МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД «УНІВЕРСИТЕТ «КРОК»»
Фаховий коледж Університету «КРОК»

ДИПЛОМНА РОБОТА
За темою

«Розробка власного трьох-вимірного ігрового рушія»

Студент 4 курсу групи ІПЗ-20к-1

Южда Богдан Олексійович
(прізвище, ім'я та бо батькові)

Керівник дипломної роботи

викладач
(посада керівника)

Чернозубкін Ігор Олександрович
(прізвище, ім'я та бо батькові керівника)

До захисту .
(резолюція «До захисту»)

(підпис студента)

10.06.24
(дата)

(підпис викладача)

Київ, 2024 рік

Скорочення

API (Application Programming Interface) - це набір правил та вказівок, які визначають спосіб взаємодії між різними програмами. API визначає, які запити можуть бути зроблені до програми, які дані можна отримати відповідь, і як самс ці дані будуть структуровані. API може бути використаний для реалізації функціональності програми, взаємодії зі зовнішніми сервісами або обміну даними між програмами.

ООП (Об'єктно-орієнтоване програмування) - це парадигма програмування, яка розглядає програму як множину об'єктів, що взаємодіють між собою. Основу ООП складають чотири основні концепції: інкапсуляція, успадкування, поліморфізм та абстракція.

Зміст

Вступ

Розвиток ігрової індустрії супроводжується зростанням потреб у гнучких ігрових рушіях. В умовах загального ринку, де переважають відомі ігрові двигуни, такі як Unity, Unreal Engine та Godot, виникає питання: нащо створювати власний ігровий рушій?

1. Унікальність проектів: Завдяки власному рушію розробник зможе створити ігру з унікальним геймплеєм та стилем, який відрізняється від загальновідомих проектів, що працюють на популярних рушіях.
2. Повний контроль над процесом розробки: Використання власного ігрового рушія дозволяє розробнику повністю контролювати аспекти гри такі як: графіку, фізику, штучний інтелект, звук і т.п.
3. Оптимізація під конкретні вимоги: Відомі рушії часто мають великий функціонал, який не завжди необхідний для конкретного проекту. Власний ігровий рушій має бути оптимізований під конкретні потреби, що може позитивно позначитися на продуктивності та унікальності гри.
4. Власні технологічні рішення: Розробник має можливість впроваджувати власні технології покращення.
5. Заощадження коштів: У довгостроковій перспективі створення власного ігрового рушія може бути більш економічно вигідним, оскільки розробнику не доведеться платити за ліцензування від великих ігрових компаній.

Сфера розробки ігрових рушіїв значною мірою орієнтована на створення продуктів, здатних видавати графіку високої якості. Це призводить до значних вимог потужності як комп'ютерів гравців, так і комп'ютерів розробників. Проте, існує чимала аудиторія гравців та розробників, які віддають перевагу класичним іграм з ретро-естетикою.

Розробка власного ігрового рушія, який поєднує в собі сучасні технології та ретро-орієнтованість, має свою цінність та набір складних задач.

Одним із головних завдань при розробці рушія буде оптимізація рендедеру, таким чином щоб забезпечити високу продуктивність для гравців як зі слабкими комп'ютерами, так і для гравців з більш потужними та дати велику гнучкість для розробників.

1.1 Існуючі технології рендеру

Відкладений рендер (Deferred Rendering) - це техніка рендеру в 3D графіці, яка використовується для покращення продуктивності та візуальної якості зображення. На відміну від традиційного рендеру, де освітлення та затінення обчислюються для кожного пікселя на сцені, відкладений рендер розбиває процес на два етапи:

1. Геометричний етап:

- На цьому етапі рендеряться лише геометричні дані сцени, такі як положення, нормалі та текстурні координати вершин.
- Кольори пікселів не розраховуються на цьому етапі.
- Інформація про глибину та нормалі записуються в буфери G-buffer та нормалей.

2. Етап освітлення:

- На цьому етапі інформація з G-buffer та буфера нормалей використовується для розрахунку освітлення та затінення кожного пікселя.
- Це дозволяє використовувати більш складні моделі освітлення та затінення без значного впливу на продуктивність.
- Результат записується в остаточний буфер кольорів.

Переваги відкладеного рендеру:

- **Покращена продуктивність:** Відкладений рендер може значно покращити продуктивність, оскільки освітлення та затінення обчислюються лише один раз для кожного пікселя, а не для кожного джерела світла.
- **Покращена якість зображення:** Відкладений рендер дозволяє використовувати більш складні моделі освітлення та затінення, що може призвести до більш реалістичного та результату.
- **Гнучкість:** Відкладений рендер дає розробникам більше гнучкості при створенні візуальних ефектів.

Недоліки відкладеного рендеру:

- **Складність реалізації:** Відкладений рендер може бути складним для реалізації, оскільки потребує додаткових буферів та шейдерів.
- **Не підходить для прозорих об'єктів:** Відкладений рендер не дуже добре підходить для прозорих об'єктів, оскільки вони потребують сортування та рендеру в окремому проході.

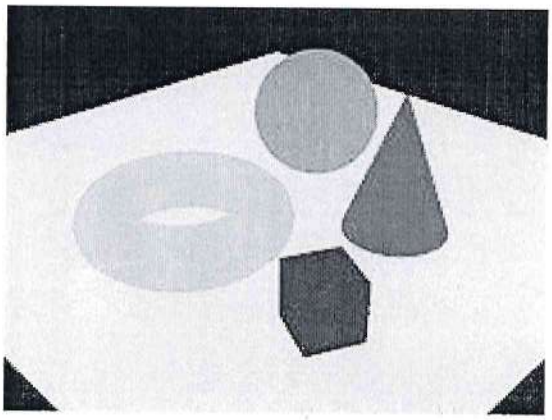

Рис 1.1 Дифузний кольоровий буфер

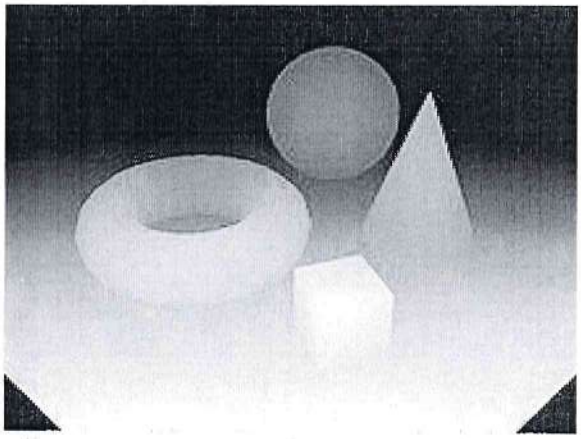

Рис 1.2 Буфер глибини

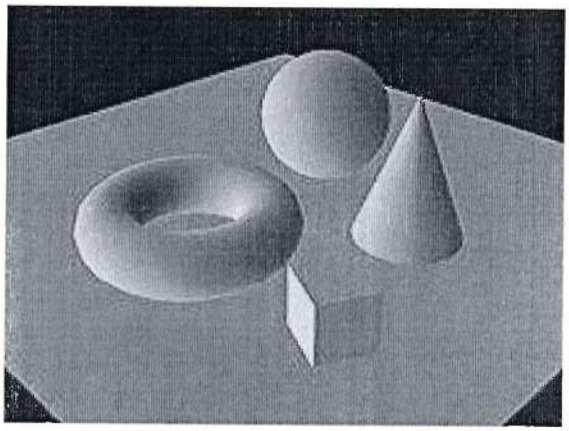

Рис 1.3 Буфер нормалі поверхні

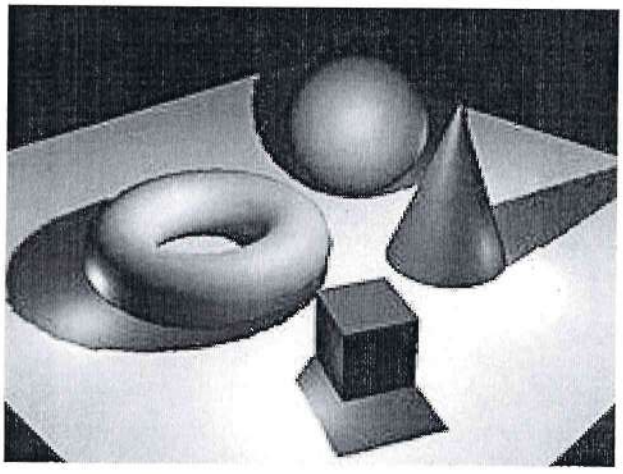

Рис 1.3 Буфер результату освітлення

Іншою ж технологією є прямий рендер (Forward Rendering) - це традиційна техніка рендеру в 3D графіці, яка полягає в послідовному рендеруванні всіх об'єктів сцени, з розрахунком освітлення та затінення для кожного пікселя на кожному об'єкті. Цей метод є простим у реалізації та добре підходить для проектів з обмеженими ресурсами.

Процес прямого рендеру:

1. **Перевірка видимості:** Визначаються об'єкти, які потрапляють у поле зору камери.
2. **Сортування:** Об'єкти сортуються за глибиною таким чином, щоб об'єкти рендерилися перед дальніми.
3. **Геометричний рендеринг:** Для кожного об'єкта рендеруються його вершини, створюючи полігональну сітку.
4. **Растеризація:** Полігональна сітка перетворюється на пікселі на екрані.
5. **Освітлення та затінення:** Для кожного пікселя розраховується його колір, беручи до уваги освітлення, матеріали та інші фактори.

Переваги прямого рендеру:

- **Простота:** Прямий рендер є простим у реалізації та добре підходить для початківців.
- **Швидкість:** Для простих сцен прямий рендер може бути швидшим за відкладений рендеринг.
- **Прозорість:** Прямий рендер добре підходить для рендеру прозорих об'єктів.

Недоліки прямого рендеру:

- **Низька продуктивність:** Для складних сцен з великою кількістю джерел світла прямий рендер може бути неефективним.
- **Складність реалізації складних ефектів:** Реалізація складних ефектів освітлення та затінення може бути складною в рамках прямого рендеру.

Одним із ключових факторів, що впливають на продуктивність рендерингу, є рендерування зайвих пікселів. Це трапляється, коли один і той самий піксель на екрані рендерується кілька разів, наприклад, коли один об'єкт перекриває інший.

Для запобігання цьому можна використовувати ранній прохід глибини (depth prepass). Цей метод полягає в тому, що перед рендерингом кольорів та освітлення для всіх об'єктів сцени, спершу рендерується лише їх глибина.

Інформація про глибину зберігається в спеціальному буфері, який потім використовується для визначення того, які пікселі потрібно рендери. Пікселі, які перекриваються іншим пікселем з більшою глибиною, просто ігноруються.

Переваги проходу глибини:

1. Зменшення часу рендеру: Завдяки тому, що рендеряться лише ті пікселі, які буде видно, значно зменшується загальний обсяг складних обчислень.
2. Підвищення продуктивності: Це може призвести до значного покращення продуктивності, особливо для сцен з великою кількістю об'єктів.
3. Економія ресурсів: Зменшення рендеру зайвих пікселів економить ресурси графічного процесора та пам'яті.

Недоліки проходу глибини:

1. Додатковий прохід рендеру: Прохід глибини додає ще один прохід рендеру до процесу, що може трохи збільшити час рендеру.
2. Складність реалізації: Реалізація проходу глибини може бути трохи складнішою, ніж прямий рендер.

Табл. 1.1 Порівняння технологій рендеру

	Прямий рендер	Прямий рендер з глибиною	Відкладений рендер
Кількість графічних проходів на об'єкт	1	2	1
Кількість графічних проходів на джерело світла	1	1	1 на кожне джерело світла
Підтримка прозорості	так	так	ні

У сучасних іграх часто використовується поєднання прямого та відкладеного рендеру, щоб отримати найкращі результати як з точки зору продуктивності, так і візуальної якості.

Переваги поєднання:

1. Покращена продуктивність: Прямий рендер використовується для освітлення та відображень на об'єктах, що може бути значно швидшим, ніж відкладений рендер для цих завдань.

2. Підвищена якість зображення: Відкладений рендер використовується для ефектів постпроцесингу, таких як HDR, розмиття та глибина різкості, що може значно покращити візуальну якість зображення.
3. Гнучкість: Цей підхід дає розробникам більше гнучкості при створенні візуальних ефектів.

Приклад:

Гра Doom (2016) використовує поєднання прямого та відкладеного рендеру. Прямий рендер з проходом глибини використовується для освітлення та відображень на об'єктах, а відкладений рендер використовується для таких ефектів постпроцесингу, як HDR, розмиття та глибина різкості. Цей підхід дозволяє Doom досягти високої продуктивності та візуальної якості.

1.2 Обраний підхід до рендеру та інші необхідні технології

Після ретельного аналізу різних методів рендеру, для нашого ігрового рушія було обрано комбінований підхід, що поєднує в собі прямий рендер та відкладений рендер.

Прямий рендер буде використовуватися для первинного рендерингу сцени, тобто для промальовування геометрії та розрахунку базового освітлення. Цей метод простий у реалізації та добре підходить для проектів з обмеженими ресурсами.

Відкладений рендер буде використовуватися для пост-процесингу, тобто для додавання складних ефектів освітлення, корекції кольорів, ефект bloom, згладжування та інших візуальних ефектів. Цей метод дозволяє отримати високу якість зображення при збереженні задовільної продуктивності.

Переваги комбінованого підходу:

1 Баланс продуктивності та якості: Завдяки використанню прямого рендеру для первинного рендеру та відкладеного рендеру для пост-процесингу, можна досягти оптимального балансу між продуктивністю та якістю зображення.
2 Гнучкість: Цей підхід дає розробникам більше гнучкості при створенні візуальних ефектів.
3 Простота реалізації: Прямий рендер та відкладений рендер є добре вивченими та відносно простими у реалізації методами.

Недоліки комбінованого підходу:

1 Складність архітектури: Комбінування двох методів рендеру може трохи ускладнити архітектуру ігрового рушія.
2 Необхідність додаткових ресурсів: Відкладений рендер потребує додаткових ресурсів, таких як буфери кадру та шейдери.

Як графічний фреймворк було обрано Monogame framework 3.8. Він є оновленою версією XNA framework, котрий був розроблений компанією Microsoft ще у 2004 році. Обраний він був оскільки ігровий рушій використовує мову програмування C# та націлений на підтримку декількох платформ з мінімальними змінами основного коду. Monogame framework підтримує такі графічні API як OpenGL, DirectX 9 та DirectX 11. Проте через відсутність підтримки шейдерної моделі версії 5 на OpenGL через обмеження компілятора шейдерів MojoShader деякі графічні ефекти будуть недоступні у версії рушія для платформ, які не підтримують DirectX. Тобто на усіх платформах окрім Windows та Xbox. Але оскільки ці платформи є домінуючими, то різницю відчує невелика кількість гравців. Також OpenGL не підтримує команди викликані з різних потоків процесора. Це призводить до того, що гра зависає коли є потреба завантажити нові графічні ресурси по типу вертексного буфера або текстури. Також це унеможливлює асинхронну презентацію кадру, що в результаті призводить до загально гіршої продуктивності.

Для основної системи анімації було обрано скелетну анімацію.

Скелетна анімація – це найбільш розповсюджений метод анімації, що знаходить застосування як у відеоіграх, так і загалом в усій графічній області обчислень. Ця технологія базується на створенні віртуального скелета, складеного з ієрархічно з'єднаних кісток, які відповідають основним частинам моделі та, частіше за все, частинам тіла . Ця структура дозволяє реалістично анімувати або симулювати рухи та деформації моделі.

Скелетна анімація є індустрійним стандартом, що робить її доступною для більшості аніматорів. ЇЇ гнічкість дозволяє створювати різноманітних рухів та деформацій. Від простих жестів до складних акробатичних трюків. Ефективність та універсальність роблять її відмінним вибором для проектів з обмеженими ресурсами, а також забезпечують сумісність з багатьма 3D-редакторами.

Проте, під час анімації деяких моделей можуть виникати проблеми з перетинанням, коли одна частина моделі проходить через іншу. Також анімація не завжди може точно передати деформації розтягування я м'язів або м'яких тканин.

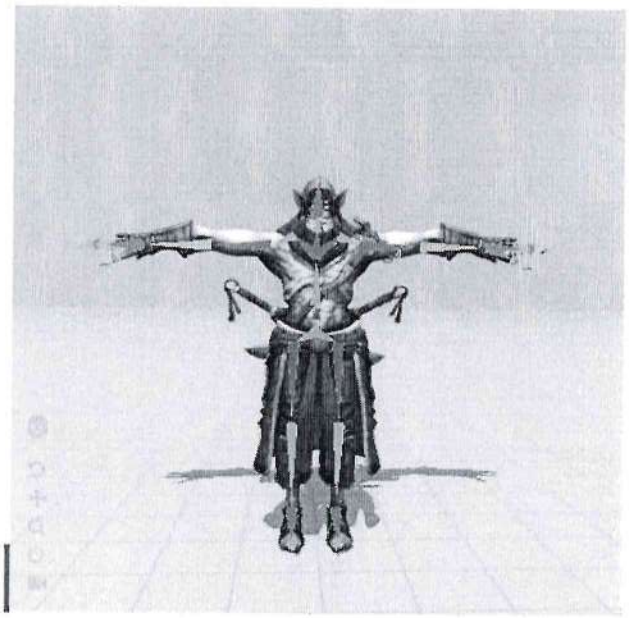

Рис 1.4 3D-модель зі скелетом

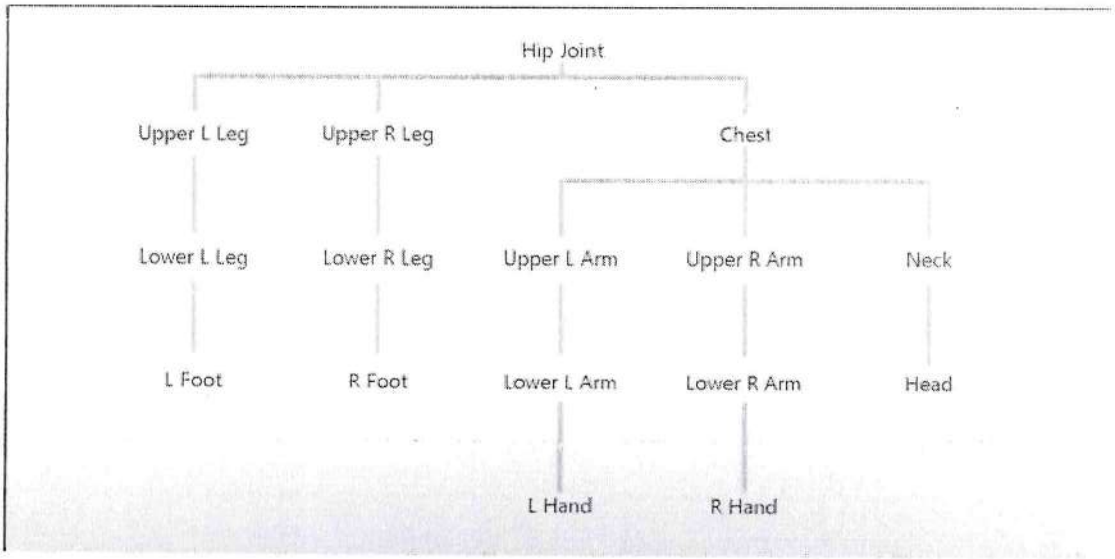

Рис 1.5 Ієрархія скелету 3D-моделі

В якості редактора рівнів для ігрового рушія було обрано використовувати вже існуючий редактор Trenchbroom. Цей редактор є широко відомим у спільноті левел-дизайнерів, оскільки він був розроблений для створення рівнів для таких класичних ігор, як Quake 1, Quake 2, Quake 3 та Half-Life 1.

Основними перевагами Trenchbroom є:

1 Гнучкість: Trenchbroom пропонує широкий спектр інструментів та функцій для створення складних та деталізованих рівнів. Він дозволяє користувачам точно розміщувати об'єкти, створювати складні геометричні форми, налаштовувати освітлення та текстури, а також писати власні сценарії для розширення функціональності.

2 Простота використання: Окрім своєї гнучкості, Trenchbroom має інтуїтивно зрозумілий інтерфейс та логічну структуру, що робить його доступним для користувачів з різним рівнем досвіду.

3 Сумісність: Trenchbroom підтримує широкий спектр форматів файлів карт, що робить його сумісним з багатьма ігровими рушіями, як старими, так і сучасними.

4 Спільнота: Завдяки своїй популярності Trenchbroom має активну спільноту користувачів, які роблять навчальні матеріали та дають корисні поради по створенню рівнів та елементів оточення загалом.

Хоча Trenchbroom спочатку був розроблений для редактування та створення рівнів для Quake-подібних ігор, його гнучкість дозволяє використовувати його і з іншими ігровими рушіями, включаючи власний.

Також важливою частиною ігрового рушія є система навігації. Існують готові алгоритми на які можна орієнтуватись при створені власної системи.

Алгоритм A* - це евристичний алгоритм пошуку шляху, який використовується для знаходження найкоротшого шляху між двома точками у світі гри. Він широко використовується в ігрових рушіях для пошуку шляху.

A* ґрунтується на двох основних принципах:

1. Інформація про пройдений шлях: Алгоритм враховує не лише відстань від поточної точки до цільової, але й довжину вже пройденого шляху. Це дозволяє йому зосередитися на більш перспективних маршрутах і уникати тупиків.
2. Евристична оцінка: A* використовує евристичну функцію, щоб оцінити відстань від поточної вершини до цільової. Ця функція дає алгоритму уявлення про те, наскільки близько він знаходиться до мети, не досліджуючи всі можливі шляхи.

Переваги використання A* в ігрових рушіях:

- Реалістичний рух: A* може допомогти створити більш реалістичний рух персонажів та NPC, знаходячи для них найкоротші та найефективніші маршрути, які враховують особливості навколишнього середовища.
- Динамічні світи: A* підходить для динамічних світів, де карта може змінюватися під час гри. Алгоритм може адаптуватися до цих змін і

знаходити нові маршрути, не втрачаючи часу на повторне дослідження вже неактуальних шляхів.

- Покращення продуктивності: Завдяки евристичній оцінці та фокусу на перспективних маршрутах A* може знаходити найкоротші шляхи швидше, ніж інші алгоритми пошуку, економлячи обчислювальні ресурси та покращуючи загальну продуктивність гри.
- Гнучкість: A* можна легко модифікувати для вирішення різних задач пошуку шляху. Наприклад, його можна налаштувати, щоб:
    - Уникати певних частин карти, наприклад, небезпечних зон.

Враховувати витрати на переміщення по різних типах місцевості, наприклад, рухатися швидше по дорогах, ніж по бездоріжжю, або воді.

Система навігаційної сітки (Navigation Mesh, NavMesh) - це поширена технологія, що використовується в ігрових рушіях для представлення та пошуку шляхів у віртуальному середовищі. Вона базується на поділі ігрового простору на сітку з'єднаних між собою вершин. Сітка утворює з себе модель, яка представляє собою ділянку світу доступну для переміщення

Принцип роботи:

- Створення сітки: На початковому етапі створюється сітка навігації, яка покриває ігровий світ. Ця сітка складається з вузлів (зазвичай це точки на карті) та ребер (з'єднань між вузлами). Вузли розміщуються в місцях, де дозволено пересування, наприклад, на рівній землі, на платформах або вздовж стін. Ребра з'єднують вузли, які можна пройти без перешкод. Це складний процес, котрий повинен проводитись на етапі запікання карти при експорті з редактору.
- Обчислення шляхів: Коли персонажу або об'єкту в грі потрібно переміститися з однієї точки в іншу, система навігаційної сітки використовується для пошуку найкращого маршруту. Для цього алгоритм

пошуку шляху досліджує сітку, переходячи від одного вузла до іншого, поки не знайде шлях до цільового вузла.

Переваги використання системи навігаційної сітки:

- Ефективність: Система навігаційної сітки може знаходити шляхи швидко та ефективно, що робить її ідеальною для використання в динамічних ігрових середовищах.
- Гнучкість: Сітка може доволі легко репрезентувати різні типи та форми оточення та ландшафту.

Мінуси системи навігаційної сітки:

- Складність створення: Створення та налаштування сітки навігації може бути складним та трудомістким завданням, особливо для великих і складних ігрових світів.
- Негнучкість: Після створення сітка навігації стає досить жорсткою. Змінювати її в реальному часі під час гри може займати забагато часу та ресурсів, що може призвести до нереалістичних маршрутів у деяких випадках.
- Неточність: Сітка навігації може неточно відображати геометрію оточення у світі гри, що може призвести до нелогічних маршрутів, особливо на нерівній місцевості або в місцях з вузькими проходами.
- Обчислювальні ресурси: Створення та оновлення сітки навігації може потребувати значних обчислювальних ресурсів, що може вплинути на продуктивність гри, особливо на малопотужних пристроях.

Система нодових графів (Node Graph) - це метод пошуку шляху в ігрових рушіях, який використовується в таких популярних іграх, як Quake, Half-Life та Counter-Strike. Цей метод пропонує простоту реалізації та гнучкість, роблячи його привабливим вибором для розробників.

Принцип роботи:

- Створення графу: На початковому етапі створюється граф, який представляє доступні для пересування ділянки ігрового світу. Граф складається з вузлів (зазвичай це точки на карті) та ребер, які з'єднують ці вузли. Вузли розміщуються в місцях, де дозволено пересування, наприклад, на рівній землі, на платформах або вздовж стін. Ребра з'єднують вузли, які можна пройти без перешкод.
- Обчислення шляхів: Коли персонажу або об'єкту в грі потрібно переміститися з однієї точки в іншу, система нодових графів використовується для пошуку найкращого маршруту. Для цього алгоритм пошуку шляху досліджує граф, переходячи від одного вузла до іншого, поки не знайде шлях до цільового вузла.
- Переміщення по маршруту: Після того як шлях було знайдено персонаж може рухатися переходячи від одного вузла до наступного.

Переваги використання системи нодових графів:

- Простота реалізації: Система нодових графів простіша у реалізації та налаштуванні порівняно з системою навігаційної сітки. Це робить її привабливим вибором для розробників з різним рівнем досвіду.
- Гнучкість: Граф можна легко модифікувати, щоб врахувати особливості ігрового світу, такі як перешкоди, небезпечні зони або різні типи місцевості.
- Ефективність: Алгоритми пошуку шляху, які використовуються з нодовими графами, можуть бути досить ефективними, особливо для невеликих і середніх ігрових світів.

Мінуси системи нодових графів:

- Складність масштабування: Система нодових графів може стати складною для масштабування в дуже великих ігрових світах. Створювати та

оновлювати великий граф може бути трудомістким завданням, а пошук шляхів у ньому може потребувати значних обчислювальних ресурсів.

- Неточність: Граф може неточно відображати геометрію ігрового світу, що може призвести до незграбних або нелогічних маршрутів, особливо на нерівній місцевості або в місцях з вузькими проходами.
- Відсутність динамічного оновлення: Система нодових графів не завжди може динамічно оновлюватися під час гри, що може призвести до неідеальних або нереалістичних маршрутів у деяких ситуаціях.

Фізичний рушій є невід'ємною складовою кожного ігрового рушія. Від цього його вибору залежить те як будуть працювати певні основні системи рушія. Було розглянуто такі варіанти:

Jolt Physics - це безкоштовний фізичний рушій з відкритим кодом, розроблений для ігрових двигунів та симуляцій. Він пропонує широкий спектр функцій для моделювання реалістичної фізики в іграх та інших віртуальних середовищах.

Переваги Jolt Physics:

- Безкоштовний та з відкритим кодом: Jolt Physics доступний безкоштовно для використання та модифікації, що робить його привабливим вибором для розробників з обмеженим бюджетом або тих, хто хоче налаштувати рушій під свої потреби.
- Висока продуктивність: Jolt Physics оптимізований для високої продуктивності, що робить його придатним для використання в іграх з високими вимогами до візуальних ефектів або фізичних симуляцій.
- Широкий спектр функцій: Jolt Physics пропонує широкий спектр функцій для моделювання реалістичної фізики, включаючи:
    - Динаміка жорстких тіл

  - Розпізнавання зіткнень
  - Фізика рідин
  - М'які тіла
  - Фізика тканин
  - З'єднання
- Підтримка різних платформ: Jolt Physics підтримує різні платформи, включаючи Windows, macOS, Linux, Android та iOS.

Мінуси Jolt Physics:

- Складність використання: Jolt Physics не такий простий у використанні та імплементації, як деякі інші фізичні рушії. Він потребує розуміння основ фізики та програмування.
- Відсутність документації: Документація Jolt Physics не така обширна, як у деяких інших фізичних рушіїв. Це може ускладнити розробникам вивчення та використання рушія.
- Невелике співтовариство: Співтовариство користувачів Jolt Physics не таке велике, як у інших фізичних рушіїв. Це дуже сильно ускладнює імплементацію та вирішення проблем.

Bullet Physics - це безкоштовний фізичний рушій з відкритим кодом, який широко використовується в іграх, симуляціях та інших віртуальних середовищах. Він пропонує широкий спектр функцій для моделювання реалістичної фізики, роблячи його популярним вибором як для досвідчених розробників, так і для початківців.

Переваги Bullet Physics:

- Простота використання: Bullet Physics порівняно простий у використанні та імплементації порівняно з деякими іншими фізичними рушіями. Він пропонує чіткий інтерфейс програмування на C++ та широкий спектр навчальних матеріалів та ресурсів.

- Висока продуктивність: Bullet Physics оптимізований для високої продуктивності, що робить його придатним для використання в іграх з високими вимогами до візуальних ефектів або фізичних симуляцій.
- Широкий спектр функцій: Bullet Physics пропонує широкий спектр функцій для моделювання реалістичної фізики, включаючи:
    - Динаміка жорстких тіл
    - Розпізнавання зіткнень
    - Фізика м'яких тіл
    - Фізика тканин
    - З'єднання
- Підтримка різних платформ: Bullet Physics підтримує різні платформи, включаючи Windows, macOS, Linux, Android та iOS. Також, завдяки тому, що цей фізичний рушій має відкритий код, можна реалізувати будь-яку іншу платформу маючи достатню кількість знань та навичок.
- Активне співтовариство: Bullet Physics має велике та активне співтовариство користувачів, що робить його легким для розробників отримання допомоги та підтримки.

Мінуси Bullet Physics:

- Складність деяких функцій: Деякі з більш просунутих функцій Bullet Physics, такі як фізика м'яких тіл та фізика рідин, можуть бути складними для розуміння та використання.
- Відсутність підтримки багато-потоку:
- Можливі проблеми з оптимізацією: У деяких випадках Bullet Physics може потребувати оптимізації для досягнення максимальної продуктивності, особливо в складних симуляціях.
- Відсутність деяких функцій: Bullet Physics не має деяких функцій, які є в інших фізичних рушіях, таких як детальна візуалізація фізичних симуляцій або інтегровані інструменти для анімації.

Написання власного простого фізичного рушія

Написання власного простого фізичного рушія з обмеженим функціоналом може бути складним та неефективним завданням, але воно може бути цінним досвідом. Цей процес може допомогти краще зрозуміти принципи фізики та програмування, а також дати можливість дослідити та реалізувати власні ідеї.

Переваги написання власного фізичного рушія:

- Глибоке розуміння фізики та програмування: Написання власного фізичного рушія змушує глибоко дослідити принципи фізики, такі як динаміка жорстких тіл та розпізнавання зіткнень.
- Можливість досліджувати та реалізувати власні ідеї: Написання власного фізичного рушія дає свободу досліджувати та реалізувати власні ідеї та алгоритми. З'являється можливість створити власний рушій, який відповідає конкретним потребам та вподобанням.

Недоліки написання власного фізичного рушія:

- Складність та час: Написання власного фізичного рушія може бути дуже складним та трудомістким завданням. Це вимагає дуже багато часу за зусиль.
- Неефективність: Власний фізичний рушій, ймовірно, буде менш ефективним, ніж існуючі рішення, які були оптимізовані та вдосконалені протягом багатьох років. Це може призвести до проблем з продуктивністю, особливо в складних симуляціях.

Інші рішення є ще менш привабливими, оскільки це або ще менш документовані та популярні рішення. Деякі з них можуть бути ефективними, але мають платну, або дуже обмежену ліцензію.

У процесі розробки ігрового рушія було прийнято рішення підтримувати дві звукові системи для відтворення аудіо: одну, що використовує Fmod, і іншу, побудовану на базі XAudio.

Одна з основних причин для підтримки двох звукових систем полягає в необхідності забезпечення незалежності ігрового рушія від сторонніх закритих та платних бібліотек. Залежність від стороннього програмного забезпечення може призвести до ряду проблем, таких як ліцензійні обмеження, зміни в умовах використання або навіть зняття продукту з підтримки. Використовуючи власну звукову систему на базі XAudio, розробники можуть гарантувати, що їхній ігровий рушій залишатиметься функціональним і незалежним від змін у сторонньому програмному забезпеченні.

Гнучкість та налаштування

Використання двох звукових систем дозволяє розробникам вибирати найбільш підходящий інструмент для конкретних задач. Fmod пропонує розширені можливості для роботи зі звуком, включаючи складну обробку аудіо, підтримку різних платформ і потужний API. Це робить його ідеальним вибором для складних звукових проектів, де важливі такі функції, як 3D-звук, фільтри та ефекти. Однак, для простих завдань або менших проектів може бути достатньо використовувати XAudio, що дозволяє знизити складність коду та покращити продуктивність.

Ліцензійна вартість

Fmod є платною бібліотекою, і його використання може вимагати значних фінансових вкладень, особливо для комерційних проектів. Підтримка

альтернативної звукової системи на базі XAudio дозволяє уникнути цих витрат і зосередитися на безкоштовних або відкритих рішеннях. Це особливо важливо для незалежних розробників та малих студій, які можуть не мати бюджету для придбання ліцензій на використання Fmod.

Надійність та контроль

Розробляючи власну звукову систему, розробники отримують повний контроль над її функціональністю і можуть швидко вносити зміни або виправлення без необхідності чекати оновлень від сторонніх постачальників. Це також підвищує надійність системи, оскільки всі аспекти її роботи зрозумілі та доступні для зміни. У разі виникнення проблем із сумісністю або продуктивністю, команда може робити необхідні корективи.

Таким чином, підтримка двох звукових систем у ігровому рушії дозволяє забезпечити гнучкість, контроль, незалежність та економічну ефективність, що є ключовими аспектами для успішної розробки ігор.

2.1 Логічна частина роботи ігрового рушія

Ігровий рушій розбитий на 4 основні потоки:

Потік ігрової логіки	На цьому потоці відбувається прораховування логіки пов'язаною з ігровим процесом. Наприклад: оновлення керування, фізики та геймплейних подій специфічних до гри
Потік рендеру	Потік рендеру коли потік ігрової логіки закінчить опрацювання кадру, формує фінальні дані що до моделей, їх параметрів та того як їх треба рендерити. Після цього запускається опрацювання ігрової логіки для наступного кадру і відбувається процес рендеру.

Потік завантаження графічних ресурсів	Оскільки гра може потребувати завантаження ресурсів під час процесу гри, то ці операції були перенесені на окремий потік, проте OpenGL не підтримує такого роду операцій, тому ці операції виконуються на потоці рендеру
Потік прораховування навігації та пошуку шляху	Процес прорахування шляху може займати дуже багато часу. Особливо якщо карта має багато точок навігації, або якщо відстань між двома точками є дуже великою, або якщо шлях є дуже комплексним, неочевидним і загалом потребує багато ресурсів та часу.

Рис 2.1 Схема основного циклу ігрового рушія

Навігація

Навігація була виведена в окремий потік, оскільки це дуже складний для опрацювання та прораховування алгоритм. Існує два підходи використання системи навігації: відкладений та миттєвий. Кожен з підходів має власні плюси та мінуси.

Миттєвий підхід виконує пошук шляху на потоці на якому він був виконаний. Його виконання передбачене на ігровому потоці. Це дозволяє використовувати дані фізичної сцени отримані у реальному часі. Тобто в такому випадку пошук шляху використовує дані про перешкоди отримані у реальному часі. Проте головний мінус цього підходу прямо витікає з його основної переваги. Оскільки він використовує дані фізичної сцени отримані у реальному часі цей підхід блокує виконання ігрового потоку, що спричиняє помітні скачки кількості кадрів в секунду.

Відкладений підхід виконує пошук шляху використовуючи інший паралельний потік. Коли грі треба прорахувати шлях вона може створити запит, який буде додано у чергу. Запити виконуються послідовно на паралельному потоці. Після виконання пошуку шляху результат передається назад. В результаті такого підходу ігровий потік витрачає ресурси тільки на те щоб створити та додати у чергу запит. Проте через обмеження фізичного рушія цей підхід не може використовувати дані фізичної сцени отримані у реальному часі, оскільки вони можуть бути змінені на іншому потоці. Такого роду операція може призвести до помилки та до вильоту програми. Тому замість даних отриманих у реальному часі цей підхід використовує статичну фізичну сцену. Вона формується ще на моменті завантаження карти з файлу у гру та створення світу. Статичний фізичний світ не змінюється у реальному часі у процесі гри, тому може бути використаний на декількох потоках відразу. Тобто система навігації може використовувати його для проведення операцій паралельно ігровому потоку. Це ж є основним мінус цього підходу. Не дивлячись на його швидкість він обмежений статичними даними отриманими під час завантаження карти. Це унеможливлює уникання динамічних перешкод під час процесу пошуку шляху.

Вибір конкретного алгоритму пошуку шляху це теж складне питання. Були розглянуті такі алгоритми як: A*, навігаційна сітка та нодова система(від англійської node graph).

Через свою простоту у реалізації та те, що вона була використана у багатьох класичних ігор, було обрано нодову систему(node graphs). При створені карти розробник повинен в ручну розставити точки(ноди) за якими буде відбуватись пошук шляху. Коли рушій завантажує карту з файлу у пам'ять генерується граф усіх точок навігації. Рушій робить просту перевірку на те чи жоден статичний об'єкт не перекриває видимість від однієї точки до іншої. Якщо перекриває, то тоді точки не є з'єднаними. В іншому випадку вони з'єднані. Інформація про те чи є точки з'єднаними зберігається у пам'яті. Коли ж треба знайти шлях, то алгоритм знаходить найбільш ближчу точку навігації до точки старту та починає шукати точку, яка має зоровий контакт з точкою призначення. При цьому в пріоритет ставляться точки, які знаходяться в стороні точки призначення. Це досягається порівнянням скалярних добутків вектору погляду від точки навігації до заданої точки призначення та векторів від точки навігації до поєднаних з нею точок навігації.

Імплементація навігаційної системи почалась з побудування даних про з'єднання. Це відбувається на етапі завантаження рівнів. Кожен нод перевіряє чи контактує він з кожним іншим нодом. Якщо так – вони поєднані. Якщо ні – вони не поєднані.

Фрагмент коду, який відповідає за побудову інформації про з'єднання:

```
public override void Start()
{
  base.Start();

  SnapToGround();

  Navigation.AddPoint(this);
}

public void SnapToGround()
{
  Position = Navigation.ProjectToGround(Position);
```

```
}

public void BuildConnectionsData()
{
  connected.Clear();

  List<NavPoint> list = Navigation.GetNavPoints();

  foreach (NavPoint p in list)
  {
    if (p == this) continue;

    var hit = Physics.LineTraceForStatic(Position.ToNumerics(), p.Position.ToNumerics());

    if (hit.HasHit) continue;
    connected.Add(p);

  }

    }
```

Наступною частиною розробки було розробити сам алгоритм пошуку шляху. Усе починається зі знаходження найближчого ноду до точки початку пошуку шляху. При цьому враховується, що він повинен бути доступним до переміщення з неї. Для цього проводиться невидима лінія між двома точками. Якщо лінія пересікає статичну геометрію рівня, то це означає, що між цим нодом та точкою старту є перешкода і між ними немає шляху. В такому випадку перевіряється наступний нод. Цей цикл повторюється доки точка старту не знайдеться, або доки усі ноди не будуть перевірені.

Фрагмент коду, котрий реалізує цю логіку:

```
public static NavPoint GetStartNavPoint(Vector3 start)
{
  navPoints = navPoints.OrderBy(point => Vector3.Distance(start, point.Position)).ToList();

  foreach (var point in navPoints)
  {
    var hit = Physics.SphereTraceForStatic(start.ToNumerics(), point.Position.ToNumerics(), 0.3f);

    if (hit.HasHit == false)
      return point;

  }

  return null;

        }
```

Метод для пошуку шляху до цільової точки починається зі створення списку для зберігання знайдених точок шляху та копії історії пройдених подів, що дозволяє зберегти контекст пошуку. Збільшується лічильник ітерацій, що допомагає відстежувати кількість спроб пошуку шляху. Якщо запит містить таймер затримки і він ще не завершився, повертається список, що містить лише цільову точку, таким чином запобігаючи подальшим обчисленням. У разі перевищення допустимої кількості ітерацій, яка встановлена для запобігання зацикленню алгоритму, метод повертає шлях, що складається з точок, отриманих з поточної історії. Для цього вибирається точка, яка знаходиться найближче до цільової, що допомагає визначити найкращу альтернативу, якщо прямий шлях неможливий.

Якщо кількість елементів в історії перевищує допустиму глибину, що може вказувати на надто довгий або складний шлях, метод повертає порожній список або шлях до найближчої до цілі точки. Це обмеження допомагає оптимізувати пошук та уникати надмірно довгих шляхів. Наступним кроком є сортування всіх з'єднаних точок за напрямком до цілі, що допомагає орієнтувати пошук у правильному напрямку. Поточний под додається до історії, що забезпечує відстеження пройдених точок і запобігає їх повторному відвідуванню.

Потім перевіряється наявність перешкоди між поточною точкою та ціллю за допомогою методу трасування, який виявляє перешкоди на шляху. Якщо перешкода присутня, кожна з'єднана точка перевіряється рекурсивно на наявність шляху до цілі. Цей процес повторюється, поки не буде знайдено можливий шлях, забезпечуючи детальний аналіз можливих маршрутів. Якщо перешкод немає, повертається шлях, що складається з точок історії та цільової точки, що дозволяє досягти цілі найкоротшим шляхом. В кінці методу повертається список, що містить послідовність точок, які формують шлях до цільової точки, враховуючи всі обмеження та перешкоди, які могли виникнути під час пошуку.

Фрагмент коду, котрий реалізує цю логіку:

```
    public List<Vector3> GetPathNext(List<NavPoint> history, Vector3 target, ref
int totalItterations, PathfindingQuery query = null)
    {
      List<Vector3> output = new List<Vector3>();

      List<NavPoint> myHistory = new List<NavPoint>(history);

      totalItterations++;

      if (query != null)
      {
        if (query.deathDelay.Wait() == false)
          return new List<Vector3> { target};
      }

      if (totalItterations > 300)
      {
        output = PointsToPositions(history);

        Vector3 closest = output.OrderByDescending(pos =>
Vector3.Distance(pos, target)).ToArray()[0];

        List<Vector3> result = new List<Vector3>();

        foreach (Vector3 p in output)
        {
          result.Add(p);
          if (Vector3.Distance(p, target) < 0.1f)
            return result;
        }
      }

      if (history.Count>MaxDepth)
      {
        return new List<Vector3>();

      }

      connected = connected.OrderByDescending(point =>
Vector3.Dot((point.Position - Position).Normalized(), (target -
point.Position).Normalized())).ToList();

      myHistory.Add(this);
```

```
        bool hited = false;
        var hit = Physics.SphereTraceForStatic(Position.ToNumerics(),
target.ToNumerics(), 0.3f);

        hited = hit.HasHit;

        if (hited)
        {
           foreach(NavPoint point in connected)
           {
              if (myHistory.Contains(point)) continue;

              List<Vector3> result = point.GetPathNext(myHistory, target, ref
totalItterations);

              if(result.Count>0)
              {
                 return result;
              }
           }
        }
        else
        {
           output = PointsToPositions(myHistory);
           output.Add(target);
           return output;
        }

        return output;
            }
```

Фізика

Фізика є однією з основних частин будь-якого ігрового рушія. Він є тим, що дозволяє світу бути динамічним та інтерактивним. Без найпростішого фізичного рушія неможливо реалізувати більшість основних і найбільш розповсюджених ігрових механік. Наприклад: переміщення гравця та неігрових персонажів, реєстрація попадань та фізичне оточення.

Обрати правильний фізичний рушій складно, оскільки кожен з них має власні переваги та недоліки. Було розглянуто декілька варіантів з різними перевагами та недоліками.

Таким чином було обрано використовувати Bullet Physics. Він простий у імплементації, використанні, підтримує мову програмування C# та має велику спільноту.

Імплементація Bullet Physics була доволі простою, оскільки механізм його роботи дуже схожий на Box2D, з яким в мене був досвід роботи. Усе починається з методу ініціалізації. Це відбувається у методі "Start", який викликається при завантажені рівня. Метод «Start» ініціалізує фізичний рушій Bullet Physics, забезпечуючи створення необхідних об'єктів та налаштування параметрів фізичного симулятора. Спочатку метод здійснює скидання та очищення існуючих об'єктів, якщо вони були створені раніше, щоб запобігти витокам пам'яті та забезпечити правильне оновлення. Скидаються та видаляються попередній солвер, широкофазний детектор, диспетчер і списки об'єктів для видалення.

Далі метод перевіряє, чи існують динамічний і статичний світи. Якщо вони вже існують, вони також видаляються для створення нових інстанцій. Після цього створюється конфігурація колізій і диспетчер, які відповідають за налаштування параметрів обробки колізій та управління ними. Конфігурація встановлює кількість ітерацій для обробки багатоточкових колізій, що забезпечує точність симуляції.

Наступним кроком є створення широкофазного детектора, який допомагає виявляти потенційні колізії між об'єктами у світі. Після цього створюється солвер, який відповідає за розв'язання фізичних обмежень і впливів між об'єктами.

Після налаштування базових компонентів створюється динамічний світ, який буде містити всі рухомі об'єкти. Встановлюється гравітація, яка буде діяти на всі об'єкти у світі, і включається безперервна обробка колізій для забезпечення великої точності симуляції. Також створюється окремий статичний світ для статичних об'єктів, які не змінюють свої позиції під час симуляції. Список статичних об'єктів очищується, щоб підготувати його до заповнення новими об'єктами.

Насамкінець, динамічному світу призначається об'єкт для візуалізації колізій, що дозволяє відображати фізичні взаємодії під час розробки та відлагодження гри. Метод завершується успішною ініціалізацією фізичного рушія, готового до симуляції фізичних взаємодій у грі.

Фрагмент коду, котрий реалізує цю логіку:

```
public static void Start()
{
  solver?.Reset();
  solver?.Dispose();
  broadphase?.Dispose();
  dispatcher?.Dispose();
  removeList.Clear();
  if (dynamicsWorld != null)
  {
    dynamicsWorld.Dispose();
    dynamicsWorld = null;
  }
```

```
            if (staticWorld != null)
            {
                staticWorld.Dispose();
                staticWorld = null;
            }
            var collisionConfig = new DefaultCollisionConfiguration();
            collisionConfig.SetConvexConvexMultipointIterations(1, 1);
            collisionConfig.SetPlaneConvexMultipointIterations(1, 1);
            dispatcher = new CollisionDispatcher(collisionConfig);
            broadphase = new DbvtBroadphase();
            solver = new SequentialImpulseConstraintSolver();
            dynamicsWorld = new DiscreteDynamicsWorld(dispatcher, broadphase,
solver, collisionConfig);
            dynamicsWorld.Gravity = new Vector3(0, -9.81f, 0); // Set gravity
            dynamicsWorld.DispatchInfo.UseContinuous = true;
  staticWorld = new DiscreteDynamicsWorld(new CollisionDispatcher(new
DefaultCollisionConfiguration()), new DbvtBroadphase(), new
SequentialImpulseConstraintSolver(), new DefaultCollisionConfiguration());
            staticBodies.Clear();
            dynamicsWorld.DebugDrawer = new
DebugDrawer(GameMain.Instance.GraphicsDevice);
        }
```

Далі треба було реалізувати метод для симуляції фізичного світу і оновлення даних для статичних об'єктів. Фізичний рушій робить більшість роботи, проте треба було зробити усе таким чином, щоб операції були потоково-безпечними.

Спочатку метод блокує доступ до статичного світу для забезпечення потокової безпеки під час оновлення статичних об'єктів. У цьому блоці код проходить через кожен статичний жорсткий тіло у списку статичних об'єктів і викликає метод оновлення з батьківського об'єкта, щоб синхронізувати їх положення і стан з поточною сценою.

Після завершення оновлення статичних об'єктів метод блокує доступ до динамічного світу, знову ж таки для забезпечення потокової бехпеки під час симуляції. Перевіряється стан гри, щоб дізнатися, чи не поставлена вона на паузу. Якщо гра не на паузі, викликається метод StepSimulation, який здійснює обчислення фізичної симуляції для поточного кадру. Часова дельта, помножена на масштаб часу, передається методу, щоб врахувати час, який минув від останнього кадру, і коригувати швидкість симуляції.

Також передаються параметри, що визначають максимальну кількість під-кроків симуляції та мінімальний час для кожного під-кроку. Це дозволяє забезпечити стабільність симуляції незалежно від продуктивності системи та зміни частоти кадрів.

```
public static void Simulate()
{
  lock (staticWorld)
  {
    foreach (StaticRigidBody staticRigidBody in staticBodies)
    {
      staticRigidBody.UpdateFromParrent();
```

```
            }
        }
        lock (dynamicsWorld)
        {
            if (GameMain.Instance.paused == false)
                dynamicsWorld.StepSimulation(Time.DeltaTime * Time.TimeScale,
steps, Math.Max(1 / 30f, Time.DeltaTime));
        }
    }
```

Далі треба оновити положення об’єктів відповідно до положення фізичних тіл у динамічному фізичному світі. Спочатку рушій проходить через всі об'єкти колізій у динамічному світі, перевіряючи кожен з них на наявність фізичного тіла. Якщо об'єкт колізії є жорстким тілом, перевіряється, чи має він зв'язок з ігровим об'єктом. Якщо жорстке тіло не прив'язане до жодного об'єкта, воно видаляється з динамічного світу для запобігання витокам пам'яті та збереження ресурсів.

Далі перевіряється, чи є об'єкт активним. Якщо він не активний, він пропускається і обробка переходить до наступного об'єкта. Для активних об'єктів метод отримує ігрову сутність, до якої прив'язане жорстке тіло, і оновлює її положення на основі поточного трансформації фізичного тіла. Це забезпечує синхронізацію ігрових об'єктів з фізичними об'єктами у світі.

Метод також обчислює орієнтацію жорсткого тіла, витягуючи матрицю обертання з його трансформації. Використовуючи цю матрицю, створюється кватерніон обертання, який потім перетворюється у вектор кутів Ейлера. Це дозволяє точно визначити орієнтацію об'єкта в трьох-вимірному просторі.

Отримані значення позиції та обертання призначаються відповідним властивостям ігрової сутності, забезпечуючи її точне положення та орієнтацію у

сцені гри. Завдяки цьому методу, всі об'єкти у грі відображаються у своїх правильних позиціях та з правильною орієнтацією, синхронізованих з фізичним симулятором, що дозволяє створити реалістичну фізичну взаємодію у грі.

Фрагмент коду, котрий реалізує цю логіку:

```
public static void Update()
{
  for (int i = 0; i < dynamicsWorld.NumCollisionObjects; i++)
  {
    CollisionObject colObj = dynamicsWorld.CollisionObjectArray[i];
    if (colObj is RigidBody rigidBody)
    {
      if (colObj.UserObject is null)
      {
        dynamicsWorld.RemoveRigidBody((RigidBody)colObj);
        continue;
      }
      if (colObj.IsActive == false) continue;
      Entity ent = (Entity)colObj.UserObject;
      Vector3 pos = colObj.WorldTransform.Translation;
      ent.Position = new Microsoft.Xna.Framework.Vector3((float)pos.X,
(float)pos.Y, (float)pos.Z);
      Matrix4x4 rotationMatrix = rigidBody.WorldTransform.GetBasis();
```

```
            Quaternion rotation =
Quaternion.CreateFromRotationMatrix(rotationMatrix);
            Vector3 rotationEulerAngles = ToEulerAngles(rotation);
            ent.Rotation = new
Microsoft.Xna.Framework.Vector3((float)rotationEulerAngles.X,
(float)rotationEulerAngles.Y, (float)rotationEulerAngles.Z);
        }
    }
}
```

Звук

Опис імплементації звуку з використанням Fmod

Клас AudioClipFmod використовує Fmod для відтворення звуку. При створенні об'єкта звук прив'язується до загальної групи каналів і налаштовуються початкові параметри, такі як фільтр нижніх частот і група каналів. Метод Update оновлює параметри звуку, зокрема висоту тону, і викликає метод Apply3D для застосування 3D-позиціонування. Apply3D робить канал тривимірним, встановлює параметри циклічності та висоти тону, а також викликає Apply3DData для оновлення 3D-даних позиції та швидкості. Метод ApplyStartSoundData встановлює початкові параметри звуку, зокрема 3D-позицію, висоту тону та циклічність. Метод ApplyDistance розраховує атенюацію звуку на основі відстані між слухачем та джерелом звуку і встановлює відповідну гучність. Метод Play відтворює звук, при необхідності оновлюючи його параметри, а методи Stop та Pause зупиняють та призупиняють звук відповідно.

Опис імплементації звуку з використанням XAudio

Клас AudioClipLegacy використовує XAudio для відтворення звуку через SoundEffectInstance. При створенні об'єкта налаштовується екземпляр звукового ефекту і його параметри циклічності. Метод Update оновлює параметри звуку, такі як висота тону та циклічність, і запускає відтворення звуку, якщо він перебував у стані очікування. Apply3D обчислює атенюацію звуку на основі відстані між слухачем та джерелом звуку, а також панорамування на основі позиції слухача відносно джерела звуку, коригуючи гучність та панорамування звукового ефекту. Метод Play запускає відтворення звуку, встановлюючи необхідні параметри та запускаючи відтворення з початку при необхідності. Методи Stop та Pause зупиняють та призупиняють звук відповідно, а метод Destroy очищає ресурси, звільняючи пам'ять.

2.2 Механізм рендеру та алгоритм освітлення

Механізм рендеру є основною і найбільш складною частиною рушія. Його структура одночасно оригінальна і запозичена. Основною метою його створення було досягнення одночасно гнучкого та оптимізованого методу рендеру комплексних та простих сцен різних форм та структур. Також важливо було зробити усе таким чином, щоб прозорість мала гарний вигляд і не спричиняла помітні графічні артефакти.

Етап рендеру починається після закінчення етапу ігрової логіки. Рушій зберігає такі фінальні дані сцени: трансформація камери, трансформація усіх кластерів направленого світла, трансформація та додаткові параметри моделей, трансформація кісток скелетних моделей. Ця інформація зберігається за для того щоб потік ігрової логіки не змінював дані під час процесу рендеру, оскільки це може викликати візуальні артефакти та критичний збій у програмі.

Після збереження візуальних даних сцени створюється список об'єктів для рендеру. На цьому етапі відбуваються перевірки, які не дозволяють об'єктам по за зоною видимості потрапити на етап відправки запитів відображення на відеокарту. Це відбувається перевіркою чи пересікає матриця зони проекції камери сферу, яка оточує модель. Після цього об'єкти додаються у список рендеру у такій послідовності:

1. Непрозорі об'єкти відсортовані від найближчого на найдальшого
2. Прозорі об'єкти від найдальшого до найближчого

Така послідовність рендеру вирішує відразу дві проблеми:

- Перемальовування пікселя – Сучасні відеокарти та відео-драйвери мають можливість завчасно перестати опрацьовувати піксель, якщо він не проходить глибинну перевірку.
- Послідовність відображення прозорих об'єктів – Прозорі об'єкти повинні вимальовуватись від найдальшого до найближчого. Таким чином новий піксель частково перекриває собою старі. Це можна порівняти із художником, який малює спочатку фон, а потім передній план, оскільки не можна намалювати щось позаду того, що вже є намальованим.

Коли етап підготовки завершений ігровий рушій переходить до етапу вимальовки.

Спочатку готуються та оновлюються буфери візуалізації. Вони використовуються для створення ефектів у екранному просторі та раннього глибинного тесту.

Після створення та оновлення буферів відбувається етап раннього тексту. Він поєднує відразу дві функції:

- Оновлення буферу глибини за для запобігання перемальовування пікселя

- Проведення тесту на те чи видно об'єкт у фінальному кадрі. Це відбувається за допомогою оточення запиту на вимальовку у «render query». Це дозволяє отримати інформацію про те скільки пікселів було отримати в результати вимальовки. Якщо це значення менше ніж один, то це означає, що об'єкт перекрито іншим і нема потреби його вимальовувати під час основного проходу вимальовки. Проте цей процес займає час, тому інформацію про результати цієї перевірки рушій отримує тільки на наступному кадрі.

Далі йде етап оновлення тіней направленого світла. Направлене світло, тобто світло він сонця або місяця, розбито на три кластера. Кластери використовуються як рівень деталізації для тіней. Тобто тіні, які ближчі до камери мають більшу роздільну здатність, ніж тіні, які знаходяться далі від неї. Це робиться оскільки тіні мають бути одночасно деталізованими та видимими на великій відстані, а використання однієї великої карти тіней є неможливим, оскільки:

- Текстури мають максимальний об'єм пам'яті, який вони можуть займати, а збільшення роздільної здатності експоненційно збільшує використання пам'яті. Наприклад текстура із в двічі більшою горизонтальною та вертикальною роздільною здатністю буде займати у 4 рази більше місця.
- Вимальовування на текстуру дуже великого розміру буде займати дуже багато часу, хоча більшість деталей будуть непомітними, оскільки вони знаходяться на великий відстані.

Таким чином ігровий рушій використовує три кластера:

1. Радіусом у вісім метрів та роздільною здатністю у 2048 пікселів
2. Радіусом у тридцять метрів та роздільною здатністю у 2048 пікселів

3. Радіусом у сто п'ятдесят метрів та роздільною здатністю у 4096 пікселів

При цьому останній кластер не вимальовує динамічні об'єкти. Це робиться за для економії ресурсів, оскільки великі об'єкти зазвичай є статичними. Також останній кластер оновлюється з фіксованим інтервалом. Це також робиться за для економії ресурсів.

Далі оновлюються точкові джерела освітлення. Вони бувають трьох типів:

- Статичні – відкидають тіні тільки від статичних об'єктів. Джерела світла такого типу є продуктивними, оскільки оновлюють тіні тільки під час етапу завантаження рівня та якщо камера рухається далеко від них(за для економії пам'яті).
- Динамічні – Відкидають тіні як від статичних, так і від динамічних об'єктів. Вони є найменш продуктивними, оскільки оновлення тіней для точкового джерела світла вимагає вимальовувати об'єкти навколо нього 6 разів кожен кадр.
- Без тіней – Точки світла, які не відкидають тіней є найбільш продуктивними, оскільки вони не споживають пам'ять та не вимагають вимальовування об'єктів навколо себе.

Після цього наступає основний етап – Прямий рендер. На цьому етапі прораховується освітлення та графічні ефекти для кожного видимого об'єкта. Спочатку вимальовуються непрозорі об'єкти від найближчого до найдальшого, а потім прозорі об'єкти від найдальшого до найближчого. Прямий рендер дозволяє використовувати один і той самий шейдер як для прозорих, так і для непрозорих об'єктів. Алгоритм шейдеру виглядає так:

1. Спочатку йде етап вертексного шейдера. На ньому прораховується трансформація кісток моделі, світова трансформація моделі та її проекція зі світового простору на екранний простір. Це найлегший етап

вимальовки, тому складні операції краще, за можливістю, проводити по вертексно, а не по піксельно.

2. Відбувається перевірка на те чи буде цей піксель перекритий в майбутньому використовуючи дані з ранньої глибинної перевірки. Якщо він перекривається, то піксель відкидається. Через те, що навіть операція зчитування однієї текстури глибини є дуже важко використовується раннє відкидання пікселів на відеокарті. За для цього об'єкти вимальовуються саме в такій послідовності.
3. Після цього зчитуються текстури з фізичними даними поверхні, такі як: колір, шорсткість, металевість та затінення. Усі дані окрім кольору зберігаються в одній текстурі у різних кольорових каналах. Це економить ресурси, бо треба зчитати тільки одну текстуру замість 3 окремих.
4. Далі йде зчитування текстури нормалі поверхні. Ця текстура використовується для надання та симуляції додаткової деталізації освітлення. Колір текстури інтерпретується як вектор відносно поверхні. Цей вектор потім переводиться у світовий простір використовуючи матрицю трансформації отриману з нормалі та дотичної поверхні.
5. Уся отримана інформація передається у функцію прораховування освітлення. В ній маючи інформацію про джерела освітлення та карти тіней можна отримати освітлення для об'єкта. Спочатку прораховуються тіні для направленого джерела світла з використанням кластерних карт тіней. Потім йде світло від неба, яке симулюється простим скалярним добутком вектору поверхні та направленого вниз вектору. Далі йде освітлення для точкових джерел світла. Для тіней зчитується значення з кубічної карти тіней використовуючи вектор нормалі поверхі як координата. Для обох точкових та направлених джерел світла використовуються модель освітлення Phong. Вони

використовує скалярний добуток поверхні та напрямку світла для самозатінення та скалярний добуток векторів пів-шляху для віддзеркаленого світла.

6. Потім використовуючи буфер з кадром сцени з повними відображеннями, котрий був прорахований на попередньому кадрі і виготовуючи значення металевості та шорсткості прораховується сила відображення поверхні. Використовуючи силу відображення поверхні фінальний колір лінійно інтерполюється до кольору відображення.
7. В кінці фінальний колір, нормаль поверхні, світове положення пікселя та сила відображення виводяться у окремі текстури для майбутнього використання у ефектах у екранному просторі.

Потім прораховується текстура відображення у екранному просторі. Використовуючи позицію пікселя у світовому просторі та напрямок його нормальні можна прорахувати відображення у екранному просторі. Це робиться за допомогою інформації про глибину, де вона використовується для визначення чи пересікає промінь, який симулює відбиття світла певну точку у світовому просторі. Якщо інформації з екранного простору не вистачає, то використовується кубічна карта відображення.

Далі наступає черга пост-процесингу. Це включає в себе корекцію кольорів, tone-mapping, bloom і т.д. Усі вони використовують інформацію з екранного простору та мають власні буфери.

В кінці додається інтерфейс та відбувається презентація кадру. Презентація кадру. В залежності від графічного API презентація відбувається або на потоці рендеру, або на фоновому потоці. Якщо презентація відбувається на фоновому потоці, то це економить час процесору, який був би витрачений на очікування обробки кадру на відеокарті, оскільки процесор тільки додає задачі в чергу на виконання.

Висновок

Написання власного ігрового рушія – це дуже цікавий та корисний досвід. При його написанні було досліджено багато тем, знання яких може бути використано як і при роботі з вже існуючими ігровими рушіями, так і загалом при будь-якої програми.

Ігровий рушій це дуже велика та дуже складна програма, яка має багато рівнів абстракції і є одним з найкращих прикладів реалізації ООП. По своїй суті ігровий рушій є абстракцією та поєднанням багатьох окремих бібліотек, систем, інструментів та концепцій таким чином, що забезпечує зручне їх використання програмістом, або гейм дизайнером. Мова програмування C# дуже сильно цьому допомагає. Вона не є настільки ж швидкою як наприклад C++, але її загальна зручність та наявність зручних можливостей, як наприклад гаряче перезавантаження коду, автоматична очистка пам'яті та можливість отримання інформації про типи даних, їх функції та властивості. Разом це все створює просте у використанні та розширені робоче середовище.

Процес розробки ігрового рушія передбачає ряд складних виборів, оскільки більшість проблем є дилемами без одного правильного рішення. Як наприклад вибір фізичного рушія. Один може бути зручним та простим у використанні, в той час як інший складніший, але більш функціональний.

Етап рендеру є найбільш комплексним в питанні вибору підходу, який буде найкраще підходити. Різні опції можуть поєднуватись одна з одною, що може призводити до різних результатів. Було обрано використання прямого рендеру з раннім глибинним проходом для основного рендеру сцени та відкладений рендер для комплексних ефектів та пост-процесингу. Цей підхід вирішив проблему із рендером прозорих об'єктів та запобігає перемальовуванню пікселя, що сильно економить ресурси.

Література

1. The Guts of Deferred Rendering [Електронний ресурс]. – 2013. – Режим доступу до ресурсу: https://gamedevelopment.tutsplus.com/forward-rendering-vs-deferred-rendering--gamedev-12342a
2. TrenchBroom 2024.1 Reference Manual [Електронний ресурс]. – 2024. Режим доступу до ресурсу: https://trenchbroom.github.io/manual/latest/
3. Skeletal Animation learnopengl.com [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://learnopengl.com/Guest-Articles/2020/Skeletal-Animation
4. MonoGame Documentation [Електронний ресурс]. – 2024. – Режим доступу до ресурсу: https://docs.monogame.net/
5. Bullet Real-Time Physics Simulation DOCUMENTATION [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://pybullet.org/wordpress/index.php/forum-2/
6. FmodForFoxes ReadMe.md [Електронний ресурс]. – 2023. – Режим доступу до ресурсу: https://github.com/Martenfur/FmodForFoxes?tab=readme-ov-file

Додаток А

Частина коду програми на мові програмування C#

```
public RenderTarget2D StartRenderLevel(Level level)
{
   if (outputPath!=null)
   DownsampleToTexture(outputPath, oldFrame);
   List<StaticMesh> renderList = level.GetMeshesToRender();

   RenderPrepass(renderList);
   PointLight.DrawDirtyPointLights();
   graphics.GraphicsDevice.DepthStencilState = DepthStencilState.Default;
   graphics.GraphicsDevice.RasterizerState = RasterizerState.CullNone;
   ShadowMapEffect.Parameters["close"].SetValue(false);
   RenderShadowMapClose(renderList);
   RenderShadowMap(renderList);

   ShadowMapEffect.Parameters["close"].SetValue(true);
   RenderShadowMapVeryClose(renderList);
   graphics.GraphicsDevice.RasterizerState
Graphics.DisableBackFaceCulling?          RasterizerState.CullNone          :
RasterizerState.CullNone;

   InitSampler(25);
   RenderForwardPath(renderList);
   graphics.GraphicsDevice.BlendState = BlendState.Opaque;
   PerformPostProcessing();
   graphics.GraphicsDevice.SetRenderTarget(null);
   if(Input.GetAction("test2").Holding())
     return DepthPrepathOutput;
   return outputPath;

}
```

```
public void InitSampler(int max = 10)
{
    samplerState = new SamplerState();
    samplerState.Filter = Graphics.TextureFiltration ? (Graphics.AnisotropicFiltration ? TextureFilter.Anisotropic : TextureFilter.Linear) : TextureFilter.PointMipLinear;
    samplerState.AddressU = TextureAddressMode.Wrap;
    samplerState.AddressV = TextureAddressMode.Wrap;
    samplerState.AddressW = TextureAddressMode.Wrap;
    samplerState.MipMapLevelOfDetailBias = -10;
    samplerState.MaxAnisotropy = 16;
    int i = 0;
    while (i<=max)
    {
      try
      {
        graphics.GraphicsDevice.SamplerStates[i] = samplerState;
        i++;
        if (i > 10) break;
      }catch(Exception e) { break; }
    }
  }
  void RenderForwardPath(List<StaticMesh> renderList, bool onlyTransperent = false)
  {
    Stats.RenderedMehses = 0;
    UpdateShaderFrameData();
    graphics.GraphicsDevice.Viewport = new Viewport(0, 0, (int)GetScreenResolution().X, (int)GetScreenResolution().Y);
    graphics.GraphicsDevice.SetRenderTargets(DeferredOutput, normalPath, ReflectivenessOutput, positionPath);
```

```
        graphics.GraphicsDevice.Clear(ClearOptions.DepthBuffer, Color.Black, 1.0f,
0);
        RenderLevelGeometryForward(renderList);
        graphics.GraphicsDevice.RasterizerState = RasterizerState.CullNone;
        if (Graphics.DrawPhysics)
          Physics.DebugDraw();
      }
      public void RenderLevelGeometryForward(List<StaticMesh> renderList, bool
onlyTransperent = false, bool OnlyStatic = false)
      {
        foreach (StaticMesh mesh in renderList)
        {
          if (mesh == null) continue;
          if (mesh.Transperent || onlyTransperent == false)
          {
            if(mesh.Static || OnlyStatic==false)
              mesh.DrawUnified();
          }
        }
      }
public void RenderLevelGeometryDepth(List<StaticMesh> renderList, bool
OnlyStatic = false, bool onlyShadowCasters = false)
{
  graphics.GraphicsDevice.DepthStencilState = DepthStencilState.Default;
  graphics.GraphicsDevice.RasterizerState = RasterizerState.CullNone;
  graphics.GraphicsDevice.BlendState = BlendState.Opaque;
  foreach (StaticMesh mesh in renderList)
  {
    if (mesh == null) continue;
    if (mesh.Transperent == false)
    {
      if (mesh.Static || OnlyStatic == false && mesh.CastShadows == true ||
onlyShadowCasters == false)
```

```
            mesh.DrawDepth();
        }
    }
}
```

Додаток Б

Частина коду програми на мові програмування HLSL

```
float4x4 GetBoneTransforms(VertexInput input)
{
    float4x4 identity = float4x4(
    1.0, 0.0, 0.0, 0.0,
    0.0, 1.0, 0.0, 0.0,
    0.0, 0.0, 1.0, 0.0,
    0.0, 0.0, 0.0, 1.0);

    float sum = input.BlendWeights.x + input.BlendWeights.y + input.BlendWeights.z
+ input.BlendWeights.w;

    if (sum < 0.05f)
        return identity;
    float4x4 mbones =
    Bones[input.BlendIndices.x] * (float) input.BlendWeights.x / sum +
    Bones[input.BlendIndices.y] * (float) input.BlendWeights.y / sum +
    Bones[input.BlendIndices.z] * (float) input.BlendWeights.z / sum +
    Bones[input.BlendIndices.w] * (float) input.BlendWeights.w / sum;

    return mbones;
}
float3 GetTangentNormal(float3 worldNormal, float3 worldTangent)
{
    float3 normalMapSample = float3(0, 0, 1);
    // Create the tangent space matrix as before
```

```
    float3 bitangent = cross(worldNormal, worldTangent);
    float3x3 tangentToWorld = float3x3(worldTangent, bitangent, worldNormal);

    // Transform the normal from tangent space to world space
    float3 worldNormalFromTexture = mul(normalMapSample, tangentToWorld);

    // Normalize the final normal
    worldNormalFromTexture = normalize(worldNormalFromTexture);
    return worldNormalFromTexture;
}
PixelInput DefaultVertexShaderFunction(VertexInput input)
{
    PixelInput output;

    float4x4 boneTrans = GetBoneTransforms(input);
    float4x4 BonesWorld = mul(boneTrans, World);
    float4 worldPos = mul(input.Position, BonesWorld);
    output.Position = worldPos;
    output.MyPosition = output.Position.xyz;
    output.Position = mul(output.Position, View);
    if (Viewmodel)
    {
        output.Position = mul(output.Position, ProjectionViewmodel);
        output.Position.z *= 0.02;
    }
    else
    {
        output.Position = mul(output.Position, Projection);
    }
    output.MyPixelPosition = output.Position;
    output.TexCoord = input.TexCoord;
```

```
        // Pass the world space normal to the pixel shader
    output.Normal = mul(input.Normal, (float3x3)BonesWorld);
    output.Normal = normalize(output.Normal);


    output.Tangent = mul(input.Tangent, (float3x3) BonesWorld);
    output.Tangent = normalize(output.Tangent);

    output.TangentNormal = GetTangentNormal(output.Normal, output.Tangent);

    if (dot(output.TangentNormal, normalize(output.MyPosition - viewPos)) > 0)
       output.TangentNormal *= -1;
    output.lightPos = mul(worldPos, ShadowMapViewProjection);
    output.lightPosClose = mul(worldPos, ShadowMapViewProjectionClose);
    output.lightPosVeryClose = mul(worldPos, ShadowMapViewProjectionVeryClose);

    output.TexCoord = input.TexCoord;
    output.Color = input.Color;

    return output;
}
void DepthDiscard(float depth, PixelInput input)
{
    if (depth < input.MyPixelPosition.z - 0.02)
       discard;
}
float SampleDepth(float2 coords)
{
    return tex2D(DepthTextureSampler, coords);
}

float3 ApplyNormalTexture(float3 sampledNormalColor, float3 worldNormal, float3
worldTangent)
```

```
{
    if (length(sampledNormalColor) < 0.1f)
        sampledNormalColor = float3(0.5, 0.5, 1);
    sampledNormalColor *= float3(1, 1, 1);
    worldNormal = normalize(worldNormal);
    worldTangent = normalize(worldTangent);
    float3 normalMapSample = sampledNormalColor * 2.0 - 1.0;
    normalMapSample *= float3(-1, -1, 1);
    normalMapSample *= 1;

    // Create the tangent space matrix as before
    float3 bitangent = cross(worldNormal, worldTangent);
    float3x3 tangentToWorld = float3x3(worldTangent, bitangent, worldNormal);

    // Transform the normal from tangent space to world space
    float3 worldNormalFromTexture = mul(normalMapSample, tangentToWorld);
    worldNormalFromTexture = normalize(worldNormalFromTexture);
    return worldNormalFromTexture;
}
float DistributionGGX(float3 N, float3 H, float a)
{
    float a2 = a * a;
    float NdotH = max(dot(N, H), 0.0);
    float NdotH2 = NdotH * NdotH;
    float nom = a2;
    float denom = (NdotH2 * (a2 - 1.0) + 1.0);
    denom = PI * denom * denom;
    return nom / denom;
}

float FresnelSchlick(float cosTheta, float metallic)
{
```

```
    return metallic + (1.0 - metallic) * pow(1.0 - cosTheta, 5.0);
}

float CalculateSpecular(float3 worldPos,float3 normal, float3 lightDir, float roughness,
float metallic)
{
    float3 vDir = normalize(viewPos - worldPos);
    lightDir *= -1;
    float3 halfwayDir = normalize(vDir + lightDir);
    float NdotH = saturate(dot(normal, halfwayDir));
    float NdotV = saturate(dot(normal, vDir));
    float specular = 0.0;

    if (NdotH > 0.0)
    {
        float roughnessSq = lerp(roughness * roughness, roughness, 0.5);
        float D = DistributionGGX(normal, halfwayDir, roughnessSq);
        float G = GeometrySmith(normal, vDir, lightDir, roughnessSq);
        float F = FresnelSchlick(NdotV, metallic);
        specular = D * G / (4 * NdotV * saturate(dot(normal, lightDir)) + 0.001) *
lerp(F,1,0.3);
    }
    specular = max(specular,0);
    return specular * 1;
}
float SampleShadowMap(sampler2D shadowMap, float2 coords, float compare)
{
    float4 sample = tex2D(shadowMap, coords);
    return step(compare, sample.r);
}
float GetShadow(float3 lightCoords,float3 lightCoordsClose,float3
lightCoordsVeryClose, PixelInput input)
{
```

```
float shadow = 0;
float dist = distance(viewPos, input.MyPosition);
if (dist > 200)
   return 0;
if (tex2D(ShadowMapSampler,lightCoords.xy).r<0.01)
   return 0;
if (lightCoords.x >= 0 && lightCoords.x <= 1 && lightCoords.y >= 0 &&
lightCoords.y <= 1)
{
   if (dist < 8 && abs(dot(input.Normal, -LightDirection))>0.3)
   {
      if (lightCoordsVeryClose.x >= 0 && lightCoordsVeryClose.x <= 1 &&
lightCoordsVeryClose.y >= 0 && lightCoordsVeryClose.y <= 1)
      {
         return GetShadowVeryClose(lightCoordsVeryClose, input);
      }
   }

   if (dist < 31)
   {
      if (lightCoordsClose.x >= 0 && lightCoordsClose.x <= 1 &&
lightCoordsClose.y >= 0 && lightCoordsClose.y <= 1)
      {
         return GetShadowClose(lightCoordsClose, input);
      }
   }
   float currentDepth = lightCoords.z * 2 - 1;
   float resolution = 1;
   int numSamples = 1; // Number of samples in each direction (total samples
numSamples^2)

   float bias = ShadowBias * (1 - saturate(dot(input.Normal, -LightDirection))) +
ShadowBias / 2.0f;
```

```
        resolution = ShadowMapResolution;
        return 1 - SampleShadowMap(ShadowMapSampler, lightCoords.xy,
currentDepth + bias);
        float size = 1;
        float texelSize = size / resolution; // Assuming ShadowMapSize is the size of your
shadow map texture

        for (int i = -numSamples; i <= numSamples; ++i)
        {
            for (int j = -numSamples; j <= numSamples; ++j)
            {
                float2 offsetCoords = lightCoords.xy + float2(i, j) * texelSize;
                float closestDepth;
                closestDepth = SampleShadowMapLinear(ShadowMapSampler,
offsetCoords, currentDepth + bias, float2(texelSize, texelSize));

                shadow += closestDepth;
            }
        }
        shadow /= ((2 * numSamples + 1) * (2 * numSamples + 1));
        return (1 - shadow) * (1 - shadow);
    }
    return 0;
}

float3 CalculatePointLight(int i, PixelInput pixelInput, float3 normal, float roughness,
float metalic)
{
    float3 lightVector = LightPositions[i] - pixelInput.MyPosition;
    float distanceToLight = length(lightVector);

    float ShadowDistance = GetPointLightDepth(i, pixelInput.MyPosition);
```

```
    if (distanceToLight>ShadowDistance)
        return float3(0, 0, 0);

    float dist = (distanceToLight / LightRadiuses[i]);
    float intense = saturate(1.0 - dist*dist);
    float3 dirToSurface = normalize(lightVector);


    if (isParticle)
        dirToSurface = normal;
    intense *= saturate(dot(normal, dirToSurface));

    float3 specular = CalculateSpecular(pixelInput.MyPosition, normal, -dirToSurface,
roughness, metalic);
    intense = max(intense, 0);
    float3 l = LightColors[i] * intense;
    return l + intense * specular;
}

float3 CalculatePointLightSpeculars(int i, PixelInput pixelInput, float3 normal, float
roughness, float metalic)
{
#ifdef NO_SPECULAR
    return float3(0,0,0);
#endif
    float3 lightVector = LightPositions[i] - pixelInput.MyPosition;
    float distanceToLight = length(lightVector);
    float intense = saturate(1.0 - distanceToLight / LightRadiuses[i]);
    float3 dirToSurface = normalize(lightVector);

    if (isParticle)
        dirToSurface = normal;
    if (Viewmodel == false)
```

```
        if (dot(dirToSurface, pixelInput.Normal) < 0)
            return float3(0, 0, 0);
    intense *= 1;
    float3 specular = CalculateSpecular(pixelInput.MyPosition,normal, -dirToSurface,
roughness, metalic);

    return LightColors[i] * max(intense, 0) * specular;
}

float3 CalculateLight(PixelInput input, float3 normal, float roughness, float metalic,
float ao)
{
    float3 lightCoords = input.lightPos.xyz / input.lightPos.w;

    float shadow = 0;
    lightCoords = (lightCoords + 1.0f) / 2.0f;
    lightCoords.y = 1.0f - lightCoords.y;
    float3 lightCoordsClose = input.lightPosClose.xyz / input.lightPosClose.w;
    lightCoordsClose = (lightCoordsClose + 1.0f) / 2.0f;
    lightCoordsClose.y = 1.0f - lightCoordsClose.y;

    float3 lightCoordsVeryClose = input.lightPosVeryClose.xyz /
input.lightPosVeryClose.w;
    lightCoordsVeryClose = (lightCoordsVeryClose + 1.0f) / 2.0f;
    lightCoordsVeryClose.y = 1.0f - lightCoordsVeryClose.y;
    shadow += GetShadow(lightCoords,lightCoordsClose,lightCoordsVeryClose,
input);
    shadow += 1 - max(0, dot(normal, normalize(-LightDirection) * 1));
    shadow = saturate(shadow);
    float specular = 0;
    specular = CalculateSpecular(input.MyPosition, normal, normalize(LightDirection),
roughness, metalic) * DirectBrightness;
    specular *= max(1 - shadow, 0);
```

```
    float3 globalSpecularDir = normalize(-normal + float3(0,-5,0) + LightDirection);
    specular += CalculateSpecular(input.MyPosition, normal, globalSpecularDir,
roughness, metalic) * 0.02 ;

    if (isParticle)
        normal = -LightDirection;

    float3 light = DirectBrightness * GlobalLightColor; // Example light direction;
    light *= (1.0) - shadow;
    float3 globalLight = GlobalBrightness * GlobalLightColor * lerp(1, 0.5, dot(normal,
float3(0, -1, 0)));

    light = max(light, 0);
    light += globalLight;

    for (int i = 0; i < MAX_POINT_LIGHTS; i++)
    {
        light += CalculatePointLight(i, input, normal, roughness, metalic);
    }
    light -= (1 - ao);
    light += specular;
    light = max(light, 0);
    return light;
}
    float2 screenCoords = WorldToScreen(pos);

    return tex2D(FrameTextureSampler, screenCoords).rgb;
}

float3 GetPosition(float2 UV, float depth)
{
    float4 position = 1.0f;
    position.x = UV.x * 2.0f - 1.0f;
```

```
    position.y = -(UV.y * 2.0f - 1.0f);
    position.z = depth;
    position = mul(position, InverseViewProjection);
    position /= position.w;
    return position.xyz;
}


float4 SampleSSR(float3 direction, float3 position, float currentDepth, float3 normal,
float3 vDir)
{

    float step = 0.012;
    const int steps = 120;
    float4 outColor = float4(0, 0, 0, 0);
    float3 selectedCoords;
    float3 dir = direction;
    float3 pos = position;
    float2 coords;
    float2 outCoords;
    float weight = -0.3;
    float factor = 1.25;
    bool facingCamera = false; dot(vDir, direction) < 0;
    float disToCamera = length(viewPos - position);

    for (int i = 0; i < steps; i++)
    {
        float3 offset = dir * (step) * disToCamera / 30 + dir * 0.02 * disToCamera;
        coords = WorldToScreen(pos + offset);
        float dist = WorldToClip(pos + offset).z;
        float SampledDepth = SampleDepthWorldCoords(pos + offset);
        selectedCoords = pos + offset;
```

```
    bool inScreen = coords.x > 0.001 && coords.x < 0.999 && coords.y > 0.001 &&
coords.y < 0.999;

    weight = clamp(weight, -500000, 5);

    if (SampledDepth < currentDepth - 0.025 && facingCamera == false)
    {
       return float4(0, 0, 0, 0);
    }

    if (inScreen == false || SampledDepth>10000)
    {
       step == 0.02;
       factor = lerp(factor, 1, 0.5);
    }

    if (SampledDepth < dist && (SampledDepth > dist - 1 || facingCamera == false))
    {
       outCoords = coords;
       step /= 2;
       factor = lerp(factor, 1, 0.5);
       weight += 1;
       continue;
    }

   step *= factor;
}

weight = saturate(weight);

outColor = float4(tex2D(FrameTextureSampler, coords).rgb,  weight);

return outColor;
```

```
}

float ReflectionMapping(float x)
{
    const float n = -0.066;
    const float v = x / 3;
    return v / ((x * 10 + 1 / n)*n);
}

float CalculateReflectiveness(float roughness, float metallic, float3 vDir, float3 normal)
{
    // Calculate the base reflectiveness based on metallic
    float baseReflectiveness = metallic * 0.5;

    // Calculate the Fresnel factor using the Schlick approximation
    float F0 = lerp(0.01, 0.5, metallic);
    float F = 1; // F0 + (1.0 - F0) * pow(1.0 - abs(dot(vDir, normal)), 5.0);

    // Adjust the base reflectiveness based on roughness
    float reflectiveness = lerp(baseReflectiveness, 0.01, roughness);

    // Modulate reflectiveness by the Fresnel factor
    reflectiveness *= F;

    reflectiveness = saturate(reflectiveness);

    reflectiveness -= 0.1;
    reflectiveness *= 2.6;
    return ReflectionMapping(saturate(reflectiveness));
}

float CalcLuminance(float3 color)
```

```
{
    return dot(color, float3(0.299f, 0.587f, 0.114f));
}

float3 ApplyReflection(float3 inColor, float3 albedo, PixelInput input,float3 normal, float roughness, float metallic)
{
    float3 WorldPos = input.MyPosition;
    float3 vDir = normalize(input.MyPosition - viewPos);
    float3 reflection = reflect(normalize(input.MyPosition - viewPos), normalize(lerp(normal, input.TangentNormal, 0.4)));

    float4 ssr = SampleSSR(reflection, input.MyPosition, input.MyPixelPosition.z, normal, vDir);

    float3 cube = SampleCubemap(ReflectionCubemapSampler, reflection);
    float3 reflectionColor = lerp(cube, ssr.rgb, ssr.w);
    float reflectiveness = CalculateReflectiveness(roughness, metallic, normal, normal);
    reflectiveness = saturate(reflectiveness);
    reflectionColor *= lerp(float3(1, 1, 1), albedo, metallic);

    return lerp(inColor, reflectionColor, reflectiveness);
}

float3 ApplyReflectionOnSurface(float3 color,float2 screenCoords, float reflectiveness)
{
    float3 reflection = tex2D(ReflectionTextureSampler, screenCoords).rgb;
    float lum = CalcLuminance(reflection);
    return lerp(color, reflection * color, saturate(reflectiveness/2 * lum + reflectiveness));
}
```