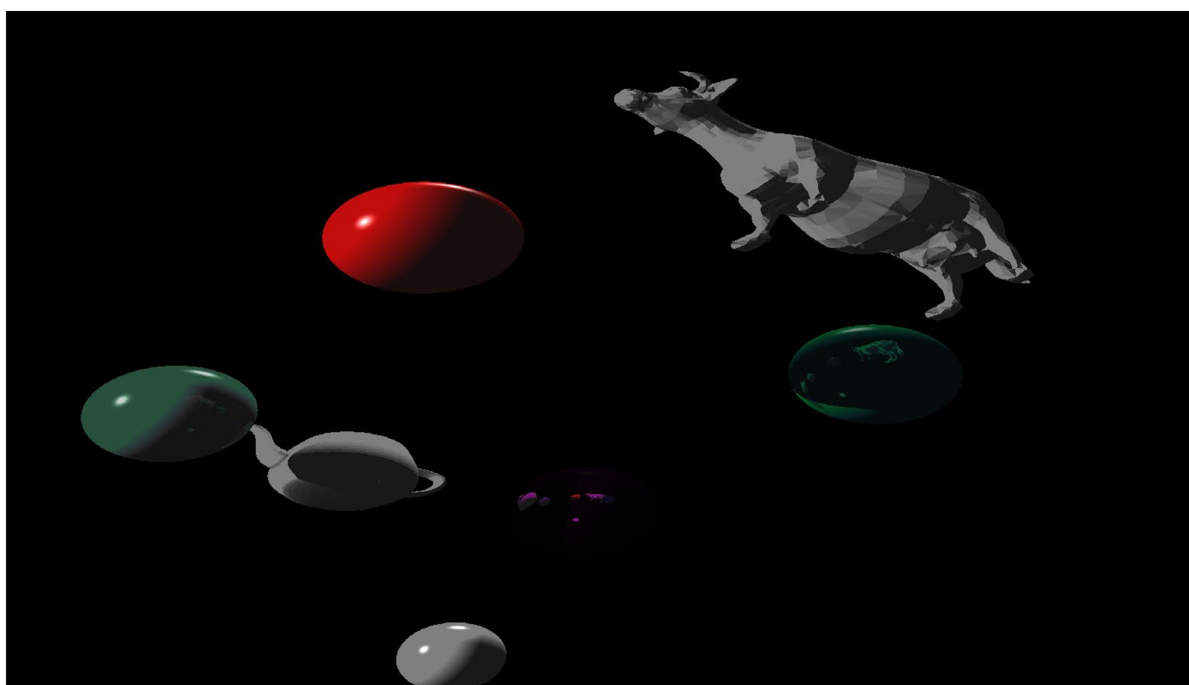


Рис. 4.3 Перша тестова сцена під другим кутом огляду

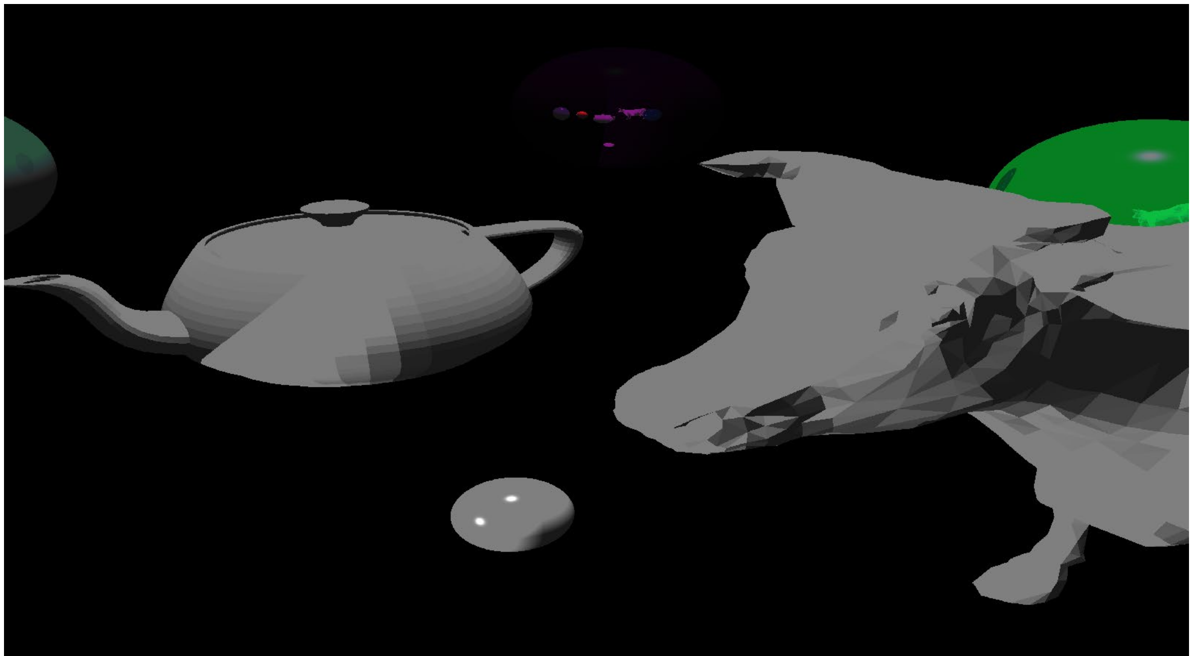


Рис. 4.4 Перша тестова сцена під третім кутом огляду та з переміщеними об'єктами

Останній прецедент для перевірки – налаштування параметрів рендерінгу, а саме кількість рекурсивних променів при обрахунку віддзеркалення об’єкту. Для цього тесту було створено сцену з кількома дзеркальними сферами та 3D-моделлю. На рисунках 4.5, 4.6, 4.7 можна побачити як одна дзеркальна сфера віддзеркалює інші об’єкти на сцені.

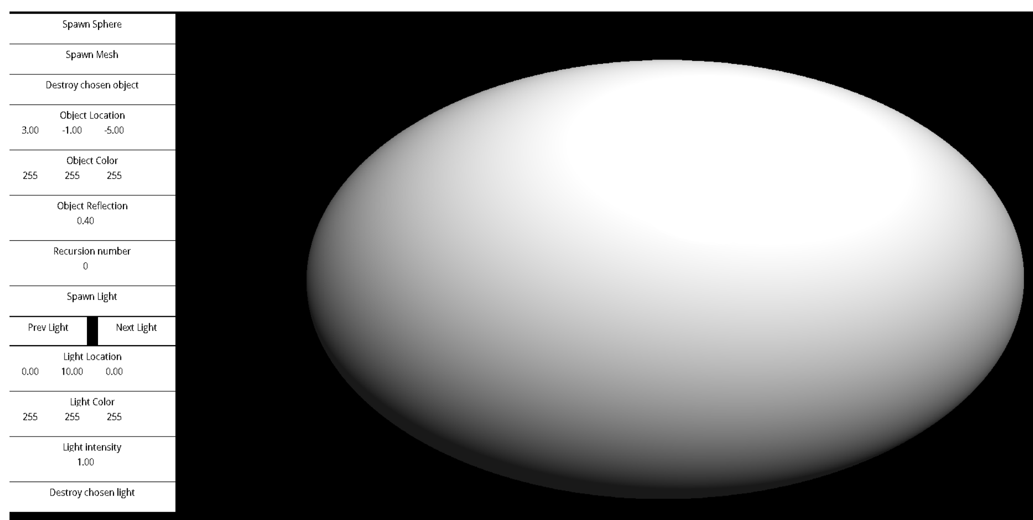


Рис. 4.5 Вид на дзеркальну сферу, при відсутності променів рекурсії

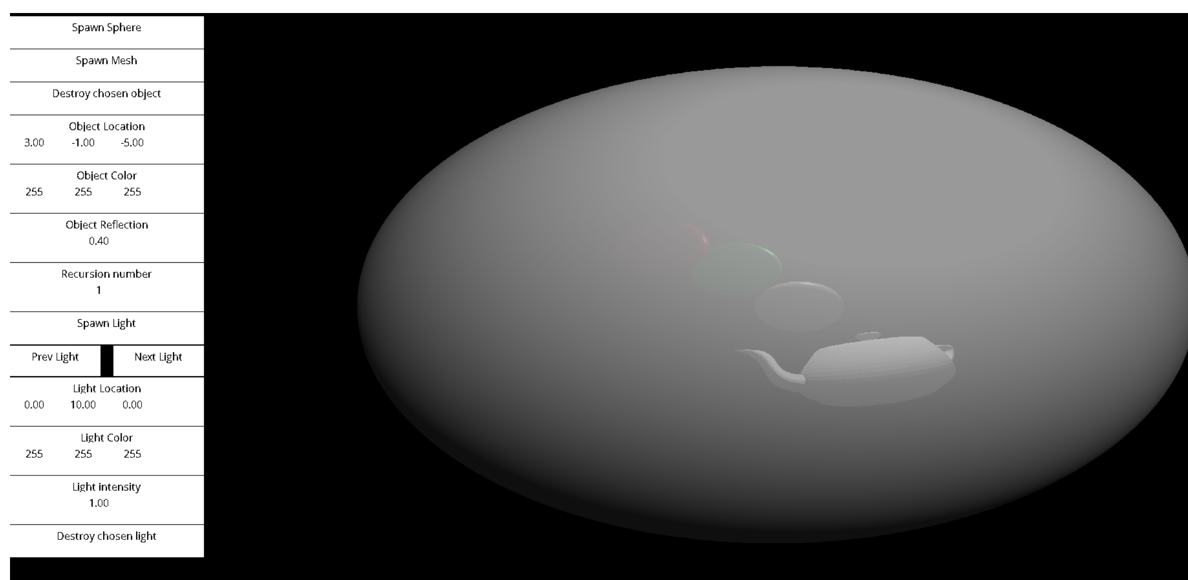


Рис. 4.6 Вид на дзеркальну сферу, при одному промені рекурсії

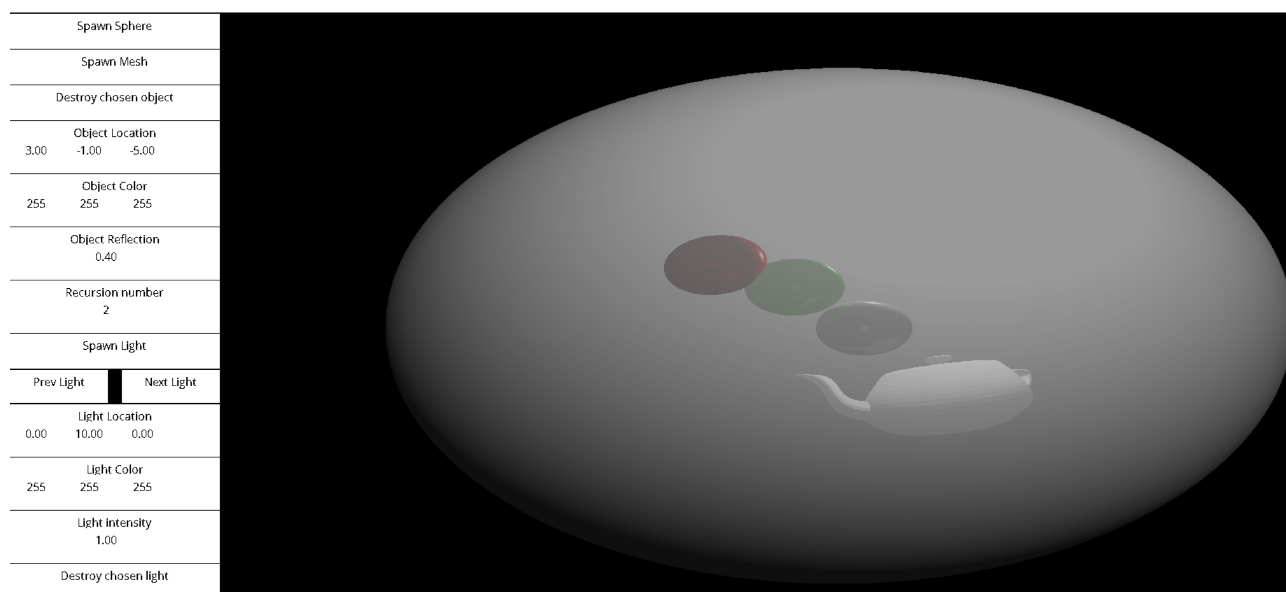


Рис. 4.7 Вид на дзеркальну сферу, при двох променях рекурсії

Як можна побачити на рисунках, кількість променів впливає на якість відображення інших об'єктів на дзеркальній поверхні.

Мануальне тестування застосунку дозволило виявити та виправити кілька помилок, наприклад не працювало налаштування кількості рекурсивних променів, що забезпечило більш стабільну та коректну роботу застосунку. Завдяки структурованому підходу до тестування вдалося перевірити всі основні

функціональні елементи інтерфейсу, оцінити зручність використання та забезпечити високу якість кінцевого продукту.

У результаті мануального тестування були досягнуті наступні результати:

- Виявлено та виправлено помилки в рендерінгу графічних елементів.
- Забезпечено коректну обробку подій миші та клавіатури.
- Забезпечено правильний функціонал всіх елементів UI.

Мануальне тестування стало важливим етапом у розробці застосунку, що дозволило досягти більш високої якості продукту.

Висновки до четвертого розділу

У четвертому розділі було проведено тестування застосунку для рендерінгу та редагування тривимірної сцени. Основною метою тестування було перевірити функціональність та стабільність роботи застосунку в реальних умовах.

Мануальне тестування включало перевірку всіх основних сценаріїв використання застосунку, визначених у функціональних вимогах. Було проведено тестування рендерінгу сцен, додавання та видалення об'єктів, управління камерою та налаштування параметрів рендерінгу.

Тестування показало, що застосунок коректно відображає 3D сцени, дозволяє користувачам додавати та видаляти об'єкти, змінювати їх позицію, а також налаштовувати параметри рендерінгу. Зокрема, було перевірено можливість управління камерою та перегляд сцени з різних кутів огляду, а також налаштування кількості рекурсивних променів при обрахунку віддзеркалення об’єктів.

Під час тестування були виявлені та виправлені кілька помилок, що забезпечило більш стабільну та коректну роботу застосунку. Наприклад, було виправлено помилки в налаштуванні кількості рекурсивних променів, що дозволило досягти більш високої якості відображення дзеркальних поверхонь.

Результати тестування підтвердили, що застосунок відповідає заявленим функціональним вимогам та забезпечує високу якість рендерінгу тривимірних сцен. Мануальне тестування також дозволило оцінити зручність використання інтерфейсу та взаємодії з користувачем.

ВИСНОВКИ

Таким чином, розроблено програмний продукт для рендерінгу та редагування тривимірної сцени, який має можливість рендерити на екрані 3D-об'єкти. В ході виконання роботи було отримано наступні результати:

1. Проведено аналіз та характеристику тривимірних графічних редакторів, що допомогло зрозуміти що вони роблять та дати означення основним поняттям, таким як рендерінг.

2. Проведено аналіз існуючих програмних рішень для редагування тривимірної сцени, який дав інформацію про те, які недоліки існуючі рішення мають та показав актуальність створення програмного продукту, побудованого спрощення рендерінгу.

3. Вибрано засоби для розробки застосунку: OpenCL для роботи з відеокартою, бо це проста для освоєння технологія яка працює з відеокартами різних виробників та бібліотеку SDL через її кроссплатформенністьПроведено моделювання вимог до програмного забезпечення і на основі цього обрано основу для архітектури застосунку, що дозволило вибрати певний підхід до розробки всіх елементів застосунку.

4. Проведено моделювання вимог до програмного забезпечення і на основі цього обрано основу для архітектури застосунку, архітектуру Pipeline, яка розбиває роботу застосунку на окремі послідовні кроки.

5. Спроектовано інтерфейс користувача, який додає такі можливості користувачу як взаємодія з об’єктами на сцені, відображення де вони знаходяться

6. Розроблено застосунок для рендерінгу та редагування тривимірної сцени.

7. Проведено мануальне тестування застосунку, що допомогло виявити та виправити проблеми та помилки, що виникли під час розробки застосунку.

8. Апробація результатів дослідження:

Черкас Б. В., Золотухіна О. А. Проблеми застосування рейтрейсінгу в іграх: Матеріали Всеукраїнської науково-технічної конференції “Застосування програмного забезпечення в інформаційно-комунікаційних технологіях”. 24 квітня 2024р., Київ, Державний університет інформаційно-телекомунікаційних технологій. Збірник тез. К: ДУІКТ, 2024. С. 370-371

Перспективами подальшого розвитку системи є розширення можливостей рендерінгу шляхом додавання текстурування об’єктів та зображення на задньому фоні. Також доцільною опцією стане додавання підтримки імпорту більшої кількості форматів 3D-моделей та дослідження і впровадження інших методів оптимізації рендерінгу.

ПЕРЕЛІК ПОСИЛАНЬ

1. Черкас Б. В. Особливості застосування рейтрейсінгу в іграх. *Застосування програмного забезпечення в інформаційно-комунікаційних технологіях* : Всеукр. науково-техн. конф., м. Київ, 24 квіт. 2024 р. Київ, 2024. С. 370–371.
2. Suffern K. Ray Tracing from the Ground Up. A K Peters/CRC Press, 2016. 784 с.
3. Autodesk 3Ds Max [Електронний ресурс] – Режим доступу: https://www.autodesk.com/products/3ds-max/overview
4. Blender [Електронний ресурс] – Режим доступу: https://www.blender.org/
5. Документація CUDA Toolkit 12.5. NVIDIA Documentation Hub - NVIDIA Docs. [Електронний ресурс] – Режим доступу: https://docs.nvidia.com/cuda/
6. OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems. The Khronos Group [Електронний ресурс] – Режим доступу: https://www.khronos.org/opencl/
7. Simple DirectMedia Layer. Simple DirectMedia Layer - Homepage. [Електронний ресурс] – Режим доступу: https://libsdl.org/index.php
8. OpenGL - The Industry Standard for High Performance Graphics. OpenGL - The Industry Standard for High Performance Graphic [Електронний ресурс] – Режим доступу
9. Специфікація формату obj, loc.gov [Електронний ресурс] – Режим доступу: https://www.loc.gov/preservation/digital/formats/fdd/fdd000507.shtml
10. Ray Tracing Gems II / ed. by A. Marrs, P. Shirley, I. Wald. Berkeley, CA : Apress, 2021. 914 с.

11. Phong B. T. Illumination for computer generated pictures. Communications of the ACM. 1975. Vol. 18, no. 6. P. 311–317. https://doi.org/10.1145/360825.360839

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка застосунку для рендерінгу та редагування 3D сцени мовою C++

Виконав студент 4 курсу
групи ПД-41
Черкас Богдан Васильович
Керівник роботи
К.т.н., доц., доцент кафедри ІПЗ Ільїн Олег Юрійович

Київ – 2024

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

- **Мета роботи:** спрощення рендерінгу та редагування 3D сцен за рахунок використання рейтрейсінгу.
- **Об’єкт дослідження:** процес рендерінгу та редагування 3D сцен.
- **Предмет дослідження:** програмні засоби рендерінгу та редагування 3D сцен.

2

ЗАДАЧІ ДИПЛОМНОЇ РОБОТИ

1. Провести аналіз методів застосунків для рендерінгу та редагування 3D сцен, в тому числі, особливості використання рейтрейсінгу.
2. Провести аналіз та обґрунтування вибору засобів розробки застосунку для рендерінгу та редагування 3D сцени.
3. Виконати моделювання вимог до застосунку для рендерінгу та редагування 3D сцени.
4. Провести проектування архітектури та інтерфейсу користувача.
5. Розробити застосунок для рендерінгу та редагування 3D-сцени.
6. Провести тестування застосунку.

3

АНАЛІЗ АНАЛОГІВ

Властивості	3Ds Max	Blender	LightWave 3D	**RTracer**
Редагування сцени	+	+	+	+
Підтримка різних видів освітлення	+	-	+	+
Проста у використанні для новачків	-	-	+	+
Рендерінг з реалістичним освітленням за допомогою рейтрейсінгу	+	-	+	+
Висока швидкість рендерінгу в реальному часі при обмежених ресурсах	-	-	-	+
Платформа	Windows	Windows OS X Linux	Windows OS X	Windows Linux

4

ВИМОГИ ДО ЗАСТОСУНКУ

Функціональні вимоги

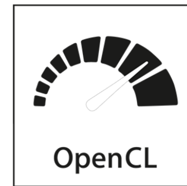
1. Можливість рендерінгу 3D сцен за допомогою рейтрейсінгу.
2. Додавання та видалення об'єктів.
3. Управління камерою.
4. Підтримка імпорту 3D-моделей та експорту створених сцен.
5. Налаштування освітлення.

Нефункціональні вимоги

1. Висока швидкість рендерінгу в реальному часі - не менше ніж 30 FPS.
2. Можливість роботи під управлінням операційних систем Windows та Linux.
3. Сумісність з відеокартами від різних виробників та підтримку різних API, зокрема Nvidia та AMD.
4. Оптимізована робота з пам'яттю, зокрема пам'яттю відеокарти

5

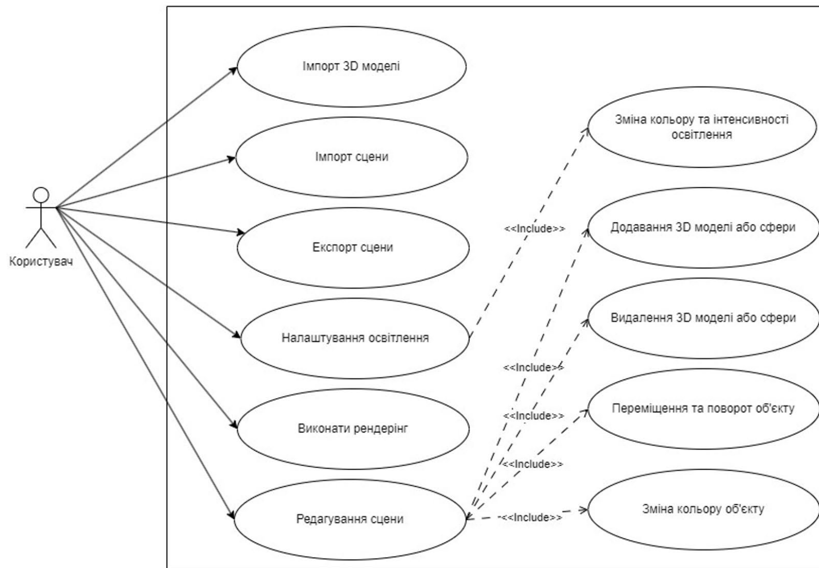
ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ



Visual Studio 2022

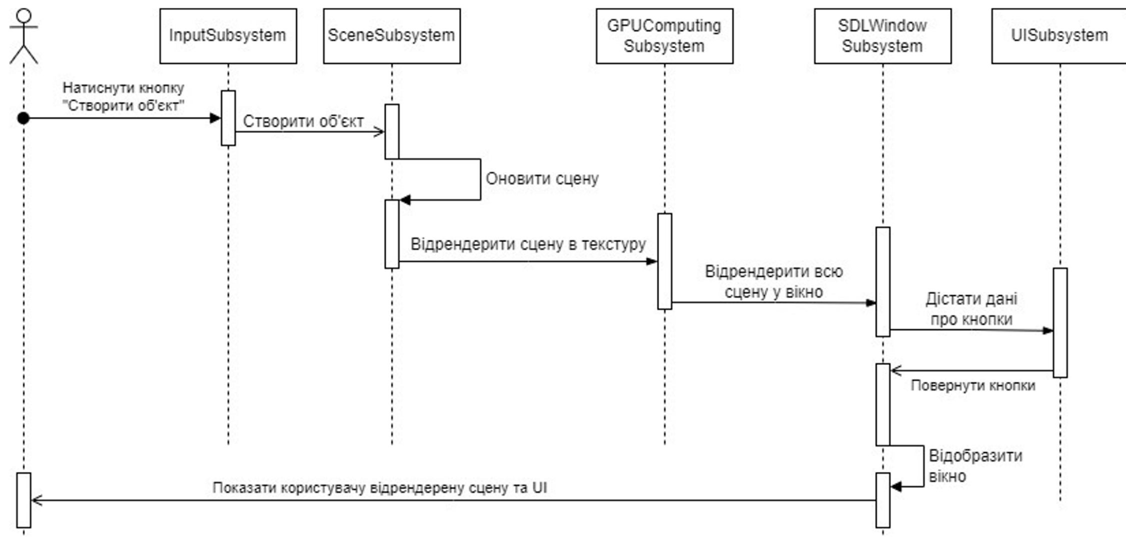
6

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ



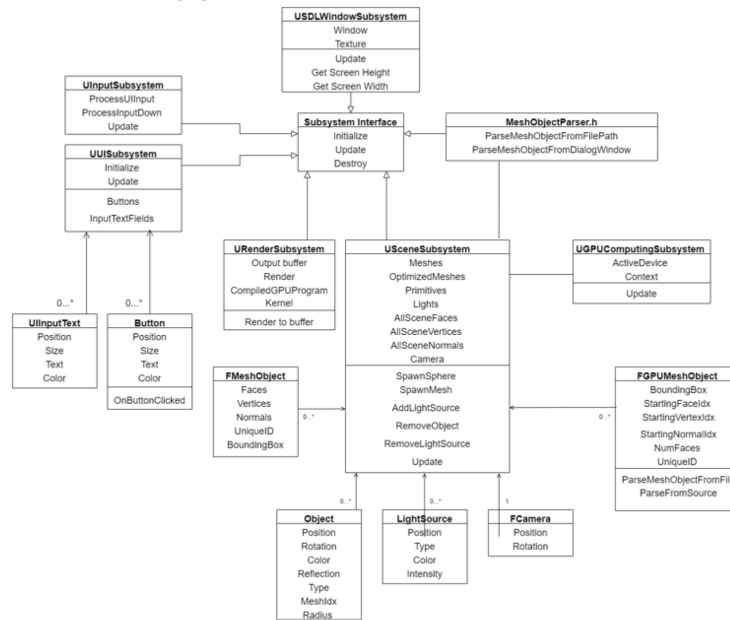
7

ДІАГРАМА ПОСЛІДОВНОСТІ РОБОТИ ЗАСТОСУНКУ ПРИ СТВОРЕННІ ОБ'ЄКТУ НА СЦЕНІ



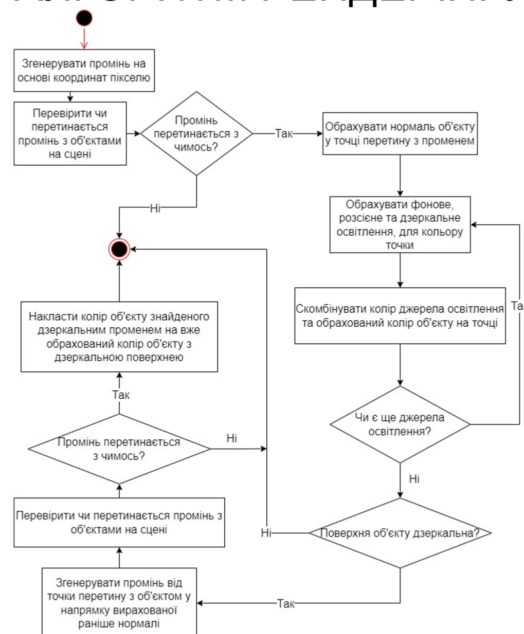
8

ДІАГРАМА КЛАСІВ



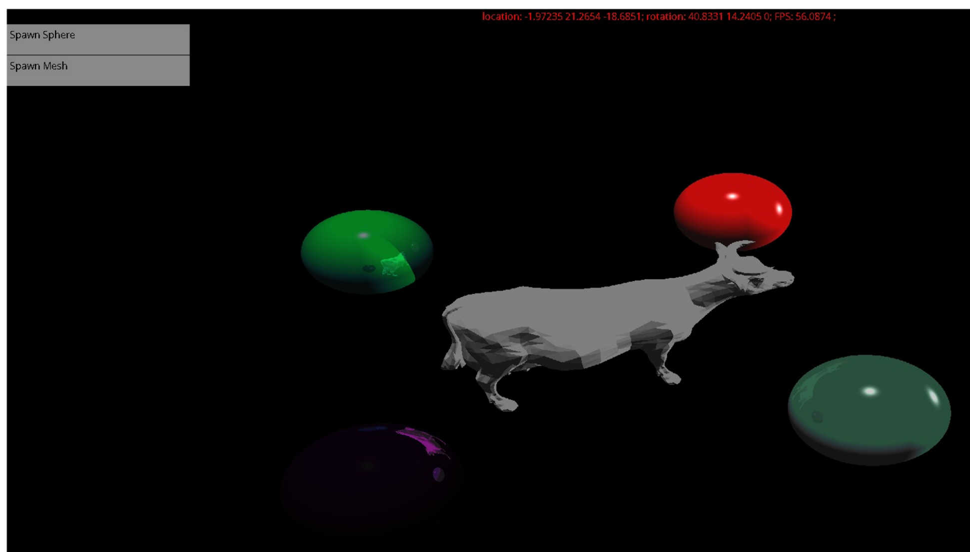
9

АЛГОРИТМ РЕНДЕРІНГУ



10

ЕКРАННІ ФОРМИ



Сцена 1: 4 сфери різного кольору та з різним віддзеркаленням ,
2 джерела світла, модель корови. Перший ракурс

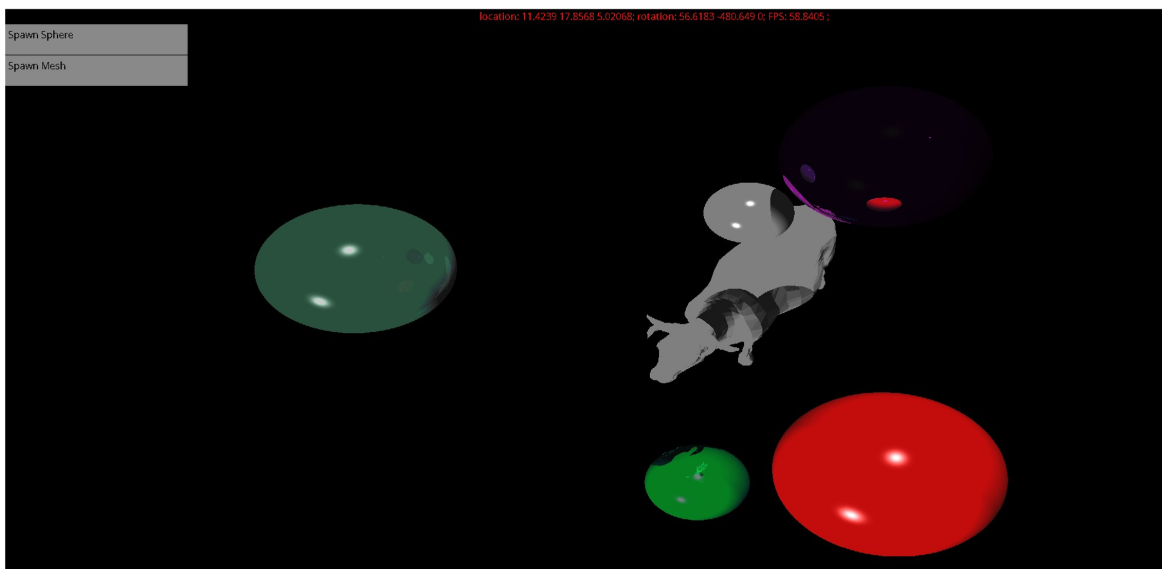
11

ЕКРАННІ ФОРМИ

Сцена 1: 4 сфери різного кольору та з різним віддзеркаленням ,
2 джерела світла, модель корови. Другий ракурс

12

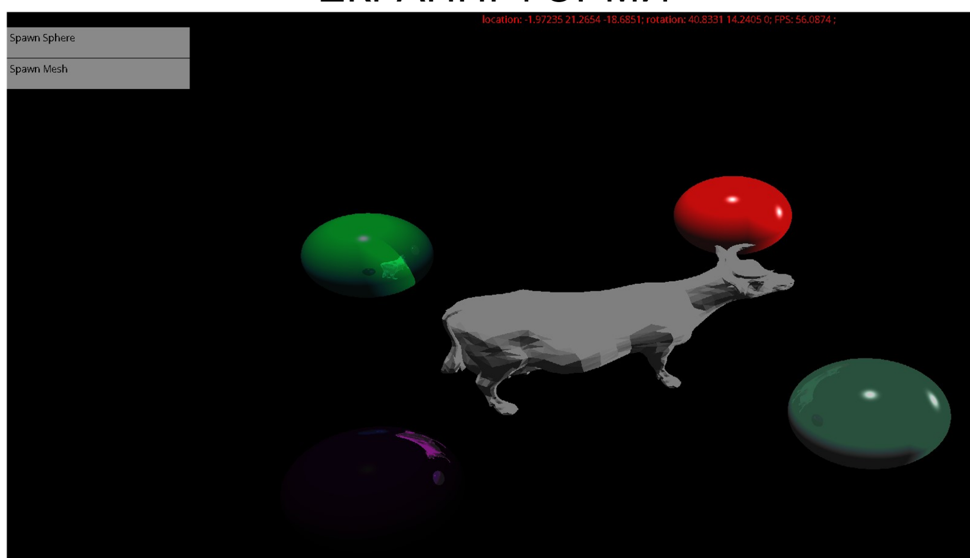
ЕКРАННІ ФОРМИ



Сцена 1: 4 сфери різного кольору та з різним віддзеркаленням ,
2 джерела світла, модель корови. Третій ракурс - об’єкти переміщені, додано сферу

13

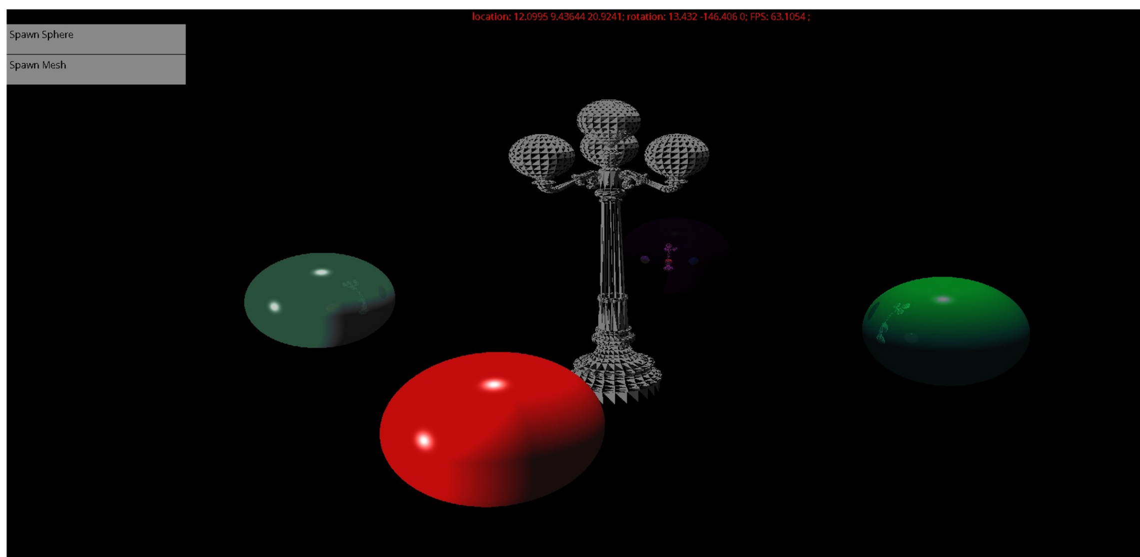
ЕКРАННІ ФОРМИ



Сцена 1: 4 сфери різного кольору та з різним віддзеркаленням ,
2 джерела світла, модель корови

14

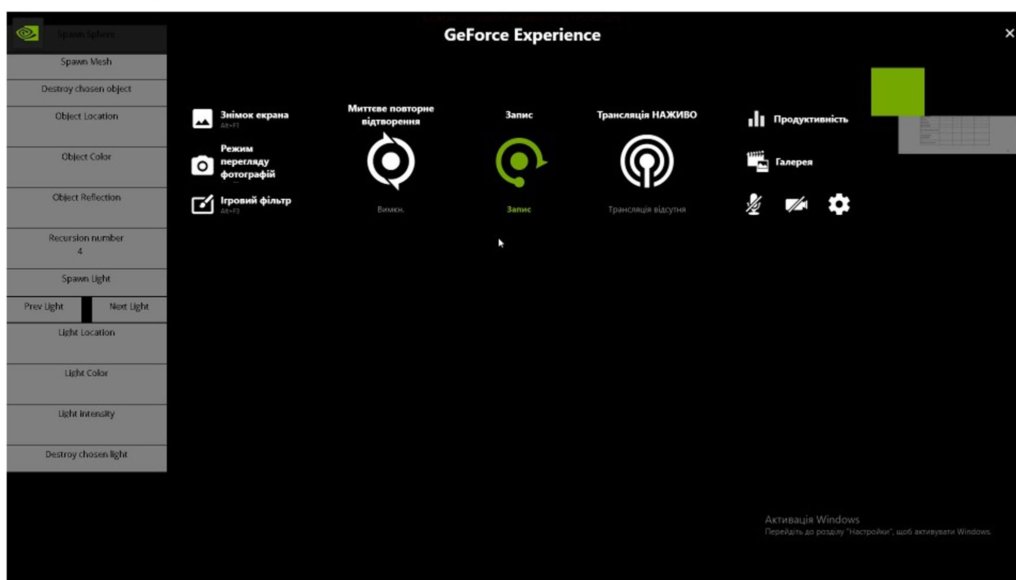
ЕКРАННІ ФОРМИ



Сцена 2: 4 сфери різного кольору та з різним віддзеркаленням ,
2 джерела світла, модель вуличного ліхтаря

15

ДЕМОНСТРАЦІЯ РОБОТИ ПРОГРАМИ



16

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Черкас Б. В., Золотухіна О. А. Проблеми застосування рейтрейсінгу в іграх: Матеріали Всеукраїнської науково-технічної конференції "Застосування програмного забезпечення в інформаційно-комунікаційних технологіях". 24 квітня 2024р., Київ, Державний університет інформаційно-телекомунікаційних технологій. Збірник тез. К: ДУІКТ, 2024. С. 370-371

17

ВИСНОВКИ

1. Проведено аналіз та характеристику існуючих тривимірних графічних редакторів за рахунок чого було виявлено їх сильні і слабкі сторони та було сформовано бачення, який функціонал повинен мати застосунок.
2. Вибрано засоби для розробки застосунку: OpenCL для роботи з відеокартою, бо це проста для освоєння технологія яка працює з відеокартами різних виробників та бібліотеку SDL через її кроссплатформенність.
3. Проведено моделювання вимог до програмного забезпечення і на основі цього обрано основу для архітектури застосунку, архітектуру Pipeline, яка розбиває роботу застосунку на окремі послідовні кроки.
4. Спроектовано інтерфейс користувача, який додає такі можливості користувачу як взаємодія з об'єктами на сцені, відображення де вони знаходяться.
5. Розроблено застосунок для рендерінгу та редагування тривимірної сцени.
6. Здійснено мануальне тестування застосунку.

18

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

FMath.h

```
#pragma once

#define CL_HPP_TARGET_OPENCL_VERSION 300

#include <CL/cl.hpp>
#include <cmath>
#include "SDL_stdinc.h"

struct FRotator;
struct FQuaternion;

struct FQuaternion
{
	cl_float4 InnerQuat;

	FQuaternion(cl_float inX = 0.f, cl_float inY = 0.f, cl_float inZ = 0.f, cl_float inW = 1.f)
		: InnerQuat{inX, inY, inZ, inW} {}

	FQuaternion(cl_float4 InQuatVector)
		: InnerQuat{ InQuatVector } {}

	FRotator ToRotator() const;

	FQuaternion& operator=(const FQuaternion& q)
	{
		InnerQuat = q.InnerQuat;
		return *this;
	}

	cl_float4 RotateVector(cl_float4 InVector) const;

	FQuaternion operator*(const FQuaternion& OtherQuat) const
	{
		return FQuaternion(
			InnerQuat.w * OtherQuat.InnerQuat.x + InnerQuat.x * OtherQuat.InnerQuat.w + InnerQuat.y * OtherQuat.InnerQuat.z - InnerQuat.z * OtherQuat.InnerQuat.y,
			InnerQuat.w * OtherQuat.InnerQuat.y - InnerQuat.x * OtherQuat.InnerQuat.z + InnerQuat.y * OtherQuat.InnerQuat.w + InnerQuat.z * OtherQuat.InnerQuat.x,
			InnerQuat.w * OtherQuat.InnerQuat.z + InnerQuat.x * OtherQuat.InnerQuat.y - InnerQuat.y * OtherQuat.InnerQuat.x + InnerQuat.z * OtherQuat.InnerQuat.w,
			InnerQuat.w * OtherQuat.InnerQuat.w - InnerQuat.x * OtherQuat.InnerQuat.x - InnerQuat.y * OtherQuat.InnerQuat.y - InnerQuat.z * OtherQuat.InnerQuat.z
		);
	}

	FQuaternion Conjugate() const
	{
		return FQuaternion(-InnerQuat.x, -InnerQuat.y, -InnerQuat.z, InnerQuat.w);
	}

};

struct FRotator
{
	// Representation in degrees
	cl_float Roll;
	cl_float Pitch;
	cl_float Yaw;

	cl_float4 ToFloat4() const;
	FQuaternion ToQuaternion() const;

	static FRotator FromFloat4Degrees(cl_float4 InData);
	static FRotator FromFloat4Radians(cl_float4 InData);
};

class FMath
{
public:
	static cl_float GetRadians(cl_float Angle);
	static cl_float GetDegrees(cl_float Angle);

	static cl_float4 GetRadiansVector(cl_float4 Angle);
	static cl_float4 GetDegreesVector(cl_float4 Angle);

	static bool SquareEquation(cl_float3 Parameters, cl_float& OutResult);

	static cl_float4 RotateVectorByDegrees(cl_float4 InVector, cl_float4 InEulerDegrees);
	static cl_float4 RotateVectorByRadians(cl_float4 InVector, cl_float4 InEulerRadians);

	static cl_float VectorLength(cl_float4 v)
	{
		return sqrt(v.x * v.x + v.y * v.y + v.z * v.z);
	}

	static cl_float4 NormalizeVector(cl_float3 v)
	{
		cl_float len = VectorLength(v);
		if (len > 0.0f)
		{
			v.x /= len;
```

```
				v.y /= len;
				v.z /= len;
			}
			return v;
		}

		static cl_float4 SetVectorLength(cl_float3 v, cl_float newlength)
		{
			cl_float4 Normalized = NormalizeVector(v);
			Normalized.x /= fabs(newlength);
			Normalized.y /= fabs(newlength);
			Normalized.z /= fabs(newlength);
			return Normalized;
		}

		static cl_float4 CrossProduct(cl_float4 A, cl_float4 B)
		{
			cl_float4 Result;
			Result.x = A.y * B.z - A.z * B.y;
			Result.y = A.z * B.x - A.x * B.z;
			Result.z = A.x * B.y - A.y * B.x;
			return Result;
		}

		static float DotProduct(cl_float4 A, cl_float4 B)
		{
			return {A.x * B.x + A.y * B.y + A.z * B.z};
		}

		static bool sq_equation(cl_float4 InEquation, float& OutResult)
		{
			float	t[2];
			float	discriminant;
			float	tmp;

			discriminant = pow(InEquation.y, 2) - (4 * InEquation.x * InEquation.z);
			if (discriminant < 0)
				return (0);
			else if (discriminant > 0)
				discriminant = sqrt(discriminant);
			t[0] = (-InEquation.y - discriminant) / InEquation.x / 2;
			t[1] = (-InEquation.y + discriminant) / InEquation.x / 2;

			if (t[0] < 0 && t[1] < 0)
			{
				return false;
			}

			if (t[0] >= 0 && t[1] >= 0)
			{
				OutResult = std::min(t[0], t[1]);
			}
			else
			{
				OutResult = std::max(t[0], t[1]);
			}
			return (true);
		}

		static cl_float4 VectorAdd(cl_float4 A, cl_float4 B)
		{
			return { A.x + B.x, A.y + B.y, A.z + B.z };
		}
};
```

GPUComputingSubsystem.h

```
#pragma once

#define CL_HPP_TARGET_OPENCL_VERSION 300

#include "ISubsystemInterface.h"
#include <CL/cl.hpp>
#include <vector>
#include "Object.h"

class USceneSubsystem;

class IGPUProgram
{
public:
	IGPUProgram() = delete;
	IGPUProgram(cl::Context& InContext, cl::CommandQueue& InCommandQueue, cl::Device& InActiveDevice);

	cl::Context& Context;
	cl::CommandQueue& CommandQueue;
	cl::Device& ActiveDevice;

	cl::Program Program;
	cl::Buffer OutputBuffer;
	cl::Kernel Kernel;

	virtual void Initialize() = 0;
	virtual bool RunProgram() = 0;
	virtual void PrepareProgram() = 0;

	cl::Buffer PrimitivesBuffer;
	cl::Buffer LightsBuffer;
	cl::Buffer SceneDataBuffer;
	cl::Buffer MeshesBuffer;
	cl::Buffer VerticesBuffer;
	cl::Buffer FacesBuffer;
	cl::Buffer NormalsBuffer;

};

class MainRaytracingProgram : public IGPUProgram
```

```
{
public:
	MainRaytracingProgram() = delete;
	MainRaytracingProgram(cl::Context& InContext, cl::CommandQueue& InCommandQueue, cl::Device& InActiveDevice);
	virtual void Initialize() override;
	virtual bool RunProgram() override;
	virtual void PrepareProgram() override;
};

class UGPUComputingSubsystem : public ISubsystemInterface
{
public:
	// Inherited via ISubsystemInterface
	bool Initialize() override;
	bool Update(float InDeltaTime) override;
	std::string GetSubsystemName() const override;
	void Destroy() override;

	static UGPUComputingSubsystem& Get()
	{
		static UGPUComputingSubsystem GPUComputingSubsystem;
		return GPUComputingSubsystem;
	}

	std::unique_ptr<MainRaytracingProgram> MainProgram;

private:

	void PrintDeviceInfo(const cl::Device& InDevice) const;

	std::vector<cl::Platform> Platforms;
	cl::Platform ActivePlatform;
	std::vector<cl::Device> Devices;
	cl::Device ActiveDevice;
	cl::Context Context;
	cl::CommandQueue CommandQueue;
};
```

IntersectTest.h

```
#pragma once

#define CL_HPP_TARGET_OPENCL_VERSION 300
#define EPSILON 0.0001f

#include <CL/cl.hpp>
#include "Object.h"
#include "USceneSubsystem.h"
#include "FMath.h"

struct Ray
{
	cl_float4 Origin;
	cl_float4 Direction;
	cl_float DirectionDot;

	static Ray CreateRay(cl_float4 InOrigin, cl_float4 InDirection);
};

struct Intersection
{
	bool hit;
	float t;
	Vertex point;
	Normal normal;
	std::shared_ptr<FMeshObject> Object;
	FObject Primitive;
};

bool RaySphereIntersect(const Ray& ray, const FObject& sphere, float& OutDistance);
bool RayTriangleIntersect(const Ray& ray, const Vertex& v0, const Vertex& v1, const Vertex& v2, float& t, float& u, float& v);
Intersection RayMeshIntersect(const Ray& ray, const FMeshObject& mesh, const FObject& Primitive);
Intersection DoPixelIntersectTest(cl_int x, cl_int y);
```

ISubsystemInterface.h

```
#pragma once

#include <string>

// Interface that declare basic subsystem methods
class ISubsystemInterface
{
public:

	ISubsystemInterface() {}

	ISubsystemInterface(const ISubsystemInterface&) = delete;

	ISubsystemInterface& operator=(const ISubsystemInterface&) = delete;

	virtual bool Initialize() = 0;

	virtual bool Update(float InDeltaTime) = 0;

	virtual std::string GetSubsystemName() const = 0;

	virtual void Destroy() = 0;

};
```

Layout.h

```
#pragma once
```

```
#define CL_HPP_TARGET_OPENCL_VERSION 300

#include <CL/cl.hpp>
#include "SDL.h"
#include "Object.h"

struct FOnButtonClickedDelegate
{
	std::function<void(bool)> Pointer;

	void Bind(std::function<void(bool)>
InFunction)
	{
		Pointer = InFunction;
	}

	void ExecuteIfBound(bool bClicked)
	{
		if (Pointer)
		{
			Pointer(bClicked);
		}
	}
};

struct FOnTextUpdatedDelegate
{
	std::function<void(std::string)> Pointer;

	void BindStr(std::function<void(std::string)>
InFunction)
	{
		Pointer = InFunction;
	}

	void ExecuteIfBound(std::string InText)
	{
		if (Pointer)
		{
			Pointer(InText);
		}
	}
};

class ILayoutElement
{
public:

	cl_bool IsPointedByMouse(cl_int InX, cl_int
InY);

	virtual void Render() = 0;
	virtual cl_float2 GetSize() const = 0;
	virtual cl_float2 GetPosition() const = 0;
	virtual const FColor& GetBorderColor() const
= 0;
	virtual const FColor& GetFillColor() const =
0;
};

class UIPanel : public ILayoutElement
{
public:
	static std::shared_ptr<UIPanel>
CreateUIPanel(cl_int2 InPosition, cl_int2 InSize,
FColor BorderColor, FColor FillColor);

	UIPanel() = delete;
	UIPanel(cl_int2 InPosition, cl_int2 InSize,
FColor BorderColor, FColor FillColor);


	void Render() override;
	cl_float2 GetSize() const override;
	cl_float2 GetPosition() const override;
	const FColor& GetBorderColor() const
override;
	const FColor& GetFillColor() const override;

	SDL_Rect Rect;
	FColor BorderColor;
	FColor FillColor;
};

class UIButton : public ILayoutElement
{
public:
	static std::shared_ptr<UIButton>
CreateUIButton(cl_int2 InPosition, cl_int2 InSize,
std::string InText, FColor InTextColor, FColor
BorderColor, FColor FillColor, cl_int2 InPadding = {
0, 0 });

	UIButton() = delete;
	UIButton(cl_int2 InPosition, cl_int2 InSize,
std::string InText, FColor InTextColor, FColor
BorderColor, FColor FillColor, cl_int2 InPadding = {
0, 0 });

	virtual cl_float2 GetSize() const override;
	virtual cl_float2 GetPosition() const override;
	virtual const FColor& GetBorderColor() const
override;
	virtual const FColor& GetFillColor() const
override;
	virtual void Render() override;
	void UpdateText(std::string NewText, FColor
InTextColor, cl_int2 InPadding = { 0, 0 });

	bool IsHovered() const;

	void OnButtonClicked(bool bClicked);

	std::unique_ptr<SDL_Texture,
std::function<void(SDL_Texture*)>> Texture;
	std::string Text;
	SDL_Rect Rect;
	FColor TextColor;
	FColor BorderColor;
	FColor FillColor;
	cl_int2 Padding;
	SDL_Rect TextRect;
```

```
	FOnButtonClickedDelegate
OnButtonClickedDelegate;

	bool bIsHovered = false;
	bool bIsClicked = false;
};

class UIInputText : public ILayoutElement
{
public:
	static std::shared_ptr<UIInputText>
CreateUIInputText(cl_int2 InPosition, cl_int2 InSize,
FColor InTextColor, FColor BorderColor, FColor
FillColor, cl_int2 InPadding = { 5, 5 });

	UIInputText() = delete;
	UIInputText(cl_int2 InPosition, cl_int2 InSize,
FColor InTextColor, FColor InBorderColor, FColor
InFillColor, cl_int2 InPadding = { 5, 5 }) :
		Texture(nullptr,
&SDL_DestroyTexture),
		Rect{ (int)InPosition.x,
(int)InPosition.y, (int)InSize.x, (int)InSize.y },
		TextColor(InTextColor),
		BorderColor(InBorderColor),
		FillColor(InFillColor),
		Padding(InPadding),
		TextRect(0, 0, 0, 0)
	{
	}

	virtual cl_float2 GetSize() const override {
return { (cl_float)Rect.w, (cl_float)Rect.h }; }
	virtual cl_float2 GetPosition() const override {
return { (cl_float)Rect.x, (cl_float)Rect.y }; }
	virtual const FColor& GetBorderColor() const
override { return BorderColor; }
	virtual const FColor& GetFillColor() const
override { return FillColor; }
	virtual void Render() override;
	void UpdateText(std::string InString, bool
bRemoveSymbol = false);
	void ApplyInput();

	std::unique_ptr<SDL_Texture,
std::function<void(SDL_Texture*)>> Texture;
	std::string Text;
	SDL_Rect Rect;
	FColor TextColor;
	FColor BorderColor;
	FColor FillColor;
	cl_int2 Padding;
	SDL_Rect TextRect;

	bool bIsHovered = false;
	bool bIsClicked = false;
	bool bIsActive = false;

	FOnTextUpdatedDelegate
OnTextUpdatedDelegate;
};
```

MeshObjectParser.h

```
#pragma once

#include <math.h>
#include <memory>
#include "Object.h"
#include "ISubsystemInterface.h"

class UMeshObjectParserSubsystem :
ISubsystemInterface
{
public:

	UMeshObjectParserSubsystem() {}

	// Inherited via ISubsystemInterface
	bool Initialize() override;
	bool Update(float InDeltaTime) override {
return false; }
	void Destroy() override;
	std::string GetSubsystemName() const
override;

	static UMeshObjectParserSubsystem& Get()
	{
		static UMeshObjectParserSubsystem
MeshObjectParserSubsystem;
		return MeshObjectParserSubsystem;
	}

	std::shared_ptr<FMeshObject>
ParseMeshObjectFromFile(std::wstring InFilePath);
	std::shared_ptr<FMeshObject>
ParseMeshObjectFromDialogWindow();

private:

	std::wstring OpenFileWinAPI();

};
```

Object.h

```
#pragma once

#define CL_HPP_TARGET_OPENCL_VERSION 300

#include <CL/cl.hpp>

using Vertex = cl_float4;
using Normal = cl_float4;
using FColor = cl_int4;
using UV = cl_float2;

enum class EObjectType : cl_uint
{
	Sphere = 1,
```

```
	Plane = 2,
	Mesh = 3
};

struct alignas(16) FFace
{
	cl_int4 Vertices = { -1, -1, -1 };
	cl_int4 Normals = {-1, -1, -1};
	cl_int4 TexCoords = { -1, -1, -1 };

	cl_float4 PreComputedNormal;
};

struct alignas(16) FMeshObject
{
	cl::vector<FFace> Faces;
	cl::vector<Vertex> Vertices;
	cl::vector<Normal> Normals;
	cl::vector<UV> TextureCoords;

	cl_float4 BoundingBox[2];

	cl_int UniqueID = -1;
};

struct alignas(16) FGPUMeshObject
{
	cl_float4 BoundingBox[2];
	cl_int StartingFaceIdx = 0;
	cl_int StartingVertexIdx = 0;
	cl_int StartingNormalIdx = 0;
	cl_int StartingTexCoordIdx = 0;
	cl_int NumFaces = 0;
	cl_int UniqueID = -1;
};

struct alignas(16) FObject
{
	cl_float4 center = {0, 0, 0};
	cl_float4 axis = { 0, 0, 0 };
	cl_float4 normal = { 0, 0, 0 };
	FColor color = { 0, 0, 0 };
	cl_float pow_radius = 0;
	cl_float reflection = 0;
	cl_int specularity = -1;
	cl_int meshidx = 0;
	cl_int type = 0;

	cl_int Idx = 0;
};

struct alignas(16) FSphereObject : public FObject
{
	FSphereObject(cl_float3 InOrigin, cl_float
InRadius, FColor InColor, cl_float InReflection);

	static FObject ConstructSphere(cl_float3
InOrigin, cl_float InRadius, FColor InColor, cl_float
InReflection);

};
```

Pixel.h

```
#pragma once

#define CL_HPP_TARGET_OPENCL_VERSION 300
#include "Cl/cl.hpp"

using Pixel = cl_uint;
```

SDLWindowSubsystem.h

```
#pragma once

#include "ISubsystemInterface.h"
#include "SDL.h"
#include "SDL_ttf.h"
#include <memory>
#include <functional>
#include "Pixel.h"
#include "Object.h"

using SDL_UniqueSurface =
std::unique_ptr<SDL_Surface,
std::function<void(SDL_Surface*)>>;
using SDL_UniqueTexture =
std::unique_ptr<SDL_Texture,
std::function<void(SDL_Texture*)>>;
using SDL_UniqueFont = std::unique_ptr<TTF_Font,
std::function<void(TTF_Font*)>>;
using SDL_UniqueWindow =
std::unique_ptr<SDL_Window,
std::function<void(SDL_Window*)>>;
using SDL_UniqueRenderer =
std::unique_ptr<SDL_Renderer,
std::function<void(SDL_Renderer*)>>;

class USDLWindowSubsystem : public
ISubsystemInterface
{
public:

	USDLWindowSubsystem();

	// ISubsystemInterface begin
	virtual bool Initialize() override;
	bool Update(float InDeltaTime) override;
	std::string GetSubsystemName() const
override;
	void Destroy() override;
	// ISubsystemInterface end

	void DrawFontText(std::string InText,
SDL_Color InColor, cl_int2 Position);

	static USDLWindowSubsystem& Get()
	{
		static USDLWindowSubsystem
SDLWindowSubsystem;
```

```
		return SDLWindowSubsystem;
	}

	Pixel* UnlockCanvas() const;
	void LockCanvas() const;

	cl_int GetRenderHeight() const;
	cl_int GetRenderWidth() const;
	cl_int GetRenderPixelsSize() const;

	SDL_UniqueWindow Window;
	SDL_UniqueRenderer Renderer;
	SDL_UniqueTexture MainRenderTexture;
	SDL_UniqueFont Font;
};
```

UInputSubsystem.h

```
#pragma once

#define CL_HPP_TARGET_OPENCL_VERSION 300

#include "ISubsystemInterface.h"
#include <CL/cl.hpp>
#include "SDL.h"

class UInputSubsystem : public ISubsystemInterface
{
public:

	// Inherited via ISubsystemInterface
	bool Initialize() override;
	bool Update(float InDeltaTime) override;
	std::string GetSubsystemName() const
override;
	void Destroy() override;

	bool InnerUpdate(float InDeltaTime);
	void ProcessInputDown(SDL_Event InEvent,
cl_float DeltaTime);
	void ProcessInputUp(SDL_Event InEvent,
cl_float DeltaTime);
	void ProcessMouseMotion(SDL_Event
InEvent, cl_float DeltaTime);
	void ProcessUIInput(SDL_Event InEvent,
cl_float DeltaTime);

	static UInputSubsystem& Get()
	{
		static UInputSubsystem
InputSubsystem;
		return InputSubsystem;
	}

	cl_int2 MousePosition;
	bool bLeftMouseButtonDown = false;
	bool bRightMouseButtonDown = false;

	std::string CurrentInputText;

private:

	bool bHasForwardMovementInput = false;
	bool bHasBackwardMovementInput = false;
	bool bHasLeftMovementInput = false;
	bool bHasRightMovementInput = false;
	bool bHasUpMovementInput = false;
	bool bHasDownMovementInput = false;

	bool bHasUpRotationInput = false;
	bool bHasDownRotationInput = false;
	bool bHasLeftRotationInput = false;
	bool bHasRightRotationInput = false;

	cl_float MovementSpeed = 10.f;
	cl_float RotationSpeed = 1.f;

};
```

USceneSubsystem.h

```
#pragma once

#include "ISubsystemInterface.h"
#include "Object.h"

struct alignas(16) FCamera
{
	cl_float4 Origin = { 0.f, 0.f, 0.f, 0.f };

	// In Radians
	cl_float4 Rotation = { 0.f, 0.f, 0.f, 0.f };
};

struct alignas(16) FLightSource
{
	cl_float4 Origin;
	FColor Color;
	cl_float Intensity = 1.f;
	cl_int Idx = -1;
};

struct alignas(16) FSceneData
{
	FCamera Camera;
	FColor AmbientLightColor = { 0xff, 0xff,
0xff, 0 };

	cl_float ScreenRelativeDistance;
	cl_float ScreenRelativeWidth;
	cl_float ScreenRelativeHeight;

	cl_float RelationX;
	cl_float RelationY;

	cl_int WindowHeight;
	cl_int WindowWidth;

	cl_float AmbientLightIntensity = 1.f;
```

```
	cl_uint NumPixels = 0;
	cl_uint NumFaces = 0;
	cl_uint NumPrimitives = 0;
	cl_uint NumMeshes = 0;
	cl_uint NumLightSources = 0;
	cl_uint NumVertices = 0;
	cl_uint NumNormals = 0;
	cl_uint NumTexCoords = 0;

	cl_int RecursionLevel = 100.f;
};

class alignas(16) USceneSubsystem : public
ISubsystemInterface
{
public:

	USceneSubsystem();

	bool Initialize() override;
	bool Update(float InDeltaTime) override;
	std::string GetSubsystemName() const
override;
	void Destroy() override;

	static USceneSubsystem& Get()
	{
		static USceneSubsystem
SceneSubsystem;
		return SceneSubsystem;
	}

	void SpawnLightSource(cl_float4 Origin =
{0.f, 2.f, 0.f, 0.f}, FColor Color = {255, 255, 255, 0},
cl_float Intensity = 1.f);
	FLightSource* GetChosenLightSource(cl_int*
IndexInArray = nullptr);
	void ChooseNextLightSource();
	void ChoosePrevLightSource();
	void RemoveChosenLightSource();

	void SpawnSphere(cl_float3 InOrigin = {0.f,
0.f, 0.f}, cl_float InRadius = 1.f, FColor InColor =
{127, 127, 127}, cl_float InReflection = 0.5f);
	void SpawnMesh(cl_float3 InOrigin = { 0.f,
0.f, 0.f }, FColor InColor = { 127, 127, 127 }, cl_float
InReflection = 0.f);
	void DestroyChosenObject();

	FObject* GetChosenObject();

	void MoveChosenObject(cl_float4
Movement);
	void MoveChosenSphere(FObject& Sphere,
cl_float4 Movement);
	void MoveChosenMesh(FObject&
MeshPrimitive, cl_float4 Movement);

	void
AddMeshToScene(std::shared_ptr<FMeshObject>
NewMesh);
	void
UpdateMeshInScene(std::shared_ptr<FMeshObject>
InMesh, cl_float4 NewLocation, cl_float4
NewRotation);
	bool FindGPUMeshByID(cl_int InUniqueID,
FGPUMeshObject& OutObject);

	void OnSceneUpdated();

	void
ComputeBoundingBoxForMesh(FMeshObject&
InMesh);

	//MeshObject&
ChooseObjectViaIntersect(cl_int2 IntersectionPixel);

	std::unique_ptr<FSceneData> SceneData;

	cl::vector<std::shared_ptr<FMeshObject>>
Meshes;

	cl::vector<FGPUMeshObject>
OptimizedMeshes;
	cl::vector<FObject> Primitives;
	cl::vector<FLightSource> Lights;

	cl::vector<FFace> AllSceneFaces;
	cl::vector<Vertex> AllSceneVertices;
	cl::vector<Normal> AllSceneNormals;
	cl::vector<UV> AllSceneTexCoords;

	cl_int ChosenObjectIdx = -1;
	cl_int MaxIdx = -1;

	cl_int MaxLightSourceIdx = -1;
	cl_int ChosenLightSourceIdx = -1;
};
```

UUISubsystem.h

```
#pragma once

#define CL_HPP_TARGET_OPENCL_VERSION 300

#include "ISubsystemInterface.h"
#include <CL/cl.hpp>
#include "SDL.h"
#include "Object.h"
#include "Layout.h"

class UUISubsystem : public ISubsystemInterface
{
public:

	bool Initialize() override;
	bool Update(float InDeltaTime) override;
	std::string GetSubsystemName() const
override;
	void Destroy() override;
```

```
	std::shared_ptr<UIInputText>
GetActiveInputTextField() const;

	cl::vector<std::shared_ptr<UIButton>>
Buttons;
	cl::vector<std::shared_ptr<UIPanel>> Panels;
	cl::vector<std::shared_ptr<UIInputText>>
InputTextFields;

	static UUISubsystem& Get()
	{
		static UUISubsystem UISubsystem;
		return UISubsystem;
	}

	std::shared_ptr<UIButton>
SpawnSphereButton;
	std::shared_ptr<UIButton>
SpawnObjectButton;

	std::shared_ptr<UIButton>
DestroyObjectButton;

	std::shared_ptr<UIButton>
ChosenObjectLocationText;
	std::shared_ptr<UIInputText>
ChosenObjectLocationX;
	std::shared_ptr<UIInputText>
ChosenObjectLocationY;
	std::shared_ptr<UIInputText>
ChosenObjectLocationZ;

	std::shared_ptr<UIButton>
ChosenObjectColorText;
	std::shared_ptr<UIInputText>
ChosenObjectColorX;
	std::shared_ptr<UIInputText>
ChosenObjectColorY;
	std::shared_ptr<UIInputText>
ChosenObjectColorZ;

	std::shared_ptr<UIButton>
ChosenObjectReflectionText;
	std::shared_ptr<UIInputText>
ChosenObjectReflection;

	std::shared_ptr<UIButton>
ReflectionIterationsText;
	std::shared_ptr<UIInputText>
ReflectionIterations;

	// Light source buttons

	std::shared_ptr<UIButton>
ChosenLightSourceLocationText;
	std::shared_ptr<UIInputText>
ChosenLightSourceLocationX;
	std::shared_ptr<UIInputText>
ChosenLightSourceLocationY;
	std::shared_ptr<UIInputText>
ChosenLightSourceLocationZ;
	std::shared_ptr<UIButton>
SpawnLightSource;
	std::shared_ptr<UIButton>
ChooseNextLightSource;
	std::shared_ptr<UIButton>
ChoosePrevLightSource;

	std::shared_ptr<UIButton>
ChosenLightSourceColorText;
	std::shared_ptr<UIInputText>
ChosenLightSourceColorX;
	std::shared_ptr<UIInputText>
ChosenLightSourceColorY;
	std::shared_ptr<UIInputText>
ChosenLightSourceColorZ;

	std::shared_ptr<UIButton>
ChosenLightSourceIntensityText;
	std::shared_ptr<UIInputText>
ChosenLightSourceIntensity;

	std::shared_ptr<UIButton>
DestroyLightSourceButton;
};
```

FMath.cpp

```
#include "FMath.h"

#pragma optimize("", off)

cl_float FMath::GetRadians(cl_float Angle)
{
	constexpr cl_float HalfCircle = 180.f;
	return (cl_float)M_PI * Angle / HalfCircle;
}

cl_float FMath::GetDegrees(cl_float Radians)
{
	constexpr cl_float HalfCircle = 180.f;
	return Radians * (HalfCircle /
(cl_float)M_PI);
}

cl_float4 FMath::GetRadiansVector(cl_float4 Angle)
{
	return { GetRadians(Angle.x),
GetRadians(Angle.y), GetRadians(Angle.z) };
}

cl_float4 FMath::GetDegreesVector(cl_float4 Angle)
{
	return { GetDegrees(Angle.x),
GetDegrees(Angle.y), GetDegrees(Angle.z) };
}

bool FMath::SquareEquation(cl_float3 Parameters,
cl_float& OutResult)
{
	cl_float PossibeResults[2];
```

```
	cl_float Discriminant;

	Discriminant = (Parameters.y * Parameters.y)
- (4 * Parameters.x * Parameters.z);
	if (Discriminant < 0.f)
		return (false);
	else if (Discriminant > 0)
		Discriminant = sqrt(Discriminant);
	PossibeResults[0] = (-Parameters.y -
Discriminant) / Parameters.x / 2;
	PossibeResults[1] = (-Parameters.y +
Discriminant) / Parameters.x / 2;
	if (PossibeResults[0] < 0 &&
PossibeResults[1] < 0)
		return (false);

	PossibeResults[0] = std::max<cl_float>(0.f,
PossibeResults[0]);
	PossibeResults[1] = std::max<cl_float>(0.f,
PossibeResults[1]);

	OutResult =
std::min<cl_float>(PossibeResults[0],
PossibeResults[1]);
	return (true);
}

cl_float4 FMath::RotateVectorByDegrees(cl_float4
InVector, cl_float4 InEulerDegrees)
{
	return
FRotator::FromFloat4Degrees(InEulerDegrees).ToQuat
ernion().RotateVector(InVector);
}

cl_float4 FMath::RotateVectorByRadians(cl_float4
InVector, cl_float4 InEulerRadians)
{
	return
FRotator::FromFloat4Radians(InEulerRadians).ToQuat
ernion().RotateVector(InVector);
}

cl_float4 FRotator::ToFloat4() const
{
	return { FMath::GetRadians(Roll),
FMath::GetRadians(Pitch), FMath::GetRadians(Yaw)
};
}

FQuaternion FRotator::ToQuaternion() const
{
	cl_float cy =
(cl_float)cos(FMath::GetRadians(Yaw) * 0.5);
	cl_float sy =
(cl_float)sin(FMath::GetRadians(Yaw) * 0.5);
	cl_float cp =
(cl_float)cos(FMath::GetRadians(Pitch) * 0.5);
	cl_float sp =
(cl_float)sin(FMath::GetRadians(Pitch) * 0.5);
	cl_float cr =
(cl_float)cos(FMath::GetRadians(Roll) * 0.5);
	cl_float sr =
(cl_float)sin(FMath::GetRadians(Roll) * 0.5);

	FQuaternion Quat;
	Quat.InnerQuat.w = cr * cp * cy + sr * sp *
sy;
	Quat.InnerQuat.x = sr * cp * cy - cr * sp * sy;
	Quat.InnerQuat.y = cr * sp * cy + sr * cp * sy;
	Quat.InnerQuat.z = cr * cp * sy - sr * sp * cy;

	return Quat;
}

FRotator FRotator::FromFloat4Degrees(cl_float4
InData)
{
	FRotator Rotator;

	Rotator.Roll = InData.x;
	Rotator.Pitch = InData.y;
	Rotator.Yaw = InData.z;

	return Rotator;
}

FRotator FRotator::FromFloat4Radians(cl_float4
InData)
{
	FRotator Rotator;

	Rotator.Roll = FMath::GetDegrees(InData.x);
	Rotator.Pitch = FMath::GetDegrees(InData.y);
	Rotator.Yaw = FMath::GetDegrees(InData.z);

	return Rotator;
}

FRotator FQuaternion::ToRotator() const
{
	FRotator Rotator;

	// Roll (x-axis rotation)
	cl_float sinr_cosp = 2 * (InnerQuat.w *
InnerQuat.x + InnerQuat.y * InnerQuat.z);
	cl_float cosr_cosp = 1 - 2 * (InnerQuat.x *
InnerQuat.x + InnerQuat.y * InnerQuat.y);
	Rotator.Roll = atan2(sinr_cosp, cosr_cosp);

	// Pitch (y-axis rotation)
	cl_float sinp = 2 * (InnerQuat.w * InnerQuat.y
- InnerQuat.z * InnerQuat.x);
	if (abs(sinp) >= 1)
		Rotator.Pitch =
(cl_float)copysign(M_PI / 2, sinp);
	else
		Rotator.Pitch = asin(sinp);

	// Yaw (z-axis rotation)
	cl_float siny_cosp = 2 * (InnerQuat.w *
InnerQuat.z + InnerQuat.x * InnerQuat.y);
	cl_float cosy_cosp = 1 - 2 * (InnerQuat.y *
InnerQuat.y + InnerQuat.z * InnerQuat.z);
	Rotator.Yaw = atan2(siny_cosp, cosy_cosp);
```

```
	// Convertion to degrees
	Rotator.Roll = Rotator.Roll * (180.0 / M_PI);
	Rotator.Pitch = Rotator.Pitch * (180.0 /
M_PI);
	Rotator.Yaw = Rotator.Yaw * (180.0 / M_PI);

	return Rotator;
}

cl_float4 FQuaternion::RotateVector(cl_float4
InVector) const
{
	FQuaternion VectorQuat(InVector.x,
InVector.y, InVector.z, 0.f);
	FQuaternion QuatConjugate = Conjugate();
	FQuaternion Result = *this * VectorQuat *
QuatConjugate;

	return cl_float4{Result.InnerQuat.x,
Result.InnerQuat.y, Result.InnerQuat.z};
}
```

GPUComputingSubsystem.cpp

```
#include "GPUComputingSubsystem.h"
#include <iostream>
#include <istream>
#include <fstream>
#include "Object.h"
#include "USceneSubsystem.h"
#include "Pixel.h"
#include "SDLWindowSubsystem.h"

#pragma optimize("", off)

IGPUProgram::IGPUProgram(cl::Context& InContext,
cl::CommandQueue& InCommandQueue, cl::Device&
InActiveDevice) : Context(InContext),
CommandQueue(InCommandQueue),
ActiveDevice(InActiveDevice) {}

MainRaytracingProgram::MainRaytracingProgram(cl::
Context& InContext, cl::CommandQueue&
InCommandQueue, cl::Device& InActiveDevice) :
IGPUProgram(InContext, InCommandQueue,
InActiveDevice) {}

void MainRaytracingProgram::Initialize()
{
   std::ofstream
OutputStream("MainRaytracingProgram_outputlog.txt
");

   cl_int Error;
	std::ifstream
kernel_file("Resources/OpenCLPrograms/render.cl");
	cl::string
src(std::istreambuf_iterator<char>(kernel_file),
(std::istreambuf_iterator<char>()));
	cl::Program::Sources sources{ src };
   Program = cl::Program(Context, sources, &Error);
	if (Error != CL_BUILD_SUCCESS)
	{
		OutputStream << "Program wasn't
created" << std::endl;
		OutputStream.flush();
		OutputStream.close();
		exit(1);
	}

	Error = Program.build();
	if (Error != CL_BUILD_SUCCESS) {
		OutputStream << "Error!\nBuild
Status: " <<
Program.getBuildInfo<CL_PROGRAM_BUILD_STA
TUS>(ActiveDevice)
			<< "\nBuild Log:\t " <<
Program.getBuildInfo<CL_PROGRAM_BUILD_LOG
>(ActiveDevice) << std::endl;
		OutputStream.flush();
		OutputStream.close();
		exit(1);
	}

	Kernel = cl::Kernel(Program, "render",
&Error);
	if (Error != CL_BUILD_SUCCESS)
	{
		OutputStream << "Kernel wasn't
created" << std::endl;
		OutputStream << "Error!\nBuild
Status: " <<
Program.getBuildInfo<CL_PROGRAM_BUILD_STA
TUS>(ActiveDevice)
			<< "\nBuild Log:\t " <<
Program.getBuildInfo<CL_PROGRAM_BUILD_LOG
>(ActiveDevice) << std::endl;
		OutputStream.flush();
		OutputStream.close();
		exit(1);
	}

	const cl::size_type NumElements =
USDLWindowSubsystem::Get().GetRenderPixelsSize(
);
	const cl::size_type NumBytes = NumElements
* sizeof(Pixel);
	OutputBuffer = cl::Buffer(Context,
CL_MEM_WRITE_ONLY |
CL_MEM_HOST_READ_ONLY, NumBytes);

	Kernel.setArg(0, OutputBuffer);
}

bool MainRaytracingProgram::RunProgram()
{
	std::ofstream
OutputStream("MainRaytracingProgram_outputlog.txt
");
```

```
	const cl::size_type NumElements =
USDLWindowSubsystem::Get().GetRenderPixelsSize(
);
	const cl::size_type NumBytes = NumElements
* sizeof(Pixel);

	PrepareProgram();

	Pixel* Pixels =
USDLWindowSubsystem::Get().UnlockCanvas();
	cl_int err =
CommandQueue.enqueueNDRangeKernel(Kernel,
cl::NullRange, cl::NDRange(NumElements));
	if (err != CL_SUCCESS)
	{
		OutputStream << "Error!
CommandQueue returned " << err << std::endl;
	}
	err =
CommandQueue.enqueueReadBuffer(OutputBuffer,
CL_TRUE, 0, NumBytes, (void*)Pixels);
	if (err != CL_SUCCESS)
	{
		OutputStream << "Error!
enqueueReadBuffer returned " << err << std::endl;
	}

	USDLWindowSubsystem::Get().LockCanvas(
);

	return true;
}

void MainRaytracingProgram::PrepareProgram()
{
	cl_mem_flags ObjectsBufferFlags =
CL_MEM_READ_ONLY |
CL_MEM_HOST_NO_ACCESS |
CL_MEM_USE_HOST_PTR;
	cl_int ErrorCode = 0;

	USceneSubsystem& Scene =
USceneSubsystem::Get();
	PrimitivesBuffer = cl::Buffer(Context,
ObjectsBufferFlags, sizeof(FObject) *
Scene.Primitives.size(), Scene.Primitives.data(),
&ErrorCode);
	if (ErrorCode != CL_SUCCESS)
	{
		//printf("Failed to create
PrimitivesBuffer\n");
	}
	LightsBuffer = cl::Buffer(Context,
ObjectsBufferFlags, sizeof(FLightSource) *
Scene.Lights.size(), Scene.Lights.data(), &ErrorCode);
	if (ErrorCode != CL_SUCCESS)
	{
		//printf("Failed to create
LightsBuffer\n");
	}
	SceneDataBuffer = cl::Buffer(Context,
ObjectsBufferFlags, sizeof(FSceneData),
Scene.SceneData.get(), &ErrorCode);
	if (ErrorCode != CL_SUCCESS)
	{
		//printf("Failed to create
SceneDataBuffer\n");
	}
	MeshesBuffer = cl::Buffer(Context,
ObjectsBufferFlags, sizeof(FGPUMeshObject) *
Scene.OptimizedMeshes.size(),
Scene.OptimizedMeshes.data(), &ErrorCode);
	if (ErrorCode != CL_SUCCESS)
	{
		//printf("Failed to create
MeshesBuffer\n");
	}
	VerticesBuffer = cl::Buffer(Context,
ObjectsBufferFlags, sizeof(Vertex) *
Scene.AllSceneVertices.size(),
Scene.AllSceneVertices.data(), &ErrorCode);
	if (ErrorCode != CL_SUCCESS)
	{
		//printf("Failed to create
VerticesBuffer\n");
	}
	FacesBuffer = cl::Buffer(Context,
ObjectsBufferFlags, sizeof(FFace) *
Scene.AllSceneFaces.size(),
Scene.AllSceneFaces.data(), &ErrorCode);
	if (ErrorCode != CL_SUCCESS)
	{
		//printf("Failed to create
FacesBuffer\n");
	}
	NormalsBuffer = cl::Buffer(Context,
ObjectsBufferFlags, sizeof(Normal) *
Scene.AllSceneNormals.size(),
Scene.AllSceneNormals.data(), &ErrorCode);
	if (ErrorCode != CL_SUCCESS)
	{
		//printf("Failed to create
NormalsBuffer\n");
	}

	Kernel.setArg(1, PrimitivesBuffer);
	Kernel.setArg(2, LightsBuffer);
	Kernel.setArg(3, SceneDataBuffer);
	Kernel.setArg(4, MeshesBuffer);
	Kernel.setArg(5, VerticesBuffer);
	Kernel.setArg(6, FacesBuffer);
	Kernel.setArg(7, NormalsBuffer);
}

bool UGPUComputingSubsystem::Initialize()
{
  cl::Platform::get(&Platforms);
  cl_int Error;

  std::ofstream OutputStream("outputlog.txt");

  if (Platforms.empty())
  {
    OutputStream << "No platforms found!" <<
std::endl;
    return false;
```

```
    }

    ActivePlatform = Platforms.front();

ActivePlatform.getDevices(CL_DEVICE_TYPE_GPU
, &Devices);

    if (Devices.empty())
    {
        OutputStream << "No devices found!" <<
std::endl;
        return false;
    }

    ActiveDevice = Devices.front();
    PrintDeviceInfo(ActiveDevice);

    Context = cl::Context(ActiveDevice, nullptr, nullptr,
nullptr, &Error);

    CommandQueue = cl::CommandQueue(Context,
ActiveDevice, cl::QueueProperties::None, &Error);

    MainProgram =
std::make_unique<MainRaytracingProgram>(Context,
CommandQueue, ActiveDevice);
    MainProgram->Initialize();

        return true;
}

bool UGPUComputingSubsystem::Update(float
InDeltaTime)
{
        MainProgram->RunProgram();
        return true;
}

std::string
UGPUComputingSubsystem::GetSubsystemName()
const
{
    return "UGPUComputingSubsystem";
}

void UGPUComputingSubsystem::Destroy()
{
}

void
UGPUComputingSubsystem::PrintDeviceInfo(const
cl::Device& InDevice) const
{
    auto name =
InDevice.getInfo<CL_DEVICE_NAME>();
    auto vendor =
InDevice.getInfo<CL_DEVICE_VENDOR>();
    auto version =
InDevice.getInfo<CL_DEVICE_VERSION>();
    auto workItems =
InDevice.getInfo<CL_DEVICE_MAX_WORK_ITEM
_SIZES>();
    auto workGroups =
InDevice.getInfo<CL_DEVICE_MAX_WORK_GRO
UP_SIZE>();
    auto computeUnits =
InDevice.getInfo<CL_DEVICE_MAX_COMPUTE_U
NITS>();
    auto globalMemory =
InDevice.getInfo<CL_DEVICE_GLOBAL_MEM_SIZ
E>();
    auto localMemory =
InDevice.getInfo<CL_DEVICE_LOCAL_MEM_SIZE
>();

    std::cout << "OpenCL Device Info:"
        << "\nName: " << name
        << "\nVendor: " << vendor
        << "\nVersion: " << version
        << "\nMax size of work-items: (" <<
workItems[0] << "," << workItems[1] << "," <<
workItems[2] << ")"
        << "\nMax size of work-groups: " << workGroups
        << "\nNumber of compute units: " <<
computeUnits
        << "\nGlobal memory size (bytes): " <<
globalMemory
        << "\nLocal memory size per compute unit
(bytes): " << localMemory / computeUnits
        << std::endl;
}
```

IntersectTest.cpp

```
#pragma once

#include "IntersectTest.h"

#pragma optimize("", off)

#define FOV 90.f

cl_float4 ScreenToWorldSpace(float screenX, float
screenY, float screenWidth, float screenHeight, float
depth)
{
        float scale = 1;
        float imageAspectRatio = screenWidth /
(float)screenHeight;

        cl_float4 Result = { (2 * (screenX + 0.5) /
(float)screenWidth - 1) * imageAspectRatio * scale,
                                                        (1
- 2 * (screenY + 0.5) / (float)screenHeight) * scale,

        depth };

        return Result;
}

Ray Ray::CreateRay(cl_float4 InOrigin, cl_float4
InDirection)
```

```
{
	Ray Output;
	Output.Direction =
FMath::NormalizeVector(InDirection);
	Output.DirectionDot =
FMath::DotProduct(Output.Direction,
Output.Direction);
	Output.Origin = {
				InOrigin.x +
(Output.Direction.x * EPSILON),
				InOrigin.y +
(Output.Direction.y * EPSILON),
				InOrigin.z +
(Output.Direction.z * EPSILON)
			};
	return Output;
}

bool RaySphereIntersect(const Ray& ray, const
FObject& sphere, float& OutDistance)
{
	const cl_float4	cam_sphere = { ray.Origin.x
- sphere.center.x, ray.Origin.y - sphere.center.y ,
ray.Origin.z - sphere.center.z };
	cl_float4 Direction = ray.Direction;
	cl_float4 k;

	k.x = ray.DirectionDot;
	k.y = 2 * FMath::DotProduct(Direction,
cam_sphere);
	k.z = FMath::DotProduct(cam_sphere,
cam_sphere) - sphere.pow_radius;
	if (!FMath::sq_equation(k, OutDistance))
		return (false);
	return (true);
}

bool RayTriangleIntersect(const Ray& ray, const
Vertex& v0, const Vertex& v1, const Vertex& v2,
float& t, float& u, float& v) {
	Vertex edge1 = { v1.x - v0.x, v1.y - v0.y, v1.z
- v0.z };
	Vertex edge2 = { v2.x - v0.x, v2.y - v0.y, v2.z
- v0.z };
	Vertex h = { ray.Direction.y * edge2.z -
ray.Direction.z * edge2.y,
				ray.Direction.z *
edge2.x - ray.Direction.x * edge2.z,
				ray.Direction.x *
edge2.y - ray.Direction.y * edge2.x };
	float a = edge1.x * h.x + edge1.y * h.y +
edge1.z * h.z;

	if (fabs(a) < EPSILON) return false;

	float f = 1.0 / a;
	Vertex s = { ray.Origin.x - v0.x, ray.Origin.y -
v0.y, ray.Origin.z - v0.z };
	u = f * (s.x * h.x + s.y * h.y + s.z * h.z);
	if (u < 0.0 || u > 1.0) return false;

	Vertex q = { s.y * edge1.z - s.z * edge1.y,
				s.z * edge1.x - s.x
* edge1.z,
				s.x * edge1.y - s.y
* edge1.x };
	v = f * (ray.Direction.x * q.x + ray.Direction.y
* q.y + ray.Direction.z * q.z);
	if (v < 0.0 || u + v > 1.0) return false;

	t = f * (edge2.x * q.x + edge2.y * q.y +
edge2.z * q.z);
	return fabs(t) > EPSILON;
}

Intersection RayMeshIntersect(const Ray& ray, const
FMeshObject& mesh, const FObject& Primitive)
{
	Intersection closestIntersection = { false,
std::numeric_limits<float>::infinity(), {0, 0, 0}, {0, 0,
0} };

	for (const FFace& face : mesh.Faces)
	{
		const Vertex v0 =
FMath::VectorAdd(mesh.Vertices[face.Vertices.s0],
Primitive.center);
		const Vertex v1 =
FMath::VectorAdd(mesh.Vertices[face.Vertices.s1],
Primitive.center);
		const Vertex v2 =
FMath::VectorAdd(mesh.Vertices[face.Vertices.s2],
Primitive.center);

		float t, u, v;
		if (RayTriangleIntersect(ray, v0, v1,
v2, t, u, v))
		{
			if (t < closestIntersection.t)
			{

	closestIntersection.hit = true;
				closestIntersection.t
= t;
			}
		}
	}

	return closestIntersection;
}

Intersection DoPixelIntersectTest(cl_int x, cl_int y)
{
	const FSceneData& SceneData =
*USceneSubsystem::Get().SceneData;
	cl_float4 RayDirection =
	{
		(-SceneData.WindowWidth / 2 + x) *
SceneData.RelationX,
		(SceneData.WindowHeight / 2 - y) *
SceneData.RelationY,
		SceneData.ScreenRelativeDistance,
		0.f
	};
```

```
	const cl_float4 PartResult =
FMath::RotateVectorByRadians(RayDirection,
SceneData.Camera.Rotation);

	Ray NewRay =
Ray::CreateRay(SceneData.Camera.Origin,
PartResult);

	USceneSubsystem& Scene =
USceneSubsystem::Get();
	Intersection OutputResult;
	OutputResult.t =
std::numeric_limits<float>::infinity();
	float Distance;
	for (FObject& Primitive :
USceneSubsystem::Get().Primitives)
	{
		if (Primitive.type ==
(cl_int)EObjectType::Sphere)
		{
			const bool bHit =
RaySphereIntersect(NewRay, Primitive, Distance);
			if (bHit && Distance <
OutputResult.t)
			{
				OutputResult.hit =
true;

	OutputResult.Primitive = Primitive;
				OutputResult.t =
Distance;
			}
		}
		else if (Primitive.type ==
(cl_int)EObjectType::Mesh)
		{
			if (Primitive.Idx >=
Scene.Meshes.size())
			{
				continue;
			}

	std::shared_ptr<FMeshObject> Mesh =
Scene.Meshes[Primitive.Idx];
			Intersection TempResult =
RayMeshIntersect(NewRay, *Mesh, Primitive);
			TempResult.Object = Mesh;
			if (TempResult.hit &&
TempResult.t < OutputResult.t)
			{
				OutputResult.hit =
true;

	OutputResult.Primitive = Primitive;
				OutputResult.t =
TempResult.t;
				OutputResult.point
= TempResult.point;

	OutputResult.Object = Mesh;
			}
		}
	}
	return OutputResult;
}
```

Layout.cpp

```
#include "Layout.h"
#include "UUISubsystem.h"
#include "SDL_ttf.h"
#include "SDLWindowSubsystem.h"
#include "USceneSubsystem.h"
#include "UInputSubsystem.h"
#include <sstream>
#include <iomanip>

#pragma optimize("", off)

cl_bool ILayoutElement::IsPointedByMouse(cl_int
MouseX, cl_int MouseY)
{
	return MouseX >= GetPosition().x &&
MouseX <= GetPosition().x + GetSize().x &&
		MouseY >= GetPosition().y &&
MouseY <= GetPosition().y + GetSize().y;
}

std::shared_ptr<UIButton>
UIButton::CreateUIButton(cl_int2 InPosition, cl_int2
InSize, std::string InText, FColor InTextColor, FColor
BorderColor, FColor FillColor, cl_int2 InPadding)
{
	USDLWindowSubsystem&
WindowSubsystem = USDLWindowSubsystem::Get();

	std::shared_ptr<UIButton> NewButton =
std::make_shared<UIButton>(InPosition, InSize,
InText, InTextColor, BorderColor, FillColor,
InPadding);
	NewButton->UpdateText(InText, InTextColor,
InPadding);

	UUISubsystem::Get().Buttons.push_back(Ne
wButton);

	return NewButton;
}

UIButton::UIButton(cl_int2 InPosition, cl_int2 InSize,
std::string InText, FColor InTextColor, FColor
InBorderColor, FColor InFillColor, cl_int2 InPadding)
:
	Texture(nullptr, &SDL_DestroyTexture),
	Rect{ (int)InPosition.x, (int)InPosition.y,
(int)InSize.x, (int)InSize.y },
	Text(InText),
	TextColor(InTextColor),
	BorderColor(InBorderColor),
	FillColor(InFillColor),
	Padding(InPadding)
{}
```

```
void UIButton::Render()
{
	USDLWindowSubsystem& WindowSubsystem = USDLWindowSubsystem::Get();

	// Draw button rectangle
	SDL_SetRenderDrawColor(WindowSubsystem.Renderer.get(), FillColor.x, FillColor.y, FillColor.z, FillColor.w);
	SDL_RenderFillRect(WindowSubsystem.Renderer.get(), &Rect);

	// Draw button border
	SDL_SetRenderDrawColor(WindowSubsystem.Renderer.get(), FillColor.x, FillColor.y, FillColor.z, FillColor.w);
	SDL_RenderDrawRect(WindowSubsystem.Renderer.get(), &Rect);

	// Draw button text
	SDL_RenderCopyEx(WindowSubsystem.Renderer.get(), Texture.get(), nullptr, &TextRect, 0, nullptr, SDL_FLIP_NONE);
}

void UIButton::UpdateText(std::string NewText, FColor InTextColor, cl_int2 InPadding /*= { 0.f, 0.f }*/)
{
	USDLWindowSubsystem& WindowSubsystem = USDLWindowSubsystem::Get();

	Text = NewText;
	TextColor = InTextColor;
	Padding = InPadding;

	const SDL_Color SDLTextColor{ (Uint8)InTextColor.x, (Uint8)InTextColor.y, (Uint8)InTextColor.z, (Uint8)InTextColor.w };

	SDL_UniqueSurface TextSurface(

	TTF_RenderText_Solid(WindowSubsystem.Font.get(), Text.c_str(), SDLTextColor),
		SDL_FreeSurface
	);

	if (TextSurface == nullptr)
	{
		printf("Unable to render text surface! SDL_ttf Error: %s\n", TTF_GetError());
		return;
	}

	Texture.reset(SDL_CreateTextureFromSurface(WindowSubsystem.Renderer.get(), TextSurface.get()));
	if (Texture == nullptr)
	{
		printf("Unable to create texture from rendered text! SDL Error: %s\n", SDL_GetError());
		return;
	}

	Rect.w = std::max<int>(Rect.w, TextSurface->w + (Padding.x * 2));
	Rect.h = std::max<int>(Rect.h, TextSurface->h + (Padding.y * 2));

	TextRect.w = std::min(TextSurface->w, Rect.w - Padding.x);
	TextRect.h = std::min(TextSurface->h, Rect.h - Padding.y);
	TextRect.x = Rect.x + (Rect.w - TextSurface->w) / 2;
	TextRect.y = Rect.y + Padding.y;
}

bool UIButton::IsHovered() const
{
	return bIsHovered;
}

void UIButton::OnButtonClicked(bool bClicked)
{
	bIsClicked = bClicked;

	OnButtonClickedDelegate.ExecuteIfBound(bClicked);
}

cl_float2 UIButton::GetSize() const
{
	return { (cl_float)Rect.w, (cl_float)Rect.h };
}

cl_float2 UIButton::GetPosition() const
{
	return { (cl_float)Rect.x, (cl_float)Rect.y };
}

const FColor& UIButton::GetBorderColor() const
{
	return BorderColor;
}

const FColor& UIButton::GetFillColor() const
{
	return FillColor;
}

std::shared_ptr<UIPanel> UIPanel::CreateUIPanel(cl_int2 InPosition, cl_int2 InSize, FColor BorderColor, FColor FillColor)
{
	USDLWindowSubsystem& WindowSubsystem = USDLWindowSubsystem::Get();

	std::shared_ptr<UIPanel> NewButton = std::make_shared<UIPanel>(InPosition, InSize, BorderColor, FillColor);

	UUISubsystem::Get().Panels.push_back(NewButton);
```

```
	return NewButton;
}

UIPanel::UIPanel(cl_int2 InPosition, cl_int2 InSize, FColor InBorderColor, FColor InFillColor) :
	Rect{ (int)InPosition.x, (int)InPosition.y, (int)InSize.x, (int)InSize.y },
	BorderColor(InBorderColor),
	FillColor(InFillColor)
{
}

void UIPanel::Render()
{
	USDLWindowSubsystem& WindowSubsystem = USDLWindowSubsystem::Get();

	// Draw rectangle
	SDL_SetRenderDrawColor(WindowSubsystem.Renderer.get(), FillColor.x, FillColor.y, FillColor.z, FillColor.w);
	SDL_RenderFillRect(WindowSubsystem.Renderer.get(), &Rect);

	// Draw border
	SDL_SetRenderDrawColor(WindowSubsystem.Renderer.get(), FillColor.x, FillColor.y, FillColor.z, FillColor.w);
	SDL_RenderDrawRect(WindowSubsystem.Renderer.get(), &Rect);
}

cl_float2 UIPanel::GetSize() const
{
	return { (cl_float)Rect.w, (cl_float)Rect.h };
}

cl_float2 UIPanel::GetPosition() const
{
	return { (cl_float)Rect.x, (cl_float)Rect.y };
}

const FColor& UIPanel::GetBorderColor() const
{
	throw std::logic_error("The method or operation is not implemented.");
}

const FColor& UIPanel::GetFillColor() const
{
	return FillColor;
}

std::shared_ptr<UIInputText> UIInputText::CreateUIInputText(cl_int2 InPosition, cl_int2 InSize, FColor InTextColor, FColor BorderColor, FColor FillColor, cl_int2 InPadding /*= { 5, 5 }*/)
{
	USDLWindowSubsystem& WindowSubsystem = USDLWindowSubsystem::Get();

	std::shared_ptr<UIInputText> NewButton = std::make_shared<UIInputText>(InPosition, InSize, InTextColor, BorderColor, FillColor, InPadding);

	UUISubsystem::Get().InputTextFields.push_back(NewButton);

	return NewButton;
}

void UIInputText::Render()
{
	USDLWindowSubsystem& WindowSubsystem = USDLWindowSubsystem::Get();

	// Draw button rectangle
	SDL_SetRenderDrawColor(WindowSubsystem.Renderer.get(), FillColor.x, FillColor.y, FillColor.z, FillColor.w);
	SDL_RenderFillRect(WindowSubsystem.Renderer.get(), &Rect);

	// Draw button border
	SDL_SetRenderDrawColor(WindowSubsystem.Renderer.get(), FillColor.x, FillColor.y, FillColor.z, FillColor.w);
	SDL_RenderDrawRect(WindowSubsystem.Renderer.get(), &Rect);

	// Draw button text
	SDL_RenderCopyEx(WindowSubsystem.Renderer.get(), Texture.get(), nullptr, &TextRect, 0, nullptr, SDL_FLIP_NONE);
}

void UIInputText::UpdateText(std::string InString, bool bRemoveSymbol)
{
	USDLWindowSubsystem& WindowSubsystem = USDLWindowSubsystem::Get();

	if (bRemoveSymbol)
	{
		Text.erase(Text.size() - 2);
	}
	else
	{
		Text = InString;
	}

	const SDL_Color SDLTextColor{ (Uint8)TextColor.x, (Uint8)TextColor.y, (Uint8)TextColor.z, (Uint8)TextColor.w };

	SDL_UniqueSurface TextSurface(

	TTF_RenderText_Solid(WindowSubsystem.Font.get(), Text.c_str(), SDLTextColor),
		SDL_FreeSurface
	);

	if (TextSurface == nullptr)
	{
```

```
            printf("Unable to render text surface!
SDL_ttf Error: %s\n", TTF_GetError());
            return;
      }

      Texture.reset(SDL_CreateTextureFromSurface
(WindowSubsystem.Renderer.get(),
TextSurface.get()));
      if (Texture == nullptr)
      {
            printf("Unable to create texture from
rendered text! SDL Error: %s\n", SDL_GetError());
            return;
      }

      Rect.w = std::max<int>(Rect.w, TextSurface-
>w + (Padding.x * 2));
      Rect.h = std::max<int>(Rect.h, TextSurface-
>h + (Padding.y * 2));

      TextRect.w = std::min(TextSurface->w,
Rect.w - Padding.x);
      TextRect.h = std::min(TextSurface->h, Rect.h
- Padding.y);
      TextRect.x = Rect.x + (Rect.w - TextSurface-
>w) / 2;
      TextRect.y = Rect.y + (Rect.h - TextSurface-
>h) / 2;
}

void UIInputText::ApplyInput()
{
      OnTextUpdatedDelegate.ExecuteIfBound(Tex
t);
}
```

MeshObjectParser.cpp

```
#include "MeshObjectParser.h"
#include <fstream>
#include <istream>
#include <iostream>
#include <sstream>
#include "USceneSubsystem.h"
#include <shobjidl.h>
#include <windows.h>

#define VERTEX_PREFIX "v"
#define NORMAL_PREFIX "vn"
#define FACE_PREFIX "f"
#define TEXTURE_PREFIX "vt"

constexpr COMDLG_FILTERSPEC
g_FileFormatFilter[] =
{
      { L"MeshObject file", L"*.obj" },
};

struct SourceFace
{
      cl_int VertexIndices[3];
      cl_int NormalIndices[3];
      cl_int TextureCoordIndices[3];
};

std::shared_ptr<FMeshObject>
UMeshObjectParserSubsystem::ParseMeshObjectFrom
File(std::wstring InFilePath)
{
      try
      {
            std::shared_ptr<FMeshObject>
ResultMeshObject =
std::make_shared<FMeshObject>();

            std::ifstream file(InFilePath);
            if (!file.is_open())
            {
                  throw
std::runtime_error("Could not open file");
            }

            std::string Line;
            while (getline(file, Line))
            {
                  std::istringstream Iss(Line);
                  std::string Prefix;

                  Iss >> Prefix;
                  if (Prefix ==
VERTEX_PREFIX)
                  {
                        Vertex NewVertex;
                        Iss >> NewVertex.x
>> NewVertex.y >> NewVertex.z;
                        ResultMeshObject-
>Vertices.push_back(NewVertex);
                  }
                  else if (Prefix ==
NORMAL_PREFIX)
                  {
                        Normal
NewNormal;
                        Iss >>
NewNormal.x >> NewNormal.y >> NewNormal.z;
                        ResultMeshObject-
>Normals.push_back(NewNormal);
                  }
                  else if (Prefix ==
TEXTURE_PREFIX)
                  {
                        UV texCoord;
                        Iss >> texCoord.x
>> texCoord.y;
                        ResultMeshObject-
>TextureCoords.push_back(texCoord);
                  }
                  else if (Prefix ==
FACE_PREFIX)
                  {
                        FFace NewFace;
                        std::string index;
```

```
cl_int VertexIdxs[3];
cl_int NormalsIdxs[3];
cl_int TexCoordsIdxs[3];

bool bFoundNormal = false;
bool bFoundTextureCoord = false;
for (int Iter = 0; Iter < 3; ++Iter)
{
    Iss >> index;
    size_t slash1 = index.find('/');
    size_t slash2 = index.rfind('/');

    VertexIdxs[Iter] = std::stoi(index.substr(0, slash1)) - 1;
    if (slash1 != -1)
    {
        std::string str = index.substr(slash1 + 1, slash2);

        int Iter = 0;

        while (Iter < str.size() && !(std::isdigit(str[Iter]) || str[Iter] == '-' || str[Iter] == '+' || str[Iter] == '.'))
        {
            Iter++;
        }

        TexCoordsIdxs[Iter] = std::stoi(str.substr(Iter)) - 1;

        bFoundTextureCoord = true;
    }
    if (slash2 != -1)
    {
        std::string str = index.substr(slash2 + 1);

        int Iter = 0;

        while (Iter < str.size() && !(std::isdigit(str[Iter]) || str[Iter] == '-' || str[Iter] == '+' || str[Iter] == '.'))
        {
            ++Iter;
        }

        NormalsIdxs[Iter] = std::stoi(str.substr(Iter)) - 1;

        bFoundNormal = true;
    }
}

NewFace.Vertices = { VertexIdxs[0], VertexIdxs[1], VertexIdxs[2] };
if (bFoundNormal)
    NewFace.Normals = { NormalsIdxs[0], NormalsIdxs[1], NormalsIdxs[2] };
if (bFoundTextureCoord)
    NewFace.TexCoords = { TexCoordsIdxs[0], TexCoordsIdxs[1], TexCoordsIdxs[2] };

ResultMeshObject->Faces.push_back(NewFace);
            }
        }

        USceneSubsystem::Get().AddMeshToScene(ResultMeshObject);
        return ResultMeshObject;
    }
    catch (const std::exception& e)
    {
        std::cerr << "Error: " << e.what() << std::endl;
        return nullptr;
    }
}

std::shared_ptr<FMeshObject> UMeshObjectParserSubsystem::ParseMeshObjectFromDialogWindow()
{
    std::wstring path = OpenFileWinAPI();
    return ParseMeshObjectFromFile(path);
}

std::wstring UMeshObjectParserSubsystem::OpenFileWinAPI()
{
    IFileOpenDialog* pFileOpen;
    HRESULT Result;
    std::wstring OutPath;

    // Create the FileOpenDialog MeshObject.
    Result = CoCreateInstance(CLSID_FileOpenDialog, NULL, CLSCTX_ALL,
        IID_IFileOpenDialog, reinterpret_cast<void**>(&pFileOpen));

    if (SUCCEEDED(Result))
    {
        // Set the options on the dialog.
```

```
		DWORD dwFlags;

		// Before setting, always get the
options first in order
		// not to override existing options.
		Result = pFileOpen-
>GetOptions(&dwFlags);
		Result &= pFileOpen-
>SetOptions(dwFlags | FOS_STRICTFILETYPES);
		Result &= pFileOpen-
>SetFileTypes(ARRAYSIZE(g_FileFormatFilter),
g_FileFormatFilter);

		if (SUCCEEDED(Result))
		{
			// Show the Open dialog
box.
			Result = pFileOpen-
>Show(NULL);

			// Get the file name from the
dialog box.
			if (SUCCEEDED(Result))
			{
				IShellItem* pItem;
				Result =
pFileOpen->GetResult(&pItem);
				if
(SUCCEEDED(Result))
				{
					wchar_t*
pszFilePath = nullptr;
					Result =
pItem->GetDisplayName(SIGDN_FILESYSPATH,
&pszFilePath);
					OutPath =
pszFilePath;

					// Display
the file name to the user.
					if
(SUCCEEDED(Result))
					{

	//MessageBoxW(NULL, pszFilePath, L"File
Path", MB_OK);

	CoTaskMemFree(pszFilePath);
					}
					pItem-
>Release();
				}
			}
		}
		pFileOpen->Release();
	}

	return OutPath;
}

bool UMeshObjectParserSubsystem::Initialize()
{
	HRESULT Result = CoInitializeEx(NULL,
COINIT_APARTMENTTHREADED |
		COINIT_DISABLE_OLE1DDE);

	return SUCCEEDED(Result);
}

std::string
UMeshObjectParserSubsystem::GetSubsystemName()
const
{
	return "MeshObjectParserSubsystem";
}

void UMeshObjectParserSubsystem::Destroy()
{
	CoUninitialize();
}
```

Object.cpp

```
#include "Object.h"

FSphereObject::FSphereObject(cl_float3 InOrigin,
cl_float InRadius, FColor InColor, cl_float
InReflection)
{
	center = InOrigin;
	pow_radius = InReflection * InRadius;
	color = InColor;
	reflection = InReflection;
}

FObject FSphereObject::ConstructSphere(cl_float3
InOrigin, cl_float InRadius, FColor InColor, cl_float
InReflection)
{
	FObject NewObject;
	NewObject.center = InOrigin;
	NewObject.pow_radius = InRadius *
InRadius;
	NewObject.color = InColor;
	NewObject.reflection = InReflection;
	NewObject.specularity = 100;
	NewObject.type =
static_cast<cl_uint>(EObjectType::Sphere);

	return NewObject;
}
```

SDLWindowSubsystem.cpp

```
#include "SDLWindowSubsystem.h"
#include <iostream>
#include <string>
#include <sstream>
#include "USceneSubsystem.h"
```

```
#include "FMath.h"
#include "UUISubsystem.h"

#define _CRT_SECURE_NO_WARNINGS

#pragma optimize("", off)

#define SCREEN_WIDTH 1920
#define SCREEN_HEIGHT 1080

#define RENDER_WIDTH 1920
#define RENDER_HEIGHT 1080

USDLWindowSubsystem::USDLWindowSubsystem() :
   Window(nullptr, &SDL_DestroyWindow),
   Renderer(nullptr, &SDL_DestroyRenderer),
   MainRenderTexture(nullptr,
&SDL_DestroyTexture),
   Font(nullptr, &TTF_CloseFont)
{
}

bool USDLWindowSubsystem::Initialize()
{
   if (SDL_Init(SDL_INIT_VIDEO) < 0)
   {
      std::cout << "SDL could not be initialized!" <<
std::endl
         << "SDL_Error: " << SDL_GetError() <<
std::endl;
      return 0;
   }

        if (TTF_Init() != 0)
   {
              return 0;
        }

   SDL_Window* WindowRawPtr =
SDL_CreateWindow("Raytracing render",
      SDL_WINDOWPOS_CENTERED,
      SDL_WINDOWPOS_CENTERED,
      SCREEN_WIDTH, SCREEN_HEIGHT,
      SDL_WINDOW_BORDERLESS);
   if (!WindowRawPtr)
   {
      std::cout << "Window could not be created!" <<
std::endl
         << "SDL_Error: " << SDL_GetError() <<
std::endl;
      return false;
   }
   Window.reset(WindowRawPtr);

   SDL_Renderer* RendererRawPtr =
SDL_CreateRenderer(Window.get(), -1,
SDL_RENDERER_ACCELERATED);
   if (!RendererRawPtr)
   {
      std::cout << "Renderer could not be created!" <<
std::endl
         << "SDL_Error: " << SDL_GetError() <<
std::endl;
      return false;
   }
   Renderer.reset(RendererRawPtr);

        SDL_Texture* TextureRawPtr =
SDL_CreateTexture(Renderer.get(),
SDL_PIXELFORMAT_RGBA8888,
SDL_TEXTUREACCESS_STREAMING,
RENDER_WIDTH, RENDER_HEIGHT);
        if (!RendererRawPtr)
        {
              std::cout << "Texture could not be
created!" << std::endl
                    << "SDL_Error: " <<
SDL_GetError() << std::endl;
              return false;
        }
   MainRenderTexture.reset(TextureRawPtr);

        TTF_Font* FontRawPtr =
TTF_OpenFont("Resources/OpenSans-Regular.ttf",
16);
        if (!FontRawPtr)
        {
              std::cout << "Font could not be
created!" << std::endl
                    << "SDL_Error: " <<
SDL_GetError() << std::endl;
              return false;
        }
        Font.reset(FontRawPtr);

   return true;
}

bool USDLWindowSubsystem::Update(float
InDeltaTime)
{
   SDL_RenderClear(Renderer.get());
   SDL_Rect dstrect = {300, 0, GetRenderWidth(),
GetRenderHeight()};
   SDL_RenderCopy(Renderer.get(),
MainRenderTexture.get(), nullptr, &dstrect);

        for (std::shared_ptr<UIPanel>& Panel :
UUISubsystem::Get().Panels)
        {
     Panel->Render();
        }

        for (std::shared_ptr<UIButton>& Button :
UUISubsystem::Get().Buttons)
        {
              Button->Render();
        }

        for (std::shared_ptr<UIInputText>& Button :
UUISubsystem::Get().InputTextFields)
        {
              Button->Render();
        }

   // Update log start
```

```
   FCamera Cam =
USceneSubsystem::Get().SceneData->Camera;
   cl_float FPS = 1.0f / InDeltaTime;
   std::stringstream stream;

   stream << "location: " << Cam.Origin.x << " " <<
Cam.Origin.y << " " << Cam.Origin.z << "; rotation: "
<<
      FMath::GetDegrees(Cam.Rotation.x) << " " <<
FMath::GetDegrees(Cam.Rotation.y) << " " <<
FMath::GetDegrees(Cam.Rotation.z) << ";" << " FPS:
" << FPS << " ; ";

   DrawFontText(stream.str(), { 255, 0, 0 }, {780, 0});
   // Update log end

   SDL_RenderPresent(Renderer.get());

   return true;
}

std::string
USDLWindowSubsystem::GetSubsystemName() const
{
   return "SDLWindowSubsystem";
}

void USDLWindowSubsystem::Destroy()
{
   Font.reset();
   MainRenderTexture.reset();

   Renderer.reset();
   Window.reset();

   TTF_Quit();
   SDL_Quit();
}

void
USDLWindowSubsystem::DrawFontText(std::string
InText, SDL_Color InColor, cl_int2 Position)
{
	SDL_Surface* textSurface =
TTF_RenderText_Solid(Font.get(), InText.c_str(),
InColor);
	if (textSurface == NULL)
	{
		printf("Unable to render text surface!
SDL_ttf Error: %s\n", TTF_GetError());
	}
	else
	{
		//Create texture from surface pixels
		SDL_Texture* mTexture =
SDL_CreateTextureFromSurface(Renderer.get(),
textSurface);
		if (mTexture == NULL)
		{
			printf("Unable to create
texture from rendered text! SDL Error: %s\n",
SDL_GetError());
		}
		SDL_Rect renderQuad = {
Position.x, Position.y, textSurface->w, textSurface->h
};

		SDL_RendererFlip Flip =
SDL_FLIP_NONE;
		SDL_RenderCopyEx(Renderer.get(),
mTexture, nullptr, &renderQuad, 0, nullptr, Flip);

		//Get rid of old surface
		SDL_FreeSurface(textSurface);
		SDL_DestroyTexture(mTexture);
	}
}

Pixel* USDLWindowSubsystem::UnlockCanvas()
const
{
	void* TexturePixels = nullptr;
	int Pitch = 0;

	SDL_LockTexture(MainRenderTexture.get(),
nullptr, &TexturePixels, &Pitch);
   return static_cast<Pixel*>(TexturePixels);
}

void USDLWindowSubsystem::LockCanvas() const
{
   SDL_UnlockTexture(MainRenderTexture.get());
}

cl_int USDLWindowSubsystem::GetRenderHeight()
const
{
   return RENDER_HEIGHT;
}

cl_int USDLWindowSubsystem::GetRenderWidth()
const
{
   return RENDER_WIDTH;
}

cl_int
USDLWindowSubsystem::GetRenderPixelsSize()
const
{
   return GetRenderHeight() * GetRenderWidth();
}
```

UInputSubsystem.cpp

```
#include "UInputSubsystem.h"
#include "SDLWindowSubsystem.h"
#include "USceneSubsystem.h"
#include "FMath.h"
#include "MeshObjectParser.h"
#include <algorithm>
#include "IntersectTest.h"
```

```
#include "UUISubsystem.h"

#pragma optimize("", off)

bool UInputSubsystem::Initialize()
{
	return true;
}

bool UInputSubsystem::Update(float InDeltaTime)
{
	SDL_Event Event;
	while (SDL_PollEvent(&Event))
	{
		switch (Event.type)
		{
		case SDL_TEXTINPUT:
		{
			CurrentInputText +=
Event.text.text;

	std::shared_ptr<UIInputText> InputTextField
= UUISubsystem::Get().GetActiveInputTextField();
			if (InputTextField)
			{
				InputTextField-
>UpdateText(CurrentInputText);
			}
			continue;
		}
		case SDL_KEYDOWN:
		{
			if (Event.key.keysym.sym
== SDLK_BACKSPACE &&
CurrentInputText.length() > 0)
			{

	std::shared_ptr<UIInputText> InputTextField
= UUISubsystem::Get().GetActiveInputTextField();
				if (InputTextField)
				{

	CurrentInputText.pop_back();

	InputTextField-
>UpdateText(CurrentInputText);
					continue;
				}
			}
			else if
(Event.key.keysym.sym == SDLK_RETURN)
			{

	std::shared_ptr<UIInputText> InputTextField
= UUISubsystem::Get().GetActiveInputTextField();
				if (InputTextField)
				{

	SDL_StopTextInput();

	InputTextField-
>UpdateText(CurrentInputText);

	InputTextField->ApplyInput();

	InputTextField->bIsActive = false;
					continue;
				}
			}

			ProcessInputDown(Event,
InDeltaTime);
			break;
		}
		case SDL_KEYUP:
			ProcessInputUp(Event,
InDeltaTime);
			break;
		case SDL_MOUSEMOTION:
		case
SDL_MOUSEBUTTONDOWN:
		case SDL_MOUSEBUTTONUP:
			ProcessMouseMotion(Event,
InDeltaTime);
			break;
		case SDL_QUIT:
			return false;
		default:
			break;
		}
	}
	std::shared_ptr<UIInputText> InputTextField
= UUISubsystem::Get().GetActiveInputTextField();
	if (InputTextField)
	{
		return true;
	}
	return InnerUpdate(InDeltaTime);
}

std::string UInputSubsystem::GetSubsystemName()
const
{
	return "InputSubsystem";
}

void UInputSubsystem::Destroy()
{
}

bool UInputSubsystem::InnerUpdate(float
InDeltaTime)
{
	FCamera& Camera =
USceneSubsystem::Get().SceneData->Camera;
	constexpr cl_float
AdditionalMovementMultiplier = 2.f;
	constexpr cl_float
AdditionalRotationMultiplier = 50.f;

	if (bHasUpRotationInput &&
!bHasDownRotationInput)
	{
```

```
			Camera.Rotation.x -=
FMath::GetRadians(1.f) * RotationSpeed *
InDeltaTime * AdditionalRotationMultiplier;
		}
		else if (bHasDownRotationInput &&
!bHasUpRotationInput)
		{
			Camera.Rotation.x +=
FMath::GetRadians(1.f) * RotationSpeed *
InDeltaTime * AdditionalRotationMultiplier;
		}

		if (bHasLeftRotationInput &&
!bHasRightRotationInput)
		{
			Camera.Rotation.y -=
FMath::GetRadians(1.f) * RotationSpeed *
InDeltaTime * AdditionalRotationMultiplier;
		}
		else if (bHasRightRotationInput &&
!bHasLeftRotationInput)
		{
			Camera.Rotation.y +=
FMath::GetRadians(1.f) * RotationSpeed *
InDeltaTime * AdditionalRotationMultiplier;
		}

		cl_float4 Result;
		bool bApplyChanges = false;

		if (bHasForwardMovementInput &&
!bHasBackwardMovementInput)
		{
			const cl_float Movement =
MovementSpeed * InDeltaTime *
AdditionalMovementMultiplier;
			Result =
FMath::RotateVectorByDegrees({ 0, 0, Movement },
FMath::GetDegreesVector(Camera.Rotation));
			bApplyChanges = true;
		}
		else if (bHasBackwardMovementInput &&
!bHasForwardMovementInput)
		{
			const cl_float Movement =
MovementSpeed * InDeltaTime *
AdditionalMovementMultiplier;
			Result =
FMath::RotateVectorByDegrees({ 0, 0, -Movement },
FMath::GetDegreesVector(Camera.Rotation));
			bApplyChanges = true;
		}

		if (bHasLeftMovementInput &&
!bHasRightMovementInput)
		{
			const cl_float Movement =
MovementSpeed * InDeltaTime *
AdditionalMovementMultiplier;
			Result =
FMath::RotateVectorByDegrees({ -Movement, 0.f, 0.f
}, FMath::GetDegreesVector(Camera.Rotation));
			bApplyChanges = true;
		}
		else if (bHasRightMovementInput &&
!bHasLeftMovementInput)
		{
			const cl_float Movement =
MovementSpeed * InDeltaTime *
AdditionalMovementMultiplier;
			Result =
FMath::RotateVectorByDegrees({ Movement, 0.f, 0.f
}, FMath::GetDegreesVector(Camera.Rotation));
			bApplyChanges = true;
		}

		if (bHasUpMovementInput &&
!bHasDownMovementInput)
		{
			const cl_float Movement =
MovementSpeed * InDeltaTime *
AdditionalMovementMultiplier;
			Result =
FMath::RotateVectorByDegrees({ 0.f, Movement, 0.f
}, FMath::GetDegreesVector(Camera.Rotation));
			bApplyChanges = true;
		}
		else if (bHasDownMovementInput &&
!bHasUpMovementInput)
		{
			const cl_float Movement =
MovementSpeed * InDeltaTime *
AdditionalMovementMultiplier;
			Result =
FMath::RotateVectorByDegrees({ 0.f, -Movement, 0.f
}, FMath::GetDegreesVector(Camera.Rotation));
			bApplyChanges = true;
		}

		if (bApplyChanges)
		{
			//if
(USceneSubsystem::Get().ChosenObjectIdx == -1)
			{
				Camera.Origin.x +=
Result.x;
				Camera.Origin.y +=
Result.y;
				Camera.Origin.z +=
Result.z;
			}
			//else
			{
			//
		USceneSubsystem::Get().MoveChosenObject(
Result);
			}
		}

		return true;
}

void
UInputSubsystem::ProcessInputDown(SDL_Event
InEvent, cl_float DeltaTime)
{
```

```
	SDL_Keycode Keycode =
InEvent.key.keysym.sym;

	switch (Keycode)
	{
	case SDLK_LEFT:
		bHasLeftRotationInput = true;
		break;
	case SDLK_RIGHT:
		bHasRightRotationInput = true;
		break;
	case SDLK_UP:
	{
		bHasUpRotationInput = true;
		break;
	}
	case SDLK_DOWN:
	{
		bHasDownRotationInput = true;
		break;
	}

	case SDLK_w:
	{
		bHasForwardMovementInput = true;
		break;
	}
	case SDLK_s:
	{
		bHasBackwardMovementInput =
true;
		break;
	}
	case SDLK_a:
	{
		bHasLeftMovementInput = true;
		break;
	}
	case SDLK_d:
	{
		bHasRightMovementInput = true;
		break;
	}
	case SDLK_SPACE:
	{
		bHasUpMovementInput = true;
		break;
	}
	case SDLK_c:
	{
		bHasDownMovementInput = true;
		break;
	}

	case SDLK_ESCAPE:
	{

	USceneSubsystem::Get().ChosenObjectIdx = -
1;
		break;
	}

	default:
		break;
	}
}

void UInputSubsystem::ProcessInputUp(SDL_Event
InEvent, cl_float DeltaTime)
{
	SDL_Keycode Keycode =
InEvent.key.keysym.sym;

	switch (Keycode)
	{
	case SDLK_LEFT:
		bHasLeftRotationInput = false;
		break;
	case SDLK_RIGHT:
		bHasRightRotationInput = false;
		break;
	case SDLK_UP:
	{
		bHasUpRotationInput = false;
		break;
	}
	case SDLK_DOWN:
	{
		bHasDownRotationInput = false;
		break;
	}

	case SDLK_w:
	{
		bHasForwardMovementInput = false;
		break;
	}
	case SDLK_s:
	{
		bHasBackwardMovementInput =
false;
		break;
	}
	case SDLK_a:
	{
		bHasLeftMovementInput = false;
		break;
	}
	case SDLK_d:
	{
		bHasRightMovementInput = false;
		break;
	}
	case SDLK_SPACE:
	{
		bHasUpMovementInput = false;
		break;
	}
	case SDLK_c:
	{
		bHasDownMovementInput = false;
		break;
	}

	default:
		break;
```

```
		}
}

void
UInputSubsystem::ProcessMouseMotion(SDL_Event
InEvent, cl_float DeltaTime)
{
	if (InEvent.type ==  SDL_MOUSEMOTION)
	{
		int xMouse, yMouse;

	SDL_GetGlobalMouseState(&xMouse,
&yMouse);
		MousePosition = { xMouse , yMouse
};
		return;
	}

	if (InEvent.type ==
SDL_MOUSEBUTTONDOWN)
	{
		if (InEvent.button.button ==
SDL_BUTTON_LEFT)
			bLeftMouseButtonDown =
true;
		else if (InEvent.button.button ==
SDL_BUTTON_RIGHT)
			bRightMouseButtonDown =
true;
		Intersection Result =
DoPixelIntersectTest(MousePosition.x,
MousePosition.y);
		if (Result.hit && Result.t <
std::numeric_limits<float>::infinity())
		{

	USceneSubsystem::Get().ChosenObjectIdx =
Result.Primitive.Idx;
		}
	}
	else if (InEvent.type ==
SDL_MOUSEBUTTONUP)
	{
		if (InEvent.button.button ==
SDL_BUTTON_LEFT)
			bLeftMouseButtonDown =
false;
		else if (InEvent.button.button ==
SDL_BUTTON_RIGHT)
			bRightMouseButtonDown =
false;
	}
}

void UInputSubsystem::ProcessUIInput(SDL_Event
InEvent, cl_float DeltaTime)
{
}
```

USceneSubsystem.cpp

```
#include "USceneSubsystem.h"
#include "SDLWindowSubsystem.h"
#include "FMath.h"
#include "GPUComputingSubsystem.h"
#include "MeshObjectParser.h"

#pragma optimize("", off)

USceneSubsystem::USceneSubsystem()
{
	SceneData =
std::make_unique<FSceneData>();

	SceneData->Camera.Origin = { 0.f, 0.f, 0.f, 0.f
};
	SceneData->Camera.Rotation = { 0.f, 0.f, 0.f,
0.f };

	SceneData->RecursionLevel = 4.f;

	SceneData->AmbientLightIntensity = 0.1f;
	SceneData->AmbientLightColor = { 0xff,
0xff, 0xff, 0 };
}

bool USceneSubsystem::Initialize()
{
	SceneData->WindowHeight =
USDLWindowSubsystem::Get().GetRenderHeight();
	SceneData->WindowWidth =
USDLWindowSubsystem::Get().GetRenderWidth();
	SceneData->NumPixels =
USDLWindowSubsystem::Get().GetRenderPixelsSize(
);

	SceneData->ScreenRelativeHeight =
static_cast<cl_float>(SceneData->WindowHeight) /
SceneData->WindowWidth;
	SceneData->ScreenRelativeWidth = 1.f;
	SceneData->ScreenRelativeDistance = 1.f;

	SceneData->RelationX = SceneData-
>ScreenRelativeHeight / SceneData->WindowHeight;
	SceneData->RelationY = SceneData-
>ScreenRelativeWidth / SceneData->WindowWidth;

	return true;
}

bool USceneSubsystem::Update(float InDeltaTime)
{
	SceneData->NumPrimitives =
(cl_uint)Primitives.size();
	SceneData->NumLightSources =
(cl_uint)Lights.size();
	SceneData->NumFaces =
(cl_uint)AllSceneFaces.size();
	SceneData->NumMeshes =
(cl_uint)OptimizedMeshes.size();
	SceneData->NumVertices =
(cl_uint)AllSceneVertices.size();
```

```
	SceneData->NumNormals = (cl_uint)AllSceneNormals.size();
	SceneData->NumTexCoords = (cl_uint)AllSceneTexCoords.size();

	return true;
}

std::string USceneSubsystem::GetSubsystemName() const
{
	return "SceneSubsystem";
}

void USceneSubsystem::Destroy()
{

}

void USceneSubsystem::SpawnLightSource(cl_float4 Origin, FColor Color, cl_float Intensity)
{
	FLightSource NewLight = { Origin, Color, Intensity, ++MaxLightSourceIdx };
	Lights.push_back(NewLight);
	ChosenLightSourceIdx = NewLight.Idx;
}

FLightSource* USceneSubsystem::GetChosenLightSource(cl_int* IndexInArray)
{
	if (ChosenLightSourceIdx == -1)
	{
		return nullptr;
	}

	for (int i = 0; i < Lights.size(); ++i)
	{
		if (Lights[i].Idx == ChosenLightSourceIdx)
		{
			if (IndexInArray != nullptr)
			{
				*IndexInArray = i;
			}
			return &Lights[i];
		}
	}
	return nullptr;
}

void USceneSubsystem::ChooseNextLightSource()
{
	if (Lights.size() > 0)
	{
		cl_int IdxToFind = -1;
		FLightSource* Source = GetChosenLightSource(&IdxToFind);

		if (Source && IdxToFind != -1)
		{
			cl_int NewIdx = (IdxToFind + 1) % Lights.size();
			FLightSource& NewChosenSource = Lights[NewIdx];
			ChosenLightSourceIdx = NewChosenSource.Idx;
		}
		else
		{
			ChosenLightSourceIdx = Lights[0].Idx;
		}
	}
}

void USceneSubsystem::ChoosePrevLightSource()
{
	if (Lights.size() > 0)
	{
		cl_int IdxToFind = -1;
		FLightSource* Source = GetChosenLightSource(&IdxToFind);

		if (Source && IdxToFind != -1)
		{
			cl_int NewIdx = IdxToFind - 1;
			NewIdx = NewIdx < 0 ? Lights.size() - 1 : NewIdx;
			FLightSource& NewChosenSource = Lights[NewIdx];
			ChosenLightSourceIdx = NewChosenSource.Idx;
		}
		else
		{
			ChosenLightSourceIdx = Lights[Lights.size() - 1].Idx;
		}
	}
}

void USceneSubsystem::RemoveChosenLightSource()
{
	cl_int IdxToRemove = -1;
	FLightSource* SourceToRemove = GetChosenLightSource(&IdxToRemove);
	if (SourceToRemove && IdxToRemove != -1)
	{
		auto IteratorToRemove = Lights.begin() + IdxToRemove;
		Lights.erase(IteratorToRemove, IteratorToRemove + 1);
	}
	ChosenLightSourceIdx = -1;
}

void USceneSubsystem::SpawnSphere(cl_float3 InOrigin, cl_float InRadius, FColor InColor, cl_float InReflection)
{
```

```
	FObject NewObject =
FSphereObject::ConstructSphere(InOrigin, InRadius,
InColor, InReflection);
	NewObject.Idx = ++MaxIdx;
	Primitives.push_back(NewObject);
	ChosenObjectIdx = NewObject.Idx;
}

void USceneSubsystem::SpawnMesh(cl_float3
InOrigin /*= { 0.f, 0.f, 0.f }*/, FColor InColor /*= {
127, 127, 127 }*/, cl_float InReflection /*= 0.f*/)
{
	UMeshObjectParserSubsystem::Get().ParseM
eshObjectFromDialogWindow();
}

void USceneSubsystem::DestroyChosenObject()
{
	if (ChosenObjectIdx == -1)
	{
		return;
	}

	int IndexToRemove = 0;
	for (int i = 0; i < Primitives.size(); ++i)
	{
		if (Primitives[i].Idx ==
ChosenObjectIdx)
		{
			IndexToRemove = i;
			break;
		}
	}

	FObject ObjectToDelete =
Primitives[IndexToRemove];
	auto IteratorToDelete = Primitives.begin() +
IndexToRemove;
	Primitives.erase(IteratorToDelete,
IteratorToDelete + 1);

	if (ObjectToDelete.type ==
(cl_int)EObjectType::Mesh)
	{
		for (int i = 0; i < Meshes.size(); ++i)
		{
			if (Meshes[i]->UniqueID ==
ObjectToDelete.meshidx)
			{
				auto
MeshIteratorToErase = Meshes.begin() + i;

	Meshes.erase(MeshIteratorToErase,
MeshIteratorToErase + 1);
				break;
			}
		}

		for (int i = 0; i <
OptimizedMeshes.size(); ++i)
		{
			if
(OptimizedMeshes[i].UniqueID ==
ObjectToDelete.meshidx)
			{
				auto
MeshIteratorToErase = Meshes.begin() + i;

				auto
FacesIteratorToEraseBegin = AllSceneFaces.begin() +
OptimizedMeshes[i].StartingFaceIdx;
				auto
FacesIteratorToEraseEnd = AllSceneFaces.begin() +
OptimizedMeshes[i].StartingFaceIdx +
OptimizedMeshes[i].NumFaces;


	AllSceneFaces.erase(FacesIteratorToEraseBeg
in, FacesIteratorToEraseEnd);


	Meshes.erase(MeshIteratorToErase,
MeshIteratorToErase + 1);
				break;
			}
		}
	}

	ChosenObjectIdx = -1;
}

FObject* USceneSubsystem::GetChosenObject()
{
	if (ChosenObjectIdx == -1)
	{
		return nullptr;
	}

	for (FObject& Obj : Primitives)
	{
		if (Obj.Idx == ChosenObjectIdx)
		{
			return &Obj;
		}
	}
}

void USceneSubsystem::MoveChosenObject(cl_float4
Movement)
{
	if (ChosenObjectIdx == -1)
	{
		return;
	}

	for (FObject& Obj : Primitives)
	{
		if (Obj.Idx == ChosenObjectIdx)
		{
			if (Obj.type ==
(cl_float)EObjectType::Sphere)
			{

	MoveChosenSphere(Obj, Movement);
```

```
                    }
                    else if (Obj.type ==
(cl_float)EObjectType::Mesh)
                    {

        MoveChosenMesh(Obj, Movement);
                    }
                    return;
                }
            }
}

void
USceneSubsystem::MoveChosenSphere(FObject&
Sphere, cl_float4 Movement)
{
        Sphere.center = { Sphere.center.x +
Movement.x, Sphere.center.y + Movement.y,
Sphere.center.z + Movement.z };
}

void USceneSubsystem::MoveChosenMesh(FObject&
MeshPrimitive, cl_float4 Movement)
{
        MeshPrimitive.center = {
MeshPrimitive.center.x + Movement.x,
MeshPrimitive.center.y + Movement.y,
MeshPrimitive.center.z + Movement.z };
}

void
USceneSubsystem::AddMeshToScene(std::shared_ptr<
FMeshObject> NewMesh)
{
        static cl_int LastUniqueID = 0;
        NewMesh->UniqueID = LastUniqueID++;

        ComputeBoundingBoxForMesh(*NewMesh);

        FGPUMeshObject
NewOptimizedMeshStructure;
        NewOptimizedMeshStructure.BoundingBox[0
] = NewMesh->BoundingBox[0];
        NewOptimizedMeshStructure.BoundingBox[1
] = NewMesh->BoundingBox[1];
        NewOptimizedMeshStructure.NumFaces =
NewMesh->Faces.size();
        NewOptimizedMeshStructure.UniqueID =
NewMesh->UniqueID;
        NewOptimizedMeshStructure.StartingFaceIdx
= AllSceneFaces.size();
        NewOptimizedMeshStructure.StartingNormal
Idx = AllSceneNormals.size();
        NewOptimizedMeshStructure.StartingVertexI
dx = AllSceneVertices.size();
        NewOptimizedMeshStructure.StartingTexCoo
rdIdx = AllSceneTexCoords.size();

        AllSceneFaces.reserve(AllSceneFaces.size() +
NewMesh->Faces.size());
        AllSceneNormals.reserve(AllSceneFaces.size(
) + NewMesh->Normals.size());
        AllSceneTexCoords.reserve(AllSceneFaces.si
ze() + NewMesh->TextureCoords.size());
        AllSceneVertices.reserve(AllSceneFaces.size()
+ NewMesh->Vertices.size());

        for (FFace& Face : NewMesh->Faces)
        {
                {
                        const Vertex& v0 =
NewMesh->Vertices[Face.Vertices.s0];
                        const Vertex& v1 =
NewMesh->Vertices[Face.Vertices.s1];
                        const Vertex& v2 =
NewMesh->Vertices[Face.Vertices.s2];

                        cl_float4 edge1 = { v1.x -
v0.x, v1.y - v0.y, v1.z - v0.z };
                        cl_float4 edge2 = { v2.x -
v0.x, v2.y - v0.y, v2.z - v0.z };
                        cl_float4 Cross =
FMath::CrossProduct(edge1, edge2);
                        cl_float4 normal =
FMath::NormalizeVector(Cross);
                        Face.PreComputedNormal =
normal;
                }
                AllSceneFaces.push_back(Face);
        }
        for (const Normal& Face : NewMesh-
>Normals)
        {
                AllSceneNormals.push_back(Face);
        }
        for (const UV& Face : NewMesh-
>TextureCoords)
        {

        AllSceneTexCoords.push_back(Face);
        }
        for (const Vertex& Face : NewMesh-
>Vertices)
        {
                AllSceneVertices.push_back(Face);
        }

        Meshes.push_back(NewMesh);
        OptimizedMeshes.push_back(NewOptimized
MeshStructure);

        FObject CompabilityObject;
        CompabilityObject.type =
static_cast<cl_int>(EObjectType::Mesh);
        CompabilityObject.meshidx =
NewOptimizedMeshStructure.UniqueID;
        CompabilityObject.color = {127, 127, 127, 0};
        CompabilityObject.Idx = ++MaxIdx;
        Primitives.push_back(CompabilityObject);

        ChosenObjectIdx = CompabilityObject.Idx;

        OnSceneUpdated();
}
```

```
void
USceneSubsystem::UpdateMeshInScene(std::shared_pt
r<FMeshObject> InMesh, cl_float4 NewLocation,
cl_float4 NewRotation)
{
	OnSceneUpdated();
}

bool USceneSubsystem::FindGPUMeshByID(cl_int
InUniqueID, FGPUMeshObject& OutObject)
{
	for (const FGPUMeshObject& GPUMesh :
OptimizedMeshes)
	{
		if (GPUMesh.UniqueID ==
InUniqueID)
		{
			OutObject = GPUMesh;
			return true;
		}
	}
	return false;
}

void USceneSubsystem::OnSceneUpdated()
{
	//UGPUComputingSubsystem::Get().MainPro
gram->PrepareProgram();
}

void
USceneSubsystem::ComputeBoundingBoxForMesh(F
MeshObject& InMesh)
{
	InMesh.BoundingBox[0] = {
std::numeric_limits<float>::infinity(),
std::numeric_limits<float>::infinity(),
std::numeric_limits<float>::infinity(), 0 };
	InMesh.BoundingBox[1] = { -
std::numeric_limits<float>::infinity(), -
std::numeric_limits<float>::infinity(), -
std::numeric_limits<float>::infinity(), 0 };

	for (Vertex& Vectice : InMesh.Vertices)
	{
		InMesh.BoundingBox[0].x =
std::min(InMesh.BoundingBox[0].x, Vectice.x);
		InMesh.BoundingBox[0].y =
std::min(InMesh.BoundingBox[0].y, Vectice.y);
		InMesh.BoundingBox[0].z =
std::min(InMesh.BoundingBox[0].z, Vectice.z);

		InMesh.BoundingBox[1].x =
std::max(InMesh.BoundingBox[1].x, Vectice.x);
		InMesh.BoundingBox[1].y =
std::max(InMesh.BoundingBox[1].y, Vectice.y);
		InMesh.BoundingBox[1].z =
std::max(InMesh.BoundingBox[1].z, Vectice.z);
	}
}
```

UUISubsystem.cpp

```
#include "UUISubsystem.h"
#include "SDL_ttf.h"
#include "SDLWindowSubsystem.h"
#include "USceneSubsystem.h"
#include "UInputSubsystem.h"
#include <sstream>
#include <iomanip>

#pragma optimize("", off)

bool UUISubsystem::Initialize()
{
	// Draw button border
	UIPanel::CreateUIPanel({0, 0}, {300, 1080},
{255, 255, 255}, {0, 0, 0});

	SpawnSphereButton =
UIButton::CreateUIButton({ 0, 25 }, { 300, 50 },
"Spawn Sphere", { 0, 0, 0 }, { 127, 127, 127 }, { 255,
255, 255 }, { 5, 5 });
	SpawnSphereButton-
>OnButtonClickedDelegate.Bind([](bool bClicked){ if
(bClicked)
USceneSubsystem::Get().SpawnSphere();});

	SpawnObjectButton =
UIButton::CreateUIButton({ 0, 76 }, { 300, 50 },
"Spawn Mesh", { 0, 0, 0 }, { 127, 127, 127 }, { 255,
255, 255 }, { 5, 5 });
	SpawnObjectButton-
>OnButtonClickedDelegate.Bind([](bool bClicked)
	{
		if (bClicked)

	USceneSubsystem::Get().SpawnMesh();
	}
	);

	DestroyObjectButton =
UIButton::CreateUIButton({ 0, 127 }, { 300, 50 },
"Destroy chosen object", { 0, 0, 0 }, { 127, 127, 127 },
{ 255, 255, 255 }, { 5, 5 });
	DestroyObjectButton-
>OnButtonClickedDelegate.Bind([](bool bClicked) { if
(bClicked)
USceneSubsystem::Get().DestroyChosenObject(); });

	{
		ChosenObjectLocationText =
UIButton::CreateUIButton({ 0, 178 }, { 300, 75 },
"Object Location", { 0, 0, 0 }, { 127, 127, 127 }, {255,
255, 255}, {5, 5});
		ChosenObjectLocationX =
UIInputText::CreateUIInputText({ 0, 203 }, { 75, 25 },
{ 0, 0, 0 }, { 255, 255, 255 }, { 255, 255, 255 }, { 5, 5
});
		ChosenObjectLocationX-
>OnTextUpdatedDelegate.BindStr([](std::string
InText)
			{
```

```
FObject* Obj =
USceneSubsystem::Get().GetChosenObject();
if (Obj)
{
Obj-
>center.x = std::stof(InText);
}
});

ChosenObjectLocationY =
UIInputText::CreateUIInputText({ 76, 203 }, { 75, 25
}, { 0, 0, 0 }, { 255, 255, 255 }, { 255, 255, 255 }, { 5,
5 });
ChosenObjectLocationY-
>OnTextUpdatedDelegate.BindStr([](std::string
InText)
{
FObject* Obj =
USceneSubsystem::Get().GetChosenObject();
if (Obj)
{
Obj-
>center.y = std::stof(InText);
}
});

ChosenObjectLocationZ =
UIInputText::CreateUIInputText({ 152, 203 }, { 75, 25
}, { 0, 0, 0 }, { 255, 255, 255 }, { 255, 255, 255 }, { 5,
5 });
ChosenObjectLocationZ-
>OnTextUpdatedDelegate.BindStr([](std::string
InText)
{
FObject* Obj =
USceneSubsystem::Get().GetChosenObject();
if (Obj)
{
Obj-
>center.z = std::stof(InText);
}
});
}

{
ChosenObjectColorText =
UIButton::CreateUIButton({ 0, 254 }, { 300, 75 },
"Object Color", { 0, 0, 0 }, { 127, 127, 127 }, { 255,
255, 255 }, { 5, 5 });
ChosenObjectColorX =
UIInputText::CreateUIInputText({ 0, 279 }, { 75, 25 },
{ 0, 0, 0 }, { 255, 255, 255 }, { 255, 255, 255 }, { 5, 5
});
ChosenObjectColorX-
>OnTextUpdatedDelegate.BindStr([](std::string
InText)
{
FObject* Obj =
USceneSubsystem::Get().GetChosenObject();
if (Obj)
{
Obj-
>color.x = std::stoi(InText);
}
});

ChosenObjectColorY =
UIInputText::CreateUIInputText({ 76, 279 }, { 75, 25
}, { 0, 0, 0 }, { 255, 255, 255 }, { 255, 255, 255 }, { 5,
5 });
ChosenObjectColorY-
>OnTextUpdatedDelegate.BindStr([](std::string
InText)
{
FObject* Obj =
USceneSubsystem::Get().GetChosenObject();
if (Obj)
{
Obj-
>color.y = std::stoi(InText);
}
});

ChosenObjectColorZ =
UIInputText::CreateUIInputText({ 152, 279 }, { 75, 25
}, { 0, 0, 0 }, { 255, 255, 255 }, { 255, 255, 255 }, { 5,
5 });
ChosenObjectColorZ-
>OnTextUpdatedDelegate.BindStr([](std::string
InText)
{
FObject* Obj =
USceneSubsystem::Get().GetChosenObject();
if (Obj)
{
Obj-
>color.z = std::stoi(InText);
}
});
}

{
ChosenObjectReflectionText =
UIButton::CreateUIButton({ 0, 330 }, { 300, 75 },
"Object Reflection", { 0, 0, 0 }, { 127, 127, 127 }, {
255, 255, 255 }, { 5, 5 });
ChosenObjectReflection =
UIInputText::CreateUIInputText({ 100, 355 }, { 75, 25
}, { 0, 0, 0 }, { 255, 255, 255 }, { 255, 255, 255 }, { 5,
5 });
ChosenObjectReflection-
>OnTextUpdatedDelegate.BindStr([](std::string
InText)
{
FObject* Obj =
USceneSubsystem::Get().GetChosenObject();
if (Obj)
{
Obj-
>reflection = std::stof(InText);
}
});
}

{
```

```
			ReflectionIterationsText =
UIButton::CreateUIButton({ 0, 406 }, { 300, 75 },
"Recursion number", { 0, 0, 0 }, { 127, 127, 127 }, {
255, 255, 255 }, { 5, 5 });
			ReflectionIterations =
UIInputText::CreateUIInputText({ 100, 431 }, { 75, 25
}, { 0, 0, 0 }, { 255, 255, 255 }, { 255, 255, 255 }, { 5,
5 });
			ReflectionIterations-
>OnTextUpdatedDelegate.BindStr([](std::string
InText)
				{

		USceneSubsystem::Get().SceneData-
>RecursionLevel = std::stoi(InText);
				});
		}

		{
			SpawnLightSource =
UIButton::CreateUIButton({ 0, 482 }, { 300, 50 },
"Spawn Light", { 0, 0, 0 }, { 127, 127, 127 }, { 255,
255, 255 }, { 5, 5 });
			SpawnLightSource-
>OnButtonClickedDelegate.Bind([](bool bClicked) { if
(bClicked)
USceneSubsystem::Get().SpawnLightSource(); });

			ChoosePrevLightSource =
UIButton::CreateUIButton({ 0, 534 }, { 140, 47 },
"Prev Light", { 0, 0, 0 }, { 127, 127, 127 }, { 255, 255,
255 }, { 5, 5 });
			ChoosePrevLightSource-
>OnButtonClickedDelegate.Bind([](bool bClicked) { if
(bClicked)
USceneSubsystem::Get().ChoosePrevLightSource();
});

			ChooseNextLightSource =
UIButton::CreateUIButton({ 160, 534 }, { 140, 47 },
"Next Light", { 0, 0, 0 }, { 127, 127, 127 }, { 255, 255,
255 }, { 5, 5 });
			ChooseNextLightSource-
>OnButtonClickedDelegate.Bind([](bool bClicked) { if
(bClicked)
USceneSubsystem::Get().ChooseNextLightSource();
});
		}

		{
			ChosenLightSourceLocationText =
UIButton::CreateUIButton({ 0, 583 }, { 300, 75 },
"Light Location", { 0, 0, 0 }, { 127, 127, 127 }, { 255,
255, 255 }, { 5, 5 });
			ChosenLightSourceLocationX =
UIInputText::CreateUIInputText({ 0, 608 }, { 75, 25 },
{ 0, 0, 0 }, { 255, 255, 255 }, { 255, 255, 255 }, { 5, 5
});
			ChosenLightSourceLocationX-
>OnTextUpdatedDelegate.BindStr([](std::string
InText)
				{
					FLightSource*
LSource =
USceneSubsystem::Get().GetChosenLightSource();
					if (LSource)
					{
						LSource-
>Origin.x = std::stof(InText);
					}
				});

			ChosenLightSourceLocationY =
UIInputText::CreateUIInputText({ 76, 608 }, { 75, 25
}, { 0, 0, 0 }, { 255, 255, 255 }, { 255, 255, 255 }, { 5,
5 });
			ChosenLightSourceLocationY-
>OnTextUpdatedDelegate.BindStr([](std::string
InText)
				{
					FLightSource*
LSource =
USceneSubsystem::Get().GetChosenLightSource();
					if (LSource)
					{
						LSource-
>Origin.y = std::stof(InText);
					}
				});

			ChosenLightSourceLocationZ =
UIInputText::CreateUIInputText({ 152, 608 }, { 75, 25
}, { 0, 0, 0 }, { 255, 255, 255 }, { 255, 255, 255 }, { 5,
5 });
			ChosenLightSourceLocationZ-
>OnTextUpdatedDelegate.BindStr([](std::string
InText)
				{
					FLightSource*
LSource =
USceneSubsystem::Get().GetChosenLightSource();
					if (LSource)
					{
						LSource-
>Origin.z = std::stof(InText);
					}
				});
		}

		{
			ChosenLightSourceColorText =
UIButton::CreateUIButton({ 0, 659 }, { 300, 75 },
"Light Color", { 0, 0, 0 }, { 127, 127, 127 }, { 255,
255, 255 }, { 5, 5 });
			ChosenLightSourceColorX =
UIInputText::CreateUIInputText({ 0, 684 }, { 75, 25 },
{ 0, 0, 0 }, { 255, 255, 255 }, { 255, 255, 255 }, { 5, 5
});
			ChosenLightSourceColorX-
>OnTextUpdatedDelegate.BindStr([](std::string
InText)
				{
					FLightSource*
LSource =
USceneSubsystem::Get().GetChosenLightSource();
```

```
					if (LSource)
					{
						LSource-
>Color.x = std::stoi(InText);
					}
				});

			ChosenLightSourceColorY =
UIInputText::CreateUIInputText({ 76, 684 }, { 75, 25
}, { 0, 0, 0 }, { 255, 255, 255 }, { 255, 255, 255 }, { 5,
5 });
			ChosenLightSourceColorY-
>OnTextUpdatedDelegate.BindStr([](std::string
InText)
				{
					FLightSource*
LSource =
USceneSubsystem::Get().GetChosenLightSource();
					if (LSource)
					{
						LSource-
>Color.y = std::stoi(InText);
					}
				});

			ChosenLightSourceColorZ =
UIInputText::CreateUIInputText({ 152, 684 }, { 75, 25
}, { 0, 0, 0 }, { 255, 255, 255 }, { 255, 255, 255 }, { 5,
5 });
			ChosenLightSourceColorZ-
>OnTextUpdatedDelegate.BindStr([](std::string
InText)
				{
					FLightSource*
LSource =
USceneSubsystem::Get().GetChosenLightSource();
					if (LSource)
					{
						LSource-
>Color.z = std::stoi(InText);
					}
				});
		}
		{
			ChosenLightSourceIntensityText =
UIButton::CreateUIButton({ 0, 735 }, { 300, 75 },
"Light intensity", { 0, 0, 0 }, { 127, 127, 127 }, { 255,
255, 255 }, { 5, 5 });
			ChosenLightSourceIntensity =
UIInputText::CreateUIInputText({ 100, 760 }, { 75, 25
}, { 0, 0, 0 }, { 255, 255, 255 }, { 255, 255, 255 }, { 5,
5 });
			ChosenLightSourceIntensity-
>OnTextUpdatedDelegate.BindStr([](std::string
InText)
				{
					FLightSource*
LSource =
USceneSubsystem::Get().GetChosenLightSource();
					if (LSource)
					{
						LSource-
>Intensity = std::stof(InText);
					}
				});
		}
		{
			DestroyLightSourceButton =
UIButton::CreateUIButton({ 0, 811 }, { 300, 50 },
"Destroy chosen light", { 0, 0, 0 }, { 127, 127, 127 }, {
255, 255, 255 }, { 5, 5 });
			DestroyLightSourceButton-
>OnButtonClickedDelegate.Bind([](bool bClicked) { if
(bClicked)
USceneSubsystem::Get().RemoveChosenLightSource()
; });
		}
		return true;
}

static std::string FloatToString(float InNumber)
{
		std::stringstream Stream;
		Stream << std::fixed << std::setprecision(2)
<< InNumber;
		return Stream.str();
}

static std::string IntToString(int InNumber)
{
		std::stringstream Stream;
		Stream << InNumber;
		return Stream.str();
}

bool UUISubsystem::Update(float InDeltaTime)
{
		USceneSubsystem& SceneSubsystem =
USceneSubsystem::Get();
		const FObject* ObjectPtr =
SceneSubsystem.GetChosenObject();
		if (ObjectPtr)
		{
			const FObject& Object = *ObjectPtr;

			if (!ChosenObjectLocationX-
>bIsActive)
				ChosenObjectLocationX-
>UpdateText(FloatToString(Object.center.x));
			if (!ChosenObjectLocationY-
>bIsActive)
				ChosenObjectLocationY-
>UpdateText(FloatToString(Object.center.y));
			if (!ChosenObjectLocationZ-
>bIsActive)
				ChosenObjectLocationZ-
>UpdateText(FloatToString(Object.center.z));

			if (!ChosenObjectColorX-
>bIsActive)
				ChosenObjectColorX-
>UpdateText(IntToString(Object.color.x));
			if (!ChosenObjectColorY->bIsActive)
				ChosenObjectColorY-
>UpdateText(IntToString(Object.color.y));
			if (!ChosenObjectColorZ->bIsActive)
```

```
                        ChosenObjectColorZ-
>UpdateText(IntToString(Object.color.z));

                if (!ChosenObjectReflection-
>bIsActive)
                        ChosenObjectReflection-
>UpdateText(FloatToString(Object.reflection));
        }
        else
        {
                ChosenObjectLocationX-
>UpdateText("");
                ChosenObjectLocationY-
>UpdateText("");
                ChosenObjectLocationZ-
>UpdateText("");
                ChosenObjectColorX-
>UpdateText("");
                ChosenObjectColorY-
>UpdateText("");
                ChosenObjectColorZ-
>UpdateText("");
                ChosenObjectReflection-
>UpdateText("");
        }

        FLightSource* LSource =
SceneSubsystem.GetChosenLightSource(nullptr);
        if (LSource)
        {
                if (!ChosenLightSourceLocationX-
>bIsActive)

        ChosenLightSourceLocationX-
>UpdateText(FloatToString(LSource->Origin.x));
                if (!ChosenLightSourceLocationY-
>bIsActive)

        ChosenLightSourceLocationY-
>UpdateText(FloatToString(LSource->Origin.y));
                if (!ChosenLightSourceLocationZ-
>bIsActive)

        ChosenLightSourceLocationZ-
>UpdateText(FloatToString(LSource->Origin.z));

                if (!ChosenLightSourceColorX-
>bIsActive)
                        ChosenLightSourceColorX-
>UpdateText(IntToString(LSource->Color.x));
                if (!ChosenLightSourceColorY-
>bIsActive)
                        ChosenLightSourceColorY-
>UpdateText(IntToString(LSource->Color.y));
                if (!ChosenLightSourceColorZ-
>bIsActive)
                        ChosenLightSourceColorZ-
>UpdateText(IntToString(LSource->Color.z));

                if (!ChosenLightSourceIntensity-
>bIsActive)

        ChosenLightSourceIntensity-
>UpdateText(FloatToString(LSource->Intensity));
        }
        else
        {
                ChosenLightSourceLocationX-
>UpdateText("");
                ChosenLightSourceLocationY-
>UpdateText("");
                ChosenLightSourceLocationZ-
>UpdateText("");
                ChosenLightSourceColorX-
>UpdateText("");
                ChosenLightSourceColorY-
>UpdateText("");
                ChosenLightSourceColorZ-
>UpdateText("");
                ChosenLightSourceIntensity-
>UpdateText("");
        }

        if (!ReflectionIterations->bIsActive)
                ReflectionIterations-
>UpdateText(IntToString(SceneSubsystem.SceneData-
>RecursionLevel));

        UInputSubsystem& InputSubsystem =
UInputSubsystem::Get();
        for (int Iter = Buttons.size() - 1; Iter >= 0; --
Iter)
        {
                std::shared_ptr<UIButton> Button =
Buttons[Iter];
                if (Button.get() != nullptr)
                {
                        if (Button-
>IsPointedByMouse(InputSubsystem.MousePosition.x,
InputSubsystem.MousePosition.y))
                        {
                                Button-
>bIsHovered = true;
                                if (Button-
>bIsClicked !=
InputSubsystem.bLeftMouseButtonDown)
                                {
                                        Button-
>OnButtonClicked(InputSubsystem.bLeftMouseButton
Down);
                                }
                        }
                        else
                        {
                                Button-
>bIsHovered = false;
                                Button->bIsClicked
= false;
                        }
                }
        }

        for (int Iter = InputTextFields.size() - 1; Iter
>= 0; --Iter)
```

```
		{
			std::shared_ptr<UIInputText>
TextField = InputTextFields[Iter];
			if (TextField.get() != nullptr)
			{
				if (TextField-
>IsPointedByMouse(InputSubsystem.MousePosition.x,
InputSubsystem.MousePosition.y))
				{
					TextField-
>bIsHovered = true;
					if (TextField-
>bIsClicked !=
InputSubsystem.bLeftMouseButtonDown)
					{
						TextField-
>bIsClicked =
InputSubsystem.bLeftMouseButtonDown;
						if
(TextField->bIsClicked)
						{

	SDL_StartTextInput();

	TextField->bIsActive = true;

	InputSubsystem.CurrentInputText =
TextField->Text;
						}
					}
				}
				else
				{
					TextField-
>bIsHovered = false;
					TextField-
>bIsClicked = false;
				}
			}
		}

	return true;
}

std::string UUISubsystem::GetSubsystemName() const
{
	return "UISubsystem";
}

void UUISubsystem::Destroy()
{

}

std::shared_ptr<UIInputText>
UUISubsystem::GetActiveInputTextField() const
{
	for (std::shared_ptr<UIInputText> Field :
InputTextFields)
	{
		if (Field->bIsActive)
		{
			return Field;
		}
	}
	return nullptr;
}
```

Main.cpp

```
#define CL_HPP_TARGET_OPENCL_VERSION 300

#define _CRT_SECURE_NO_WARNINGS

#define RETURN_IF_FALSE(x) if (!(x)) {return
false;}

#pragma optimize("", off)

#include "SDLWindowSubsystem.h"
#include "GPUComputingSubsystem.h"
#include <Windows.h>
#include "MeshObjectParser.h"
#include "USceneSubsystem.h"
#include "UInputSubsystem.h"
#include "UUISubsystem.h"

void Initialize()
{
	AllocConsole();
	freopen("CONOUT$", "w", stdout);
	freopen("CONOUT$", "w", stderr);

	USDLWindowSubsystem::Get().Initialize();
	UInputSubsystem::Get().Initialize();
	UMeshObjectParserSubsystem::Get().Initializ
e();
	USceneSubsystem::Get().Initialize();
	UUISubsystem::Get().Initialize();
	UGPUComputingSubsystem::Get().Initialize()
;

	USceneSubsystem& Scene =
USceneSubsystem::Get();

	{


	//UMeshObjectParserSubsystem::Get().Parse
MeshObjectFromDialogWindow();


	//UMeshObjectParserSubsystem::Get().Parse
MeshObjectFromDialogWindow();

		//Scene.SpawnSphere({ 6, 0, 6 },
2.5f, { 194, 13, 13 }, 0.0f);

		//Scene.SpawnSphere({ 6, 0, -6 },
2.5f, { 50, 100, 75 }, 0.2f);

		//Scene.SpawnSphere({ -6, 0, 6 },
2.5f, { 13, 255, 67 }, 0.5f);
```

```
			//Scene.SpawnSphere({ -6, 0, -6 },
3.f, { 0, 255, 0 }, 0.5f);

			//Scene.SpawnSphere({ 6, 0, 6 }, 3.f,
{ 255, 255, 255 }, 0.5f);

			//Scene.SpawnSphere({ 6, 0, -6 }, 3.f,
{ 0, 0, 255 }, 0.5f);

			//Scene.SpawnSphere({ -6, 0, 6 }, 3.f,
{ 255, 0, 0 }, 0.5f);

			//Scene.SpawnLightSource({ 100, -1,
0 }, { 0xff, 0xff, 0xff }, 1.f);
			//Scene.SpawnLightSource({ 0, 100,
0 }, { 0xff, 0xff, 0xff }, 1.f);
		}
}

void Quit()
{
	UGPUComputingSubsystem::Get().Destroy();
	USceneSubsystem::Get().Destroy();
	UMeshObjectParserSubsystem::Get().Destroy
();
	UInputSubsystem::Get().Destroy();
	UUISubsystem::Get().Destroy();
	USDLWindowSubsystem::Get().Destroy();
}

bool Update(float InDeltaTime)
{
	RETURN_IF_FALSE(UInputSubsystem::Get(
).Update(InDeltaTime));
	RETURN_IF_FALSE(USceneSubsystem::Get
().Update(InDeltaTime));
	RETURN_IF_FALSE(UGPUComputingSubs
ystem::Get().Update(InDeltaTime));
	RETURN_IF_FALSE(UUISubsystem::Get().
Update(InDeltaTime));
	RETURN_IF_FALSE(USDLWindowSubsyste
m::Get().Update(InDeltaTime));
	return true;
}

int WINAPI WinMain(_In_ HINSTANCE hInstance,
	_In_opt_ HINSTANCE hPrevInstance,
	_In_ LPSTR lpCmdLine,
	_In_ int nShowCmd)
{
	Initialize();

	Uint64 MilisecondsLastFrame =
SDL_GetPerformanceCounter();
	cl_float DeltaTime = 0.f;

	bool bShouldContinue = true;
	while (bShouldContinue)
	{
		Uint64 MilisecondsStart =
SDL_GetPerformanceCounter();
		bShouldContinue =
Update(DeltaTime);
		Uint64 MilisecondsEnd =
SDL_GetPerformanceCounter();

		DeltaTime = (MilisecondsEnd -
MilisecondsStart) /
(cl_float)SDL_GetPerformanceFrequency();
	}

	Quit();

   return 0;
}
```

Render.cl

```
/*
** Protopypes
*/

# define SPHERE 1
# define PLANE 2
# define MESH 3

# define DIRECFLightSource 1
# define DOFLightSource 2
# define PROJECTOR_LIGHT 3
# define AMBIENFLightSource 4

typedef float4 Vertex;
typedef float4 Normal;
typedef int4 FColor;
typedef float2 UV;

static __constant const float EPSILON = 0.00001f;

__attribute__((aligned(16))) typedef struct
FGPUMeshObject
{
	float4 BoundingBox[2];
	int StartingFaceIdx;
	int StartingVertexIdx;
	int StartingNormalIdx;
	int StartingTexCoordIdx;
	int NumFaces;
	int UniqueID;
} FGPUMeshObject;

__attribute__((aligned(16))) typedef struct
FDirectionalRay
{
	float4 start;
	float4 ray;
	float len;
} FDirectionalRay;

__attribute__((aligned(16))) typedef struct FSavedObj
```

```
{
    FDirectionalRay ray;
    float4 intersection_point;
    float4 normal;
    __global const struct s_object *obj;
    float  distance;
    int recursion;
} FSavedObj;

__attribute__((aligned(16))) typedef struct s_object
{
    float4 center;
    float4 axis;
    float4 normal;
    FColor color;
    float pow_radius;
    float reflection;
    int specularity;
    int meshidx;
    int type;
    int Idx;
} FObject;

__attribute__((aligned(16))) typedef struct s_Face
{
    int4 Vertices;
    int4 Normals;
    int4 TexCoords;

    float4 PreComputedNormal;
} FFace;

__attribute__((aligned(16)))
__attribute__((aligned(16))) typedef struct s_Camera
{
    float4 Origin;
    float4 Rotation;
} FCamera;

__attribute__((aligned(16))) typedef struct
s_LightSource
{
    float4 Origin;
    FColor Color;
    float Intensity;
    int Idx;
} FLightSource;

__attribute__((aligned(16))) typedef struct
s_SceneData
{
    FCamera Camera;
    FColor AmbientLightColor;

    float ScreenRelativeDistance;
    float ScreenRelativeWidth;
    float ScreenRelativeHeight;

    float RelationX;
    float RelationY;

    int WindowHeight;
    int WindowWidth;
    float AmbientLightIntensity;

    uint NumPixels;
    uint NumFaces;
    uint NumPrimitives;
    uint NumMeshes;
    uint NumLightSources;
    uint NumVertices;
    uint NumNormals;
    uint NumTexCoords;

    int RecursionLevel;
} FSceneData;

FDirectionalRay CreateDirectionalRay(float4 start,
float4 direction);
FDirectionalRay CreateDirectionalRay2(float4 start,
float4 direction);

float ray_cos(float4 vect1, float4 vect2);

float4 rotate_z(float4 rotate, float4 ray);
float4 rotate_xy(float4 rotate, float4 ray);
float4 rotate_vector(float4 vector, float4 rotate);

int myfloor(int num, float rel);
FColor ClampRGB(FColor rgb, FColor limit);
int rgb_to_int(FColor c);
FColor mult_color(FColor color, float mult);

FColor MultiplicateLightSources(FColor obj_color,
FColor light_color, float obj_mult, float light_mult);
FColor mult_colors(FColor obj_color, FColor
light_color, FColor spec_col);
FColor mix_colors(FColor color, FColor color2);

FColor sample(__global const FObject *objs,
    __global const FLightSource *lights,
    __local const FSceneData* SceneData,
    __global const FGPUMeshObject* Meshes,
    __global const Vertex* Vertices,
    __global const FFace* Faces,
    __global const Normal* Normals,
    float y,
    float x);

int choose_intersect(__global const FObject *obj, const
FDirectionalRay *ray, FSavedObj* save, float *t,
__global const FGPUMeshObject* Meshes, __global
const Vertex* Vertices, __global const FFace* Faces,
__global const Normal* Normals);
float4 choose_normal(FSavedObj *save, __global
const FGPUMeshObject* Meshes);
FColor choose_color(FSavedObj const *const save,
__global const FGPUMeshObject* Meshes, __global
const Vertex* Vertices, __global const FFace* Faces,
__global const Normal* Normals);

FColor get_pixel_color(FSavedObj *save, __global
const FObject *objs, __global const FLightSource
*light, __local const FSceneData* SceneData, __global
const FGPUMeshObject* Meshes, __global const
```

```
Vertex* Vertices, __global const FFace* Faces,
__global const Normal* Normals);

float get_shadow(const FDirectionalRay ray, __global
const FObject *obj, FSavedObj *save, __global const
FGPUMeshObject* Meshes, __global const Vertex*
Vertices, __global const FFace* Faces, __global const
Normal* Normals);

float diffuse(const float4 intersection_point, const
float4 normal, __global const FLightSource *light);
float specular(const float4 normal, const float4
ray_direction, __global const FLightSource *light,
const FSavedObj *save);

float4 view3d(__local const FSceneData* SceneData,
float y, float x);

int plane_intersection(__global const FObject *obj,
const FDirectionalRay *ray, float *t);
float4 plane_normal(FSavedObj const *const save);
FColor plane_color(FSavedObj const *const save);

int sphere_intersection(__global const FObject *obj,
const FDirectionalRay *ray, float *t);
float4 sphere_normal(FSavedObj const *const save);
FColor sphere_color(FSavedObj const *const save);

int mesh_intersection(__global const FObject *obj,
const FDirectionalRay *ray, FSavedObj* save, float *t,
__global const FGPUMeshObject* Meshes, __global
const Vertex* Vertices, __global const FFace* Faces,
__global const Normal* Normals);
float4 mesh_normal(FSavedObj const *const save);
FColor mesh_color(FSavedObj const *const save);

int sq_equation(float4 k, float *res);
int intersect_point(const FDirectionalRay *ray,
__global const FObject *obj, FSavedObj *save,
__global const FObject *ignore, __local const
FSceneData* SceneData, __global const
FGPUMeshObject* Meshes, __global const Vertex*
Vertices, __global const FFace* Faces, __global const
Normal* Normals);

FColor reflection(FSavedObj *save, __local const
FSceneData* SceneData, __global const FObject
*objs, __global const FLightSource *lights, int rec, int
max_rec, __global const FGPUMeshObject* Meshes,
__global const Vertex* Vertices, __global const FFace*
Faces, __global const Normal* Normals);

FDirectionalRay reflect_ray(float4 ray, FSavedObj
*save);
int4 return_color(__local const FSceneData*
SceneData, int4 *color, FSavedObj *save, int i);

inline void ForceCrash()
{
	float crash = 1 / 0;
	__global int *nus = 0;
	float* kek = (float*)nus;
	*kek = INFINITY;
}

/*
** vector and lines
*/

inline FDirectionalRay CreateDirectionalRay(float4
start, float4 direction)
{
	FDirectionalRay line;

	line.ray = normalize(direction - start);
	line.start = start + (line.ray * EPSILON);
	line.len = dot(line.ray, line.ray);
	return (line);
}

inline FDirectionalRay CreateDirectionalRay2(float4
start, float4 direction)
{
	FDirectionalRay line;

	line.ray = direction - start;
	line.start = start + (normalize(line.ray) *
EPSILON);
	line.len = dot(line.ray, line.ray);
	return (line);
}

inline float ray_cos(float4 vect1, float4 vect2)
{
	float save = dot(vect1, vect2);

	if (save <= 0)
		return (-1);
	save /= length(vect1);
	save /= length(vect2);
	return (save);
}

inline float4 rotate_z(float4 rotate, float4 ray)
{
	const float	cos_z = cos(rotate.z);
	const float	sin_z = sin(rotate.z);

	rotate.x = (ray.x * cos_z) - (ray.y * sin_z);
	rotate.y = (ray.x * sin_z) + (ray.y * cos_z);
	rotate.z = ray.z;
	return (rotate);
}

inline float4 rotate_xy(float4 rotate, float4 ray)
{
	const float	cos_x = cos(rotate.x);
	const float	sin_x = sin(rotate.x);
	const float	cos_y = cos(rotate.y);
	const float	sin_y = sin(rotate.y);

	rotate.y = (ray.y * cos_x) - (ray.z * sin_x);
	rotate.z = (ray.y * sin_x) + (ray.z * cos_x);
	rotate.x = ray.x;
	ray = rotate;
	rotate.x = (ray.x * cos_y) + (ray.z * sin_y);
```

```
        rotate.z = (-ray.x * sin_y) + (ray.z * cos_y);
        rotate.y = ray.y;
        return (rotate);
}

inline float4 rotate_vector(float4 vector, float4 rotate)
{
        vector = rotate_xy(rotate, vector);
        vector = rotate_z(rotate, vector);
        return (vector);
}


/*
** Colors
*/

inline int myfloor(int num, float rel)
{
        return ((int)floor(num * rel));
}

inline FColor mult_color(FColor color, float mult)
{
        color.x = (int)(floor(color.x * mult));
        color.y = (int)(floor(color.y * mult));
        color.z = (int)(floor(color.z * mult));
        return (color);
}

inline FColor ClampRGB(FColor rgb, FColor limit)
{
        if (rgb.x > limit.x)
                rgb.x = limit.x;
        else if (rgb.x < 0)
                rgb.x = 0;
        if (rgb.y > limit.y)
                rgb.y = limit.y;
        else if (rgb.y < 0)
                rgb.y = 0;
        if (rgb.z > limit.z)
                rgb.z = limit.z;
        else if (rgb.z < 0)
                rgb.z = 0;
        return (rgb);
}

inline FColor MultiplicateLightSources(FColor
obj_color, FColor light_color, float obj_mult, float
light_mult)
{
        const float        mult = obj_mult * (1.0f -
light_mult);

        obj_color.x = myfloor(obj_color.x, mult) +
myfloor(light_color.x, light_mult);
        obj_color.y = myfloor(obj_color.y, mult) +
myfloor(light_color.y, light_mult);
        obj_color.z = myfloor(obj_color.z, mult) +
myfloor(light_color.z, light_mult);
        obj_color = ClampRGB(obj_color,
(FColor)(255, 255, 255, 0));
        return (obj_color);
}

inline FColor mult_colors(FColor obj_color, FColor
light_color, FColor spec_color)
{
        return ClampRGB(ClampRGB(light_color,
obj_color) + spec_color, (FColor)(255, 255, 255, 0));
}

inline FColor mix_colors(FColor color, FColor color2)
{
        FColor tmp = {color.x / 255.0f, color.y /
255.0f, color.z / 255.0f, 0};

        color.x += floor(color2.x * (1.0f - tmp.x));
        color.y += floor(color2.y * (1.0f - tmp.y));
        color.z += floor(color2.z * (1.0f - tmp.z));
        return (color);
}

int rgb_to_int(FColor c)
{
        return ((int)c.x << 24) | ((int)c.y << 16) |
((int)c.z << 8);
}

/*
** Light and shadow
*/

/*
** i_s is intension and specularity
*/


FColor get_pixel_color(FSavedObj *save, __global
const FObject *objs, __global const FLightSource
*light, __local const FSceneData* SceneData, __global
const FGPUMeshObject* Meshes, __global const
Vertex* Vertices, __global const FFace* Faces,
__global const Normal* Normals)
{
        float i_s[2];
        FColor color = mult_color(SceneData-
>AmbientLightColor, SceneData-
>AmbientLightIntensity);
        FColor spec_col = (FColor)(0, 0, 0, 0);

        i_s[0] = 0.0;
        i_s[1] = 0.0;
        save->normal = choose_normal(save,
Meshes);

        for (int i = 0; i < SceneData-
>NumLightSources; ++i)
        {
                const float shadow =
get_shadow(CreateDirectionalRay2(save-
>intersection_point, light->Origin), objs, save, Meshes,
Vertices, Faces, Normals);
                if (shadow > 0)
                {
```

```
                    i_s[0] = diffuse(save-
>intersection_point, save->normal, light) * shadow;
                    i_s[1] = specular(save-
>normal, save->ray.ray, light, save) * shadow;
                    color = mix_colors(color,
mult_color(light->Color, i_s[0]));
                    spec_col =
mix_colors(spec_col, mult_color(light->Color, i_s[1]));
            }
            light++;
        }
        return (mult_colors(choose_color(save,
Meshes, Vertices, Faces, Normals), color, spec_col));
}

float get_shadow(const FDirectionalRay ray, __global
const FObject *obj, FSavedObj *save, __global const
FGPUMeshObject* Meshes, __global const Vertex*
Vertices, __global const FFace* Faces, __global const
Normal* Normals)
{
        float distance;

        while (obj->type)
        {
            if (save->obj != obj &&
choose_intersect(obj, &ray, save, &distance, Meshes,
Vertices, Faces, Normals) && (1.f - distance >
EPSILON) && distance > EPSILON)
                    return (1);
            obj++;
        }
        return (1);
}

inline float diffuse(const float4 intersection_point,
const float4 normal, __global const FLightSource
*light)
{
        const float Intensity = ray_cos(normal, light-
>Origin - intersection_point);
        return select( light->Intensity * Intensity, 0.f,
Intensity <= 0.f );
}

inline float specular(const float4 normal, const float4
ray_direction, __global const FLightSource *light,
const FSavedObj *save)
{
        if (save->obj->specularity < 0.f)
            return (0.f);
        const float4 light_vector = light->Origin -
save->intersection_point;
        const float4 mid_vector = (normal * (2.f *
dot(normal, light_vector))) - light_vector;
        const float res = ray_cos((ray_direction * -
1.f), mid_vector);

        return select(light->Intensity * pown(res,
save->obj->specularity), 0.f, res <= 0.f);
}

/*
** Object choosers
*/

inline int choose_intersect(__global const FObject
*obj, const FDirectionalRay *ray, FSavedObj* save,
float *t, __global const FGPUMeshObject* Meshes,
__global const Vertex* Vertices, __global const FFace*
Faces, __global const Normal* Normals)
{
        switch (obj->type)
        {
            case SPHERE:
                    return
sphere_intersection(obj, ray, t);
                    break;
            case PLANE:
                    return
plane_intersection(obj, ray, t);
                    break;
            case MESH:
                    return
mesh_intersection(obj, ray, save, t, Meshes, Vertices,
Faces, Normals);
                    break;
            default: return 0;
        };
}


inline float4 choose_normal(FSavedObj *save,
__global const FGPUMeshObject* Meshes)
{
        switch (save->obj->type)
        {
            case SPHERE:
                    return sphere_normal(save);
                    break;
            case PLANE:
                    return plane_normal(save);
                    break;
            case MESH:
                    return mesh_normal(save);
                    break;
            default: return 0.f;
        };
}

inline FColor choose_color(FSavedObj const *const
save, __global const FGPUMeshObject* Meshes,
__global const Vertex* Vertices, __global const FFace*
Faces, __global const Normal* Normals)
{
        if (!save || !save->obj)
            return ((FColor)(0, 0, 0, 9));

        switch (save->obj->type)
        {
            case SPHERE:
                    return sphere_color(save);
                    break;
            case PLANE:
                    return plane_color(save);
                    break;
```

```
			case MESH:
				return mesh_color(save);
				break;
			default: return ((FColor)(0, 0, 0, 0));
		};

}

/*
** Intersection
*/

inline float4 view3d(__local const FSceneData*
SceneData, float y, float x)
{
		float4 vector;

		vector.x = (-SceneData->WindowWidth / 2 +
x) * SceneData->RelationX;
		vector.y = (SceneData->WindowHeight / 2 -
y) * SceneData->RelationY;
		vector.z = SceneData-
>ScreenRelativeDistance;
		vector = rotate_vector(vector, SceneData-
>Camera.Rotation);
		vector = vector + SceneData->Camera.Origin;
		return vector;
}

inline int sq_equation(float4 k, float *res)
{
		float t[2];

		float discriminant = pow(k.y, 2) - 4 * k.x * k.z;
		if (discriminant < 0)
			return (0);
		else if (discriminant > 0)
			discriminant = sqrt(discriminant);
		t[0] = (-k.y - discriminant) / k.x / 2;
		t[1] = (-k.y + discriminant) / k.x / 2;
		if (t[0] < 0 && t[1] < 0)
			return (0);
		*res = (t[1] - t[0] > EPSILON) ? t[0] : t[1];
		float tmp = (t[0] - t[1] > EPSILON) ? t[0] :
t[1];
		if ((tmp - *res) > EPSILON && *res < 0)
			*res = tmp;
		return (1);
}

inline int plane_intersection(__global const FObject
*obj, const FDirectionalRay *ray, float *t)
{
		const float d_v = dot(ray->ray, obj->normal);

		if (d_v == 0)
			return (0);
		float x_v = dot(obj->center - ray->start, obj-
>normal);
		if ((d_v < 0 ? -1 : 1) != (x_v < 0 ? -1 : 1))
			return (0);
		*t = x_v / d_v;
		return (1);
}

inline float4 plane_normal(FSavedObj const *const
save)
{
		if (dot(save->ray.ray, save->obj->normal) > 0)
			return save->obj->normal * -1;
		return (save->obj->normal);
}

inline FColor plane_color(FSavedObj const *const
save)
{
		return save && save->obj ? save->obj->color
: (FColor)(0, 0, 0, 0);
}

/* MESH start */

inline int IntersectBoundingBox(float4 min, float4
max, const FDirectionalRay *ray)
{
   float t1 = (min.x - ray->start.x) / ray->ray.x;
   float t2 = (max.x - ray->start.x) / ray->ray.x;
   float tmin = fmin(t1, t2);
   float tmax = fmax(t1, t2);

   t1 = (min.y - ray->start.y) / ray->ray.y;
   t2 = (max.y - ray->start.y) / ray->ray.y;
   tmin = fmax(tmin, fmin(t1, t2));
   tmax = fmin(tmax, fmax(t1, t2));

   t1 = (min.z - ray->start.z) / ray->ray.z;
   t2 = (max.z - ray->start.z) / ray->ray.z;
   tmin = fmax(tmin, fmin(t1, t2));
   tmax = fmin(tmax, fmax(t1, t2));

   return tmax > fmax(tmin, 0.f);
}

float RayTriangleIntersect(float4 orig, float4 dir, float4
v0, float4 v1, float4 v2, float *u, float *v)
{
   const float4 edge1 = v1 - v0;
   const float4 edge2 = v2 - v0;

   const float4 h = cross(dir, edge2);

   const float a = dot(edge1, h);
   if (fabs(a) < EPSILON)
			return INFINITY;

   const float f = 1.0f / a;
   const float4 s = orig - v0;

   *u = f * dot(s, h);
   if (*u < 0.0 || *u > 1.0)
			return INFINITY;

   const float4 q = cross(s, edge1);

   *v = f * dot(dir, q);
   if (*v < 0.0 || (*u + *v) > 1.0)
```

```
                return INFINITY;

   return f * dot(edge2, q);
}

int mesh_intersection(__global const FObject *obj,
const FDirectionalRay *ray, FSavedObj* save, float *t,
__global const FGPUMeshObject* Meshes, __global
const Vertex* Vertices, __global const FFace* Faces,
__global const Normal* Normals)
{
   const int meshIndex = obj->Idx;
   __global const FGPUMeshObject* Mesh =
&Meshes[meshIndex];
   float tNear = INFINITY;
   int hit = 0;

        if (!IntersectBoundingBox(Mesh-
>BoundingBox[0] + obj->center, Mesh-
>BoundingBox[1] + obj->center, ray))
        {
                return 0;
        }

        float4 normal;
        Faces += Mesh->StartingFaceIdx;
   for (int i = Mesh->StartingFaceIdx; i < Mesh-
>StartingFaceIdx + Mesh->NumFaces; i++)
        {
      float u, v;
                const float4 VertexA =
Vertices[Faces->Vertices.s0 + Mesh-
>StartingVertexIdx] + obj->center;
                const float4 VertexB =
Vertices[Faces->Vertices.s1 + Mesh-
>StartingVertexIdx] + obj->center;
                const float4 VertexC =
Vertices[Faces->Vertices.s2 + Mesh-
>StartingVertexIdx] + obj->center;

                const float res =
RayTriangleIntersect(ray->start, ray->ray, VertexA,
VertexB, VertexC, &u, &v);
                if (res < INFINITY && res < tNear)
                {
         tNear = res;
                        if (Faces->Normals.s0 == -1
|| Faces->Normals.s1 == -1 || Faces->Normals.s2 == -1)
                        {
                                normal = Faces-
>PreComputedNormal;
                        }
                        else
                        {
                                const float4
NormalA = Normals[Faces->Normals.s0 + Mesh-
>StartingNormalIdx];
                                const float4
NormalB = Normals[Faces->Normals.s1 + Mesh-
>StartingNormalIdx];
                                const float4
NormalC = Normals[Faces->Normals.s2 + Mesh-
>StartingNormalIdx];
                                normal =
normalize(NormalA * (1 - u - v) + NormalB * u +
NormalC * v);
                        }
         hit = 1;
      }

                        Faces++;
   }

   if (hit)
        {
      *t = tNear;
                        save->normal = normal;
   }
   return hit;
}

float4 mesh_normal(FSavedObj const *const save)
{
   return save->normal;
}

FColor mesh_color(FSavedObj const *const save)
{
   if (save && save->obj)
        {
      return save->obj->color;
   }
   return (FColor)(0, 0, 0, 0);
}

/* MESH end */

inline int sphere_intersection(__global const FObject
*obj, const FDirectionalRay *ray, float *t)
{
        const float4 CameraToCenterVector = ray-
>start - obj->center;
        float4 k = (float4)(ray->len, 2.f * dot(ray->ray,
CameraToCenterVector), dot(CameraToCenterVector,
CameraToCenterVector) - obj->pow_radius, 0.f);

        return select(1.f, 0.f, !sq_equation(k, t));
}

inline float4 sphere_normal(FSavedObj const *const
save)
{
        return (normalize(save->intersection_point -
save->obj->center));
}

inline FColor sphere_color(FSavedObj const *const
save)
{
        if (save && save->obj)
                return (save->obj->color);
        return ((FColor)(0, 0, 0, 0));
}

int intersect_point(const FDirectionalRay *ray,
__global const FObject *obj, FSavedObj *save,
```

```
__global const FObject *ignore, __local const
FSceneData* SceneData, __global const
FGPUMeshObject* Meshes, __global const Vertex*
Vertices, __global const FFace* Faces, __global const
Normal* Normals)
{
	float d = 0;

	save->distance = INFINITY;
	save->obj = NULL;
	for (int i = 0; i < SceneData->NumPrimitives;
++i)
	{
		if (obj != ignore)
		{
			if (choose_intersect(obj, ray,
save, &d, Meshes, Vertices, Faces, Normals) && d > 0
&& (save->distance - d > EPSILON))
			{
				save->distance = d;
				save->obj = obj;
			}
		}
		obj++;
	}
	if (save->obj == NULL)
		return (0);
	save->intersection_point = ray->start + (ray-
>ray * save->distance);
	return (1);
}

/*
* Tracing ray
*/

inline FDirectionalRay reflect_ray(float4 ray,
FSavedObj *save)
{
	FDirectionalRay line;

	ray = ray * -1;
	line.start = save->intersection_point;
	line.ray = (save->normal * (2 * dot(save-
>normal, ray))) - ray;
	line.len = dot(line.ray, line.ray);
	return (line);
}

FColor reflection(FSavedObj *save, __local const
FSceneData* SceneData, __global const FObject
*objs, __global const FLightSource *lights, int rec, int
max_rec, __global const FGPUMeshObject* Meshes,
__global const Vertex* Vertices, __global const FFace*
Faces, __global const Normal* Normals)
{
	FColor color = (FColor)(0, 0, 0, 0);
	__global const FObject *ignore = save->obj;
	FDirectionalRay line = reflect_ray(save-
>ray.ray, save);

	save->ray = line;
	if (intersect_point(&line, objs, save, ignore,
SceneData, Meshes, Vertices, Faces, Normals))
	{
		color = get_pixel_color(save, objs,
lights, SceneData, Meshes, Vertices, Faces, Normals);
		if (rec == max_rec && save->obj-
>reflection)
			color = mult_color(color, 1 -
save->obj->reflection);
	}
	return (color);
}

FColor sample(__global const FObject *objs,
	__global const FLightSource *lights,
	__local const FSceneData* SceneData,
	__global const FGPUMeshObject* Meshes,
	__global const Vertex* Vertices,
	__global const FFace* Faces,
	__global const Normal* Normals,
	float y,
	float x)
{
	FDirectionalRay line =
CreateDirectionalRay(SceneData->Camera.Origin,
view3d(SceneData, y, x));

	FSavedObj save;
	FColor color = (FColor)(0, 0, 0, 0);

	save.ray = line;
	if (intersect_point(&line, objs, &save, NULL,
SceneData, Meshes, Vertices, Faces, Normals))
	{
		color = get_pixel_color(&save, objs,
lights, SceneData, Meshes, Vertices, Faces, Normals);
		if (SceneData->RecursionLevel > 0
&& save.obj && save.obj->reflection)
		{
			FColor	color2;
			FColor	color_save =
save.obj->color;
			int	rec = 1;
			float	reflection_coef =
save.obj->reflection;

			while (1)
			{
				color2 =
reflection(&save, SceneData, objs, lights, rec,
SceneData->RecursionLevel, Meshes, Vertices, Faces,
Normals);
				color2 =
ClampRGB(color2, color_save);
				color_save =
color2;
				color =
MultiplicateLightSources(color, color2, 1,
reflection_coef);
				if (rec >=
SceneData->RecursionLevel || !save.obj || !save.obj-
>reflection)
					break ;
```

```
                                        reflection_coef *=
save.obj->reflection;
                                        ++rec;
                                }
                        }


                }
                return (color);
}

__kernel void render(__global /*__write_only*/ uint
*OutputPixels,
                __global const FObject* Primitives,
                __global const FLightSource* LightSources,
                __global const FSceneData* SceneData,
                __global const FGPUMeshObject* Meshes, /*
Has info with offsets of his own faces, normans and
vertices */
                __global const Vertex* Vertices,
                __global const FFace* Faces,
                __global const Normal* Normals)
{
                const int i = get_global_id(0);
                __local FSceneData LocalScene;
                LocalScene = *SceneData;


                float y = i / LocalScene.WindowHeight;
                float x = i % LocalScene.WindowWidth;
                FColor OutputColor = sample(Primitives,
LightSources, &LocalScene, Meshes, Vertices, Faces,
Normals, y, x);
                OutputPixels[i] = rgb_to_int(OutputColor);
}
```