Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

Розробка інтелектуальної гри на рушії Unity
Текстова частина до курсової роботи
за спеціальністю «Інженерія програмного забезпечення»- 121

Керівник курсової роботи

к-т фіз.-мат. наук, доцент

Жежерун О.П.

(прізвище та ініціали)

_____

(підпис)

“____” _____ 2021 р.

Виконав студент Рибка М.К.

(прізвище та ініціали)

_____

(підпис)

“____” _____ 2021 р.

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри інформатики,

к. ф.-м. н. С. С. Гороховський

_____

(підпис)

“____” _____ 2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

Дата видачі “____” _____ 2021 р. Керівник

_____

(підпис)

Завдання отримав

_____

(підпис)

Київ 2021

Календарний план виконання роботи

Тема: Розробка інтелектуальної гри на рушії Unity

№	Назва етапу	Термін виконання	Примітка
1	Отримання теми курсової роботи	23.10.2020	
2	Пошук літератури за темою роботи	30.10.2020	
3	Ознайомлення з науковою літературою	10.11.2020	
4	Визначення структури програми	20.02.2021	
5	Написання першої частини курсової роботи	01.03.2021	
6	Написання другої частини курсової роботи	29.03.2021	
7	Написання висновків курсової роботи	01.04.2021	
8	Перегляд змісту роботи з керівником	05.04.2021	
9	Внесення змін до роботи	06.04.2021	
10	Створення презентації	12.04.2021	
11	Завантаження курсової роботи	12.04.2021	

Київ 2021

ЗМІСТ

ВСТУП 5

РОЗДІЛ 1. ТЕОРЕТИЧНЕ ПОЯСНЕННЯ ТЕМИ 7

1.1. Опис ігрового рушія Unity. 7

1.2. Технології оптимізації. 9

1.2.1. Object Pool 10

1.2.2. Light Baking 11

1.2.3. LOD 11

1.2.4. Texture Atlasing 12

1.2.5. DOTS & ECS 13

1.3. Технології рендерингу 15

1.3.1. Shader Graph 16

1.3.2. Post-processing 17

1.4. Houdini Engine. 18

Висновок 19

РОЗДІЛ 2. ОБРАНІ ТЕХНОЛОГІЇ 20

2.1. Використання Object Pool 20

2.2. Використання Scriptable Object 20

2.3. Shader Graph та Post-Processing 21

2.4. Houdini Engine. 21

Висновок 22

РОЗДІЛ 3. СТВОРЕННЯ ГРИ В UNITY 23

3.1. Створення проекту. 23

3.2. Створення ігрової логіки. 24

3.3. Створення графічної частини 26

3.4. Використання Houdini Engine. 28

Висновок 32

ВИСНОВКИ 33

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 34

ВСТУП

В сучасному світі ігрова індустрія розвивається з неймовірною швидкістю. Постійно з’являються нові технології для створення кращої графіки, фізики та загалом покращення ігрового досвіду. Тож дуже важливим є спостереження та вивчення сучасних методів розробки. Недостатньо мати унікальну ідею гри, адже з поганою реалізацією таку гру ніхто не оцінить. Саме тому варто відразу думати про розробку, слідкувати за сучасними тенденціями та використовувати технології, які поліпшать створення додатку, зекономлять час і ресурси.

Актуальність дослідження.

Наразі Unity є найпопулярнішим ігровим рушієм, завдяки своєму низькому порогу входження та постійні підтримці і модернізації. Загалом, щоб почати вивчення створення гри, обирають цей рушій. Вони використовують різні готові рішення та не звертають увагу на оптимізацію, тим самим занижують репутацію рушія поміж інших. Тож актуальним є дослідження та опис широких та потужних можливостей Unity.

Об’єкт дослідження – процес побудови гри в Unity.

Предмет дослідження – особливості побудови, використовуючи сучасні методи.

Мета – дослідити та розкрити сучасні методи побудови застосунку в Unity, проаналізувати їх корисність.

Завдання дослідження :

1) Дослідження існуючих засобів для побудови гри в рушії.

2) Дослідження нових технологій та методів.

3) Аналіз корисності нових методів.

4) Поетапне створення гри з використанням досліджених технологій.

РОЗДІЛ 1. ТЕОРЕТИЧНЕ ПОЯСНЕННЯ ТЕМИ

1.1. Опис ігрового рушія Unity.

Unity – на сьогодні, це найпопулярніший ігровий рушій. Він став найкращим варіантом для незалежних розробників завдяки своїй простоті використання та низькій вартості. Універсальність Unity дозволяє користувачам легко створювати будь-який жанр гри або ж просто додаток.

Розробка ігрового процесу відбувається завдяки створенню коду, використовуючи C#. Це називається “Скриптинг” — він відрізняється від звичайного програмування тим, що не потрібно писати програмний код, що виконується додатком, адже Unity бере це на себе. Замість цього, ми фокусуємось на ігровому процесі, що створюється саме скриптами. Скрипти пишуться на об’єктно орієнтованій мові C#. Починаючи з версії 2018.1, є можливість використовувати Visual Studio for Unity Community, що допоможе з завершенням написаного коду, використовувати скорочення та вказувати на синтаксичні помилки [1]. Для старіших версій використовувався окремий текстовий редактор MonoDevelop. Але насправді вже існують плагіни для виконання цих завдань на більшості відомих текстових редакторів.

Однією з речей чому Unity є лідером — це кількість підтримуваних платформ. Завдяки Unity можливо одночасно створювати додаток як на мобільні девайси (Android, iOS), так і на десктоп, ігрові консолі, веб тощо. Всього Unity підтримує більше 20 різних платформ [2]. Також для певних платформ Unity має “Unity Cloud Build”, що автоматично компілює, розгортає та тестить додаток для обраної платформи.

Загалом звісно Unity обирають саме через низький поріг входження, адже немає необхідності мати великий досвід в програмуванні на C#, та й до того ж можливим є створення додатку без написання коду. Нещодавно Unity навіть випустила інструмент Bolt, що дозволяє створювати механіки

та логіку взаємодії за допомогою візуальної графової системи, замість роботи з C#. Але це зовсім не означає, що Unity не підходить для більш досвідчених користувачів, адже рушій пропонує велику кількість технологій та можливостей для створення та використання при необхідності. Наприклад в Unity можливим є створення власних вікон для редактора.

Також Unity має Asset Store — це певна бібліотека асетів[1] в якій знаходяться як безкоштовні, так і платні товари, і звісно, якщо є бажання, можливо самому стати видавцем та продавати свої асети. Там можна знайти будь-які необхідні файли як 3д-моделі, аудіо-файли, зображення, скрипти, що реалізують певну логіку, тощо. Всі асети можливо завантажити у проект не виходячи з Unity. Використання готових асетів дозволяє інтегрувати в свій проект різний функціонал, заощадивши місяці роботи.

[1] Асет - елемент, що можна використовувати в Unity. Може складатися з будь-яких файлів, що розпізнаються в Unity.

1.2. Технології оптимізації.

Низький поріг входження це як і плюс, так і мінус Unity. Адже можна просто скласти проект з готових асетів, зробити декілька простих скриптів і це, можливо, навіть буде працювати. Але такий проект скоріше за все потрібно буде повністю переробляти при його розростанні. Щоб дозволити собі гарну якість додатку, необхідно не забувати про оптимізацію. На щастя, Unity має перечень зручних інструментів для дебагу та моніторингу ресурсів, що використовуються. Розглянемо деякі з них:

Unity Profiler — інструмент, за допомогою якого можливо отримати та переглянути інформацію про продуктивність додатку. Переглянути цю інформацію можна як з будь-якого девайсу у вашій локальній мережі або ж під'єднаного до вашого комп'ютеру, так і з самого редактора без неохідності роботи білд[2]. Завдяки цьому можливо відразу переглянути як ваш застосунок працює на запланованій платформі та оптимізувати застосунок, знаючи, що саме спричиняє складності. Unity Profiler показує покрокове виконання коду у виді ієрархії, там відображається така інформація, як скільки часу займає певна команда та вплив налаштувань сцени на додаток.

Frame Debugger — відображає інформацію про draw calls[3] та дозволяє контролювати покрокове створення кадру. В цілому, кількість draw calls необхідно максимально зменшити, а Frame Debugger якраз допоможе з цим. Він зможе показати послідовність викликів draw call команд та інших подій, як очищення буферу, у вигляді ієрархії, що визначає звідки вони походять. Також можна побачити інформацію про кожну викликану команду, наприклад деталі геометрії, кількість вершин, матеріал.

[2] Білд - збірка додатку на обрану платформу.

[3] Draw call - команда, яку рушій відправляє графічному API для відтворення об'єкту.

Отже, ми розглянули засоби моніторингу продуктивності, перейдемо саме до оптимізації. Дивлячись що саме спричиняє проблеми з продуктивністю, рішення можуть бути різними. Якщо ж за допомогою Unity Profiler бачимо, що якась частина коду займає дуже багато часу, необхідно подивитися, що ж може створювати таку проблему. Звісно ж у більшості випадків можливо обійтись звичайними змінами в коді, як кешування можливих даних, видалення пустих методів Update, FixedUpdate та LateUpdate, що за замовчуванням в Unity виконуються постійно. Але для деяких проблем потрібні більш делікатні рішення, наприклад, якщо проблема полягає у частому інстанціюванні[4] об'єктів, то найкращим рішенням буде створення пулу об'єктів (Object Pool).

1.2.1. Object Pool

Object Pool — це спосіб зменшити навантаження на процесор, коли потрібно швидко створювати та знищувати об'єкти. Він допомагає зменшити навантаження на CPU та не бути переповненим повторюваним створенням об'єктів. Використовуючи такий шаблон, замість постійного створення та знищення об'єктів, необхідно створити певну їх кількість на початку, а потім замість видалення вже непотрібних об'єктів просто приховувати їх та показувати при необхідності. В іграх це є дуже важливим шаблоном, адже він запобігає постійному видаленню об'єктів збирачем сміття. Частіше за все, це робиться як массив, що зберігає колекцію певних прихованих об'єктів. Наприклад для ігор, де відбувається постріл кулею, замість виділення пам'яті та створенню нового об'єкту при кожному пострілі, ми просто дістаємо кулю з нашого пулу, переміщуємо в необхідну нам позицію та активуємо її. Ну і звісно ж, при зіткненні, ми її приховуємо, а не знищуємо, та додаємо назад до пулу для подальшого використання.

[4] Інстанціація - створення нового об'єкта викликом методу Instantiate.

Таким чином, можемо забезпечити появу та зникнення об’єктів без необхідності виділення пам’яті та проблем із збирачем сміття.

1.2.2. Light Baking

Також варто не забувати про освіщення сцени. Надмірне використання тіней та динамічного світла може самостійно спричинити низьку продуктивність додатку. На допомогу приходить запікання світла.

Запікання світла (Light Baking) – процес обрахування інформації про освітлення для статичних сцен та світла[3]. Статичне світло можна зробити realtime, тобто таким самим як і динамічне, його вплив на все навколо буде обраховувати кожен кадр. Або ж попередньо запекти світло на саму сцену, що, очевидно, буде використовувати менше ресурсів. Запікання світла – найпоширеніший підхід для досягнення глобальної ілюмінації в іграх та 3д застосунках. При запіканні беруться текстури всіх об’єктів навколо світла, рушій обраховує вплив на ці текстури та зберігає цю інформацію у файлі, що називається світлова карта (lightmap). Крім відносно низької обчислювальної вартості, запікання світла робить можливим досягти кращої якості освітлення, адже немає необхідності знижувати такі налаштування, як кількість відбивань світла та накладання певних фільтрів, бо після запікання вони не будуть впливати на головний процес.

1.2.3. LOD

Щодо моделей, що використовуються, завдяки Frame Debugger можемо подивитися кількість відмальованих вершин в сцені. Максимально допустима кількість вершин у сцені залежить від обраної платформи та обраних девайсів, наприклад, для підтримки максимальної кількості мобільних пристроїв, правилом є не перевищення порогу в 100,000 вершин. Щоб не втрачати якість, зменшуючи кількість вершин самої моделі,

використовують рівні деталізації (Level of detail), оскільки багато трикутників та вершин моделі у полі зору будуть непомітні для користувача. На великій відстані до точки огляду їх проекція на екран істотно менша, ніж роздільна здатність пікселя дисплея. Тож немає необхідності навантажувати систему обрахуванням трикутників, що не помітні користувачеві[4]. В свою чергу LOD передбачає зменшення кількості трикутників по мірі віддалення від точки огляду або за такими показниками, як важливість об’єкту, швидкість відносно точки огляду, тощо. Знижена візуальна якість моделі непомітна через невеликий вплив на зовнішній вигляд предмета при віддалені або швидкому переміщенні. Рівні деталізації застосовуються не тільки для геометрії, а також для контролювання складності пікселя в шейдерах та застосовуються до текстур, але під назвою mipmapping, що використовує копії текстури, але з різною роздільною здатністю.

1.2.4. Texture Atlasing

Але оптимізованість 3д об’єктів залежить не тільки від кількості вершин, а й від кількості матеріалів, що використовуються. Наприклад, якщо в сцені буде чотири різних об’єкти та кожен з них буде мати окремий матеріал, то за допомогою Frame Debugger зможемо побачити, що буде викликано draw call на кожен з об’єктів. Для вирішення такої ситуації зазвичай використовують Атласи текстур (Texture Atlasing).

Texture Atlasing — це спосіб логічно згрупувати всі спрайти[5] або ж текстури в один файл. При створенні моделі в програмах моделювання координати зображення можуть бути відображені не певних об’єктах. Завдяки цьому можливим є створення спільної текстури та матеріалу в Unity

[5] Спрайт - простий 2D об’єкт, на якому є графічне зображення. В Unity використовується для UI, або ж 2D додатків.

для декількох об’єктів. Кожен об’єкт при використанні одного матеріалу буде спиратися на вказані координати на текстурі, тим самим зможемо зменшити кількість draw call викликів. Можемо піти навіть далі та зробити об’єкти статичними, це дозволить Unity використовувати static-batching, що в свою чергу швидше ніж альтернатива — dynamic-batching, адже не змінює вершини на CPU, але використовує більше пам’яті.

На продуктивність застосунку може впливати навіть невірно зроблена архітектура. Але з архітектурою справа не тільки в продуктивності, а й в швидкості та складності додавання нового функціоналу, механік, складнішої логіки, тощо. Насправді майже неможливо відразу визначити необхідну архітектуру проекту, адже при розвитку додатку якісь частини проекту міняються та з’являються нові, що ніяк не пов’язані зі старою логікою. Саме для того, щоб була певна декомпозиція коду на ізольовані блоки в Unity, кожен об’єкт в сцені складається з компонентів. За замовчуванням до кожного елементу додається компонент Transform, що відповідає за положення, поворот та розмір. У рушії є безліч готових компонентів, що дозволяють визначити чи є об’єкт камерою, світлом, який має меш[6] тощо. Також, якщо наслідувати MonoBehaviour, то створений скрипт можливо використовувати як компонент. Але з одними компонентами не вийде досягти необхідної гнучкості коду та продуктивності.

1.2.5. DOTS & ECS

Наразі розробники Unity обрали направлення на дата-орієнтований технологічний стек (DOTS)[5]. Цей напрямок був обраний оскільки традиційний C# не найкраща мова програмування з точки зору продуктивності, адже неможливо контролювати, де саме в пам’яті

[6] Меш — набір вершин, ребер та граней, що описують форму багатогранного об’єкта.

розташовуються дані. Також, працюючи з фрагментом коду в якому критично важливим є створення максимально оптимізованого та продуктивного коду, стандартна бібліотека, що організована навколо “Об’єктів в купі” та “Об’єктів, що мають посилання на інші об’єкти” взагалі є зайвою. Так само зайвим є збирач сміття та операції виділення пам’яті, тобто замість звиклих класів краще використовувати структури. Якщо до цього ще додати деякі важливі контейнери, як NativeArray, то завдяки тим елементам C# мови, що залишилися, швидкість компіляції буде майже як в C++. Unity називає це “High Performance C#” (HPC#) і для нього окремо створили компілятор Burst. Щодо побудови правильної архітектури, використовуючи HPC# був створений шаблон ECS.

ECS (Entity Component System) — шаблон проектування “Сутність Компонент Система”. В такому шаблоні сутність — це контейнери, що виступають сховищами для компонентів. В свою чергу компоненти — блоки даних, які визначають властивості будь-яких ігрових об’єктів. А вже системи відповідають за логіку, що обробляє дані з контейнерів. Основною різницею між звичайною системою компонентів Unity та ECS є те, що логіка в останньому має обов’язково бути відділена від даних. При реалізації цього шаблону в простому проекті створиться набагато більше коду та файлів, ніж могло б бути при використанні звичайної системи компонентів. Але в такій реалізації є переваги, які можуть бути критичними для великих проектів:

- Проект буде гнучким та масштабованим, можливим буде додавання та заміна систем без зачеплення інших.
- Можливим буде використання вже створених систем для нових елементів, та їх поєднання.
- Ефективне використання пам’яті. У випадку з MonoBehaviour та звичайною системою компонентів Unity відбувається дуже багато звертань до функцій рушія, що знижує продуктивність[6].

- Можливість використовувати створений код поза рушієм, наприклад на сервері.
- Оскільки всі системи працюють незалежно одна від одної, при необхідності, їх можна розподілити між потоками. Але є сенс це використовувати тільки в дуже вузьких ситуаціях, адже синхронізація потоків може зайняти більше часу, ніж саме обрахування і вийде навіть повільніше.

Для повного використання переваг багатоядерних процесорів в DOTS була створена “Система задач” (Jobs System). Вона допомагає розробникам створювати безпечний, швидкий та багатопоточний код. Використання такого коду може забезпечити високу продуктивність, а в поєднанні з Burst компілятором, дає покращену якість генерації програмного коду, що також приводить до значного зменшення споживання батареї на мобільних пристроях [7]. Важливим є те, що система задач інтегрується з C++ системою, яку Unity використовує внутрішньо. Завдяки такій співпраці виходить уникнути створення більшої кількості потоків ніж самих ядер у процесорі, що може спричинити суперечку щодо ресурсів.

Unity активно працює над повною інтеграцією DOTS, до стеку постійно додаються нові пакети. Нещодавно з’явилась підтримка Unity Physics, Animation, DSPGraph (низькорівнева звукова підсистема), які працюють з Burst компілятором, покращуючи продуктивність.

1.3. Технології рендерингу

Перейдемо до візуальної частини створення застосунку. В Unity є можливість використовувати різні технології рендеренгу “Render Pipelines”.

- Standard Render Pipeline — технологія рендерингу за замовчуванням. Має недоліки в порівнянні з іншими технологіями, адже немає підтримки таких потужних речей як Shader Graph та Post-Processing.
- Universal Render Pipeline (URP) — оптимальний варіант для захвату максимальної кількості платформ. Ця технологія забезпечує високу продуктивність та кращу графіку, ніж в стандартному рішені. Використовуючи такий рендеринг, Unity дозволяє оптимізувати застосунок під конкретні платформи, використовувати більше налаштувань рендеренгу, контролювати використання ресурсів. Також URP має підтримку Post-Processing та Shader Graph.
- High Definition Render Pipeline (HDRP) — найкращий вибір для проектів для потужних систем з підтримкою паралельних обчислень. Він дозволяє створювати максимально якісні візуальні ефекти.

1.3.1. Shader Graph

В кожної доданої до Unity моделі має бути певний матеріал. Він відповідає за те, як саме буде відображатися модель. В свою чергу шейдер[7] описує спосіб вичислення того, як саме виглядає матеріал моделі. В Unity присутні Standard Shaders, наприклад Lit, який обробляє освітлення сцени, Unlit — не реагує на освітлення, спеціальні шейдери, що оптимізовані для використання на мобільних девайсах, тощо. Але інколи цього не достатньо і з’являється необхідність створення власного ефекту. Раніше для створення власних шейдерів необхідно було використовувати такі мови програмування, як HLSL або Cg, а оскільки Unity працює над зниженням порогу входження, вони створили Shader Graph. Він дозволяє спростити написання шейдерів, адже завдяки цій технології, вони створюються у візуальному інтерфейсі з використанням нод та створенням графу.

[7] Шейдер - невелика програма, що має інструкції для GPU.

1.3.2. Post-processing

Постобробка (Post-processing) — це процесс накладення певних фільтрів та ефектів на буфер камери. Ці ефекти загалом направлені на покращення візуальної якості зображення та не потребують навиків програмування [7].

В цілому в Unity був створений “Стек постобробки”, що тримає в собі повний набір всіх можливих візуальних ефектів. Таке групування є як і оптимізованим, адже підтримується поєднання деяких ефектів за один прохід, так і зручним для використання, бо всі ефекти знаходяться в одному графічному інтерфейсі.

Нижче наведені деякі з ефектів :

- Ambient Occlusion — в режимі реального часу затемнює отвори, перехрестя, складки та поверхні, що знаходяться близько один до одного. Він є необхідним, щоб досягти більшої реалістичності, адже в реальному житті такі зони перекривають навколишнє світло. Але такий ефект є доволі ресурсозатратним.
- Auto Exposure — здатність реалістично пристосовуватися до різних рівнів темряви та світла.
- Bloom — ефект на камерах, який відтворює смуги світла, що простягаються від меж яскравих ділянок зображення.
- Chromatic Aberration — викривлення зображення, зумовлене дисперсією світла в лінзах. Такий ефект веде до зниження чіткості зображення і до появи кольорових контурів, смуг, плям.
- Color Adjustments — процес зміни кольору та яскравості зображення.

- Depth of Field — ефект, що імітує фокусні властивості об’єктива камери. Об’єкти, що розташовані ближче або ж далі, ніж вказана глибина, будуть розфокусовані.
- Tonemapping – процес відображення кольорових значень від високого (HDR) до низького динамічного діапазону (LDR). Це означає, що 16-бітові значення кольору з плаваючою крапкою будуть зіставлені з 8-бітним значенням.

Unity постійно працює над оптимізацією та створює нові інтерфейси для роботи з постобробкою, наприклад компонент з інтерфейсом постобробки для камери, компонент для відладки, тощо.

1.4. Houdini Engine.

Houdini – це продвинута програма для побудови 3д моделей, анімацій, ефектів, симуляцій тощо. Особливістю Houdini є те, що вона повністю базується на процедурній побудові [9]. Тобто розробник передає певні параметри та правила програмі, потім вона створює дещо, базуючись на обмеженнях та вказаних даних. Ці дані можна легко змінювати, що економить дуже багато часу та ресурсів. Звісно ж, коли модель або ефект створюється людиною вручну, то присутній більший контроль над творчими компонентами, однак процедурне створення є незамінним для виконання більш механічних задач та економії часу, наприклад створення реалістичних будівель, дерев, доріг тощо.

В Houdini використовуються ноди з певною логікою, що поєднуються між собою утворюючи граф. Завдяки такій системі можливо змінювати правила, параметри і відразу отримати результат, без необхідності створення нового графу. В створених прототипах можливо переробити тільки конкретні частини, що не сподобались, а не витрачати час на перероблення всієї системи. Також присутня можливість запаковування

створених графів у власний інструмент з інтерфейсом, для подальшого використання, в Houdini це називається “digital asset”.

Оскільки Houdini все більше і більше використовується в створенні ігор, для найпопулярніших рушіїв Unity та Unreal створили Houdini Engine. Це дуже потужний плагін, що дозволяє глибоку інтеграцію Houdini та її процедурний робочий процес в ігровий рушій. Завдяки ньому можливо створювати продукт та одразу бачити результат в самому рушії, де є можливість змінювати його за вказаними параметрами.

Нещодавно цей плагін став безкоштовним для використання, тож це, напевне, найкращий час для його вивчення та використання.

Висновок

Отже, ми поговорили про деякі тонкощі створення додатку використовуючи Unity. Розглянули, на що необхідно звертати увагу для оптимізації під час створення додатку, та які саме існують підходи для цього. Визначили, які технології рендеренгу присутні та які можливості надаються при використанні URP або ж HDRP. І дослідили найновіші технології для створення ефектів та 3д, використовуючи Houdini.

РОЗДІЛ 2. ОБРАНІ ТЕХНОЛОГІЇ

Для демонстрації описаних технологій та матеріалу в цій роботі, було обрано просту endless-running гру. Найвідомішим прикладом такої гри є Subway Surfer. Хоча сама гра є досить простою, через те, що спрямована на слабкі девайси, вона вимагає ретельної оптимізації, тож гарно підходить для демонстрації описаних вище технологій.

2.1. Використання Object Pool

Основною складністю при створенні такої гри є підвантаження рівня при просуванні. По-перше, необхідно обов’язково видаляти ресурси, що вже не використовуються, аби не виник витік ресурсів. По-друге, створення великої кількості нових об’єктів буде сильно впливати на продуктивність. Отже, ідеальним варіантом для реалізації такої механіки є використання Object Pool. Замість того, щоб створювати та знищувати об’єкти повз які користувач пробіг, ми просто будемо створювати необхідну кількість в самому початку та помічати як готові до подальшого використання, коли об’єкт вже не потрібен.

2.2. Використання Scriptable Object

Наступним питанням є створення виду навколишнього середовища та позицій перешкод. Щоб надати унікальність кожній грі, рівень повинен процедурно створюватися випадковим чином. Насправді це псевдовипадковість, адже випадковим чином буде важко контролювати якість рівня та те, що перешкоди взагалі можливо перейти. Отже, необхідно створити певні шаблони, в яких визначено позиції перешкод, навколишнього середовища, дороги та монет для збирання. Для зручного створення та зберігання даних про такі шаблони, необхідно створити ScriptableObject.

ScriptableObject — контейнер даних, який можна використовувати для збереження великих обсягів даних, незалежно від екземплярів класу[10].

2.3. Shader Graph та Post-Processing

Щодо графічної частини застосунку, то в цілях оптимізації, необхідно позбутися динамічного освітлення та тіней. Для підтримки більшої кількості слабких девайсів, була обрана більш мінімалістична та мультиплікаційна графіка, тож такі речі як тінь та динамічне освітлення, не потрібні. Також потрібен буде шейдер для створення матеріалів з AO картою, яка буде імітувати освітлення без зайвого використання ресурсів.

Рівень буде створюватися процедурно, тобто буде видно, що спереду з'являються нові об'єкти, тому необхідним є також створення шейдеру туману, який буде приховувати як збирається рівень.

Для цих двох шейдерів буде продемонстроване використання Shader Graph. Тобто необхідно встановити URP, адже так можливо буде використовувати вищезгадану технологію. Разом з цим буде підтримуватися максимальна кількість платформ, що є головною ціллю.

Також можемо використати Post-Processing для досягнення кращої якості. А саме, необхідно буде збільшити насиченість та встановити низький динамічний діапазон, щоб забезпечити 8-бітове значення кольору в вихідному зображені.

2.4. Houdini Engine.

В цьому проекті є сенс використати Houdini та Houdini Engine для процедурної побудови навколишнього середовища. Для демонстрації було обрана побудова дерев. Необхідність в генерації дерев виникає дуже часто і тому це доволі затребувана тема. Моделювання дерев вручну займає багато часу, тож створення асету для побудови процедурних дерев є незамінним. З цією задачею чудово впорається Houdini.

Висновок

Отже, було обрано гру в жанрі endless-runner та обрані наступні технології для використання:

- Object Pool – для оптимізованого використання ресурсів;
- ScriptableObject – для зберігання даних про шаблони;
- Shader Graph – для створення ефекту туману;
- Post-Processing – для досягнення кращої якості зображення.
- Houdini Engine – для швидкої та зручної побудови дерев.

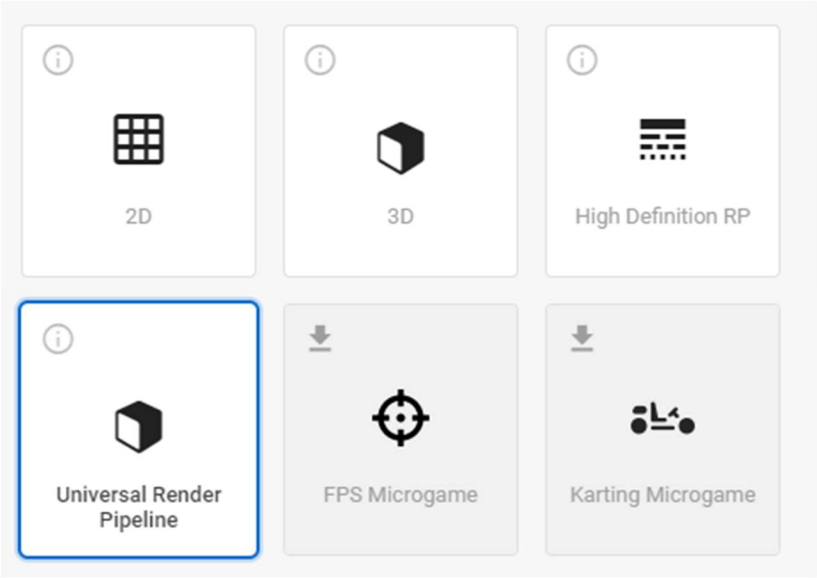
РОЗДІЛ 3. СТВОРЕННЯ ГРИ В UNITY

3.1. Створення проекту.

Отже, перейдемо безпосередньо до самого створення гри. Перш за все, необхідно обрати версію самої Unity. Взагалі пропонується два варіанти версій :

- LTS (Long Term Support) — такі версії забезпечують кращу стабільність та будуть підтримуватися 2 роки після свого виходу.
- Tech Stream — на відміну від LTS версій, надають доступ до найновіших технологій.

Далі при створенні нового проекту в обраній версії є можливість обрати шаблон проекту (зобр. 1), він відразу виставить відповідні налаштування на основі найкращих практик.



Зображення 1.

Шаблони прискорюють процес підготовки проекту та використовують деякі важливі плагіни, що не включені у початковий проект за замовчуванням.

3.2. Створення ігрової логіки.

Для оптимізованої реалізації підвантаження рівня, використовуємо Object Pool. Оскільки одночасно будуть створюватися різні за логікою об’єкти, можемо створити базовий клас SpawnableObjectPool. Він буде мати методи для заповнення пулу об’єктів, знищення пулу та отримання об’єкту. Базовий клас пулу об’єктів буде виглядати приблизно так :

```
public class SpawnableObjectPool
{
    /*Пул об'єктів у вигляді черги*/
    [SerializeField] private Queue<SpawnableObject> _pool;

    /*Створює та заповнює пул*/
    public void SpawnPool(Transform parent) { ... }

    /*Знищую та чистить пул*/
    public void DespawnPool() { ... }

    /*Дістає об'єкт з черги*/
    public SpawnableObject GetObject() { ... }
}
```

Також, необхідно створити відповідний клас SpawnableObject, що й буде створюватися та використовуватися пулом. Він може мати декілька івентів[8], які будуть говорити про те, що об’єкт був показаний або прихований, методи, які власне показують та приховують об’єкт. Для більшої варіативності, можемо додати список різних зовнішніх виглядів конкретного об’єкта та метод для обрання стилю випадковим чином.

[8] Івент - певне повідомлення про те, що відбулось деяка подія.

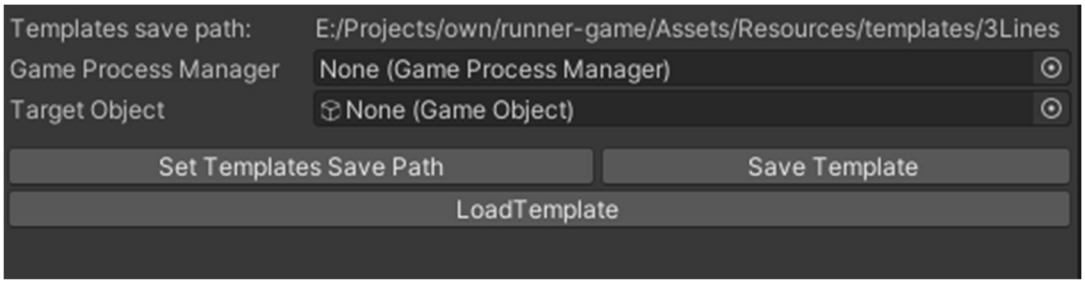
```
public abstract class SpawnableObject : MonoBehaviour
{
    /*Івенти для контролювання того, коли об'єкт з'являється, а коли
приховується*/
    public event Action onShown;
    public event Action onHidden;

    /*Варіанти зовнішнього вигляду об'єкту*/
    public List<SpawnableObjectVariant> variants;

    /*Показує об'єкт*/
    public virtual void Show() { ... }

    /*Ховає об'єкт*/
    public virtual void Hide() { ... }

    /*Випадковим чином обирає стиль об'єкту з вказаних варіантів*/
    public virtual void SetRandomVariant() { ... }
}
```

Для зберігання та використання шаблонів з об’єктами навколишнього середовища, створюємо відповідний ScriptableObject. В ньому повинні зберігатися дані про позиції, поворот та тип об’єктів, щоб потім по цим даним можливо було створити рівень. В нашому випадку зберігаються дані про монети; перешкоди, а саме бар’єри, машини та автобуси; дороги та навколишнє середовище, тобто будівлі та дерева.

```
/*Тут вказуються налаштування для Unity, тобто як буде називатися файл
при створені, в якому розділі меню його можна знайти, тощо*/
[CreateAssetMenu(fileName = "LevelPieceTemplate", menuName =
"Templates/LevelPieceTemplate", order = 1)]
public class LevelPieceTemplate : ScriptableObject
{
    /*Позиції, поворот та вид дороги*/
    public List<RoadPosition> roadPositions;
    /*Позиції, поворот та вид перешкоди*/
    public List<ObstaclePosition> obstaclePositions;
    /*Позиції, поворот та вид монет, бонусів ...*/
    public List<EarnablePosition> earnablePositions;
    /*Позиції, поворот та вид навколишнього середовища (будівля,
```

```
дерево ...)*/
    public List<EnvironmentPosition> environmentPositions;
}
```

І для того, щоб було зручно створювати нові шаблони та змінювати старі, було створено окремий інтерфейс в редакторі, який відповідає за це (зобр. 2). В ньому є можливість встановити шлях збереження файлу шаблону, зберегти файл або ж завантажити збережений шаблон.



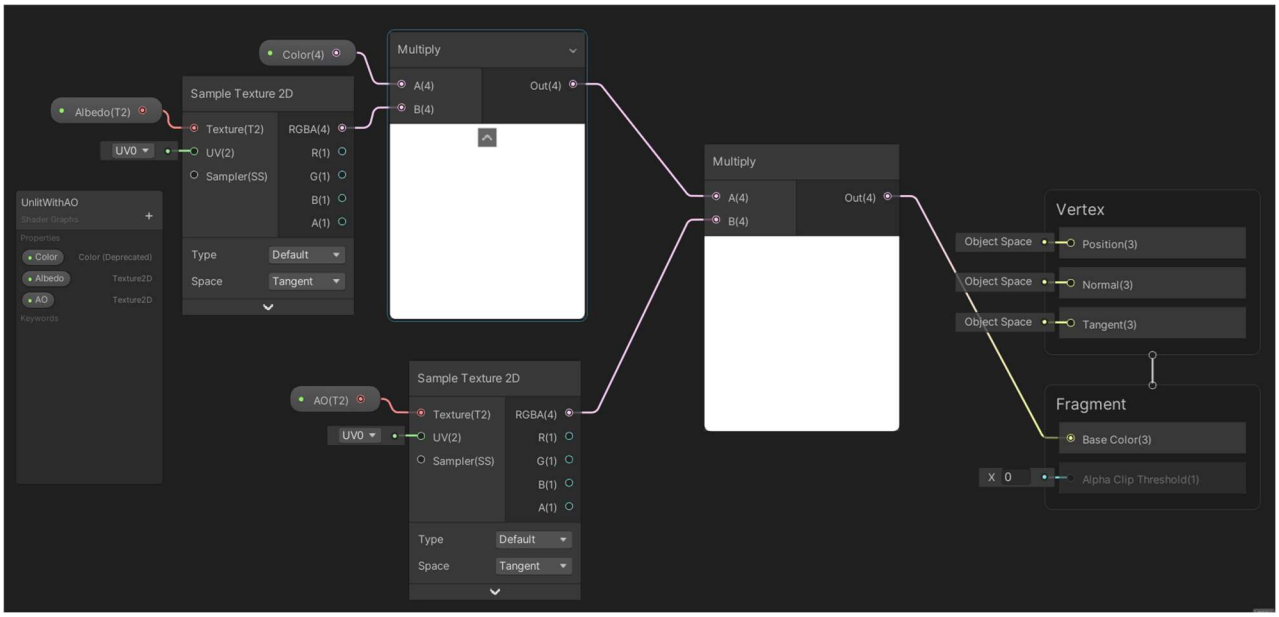
Зображення 2.

3.3. Створення графічної частини

Перш за все, необхідно зробити матеріал, що буде використовувати як звичайну текстуру, так і карту Ambient Occlusion[9], бо Unlit матеріал за замовчуванням АО карти не має. Оскільки АО карта це просто чорно-біле зображення, в якому білими є світлі частини, а чорними — темні. Такий шейдер дуже легко створити, достатньо просто перемножити текстуру самого об’єкту на його АО карту. В Shader Graph це можна зробити за допомогою ноди “Multiply” (зобр. 3). Також там доданий колір, що так само

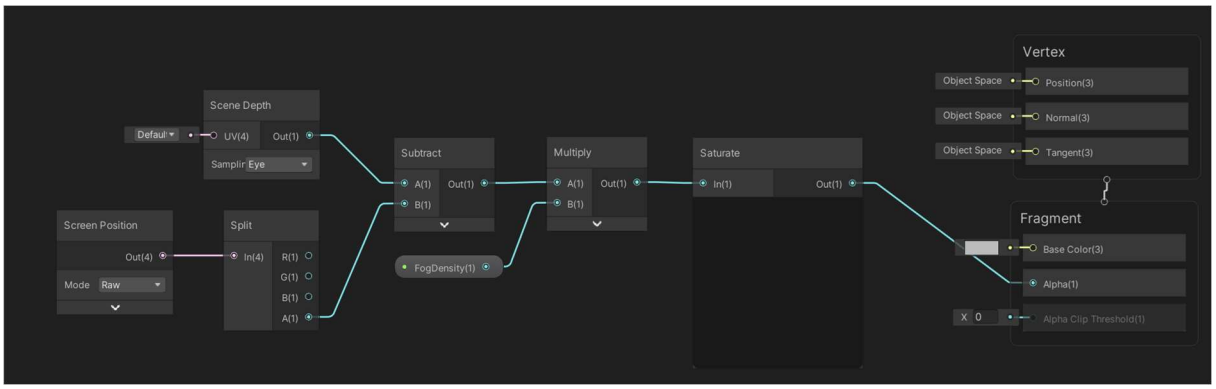
[9] Карта Ambient Occlusion (AO) — окрема текстура, яка містить дані про освітлення.

множиться на головну текстуру для того, щоб була можливість змінити загальний колір або ж використовувати тільки колір без самої текстури.



Зображення 3.

Щоб створити ефект туману потрібно використовувати “Scene Depth”. Вона показує як далеко об’єкти розташовані відносно камери. Наступним потрібно взяти глибину самого об’єкту туману, для цього можна використати ноду “Screen Position” та встановити режим Raw, адже саме він дістає глибину об’єкта. Далі потрібно відняти два значення глибини, щоб отримати різницю в глибині між поверхнею об’єкта та відстань об’єкта до камери. Чим далі розташовані ці два значення глибини, тим більшим буде вихідне значення, створюючи ефект туману. Також можемо додати певну щільність туману просто помноживши на певне число, але після цього варто помістити отриманий результат у ноду Saturate, щоб значення було в межах від 0 до 1, адже якщо використовувати значення більше 1, то можуть з’явитися яскраві артефакти. Отриманий результат необхідно вивести у альфа канал (зобр. 4).



Зображення 4.

Наостанок, варто зазначити, що для покращення зовнішнього виду додатку, використовувалась постобробка, а саме ефект “Tonemapping”, та “Color Adjustments” для налаштування насиченості та кольору.

3.4. Використання Houdini Engine.

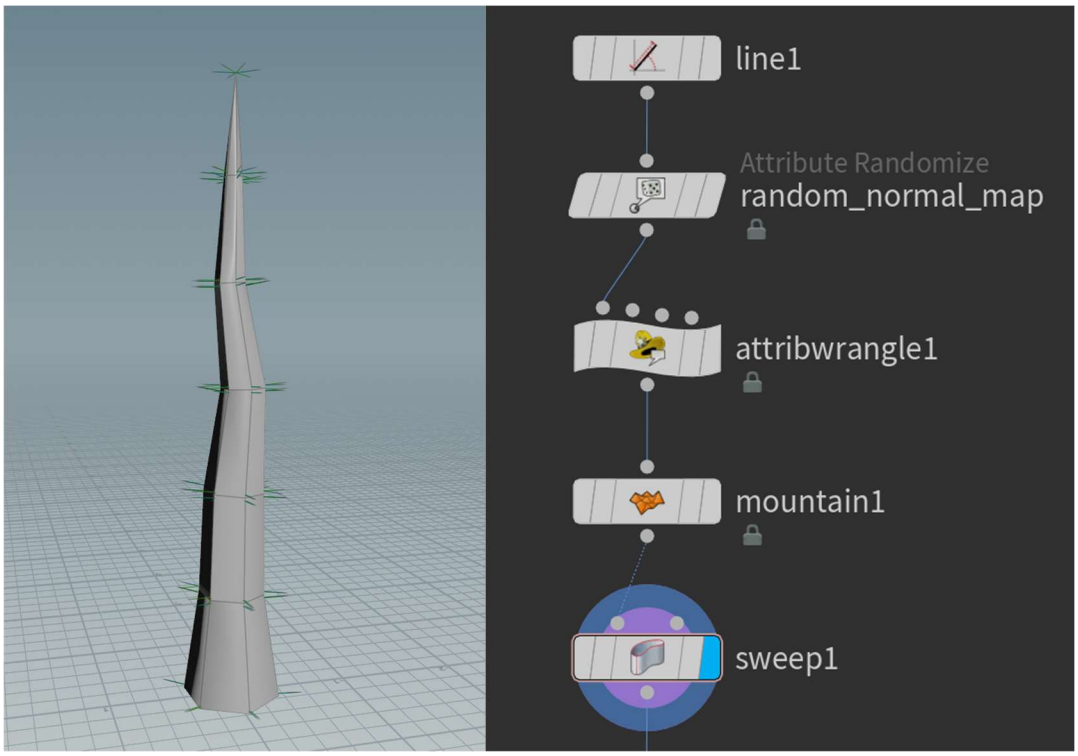
Перейдемо до побудови дерев використовуючи Houdini та Houdini Engine. Перш за все, необхідно створити стовбур дерева. Для цього можна використати ноду “Line”, в ній можемо поділити лінію на сегменти та вказати їх кількість. Далі використати “Attribute Randomizer” для випадкового направлення нормалей кожної з вершин лінії і за допомогою ноди “Mountain” змінити викривлення лінії, відповідно до нормалей. Також за допомогою мови програмування в Houdini — VEX, можна зробити певний градієнт, через який буде контролюватися вага нормалей відповідно до вершини – це робиться, щоб досягти ефекту, що чим вище вершина, тим сильніше вона буде викривлена (зобр. 5). Тут використовується зміна @N, яка відповідає за нормаль вершины.

VEXpression

```
f@gradient = (float)@ptnum/ (float)(@numpt - 1);
@N *= chramp("NoiseFalloff", @gradient);
```

Зображення 5.

І наостанок, треба використати “Sweep” ноду, для надання об’єму лінії. Нинішній результат показано на зображенні 6.



Зображення 6.

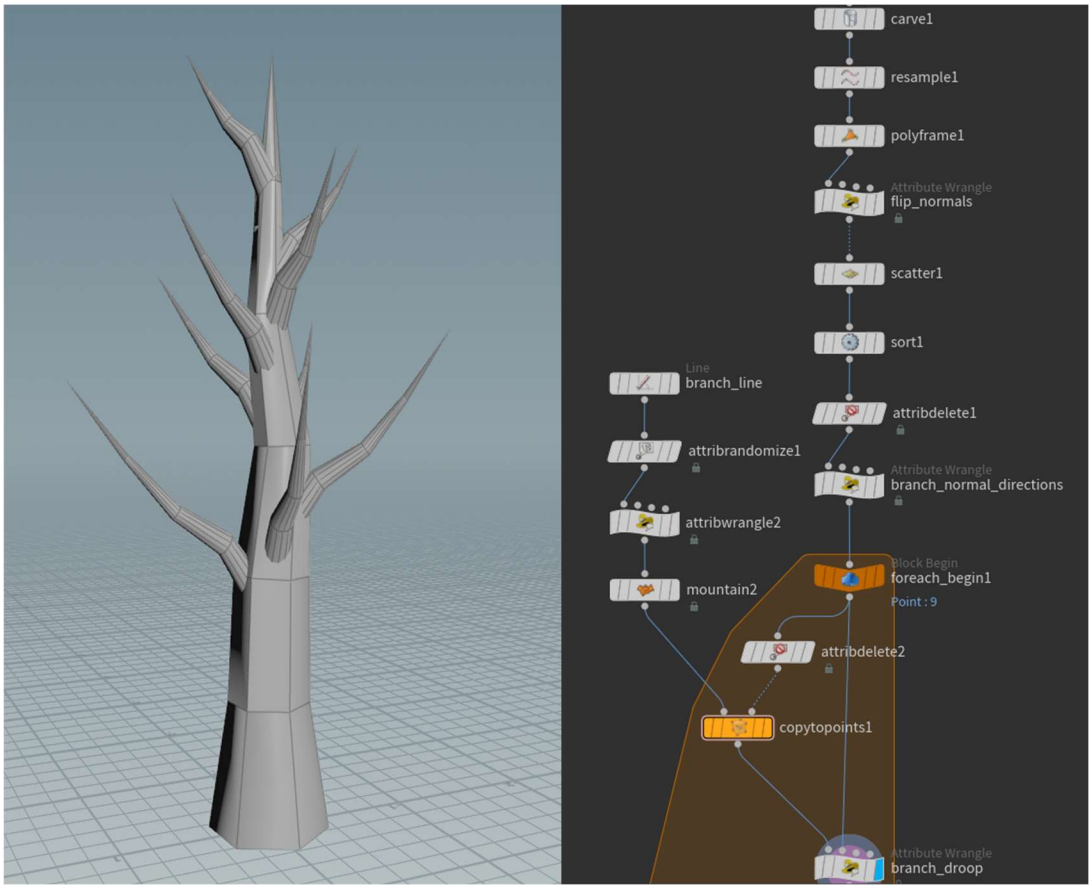
Наступним є створення гілок з головного стовпу. За допомогою ноди “Carve” та раніше створеної лінії стовпу дерева можливо встановити зону на якій будуть з’являтися гілки. Потім, використовуючи “Resample” ділимо лінію на необхідну кількість гілок, тут також можна додати деяку випадковість у розташуванні гілок за допомогою ноди “Scatter”. Щоб направлення гілок відрізнялося, але було рівномірним, можемо створити ноду з відповідним VEX кодом (зобр. 7). Тут знову створюється певний градієнт, що допоможе контролювати направлення гілок по вісі y. Далі використовується параметр rotationAmount, що буде змінювати закрученість спіралі. Використовуючи математичні операції sin та cos, знаходимо направлення гілок по вісям x та z відповідно.

VEXpression

```
f@gradient = (float)@ptnum/ (float)(@numpt - 1);

//Y normal
f@yPos = chramp("YPos", @gradient);

//x and z normals
float rotationAmount = chf("RotationAmount");
float xPos = sin(@gradient * rotationAmount);
float zPos = cos(@gradient * rotationAmount);

@N.x = xPos;
@N.z = zPos;
@N.y = @yPos;

@N = normalize(@N);
@pscale = chramp("GlobalBranchSize",@gradient);
```

Зображення 7.

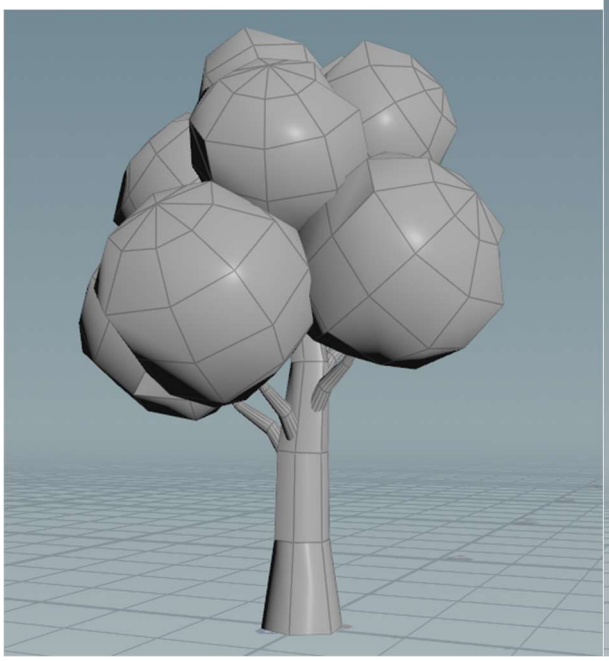
Тепер використовуючи блок “Foreach”, треба пройтись по кожній вершині та побудувати гілку. Сама форма гілки буде створюватися таким самим чином, як створювався стовп (зобр. 6), але за вказаним нормаллю направленням. Маємо результат (зображення 8.) :



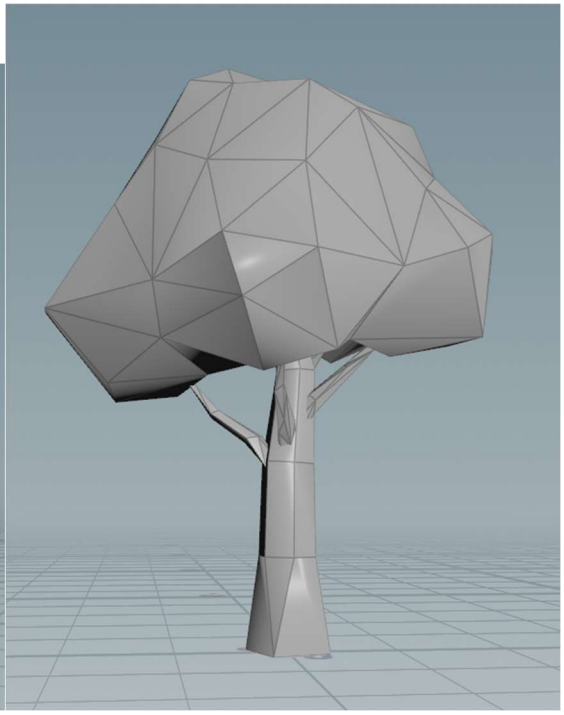
Зображення 8.

І тепер потрібно створити самі кущі. Оскільки направлення графіки було взяте мінімалістичне, немає сенсу ускладнювати задачу та робити окремі листки. Отже, можна просто використати можливість створення примітивних фігур та створити на кожну гілку сферу, після чого деформувати її, використовуючи згадану ранішу ноду “Mountain”. Тепер достатньо тільки поставити поворот відповідний до гілки і маємо процедурно створене дерево (зобр. 9).

Наостанок варто ще перетворити всі сфери кущів в один меш та зменшити кількість полігонів. Це можливо зробити, використовуючи функціонал VDB, перетворивши все у воксель [10]. Для цього в Houdini існує “VDB From Polygon” та “Poly Reduce”. І тепер маємо досить оптимізоване дерево, готове до використання (зобр. 10). В Houdini є можливість встановити будь-які параметри з нод для налаштування у запакованому асеті. Використовуючи Houdini Engine, можемо перенести створений асет до Unity та змінювати його по зазначеним параметрам.



Зображення 9.



Зображення 10.

[10] Воксель - аналог пікселя для тривимірного простору.

Висновок

Отже, в практичній частині було наведено приклад застосування технологій, згаданих та описаних в теоретичній. В результаті було створено мобільну гру в жанрі endless-runner. Для оптимізованої генерації рівня використовувався Object Pool. Були створені матеріали та шейдери для туману та інших об’єктів. Також було продемонстроване використання Houdini та Houdini Engine для створення дерев.

ВИСНОВКИ

Отже, в даній курсовій роботі було детально вивчено та описано створення гри в рушії Unity. Насамперед, увага була приділена оптимізації застосунку та створенню мультиплікаційного стилю. Також були розглянуті та використані такі сучасні технології, як Shader Graph, Post-Processing та Houdini Engine. Для покращення продуктивності створення рівня використовувався Object Pool. За допомогою ShaderGraph, було розроблено шейдер туману, який приховує появу нових об’єктів. Houdini та Houdini Engine використано для процедурного створення дерев. В результаті була розроблена гра в жанрі endless-runner, яка демонструє всі вищезгадані технології.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. “Програмування на C# в Unity для початківців”, Unity [Електронний ресурс] URL: https://unity3d.com/ru/learning-c-sharp-in-unity-for-beginners
2. “Мультиплатформеність в Unity”, Unity [Електронний ресурс] URL: https://unity.com/ru/features/multiplatform
3. David Larsson, Autodesk Inc. Pre-computing Lightning in Games, 2010. [Електронний ресурс] URL: https://cgg.mff.cuni.cz/~jaroslav/gicourse2010/giai2010-06-david_larsson-slides.pdf
4. David Luebke, Level of Detail for 3D Graphics, 2003. [Електронний ресурс] URL: http://index-of.co.uk/Game-Development/Programming/Level%20of%20Detail%20for%203D%20Graphics.pdf
5. Lucas Meijer, On DOTS: C++ & C#, 2019 [Електронний ресурс] URL: https://blogs.unity3d.com/ru/2019/02/26/on-dots-c-c/
6. “10 000 викликів Update ()”, Valentin Seminov, 2015 [Електронний ресурс] URL: https://blogs.unity3d.com/ru/2015/12/23/1k-update-calls/
7. C# Job System Overview, Unity [Електронний ресурс] URL: https://docs.unity3d.com/2019.3/Documentation/Manual/JobSystemOverview.html
8. “Turn on gorgeous effects with the Post-Processing Stack”, Unity, 2019 [Електронний ресурс] URL:

https://unity3d.com/how-to/set-up-post-processing-stack

9. “Introduction to Houdini”, SideFX
   [Електронний ресурс] URL:
   https://www.sidefx.com/docs/houdini/basics/intro.html
10. “ScriptableObject”, Unity
    [Електронний ресурс] URL:
    https://docs.unity3d.com/Manual/class-ScriptableObject.html