ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ ІМЕНІ СЕМЕНА КУЗНЕЦЯ

ФАКУЛЬТЕТ ЕКОНОМІЧНОЇ ІНФОРМАТИКИ

КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ

Пояснювальна записка

до дипломної роботи

МАГІСТРА

на тему: «Оптимізація продуктивності аркадних ігор на мобільних пристроях»

Виконав:
студент 2 року навчання,
за освітнім ступенем «магістр»
зі спеціальності 122 «Комп'ютерні науки»
Демченко. І.Б.

Керівник: к.т.н., доц. Дорохов О.В.

Харків – 2019 рік

РЕФЕРАТ

Пояснювальна записка до дипломної роботи містить: 74 сторінок, 46 рисунків, 3 таблиці, 15 джерел, 1 додаток.

Метою роботи є оптимізація продуктивності розробленого проекту за допомогою засобів движка Unity.

Об’єктом дослідження є процес оптимізації продуктивності гри на мобільних пристроях.

Предметом дослідження є Android проект у середовищі Unity.

Результатом досліджень є покращена продуктивність проекту за допомогою методів оптимізації і засобів середовища Unity.

Алгоритми реалізовані в середовищі Unity 2019 та Microsoft Visual Studio 2017 на мові C#.

Трьохмірні моделі та графічні текстури були створені у середовищі Blender 2.8 та Photoshop 2019 CC.

UNITY, LOD, ОБ’ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ, ОПТИМІЗАЦІЯ ПРОДУКТИВНОСТІ, ANDROID, ПРОФІЛЮВАННЯ, .NET, ЗАПІКАННЯ СВІТЛА.

ABSTRACT

Explanatory note to the diploma paper contains: * pages, * figures, * tables, * sources, * applications.

The purpose of the work is to optimize the productivity of the developed project with the help of the Unity engine.

The object of the research is the process of optimizing the productivity of the game on mobile devices.

The subject of the research is the Android project in the Unity.

The results of the research are the indicators that were obtained with the help of testing and diagnostic tools of the application productivity.

The algorithms are implemented in the Unity 2019 and Microsoft Visual Studio 2017 in C#.

Three-dimensional models and graphical textures were created in Blender 2.8 and Photoshop 2019 CC.

UNITY, LOD, OBJECT-ORIENTED PROGRAMMING, PRODUCTIVITY OPTIMIZATION, ANDROID, PROFILING, .NET, LIGHT BAKING.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

LOD - level of detail;

ПЗ - програмне забезпечення;

FPS - frame per second

AI - artificial intelligence

DC - draw call

ВСТУП

У нинішній час, практично кожна людина має мобільний пристрій або який-небудь портативний гаджет. Багато хто використовує смартфони для роботи, навчання, спілкування, відпочинку. Всі сучасні мобільні пристрої працюють під управлінням операційних систем Android і IOS. Кожна з систем має свій майданчик з додатками. Android містить PlayMarket, а IOS - Appstore.

Проаналізована статистика по використанню даних сервісів каже, що щодня користувачі роблять мільйони завантажень різних додатків. Це можуть бути бізнес-додатки, додатки, які допомагають стежити за здоров'ям, різні GPS навігатори, а також мобільні ігри.

Мобільні ігри з кожним днем набирають популярність серед молодого, і не тільки, покоління. Дуже зручно вбивати час поки добираєшся до роботи, очікуєш черзі або просто лежиш на дивані. Де б людина не знаходилася, смартфони допомагають своїм власникам вирішувати повсякденні завдання, вивчати нову інформацію, а також просто відпочивати.

Так як мобільних пристроїв безмежне кількість, дуже складно робити універсальні додатки та ігри, які однаково добре будуть працювати. Розробник стикається з купою проблем, пов'язаних з технічними характеристиками мобільних пристроїв. Одні пристрої можуть спокійно обробляти важку графіку, інші здатні тільки для роботи в інтернеті.

Обчислювальні потужності ростуть з приголомшливою швидкістю. Нові покоління мобільних GPU можуть бути в 5 разів швидше своїх попередників. Швидкість новітніх моделей вже можна порівняти зі швидкістю ПК. Звичайно ж не всі програми можуть бути запущені на всіх пристроях, але існують рамки, в які необхідно укладатися, для того, що охопити більшу аудиторію користувачів.

Для цього використовуються різні способи і методи оптимізації додатків. Оптимізація мобільної гри - це коли гра працює з однаковою

частотою кадрів на широкій лінійці девайсів, включаючи найслабші платформи.

За допомогою інструментів діагностики можна знаходити вузькі місця в додатку, проводити тестування різних програмних компонентів, а за результатами дослідження вибирати ті або інші способи оптимізації, які розглянуті в другому розділі дипломної роботи.

Метою даної дипломної роботи є дослідження таких методів оптимізації і застосування їх на реальному проекті.

РОЗДІЛ 1
ОСНОВНІ ПРОБЛЕМИ ПРОДУКТИВНОСТІ МОБІЛЬНИХ ІГОР ТА ІНСТРУМЕНТИ ЇХ ВИЯВЛЕННЯ

1.1. Основний цикл оптимізації

Unity - це середовище для розробки комп'ютерних ігор, в якій об'єднані різні програмні засоби, що використовуються при створенні ПЗ - текстовий редактор, компілятор, відладчик і так далі. При цьому, завдяки зручності використання, Unity робить створення ігор максимально якісним і комфортним, а мультиплатформеність движка дозволяє розробнику охопити якомога більшу кількість ігрових платформ і операційних систем.

Оптимізація продуктивності - це процес покращення роботи гри. Зазвичай основна причина оптимізації полягає в тому, щоб зробити геймплей більш плавним або зробити гру більш доступною для широкої аудиторії, щоб гра працювала краще на нижчих пристроях.

Загальний цикл оптимізація виглядає як послідовне виконання певних ітерацій.

Якщо на цільових платформах гра працює без затримок або зниження FPS, то гру не потрібно оптимізувати. Треба акцентувати увагу на більш важливі речі, такі як дизайн або геймплей. Тому що оптимізація заради оптимізації - це марна трата часу.

Якщо під час тестів були помічені затримки або зниження FPS, то першим кроком виявлення проблеми э використання профілювання (Unity Profiler).

Після визначення проблем з продуктивністю та перед початком процесу оптимізації доцільно зайняти деякий час та розглянути, яка область оптимізації є пріоритетною. Це робиться шляхом порівняння приблизного часу, яке потребує кожна конкретна реалізація оптимізації, та підвищення результативності. Отриманий впорядкований перелік завдань з оптимізації

буди служити орієнтиром для всього процесу оптимізації, починаючи з найефективніших дій. Це також допомагає отримати найвищу ефективність з найменшими зусиллями. Треба намагатися дотримуватись цього списку, поки гра на цільових платформах не буде мати плавного ігрового процесу та стабільної частоти кадрів.

Якщо гра відмінно працює без видимих проблем і список оптимізаційних завдань не порожній, то не потрібно витрачати час на детальний процес оптимізації. Процес створення ігор повинен постійно балансувати тривалість розробки і витрачений час на оптимізацію, тому що, якщо займатися не значними проблемами,то гра може запросто не дійти до релізу.

Якщо гра все ще має проблеми, то проект повинен буди профільований, а поточні дані повинні будити збережені для подальшого порівняння. Наступні збереженні дані можна порівняти з попередніми, що дозволяє більш детально вивчити стан всього процесу оптимізації. Повторне профілювання дозволяє зрозуміти, що сталося після виконання однієї задачі оптимізації, незалежно від того, чи принесла ця процедура поліпшення продуктивності чи ні.

1.2. Типи проблем продуктивності

В Unity гру можна оптимізувати різними способами в залежності від характеру проблем. Проблеми продуктивності можна розділити на кілька типів залежно від їх характеру.

Спайк (з англ. Spike шип) - це раптове падіння частоти кадрів в грі. Це помічається, коли гра раптово зупиняється і не рухається протягом помітного часу. Це може порушити занурення гравця в гру або змусити його зробити помилку, яку він би не зробив інакше. Пікові навантаження можна розглядати, як високі точки на графіку профілювання. Приклад такого результату зображений на рисунку 1.1.

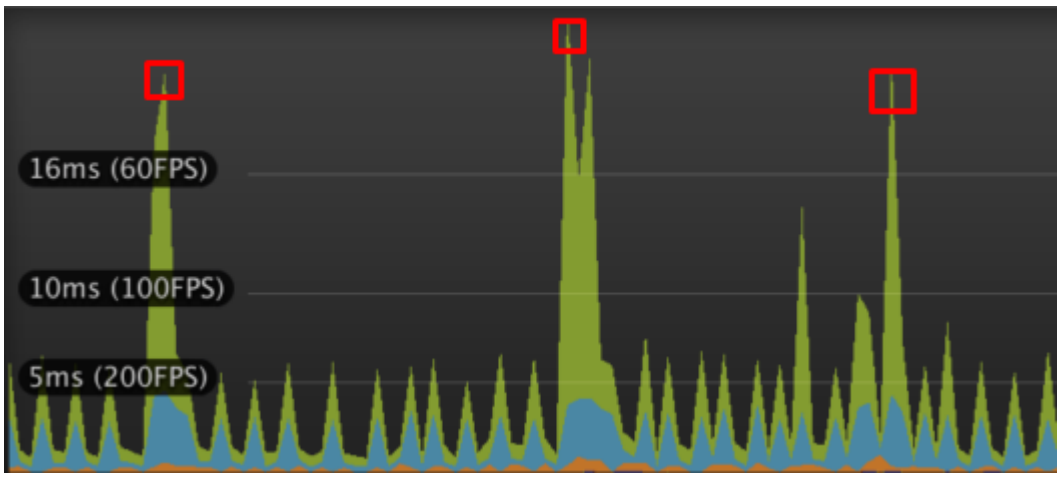

Рис. 1.1. Раптове падіння частоти кадрів

Ці різкі скачки в основному викликані складними розрахунками або складними операціями, що здійснюються протягом одного кадру. Це раптове зниження частоти кадрів є проблемою в іграх високої інтенсивності, де потребується стабільна частота кадрів і можливість чітко контролювати дії в грі, щоб відчувати себе комфортно під час водіння транспорту або прицільній стрільбі.

Також зниженню FPS сприяє «збирач сміття». Пікові навантаження збирача сміття - це падіння продуктивності, викликане специфікою системи збору сміття Unity. Це можуть бути величезні стрибки частоти кадрів і легко помітні під час гри. Пікові навантаження відбувається тоді, коли сміття пам’яті вичерпується і збирач починає працювати, прибираючи непотрібні об’єкти з пам’яті. Їх частота залежить від того, скільки сміття генерується грою в кожному кадрі.

Частота може бути знижена за рахунок зменшення кількості сміття під час роботи. Єдиний спосіб повністю позбавитись цих сплесків - не обробляти сміття під час роботи. Це дуже важлива річ, яку треба розглядати з самого початку розробки проекту.

Витрати на кожен кадр - розрахунки і операції, які виконуються на кожному кадрі. Це можуть бути, наприклад фізичні розрахунки, робота AI поведінки або робота з анімацією персонажів і об’єктів.

Вартість кожного кадру уповільнює загальну частоту кадрів в грі. Це дрібниці, які уповільнюють гру і роблять її більш вразливою. Якщо гра взагалі погано працює, то це та область, яка потребує допрацювання.

Час завантаження - це те, як довго гра завантажується. Це включає в себе перше завантаження гри при відкритті і завантаження, яке відбувається під час виконання, наприклад, між сценами або рівнями.

Хоча зазвичай це не є серйозною проблемою, але занадто часта поява екрану завантаження або надмірно тривалий час завантаження може негативно позначитися на відношенні користувача до гри.

Щоб зменшити тривалість екранів завантаження, треба розглянути можливість поділу робіт, які виконуються під час них. Це може бути попереднє завантаження ресурсів, для зменшення кількості об’єктів, які повинні бути під час запуску рівня, або зменшення складності сцен.

У грі яка має відкритий світ, де необхідно завантажувати багато об’єктів під час роботи, можна реалізувати метод переробки або потокової передачі даних. В деяких іграх невелика частина кожного кадру присвячена для завантаження або розвантажування даних. Це дозволяє відтворювати весь цикл початкового завантажування протягом усієї роботи гри.

Не мало важною проблемою э надмірне споживання пам’яті. Пам’ять RAM (пам’ять випадкового доступу) - це пам’ять, на яку завантажується гра під час роботи. Оперативна пам’ять зберігає все необхідне для гри під час виконання.

VRAM (відео пам’ять випадкового доступу) знаходиться на відеокарті і призначена для створення графічних ефектів. VRAM - це сховище, призначене для зберігання текстур і моделей, намальованих відеокартою.

Оперативна пам’ять - це те, що використовує процесор, а VRAM - це те, що використовує GPU. І те, і інше може стати вузьким місцем у проекті, що працює з не оптимізованими або просто з дуже великою кількістю активів. При нестачі пам’яті може трапитися збій.

Щоб зменшити обсяг пам’яті, необхідний для проекту, можна керуватися такими порадами:

1) Зменшити складність об’єктів і розмір текстур;
2) Зробити менш унікальними об’єкти в проекті;
3) Використовувати Object Pools.

1.3. Інструменти виявлення проблем оптимізації

У попередніх частинах я неодноразово акцентував увагу на профілюванні. Для цього процесу оптимізації і вияву проблем Unity пропонує інструмент, який називається Profiler.

Unity Profiler - це інструмент, який можна використовувати для отримання інформації про продуктивність ігри. Його можна підключити до пристроїв у мережі або пристроїв, підключеним до комп’ютера, щоб перевірити, як додаток працює на передбачуваній платформі випуску. Інструмент профілювання також можна запустити в редакторі, щоб отримати огляд ресурсів під час розробки програми.

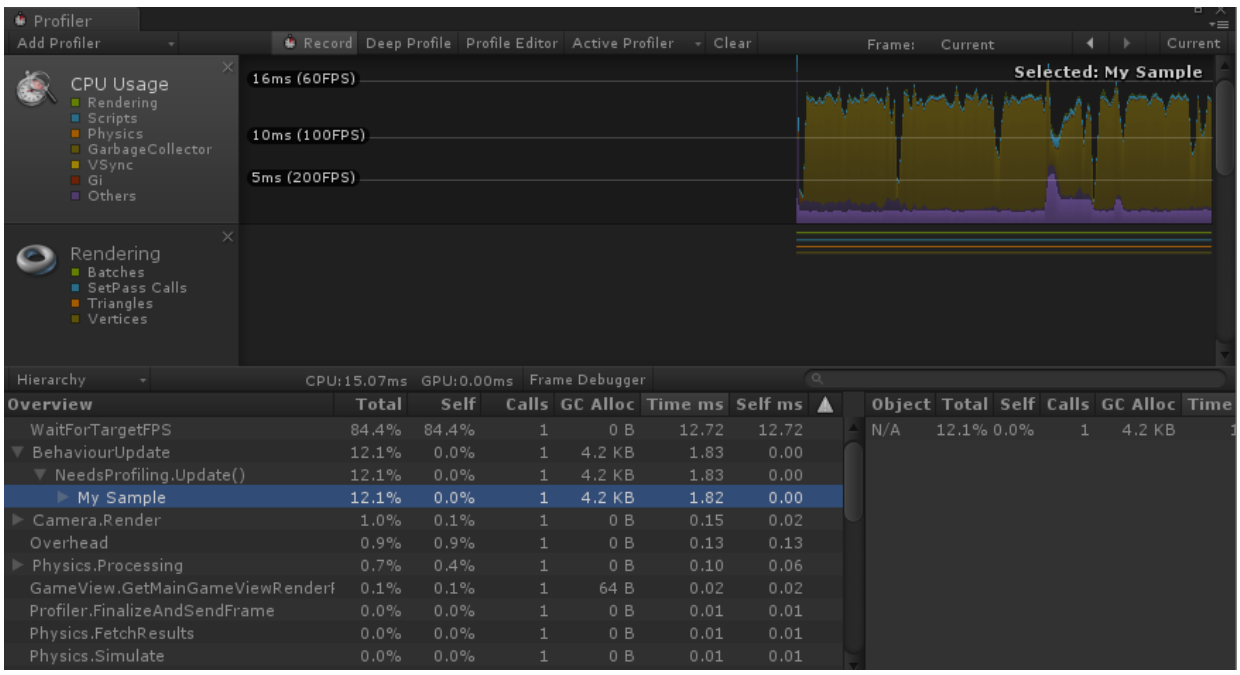

Рис. 1.2. Unity Profiler

Профілювальник збирає і відображає дані про продуктивність ігри в таких областях, як центральний процесор, пам’ять, рендерінг та аудіо. Це корисний інструмент для визначення областей, які потребують поліпшення продуктивності. Від дозволяє точно визначити як код, активи, рівні, налаштування, камери і конфігурація збірки впливають на продуктивність програми, а також відображає результати в серії графіків. Графік наочно показує де відбуваються різке зниження частоти кадрів, підвищення використання пам’яті та інше. Приклад роботи профілювальника зображений на рисунку 1.2.

Інструмент профілювання містить багато методів і режимів виявлення проблем продуктивності гри.

Одним із режимів відображення є режим ієрархії. Поруч з назвою функції відображаються різні значення. Ці значення надають інформацію про роботу функції в обраному кадрі і є відсотком від загального часу процесора, витраченого на цю функцію. Також ця інформація показує скільки разів функція була викликана, коли ця функція викликається в обраному фреймі, скільки сміття генерується функцією і час, витрачений на завершення функції в мілісекундах.

Якщо ці функції самі викликають інші, поруч з назвою відображається маленька стрілка. Натискання на цю стрілку відкриває ієрархію функцій, яка показує, як повний час роботи виконання розділено між викликаними нею функціями.

Висота ліній, показаних у верхній частині вікна, показує, скільки часу знадобилося для завершення обраного кадру. Якщо ці лінії мають помітні сплески, то відбувається зниження частоти кадрів. Пошук того, що викликало ці сплески - це спосіб зробити частоту кадрів і ігровий процес більш плавними.

Приклад роботи режиму ієрархії зображений на рисунку 1.3.

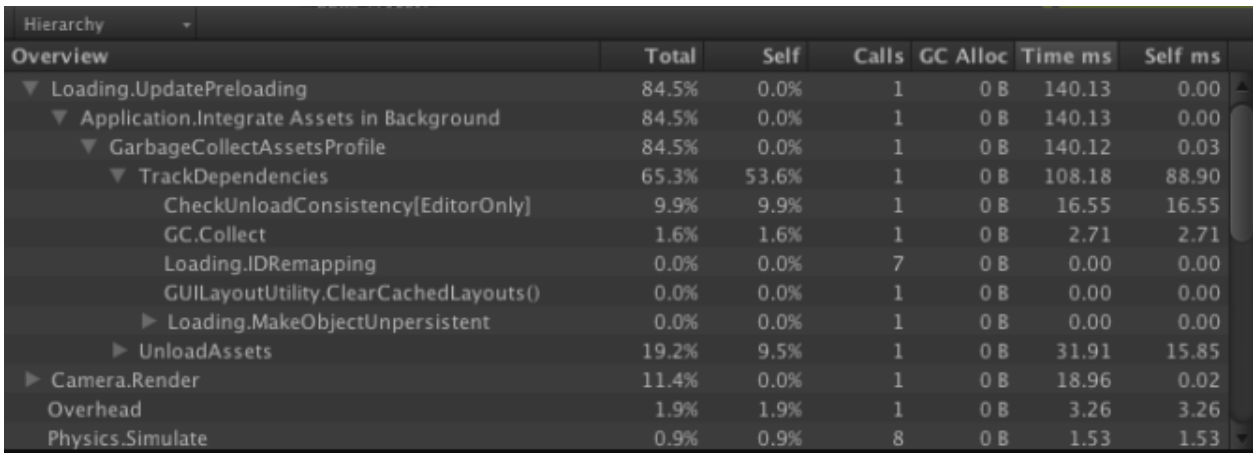

Рис. 1.3. Режим ієрархії

Шкала часу - це ще один спосіб показати данні. Ця функція корисна тим, що вона показує більш деталей, ніж звичайна версія. За допомогою шкали часу можна виявити деякі вузькі місця, наприклад, рендерінг займає дуже мало часу, але процесор дуже довго чекає, щоб відправити команду.

По суті режим ієрархії і шкала часу показує однакові дані, але в деяких випадках легше виявити певні проблеми, коли ми можемо графічно порівняти два елементи. Режим ієрархії краще, коли треба сфокусуватися на певних деталях, а шкала часу краще для розуміння загальної картини.

Unity Profile Analyzer - це додаток, який доповнює однокадровий аналіз, додаючи можливість аналізувати відразу декілька кадрів. Це корисно, коли важливо мати більш широкий погляд на роботу проекту, наприклад:

1) Оновлення версії Unity;
2) Тестування переваг оптимізації;
3) Відстежування продуктивності, як частина циклу розробки.

Він аналізує дані CPU фрейму і маркера, які витягуються з активного набору кадрів, завантажених в даний момент в Unity Profiler або завантажених з раніше збереженої сесії. Ці дані підсумовуються і відображаються за допомогою гістограм, боксових графіків та інших. Кожен маркер доповнюється упорядкованим списком:

1) Мінімум;
2) Максимум;

3) Середнє;
4) Кількість екземплярів
5) Радіус дії.

Приклад роботи Unity Profile Analyzer зображений на рисунку 1.4.

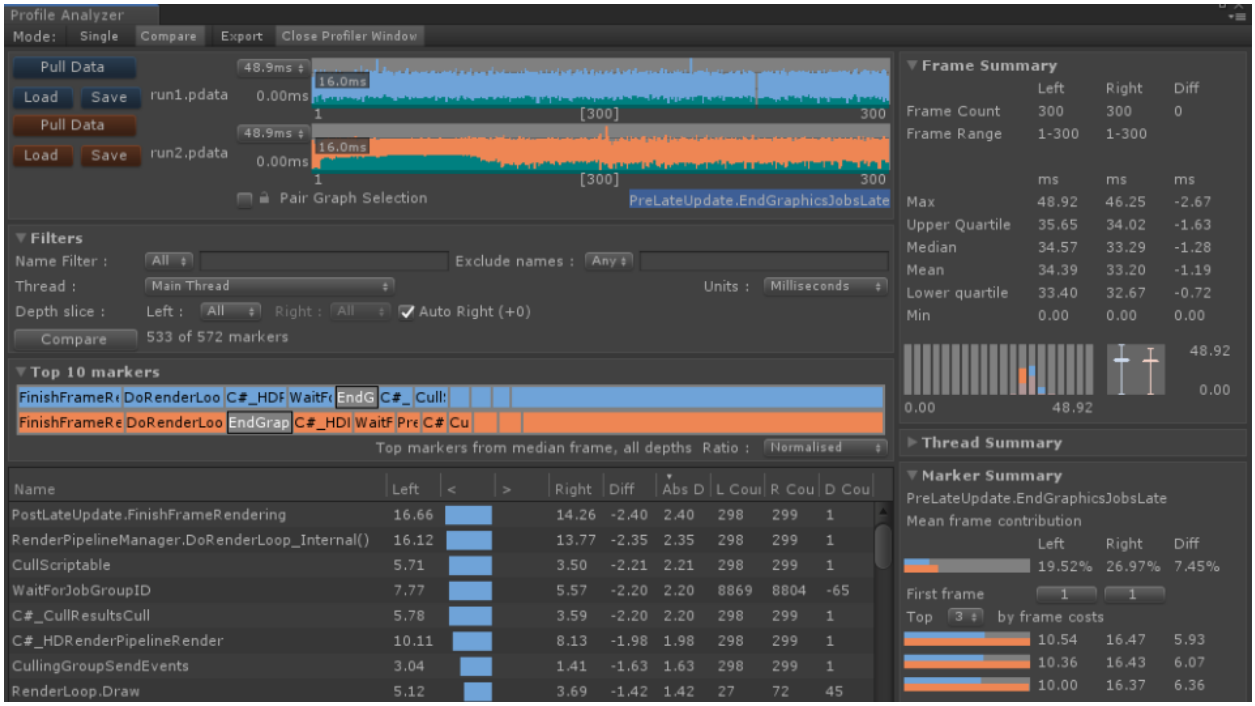

Рис. 1.4. Profile Analyzer

Отладчик кадрів Unity - це ще один інструмент покадрового тестування процесу рендерінгу. Draw Call - одинична задача, яка виконується на GPU для малювання екрану. Один кадр складається з декількох викликів Draw. Сучасні 3D ігри зі складними графічними ресурсами і ефектами можуть здійснювати тисячі викликів для відтворення кожного кадру на екрані.

Зменшення кількості Draw Call - це простий спосіб зменшити навантаження на GPU.

Для отримання інформації про те, на що GPU витрачає свої виклики Draw, можна скористатися отладчиком кадрів. Цей інструмент дозволяє отримати більш повне уявлення про процес рендерінгу. Він також використається для пошуку областей, де витрачається найбільше Draw Call, і

де потрібна оптимізація. Приклад роботи отладчика кадрів забражено на рисунку 1.5.

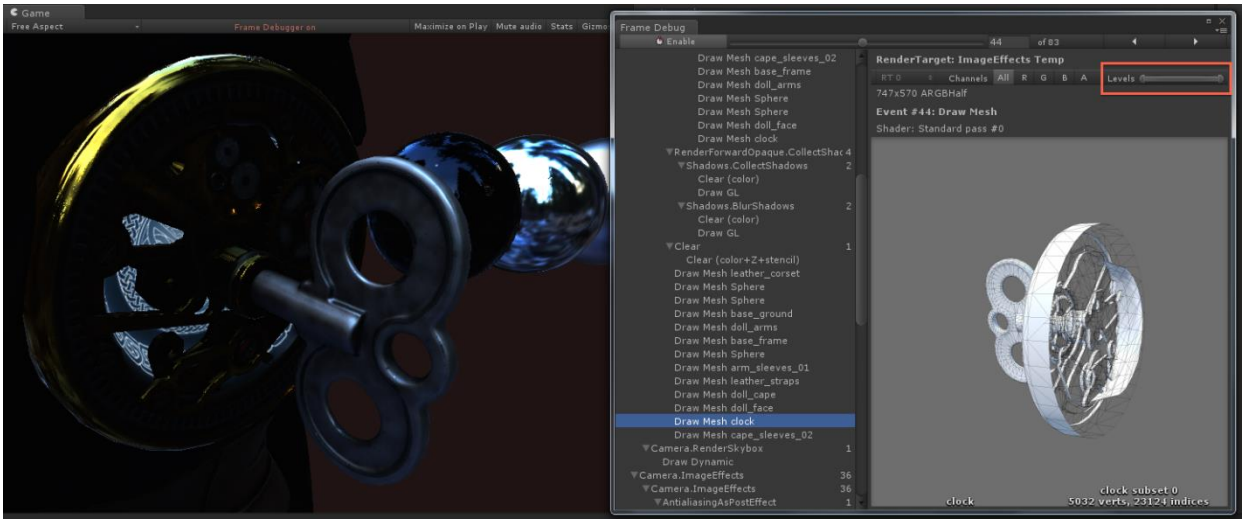

Рис. 1.5. Отладчик кадрів Unity

Отладчик кадрів підключається так само, як і профілювальник, який може бути легко використаний як всередині редактора Unity, так і на окремому пристрої.

При натискання на один виклик відображення в правій частини вікна з’являється додаткова інформація про цей конкретний виклик. Перемикання між режимом Draw Call і переглядом змін на екрані може дати достатньо інформації про те, де потрібна оптимізація.

Якщо велика частина викликів Draw Call витрачається на малювання одного об’єкта або персонажа в сцені, може знадобитися необхідність в допрацьовуванні цього об’єкта.

Unity Memory Profiler - це інструмент для профілювання пам’яті. Використання профілювальника пам’яті необхідне для визначення потенційних областей в проекті, де можна скоротити споживання пам’яті. Наприклад, можна зробити знімок використання всієї пам’яті проекту, а потім порівняти два знімка пам’яті.

Профілювальник пам’яті це рішення, яке дозволяє дослідити як невеликі проекти на мобільних пристроях, так і великі AAA-проекти на високотехнологічних машинах.

Даний інструмент - це вікно Unity редактора з оглядом нативного і керованого розподілу пам’яті, яке допомагає виявити витоку і фрагментацію пам’яті за допомогою різних методів відображення:

1) Карта дерев;
2) Карта пам’яті (послідовне візуальне уявлення блоків пам’яті);
3) Таблиці (списки об’єктів з фільтрацією);
4) Огляд (візуальних графік підключених об’єктів).

Приклад роботи Unity Memory Profiler зображений на рисунку 1.6.

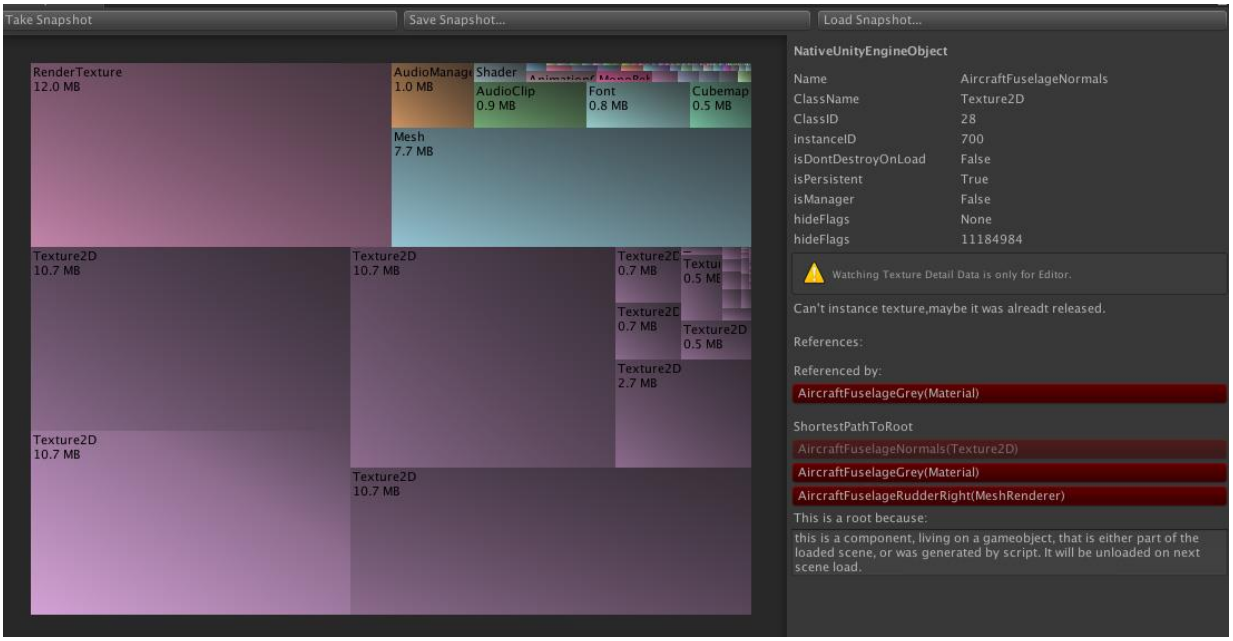

Рис. 1.6. Карта дерев в профілювальнику пам’яті

Наприклад, карта дерев - це різнокольорові квадрати, які позначають пам’ять, займану різними типами об’єктів, такі як текстури або моделі. При натисканні на квадрат, він ділиться далі на менші квадрати, що представляють окремі об’єкти. Більш детальну інформацію про ці окремі об’єкти можна побачити в правій частині вікна, а також посилання на те, що використовує об’єкт.

Unity Physics Debugger - інструмент для швидкого огляду геометрії колайдерів у сцені і профілювання поширених сценаріїв на основі фізики.

Він забезпечує візуалізацію того, які ігрові об’єкті повинні і не повинні стикатися один з одним. Це особливо корисно, коли в сцені багато колайдерів, або якщо є проблеми з синхронізацію геометрії зіткнення і рендерінгу.

В основному цей інструмент використовують для усунення несправностей фізичної активності в грі. Також можна налаштовувати відображення типів колайдерів або Rigidbody компонентів у візуалізаторі, щоб знайти джерело проблем. Приклад роботи Unity Physics Debugger зображений на рисунку 1.7.

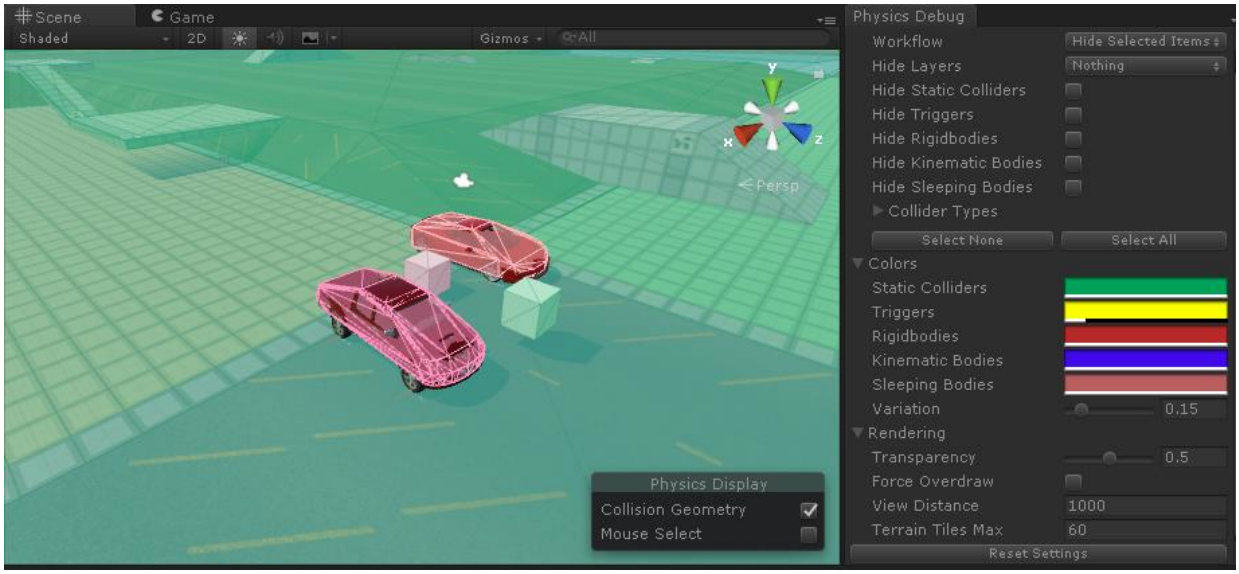

Рис. 1.7. Unity Physics Debugger

Отладчик елементів інтерфейсу Unity - це інструмент налагодження розширень редактора Unity, написаний за допомогою UIElements. UIElements - це нова графічна система для розширення редактора Unity 2019 і вище. Дана функція корисна для видавців Asset Store, які розробляють інструменти Unity.

Дуже часто використовується порівняльний аналіз для відстеження процесу оптимізації. Еталонне тестування відноситься до збору даних за проектом для оцінки його ефективності. Це відрізняється від профілювання тим, що дані аналізу повинні бути якомога ближче до характеристик кінцевого продукту. Тому дані не повинні збиратися за допомогою інструментів профілювання Unity, так як тільки запуск інструменту впливає на продуктивність.

Тестування повинно бути проведено на тому ж самому пристрої, на якому повинен працювати кінцевий продукт. Одного пристрою може бути досить, але для отримання більшої кількості даних, необхідно порівняти кілька пристроїв з різними можливостями продуктивності. Це дає більше інформації про продуктивність на пристроях середнього і високого рівня.

При тестуванні частоти кадрів проекту можна використовувати окремий скрипт для її відстеження замість інструменту профілювання.

Автоматична система контрольних показників - це запрограмована система, яка повторює однакові дії під час кожного тесту.

Для підтримки точності даних частоти кадрів між тестами, ігровий процес повинен відтворюватися якомога точніше. Кращий спосіб справитися з цим - це мати сценарії, який кожен раз грає однаково. Коли ігровий процес ідентичний між тестами, їх можна порівняти один з одним і легко контролювати процес оптимізації.

1.4. Проблеми оптимізації освітлення

Освітлення в Unity є одним із найбільш ресурсовитратних процесів. Щоб розуміти, як правильно можна провести процес оптимізації освітлення, треба знати як працює освітлення в Unity, і які техніки для цього використовуються.

Освітлення в Unity можна розглядати як «освітлення в реальному часі», або як «попередньо розраховане», і обидва метода можуть використовуватися в поєднанні для створення ефектного освітлення сцени.

Першим ділом розберемо принцип роботи освітлення в реальному часі. За замовчуванням, освітлення в Unity - спрямоване, конусне, точкове і розраховується в реальному часі. Це означає, що воно випромінює пряме світло на сцену і оновлюється кожен кадр.

Як тільки джерело світлу і об’єкт на сцені буде переміщуватися, освітлення буде негайно оновлено. Це можна спостерігати як на сцені, так і на ігровому екрані. Приклад роботи конусного освітлення зображено на рисунку 1.8.

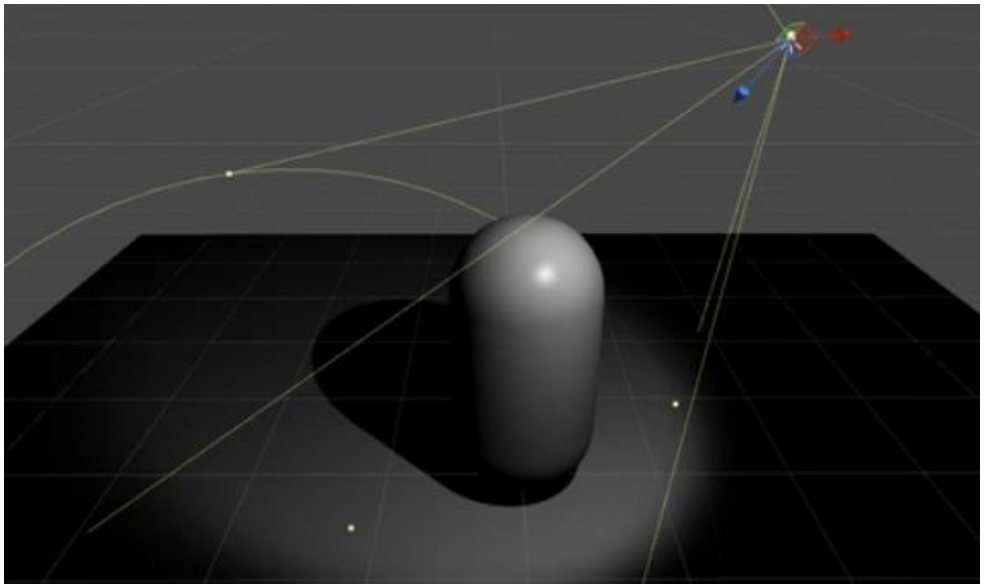

Рис. 1.8. Конусне освітлення в реальному часі

Освітлення в реальному часі є найпростішим способом освітлення об’єктів сцени і корисно для освітлення персонажів або інших рухомих геометричних фігур. На жаль, світлові промені, що випромінюються лампами Unity в реальному часі, не відображаються, коли вони використовуються самі по собі. Для створення більш реалістичних сцен с використанням таких технологій, як глобальне освітлення, необхідно використовувати рішення Unity з попереднім розрахунком освітлення.

Для оптимізації світла найкращим методом є запікання глобального освітлення.

При запіканні світлової карти розраховується вплив світла на статичні об’єкти сцени і результати записуються в текстури, які накладаються поверх геометрії сцени для створення ефекту освітлення.

Ці світлові карти можуть включати як пряме світло, що падає на поверхню, так і непряме світло, що відбивається від інших об’єктів або поверхонь сцени. Цю текстуру можна використовувати разом з інформацією про поверхні, такі як колір (альбедо) і рельєф (карти нормалі), за допомогою шейдерів, які пов’язані з матеріалом об’єкта.

При випеченому освітленні ці легкі текстури (світові карти) не змінюються в процесі гри і тому називаються «статичними». Освітлення в реальному часі можна накладати одне на інше і використовувати в якості доповнення поверх запеченого світла, але не можна змінювати самі світлові карти в інтерактивному режимі.

При такому підході ми обмінюємо можливість переміщати освітлення в ігровому процесі на потенційне підвищення продуктивності, що підходить для менш потужних пристроїв. Приклад запеченого світла зображений на рисунку 1.9.

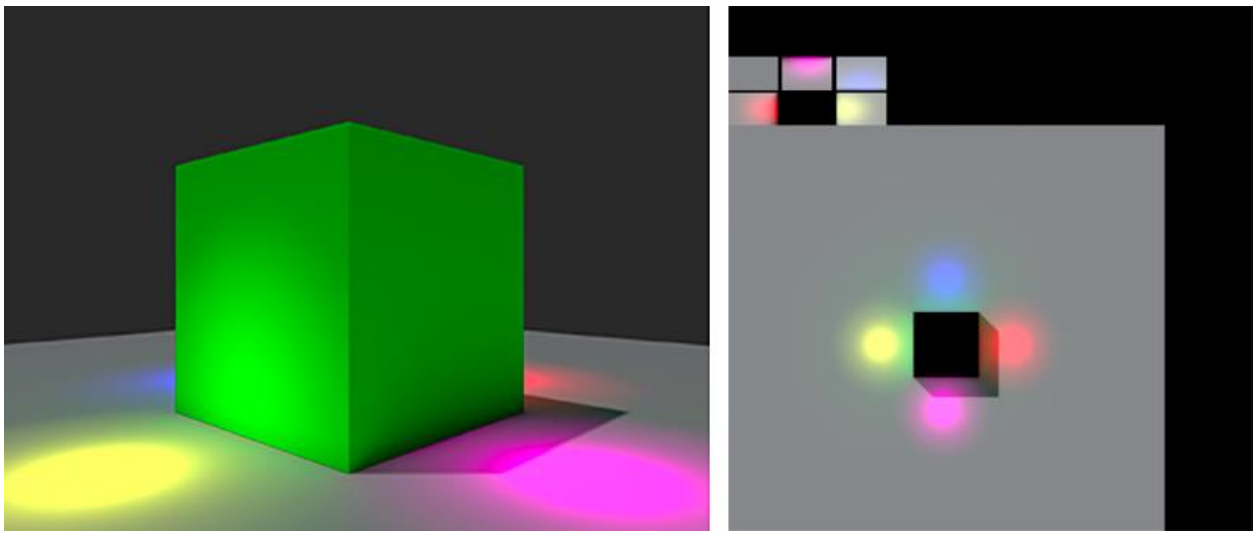

Рис. 1.9. Приклад запеченого світла і світлової карти

Іншим способом оптимізації є попередньо обчислене глобальне освітлення в реальному часі.

У той час як традиційні статичні світові карти не можуть реагувати на зміни умов освітлення в сцені, попередньо розраховане глобальне освітлення в реальному часі пропонує методику інтерактивного поновлення складної системи освітлення сцени.

Завдяки такому підходу можна створювати освітлене приміщення з багатьма глобальними освітленнями, яке реагує в режимі реального часу на зміни джерел світла. При традиційному освітленні запіканням це не можливо.

Для того щоб забезпечити ці ефекти без зниження частоти кадрів, нам потрібно перевести частину тривалих обчислень з процесу реального часу на процес попереднього обчислення. Попередні обчислення переносять тягар обчислення складної поведінки світла з того, що відбувається під час гри, на те, що може бути обчислено, коли час перестає бути критичним. Це називається автономним процесом.

Найчастіше, попередньо розраховане глобальне освітлення - це непряме (відбите) світло, яке ми хочемо зберегти на світлових картах, коли намагаємося створити реалізм в освітленні сцени. На щастя, воно зазвичай м'яке, з невеликою кількістю різких або "високочастотних" змін у кольорі. Попередньо обчислене глобальне освітлення Unity в режимі реального часу використовує ці "розсіяні" характеристики непрямого світла в наших інтересах.

Більш тонкі деталі освітлення, такі як чітке затінення, зазвичай краще створювати при освітленні в реальному часі, ніж запікати їх на світових картах. Якщо не потрібно фіксувати ці складні деталі, можна значно знизити дозвіл нашого глобального рішення щодо висвітлення.

Зробивши це спрощення під час попереднього розрахунку, ми ефективно скорочуємо кількість розрахунків, необхідних для поновлення освітлення нашого глобального освітлення під час гри. Це важливо, якщо ми

повинні змінити властивості наших джерел світла - такі як колір, обертання або інтенсивність, або навіть змінити поверхню сцени.

Для прискорення подальшої роботи Unity не працює безпосередньо з текселями світлових карт, а замість цього створює апроксимацію статичної геометрії в світі з низьким дозволом, звану "кластерами".

1.5. Висновки до розділу

1. Досліджений основний цикл оптимізації в Unity. Також були досліджені способи пошуку проблем продуктивності в додатках мобільних платформ.

2. Вивчено типи проблем оптимізації. Детально розглянутий кожен з типів, а також причини їх появи.

3. Досліджений основний інструмент діагностики додатків Unity Profiler.

4. Розглянуті основні інструменти тестування продуктивності, а також їх функції.

5. Вивчена технологія освітлення в Unity. Розглянуто типи джерел світла. Досліджена технологія запікання світла в текстурні карти.

6. Розглянуто технологію попереднього розрахунку глобального освітлення в реальному часі.

7. З'ясовано вплив освітлення на загальну продуктивність програми.

РОЗДІЛ 2
СПОСОБИ ОПТИМІЗАЦІЇ ПРОДУКТИВНОСТІ В МОБІЛЬНИХ ІГРАХ ЗА ДОПОМОГОЮ UNITY

2.1. Оптимізація графічної складової

Перш ніж приступити до видалення рядків коду, вдосконалення збірок даних і намагатися зробити все ефективним, потрібно знати, що насправді викликає проблеми з продуктивністю. Profiller - це чудовий спосіб детально ознайомитись із результатами гри. Приклад аналізу використання обчислювальних ресурсів процесора зображено на рисунку 2.1.

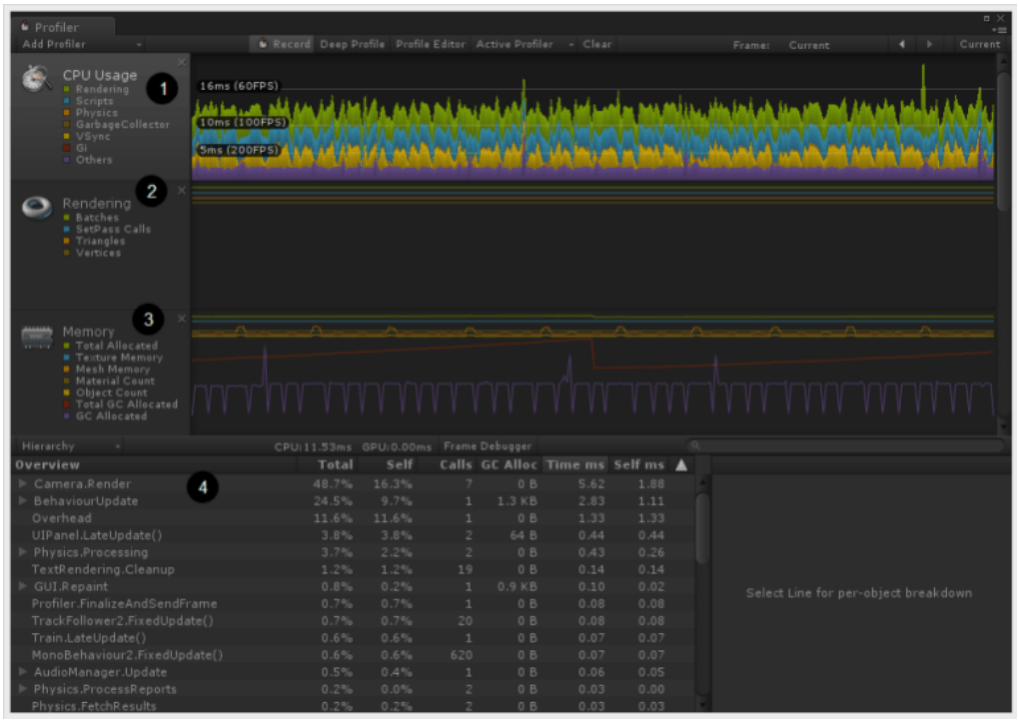

Рис. 2.1. Аналіз використання CPU

Він відображає такі категорії, як використання процесора, використання графічного процесора, візуалізація, фізика, аудіо та інше. За бажанням можна звузити специфіку в межах кожної категорії.

Unity дозволяє групувати ігрові об’єкти для покращення завантаження сцен та об’єднання дій над ними.

Часто візуальні аспекти гри - це одна з великих областей, в якій можна підвищити продуктивність. Візуальні елементи впливатимуть на draw calls. Простіше кажучи, все, що з’являється на екрані, повинно бути «намальовано». Можна уявити, що для сцени є 100 різних об’єктів, а оптимізація сцени - менше 5.

Static Batching - використовується, коли ви встановлюєте ігровий об'єкт статичним. Це означає, що об’єкт не буде рухатися, масштабуватися чи обертатися. Об'єкти, які діляться одними і тими ж матеріалами, будуть зібрані разом.

Статична партія буде найбільш ефективною, тому треба встановлювати об'єкти статичними, коли це можливо. Приклад налаштування статичних об’єктів зображено на рисунку 2.2.

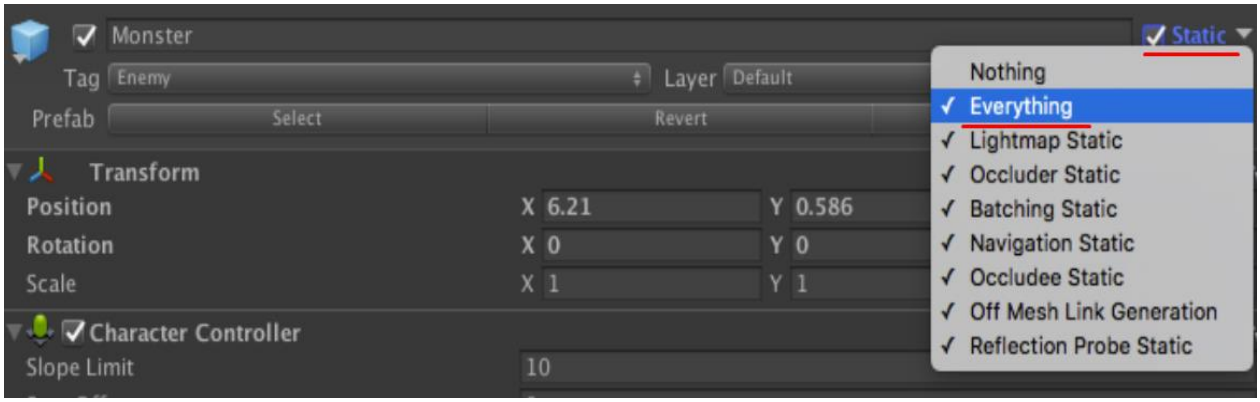

Рис. 2.2. Налаштування статичних елементів

Dynamic Batching - використовується для об'єктів, які будуть рухатися. Подібно до Static Batching воно буде об'єднувати елементи подібних матеріалів. У Dynamic Batching є деякі унікальні вимоги відповідно до Unity:

1) Batching динамічних ігрових об’єктів має певні накладні витрати на вершину, тому Dynamic Batching застосовується лише до Meshes, що містять менше 900 атрибутів вершин.

2) Ігрові об’єкти не видозмінюються, якщо вони містять дзеркальне відображення при перетворенні

3) Використання різних екземплярів матеріалу призводить до того, що ігрові об’єкти не збираються разом, навіть якщо вони по суті однакові.

4) Ігрові об’єкти зі світловими картами мають додаткові параметри візуалізації: індекс та зміщення / масштаб у карті.

5) Багатошарові шейдери розбивають пакетну групу.

Не все буде об'єднано в Unity. Такі речі, як шкіряні сітки, тканина та інші типи компонентів візуалізації, не збираються.

Також використовується технологія зменшення та повторного використання текстур.

Оскільки batching працює на основі подібних матеріалів, можна комбінувати багато об'єктів разом, якщо вони мають одну велику текстуру. Кілька текстур високої роздільної здатності уповільнить продуктивність. Незважаючи на те, що можна мати їх у своїй грі, потрібно переконатися, що ми вибірково ставимось до того, як вони використовуються.

Потрібно використовувати текстурний атлас, щоб комбінувати декілька карт текстур в одну велику карту текстур. Це звичайна методика в іграх AAA. Це не тільки сприяє зменшенню кількості використовуваних карт текстур, але також робить все набагато простішим для організації. Це робилося вкрай часто в таких іграх, як Rage and Doom, з використанням Megatextures та віртуальної текстуризації.

На великій локації дуже часто використовується метод Culling.

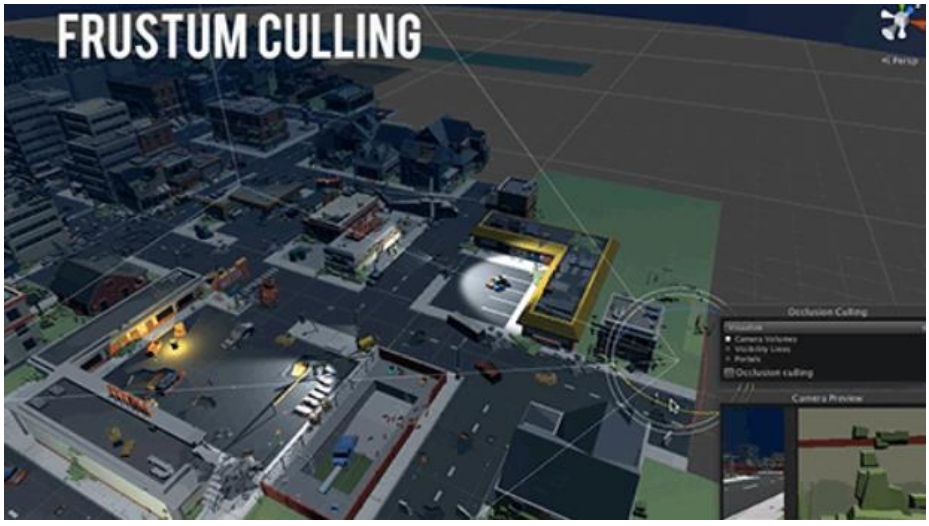

Рис. 2.3 Приклад роботи Culling

Близькі та далекі площини, а також площини, визначені полем зору камери, описують те, що в народі називають камерою усікання. Unity гарантує, що під час рендерінгу об'єктів ті, які знаходяться повністю поза цим конусом, не відображаються.

Це називається Frustum Culling. Frustum Culling відбувається незалежно від того, використовуємо ми Occlusion Culling у своїй грі. Приклад роботи Frustum Culling зображено на рисунку 2.3.

Frustum Culling - це чудовий спосіб покращити продуктивність, і те, що Unity робить за замовчуванням. Проблема лише в тому, що він може відображати предмети, які не знаходяться у прямій видимості. Уявіть, що ви стоїте перед дверима і все-таки обчислюєте всі предмети за тими дверима. Тут приходить на допомогу Occlusion Culling.

Оклюзія - це блокування, і в цьому випадку ігровий об’єкт блокує перегляд інших ігрових об’єктів. Ми можемо сказати Unity не відображати об'єкти, оклюдовані за допомогою певних параметрів, які ми позначаємо у вікні відключення оклюзії. Це дозволяє нам відображати лише ті об'єкти, до яких ми маємо прямий зір. Немає жодної причини відображати об’єкт у дальній частині зору нашої камери, якщо ми не зможемо побачити його безпосередньо.

Видимі об’єкти також піддаються оптимізації. Обмеження розмірів текстури та комбінування сіток - це чудовий спосіб покращити продуктивність разом із відсіканням усіх типів.

Рис. 2.4 Приклад деталізації текстури

Як ми можемо покращити працездатність об’єктів, які не відсікаються, але занадто далеко, щоб їх детально побачити?

LODs (Levels of Detail) - це спосіб вивести менш деталізовану версію сітки, коли вона знаходиться поза певним діапазоном. Сітка, яка знаходиться досить далеко, може бути неймовірно слабо деталізована, щоб поліпшити продуктивність.

MipMaps - це як LOD для карт текстур. MipMaps дозволяє зменшити роздільну здатність текстур, коли вони далеко від камери. MipMaps включені за замовчуванням для текстур, які імпортуються в Unity, і їх слід увімкнути, якщо ми не використовуємо камеру з фіксованою відстанню постійно або використовуємо власні унікальні інструменти для досягнення кращої продуктивності текстур. Приклад деталізації текстури зображено на рисунку 2.4.

Запікання світла є одним з кращих способів оптимізувати освітлення. Освітлення може бути досить складною темою, але в цілому треба використовувати мінімальну кількість джерел світла, необхідну для досягнення бажаного стилю. Світло може бути одним з найдорожчих аспектів гри, тому платформи часто борються з динамічним освітленням.

Рис. 2.5. Приклад сцени із запеченим світлом

У Unity є три режими світла: Реальний час, Змішаний та Запечений. У режимі реального часу світло виглядає найкраще, але це знижує продуктивність. Це також дозволяє робити динамічні тіні, які також дорого використовувати. Печене освітлення можна і потрібно використовувати, коли це можливо. Це дозволяє додати освітлення до нашого світу, одночасно надаючи переваги продуктивності від того, що нам не доведеться обчислювати динамічне світло завжди. Приклад сцени, яка має запечене освітлення зображено на рисунку 2.5.

Треба мати на увазі, що ми можемо «підробити» освітлення, використовуючи емісійні карти, які спричиняють випромінювання світла частинами текстури. Прикладом цього може бути приладова панель літака, що має багато маленьких вогнів. Створення точкового світла для кожного з них було б неймовірно дорогим, але використання емісійних областей на одній великій карті текстури не тільки служить одній і тій самій цілі, але й набагато ефективніше.

Рівень деталізації - це загальний термін проектування ландшафтів відеоігор, в яких ближчі об’єкти надаються з більшою кількістю полігонів, ніж об’єкти, які знаходяться далі. Взагалі рівень деталізації диктується системними вимогами гри.

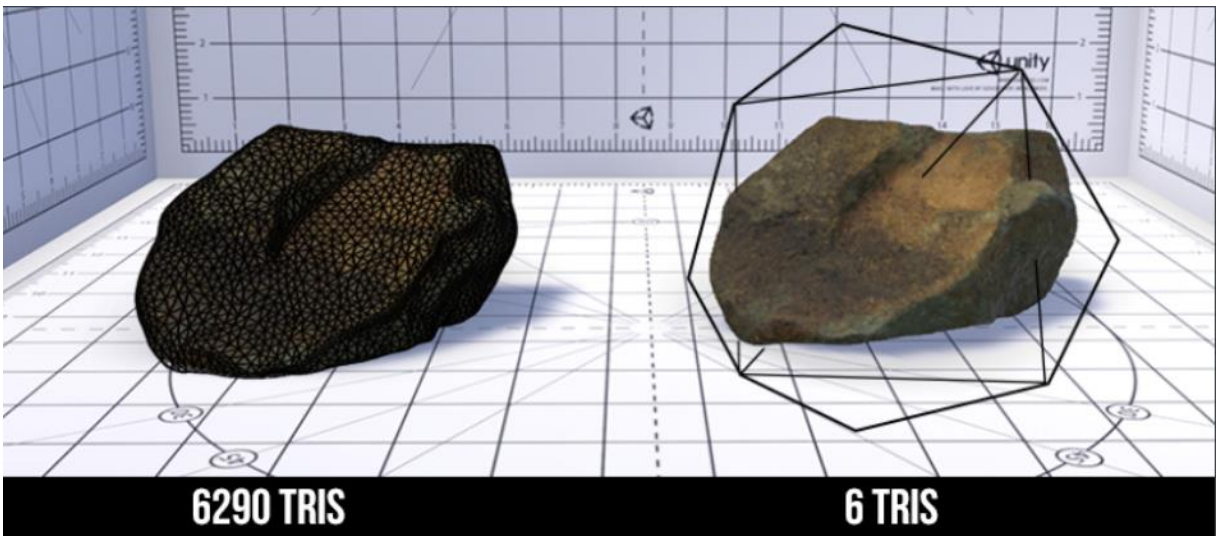

Рис. 2.6 Приклад використання LOD`s

Ранні відеоігри мали дуже помітні зміни в рівні деталізації: об'єкти на задньому плані помітно набирали полігони, коли вони виходили на перший план. Це був простий ярлик, щоб заощадити обробну потужність для виконання інших завдань, все ще представляючи гравцеві прохідний фон.

Сучасні відеоігри все ще використовують ту саму процедуру, але базовий рівень, з якого вони візуалізуються, має значно більшу кількість полігонів, тобто мало хто може помітити різницю між переднім планом та фоном неозброєним оком.

2.2. Оптимізація коду та фізики

Щоб досягти потрібної оптимізації, необхідно стежити за порядком фізичних обчислень. Розрахунки охоплюють широкий спектр механіки в Unity, але я зупинюсь на кількох, які використовуються досить часто.

Raycasts часто використовується для виявлення інших об'єктів або різних речей, таких як перевірка відстані удару зброї, напрямок тощо. Не використовуйте декілька променів, якщо їх вистачить, і не продовжуйте його на довжину, яка вам потрібна для визначення. Приклад роботи RayCast зображено на рисунку 2.7.

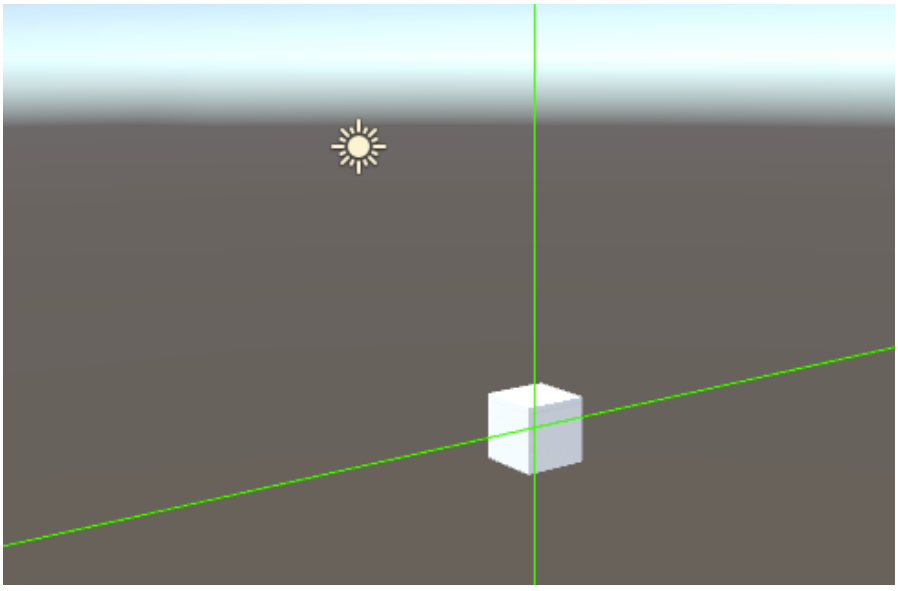

Рис. 2.7 Приклад роботи RayCast

Для визначення взаємодії між об'єктами використовують колайдери. В Unity ми можемо використовувати декілька різних колайдерів, починаючи від кубових колайдерів, до капсульних колайдерів, сітчастих колайдерів і навіть 2D-колайдерів. Потрібно створювати примітивні колайдери, коли це можливо. Існують основні форми колайдерів, такі як коробка, сфера або капсула. Сітчасті колайдери мають форму тієї сітки, яку ми вказали. Прикладом цього може бути людський персонаж, який має коллайдер, що відповідає його формі.

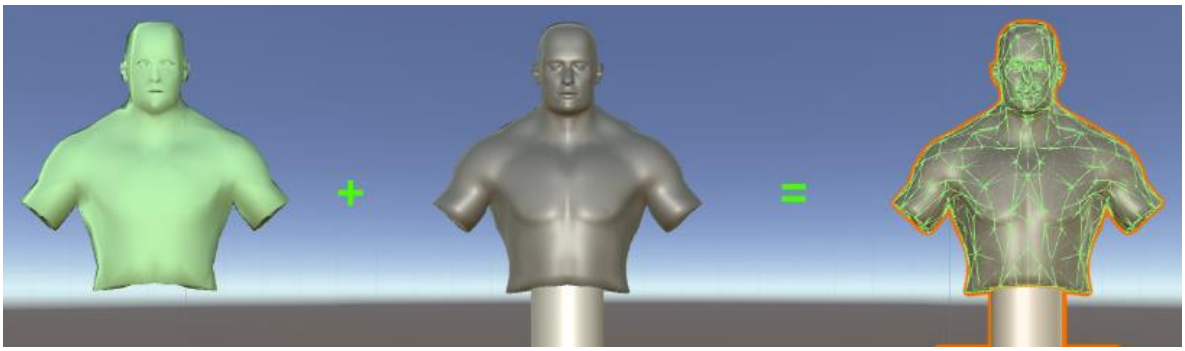

Рис. 2.8. Приклад використання спрощеної версії колайдеру

Це неймовірно дорого у використанні, і його слід уникати, якщо можливо. Якщо це абсолютно необхідно, необхідно створити низькополігонну версію сітки та призначити її як сітчастий колайдер. Приклад спрощеною версії колайдеру зображено на рисунку 2.8.

Жорсткі тіла зазвичай використовуються для додавання ваги предмету. Якщо до об'єкта приєднано Rigidbody, на нього може впливати така фізика, як гравітація інших сил. Важливо зауважити, що занадто багато об'єктів Rigidbody у грі негативно вплине на продуктивність. Для перевірки спробуйте побудувати масивну стіну з кубиків Rigidbody, які падають на землю, і подивіться, як це впливає на продуктивність у грі. Зменшення їх до мінімальної необхідної кількості - це перший крок. Ми також можемо покращити продуктивність жорстких тіл, які ми використовуємо,

визначивши, коли вони повинні “спати”.

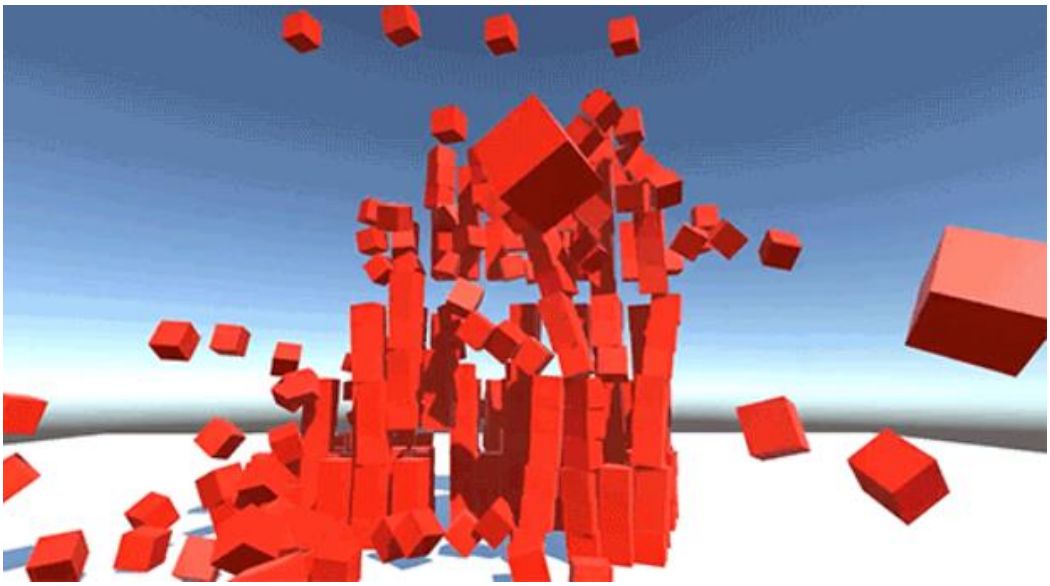

Рис. 2.9. Приклад роботи Rigidbody

Під час “сну” обчислення цих предметів значно падає і залишається таким, доки вони знову не будуть використані. Важливо додати компонент Rigidbody до об'єктів, по яким ми будете рухатися в грі, навіть якщо вони не будуть переміщуватися за допомогою сил. Об'єкти без Rigidbody вважаються статичними і їх не слід переміщувати. Ми все одно можете перемістити їх, але це призведе до зниження продуктивності. Приклад роботи Rigidbody зображено на рисунку 2.9.

Продуктивність гри також дуже сильно залежить від того, які шейдери ми використовуємо. Я не можу вникнути в складності оптимізації шейдерного коду тут, але всі ми використовуємо шейдери незалежно від того, створюємо їх ми самі чи використовуємо вбудовані шейдери. Шейдери керують усіма візуальними елементами у грі, тому оптимізовані шейдери можуть значно покращити продуктивність, оскільки для них потрібна велика кількість обчислень. Хоча ми можемо зайти і почати займатися самим кодом, я хотів би зазначити деякі загальні міркування щодо використання шейдерів щодо продуктивності.

Це, як правило, ефекти на основній камері, які застосовуються до всього погляду, що бачить камера, а це зазвичай весь екран. Такі речі, як

Global Fog і Fisheye - це звичайні ефекти зображення, які дуже часто використовують розробники. Вони майже завжди знижують продуктивність, і, якщо вони використовуються, їх слід максимально оптимізувати. Я схильний використовувати ефекти зображення як дещо зайве щось, щоб зробити візуальні зображення трохи привабливішими або веселішими, але укладання шейдерів один на одного - це вірний спосіб уповільнити роботу.

Ще одним способом підвищення продуктивності є використання зручних для мобільних пристроїв шейдерів, навіть на більш високих платформах. Якщо ви працюєте над мобільною грою, ви обов'язково повинні використовувати мобільні шейдери, оскільки для них потрібно менше обчислень. Перелік стандартних шейдерів Unity зображено на рисунку 2.10.

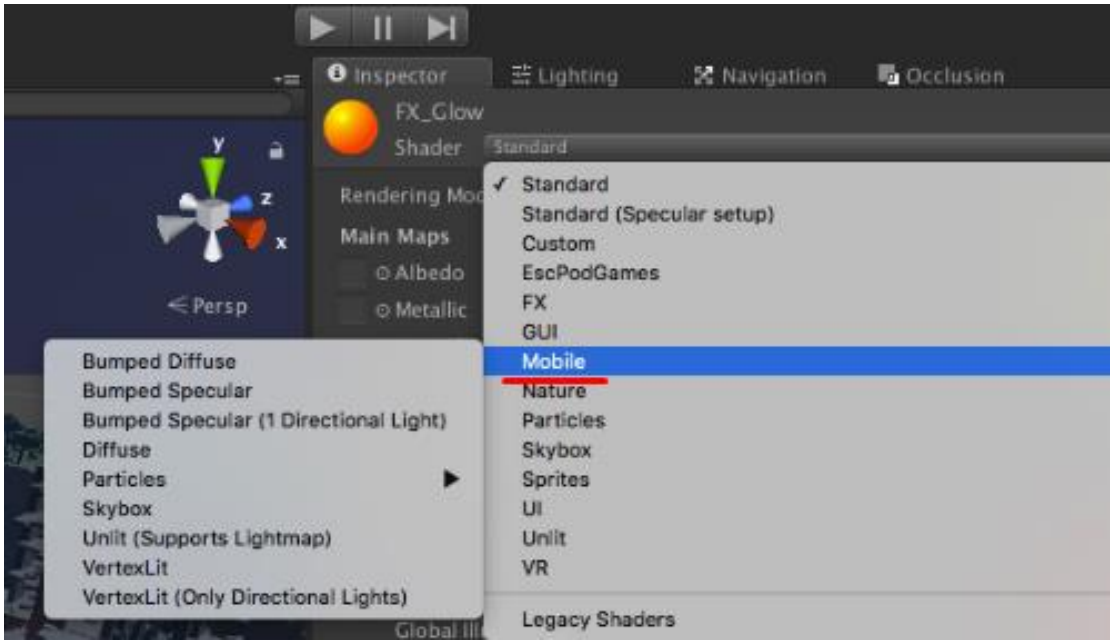

Рис. 2.10. Налаштування шейдера матеріалу об’єкта

Також необхідно ознайомитися з використанням належних типів стискання та завантаження аудіо.

Аудіо іноді не помічається при спробі оптимізації гри, але це може впливати на продуктивність так само, як і на будь-що візуальне. За замовчуванням він імпортує аудіокліпи для використання типу завантаження Decompress On Load разом із стисненням Vorbis. Параметри стискання

зображені на рисунку 2.11.

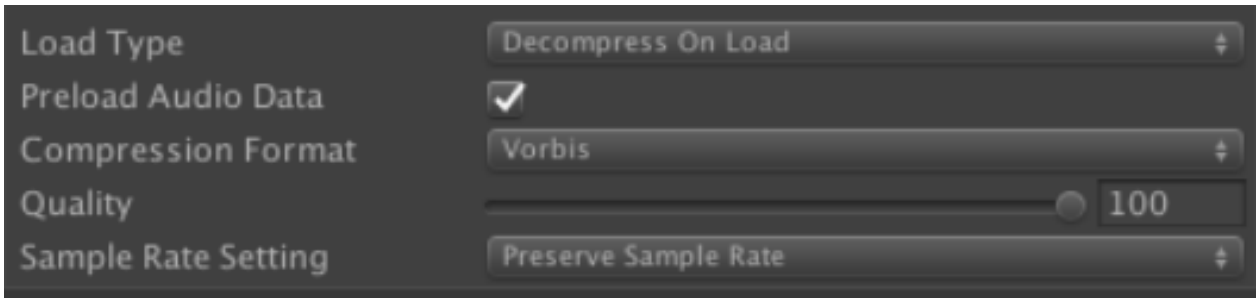

Рис. 2.11 Налаштування стискання аудіо

Важливо відзначити розміри, які ми бачимо на зображенні. Для цього аудіо, яке імпортується, імпортований розмір становить 3,3 Мб, що точно настільки збільшує пам'ять, необхідну для нашої гри. Оригінальний розмір - це кількість ОЗУ, необхідного для відтворення одного кліпу.

Звукові ефекти, як правило, короткі і тому мають невеликі потреби в пам'яті. Для цього найкраще працюватиме Decompress on Load, але тип стиснення повинен бути або PCM, або ADPCM. PCM забезпечує більш високу якість, але має більший розмір файлу, що добре для дуже короткого, але важливого звукового ефекту. ADPCM має коефіцієнт стиснення в 3,5 рази менший, ніж PCM, і найкраще використовується для аудіоефектів, які використовуються дуже часто, такі як кроки, удари, зброя тощо.

Для більш довгих аудіокліпів, таких як фонова музика або інші великі файли, краще використовувати стиснуту пам'ять, що призводить до декомпресії файлу безпосередньо перед відтворенням. Трансляція - це ще один варіант. Згідно з документами Unity, потокова передача використовує мінімальний об'єм пам'яті для буферизації стислих даних, які потім поступово зчитуються з диска та декодуються на льоту.

2.3. Висновки до розділу

1. Досліджені основні способи оптимізації продуктивності в мобільних іграх за допомогою засобів Unity.

2. Розглянуто способи зменшення використання обчислювальних ресурсів, за рахунок попередньо оброблених даних.

3. Вивчено способи оптимізації графічної частини програми.

4. Досліджено методи групування ігрових об'єктів.

5. Вивчено методи економного використання текстур і кількості оброблюваних вершин моделей на сцені.

6. Розглянуто технологію Unity по оптимізації відображення об'єктів сцени поза зоною видимості.

7. Досліджено способи оптимізації фізичних розрахунків.

8. Вивчена технологія оптимізації освітлення статичних об'єктів.

РОЗДІЛ 3
ПРОВЕДЕННЯ ОПТИМІЗАЦІЇ ПРОДУКТИВНОСТІ НА ПРИКЛАДІ АРКАДНОЇ ГРИ ДЛЯ МОБІЛЬНИХ ПРИСТРОЇВ

3.1. Тестування і проведення оптимізації освітлення

Для дослідження методів оптимізації продуктивності, був розроблений проект на движку Unity. Даний проект є аркадной грою типу Runner. Так як цей проект розрахований на мобільні пристрої, то буде проводитися оптимізація з урахуванням особливостей мобільної платформи.

Мобільні пристрої не мають великих обчислювальних потужностей, тому потрібно якомога більше приділяти увагу деталям, які на стаціонарному комп'ютері могли б бути непомітні.

Насамперед потрібно відразу вирішити проблему освітлення статичних об'єктів. Розрахунок освітлення в реальному часі завжди є одним з найбільш ресурсоємних процесів.

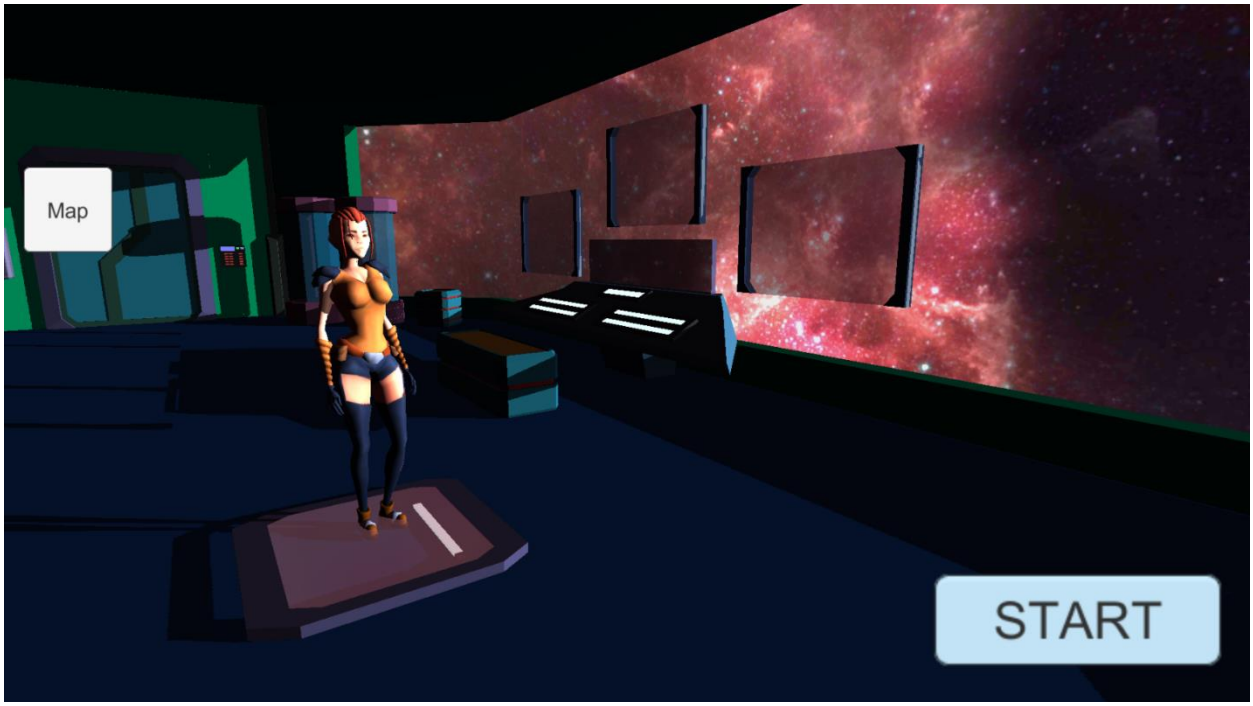

Рис. 3.1. Стартова сцена

В даному проекті статичні об'єкти знаходяться у стартовій сцені. На її прикладі, буде досліджено запікання світла і переваги використання даного методу.

Для початку розглянемо сцену з освітленням в реальному часі. Стартова сцена зображена на рисунку 3.1.

Даний тип освітлення є менш якісний, ніж запечений. Можна помітити дефекти в тінях, і в окремих деталях. Для того, щоб уникнути різні дефекти, нам необхідно збільшити якість освітлення. Наприклад збільшити дозвіл тіней в реальному часі, змінити тип тіней на "м'які тіні". Чим вище якість тіней ми вимагаємо, тим більше ресурсів буде споживати обчислення світла.

Проведемо аналіз рендерінгу сцени з освітленням в реальному часі на мобільному пристрої. Для проведення тестів використовується смартфон - ZTE Blade V10, який оснащений процесором Helio P70.

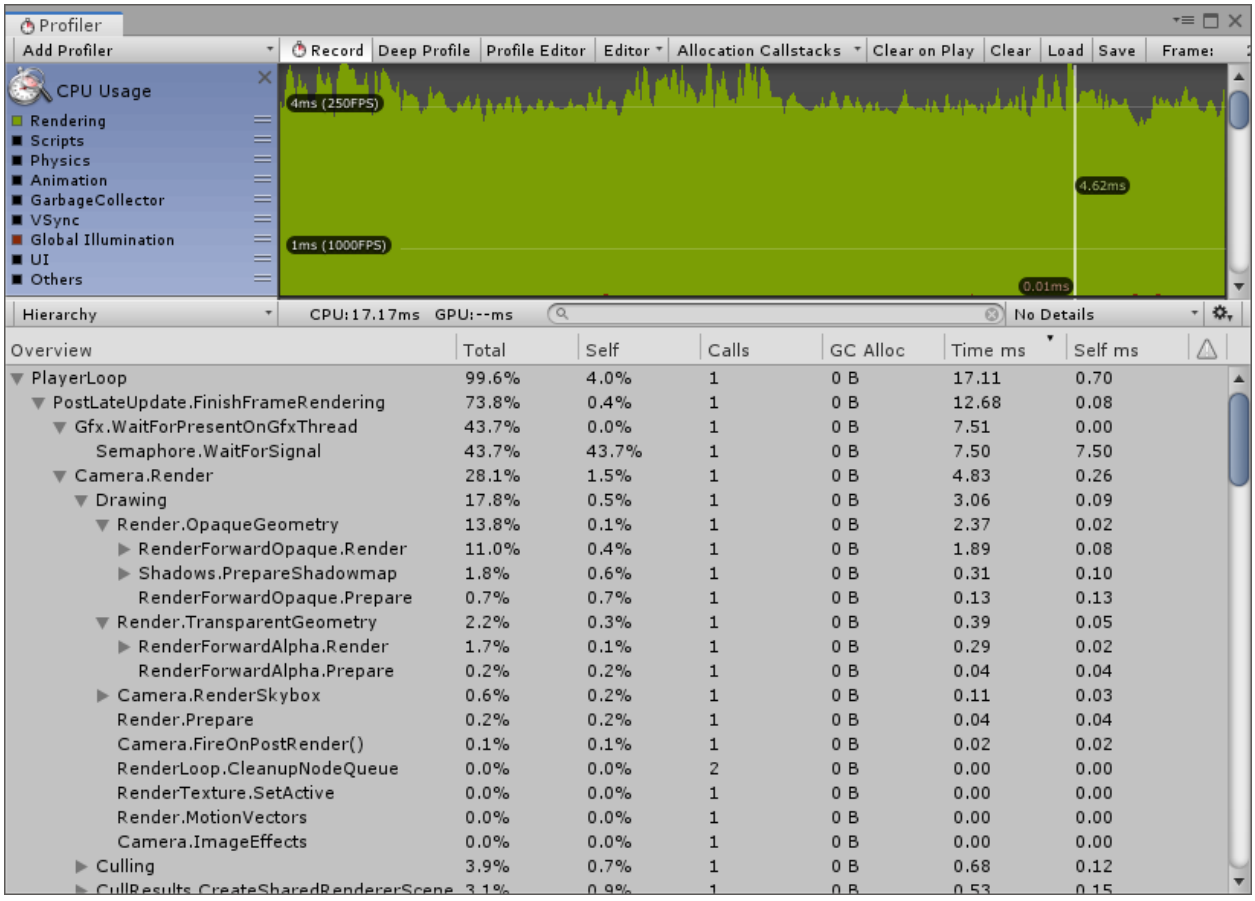

Рис. 3.2. Аналіз використання CPU

Для відображення одного кадру процесору потрібно близько 4 мілісекунд часу. Середня частота кадрів знаходиться в районі 250. Також на відображення одного кадру використовується 28 відсотків загального часу виконання роботи процесора. З іншими показниками, можна ознайомитися на рисунку 3.2.

Тепер зробимо процес запікання світла. Для цього потрібно налаштувати параметри освітлення, а також параметри LightMaper.

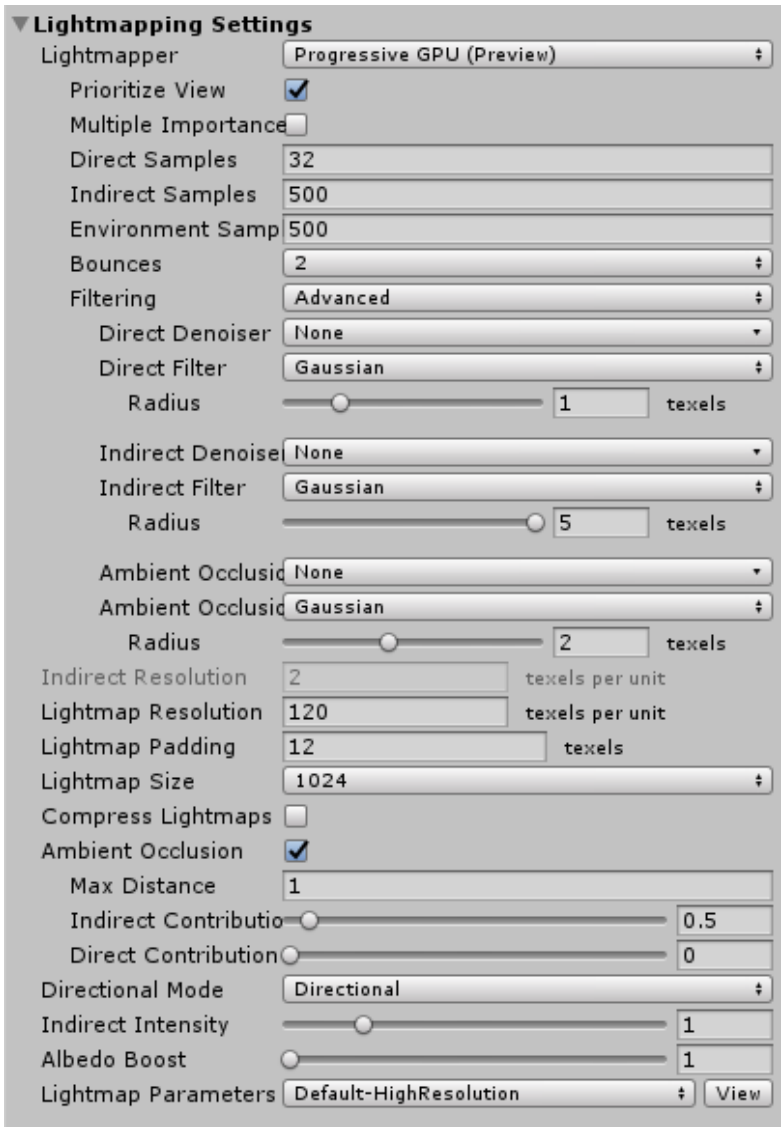

Рис. 3.3. Налаштування запікання світла

На рисунку 3.3. зображені настройки LightMaper. Для запікання світла було вибрано дозвіл світловий карти 1024 х 1024 пікселів.

Direct Samples - 32. Це кількість зразків (шляхів), знятих з кожного текселя. Цей параметр контролює кількість зразків, яку Progressive Lightmapper використовує для розрахунків прямого освітлення. Збільшення цього значення може поліпшити якість світлових карт, але збільшує час випічки.

Також для підвищення якості запікаємо світла, потрібно відключити стискання світлових карт, так як стиснення буде вибрано окремо, після процесу запікання.

Bounces - 2. Це кількість відскоків світла при трасування шляхів. Для більшості сцен досить 2-ух відскоків.

Потрібно не забувати, що запікання світла працює, тільки зі статичними об'єктами на сцені, тому встановлюємо «Static» для кожного об'єкта на сцені, який не буде піддаватися переміщенню або трансформації.

Також джерело освітлення повинно бути виставлено в режим запікання.

Після проведення запікання світла отримуємо результат, зображений на рисунку 3.4.

Рис. 3.4. Сцена після запікання світла

Зараз на сцену не впливає не одне джерело світла, але приміщення здається освітленим і більш якісним, в порівнянні з освітленням в реальному часі.

Після запікання, ми можемо побачити інформацію про кількість згенерованих світлових карт і обсяг пам'яті, зайнятий цими картами, на рисунку 3.5.

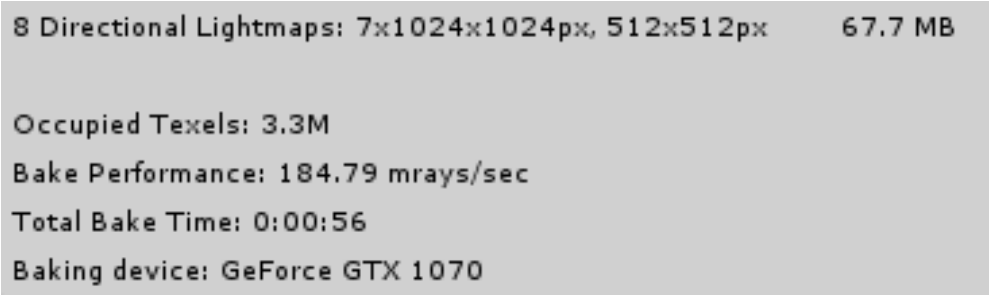

Рис. 3.5. Інформація про запечені світлові карти

Приклад самої світлової карти зображено на рисунку 3.6.

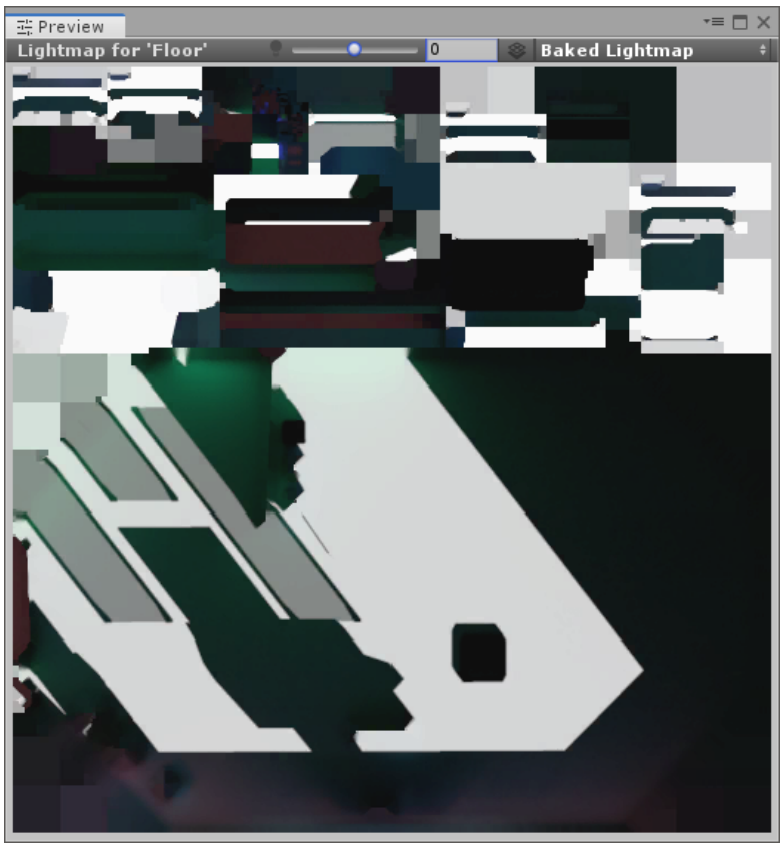

Рис. 3.6. Обчислена світлова карта

Було отримано 8 світлових карт, загальним об'ємом 67,7 Мб. Це дуже великий обсяг, для такої невеликої сцени. Спочатку було відключено стиснення світових карт, тому всі текстури були згенеровані в форматі RGB 24 bit. Для мобільних пристроїв слід використовувати інший формат стиснення.

Для Android часто використовуються наступні формати стиснення, але це не весь список доступних форматів (таблиця 3.1).

Таблиця 3.1 - Характеристика форматів стиснення текстур

1	2	3	4
Формат	Опис	Розмір текстури 256 x 256 пікселів	Підтримувана платформа
RGB Compressed ETC	Стисла RGB текстура. Це формат стиснення текстур за замовчуванням для текстур без альфа-каналу для проектів під управлінням Android.	32 КБ (4 біта на піксель)	Android, iOS, tvOS. Примітка: ETC1 підтримується всіма GPU OpenGL ES 2.0. Він не підтримує альфа.
RGB Compressed ETC2	Стисла RGB текстура.	32 КБ (4 біта на піксель)	Android (OpenGL ES 3.0) Примітка: На платформах Android, які не підтримують ETC2, текстура розпаковується під час виконання в форматі, зазначеному в Build Settings (Настройки збірки) в ETC2.

Закінчення таблиці 3.1

1	2	3	4
RGB(A) Compressed ASTC	Текстура стисненого блоку змінного розміру RGB або RGBA Texture.	10x10: 1,28 біта на піксель (10,56 КБ для текстур розміром 256x256). 8x8: 2 біта на піксель (16KB для текстури 256x256); 6x6: 3,56 біта на піксель (28,89 КБ для текстури 256x256).	tvOS (всі), iOS (A8), Android (PowerVR 6XT, Mali T600, Adreno 400, Tegra K1).
RGB Compressed PVRTC 2 bits	RGB текстура високого стиснення. Низька якість, але менший розмір, що призводить до підвищення продуктивності.	16 КБ (2 біта на піксель)	Android (PowerVR), iOS, tvOS.

Для стиснення світових карт був обраний формат RGB (A) Compressed ASTC 6x6 block. Після стиснення загальний обсяг текстур став 8.7 МБ. Це менше в 7.7 разів у порівнянні з попереднім результатом.

У режимі перегляду світлових карт, можна побачити, як тінь накладається на об'єкти сцени (рисунок 3.7).

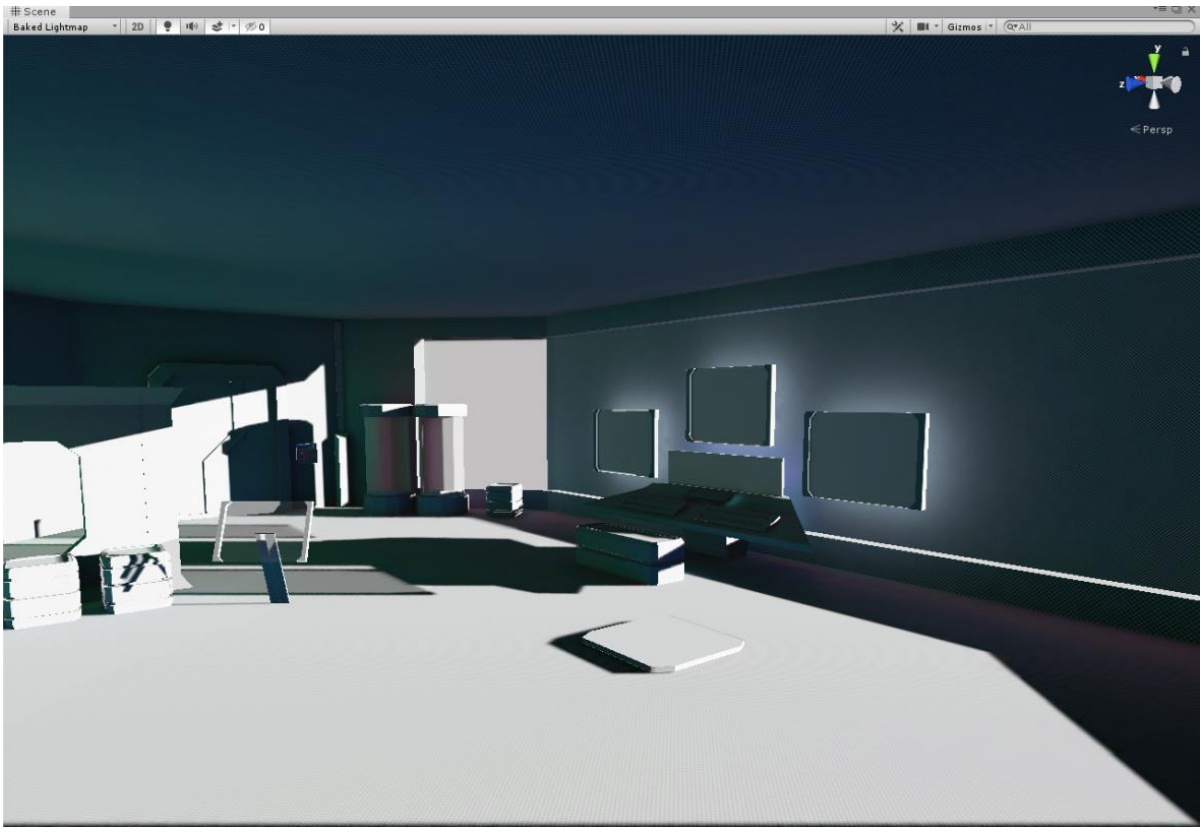

Рис. 3.7. Режим перегляду світових карт

Проведемо аналіз продуктивності після процесу запікання світла. Середній час, витрачений на відображення одного кадру, зменшився з 4 мс до 2.5 мс. Використання часу процесора,яке витрачено на відображення одного кадру, зменшилось з 28.1% до 13.9%. Середня частота кадрів збільшилася до 450. Основні показники використання CPU зображені на рисунку 3.8.

За отриманими результатами, можна зробити висновок, як освітлення впливає на загальну продуктивність гри. У більш об'ємних сценах з великою кількістю джерел освітлення, оптимізація просто необхідна, щоб процес гри був плавним і приємним для гравця.

Також для того, щоб не використовувати зайві обчислювальні ресурси процесора, слід обмежити максимальну частоту кадрів. На багатьох мобільних пристроях, частота екрану дорівнює 60 Гц. Замість того, щоб обробляти 400 кадрів в секунду, доцільно встановити обмеження до 60 кадрів в секунду. Це обмеження називається вертикальною синхронізацією, коли кадрова частота синхронізується з частотою вертикальної розгортки

монітора. Це хороший спосіб зменшити навантаження на пристрій, якщо в грі не потрібна висока точність управління. У таких випадках, чим більше частота кадрів, тим більше точно відчувається управління в грі.

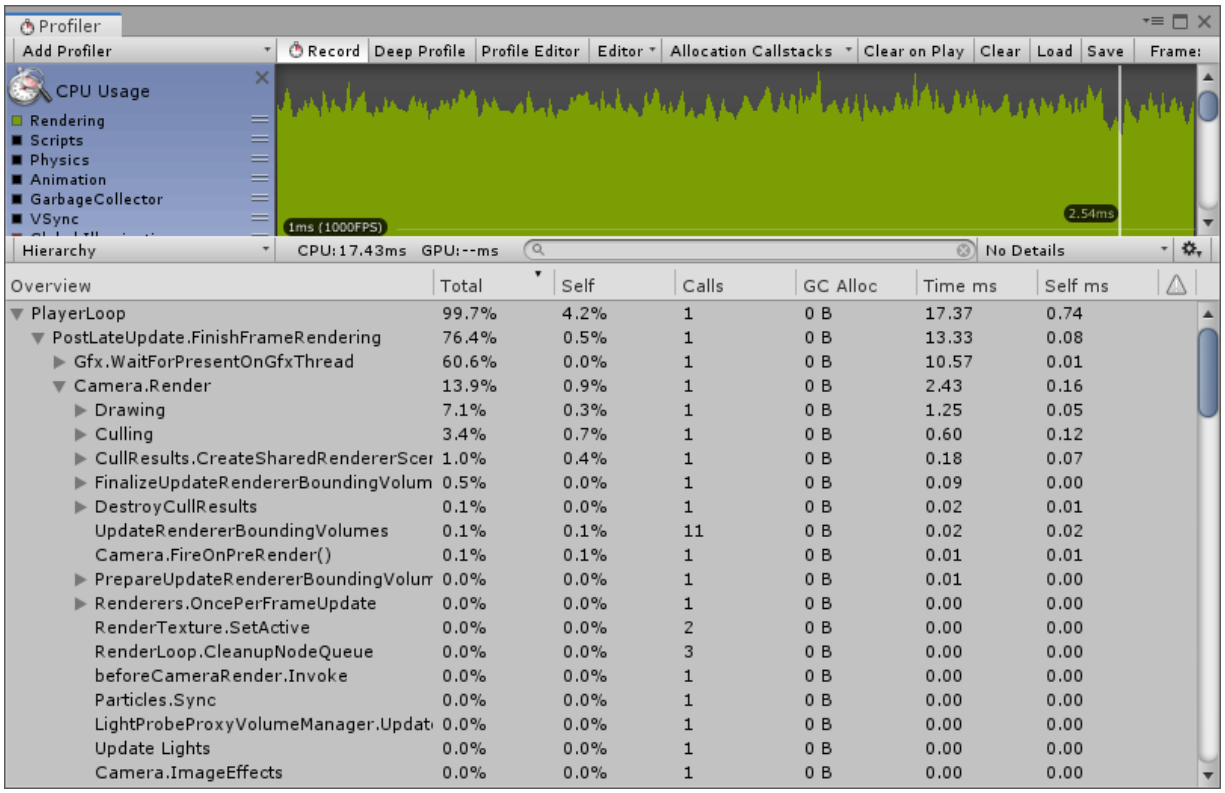

Рис. 3.8. Аналіз використання CPU

Після того, як ми провели запікання світла на статичних об'єктах, ми втратили джерела світла реального часу, тому всі динамічні об'єкти на сцені не будуть висвітлені. Динамічним об'єктом є сам персонаж, який відтворює анімації і переміщається по сцені. Для вирішення цієї проблеми були використані зонди освітлення. Вони являють собою невеликі сфери, що вбирають в себе інформацію про висвітлення навколишнього простору.

Як і на світових картах, світлові зонди зберігають "запечену" інформацію освітлення на сцені. Різниця полягає в тому, що в той час як карти освітлення зберігають інформацію про попадання світла на поверхню сцени, світлові зонди зберігають інформацію про проходження світла через порожній простір сцени. Приклад розташування світових зондів зображено на рисунку 3.9.

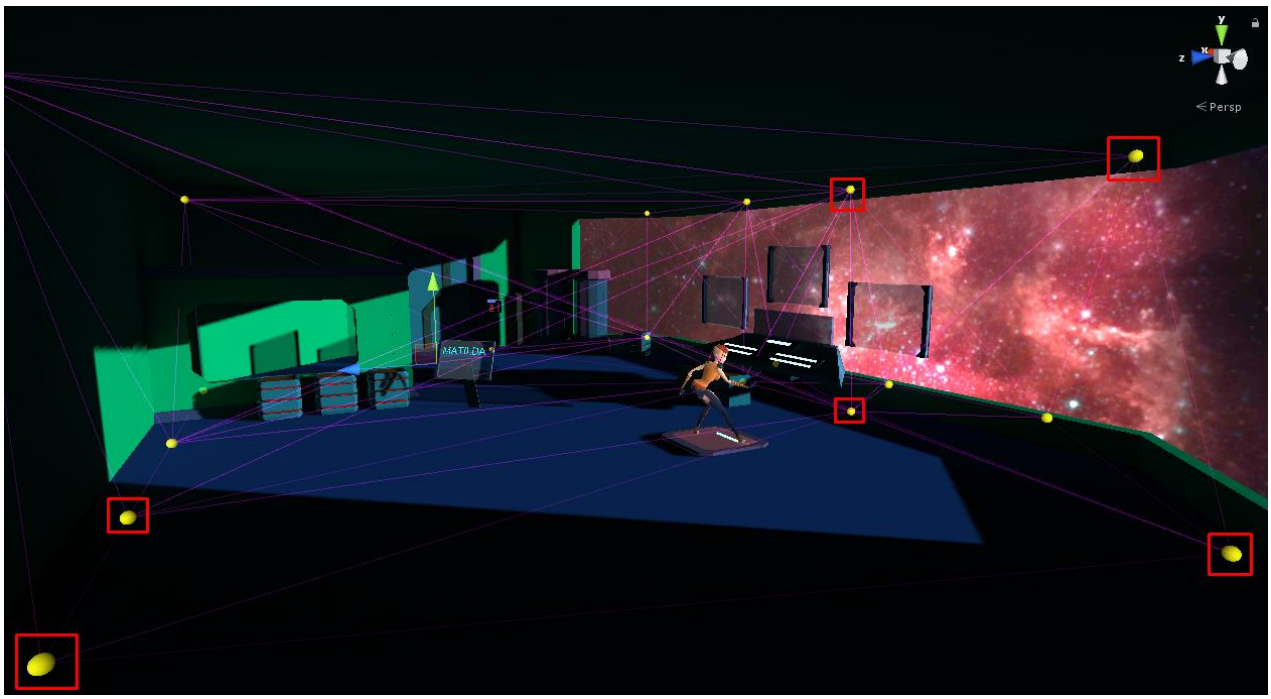

Рис. 3.9. Світові зонди

Наступним пунктом оптимізації є використання LOD системи. Під час гри, на сцені може відображатися велика кількість моделей, що складаються з безліч полігонів і вершин. Чим більше вершин містить сцена, тим більше обчислень проводить CPU для побудови сцени. LOD`s дозволяють зменшити кількість вершин за допомогою візуального сприйняття віддалених об'єктів. Чим далі знаходиться об'єкт від гравця, тим менше деталізована копія цього об'єкта відображається під час рендерінгу.

Для прикладу була взята модель каменю, яка є перешкодою під час гри. Генерація даного каменю відбувається на певній відстані від гравця, яка скорочується з часом. Отже, під час генерації нам не потрібна високо деталізована копія, а тільки тоді, коли гравець підбіжить до моделі на відстань задану LOD налаштуваннями.

Самий деталізований LOD містить 285 вершин. Кожен наступний містить в 2 рази менше вершин, ніж попередній.

Для створення рівнів LOD була використана програма Blender 2.8. Unity зручна тим, що якщо в 3D редакторі, експортована модель в форматі FBX була названа відповідною назвою LOD (LOD0, LOD1), то движок сам

додає до цієї моделі LOD групу. Її залишається тільки налаштувати. Приклад LOD групи зображений на рисунку 3.10.

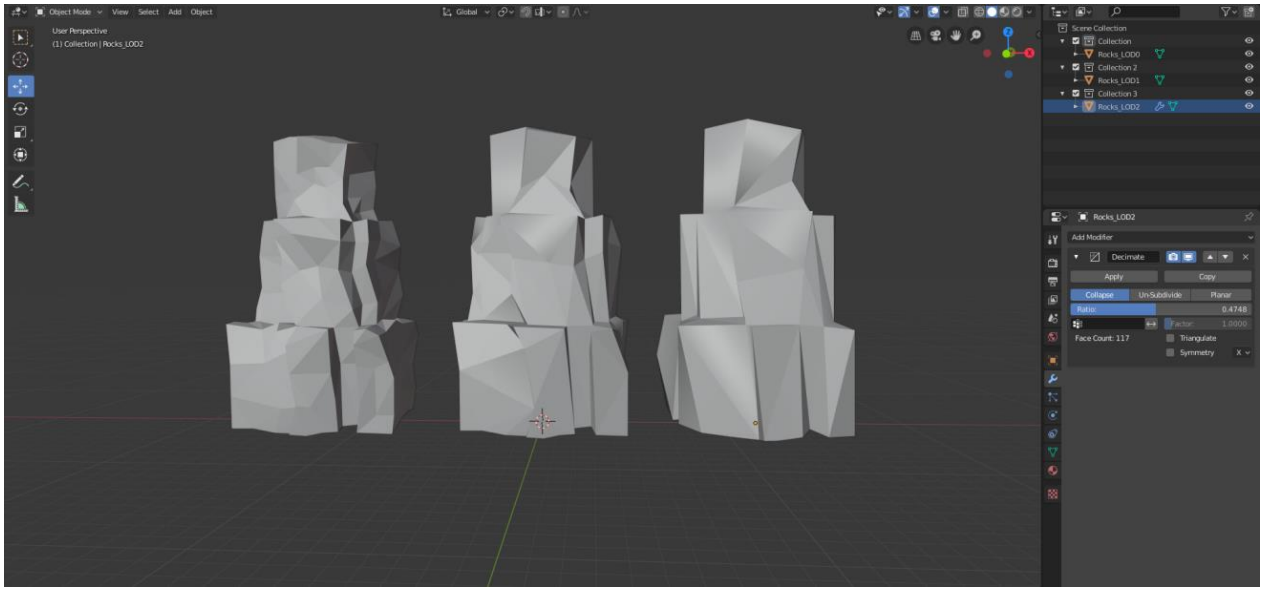

Рис. 3.10. LOD група

Меню налаштування LOD групи зображено на рисунку 3.11. На горизонтальній смужці під значком камери видно різні LOD рівні (LOD: 0, LOD: 1, LOD: 2, і т.д.). Відсоток на смужці LOD означає частку глибини в піраміді видимості, на якій даний LOD рівень стає активним. Можна змінити значення відсотків, перетягнувши одну з вертикальних ліній, що розділяють смужку.

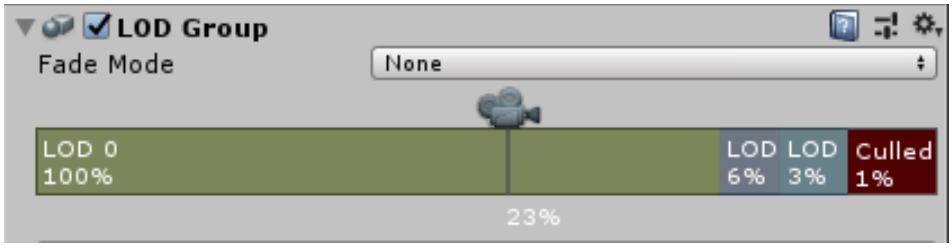

Рис. 3.11.Налаштування LOD групи

Також можна додавати і видаляти LOD рівні на смужці, клікнувши правою кнопкою миші і вибравши “Insert Before” або “Delete” з контекстного меню.

LOD група була налаштована таким чином, щоб кожні 50 метрів відбувалася активація наступного рівня LOD. Результат роботи можна побачити на рисунку 3.12.

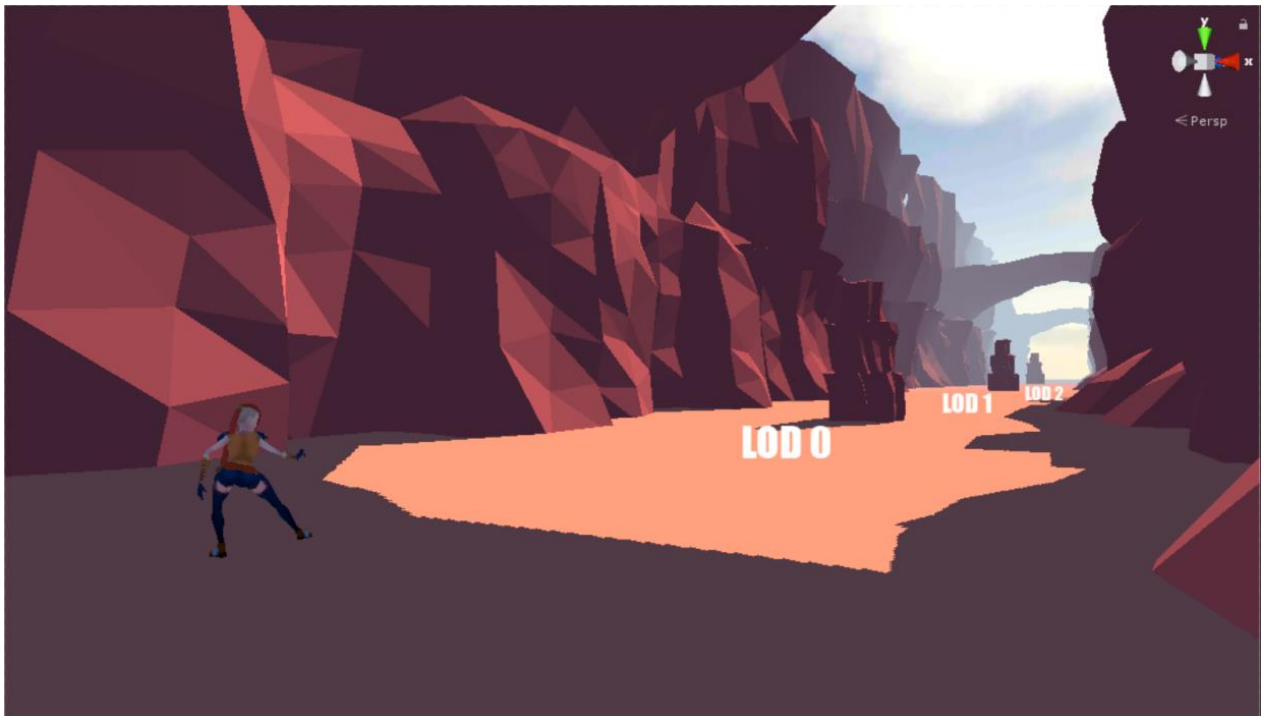

Рис. 3.12. Результат роботи LOD групи

Кількість вершин кожного об’єкту LOD групи описано у таблиці 3.2.

Таблиця 3.2. - Опис кількості вершин та полігонів моделей.

LOD рівень	Кількість вершин	Кількість полігонів
LOD0	285	533
LOD1	140	248
LOD2	70	117

Тим самим, застосувавши LOD групи до всіх моделей, можна отримати велике скорочення вершин, не зіпсувавши візуальну складову сцени. Статистика кількості вершин і полігонів сцени зображена рисунку 3.13. Зліва можна побачити статистику без використання LOD груп, праворуч навпаки.

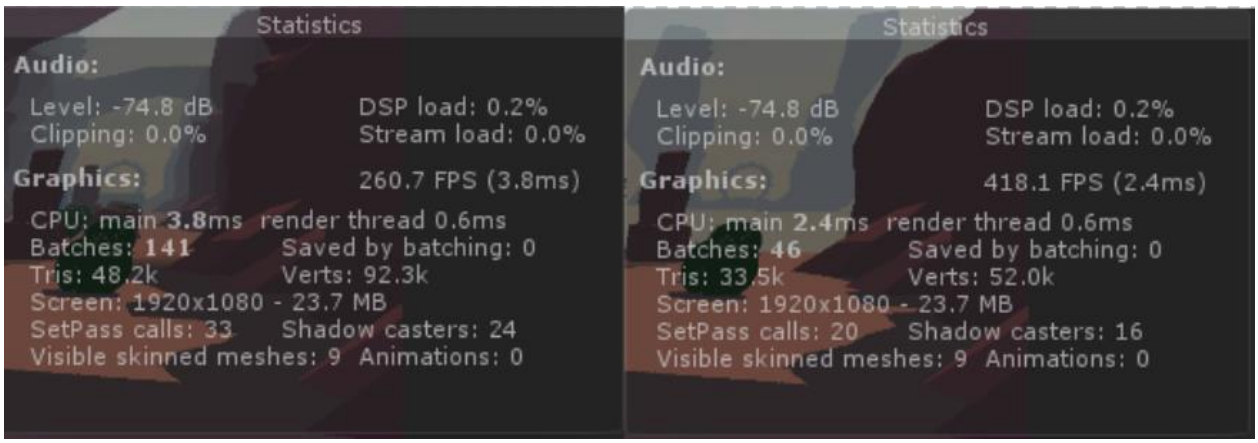

Рис. 3.13. Статистика рендерінгу сцени

Результати показали, що вдалося досягти зниження кількості вершин майже в двоє, а отже збільшити частоту кадрів. Порівняння показників відображено у таблиці 3.3.

Таблиця 3.3 - Порівняльний аналіз використання LOD групи.

	Без LOD груп	З LOD групами
Вершин	92300	52000
Полігонів	48200	33500
Частота кадрів в секунду	260.7	418.1
Час на відображення одного кадру GPU (мс)	3.8	2.4
Час на відображення одного кадру CPU (мс)	3.8	2.4

Наступний етап оптимізації передбачає позбавлення від великої кількості матеріалів. Це робиться для того, що зменшити кількість викликів відтворення. В процесі рендерінгу, якщо кожен елемент інтерфейсу або сцени має окрему текстуру, то потрібно викликати метод відтворення для кожного зображення. На кожен виклик відтворення потрібен час, що робить процес рендерінгу довшим.

Для цього використовують текстурні атласи. На кожен об'єкт використовується однаковий матеріал, який містить текстуру всіх елементів сцени. Вона має більший розмір, але візуалізується за один виклик методу відтворення.

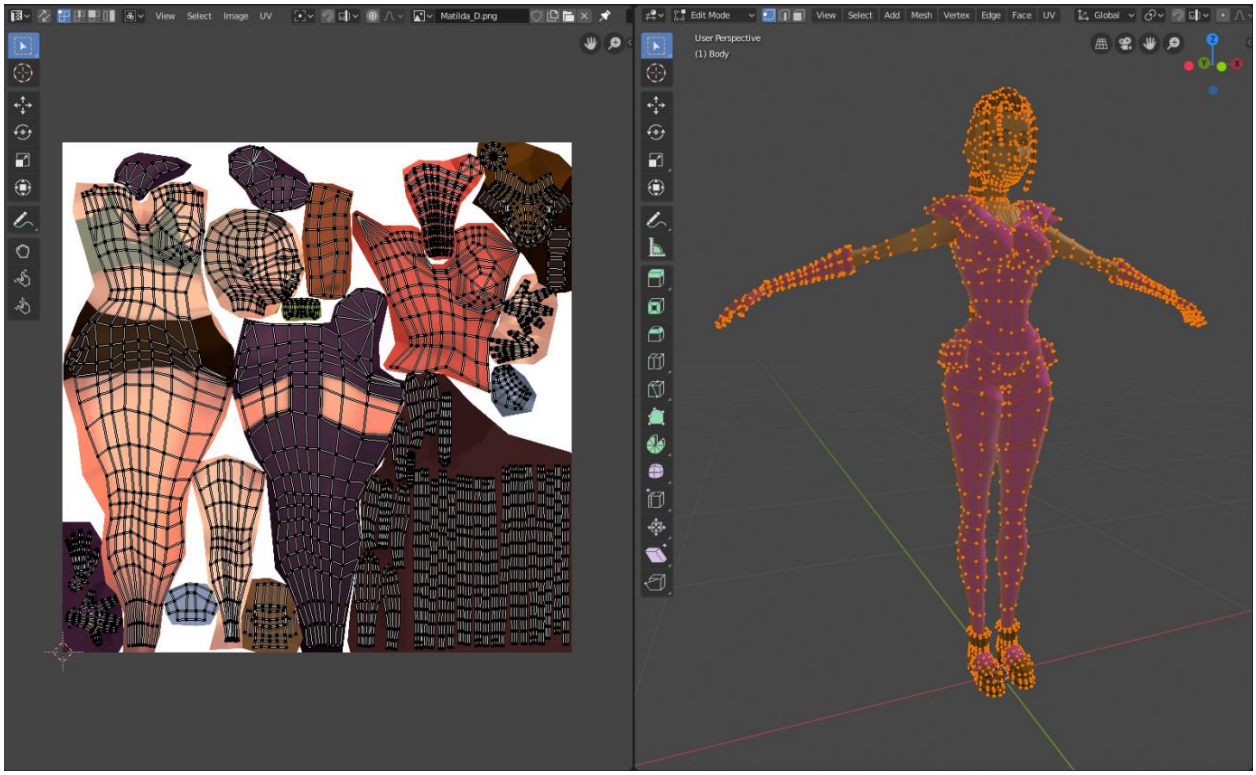

Рис. 3.14. Розгортка моделі персонажа

Для того щоб реалізувати текстурний атлас, необхідно в будь-якому 3D редакторі правильно налаштувати розгортку моделі. Кожна площина моделі буде займати певну позицію на текстурному атласі. Приклад розгортки моделі персонажа зображений на рисунку 3.14.

Так як гра зроблена в низькополігональному стилі з одноколірними текстурами, атлас текстур буде виглядати як набір різних кольорів, на який будуть накладати розгортки всіх моделей сцени. Приклад текстурного атласу зображений на рисунку 3.15.

Проведемо аналіз роботи текстурного атласу. На рисунку 3.16 зображена статистика сцени без текстурного атласу і з ним.

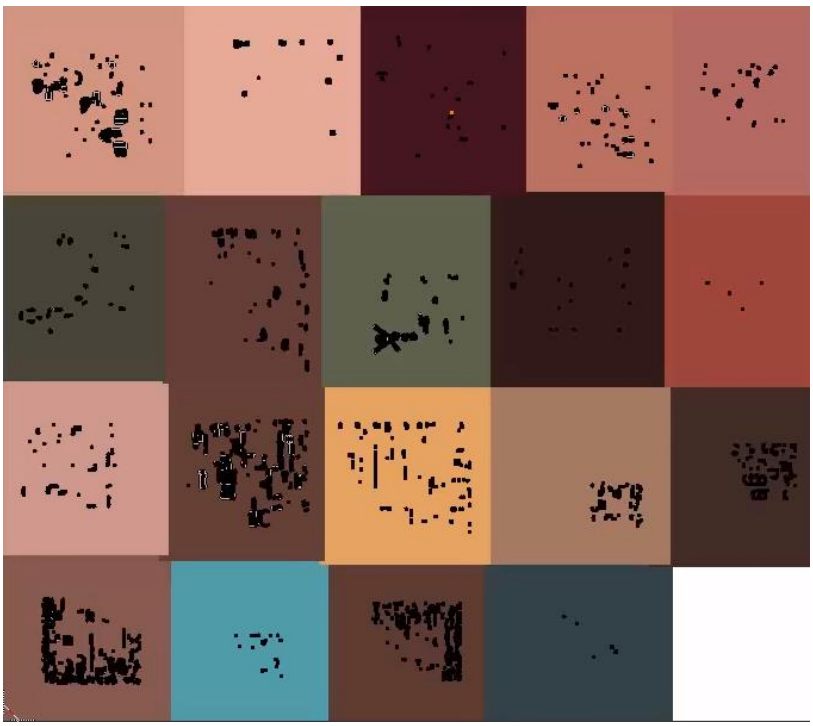

Рис. 3.15. Текстурний атлас

Можна помітити, що кількість викликів методу відтворення скоротилося з 28 до 5. Цей результат показує, що текстурний атлас скорочує загальний витрачений час на рендерінг одного кадру, що підвищує частоту кадрів в секунду.

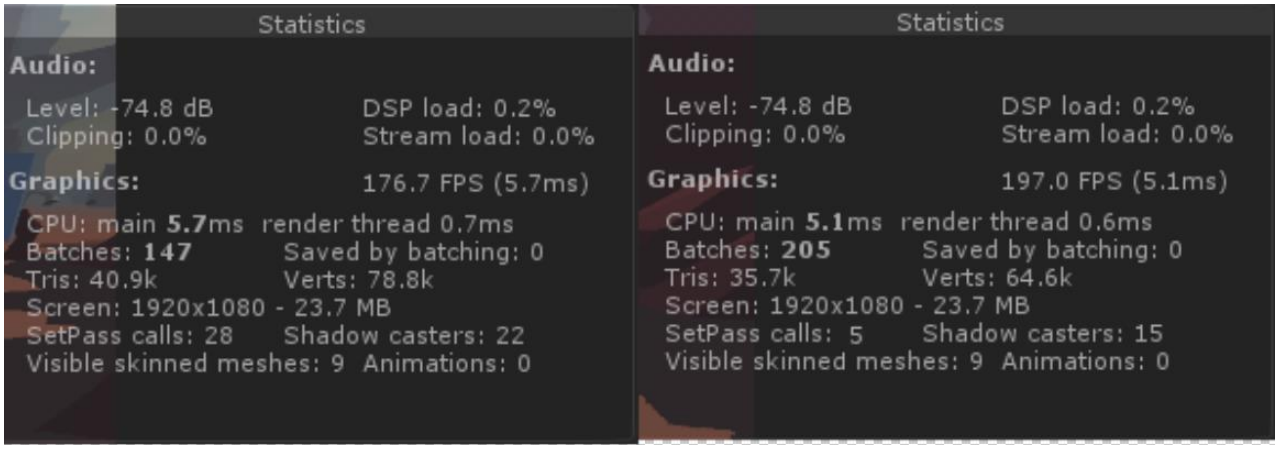

Рис. 3.16. Статистика рендерінгу сцени

Не мало важливим пунктом є використання правильних шейдерів. Для різних платформ рекомендується використовувати спеціальні шейдери.

Шейдер (англ. Shader - затіняюча програма) - це програма для GPU, яка використовується в тривимірній графіці для визначення остаточних параметрів об'єкта або зображення, може включати в себе опис поглинання і розсіяння світла, накладання текстури, відображення і заломлення, затінення, зміщення поверхні і безліч інших параметрів.

Шейдери можна писати свої на спеціальній мові, щоб досягти необхідного результату, або використовувати стандартні, які запропоновані самим движком.

Вбудовані шейдери можна відсортувати в міру збільшення складності.

1) Unlit. Це просто текстура, на яку не впливає будь-яке висвітлення.
2) VertexLit.
3) Diffuse.
4) Normal mapped. Це трохи більше ресурсомісткий шейдер, ніж Diffuse: в ньому додана ще одна текстура (карта висот) і кілька додаткових шейдерних інструкцій.
5) Specular. У цьому доданий розрахунок дзеркальних відблисків.
6) Normal Mapped Specular. Знову ж таки, дещо ресурсомісткий шейдер в порівнянні зі Specular.
7) Parallax Normal mapped. У цьому додані розрахунки параллаксового застосування карт нормалей.
8) Parallax Normal Mapped Specular. У цьому додані розрахунки і параллаксового застосування карт нормалей і дзеркальних відблисків.

Крім того, в Unity існує кілька спрощених шейдерів для використання на мобільних платформах, вони розташовані в категорії "Mobile". Ці шейдери працюють і на інших платформах, якщо необхідні спрощення, які в них внесені (наприклад, усереднені відображення, немає підтримки кольорів для кожного матеріалу і т.д.), можна використовувати їх.

Нижче наведені деякі приклади змін, характерних для мобільних шейдерів:

1) Матеріального кольору або основного кольору для тонування

шейдера не існує.

2) Для шейдерів, які отримують звичайну карту, використовуються плитка та зміщення від базової текстури.

3) Шейдери частково не підтримують AlphaTest або ColorMask.

4) Обмежена функціональність і підтримка освітлення - наприклад, деякі шейдери підтримують тільки одне направлене світло.

Так як нам необхідний рендерінг тільки базового кольору, без текстур відображень, відблисків і нормалей, для мобільних пристроїв слід вибирати шейдер Mobile / Diffuse або Legasy Shader / Diffuse.

Для персонажа був обраний шейдер з категорії «Legasy Shader», який зображений на рисунку 3.17.

Рис. 3.17. Налаштування матеріалу моделі

Так як в ігровій сцені всі об'єкти динамічні, через генерації рівня в процесі гри, то звичайний спосіб запікання світла не підходить для вирішення проблем з освітленням реального часу. Для цього було прийнято рішення використовувати пряме джерело світла з маленьким дозволом і жорстким типом тіней.

Важливим налаштуванням освітлення, в даному випадку, є дистанція відображення тіней. Значення цього параметра було виставлено на 50. Тобто тіні від об'єктів, які знаходяться на відстані більше, ніж 50 метрів, обчислюватися не будуть.

3.2. Тестування і проведення оптимізації коду

Гра являє собою нескінченну смугу перешкод, яка генерується в міру проходження. Дана механіка гри вимагає оптимізацію з боку генерації рівня. Спочатку було прийнято рішення використовувати пул об'єктів.

Instantiate () і Destroy () є корисними і необхідними методами в процесі гри. Кожен з них зазвичай вимагає мінімального процесорного часу.

Однак для об'єктів, створених в процесі гри, які мають короткий термін служби і руйнуються в величезних кількостях в секунду, процесор повинен виділяти значно більше часу.

Крім того, Unity використовує функцію "прибирання сміття", щоб звільнити невикористану пам'ять. Повторні виклики функції Destroy() часто запускають цю задачу, і вона має здатність сповільнювати роботу процесорів і вводити паузи в ігровий процес.

Така поведінка критично важливо в умовах обмежених ресурсів, таких як мобільні пристрої і веб-збірки.

Пул об'єктів - це коли ми попередньо інстанціюємо всі об'єкти, які нам знадобляться в будь-який конкретний момент перед початком гри - наприклад, під час завантаження екрану. Замість того, щоб створювати нові

об'єкти і знищувати старі в процесі гри, гра повторно використовує об'єкти з "пулу". Приклад коду наведений у додатку А.

В даному проекті пул об'єктів використовується для зберігання сегментів рівня. Всі сегменти завантажуються в оперативну пам'ять при завантаженні сцени і встановлюються в чергу. Коли сегмент виявляється позаду гравця і знаходиться поза зоною видимості, він не знищується, а переміщається в початок черги, тим самим утворюючи нескінченну смугу. Схематичний приклад позиціонування сегментів зображений на рисунку 3.18.

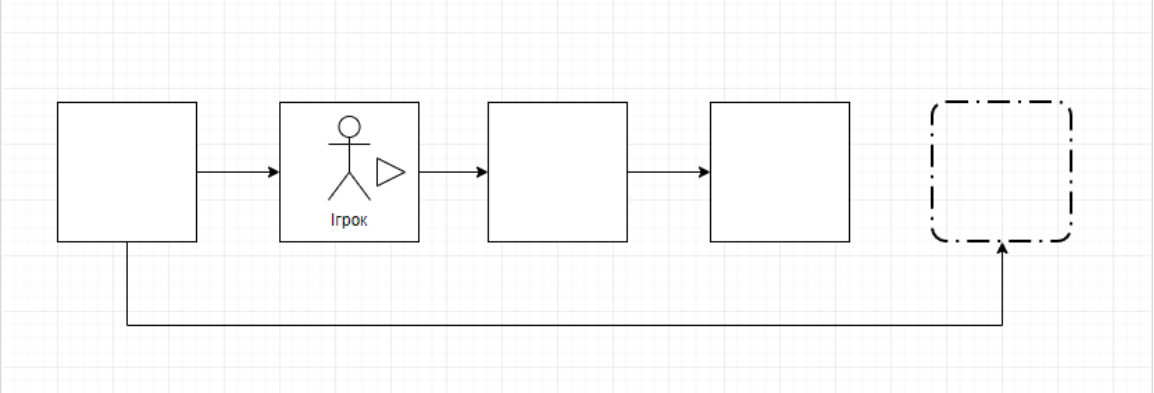

Рис. 3.18. Схема черги сегментів

Кожен сегмент містить свою логіку генерації перешкод. Всі перешкоди вибираються випадковим чином, виключаючи безвихідний випадок програшу.

Приклад згенерованої смуги і одиничний сегмент показаний на рисунку 3.19.

Зміна рівня проводиться через ігровий портал, який візуально розділяє локації. Принцип роботи полягає в тому, що на сцені присутні дві камери, які обробляють зображення на різних шарах. На кожному з шарів знаходиться свій рівень.

Портал містить рендерінг текстуру на яку надходить інформація, зчитана з другої камери. Коли гравець проходить через портал, камери змінюються між собою і гравець починає бачити наступний рівень, коли

попередній ховається. Тим самим досягається ефект безшовної зміни локацій.

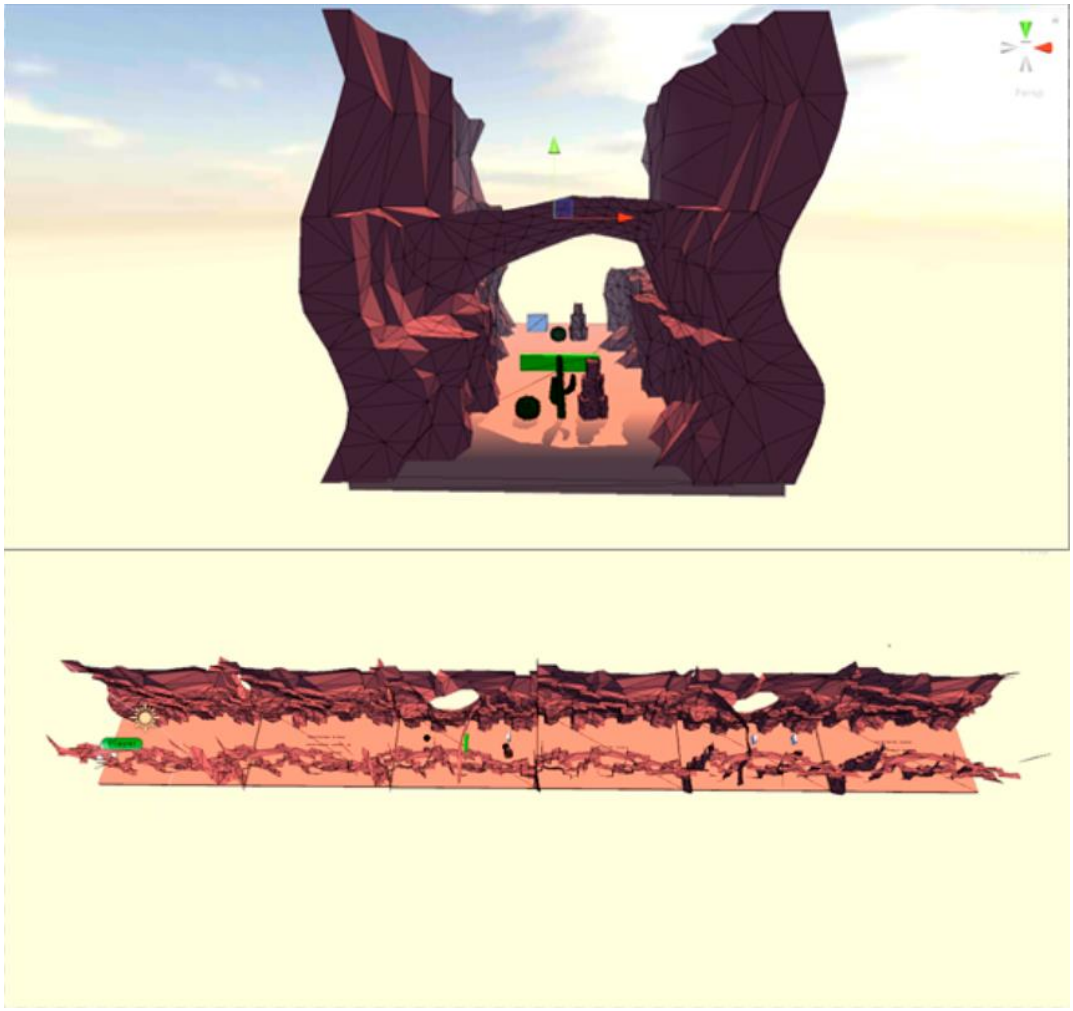

Рис. 3.19. Приклад згенерованої смуги

Вид гравця і вид в режимі розробника зображений на рисунку 3.20.

Кожне зіткнення в грі обробляють колізії. Колізії можуть складатися з примітивів або повторювати сітку моделі. Чим більш деталізовану колізію містить модель, тим сильніше це позначається на продуктивності гри. Потрібно уникати складних форм, для спрощення розрахунків.

Основні колізії встановлені на персонажа і перешкоди, для визначення зіткнення гравця з ними.

Перешкоди являють собою складні об'єкти, наприклад кактуси, різні камені. Нам не потрібно повторювати сітку їх моделі, досить

використовувати кубічні колізії, так як гра складається з трьох ліній, а гравець біжить в одному напрямку. Приклад такої колізії зображений на рисунку 3.21.

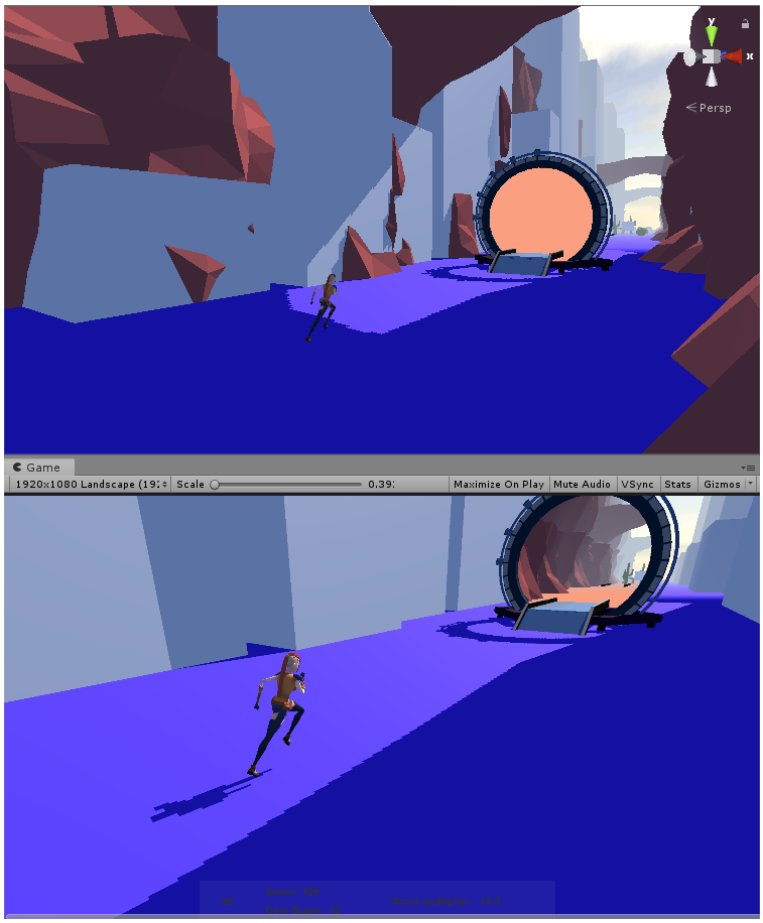

Рис. 3.20. Вид гравця та розробника

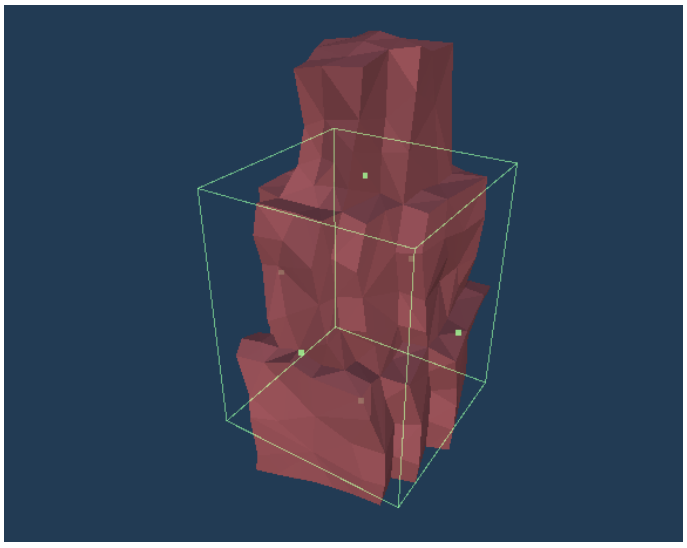

Рис. 3.21. Приклад примітивної колізії

3.3. Кінцеве тестування проекту

За допомогою профілювальника проведемо основне тестування програми. Для швидкого пошуку проблем, в режимі тимчасової шкали, можна перевірити, чи є зниження частоті кадрів або знайти перевантажені місця.

Профілювальник зручно показує витрату часу на відображення одного кадру, а також показує час роботи кожного з компонентів, який викликається кожен кадр. На рисунку 3.22 зображено використання ресурсів процесора.

В процесі гри не було помітно зниження частоти кадрів. Середня витрата часу на обробку одного кадру становить близько 1.2 мс.

Функція Camera.Render, яка відповідає за відображення графічної частини, використовує 0.443 мс часу процесора.

Всі показники FPS не менше середніх показників частот сучасних моніторів мобільних пристроїв, що дозволяє гравцеві грати на максимально можливій частоті кадрів.

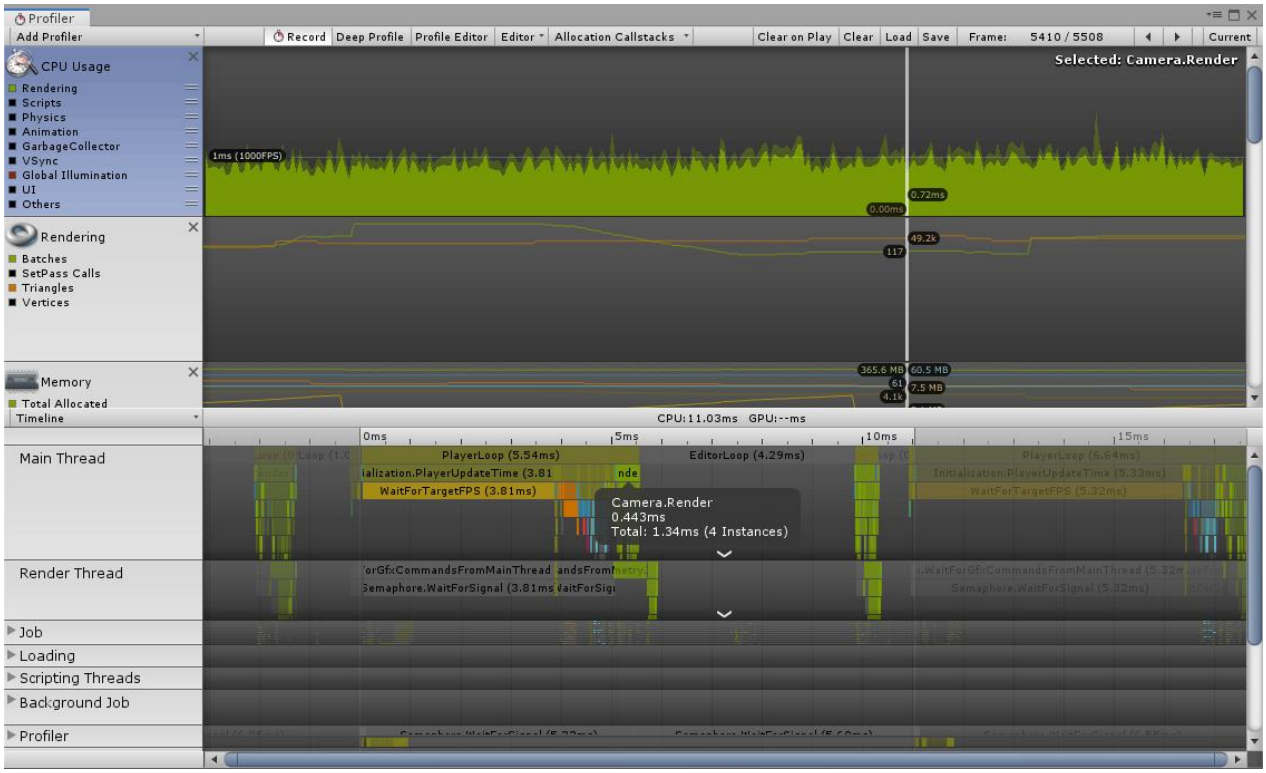

Рис. 3.22. Аналіз використання CPU

Тести проводяться на самій навантаженій сцені, з найбільшою кількістю текстур і моделей, що дозволяє знайти нижню межу продуктивності.

Далі проведемо аналіз пам'яті. На рисунку 3.23 зображено кількість використовуваної пам'яті додатком і окремими її складовими.

Були обрані усереднені показники, так як на кожному кадрі результати можуть відрізнятися, в залежності від стану сцени, стану генерації рівня і місця розташування гравця.

Загальний обсяг зайнятий текстурами становить 367,2 МБ. При цьому моделі займають 7.6Мб, матеріали - 126 Кб, анімаційні кліпи - 1.4 МБ.

Також можна визначити кількість об'єктів, які використовуються на сцені і в загалі.

Всього використовується 312 текстур, 100 моделей, 61 матеріал. 13 анімаційних кліпів. Всього ігрових об'єктів на сцені і на поточному кадрі - 341, загальна кількість об'єктів на сцені - 1515, загальна кількість об'єктів в грі - 4085.

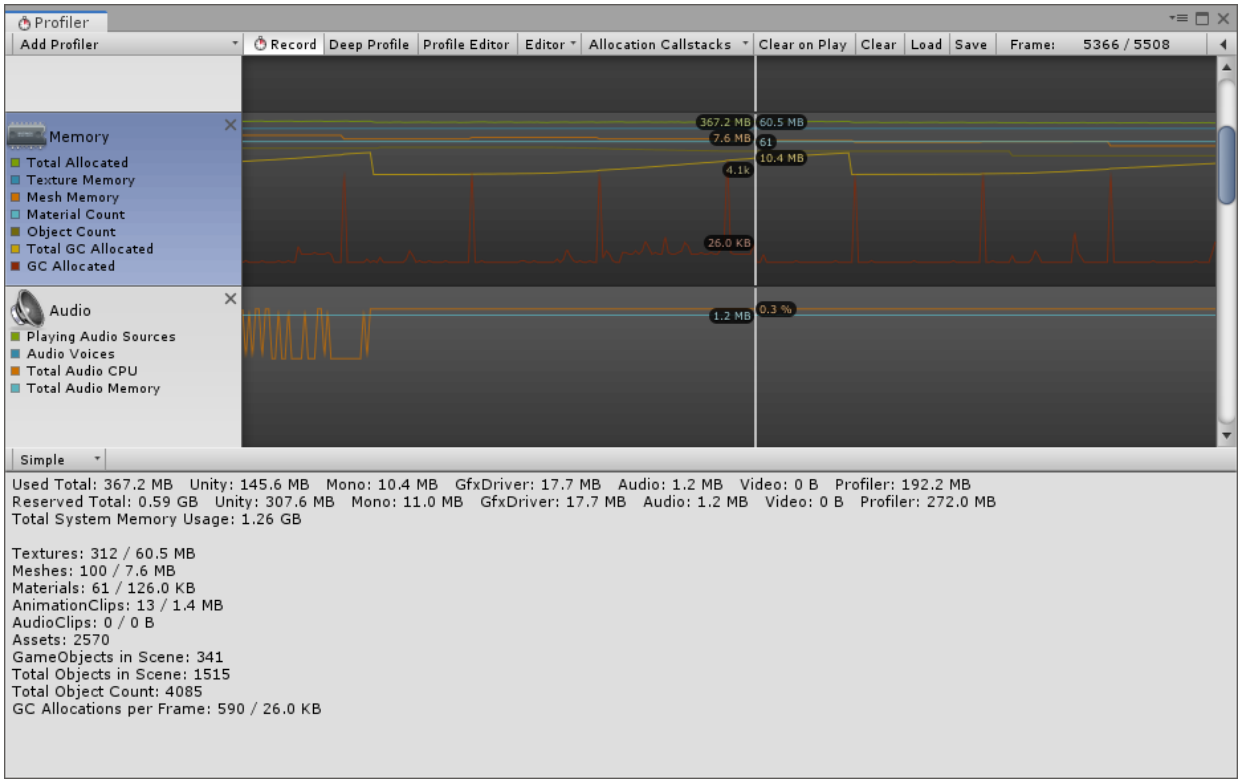

Рис. 3.23. Аналіз використання пам’яті

Не мало важливим показником є обсяг пам'яті, виділений на відображення одного кадру.

Цей обсяг варіюється від 14 до 69 Кб. Різкий стрибок виділеної пам'яті обумовлюється генерацією рівня через певні проміжки часу. Це стрибок не є критичним і не впливає на продуктивність системи.

Наступні показники, на які необхідно звернути увагу, це показники рендерінгу. Результати тестів рендерінгу сцени зображені на рисунку 3.24.

Тут можна побачити такі значення, як кількість вершин моделей, полігонів, викликів методів відтворення, зайнятий обсяг відео пам'яті, обсяг виділений на зберігання текстур, а також кількість об'єктів, які піддалися операції пакетування.

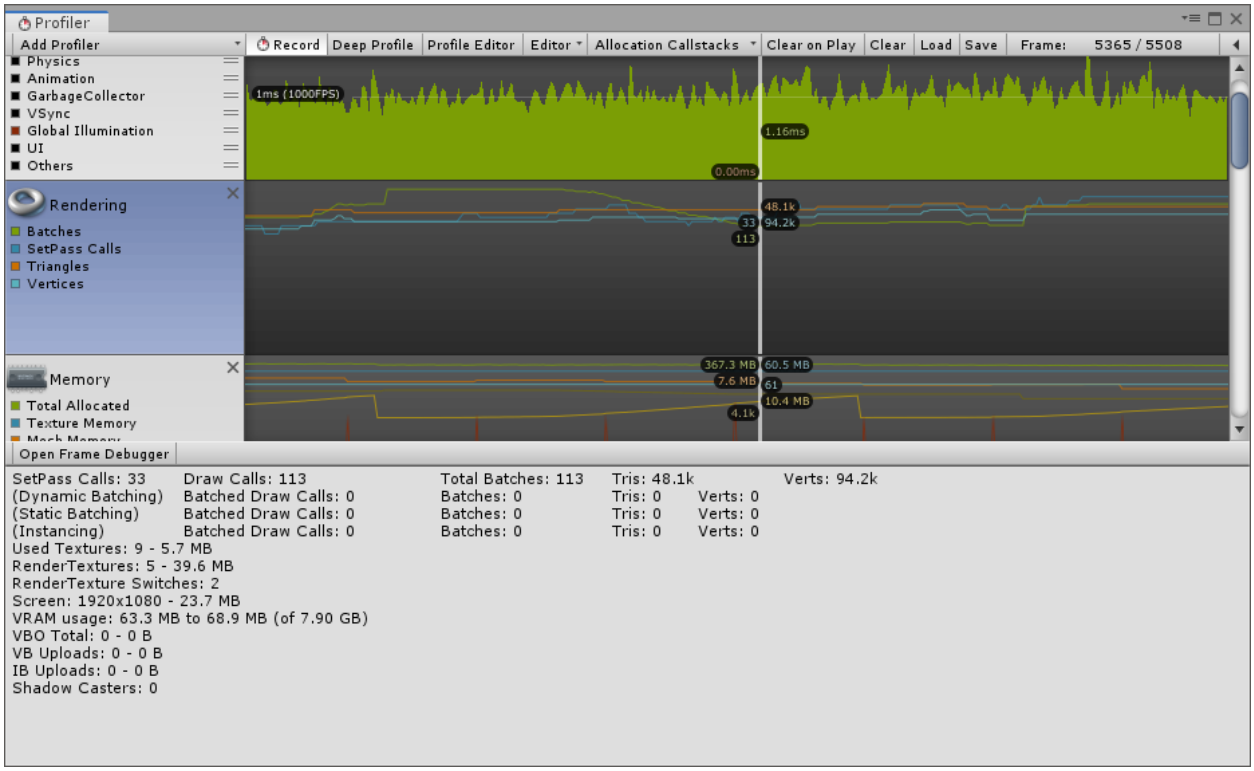

Рис. 3.24. Аналіз рендерінгу сцени

Unity використовує дві техніки пакетування:

1) Статична пакетна обробка: об'єднання статичних (тобто нерухомих) об'єктів в великі сітки і рендерінг їх одним викликом відтворення.

2) Динамічне пакетування: для досить маленьких сіток, трансформує їх вершини на процесорі, групує безліч подібних сіток разом і відображає їх одним викликом методу відтворення.

Динамічне пакетування автоматично об'єднує рухомі об'єкти в один DC, якщо вони використовують загальний матеріал і відповідають ряду інших критеріїв. Динамічне пакетування застосовується автоматично і не вимагає додаткових дій з боку розробника.

Динамічне пакетування пов'язано з додатковим навантаженням для кожної вершини, так що він застосовується лише до моделей, що містить менше 900 вершин в сумі.

Якщо шейдер використовує Vertex Position, Normal і єдиний UV, то можна об'єднувати до 300 вершин. Якщо шейдер використовує Vertex Position, Normal, UV0, UV1 і Tangent, то тільки 180 вершин.

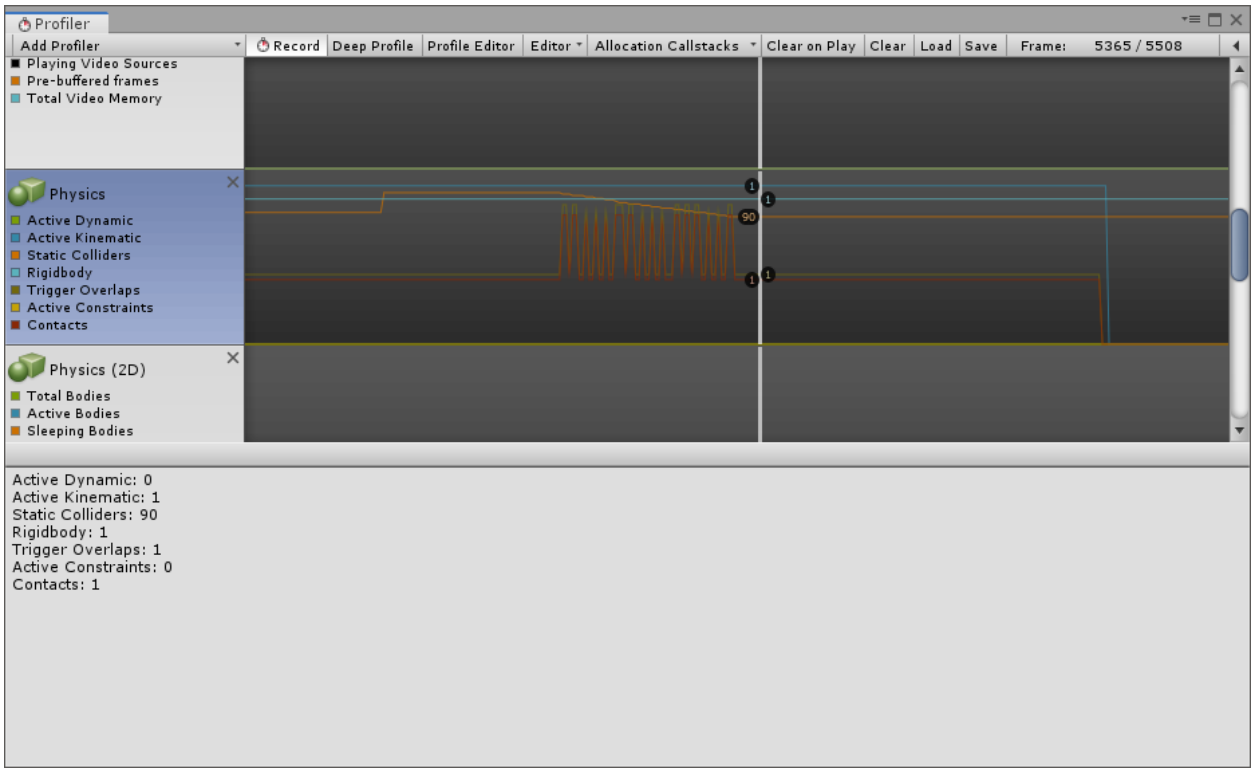

Рис. 3.25. Аналіз фізики

Статичне пакетування дозволяє движку знизити кількість DC для геометрії будь-якого розміру, якщо вона не рухається і використовує

загальний матеріал. Статичне пакетування більш ефективне, ніж динамічне. Слід використовувати статичну пакетування, щоб знизити навантаження на CPU.

В аналізі фізики, який можна побачити на рисунку 3.25, Зображено кількість колізій і компонентів Rigidbody, а також кількість взаємодій в певному кадрі.

Всього об'єктів на сцені, які містять компонент колізії - 90, 1 об'єкт містить включений режим кінематики і не піддається гравітації і іншим фізичним взаємодіям. 1 об'єкт містить компонент Rigidbody, а саме сам персонаж гри.

Далі запустимо додаток на мобільному пристрої в збірці релізу, без засобів налагодження. На призначений для користувача інтерфейс виведемо частоту кадрів в секунду. Знімок гри зображений на рисунку 3.26.

Рис. 3.26. Знімок екрану мобільного пристрою

Гра йде на максимально підтримуваної частоті кадрів - 80. В процесі тестування гри на мобільному пристрої не було виявлено різке зменшення продуктивності. Зміна локацій відбувається плавно і швидко.

3.4. Висновки до розділу

1. Проведена діагностика додатку за допомогою засобів профілювання Unity, а також зроблені висновки щодо усунення проблем, які впливають на продуктивність гри.

2. Проведена оптимізація освітлення за допомогою технології по запіканню освітлення статичних об'єктів.

3. Налагоджена оптимальна конфігурація запікання освітлення, а також досліджений отриманий результат світлових карт.

4. Досліджено формати стиснення текстурних карт.

5. Проведена компресія текстурних карт відповідно до обраної платформи і необхідного результату.

6. Проведений тест рендерінгу сцени після процесу запікання освітлення.

7. Вирішено проблему освітлення динамічних об'єктів за допомогою зондів освітлення.

8. Проведена оптимізації моделей за допомогою технології LOD груп. Розроблена група моделей для використання даною технології. Налаштовані діапазони видимості кожного об'єкта групи.

9. Проведені порівняльні тести продуктивності після використання LOD груп.

10. Розроблені текстурні атласи для оптимізації кількості викликів методів відтворення на кожному кадрі.

11. Проведені тести продуктивності після використання текстурних атласів.

12. Досліджені стандартні шейдери Unity. Обрані оптимальні шейдери матеріалів для додатка.

13. Розроблено та реалізовано алгоритм нескінченної генерації рівня з урахуванням нюансів використання обчислювальних ресурсів.

14. Розроблений та реалізований спосіб безшовної зміни рівня за

допомогою шарів видимості.

15. Встановлено оптимальні колізії для взаємодії фізичних об'єктів.

16. Проведені остаточні тести продуктивності на цільовій платформі.

ВИСНОВКИ

1. У першому розділі проведений аналіз існуючих проблем оптимізації мобільних ігор і додатків.

2. Розглянуто типи проблем, причини їх появи.

3. У движку Unity присутні всі основні засоби діагностики і тестування додатків. Були розглянуті найбільш використовувані інструменти для виявлення проблем продуктивності.

4. Вивчено технології освітлення об'ємних сцен, види джерел світла, їх вплив на загальну продуктивність системи, а також досліджені способи оптимізації освітлення.

3. У другому розділі були детально розглянуті способи проведення оптимізації програми за допомогою засобів Unity.

4. Основне навантаження в 3D іграх викликає графічна частина, тому в більшій мірі були вивчені прийоми зменшення споживання обчислювальних ресурсів за допомогою попереднього розрахунку графіки, а також за допомогою технологій оптимізації в реальному часі, таких як LOD, Culling, MipMaps.

5. Досліджені способи оптимізації розрахунків, пов'язані з фізичними взаємодіями, такі як RigidBody, Collider.

6. У третьому розділі на прикладі реального проекту були застосовані способи оптимізації, які розглянуті у другому розділі.

7. Проведено тестування програми на наявність проблемних місць, а саме: зниження частоти кадрів, довге завантаження рівнів, великий обсяг використання відео і оперативної пам'яті, зниження продуктивності під час генерації рівня.

8. Застосовані способи оптимізації графіки та освітлення. Освітлення статичних об'єктів запечене в текстурні атласи. Налаштовані зонди освітлення для коректного освітлення динамічних об'єктів.

9. Перевизначені настройки стиснення текстур для Android пристроїв

на більш оптимальні, а також досліджені можливі формати стиснення для мобільних платформ.

10. Проведено оптимізацію моделей.

11. Розроблені текстурні атласи, для зменшення навантаження під час виклику методів відтворення.

12. Розроблені та реалізовані алгоритми генерації нескінченного рівня, а також алгоритми безшовної зміни локацій.

13. Проведені остаточні тести за допомогою засобів движка Unity і на мобільному пристрої.

14. Дослідивши аспекти оптимізації графіки, була виявлена проблема, через яку динамічні об'єкти не здатні відкидати тінь на об'єкти з запеченим освітленням. Дана проблема полягає в реалізації технології самого движка і вирішується за допомогою сторонніх рішень та шейдерів. У режимах джерела світла існує режим змішаного освітлення, але він не дозволяє динамічним об'єктам відкидати тінь, на повністю запечене світло.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Абляев М.Р., Аметов Ф.Р., Мевлют И.Ш., Адильшаева Э.И.. "Программа Blender как основная среда 3D моделирования для разработки игр в unity" Таврический научный обозреватель, №. 6 (11), 2016, 190-192с.

2. Аметов Ф.Р., Мевлют И.Ш., Ворожейкин Р.В., Адильшаева Э.И.. "Моделирование игровой сцены при разработке компьютерной игры на платформе Unity 3D" Таврический научный обозреватель, №. 6 (11), 2016, 193-195с.

3. Андерсон М.И., Адаменко К.А. "Влияние игр аркадного направления на развитие индустрии компьютерных игр" Молодой исследователь Дона, №. 2 (17), 2019, 85-90с.

4. Бочкарев Н.А., Молотов Р. С., "Подходы к трансформации объектов виртуальных пространств в среде Unity" Вестник Ульяновского государственного технического университета, №. 3 (75), 2016, 38-41с.

5. Дашкевич А.А., Анисимов К.В. "Геометрическое моделирование процесса освещения" Вестник Национального технического университета Харьковский политехнический институт. Серия: Информатика и моделирование, №. 19 (992), 2013, 24-29с.

6. Иванько М.А., Крюкова А.Г. "Графические возможности мобильных устройств" Вестник Московского государственного университета печати, №. 5, 2015, 53-55с.

7. Рвачёва О.В., Чмутин А.М. "Управление яркостью в компьютерной графике: нелинейный аспект" Инженерный вестник Дона, №. 1(44), 2017, 21с.

8. Реутская Ю.Ю., Новиченко А.А. "Производительность и оптимизация программ. Популярные алгоритмы" Вісник Національного технічного університету України Київський політехнічний інститут. Серія: Радіотехніка. Радіоапаратобудування, №. 41, 2010, 137-147с.

9. Русанова И.В. "Анализ платформ для разработки гибридного

мобильного приложения для систем iOS и Android" Актуальные проблемы авиации и космонавтики, vol. 3, no. 13, 2017, 1100-1102с.

10. Polotnyanschikov I.S., and Zalogova L.A.. "Technology for creating 3D realtime applications in Android OS" Proceedings of the Spring/Summer Young Researchers' Colloquium on Software Engineering, №. 6, 2012.

11. Uyen Vu, Thao Le, Hieu Le, Khoe Nguyen, Lam Nguyen, and Huong Luu. "Android. Performance Patterns" European science, №. 2 (24), 2017, 32-34с.

12. Lengyl E., "Mathematics for 3D Game Programming and Computer Graphics", Course Technology, a part of Cengage Learning, 2012, 158-164с.

13. Bailey M., Cunningham S., "Graphic Shaders", International Standart Book, 2017, 123-129с.

14. Shirley P., Marschner S., "Fundamentals od Computer Graphics", International Standart Book, 2002, 653-664с.

15. Огляд графічних можливостей Unity. URL: https://docs.unity3d.com /ru/current/Manual/GraphicsOverview.html (дата звернення 09.11.2019).