МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет радіоелектроніки

Факультет _____Центр післядипломної освіти_____
(повна назва)
Кафедра_____Програмної інженерії_____
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти - другий (магістерський)

Дослідження алгоритмів інтерактивної поведінки рослинного оточення для 3D сцени
(тема)

Виконав: студент 2 курсу, групи __ІПЗздм-19-1___
_____Алєйнікова В.М._____
(прізвище, ініціали)

спеціальності 121- Інженерія програмного забезпечення
(код і повна назва спеціальності)

Освітньо-наукової програми_____
(тип програми)

Інженерія програмного забезпечення__
(повна назва освітньої програми)

Керівник _____ к.т.н. доц. Шевченко О.Л.___
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри, проф. _____ З.В.Дудар

2021р.

Харківський національний університет радіоелектроніки

Факультет_ Центр післядипломної освіти_____
Кафедра_Програмної інженерії_____
Рівень вищої освіти - другий (магістерський)

Спеціальність _121-Інженерія програмного забезпечення_____
(код і повна назва)
Тип програми освітньо-наукова програма

Освітня програма_Інженерія програмного забезпечення_____

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«_26»__березня_____ 2021 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові _ Алєйніковій Валерії Максимівні_____
(прізвище, ім'я, по батькові)

1. Тема роботи _Дослідження алгоритмів інтерактивної поведінки рослинного оточення для 3D сцени_____

затверджена наказом університету від "_26_" _березня_2021 р № _34Стз__

2. Термін подання студентом роботи до екзаменаційної комісії 14 травня 2021р.

3. Вихідні дані до роботи *електронні ресурси за обраною тематикою, мінімальні вимоги до функціональності програми, загальні вимоги до архітектури системи*_____

4. Перелік питань, що потрібно опрацювати в роботі *аналіз предметної області і постановка задачі, аналіз вимог до програмної системи, аналіз матеріалів та шейдерів, дослідження алгоритму малювання за допомогою Render Targets в Unreal Engine, алгоритм створення інтерактивної трави в Unreal Engine, опис візуальної та програмної складової роботи алгоритму, аналіз можливих застосувань та тестування програмної системи.*

5 Консультанти розділів роботи

Найменування розділу	Консультант (посада, прізвище, ім’я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Спецчастина	к.т.н. доц. Шевченко О.Л.		20.04.21

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Аналіз проблемної області та постановка задачі	25.01.2021	виконано
2	Формування вимог до програмної системи	25.02.2021	виконано
3	Аналіз проблемної галузі	26.03.2021	виконано
4	Дослідження алгоритмів впливу на поведінку рослинності на 3D сцені	28.03.2021	виконано
5	Розробка моделей поведінки рослин	05.04.2021	виконано
6	Розробка програмного модулю візуалізації алгоритму	14.04.2021	виконано
7	Оптимізація програмного модулю	18.04.2021	виконано
8	Тестування програмного модулю	20.04.2021	виконано
9	Підготовка пояснювальної записки	27.04.2021	виконано
10	Підготовка презентації та доповіді	4.05.2021	виконано
11	Перевірка на плагіат	6.05.2021	виконано
12	Попередній захист	11.05.2021	виконано
13	Нормоконтроль, рецензування	12.05.2021	виконано
14	Занесення диплома в електронний архів	13.05.2021	виконано
15	Допуск до захисту у зав. кафедри	14.05.2021	виконано

Дата видачі завдання ___25 березня__2021 р.

Студент _____ Алєйнікова В.М._____
(підпис)

Керівник роботи _____ к.т.н. доц. Шевченко О.Л.___
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Кваліфікаційна робота магістра містить: 117 с., 58 рис., 18 джер.

АЛГОРИТМ, ВЕРТЕКСИ, ВІТЕР, ЕФЕКТИ, ПОВЕДІНКА, РОСЛИНИ, СЦЕНА, ТРИКУТНИКИ, ШЕЙДЕР, CULLING, DRAW CALLS, INSTANCE MESHES, FPS, LOD, UE4

Метою цієї роботи є підвищення оптимізація поведінки рослинного покриття на ігровій сцені.

Об’єктом дослідження є алгоритмізація поведінки рослин в ігровому світі.

Методи розробки базуються на таких поняттях та технологіях, як вертекси шейдер, culling, draw calls, instance meshes, fps, lod, ue4.

В результаті роботи було досліджено алгоритми інтерактивної поведінки рослинного оточення для 3D сцени та реалізовано програмний модуль, який може бути інтегрован в будь-який проект на рушії UE4.

ALGORITHM, VERTEXES, WIND, EFFECTS, BEHAVIOR, PLANTS, SCENE, TRIANGLES, SHADER, CULLING, DRAW CALLS, INSTANCE MESHES, FPS, LOD, UE4

The purpose of this work is to increase the optimization of the behavior of vegetation on the game stage.

The object of research is the algorithmization of plant behavior in the game world.

Development methods are based on such concepts and technologies as shader vertices, culling, draw calls, instance meshes, fps, lod, ue4.

As a result, the algorithms of interactive behavior of the plant environment for the 3D scene were investigated and a software module that can be integrated into any project on the UE4 engine was implemented.

Я, Алєйнікова Валерія Максимівна, студент групи ІПЗздм-19-1, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження алгоритмів інтерактивної поведінки рослинного оточення для 3D сцени», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElAr KhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений (а) з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ВСТУП

Ігри з відкритим світом стабільно набирають популярність і знаходяться на перших місцях в списках бестселерів. Кожна нова гра піднімає планку розмірів і складності світу. Просто дивлячись на трейлери останніх ігор з відкритим світом, можна зрозуміти, що їхня мета - створення відчуття величезного масштабу.

Споруда таких світів ставить перед розробниками велике питання - як ефективно заповнити подібні просторі світи? Ніхто не хоче розставляти кожне дерево вручну, особливо якщо команда розробки мала. Зрештою, розробка ігор завжди пов'язана з розумними компромісами.

Якщо подивитися на типову гру з відкритим світом, то можна побачити в дії принцип Парето - 20% контенту становлять основний шлях гравця, а 80% - це фон. Основний шлях гравця повинен відрізнятися високою якістю і художнім змістом, тому що гравці проведуть на ньому більшу частину часу. Фони, в тому числі великі ліси або пустелі навколо головних міст, не вимагають такої уваги до деталей. Ці 80% є відмінною мета для розумних інструментів розміщення вмісту, які злегка жертвують якістю і художнім оформленням на користь швидкості і простоти створення контенту[1].

Ще один приклад - інструмент створення процедурної рослинності в UE4, який симулює зростання наступних один за одним поколінь рослинності. Прикладами онтогенетических способів можна вважати процедурну генерацію на основі Houdini, в якому технічні художники пишуть правила самостійно, наприклад, як в" Ghost Recon Wildlands ".

1 АНАЛІЗ ПРОБЛЕМНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз проблемної області створення реалістичної поведінки рослинного оточення в іграх

Найкраще для аналізу предметної галузі підійде гра 2020 року «Ghost of Tsushima», як використовує алгоритми поведінки рослинності.

Острів Цусіма складається з тисяч і тисяч крихітних деталей. Світ «Примари Цусіми» істотно відрізняється від попередньої гри, inFAMOUS: Second Son, насиченою яскравими візуальними ефектами. Якщо коротко описати суть роботи, вона полягає у вирішенні проблем графіки та дизайну за допомогою технологій. Проаналізуємо методи, які ми використовували для створення візуальних ефектів «Примари Цусіми».

Було виділино кілька ключових областей, де було проведено поліпшення, взявши за основу вимоги проекту. Перше, що зробили, - підвищити рівень інтерактивностів системі частинок. Розробники провели величезну роботу над системами частинок в грі Second Son - там вони використовувалися для відображення широкого спектру магічних сверхспособностей. Однією з основних цілей в роботі над «Примарою Цусіми» був висновок цієї системи на новий рівень інтерактивності. З самого початку знали, що вітер буде найважливішим елементом, який необхідно реалізувати в наших системах частинок. Також додали в гру тварин, створити епічні пейзажі, передавши всю різноманітність природи, плюс згадані «бруд, кров і сталь» хотілося зробити відчутними: персонажі не можуть вийти з битви, що не забруднити. Підвищити інтерактивність частинок можна було багатьма шляхами, але всі вони вимагали додаткової інформації про ігровому світі. Який саме інформації? Ось, наприклад: звичайний вітер, вітер від рухів персонажа[2].

Другий глобальним завданням була реалізація всіх систем в глобальному масштабі . Світ «Примари Цусіми» набагато об'ємніший, ніж в Second Son, а команда візуальних ефектів Sucker Punch більшу частину часу складалася з

двох чоловік. А це означало необхідність максимально автоматизувати оновлення контенту. Також потрібно було підтримувати мінливу погоду на величезних просторах 24 години на добу. І нарешті, ми вручну ввели в гру кілька елементів, які допомагають гравцеві досліджувати острів Цусіма.

Одне з основних завдань спочатку полягала в тому, що в грі все повинно рухатися. Зрозуміло, що саме тут в нагоді б частинки - опадає листя, пилок в повітрі. Але щоб створити ілюзію дме вітру, необхідно задіяти безліч злагоджено працюють систем. Крім частинок, є дерева, трава, тканина, мотузки і все, що може рухатися на вітрі. Всі ці елементи були узгоджені так, щоб рухатися відповідно при різному вітрі. Було інтегрован вплив вітру практично в кожен візуальний ефект. Коли вибухає бомба чи загоряється багаття, дим завжди пливе в правильному напрямку. Вітер впливає і на вогонь, на іскри - так майже на все. При збільшенні швидкості вітру додається додаткова турбулентність.

За допомогою вітру булостворено «дороговказного вітру», що показує напрямок на ціль, а вже маршрут кожен гравець вибирає для себе сам. Зробивши так, щоб частинки «огортали» ландшафт, піднімалися й опускалися, повторюючи обриси гір і долин. Однак в силу ряду причин з цим виникли проблеми. Перша полягала в різноманітності моделей каменів, використовуваних для гірських обривів. Моделі скель не включені в інформацію про ландшафті, туди входить тільки базова динамічна мозаїчна сітка. Виходило, що частинки входять в усі скелі і зникають, а значить, гравцеві важче слідувати за «дороговказом вітром». Крім того, коли частинки настільки ідеально огинають елементи ландшафту, це виглядає якось неприродно. При наступній спробі місцевість стала свого роду базою: на підйомі частки несуться вгору, а коли місцевість йде під ухил, повільно дрейфують вниз. Коли на шляху виникають пагорби, швидкість підйому частки вгору росте. Для цього довелося провести кілька тестів і подивитися, як частинки будуть вести себе далі по шляху руху. У кожному з цих тестів з'ясовували, наскільки близько частка знаходиться до землі і не йде чи в неї. Якщо частка занадто близько до землі

або йде в неї в найближчих точках, значить, їй потрібно надати додаткову вертикальну швидкість. І останнє. У дороговказною вітру можуть бути різні елементи, в залежності від середовища; в полях це трава і пушинки, в лісах - листя, в зонах пожеж - частинки попелу тощо. У дороговказною вітру можуть бути різні елементи, в залежності від середовища; в полях це трава і пушинки, в лісах - листя, в зонах пожеж - частинки попелу тощо. У дороговказною вітру можуть бути різні елементи, в залежності від середовища; в полях це трава і пушинки, в лісах - листя, в зонах пожеж - частинки попелу тощо.

У дерев є листя та інші елементи, які реагують на швидкість вітру. Ця гнучка система, в свою чергу, дозволила провести широке розмаїття дерев і чагарників. На рух стовбурів і гілок ми наклали більш дрібне - тремтіння поверхні листя.

Трави і трав'яні поля також були одним з особливих елементів гри, їх створення зажадало постійної взаємодії між командами рендеринга, оточення і візуальних ефектів. Поля є поєднанням процедурно згенерованих трикутників для трави і модельованих об'єктів - високих стебел і пучків трави. Спочатку розробники спробували використовувати частки, щоб домогтися хвилеподібного руху. Рух трави під вітром моделювалося пересічними дугами. Візуальне якість трави було непоганим, однак подібний рух дерев і кущів виглядало неправильним, а вартість втілення цієї ідеї була більше, ніж ми могли собі дозволити. При другій спробі ми додали в оточення два шари руху. Перший - основний малюнок руху, який змінюється в залежності від напрямку вітру; поверх нього накладено другий шар, текстура, працює для більш дрібних елементів ландшафту, зокрема трави. Перевага такого підходу (крім зниження вартості) полягало в тому, що в результаті рух дерев, чагарників, трави і листя складалося в природну картину[3].

В кінцевому підсумку розробники використовували технологію зміщення частинок трави для відображення її природного колихання від рухів гравця. Це не нова технологія в іграх, але ми її поліпшили - використовуючи систему частинок для управління зміщенням, змогли зробити «поведінку» трави ще

більш природним. Наприклад, використання загасаючої хвилі, пропорційної силі зсуву, дозволяє більш природно показати «поведінку» трави після того, як через неї пройшов гравець: трава повертається до вихідного положення поступово, а не відразу. На рисунку 1.1 показано, як рухається трава в описаних випадках, а також можна побачити відображення системи частинок, які контролюють її рух. Зелена частина сліду відображає зміщення, червона - загасання зміщення. Якщо ви уважно придивитеся, то побачите, що амплітуда зміщення знижується в міру того, як герой йде далі.

Рисунок 1.1 – Поведінка трави з використанням алгоритму загасання

Оскільки вітрі в грі відведена важлива роль, розробники розуміли, що буде потрібно додати листя - багато листя, кружляють на вітрі, і кружляти вони повинні природно і красиво. У будь-який момент гри на екрані можуть виявитися тисячі листів, і всі вони будуть взаємодіяти з вітром, навколишнім середовищем і персонажами.

Щоб листя виглядали природно, розробники витратили масу зусиль на те, щоб вони правильно лягали на землю в різній місцевості. Кожен листок

моделюється у вигляді диска. За допомогою 3D-математики вони змусили листя обертатися в потрібному напрямку і лягати на землю так, як потрібно. Крім моменту приземлення, вони також змоделювали деякі більш «просунуті» моменти поведінки листя. Він лягають на водяну гладь, пливуть за течією, падають вниз разом із струменями водоспаду і через деякий час тонуть.

Ближче до кінця роботи над «Примарою» тестувальники повідомили про дивну помилку: листя падало в багаття і неушкодженими залишалися лежати на землі під вогнем або поруч з ним. Для вирішення проблеми розробники помістили в кожен багаття джерело вітру, на який повинні були реагувати падаючі листя. Висхідний потік не давав листю падати в вогонь. Вийшла відмінна деталь.

У багатьох поєдинках присутні листя, що реагують на рухи гравця і комп'ютерних персонажів. Для цього було застосувано ту ж техніку, що і для руху трави, - дозволили часткам використовувати ту ж інформацію. Таким чином, вдалося змусити листя красиво розлітатися при швидкому русі персонажів. У поєдинках метою не була реалізм, а краса і ефектність. Описану вище систему подій налаштували на моменти, коли значення переміщення гравця були вище порогових. Потім ми створили спад, щоб незабаром після початкового руху листя вляглися - і наступний рух знову піднімало їх в повітря. Також використовували наступний патерн шуму: від пориву вітру листя злітають з землі і знову опускаються. Це робить вітер видимим і підкреслює легкість опалого листя.

Наведу простий приклад: частинки-листя повинні падати тільки в лісистих зонах, а не на луках і полях. Спершу розробники використовували прямолінійний підхід: помістили ефект в дерева, що ростуть на острові. На жаль, незабаром такий підхід був визнаний провальним: ландшафт включав сотні дерев на невеликих просторах. З точки зору геометрії все було прекрасно, але застосовувати систему частинок було занадто дорого. Вони домоглися невеликого успіху, налаштувавши наші частки так, щоб вони з'являлися тільки дуже близько до дерев (в радіусі близько 15 метрів), але при загальній широті

огляду рішення виявилося не надто вдалим. До того ж часто листя виявлялися просто не видно. Коли лист падав з висоти 12 метрів, а гравець їхав на коні, лист просто не потрапляв в поле зору - він досягав землі занадто пізно[4].

Тоді вони вирішили використовувати систему зростання тим же чином, що і в навколишньому середовищі: системи частинок розміщувалися поруч з деревами з використанням тих же масок. Їм потрібно було розмістити накладаються один на одного кола листя в рамках маски, іноді складної форми. Однак при такому розміщенні кіл виникали області, в яких зовсім не було листя: це було викликано правилами зростання. Розробники могли б виправити становище, створивши додаткові джерела частинок меншого розміру, але це вимагало б додаткових ресурсів. Також було б важко виправити становище для окремих ділянок: якби ми змінили правила зростання, щоб закрити «дірки» або відрегулювати інтенсивність частинок, це призвело б до несправностей в областях лісу, де колись все працювало нормально. І все ж ця техніка працювала, і, незважаючи на деякі недоліки, як зображено на риисунку 1.2.

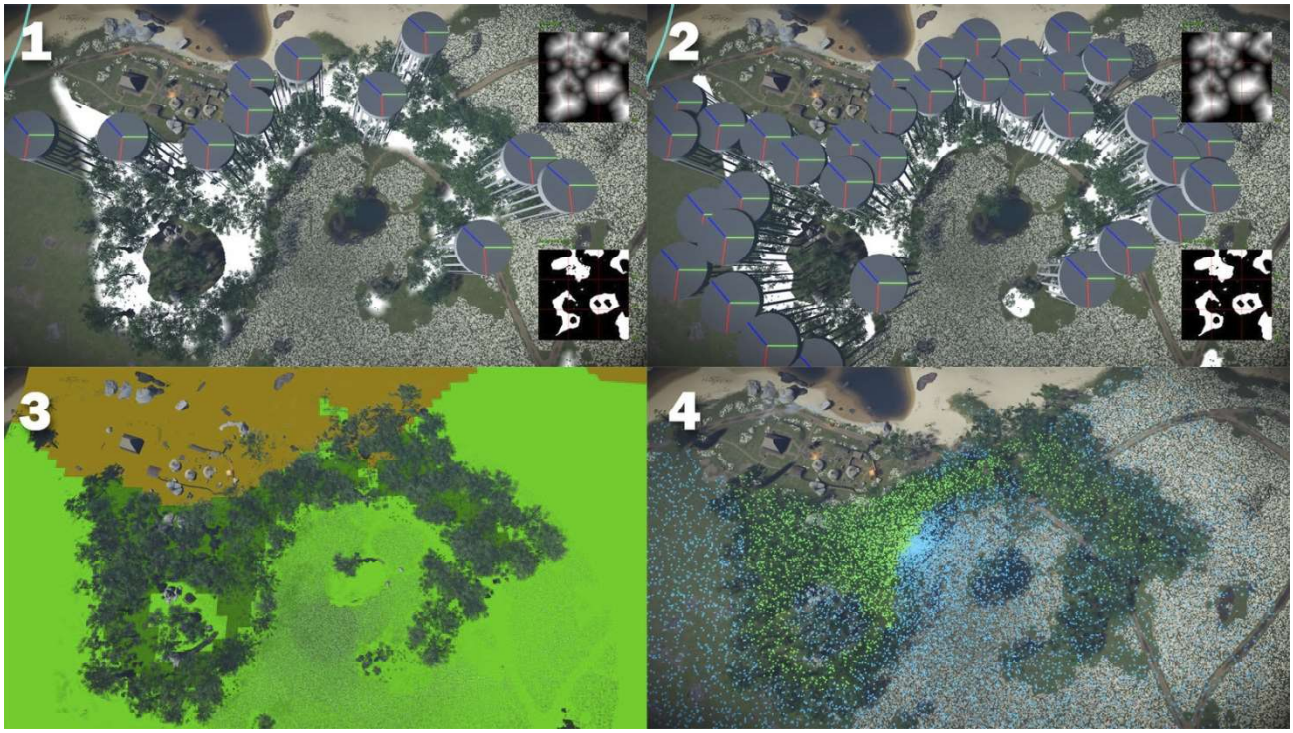

Рисунок 1.2 – Ілюстрації додавання часток

На першому скріншоті ви можете бачити, як для додавання частинок використовується інструмент зростання: частинки з'являються там, де ростуть дерева. Тут можна помітити великі розриви, що нас зовсім не влаштовувало. На другому скріншоті значно збільшено кількість джерел частинок. Це працює, але вимагає великої кількості ресурсів, які могли бути використані десь ще. На третьому скріншоті показаний третій підхід, карта биома. Програмісти рендеринга працювали над однією функцією для команди освітлення. Ця функція повинна була використовувати дані з масок зростання навколишнього середовища, отримуючи до них доступ в реальному часі. Так система частинок отримала можливість «дізнатися», в якому саме биоме виникає та чи інша частка. Четвертий скріншот показує систему налагодження і то, як частки зчитують дані. Сині області - це луки, а зелені - ліс.

У перший запуск все працювало трохи незграбно, але коли застосували текстуру шуму для модифікації інформації, система стала набагато більш органічною і зручною у використанні. Це стало ядром нової системи створення атмосфери - в поєднанні з іншими функціями, такими як визначення положення на місцевості, матеріалу і напрямку вітру. У всіх стандартних зонах ми змогли видалити з навколишнього середовища сотні систем частинок і замінити їх єдиною системою, наступного за камерою і набагато краще відповідної Біоми.

1.2 Аналіз аналогів створення шейдера трави в рушії Unity

Геометричні шейдери — це необов'язкова частина конвеєра рендеринга. Вони виконуються після вершинного шейдерів (або шейдера тесселяції, якщо використовується тесселяція) і до того, як вершини обробляються для фрагментного шейдера[5].

Блок-схуму роботи шейдеру «Графічний конвеєр Direct3D 11» зображено на рисунку 1.3.

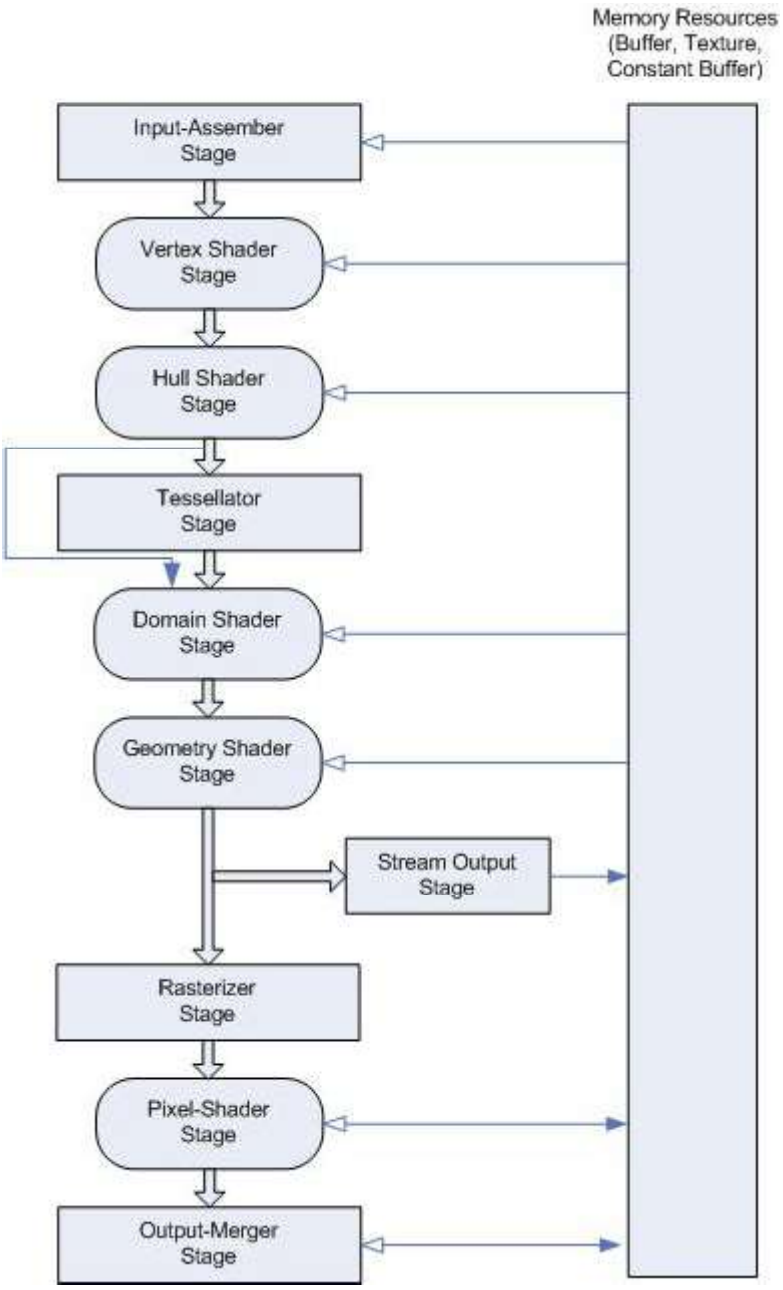

Рисунок 1.3 – Графічний конвеєр Direct3D 11

Геометричні шейдери отримують на вході одиночний примітив і можуть згенерувати нуль, один або безліч примітивів. Ми почнемо з того, що напишемо геометричний шейдер, одержує на вході вершину (або точки), а на вихід подає один трикутник, який представляє травинку.

Представлений на рисунку 1.4 код оголошує геометричний шейдер під назвою geo з двома параметрами. Перший, triangle float4 IN[3]повідомляє, що він буде брати в якості введення один трикутник (що складається з трьох

точок). Другий, типу TriangleStream, налаштовує шейдер для виведення потоку трикутників, щоб кожна вершина використовувала для передачі своїх даних структуру geometryOutput.

```
// Add inside the CGINCLUDE block.
struct geometryOutput
{
        float4 pos : SV_POSITION;
};

[maxvertexcount(3)]
void geo(triangle float4 IN[3] : SV_POSITION, inout TriangleStream<geometryO
utput> triStream)
{
}

…

// Add inside the SubShader Pass, just below the #pragma fragment frag line.
#pragma geometry geo
```

Рисунок 1.4 – Геометричний шейдер під назвою geo

Крім того, ми додаємо останній параметр у квадратних дужках над оголошенням функції: [maxvertexcount(3)]. Він каже GPU, що ми будемо виводити (але не зобов'язані це робити) не більше 3 вершин. Також ми робимо так, щоб SubShader використовував геометричний шейдер, оголосивши його всередині Pass.

Це дало дуже дивні результати. При переміщенні камери стає ясно, що трикутник рендерится в екранному просторі. Це логічно: оскільки геометричний шейдер виконується безпосередньо перед обробкою вершин, він забирає у вершинного шейдера відповідальність за те, щоб вершини виводилися в просторі усікання[6].

На рисунку 1.5 надається приклад коду з усіканням вершин.

```
// Update the return call in the vertex shader.
//return UnityObjectToClipPos(vertex);
return vertex;

…

// Update each assignment of o.pos in the geometry shader.
o.pos = UnityObjectToClipPos(float4(0.5, 0, 0, 1));

…

o.pos = UnityObjectToClipPos(float4(-0.5, 0, 0, 1));

…

o.pos = UnityObjectToClipPos(float4(0, 1, 0, 1));
```

Рисунок 1.5 – Приклад коду з усіканням вершин

Тепер, як зображено на рисунку 1.6, трикутник рендерится в світі правильно. Однак, схоже, він створюється тільки один. Насправді, по одному трикутнику отрісовиваємих для кожної вершини нашого меша, але позиції, визначені вершин трикутника, постійні — вони не змінюються для кожної вхідної вершини. Тому всі трикутники розташовуються один на іншому.

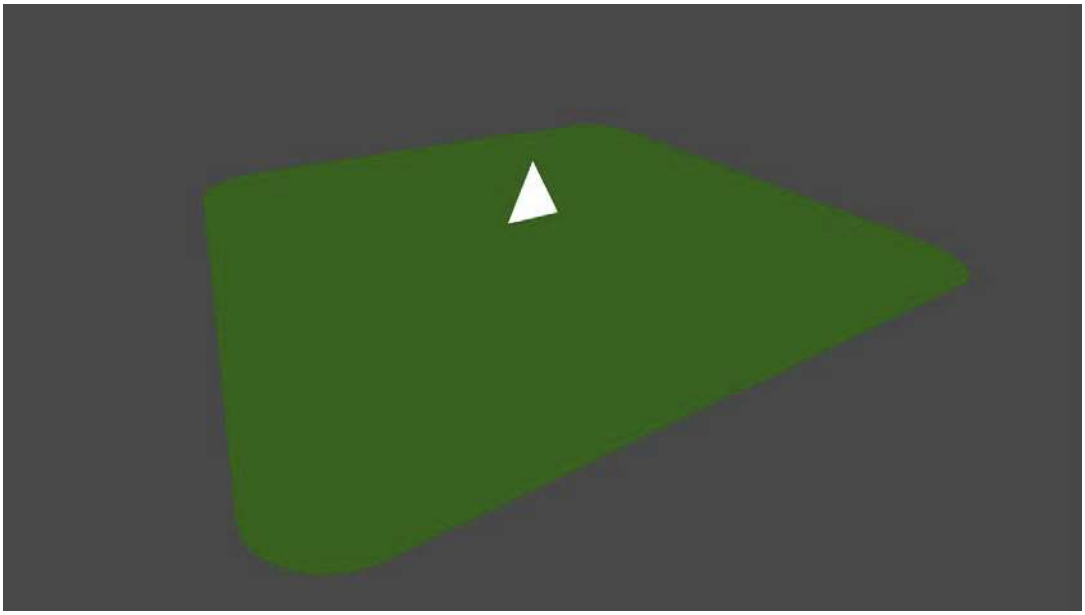

Рисунок 1.6 – Видображення роботи програмного коду

Виправимо це, зробивши виходять позиції вершин зміщеннями щодо вхідної точки. Для цього внесемо зміни в програмний код, як показано на рисунку 1.7.

```
// Add to the top of the geometry shader.
float3 pos = IN[0];

…

// Update each assignment of o.pos.
o.pos = UnityObjectToClipPos(pos + float3(0.5, 0, 0));

…

o.pos = UnityObjectToClipPos(pos + float3(-0.5, 0, 0));

…

o.pos = UnityObjectToClipPos(pos + float3(0, 1, 0));
```

Рисунок 1.7 – Зміна позицій вершин

Трикутники тепер відмальовує правильно, як зображено на рисунку 1.8, а їх основа розташоване у випромінює їх вершині. Перш ніж рухатися далі, зробимо об'єкт GrassPlane неактивним у сцені, а об'єкт GrassBall зробимо активним. Ми хочемо, щоб трава правильно генерувалася на різних типах поверхонь, тому важливо протестувати її на мешах різної форми.

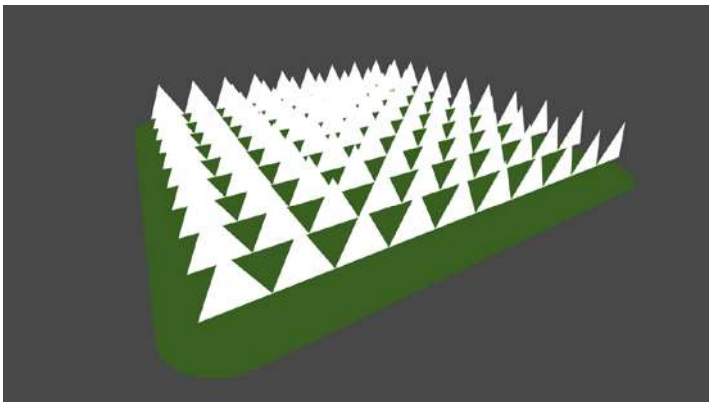

Рисунок 1.8 – Видображення зміни розташування вершин

Поки всі трикутники випускаються в одному напрямку, а не назовні від поверхні сфери. Щоб вирішити цю проблему, ми будемо створювати травинки в дотичному просторі.

В ідеалі ми б хотіли створювати травинки, встановлюючи різну ширину, висоту, кривизну і поворот, не враховуючи кут поверхні, з якої випускається травинка. Простіше кажучи, ми задамо травинку в просторі, локальному до випромінює її вершині, а потім перетворимо її так, щоб вона була локальною до мешу. Цей простір називається дотичним простором, дивись рисунок 1.9.

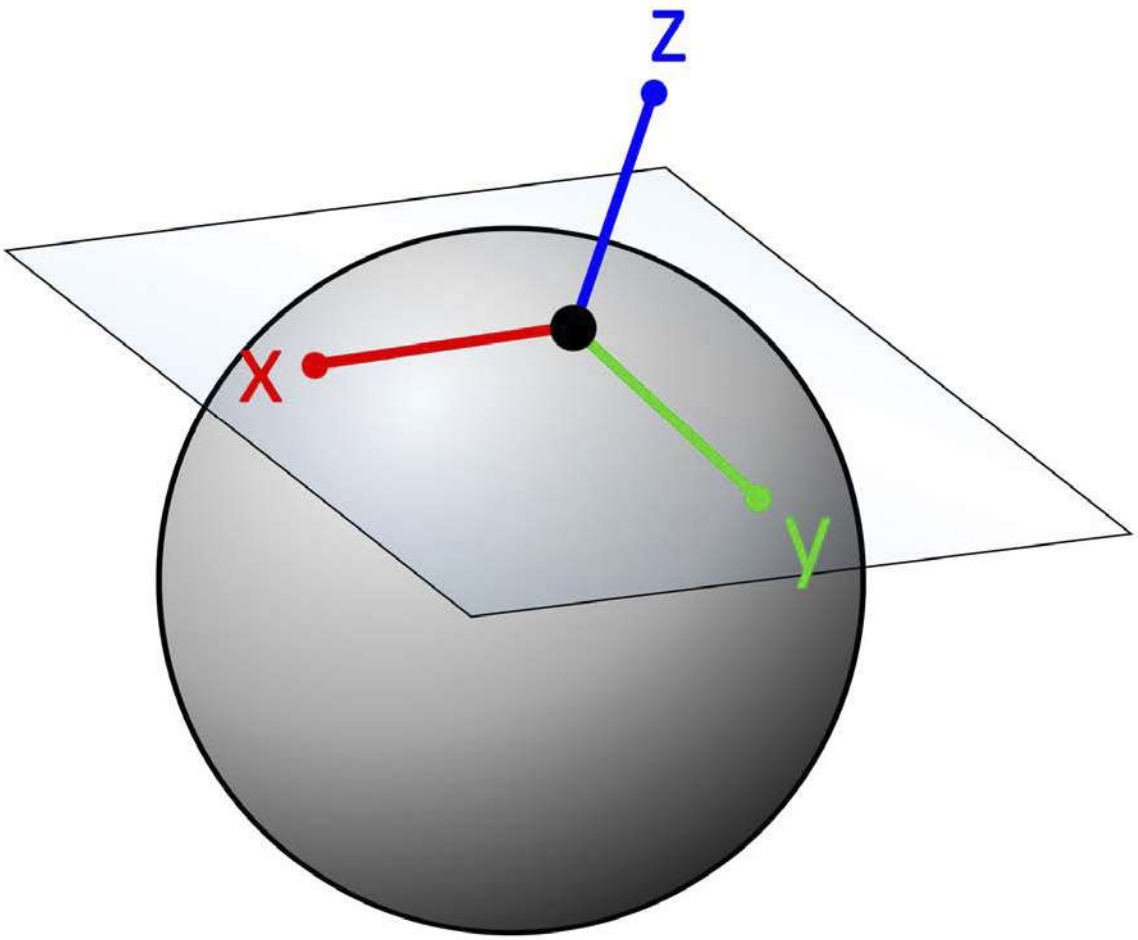

Рисунок 1.9 – Дотичний простір

В дотичному просторі осі X, Y і Z задаються відносно нормалі і позиції поверхні (в нашому випадку вершини).

Як і будь-який інший простір, ми можемо задати дотичний простір вершини трьома векторами: right, forward і up. З допомогою цих векторів ми можемо створити матрицю для повороту травинки з стосовного в локальний простір.

Можна отримати доступ до векторів right і up, додавши нові вхідні дані вершин, як наведено на рисунку 1.10.

```
// Add to the CGINCLUDE block.
struct vertexInput
{
        float4 vertex : POSITION;
        float3 normal : NORMAL;
        float4 tangent : TANGENT;
};

struct vertexOutput
{
        float4 vertex : SV_POSITION;
        float3 normal : NORMAL;
        float4 tangent : TANGENT;
};

…

// Modify the vertex shader.
vertexOutput vert(vertexInput v)
{
        vertexOutput o;
        o.vertex = v.vertex;
        o.normal = v.normal;
        o.tangent = v.tangent;
        return o;
}

…

// Modify the input for the geometry shader. Note that the SV_POSITION seman
tic is removed.
void geo(triangle vertexOutput IN[3], inout TriangleStream<geometryOutput> t
riStream)

…

// Modify the existing line declaring pos.
float3 pos = IN[0].vertex;
```

Рисунок 1.10 – Доступ до векторів right і up

Третій вектор можна обчислити, взявши векторний добуток між двома іншими. Векторне твір повертає вектор, перпендикулярний до двох вхідних векторів, як наведено на рисунку 1.11.

```
// Place in the geometry shader, below the line declaring float3 pos.
float3 vNormal = IN[0].normal;
float4 vTangent = IN[0].tangent;
float3 vBinormal = cross(vNormal, vTangent) * vTangent.w;
```

Рисунок 1.11 – Доступ до третього вектору

Маючи всі три вектори, ми можемо створити матрицю для перетворення між дотичним та локальним просторами. Ми будемо множити кожну вершину травинки на цю матрицю перед передачею в UnityObjectToClipPos, який очікує вершину в локальному просторі, як зображено на рисунку 1.12.

```
// Add below the lines declaring the three vectors.
float3x3 tangentToLocal = float3x3(
        vTangent.x, vBinormal.x, vNormal.x,
        vTangent.y, vBinormal.y, vNormal.y,
        vTangent.z, vBinormal.z, vNormal.z
        );
```

Рисунок 1.12 – Матриця для перетворення між дотичним та локальним просторами

Перш ніж використовувати матрицю, ми перенесемо код виведення вершин у функцію, щоб не писати знову і знову однакові рядки коду. Це називається принципом DRY, або don't repeat yourself («не повторюйте»)

Нарешті, ми помножимо вихідні вершини на матрицю tangentToLocal, правильно вирівнявши їх з нормаллю їх вхідної точки.

Це вже більше схоже на те, що нам потрібно, але не зовсім вірно. Проблема тут полягає в тому, що спочатку ми призначили напрям «up» (вгору)

осі Y; проте в дотичному просторі напрямок «вверх» зазвичай розташовується уздовж осі Z. Зараз ми внесемо ці зміни, як зображено на рисунку 1.13.

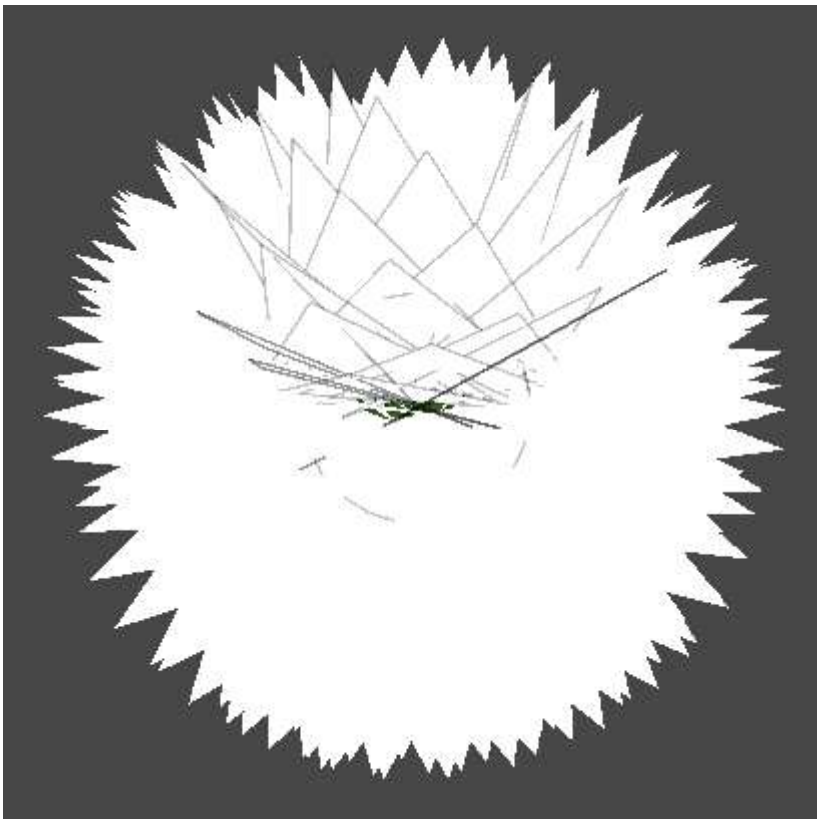

Рисунок 1.13 – Результат перетворення з локального до дотичним простораму

Щоб трикутники більше походили на травинки, потрібно додати квітів і варіативності. Почнемо ми з додавання градієнта, що йде з верхівки травинки вниз.

Мета полягає в тому, щоб дозволити художнику задати два кольори — верхівки та низу, і виконувати інтерполяцію між цими двома кольорами він кінчика до підстави травинки. Ці кольори вже визначені в файлі шейдера як _TopColor і _BottomColor. Для їх правильного семплювання потрібно передати фрагментному шейдеру UV-координати.

Ми створили UV-координати для травинки у формі трикутника, дві вершини основи якого знаходяться ліворуч і праворуч внизу, а вершина кінчика розташована по центру вгорі, як зображено на рисунку 1.14.

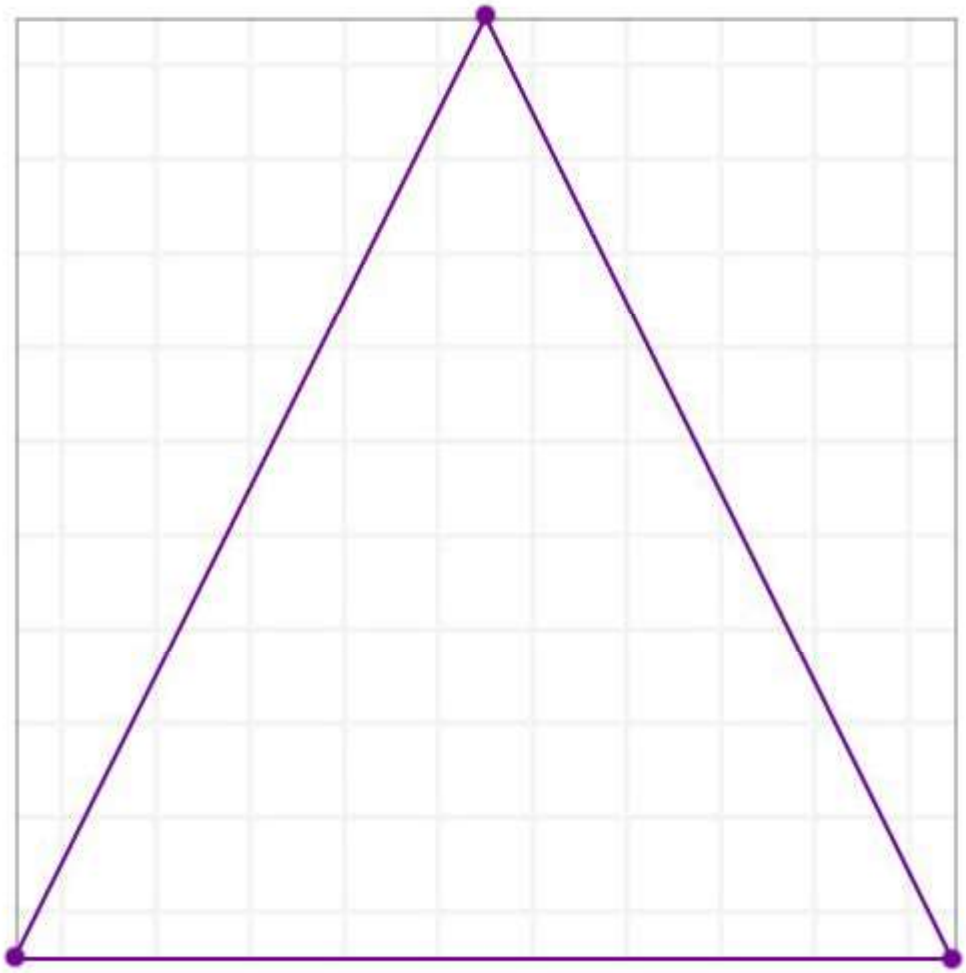

Рисунок 1.14 – UV-координати трьох вершин травинок

Тепер ми можемо сэмплировать верхній і нижній кольору під фрагментном шейдере за допомогою UV, а потім інтерполювати їх за допомогою lerp. Також нам знадобиться модифікувати параметри фрагментного шейдера, зробивши вхідними даними geometryOutput, а не тільки позицію float4

Щоб створити варіативність і надати траві більш природний вигляд, ми змусимо кожну травинку дивитися у випадковому напрямку. Для цього нам знадобиться створити матрицю повороту, повертаючу травинку на випадкову величину навколо її осі up[7].

У шейдерів є дві функції, які допоможуть нам це зробити: rand, генерує випадкове число з тривимірного введення, і AngleAxis3x3, одержує кут (у радіанах) і повертає матрицю, яка виконує поворот на цю величину навколо

зазначеної осі. Остання функція працює точно так само, як функція C# Кватерніонів.AngleAxis (тільки AngleAxis3x3 повертає матрицю, а не кватернион).

Функція rand повертає число в інтервалі 0...1; ми помножимо його на 2 Pi, щоб отримати повний інтервал кутових значень.

Ми використовуємо входить позицію pos як seed для довільного повороту. Завдяки цьому кожна травинка буде мати власний поворот, постійний в кожному кадрі.

Поворот можна застосувати до травинки, помноживши його на утворену матрицю tangentToLocal. Врахуйте, що множення матриць не є коммутативным; порядок операндів важливий.

Зараз окремі травинки задаються одним трикутником. На великих відстанях це не проблема, але поблизу травинки виглядають дуже жорсткими і геометричными, а не органічними і живими. Ми виправимо це, побудувавши травинки з декількох трикутників і зігнувши їх вздовж кривої.

Кожна травинка буде поділено на кілька сегментів. Кожен сегмент буде мати прямокутну форму і складатися з двох трикутників, за винятком верхнього сегмента — він буде одним трикутником, що позначає кінчик травинки.

Поки ми виводили тільки три вершини, створюючи єдиний трикутник. Як же при наявності більшої кількості вершин геометричний шейдер дізнається, які з них потрібно з'єднувати і утворювати трикутники? Відповідь знаходиться в структурі даних triangle strip. Перші три вершини з'єднуються і утворюють трикутник, а кожна нова вершина утворює трикутник з попередніми двома.

Подразделенная травинка, представлена у вигляді triangle strip і створювана по одній вершині за раз. Після перших трьох вершин кожна нова вершина утворює новий трикутник з попередніми двома вершинами.

Це не тільки більш ефективно з точки зору використання пам'яті, але і дозволяє легко і швидко створювати в коді послідовності трикутників. Якби ми

хотіли створити кілька смуг трикутників, то могли б викликати для TriangleStream функцію RestartStrip.

Перш ніж ми почнемо виводити з геометричного шейдера більше вершин, нам потрібно збільшити maxvertexcount. Ми скористаємося конструкцією #define, щоб дозволити авторові шейдера управляти кількістю сегментів і обчислювати з нього кількість нових вершин.

Спочатку ми задаємо кількість сегментів рівним 3 і оновлюємо maxvertexcount, щоб обчислити кількість вершин на підставі кількості сегментів.

Для створення сегментованої травинки ми використовуємо цикл for. Кожна ітерація циклу буде додавати по дві вершини: ліву і праву. Після завершення верхівки ми додамо останню вершину на кінчику травинки.

1.3 Постановка задачі

Метою цієї роботи є підвищення оптимізація поведінки рослинного покриття на ігровій сцені.

Об’єктом дослідження є алгоритмізація поведінки рослин в ігровому світі.

Для вирішення практичної задачі моделювання поведінки рослинного оточення необхідно створити програмний модуль на рушії UE4, яка буде складатися з наступних частин:

- ігрове середовище на якому будуть розміщуватися рослини;
- рослини, поведінку яких ми будемо моделювати;
- об’єкти та ефекти, що будуть впливати на поведінку рослин;
- вітер, шо симулює поведінку рослин без зовнішнього втручання;

– тіні від хмар, що створють реалістичну атмосферу на сцені.

Програмний продукт який буде створено, повинен реалізовувати наступні функції:

– поведінка рослин в спокої;
– вплив актора на рослини;
– вплив вибухів та магічних ефектів на рослини;
– вплив вітру на рослини;
– бленд усіх ефектів;
– реалістичні тіні від хмар.

Програмне забезпечення повинна відповідати основним вимогам, що висуваються до подібних програмних модулів, такі як стійкість до навантаження, оптимальне використання ресурсів та оптимізація графічної складової.

Таким чином, протягом виконання кваліфікаційної роботи магістра необхідно виконати такі задачі:

– провести аналіз проблемної галузі та базове моделювання предметної галузі;
– розробити алгоритм поведінки рослин на ігровій сцені в стані спокою;
– адаптувати алгоритм поведінки під зовнішній вплив гравця та інших ігрових об’єктів;
– організувати алгоритм впливу вітру на рослини;
– створити бленд впливу декількох факторів на рослини;
– організувати оптимізацію алгоритму;
– створити можливість вільної інтеграції програмного модулю в інщі проекти на рушії UE4.

Після вирішення усіх поставлених задач слід сформувати звіт у вигляді пояснювальної записки до кваліфікаційної роботи магістра.

2 ФОРМУВАННЯ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

2.1 Призначення розробки

Програмна система призначена для поліпшення алгоритму поведінки рослин в комп'ютерних іграх на рушії UE4. Головною відмінністю від конкурентів є організація бленду усіх ефектів впливу на рослини, враховуючи їх вагу, радіус та силу. Програмний модуль буде призначений для користування розробниками ігор на рушії UE4.

2.2 Вимоги до програмного модулю

Програмна модуль повин мати компонентно орієнтовану архітектуру, що поліпшить його інтеграцію в інші ігрові програмні продукти. Перш за все, потрібно розробити модель поведінки рослин, та врахувати які саме фактори будуть впливати на їх поведінку. Також модуль повинен мати механізми легкої інтеграції в інщі програмні продукти та простий інтерфейс налаштування. Обов'язково модуль має бути максимально оптимізован, в зв'язку з чим інтегровані моделі повинні мати можливість бути заміненими на різно полігональні моделі, відповідно до вимог користувача.

Параметри впливу повинні мати доступний та зрозумілий інтерфейс корегування для того, щоб користувач мав можливість підлаштовувати їх під себе.

Програмний модуль повинен мати можливісь маштабування згідно вимог користувача, щоб враховувати ще більше параметрів впливу на рослинність.

Використання технологій блюпринт дозволить з легкістью налаштовувати компоненти модуля, регулюючи іх вплив на рослини та друг на друга.

2.3 Вимоги до оптимізації

Програмний модуль повинен використовувати всі принципи оптимізації шейдерів.

Кількість вертексів на сцені не повинно привищувати максимально допустиму кількість.

Модуль не повинен використовувати функції створеня та виделення обїектів, базуючись на основних принципах роботи рушія.

Draw Calls не повинні привищувати допустимих значень видрисовки на обній сцені камери.

Використання Instance Meshes дозволить знизити навантаження при роботі зі сценою.

3 ОПИС ПРИЙНЯТИХ ПРОЕКТНИХ РІШЕНЬ

3.1 Аналіз матеріалів та шейдерів

Material Functions (Функції матеріалів) - невеликі сніппети (макроси) графів матеріалів, які можуть бути збережені для подальшого використання в інших матеріалах. Функції можуть бути використані в складних матеріалах, при цьому утримуючи в собі не менш складний граф матеріалу, тим самим допомагаючи художникам в створенні матеріалів[8].

Найкраще відредагувати одиночну функцію, яка використовується у всій мережі матеріалів. Тому, якщо вам необхідно буде змінити функцію, вам не доведеться переробляти матеріали які використовують цю функцію.

Функції матеріалів - це невеликі фрагменти які можна записати в функцію і використовувати між декількома матеріалами. Вони створені щоб спростити процес створення матеріалу надаючи швидкий доступ до часто використовуваних мереж створеним з різних виразів. Наприклад, якщо ви помічаєте, що часто створюєте мережу для хаотичного панорамування, тоді буде набагато простіше зберегти цю частину мережі у вигляді матеріал-функцій і просто використовувати її там, де вона вам знадобиться.

Функції редагуються в редакторі матеріалів, також як і звичайний матеріал, але з деякими обмеженнями на тих виразах, які можуть бути використані. При правильному використанні, вони можуть скоротити надмірність матеріалу, яка, в свою чергу, знижує витрати сил художника на збереження схожості однакових виразів і неминучі помилки які з'являються коли один зразок втрачається в процесі модифікації.

Функції матеріалів, також є Ассет, які можуть з'явитися в контент браузері. Їх графи будуть відрізнятися від звичайних матеріалів. Замість основного вираження матеріалу, функції матеріалу мають виходять вирази, які представляють собою виходять з'єднання кінцевої функції.

Замість основного вирази, яке має матеріал, функції матеріалу мають виходять вирази, які представляють собою виходять з'єднання для кінцевої функції.

Можливо, буде простіше уявити функцію як якусь оболонку. Ви можете додати стільки входів і виходів, скільки вважаєте за потрібне (дивись рисунок 3.1). Підставою функції є те, що відбувається між її входами і виходами. У цьому прикладі вона бере два шари і змішує їх разом, як змішує зображення фотошоп (в режимі змішування screen). Функція приховує деталі від художника, який можливо захоче використовувати її, так що їм не обов'язково розбиратися в логіці змішування зображення використовувану в операції змішування. Якщо хтось захоче пізніше змінити спосіб змішування зображення, вони можуть застосувати його до цієї функції і всі зміни автоматично перенесуться на всі матеріали, які використовують цю функцію.

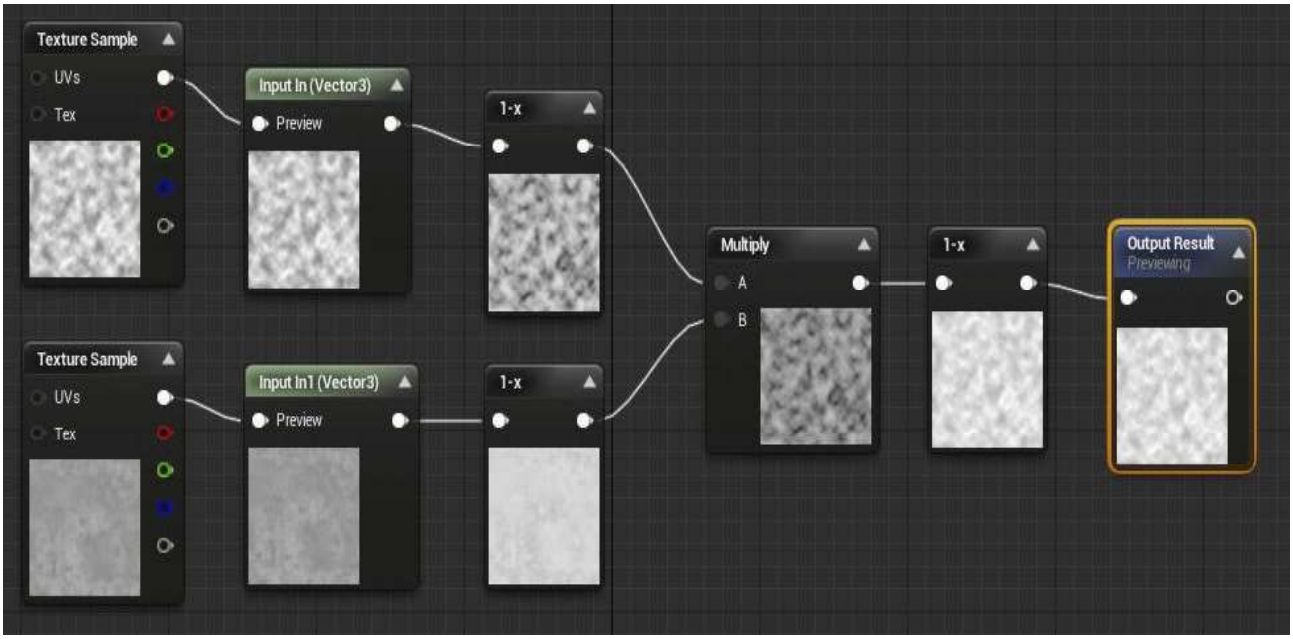

Рисунок 3.1 – Функція матеріалу

Як видно вище, то, що відбувається між входом і виходом повністю залежить від вас і буде визначено будь-якою мережею звичайних нод матеріалу. Проте, коли ви розмістите функцію матеріалу в матеріалі, ви будете

бачити тільки вираз функції з входами і виходами, як можна бачити на рисунку 3.2.

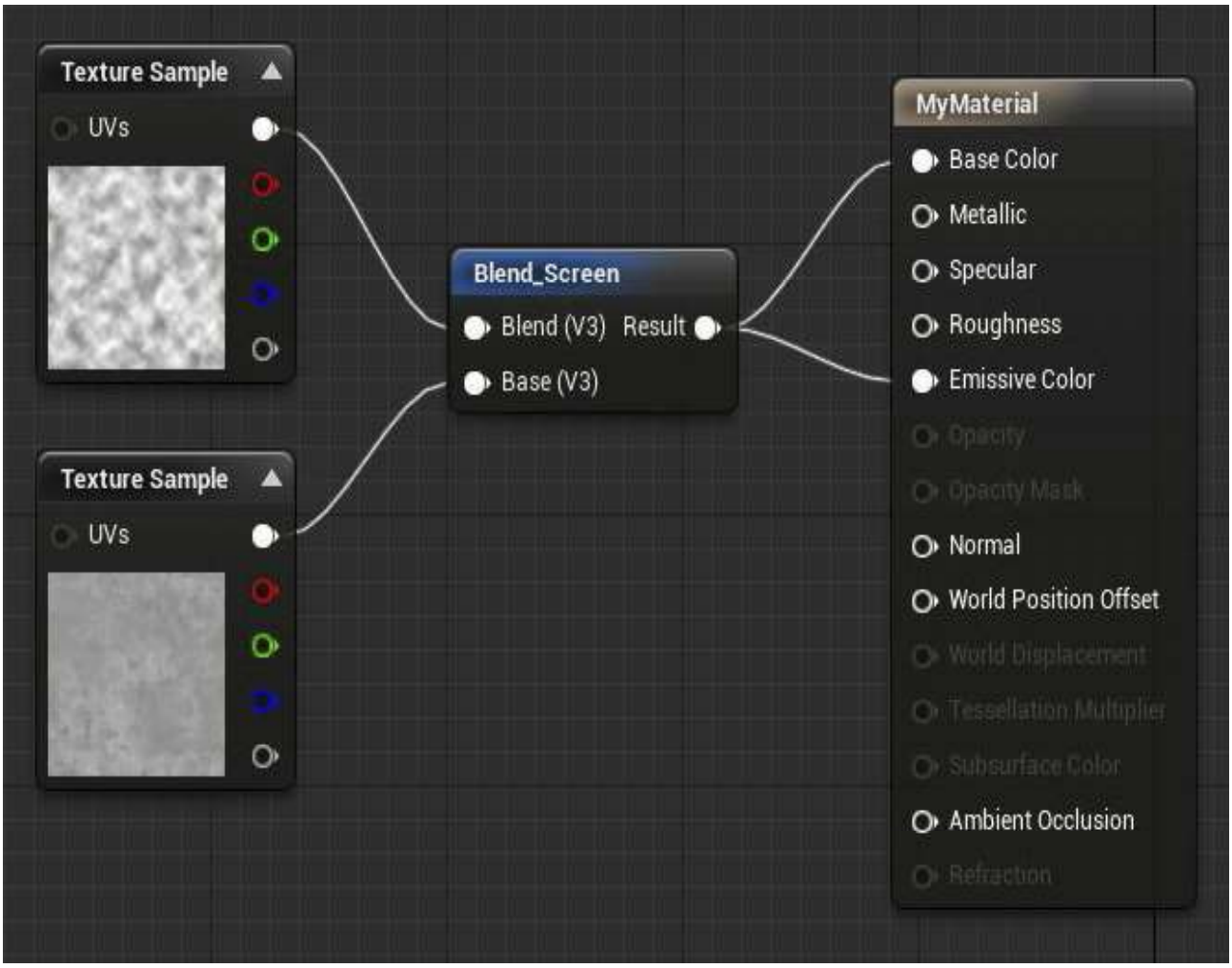

Рисунок 3.2 – Функція створення матеріалу

Коли ваша матеріал-функція законченна, її можна опублікувати в бібліотеці функцій матеріалу для швидкого доступу при створенні матеріалу. Бібліотека матеріал-функцій, це вікно в редакторі матеріалів, яке містить категорії і фільтрується список доступних матеріал-функцій. Цей список наповнений будь-якими завантаженими функціями, але також з будь-якої функції яка не завантажена, але була знайдена через контент браузер база даних яка використовується через контент браузер.

Нижче знаходяться вираження матеріалу, які відносяться до функцій матеріалу, поряд з їх призначенням:

MaterialFunctionCall - дозволяє використовувати додаткову функцію з іншого матеріалу або функції. Додаткові входи і виходи виразів стають входами і виходами самовиразу функції.

FunctionInput - може бути доданий тільки в функції матеріалу, де створює входи функції.

FunctionOutput - може бути доданий тільки в функції матеріалу, де створює виходи функції.

TextureObject -

Корисно, коли необхідно надати текстуру за замовчуванням для вхідної функції текстури всередині функції.Це текстура не є зразком текстури, тому вона повинна бути використана в поєднанні з нодою TextureSample .

TextureObjectParameter - визначає параметри текстури і виводить текстурний об'єкт, використовується в матеріалах які називаються функціями з текстурними входами. Цей вислів не є зразком текстури, так що його слід використовувати в поєднанні з нодою TextureSample .

StaticSwitch - виробляє компіляцію часу обрану між двома входами, засновану на вхідному значенні.

StaticBool - зручний при наданні стандартного булевого значення для статичної булевої функції на вході самої функції. Це вираження не перемикається між іншими, тому його потрібно використовувати в поєднанні з виразом StaticSwitch .

StaticBoolParameter - Визначає статичний булевий параметр і виводить статичне логічне значення, яке використовується в матеріалах, яке викликає функцію зі статичними булеві входами. Це вираження не перемикається між іншими, тому його слід використовувати в поєднанні з виразом StaticSwitch[9].

Оскільки функції матеріалів укладені в оболонку, то залишається за користувачем переконатися, що інформація може передаватися у всередину і з них. Це виконується за допомогою виразів FunctionInput і FunctionOutput . Розуміння цих виразів критично для використання матеріал-функцій.

Самі по собі вирази FunctionInput , FunctionOutput виглядають так, як зображено на рисунку 3.3.

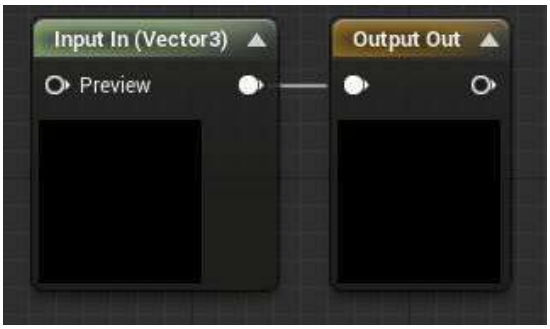

Рисунок 3.3 – Функція FunctionInput

З боку, проте коли функція використовується в матеріалі, ці вирази служать в якості контактів на вході і виході (дивись рисунок 3.4).

Рисунок 3.4 – Функція PassThroungh

Як згадано, вираження Material Input служать як ворота через які інформація надходить в функцію матеріалу. Отримана функція може мати будь-яку кількість цих вхідних виразів, кожне з котори з відповідає вхідному контакту який з'явиться на самій функції.

Output Name – являє ім'я для виходу, яке буде відображатися поза функцією.

Description – являє опис цього входу, яке буде видно коли користувач наведе курсор на виходить контакт функції.

Sort Priority – це число використовується для контролювання порядку в якому входить контакт буде перераховуватися на вираженні функції. У порядку від меншого до більшого.

Input Pins (Unlabeled) – забезпечує введення даних які були оброблені в функції. Ці дані будуть відправлені з функції для використання в матеріалі.

Входи мають специфічний тип потрібного для будь-яких виразів пов'язаних з ними. Він задається через властивість Input type в вираженні FunctionInput . Поза функції, цей тип потім відображається в декількох повідомленнях поруч з вхідними з'єднаннями коли функція використовується в матеріалі. В цьому випадку, обидва входи де Vector3 відображаються як V3 . Все що підключено до входу при використанні в матеріалі має бути Перетворювані до типу входу інакше ви отримаєте помилку. На рисунку 3.5 зображено налаштування Blend_colorBurn.

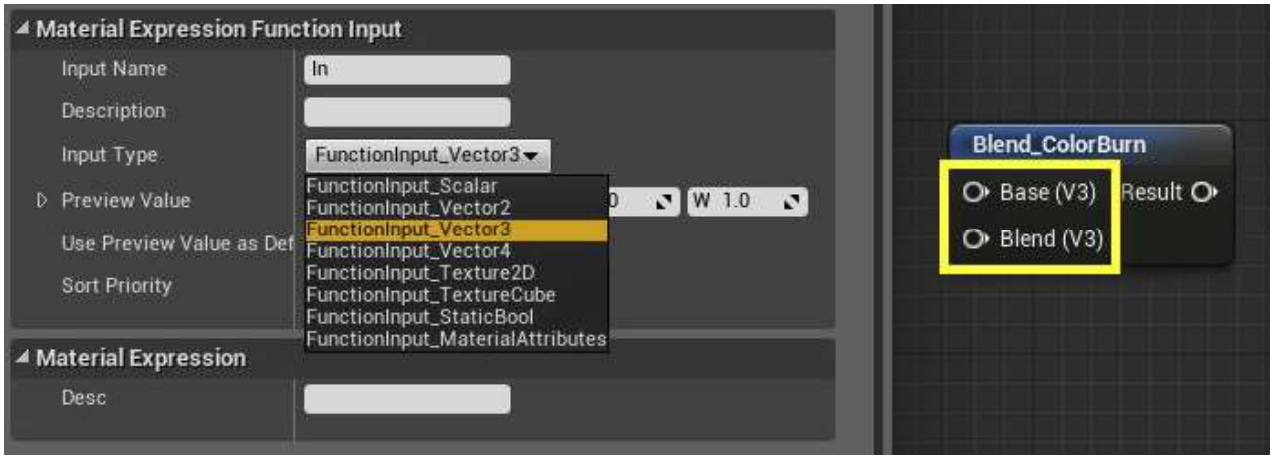

Рисунок 3.5 – Налаштування Blend_colorBurn

Нижче, на рисунку 3.6, перераховані доступні типи входів і їх асоціативні абревіатури.

Тип входу	абревіатура
Scalar	S
Vector2	V2
Vector3	V3
Vector4	V4
Texture2D	T2D
TextureCube	TCube
StaticBool	B

Рисунок 3.6 – Доступні типи входів

При створенні функції, зніміть позначки з усіх виразів щоб отримати доступ до властивостей самої функції.

При редагуванні функції матеріалу, вікно попереднього перегляду відображається в незалежності від виразу відображуваного в попередньому перегляді. Ви можете натиснути правою кнопкою миші по будь-якому вираженню і вибрати Start Previewing Node щоб відобразити результат вашої мережі на даному моменті.

Вхідні вирази функції мають деякі опції для вказування значень попереднього перегляду, так як вони не знають які значення будуть використовуватися в матеріалі. Кожен вхід має вбудоване значення предпросмотра яке може бути використане для відображення постійної для змінних вхідних типів.

Входи функції також мають «preview» коннектор, як показано на рисунку 3.7, який дозволить вам переписати вбудовані значення на будь-яке значення яке збігається з типом входу. У цьому прикладі, зразок текстури був використаний щоб надати його для входу float 3 .

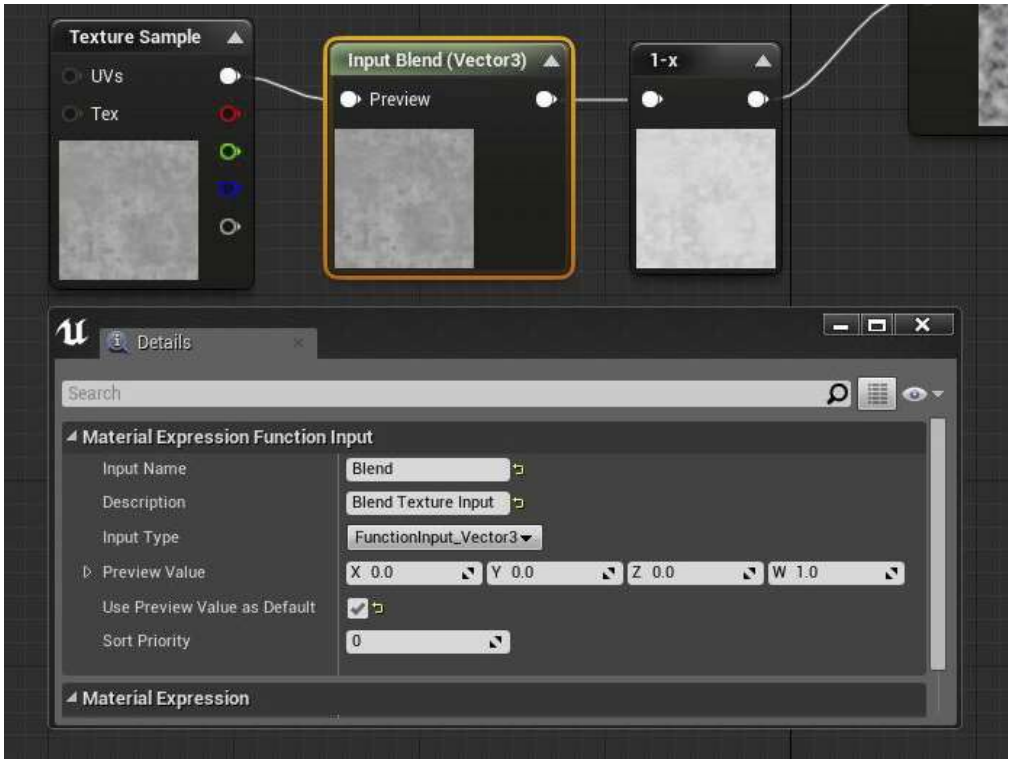

Рисунок 3.7 – «Preview» коннектор

У цьому прикладі, вираз Static Bool використовується щоб надати стандартне значення на вході static bool, який надається на рисунку 3.8.

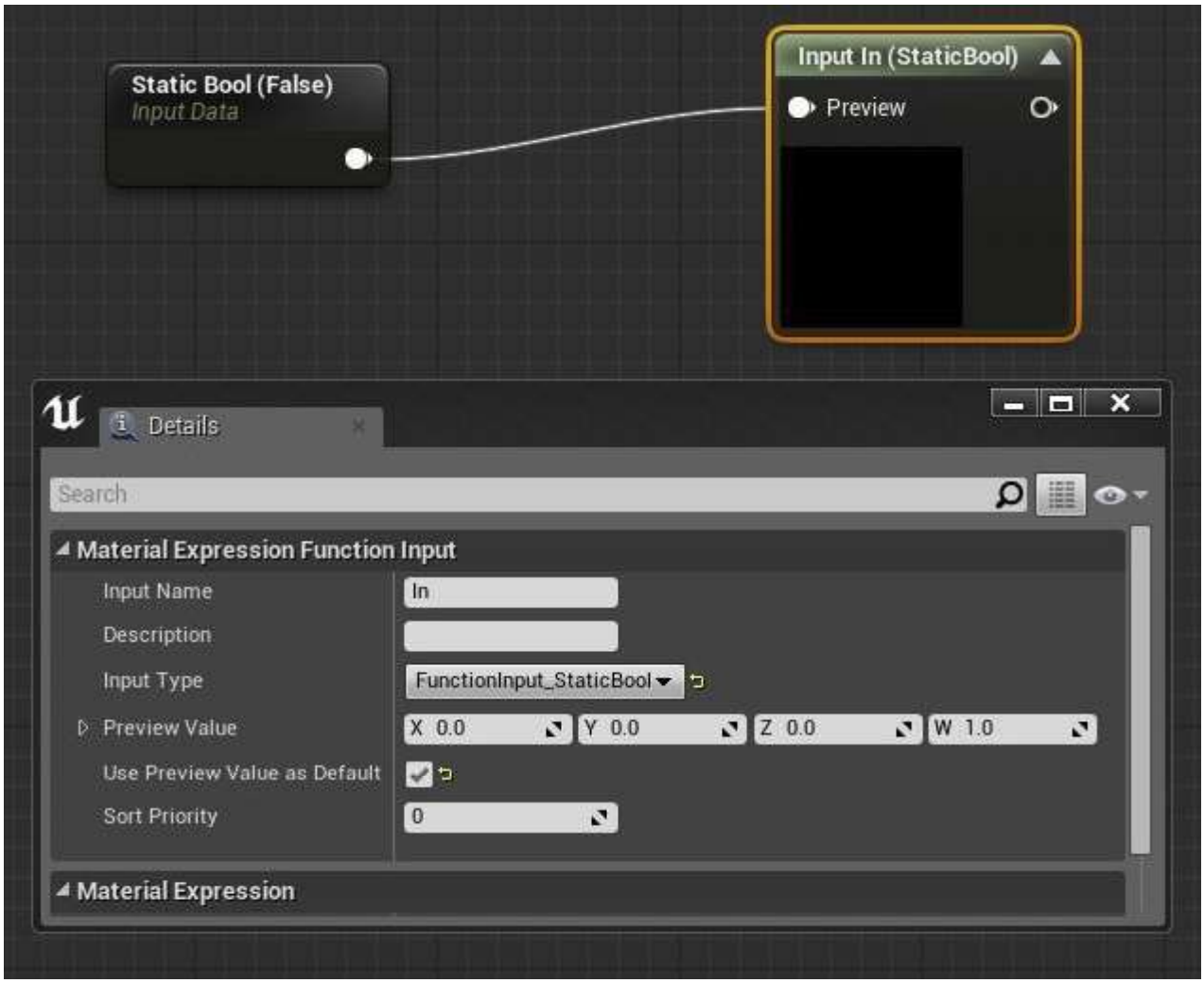

Рисунок 3.8 – Вираз Static Bool

Відзначимо, що вхід має опцію названу « Use Preview Value As Default «. Коли вона включена, значення предпросмотра буде використовуватися в будь-який час коли функцію використовується в матеріалі і нічого не підключено до цього входу, замість створення помилки компіляції. Це робить вхід опціональним, тому він забарвлений сірим.

Функції тепер можуть містити вирази типів параметрів. Ці параметри можуть бути пропущені безпосередньо до будь-якого матеріалу для використання[10].

Щоб використовувати параметр текстури з функцією, створіть texture input і з'єднайте його до texture object перевизначивши його в ноді texture sample, як показано на рисунку 3.9.

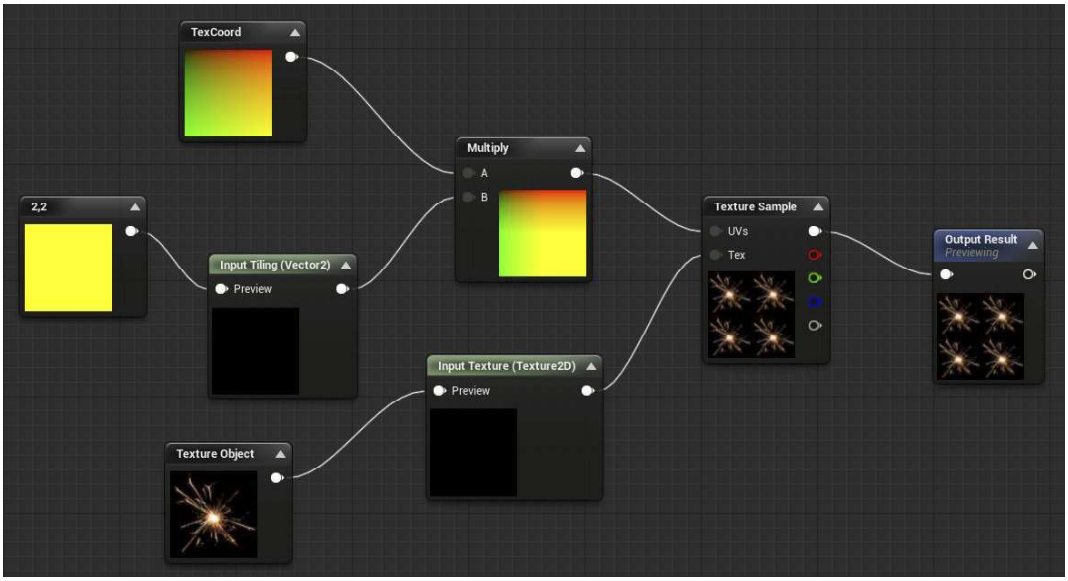

Рисунок 3.9 – Нода texture sample

Потім, в матеріалі який використовує функцію, розмістіть вираз TextureObjectParameter і з'єднайте його до входу в текстуру, як показано на рисунку 3.10

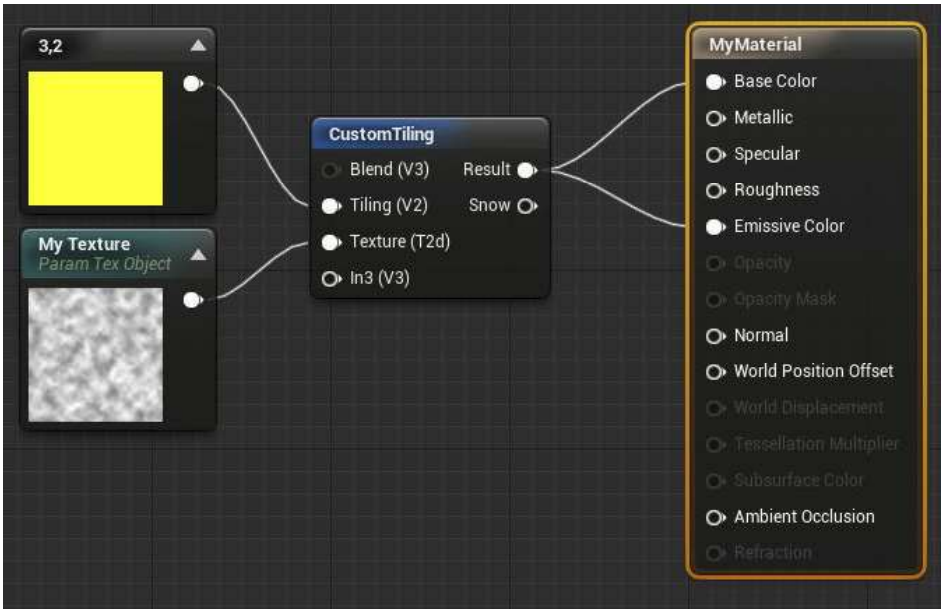

Рисунок 3.10 – Вираз TextureObjectParameter

Також для static switch параметрів, створіть Static Bool вхід і з'єднайте його з нодою StaticSwitch, як відображено на рисунку 3.11

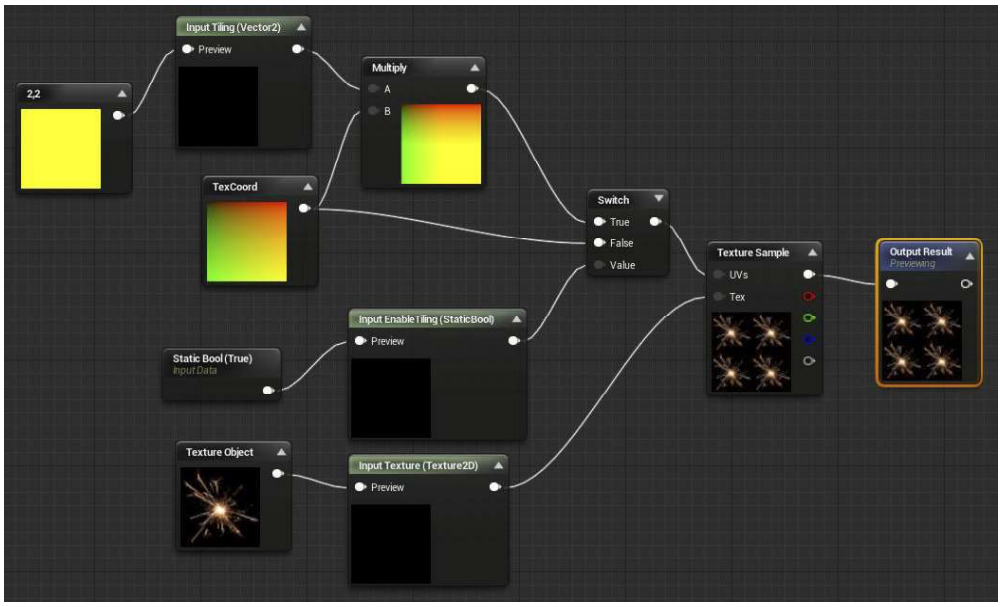

Рисунок 3.11 – Вираз Static Bool

Потім, в матеріалі який використовує функцію, розмістіть вираз StaticBoolParameter і з'єднайте його з входом static bool, як зображено на рисунку 3.12.

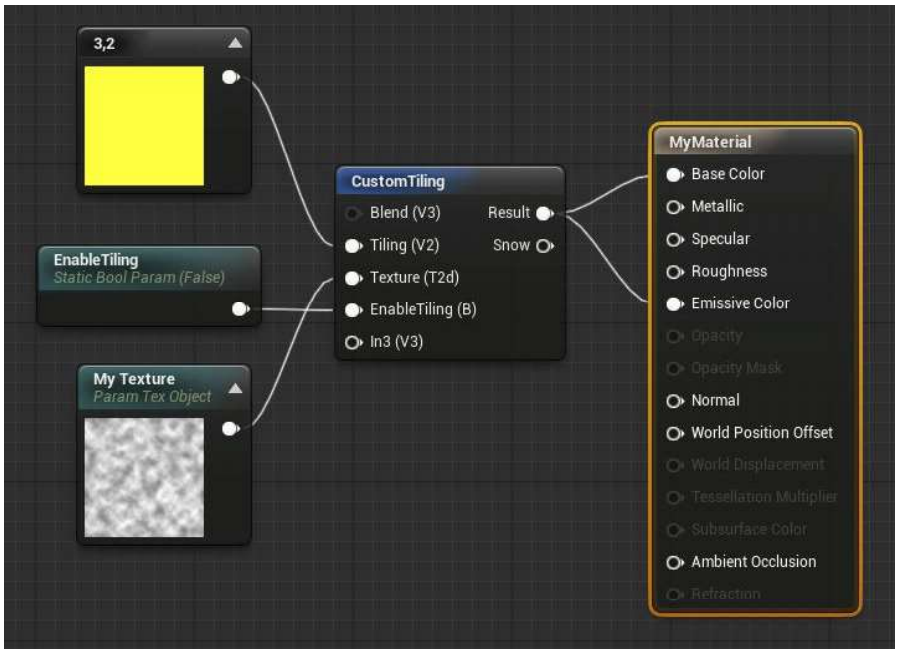

Рисунок 3.12 – Вираз StaticBoolParameter

Очікується що функції будуть розроблятися кількома людьми, але використовуватися багатьма, тому важливо мати хорошу документацію того, що функції виконують, які значення потрібні для їх входів і виходів. На цей випадок, функції мають кілька документаційних полів зверху назви функції і вхідних / вихідних імен:

Function Description - Натисніть на порожню область щоб побачити властивості функції, де розташоване Description . Якщо ви збираєтеся заповнити тільки одне поле опис, використовуйте це! Воно буде відображатися як підказка де Якби функція не з'явилася (Content Browser, material function library, function call node).

Input / Output Descriptions - Вони розташовуються на вхідних і вихідних виразах функції. Вони спливуть як підказки при наведенні миші на входи і виходи самовиразу функції.

Заповнена опис функції представлена на рисунку 3.13.

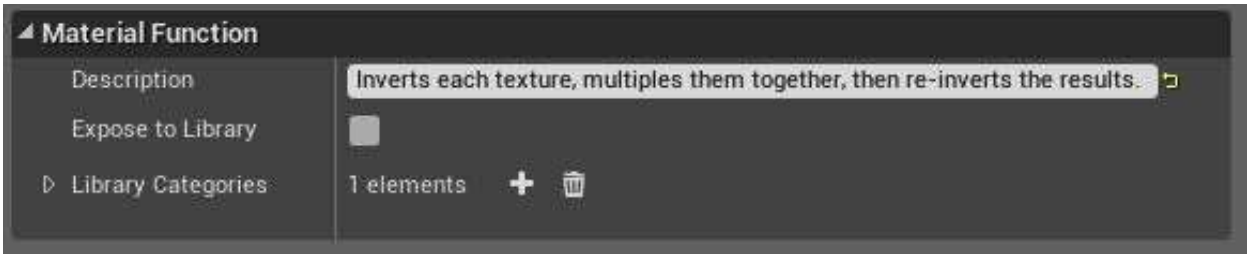

Рисунок 3.13 – Заповнена Function Description

Відповідна підказка при використанні наведена на рисунку 3.14.

Рисунок 3.14 – Підказка при використанні в матеріалі

Коли ви редагуєте функцію і натискаєте на « apply changes» , нова версія передаються на всі завантажені матеріали або функції які відсилаються до цієї функції. Будь-які не завантажені матеріали які посилаються на функцію будуть оновлені при наступному завантаженні.

Коли вхід або вихід видаляється з функції і ви застосовуєте зміни, будь-які зв'язки до віддалених з'єднань будуть розірвані! Важливо уникати цього, бо передачу можна скасувати. Чим більше матеріалів використовують функцію, тим більше буде вірогідність зламати всі, тому будьте обережні.

Всі завантажені матеріали, які використовують функцію будуть відзначені " брудними " коли зміна функції буде застосовано, яка може використовуватися і подивіться які пакети можуть бути дозволені щоб запобігти збільшенню часу завантаження. Ви можете знайти всі завантажені матеріалу які використовують функцію нажам правою кнопкою миші в контент браузері і вибравши цю опцію, як показано на рисунку 3.15.

Рисунок 3.15 – Пошук всих завантажених матеріалів

Функції можуть бути вкладені (функція в функцію) і пов'язані один з одним довільно, крім випадків коли вони створюють циклічну залежність.

Помилки компіляції функції будуть підсвічуватися в вираженні MaterialFunctionCall в червоному в матеріалах де воно використовується.

Повідомлення про помилку також скаже, яка функція викликає помилку, як показано на рисунку 3.16. У цьому прикладі, помилкою є що входи були підключені.

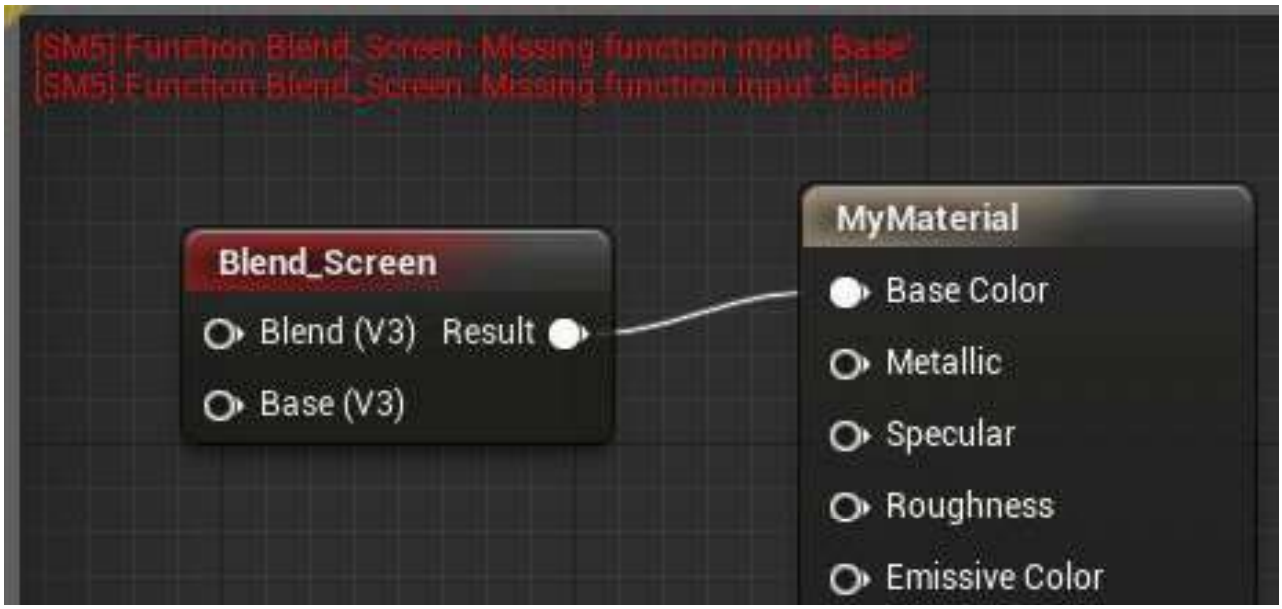

Рисунок 3.16 – Помилка компіляції функції

Редактор Material Instance можна відкрити натиснувши подвійним клацанням миші на будь-який зразок Material Instance або через контекстне меню матеріалу, що відкривається правим клацанням миші в контент браузері. Будь-який з цих способів відкриє редактор для конкретно обраного Material Instance. Інший спосіб відкрити редактор Material Instance, натисніть правою кнопкою миші на об'єкт і виберіть Edit Material Instance з меню, що випадає.

3.2 Дослідження алгоритму малювання за допомогою Render Targets в Unreal Engine

Почнемо з завантаження матеріалів для цього туторіал (взяти їх можна тут). Розпакуйте їх, перейдіть до CanvasPainterStarter і відкрийте

CanvasPainter.uproject . Якщо ви натиснете Play , то побачите наступне, що зображено на рисунку 3.17

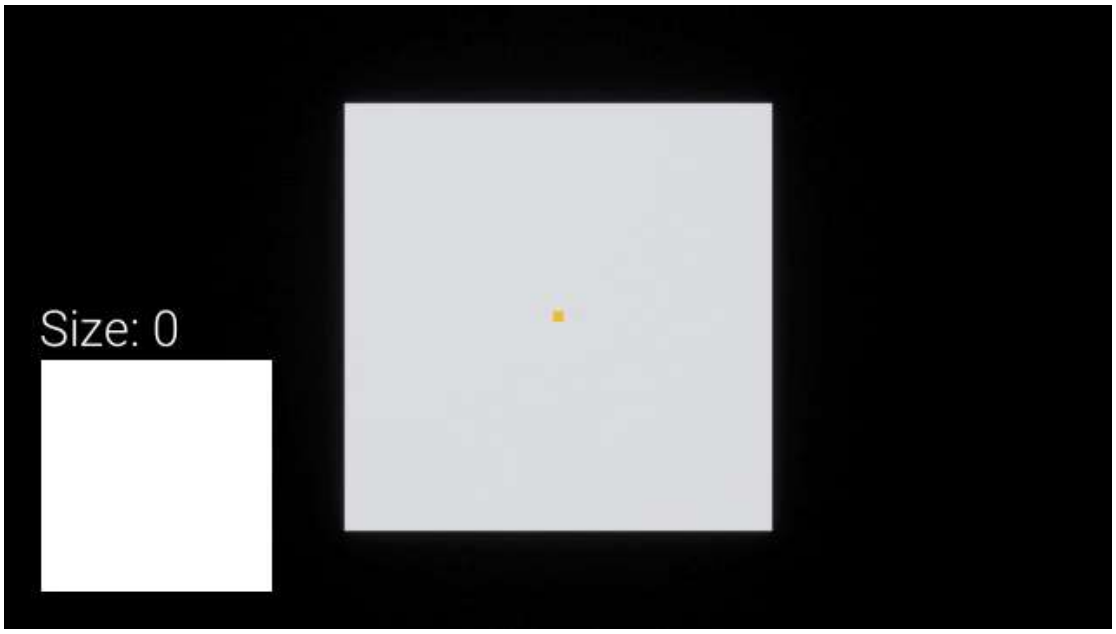

Рисунок 3.17 – CanvasPainterStarter

Квадрат посередині (canvas) - це те, на чому ми будемо малювати. Елементи UI зліва будуть текстурою, якої ми будемо малювати, і її розміром.

Для початку давайте розберемося зі способом, який використовується для малювання.

Перше, що нам потрібно - це render target, який використовується в якості полотна (canvas). Для визначення того, де виконувати малювання на render target, ми оттрассіруем пряму, що виходить з камери вперед. Якщо пряма перетинає полотно, то ми можемо отримати місце перетину в UV-просторі.

Наприклад, якщо полотно має ідеальну прив'язку UV-координат, то перетин в центрі поверне значення (0.5, 0.5) . Якщо пряма перетинає полотно в правому нижньому кутку, то ми отримаємо значення (1, 1). Потім можна використовувати прості обчислення для розрахунку місця малювання.

Але навіщо отримувати координати в UV-просторі? Чому б не використати координати реального простору світу? При використанні простору

світу нам спочатку доведеться обчислювати місце перетину щодо площині. Також доведеться враховувати поворот і масштаб площині.

При використанні UV-простору всі ці обчислення не потрібні. На площині з ідеальною прив'язкою UV-координат перетин з серединою завжди повертає (0.5, 0.5) , незалежно від розташування і повороту площині.

Спочатку ми створимо матеріал, який буде відображати render target.

Перейдіть в папку Materials і відкрийте M_Canvas .

Ми створимо render target динамічно, за допомогою блюпрінтов. Це означає, що нам доведеться налаштувати текстуру як параметр, щоб можна було передавати його render target. Для цього створимо TextureSampleParameter2D і назвемо його RenderTarget . Потім з'єднаємо його з BaseColor, дивись рисунок 3.18.

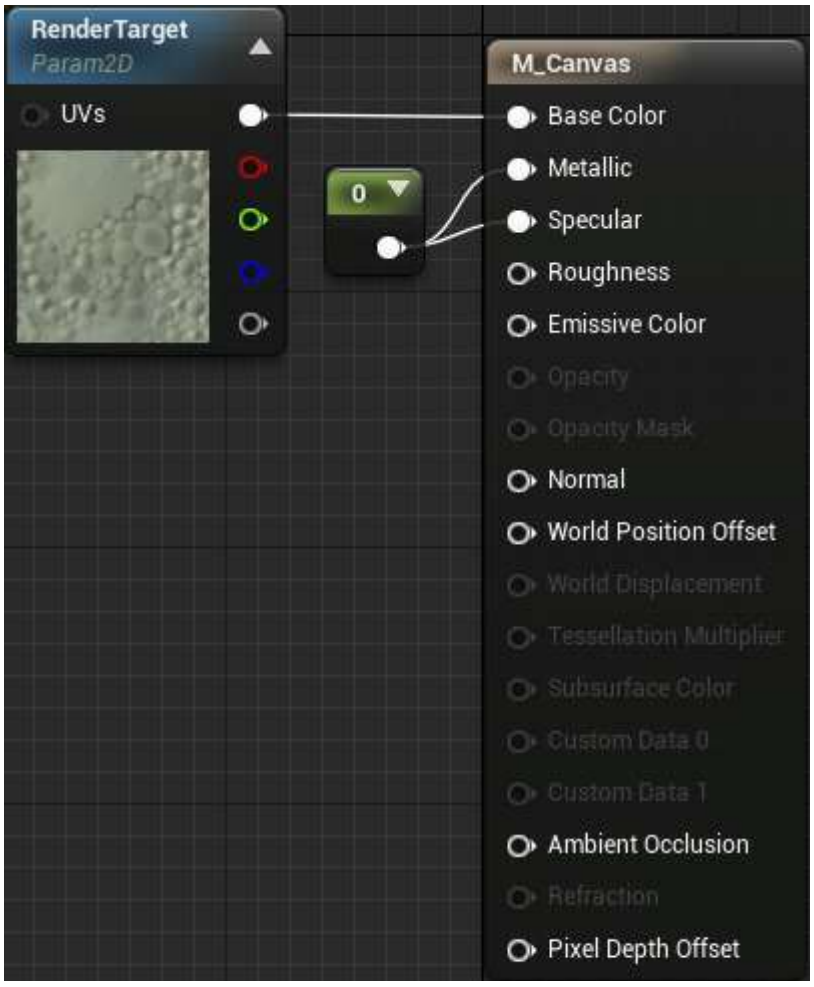

Рисунок 3.18 – TextureSampleParameter2D

Натисніть Apply , а потім закрийте M_Canvas .

Наступним етапом буде створення render target, після чого ми використовуємо його в якості матеріалу полотна.

Існує два способи створення render target. Перший: створення в редакторі натисканням на Add New \ Materials & Textures \ Render Target . Цей спосіб дозволяє зручно посилатися на один і той же render target кільком акторам. Однак якщо нам знадобляться кілька полотен, то доведеться створювати render target вручну для кожного полотна.

Тому краще створювати render target за допомогою блюпрінтов. Перевага такого підходу в тому, що ми створюємо render targets тільки при необхідності і вони не роздмухують обсяг файлів проекту.

Для початку нам потрібно створити render target і зберегти його як змінну для подальшого використання. Перейдіть в папку Blueprints і відкрийте BP_Canvas . Знайдіть Event BeginPlay і додайте виділені вузли, як показано на рисунку 3.19.

Рисунок 3.19 – Event BeginPlay

Дайте параметрам Width і Height значення 1024 . Так ми змінимо дозвіл render target на 1 024 × 1024 . Чим більше значення, тим вище якість зображення, але і більше витрати відеопам'яті.

Далі йде нод Clear Render Target 2D . Ми можемо використовувати цей нод для завдання кольору render target. Задайте Clear Color значення (0.07, 0.13, 0.06). При цьому весь render target заповниться зеленим кольором.

Тепер нам потрібно відобразити render target на меше полотна.

На даному етапі меш полотна використовує матеріал за замовчуванням. Для відображення render target потрібно створити динамічний екземпляр M_Canvas і передати йому render target[11]. Потім потрібно застосувати динамічний екземпляр матеріалу до Мішу полотна. Для цього ми додамо виділені Ноди, як надано на рисунку 3.20.

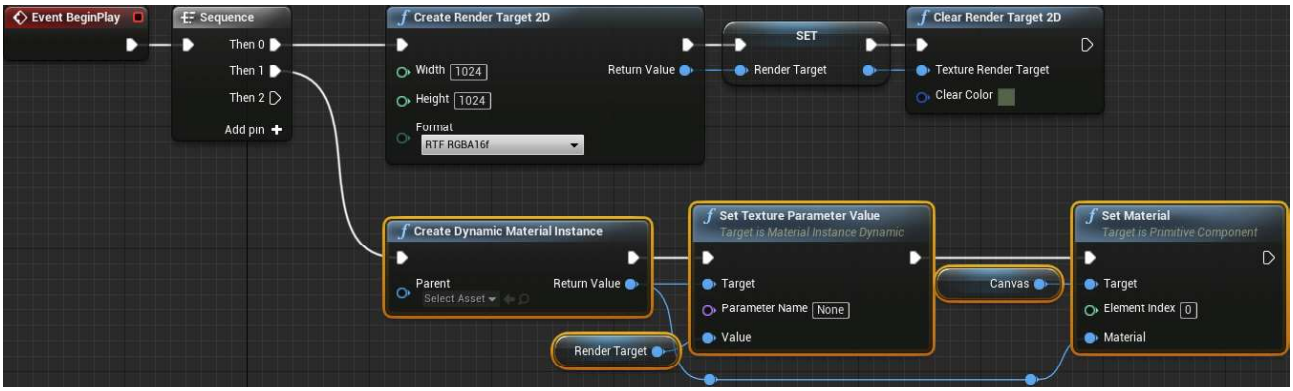

Рисунок 3.20 – Динамічний екземпляр M_Canvas

Спочатку перейдемо до ноду Create Dynamic Material Instance і поставимо в якості Parent значення M_Canvas . Так ми створимо динамічний екземпляр M_Canvas.

Далі перейдемо до ноду Set Texture Parameter Value і задамо для Parameter Name значення RenderTarget . Так ми передамо render target створеному раніше параметру текстури.

Тепер на меше полотна буде відображатися render target. Натисніть на Compile і поверніться в основний редактор. Натисніть на Play , щоб побачити, як полотно змінить колір.

Тепер, коли у нас є полотно, нам потрібно створити матеріал, який можна використовувати в якості кисті.

Перейдимо в папку Materials . Створіть матеріал M_Brush і відкрийте його. Спочатку задайте для Blend Mode значення Translucent . Це дозволить нам використовувати текстури з прозорістю.

Як і у випадку з матеріалом полотна, ми задаємо текстуру для кисті в блюпрінтах. Створіть TextureSampleParameter2D і назвіть його BrushTexture . З'єднайте його наступним чином, як зображено на рисунку 3.21.

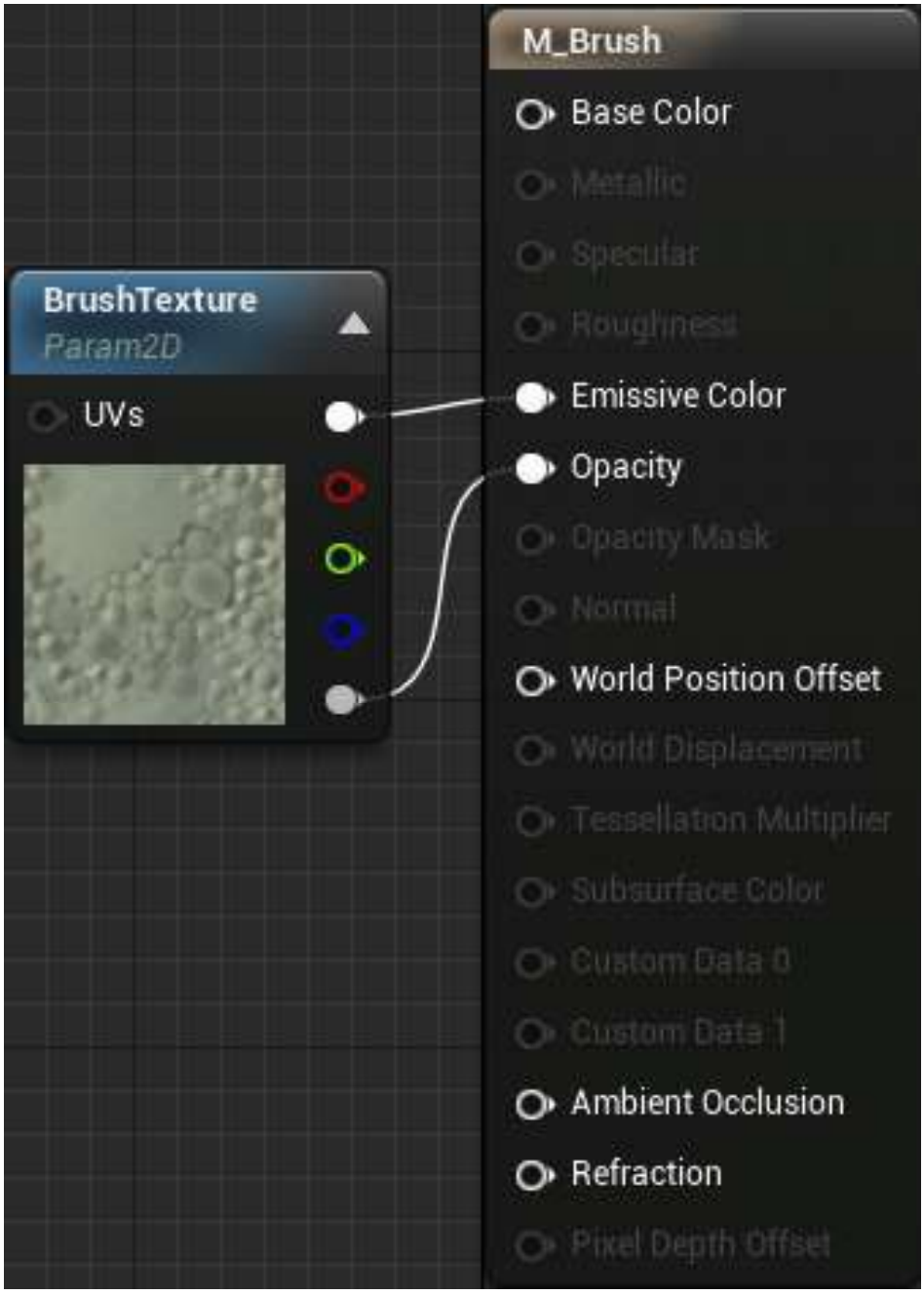

Рисунок 3.21 – BrushTexture

Наступне, що потрібно зробити - створити динамічний екземпляр матеріалу кисті, щоб можна було міняти текстуру кисті. Відкрийте BP_Canvas і додайте виділені вузли, як показано на рисунку 3.22.

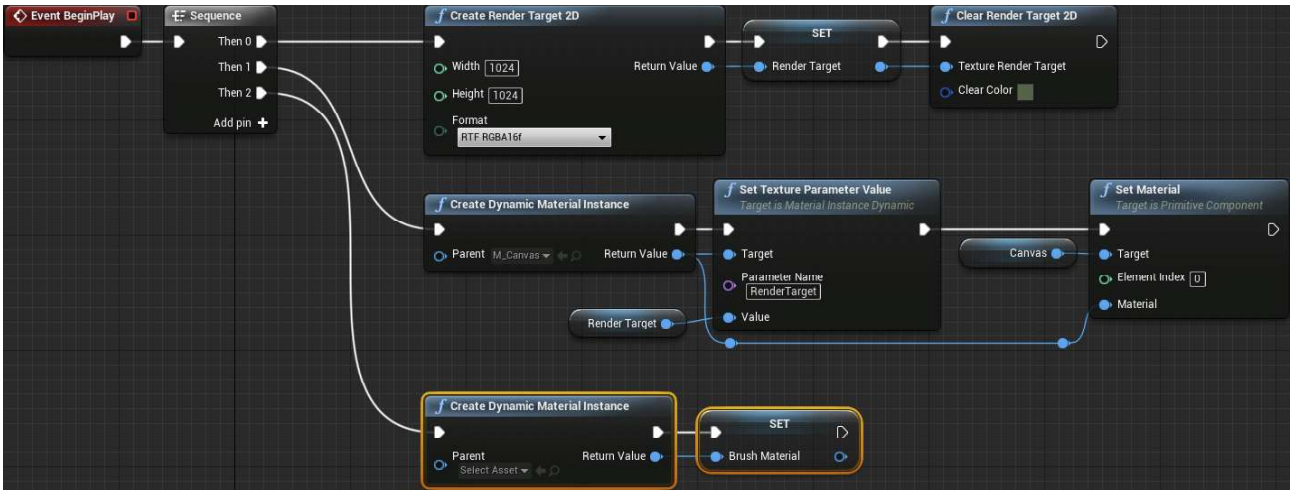

Рисунок 3.22 – Блюпринт BP_Canvas

Далі перейдіть в нод Create Dynamic Material Instance і задайте для Parent значення M_Canvas.

Ми створили матеріал кисті, і тепер нам потрібна функція для малювання пензлем на render target.

Створіть нову функцію і назвіть її DrawBrush . Спочатку нам знадобляться параметри: використовувана текстура, розмір кисті і місце для малювання. Створіть наступні вхідні вхідні дані:

– BrushTexture: виберіть тип Texture 2D;
– BrushSize: виберіть тип float;
– DrawLocation: виберіть тип Vector 2D.

Перш ніж малювати кисть, нам потрібно задати її текстуру. Для цього створимо схему, показану нижче. Переконайтеся, що в якості Parameter Name вибрано значення BrushTexture, як показано на рисунку 3.23.

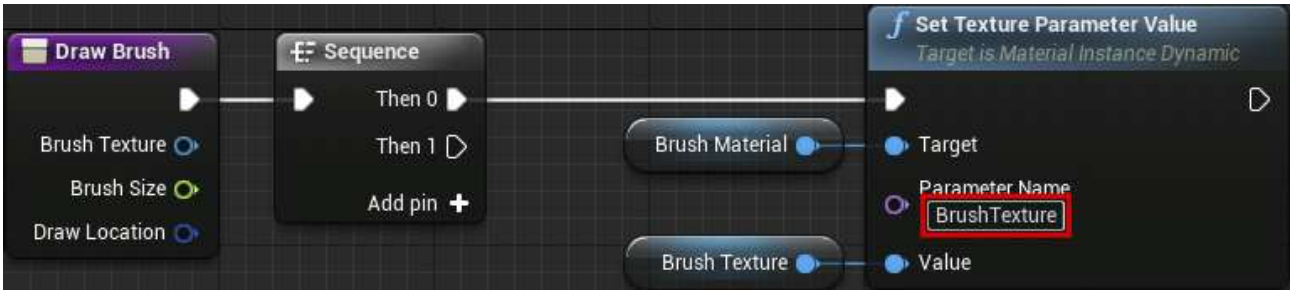

Рисунок 3.23 – В якості Parameter Name вибрано BrushTexture

Тепер нам потрібно виконувати отрисовку в render target. Для цього створимо виділені Ноди, як зображено на рисунку 3.24.

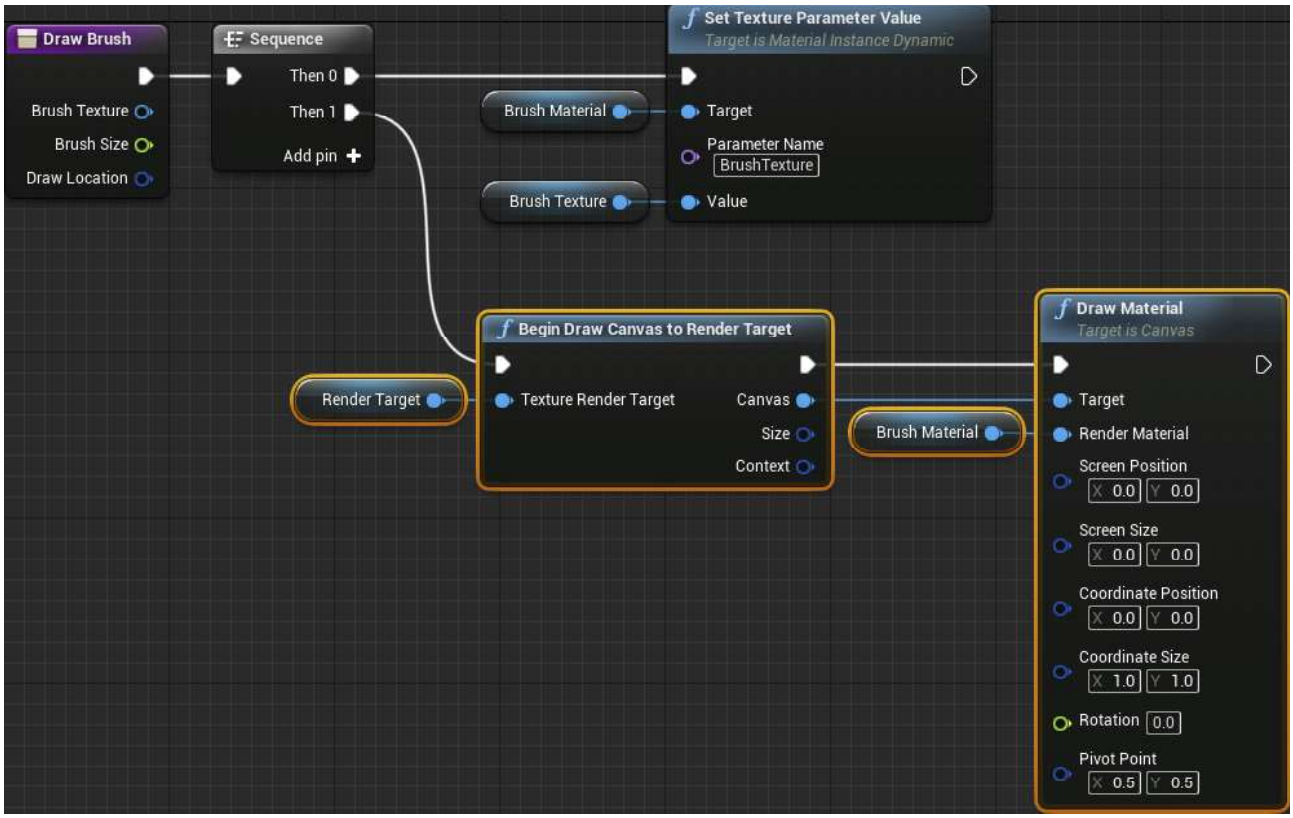

Рисунок 3.24 – Нода отрисовки в render target

Begin Draw Canvas to Render Target дозволить движку дізнатися, що ми хочемо почати отрисовку в певний render target. Draw Material дозволить малювати матеріал за вказаною розташування сумісні з вибраним і поворотом.

Обчислення позиції відтворення - це двоетапний процес. Спочатку нам потрібно отмасштабовані DrawLocation , щоб умістится в дозвіл render target. Для цього помножимо DrawLocation на Size.

За замовчуванням движок буде малювати матеріали, використовуючи в якості вихідної точки верхній лівий кут. Тому текстура кисті не буде центрована нам, де ми хочемо виконати отрисовку. Щоб виправити це, нам потрібно поділити BrushSize на 2 , а потім відняти результат з попереднього етапу.

Нарешті нам потрібно повідомити движку, що ми хочемо зупинити отрисовку в render target. Додамо нод End Draw Canvas to Render Target і з'єднаємо його наступним чином, як зображено на рисунку 3.25.

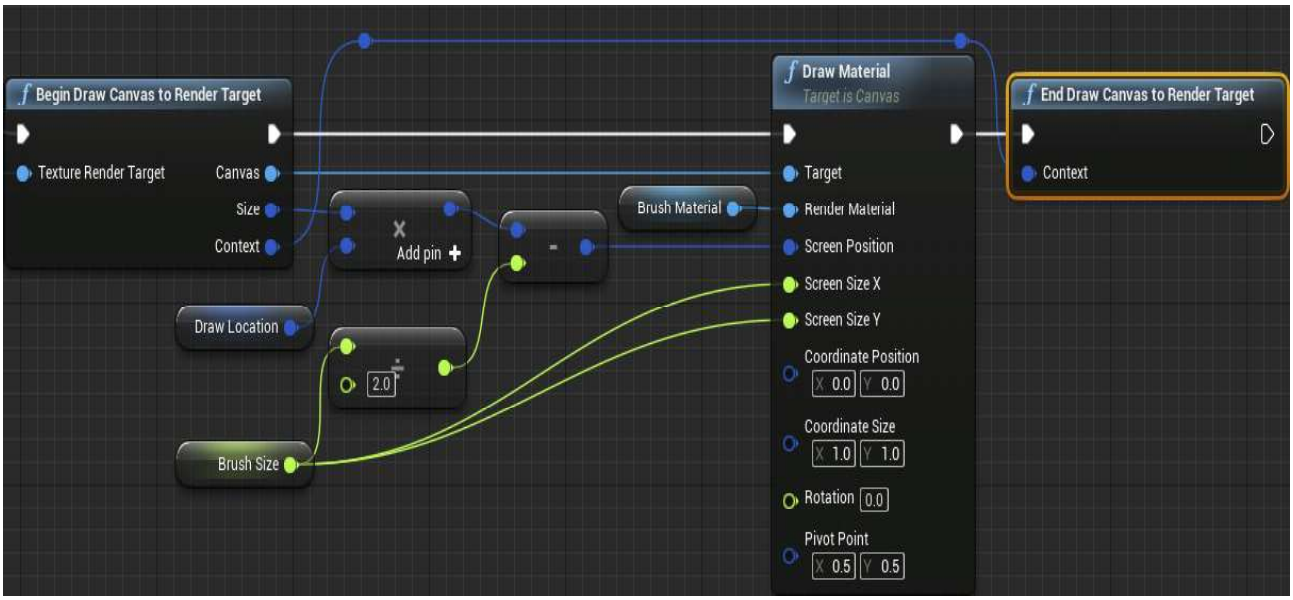

Рисунок 3.25 – Нода Render Target

Тепер при кожному виконанні DrawBrush вона буде спочатку встановлювати в якості текстури для BrushMaterial передану текстуру. Потім вона буде малювати BrushMaterial в RenderTarget , користуючись переданими позицією і розміром.

І на цьому функція відтворення готова. Натисніть на Compile і закрийте BP_Canvas . Наступним кроком буде трасування прямий з камери і малювання в тому місці полотна, де відбувся перетин.

Перш ніж малювати на полотні, нам потрібно вказати текстуру кисті і розмір. Перейдемо в папку Blueprints і відкриємо BP_Player . Потім дамо змінної BrushTexture значення T_Brush_01 , а змінної BrushSize значення 500 . Так ми призначимо кисті зображення мавпи розміром 500 × 500 пікселів.

Далі необхідно виконати трасування прямий. Знайдіть InputAxis Paint і створіть наступну схему, як показано на рисунку 3.26.

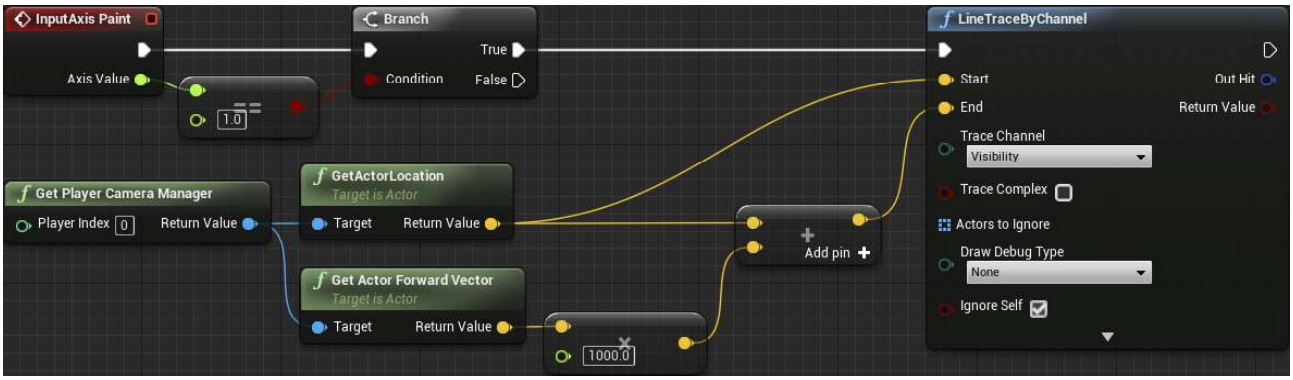

Рисунок 3.26 – Схема InputAxis Paint

Так ми будемо виконувати трасування прямий, спрямованої з камери прямо, поки гравець буде утримувати клавішу, призначену для Paint (в нашому випадку це ліва клавіша миші).

Тепер нам необхідно перевірити, перетнула чи пряма полотно. Додамо виділені Ноди, зображені на рисунку 3.27.

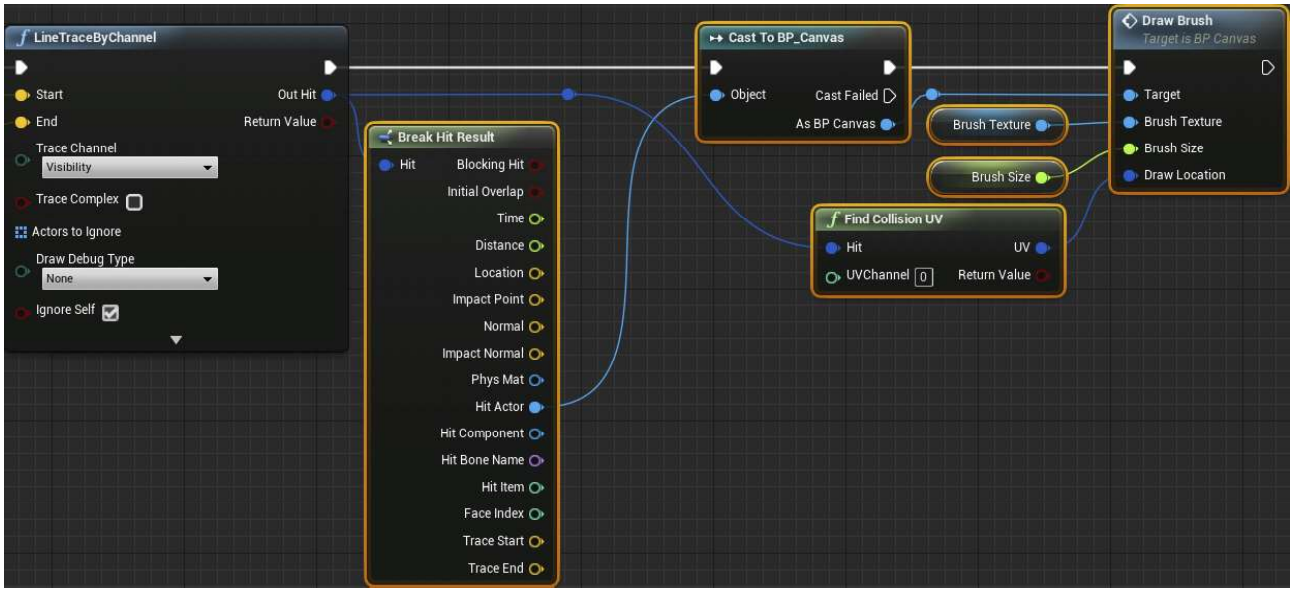

Рисунок 3.27 – Схема трасування прямий, спрямованої з камери

Тепер при перетині прямий і полотна буде виконуватися функція DrawBrush , що використовує передані їй змінні кисті і UV-координати.

Щоб нод Find Collision UV заробив, нам потрібно змінити два параметри. По-перше, перейдемо в нод LineTraceByChannel і включимо Trace Complex .

По-друге, перейдемо в Edit \ Project Settings , а потім в Engine \ Physics . Включимо Support UV From Hit Results і перезапустити проект.

Після перезапуску для малювання на полотні натисніть Play і ліву кнопку миші.

Можна навіть створити кілька полотен і малювати на кожному з них окремо. Це можливо, тому що кожен полотно динамічно створює власний render target.

Зараз реалізуємо функціонал зміни гравцем розміру кисті.

Відкрийте BP_Player і знайдіть нод InputAxis ChangeBrushSize. Ця прив'язка осі налаштована на використання колеса миші . Для зміни розміру кисті нам досить міняти значення BrushSize в залежності від Axis Value . Для цього створимо таку схему, що зображено на рисунку 3.28.

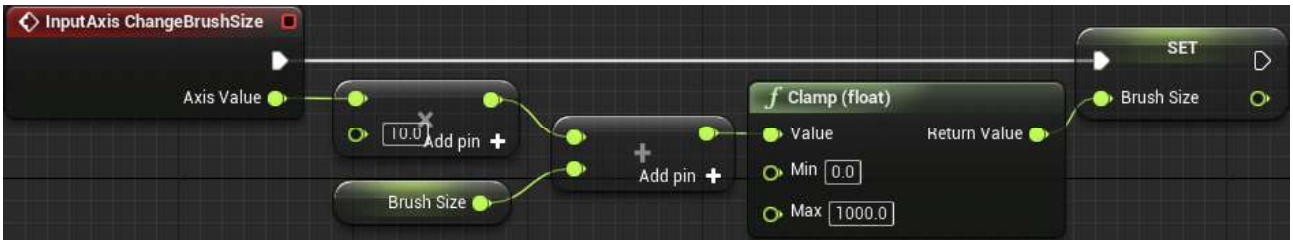

Рисунок 3.28 – Схема InputAxis ChangeBrushSize

Вона буде виконувати додаток або віднімання з BrushSize при використанні гравцем колеса миші. Перше множення визначає швидкість додавання або віднімання. В якості міри безпеки додано Clamp (float) . Він гарантує, що розмір кисті не стане менше 0 або більше 1000 .

Натисніть на Compile і поверніться в основний редактор. Покрутіть колесо миші , щоб змінити розмір кисті при малюванні.

В останньому створимо функціонал, що дозволяє гравцеві міняти текстуру кисті.

Для початку нам потрібен масив для зберігання текстур, які може використовувати гравець. Відкрийте BP_Player і створіть змінну array . Виберіть для неї тип Texture 2D і назвіть її Textures.

Потім створіть три елемента в Textures . Дайте їм у такому значенні:

– T_Brush_01;

– T_Brush_02;

– T_Brush_03.

Це будуть текстури, якими зможе малювати гравець. Для додавання нових текстур досить додати їх в цей масив.

Далі нам необхідна змінна для зберігання поточного індексу масиву. Створіть змінну integer і назвіть її CurrentTextureIndex.

Далі нам необхідний спосіб обходу в циклі всіх текстур. Налаштуємо прив'язку дії (action mapping) під назвою NextTexture і прив'яжемо її до правої клавіші миші . Коли гравець натискає цю кнопку, повинен виконуватися перехід до наступної текстурі. Для цього знайдіть нод InputAction NextTexture і створіть наступну схему, як зображено на рисунку 3.29

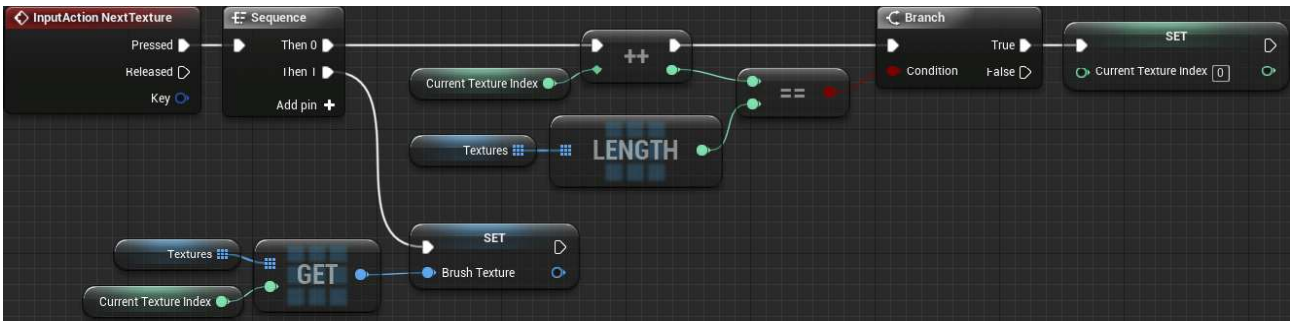

Рисунок 3.29 – Схема action mapping під назвою NextTexture

Ця схема буде збільшувати CurrentTextureIndex при кожному натисканні на праву кнопку миші . Якщо індекс досягає кінця масиву, то він знову скидається на 0 . Нарешті, BrushTexture задається відповідна текстура.

Натисніть на Compile і закрийте BP_Player . Натисніть Play і клацайте правою клавішею миші для перемикання між текстурами.

Render target - це надзвичайно потужний інструмент.

3.3 Алгоритм створення інтерактивної трави в Unreal Engine

До недавнього часу трава в іграх зазвичай позначалася текстурою на землі, а не рендерингом окремих стебел. Але зі збільшенням потужності заліза з'явилася можливість рендерить траву. Чудові приклади такого рендеринга можна побачити в іграх на зразок Horizon Zero Dawn і The Legend of Zelda: Breath of the Wild . У цих іграх гравець може бродити по трав'яним лугах, і, що більш важливо, трава реагує на дії гравця[12].

Для початку давайте розглянемо інший спосіб створення інтерактивної трави. Найбільш поширений спосіб полягає в передачі координат гравця матеріалу трави і в використанні сферичної маски для відгинання трави в певному радіусі від гравця.

Хоча цей підхід цілком хороший, він погано масштабується, коли ми хочемо додати більше впливають на траву акторів. Для кожного додається актора до матеріалу доведеться додавати ще один параметр координат і сферичну маску. Більш масштабований метод полягає в використанні векторного поля .

Векторне поле - це просто текстура, кожен піксель якого відповідає напрямку. Якщо ви раніше працювали з картами потоків, то вони аналогічні. Але замість переміщення UV ми будемо за допомогою контакту World Position Offset переміщати вершини. На відміну від рішення зі сферичною маскою, для отримання направлення згинання матеріалами досить тільки один раз семпліровалісь векторне поле.

Давайте дізнаємося, як можна зберігати напрямки в текстуру. Подивіться на цю сітку, що зображено на рисунку 3.30.

Припустимо, червона точка - це об'єкт, який ми хочемо перемістити. Якщо ми перемістимо його в правий нижній кут, то який вектор буде позначати це переміщення? Якщо ви відповіли (1, 1) , то ви маєте рацію! Як ви, напевно,

знаєте, можна також представити вектори у вигляді квітів, і таким чином зберегти їх в текстуру.

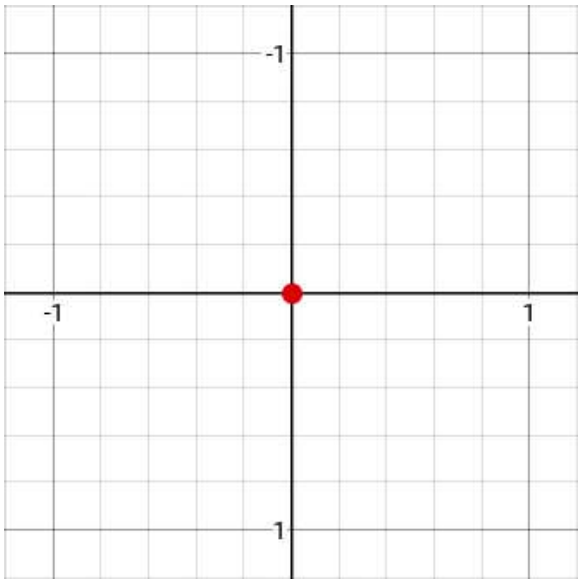

Рисунок 3.30 – Векторна сітка

Давайте вставимо цей вектор в color picker Unreal і подивимося, який колір він поверне, який зображен на рисунку 3.31.

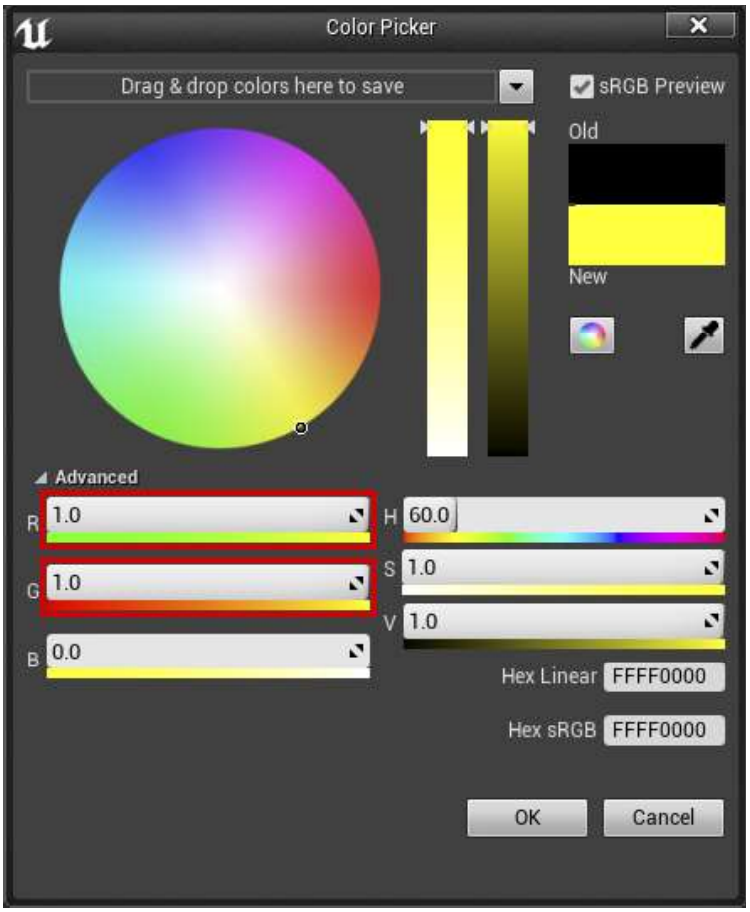

Рисунок 3.31 – Color picker Unreal

Як бачите, напрямок (1, 1) повертає жовтий колір. Це означає, що якщо ми захочемо зігнути траву у напрямку позитивних осей XY, то нам доведеться використовувати цей колір текстури. Давайте тепер подивимося на кольори всіх векторів, що показано на рисунку 3.32.

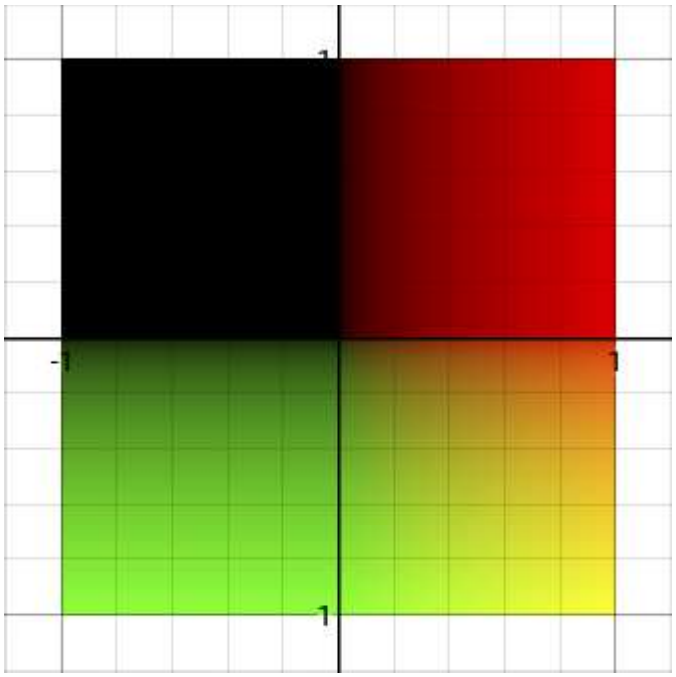

Рисунок 3.32 – Кольори всіх векторів

Правий нижній квадрат виглядає досить добре, тому що має градієнти по обох осях. Це означає, що ми можемо зберігати в цьому квадраті у вигляді кольору будь-який вектор, тому що кожен вектор має унікальний колір.

Але з іншими трьома квадрантами виникає проблема. У нас є градієнт тільки по одній осі, чи ні градієнта взагалі. Це означає, що кілька векторів матимуть один колір. Наприклад, ми ніяк не зможемо розрізнити вектори (-1, 1) і (0, 1) .

Ці три квадранта не мають унікальних квітів для кожного вектора тому, що ми можемо уявити кольору тільки з допомогою значень від 0 до 1. проте в цих трьох квадрантах використовуються негативні значення, що знаходяться поза цим інтервалом.

Рішення полягає в перерозподілі векторів таким чином, щоб всі вони помістилися в інтервалі від 0 до 1. Це можна зробити, помноживши вектор на 0.5 і додавши 0.5 .

Тепер у кожного вектора є унікальний колір. Коли нам потрібно використовувати його для обчислень, ми просто перерозподіляємо його назад в

інтервал від -1 до 1. Ось кілька кольорів, і відповідні їм напрями після перерозподілу:

- (0, 0): негативні X і Y;
- (0.5, 0.5): немає руху;
- (0, 1): негативний X і позитивний Y;
- (1, 0): позитивний X і негативний Y.

Ми будемо малювати на render target за допомогою «кистей». Це будуть просто зображення обраного векторного поля. Я буду називати їх кистями напрямків.

Замість малювання на render target за допомогою блюпрінтов, ми можемо використовувати частки. Частинки будуть відображати кисть напрямків і випускати з гравця. Для створення векторного поля нам досить використовувати захоплення сцени (scene capture) і виконувати захоплення тільки частинок. Перевага цього методу в тому, що створювати сліди дуже просто. Крім того, він дозволяє з легкістю управляти такими властивостями, як тривалість збереження слідів і їх розмір. Крім того, частинки створюють тимчасово зберігаються сліди, тому що вони існують після відходу з області захоплення і повернення в неї.

Існує два способи створення кисті напрямків, що наведені нижче.

Математичний: напрямки і форма задаються всередині матеріалу. Перевага його в тому, що він не вимагає стороннього ПО і зручний для простих форм.

Перетворення карти нормалей: створення карти нормалей потрібних напрямків і форми. Для перетворення карти в підходяще векторне поле нам досить видалити синій канал. Перевага цього методу в тому, що можна дуже просто створювати складні форми. Нижче представлений приклад кисті, яку складно буде створити математично.

Перейдіть в папку Materials і відкрийте M_Direction . Зауважте, що для цього матеріалу обрана модель затінення (shading model) Unlit . Це важливо,

тому що дозволяє захоплення сцени захоплювати частинки без впливає на них освітлення.

Щоб не ускладнювати, ми створимо матеріал, який змусить траву відсуватися від центру частинки. Для цього створимо таку схему, як та, що зображена на рисунку 3.33.

Рисунок 3.33 – Схема матеріалу зсуву трави

Тепер нам потрібно виконати перерозподіл. Для цього додайте виділені Ноди і з'єднайте все наступним чином, як зображено на рисунку 3.34.

Рисунок 3.34 – Схема з’єднання нодів

Додамо кисті круглу форму. Для цього додамо виділені Ноди, як показано на рисунку 3.35.

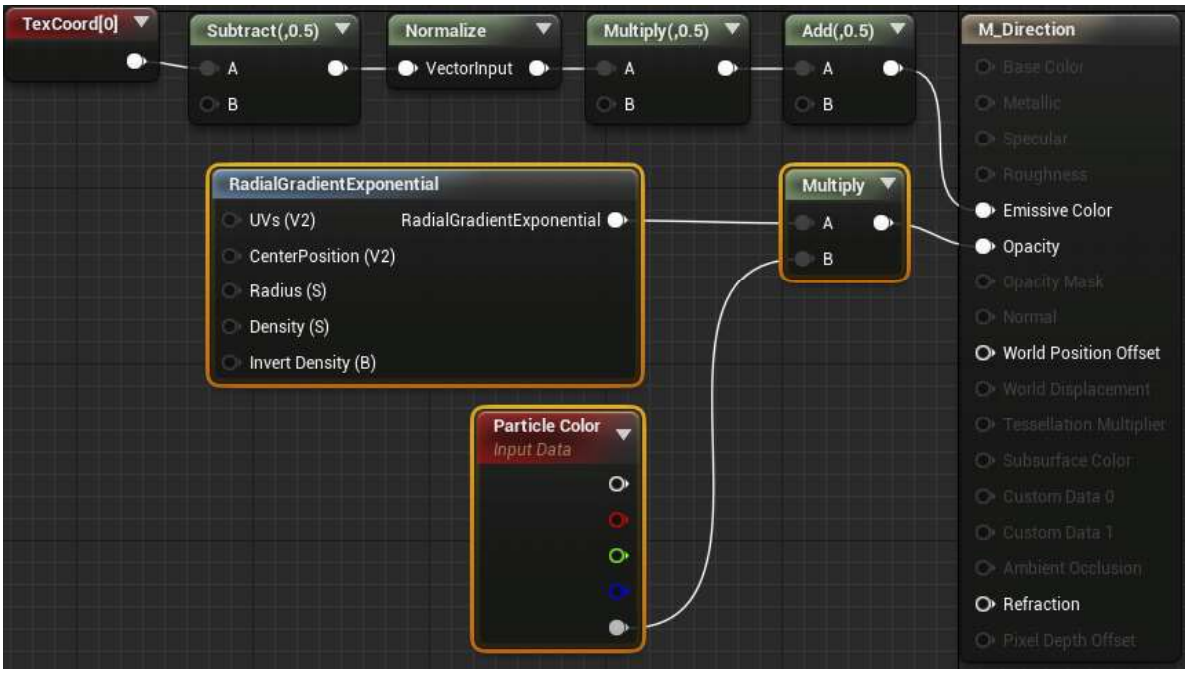

Рисунок 3.35 – Схема кисті круглої форми

RadialGradientExponential управляє розміром і різкістю окружності кола. Множення його на Particle Color дозволяє управляти непрозорістю частинок з системи частинок. Детальніше я розповім про це в наступному розділі.

Кисть виглядає, як показано на рисунку 3.36.

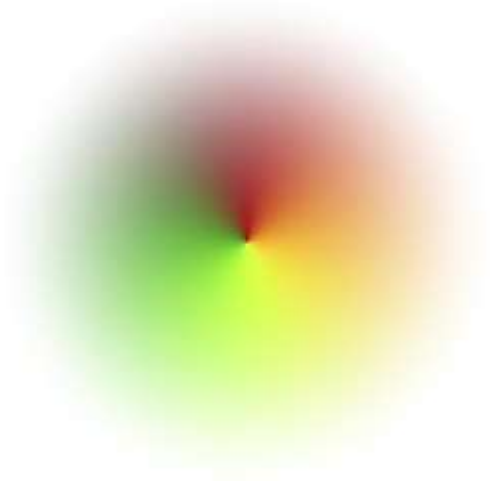

Рисунок 3.36 – Кисть

Натисніть Apply і закрийте матеріал. Тепер, коли ми створили матеріал, настав час приступити до системи частинок слідів.

Перейдіть в папку ParticleSystems і відкрийте PS_GrassTrail . Для економії часу я вже створив всі необхідні модулі.

Ось, як кожен модуль впливає на сліди на траві:

- Spawn це частота створення впливає на плавність слідів. Якщо сліди виглядають переривчастими, то варто збільшити частоту створення;
- Lifetime це час існування сліду до повернення трави до вихідного стану;
- Initial Size це розмір сліду;
- Color Over Life це оскільки ми використовуємо в матеріалі Particle Color, тут можна управляти непрозорістю. Також можна змінювати альфа-криву для управління зникненням сліду. Наприклад, можна вибрати лінійне пропажа, easing in і / або easing out. У цьому туторіали ми залишимо настройку за умовчанням, тобто лінійне пропажа;
- Lock Axis це використовується для того, щоб частинки були спрямовані в бік захоплення сцени;
- Initial Rotation це використовується для того, щоб частинки були орієнтовані по правильним осях.

Спочатку нам потрібно задати матеріал. Виберіть модуль Required і задайте для Material значення M_Direction . Також задайте для Sort Mode значення PSORTMODE Age Newest First .

Цей режим сортування дозволяє рендерить нові частинки поверх старих. Якщо цього не зробити, то на траву будуть впливати не нові, а старі частинки.

Далі йде тривалість існування сліду. Виберіть модуль Lifetime і задайте Constant значення 5 . Завдяки цьому слід буде пропадати протягом п'яти секунд.

Тепер перейдемо до розміру сліду. Виберіть модуль Initial Size і задайте для Constant значення (150, 150, 0) . Завдяки цьому кожна частка буде покривати область 150 × 150.

Тепер нам потрібно зробити так, щоб дивилися в напрямку захоплення сцени. Оскільки захоплення сцени виконуватися з виду зверху, то частинки

повинні дивитися в напрямку позитивної осі Z. Для цього виберіть модуль Lock Axis і задайте для Lock Axis Flags значення Z .

Нарешті, нам потрібно задати поворот частинок. На поточний момент кольору кисті не вирівняні з напрямом, який вони представляють. Так вийшло тому, що за замовчуванням система частинок застосовується з поворотом в 90 градусів. Щоб виправити це, виберіть модуль Initial Rotation і задайте для Constant значення -0.25 . Це поверне частки на 90 градусів проти годинникової стрілки[13].

І це все, що нам потрібно для системи частинок, тому давайте закриємо її.

Далі нам потрібно прикріпити систему частинок до того, що повинно створювати сліди. У нашому випадку потрібно прикріпити її до персонажу гравця.

Перейдіть в Characters \ Mannequin і відкрийте BP_Mannequin . Далі створіть компонент Partice System і назвіть його GrassParticles.

Далі нам потрібно задати систему частинок. Перейдіть в панель Details і задайте для Particles \ Template значення PS_GrassTrail .

Було б дивно, якби гравець міг бачити слід в грі, тому варто приховати його. Для цього включимо Rendering \ Owner No See.

Оскільки система частинок прикріплена до гравця (owner), то гравець не побачить її, але вона буде видима всьому іншому.

Натисніть Compile , а потім натисніть Play . Зауважте, що частки не з'являються для камери гравця, але відображаються на render target.

Поки захоплення сцени налаштований на захоплення всього . Очевидно, що це нам не підходить, тому що на траву впливають тільки частинки. У наступному розділі ми дізнаємося, як виконувати захоплення тільки частинок.

Якщо ми будемо захоплювати частинки зараз, то буде виконуватися непотрібне нам згинання в областях без частинок. Так відбувається тому, що фоновим кольором render target є чорний. Згинання відбувається тому, що чорний позначає рух у напрямку до негативних осях XY (після перерозподілу). Щоб порожні області не містили руху, нам потрібно зробити так, щоб фоновим

кольором render target був (0.5, 0.5, 0) . Найпростіше це зробити, створивши величезну площину і прикріпивши його до гравця.

Спочатку створимо матеріал для фону. Поверніться в Content Browser і відкрийте Materials \ M_Background . Потім з'єднайте константу (0.5, 0.5, 0) з Emissive Color.

Натисніть Apply і закрийте матеріал. Поверніться до BP_Mannequin , а потім створіть новий компонент Plane . Назвіть його Background.

Далі задайте наступні властивості:

- Location (0, 0, -5000). Ми розміщуємо площину так низько, щоб вона не перекривала ніякі частинки;
- Scale (100, 100, 1). Так ми отмасштабіруем до розміру, достатнього для покриття всієї області захоплення;
- Material M_Background.

Як і у випадку з частинками, було б дивно, якби гравець бачив під собою величезну жовту площину. Щоб приховати її, включіть Rendering \ Owner No See.

Тепер, коли ми налаштували фон, настав час для захоплення частинок. Ми можемо зробити це, додавши систему частинок до списку show-only list захоплення сцени. Це список компонентів, які буде захоплювати захоплення сцени.

Перш ніж ми отримаємо можливість додавати в show-only list, нам необхідний спосіб отримання всіх впливають на траву акторів. Один із способів їх отримання - використання тегів . Теги - це прості рядки, які можна привласнювати акторам і компонентів. Потім можна використовувати нод Get All Actors With Tag для отримання всіх акторів з відповідним тегом.

Оскільки актор гравця повинен впливати на траву, йому потрібен тег. Для додавання тега натисніть на кнопку Class Defaults . Потім створіть в Actor \ Tags новий тег і назвіть його GrassAffector.

Оскільки в список show-only list можна передавати тільки компоненти , нам потрібно додати теги і до впливає на траву компонентів. Виберіть компонент GrassParticles і додайте новий тег, розташований в розділі Tags . Назвіть його теж GrassAffector (необов'язково використовувати саме цей тег). Повторіть те ж саме для компонента Background.

Тепер нам потрібно додати впливають на траву компоненти в show-only list захоплення сцени. Натисніть Compile і закрийте BP_Mannequin . Потім відкрийте Blueprints \ BP_Capture . Перейдіть до Event BeginPlay і додайте виділені Ноди.

Після чого вона буде перевіряти, чи є у актора компоненти з таким тегом, і додавати їх в show-only list.

Далі нам потрібно сказати захоплення сцени, щоб він використовував тільки show-only list. Виберіть компонент SceneCapture і перейдіть в розділ Scene Capture . Задайте для Primitive Render Mode значення Use ShowOnly List .

Натисніть Compile і закрийте блюпрінт. Якщо натиснути на Play , то ви побачите, що render target тепер виконує захоплення тільки частинок і площини фону.

Спочатку нам потрібно спроектувати render target на траву. Перейдіть в папку Materials і відкрийте M_Grass . Потім створіть показані нижче Ноди. Задайте в якості текстури RT_Capture.

Оскільки ми перерозподілили кольору в інтервал від 0 до 1, то перед використанням їх потрібно перерозподілити назад в інтервал від -1 до 1.

Тепер, коли у нас є напрямок згинання, нам потрібен якийсь спосіб повернути траву в цьому напрямку. На щастя, для цього існує нод під назвою RotateAboutAxis . Давайте створимо його.

Давайте почнемо з контакту NormalizedRotationAxis . Як зрозуміло з назви, це вісь, навколо якої буде повертатися вершина. Для обчислення нам всього лише потрібно векторний добуток напрямки згинання на (0, 0, -1) .

Також нам потрібно вказати RotationAngle , тобто величину повороту вершини щодо точки обертання. За замовчуванням значення має перебувати в

інтервалі від 0 до 1, де 0 - це 0 градусів, а 1 - це 360 градусів. Для отримання кута повороту ми можемо використовувати довжину напрямки згинання, помножену на максимальний поворот.

Множення на максимальний поворот в 0.2 означатиме, що максимальний кут повороту дорівнює 72 градусам.

Обчислення PivotPoint - трохи складніше завдання, тому що один меш трави містить кілька стебел. Це означає, що ми не можемо використовувати щось на зразок нода Object Position , тому що він буде повертати одну точку для всіх стебел трави[14].

В ідеалі слід використовувати сторонній 3D-редактор для збереження точок обертання всередині UV-каналів. Але для цього ми просто аппроксимируем точку обертання. Це можна зробити, пересунувшись з вершини вниз на певний зсув, як показано на рисунку 3.37.

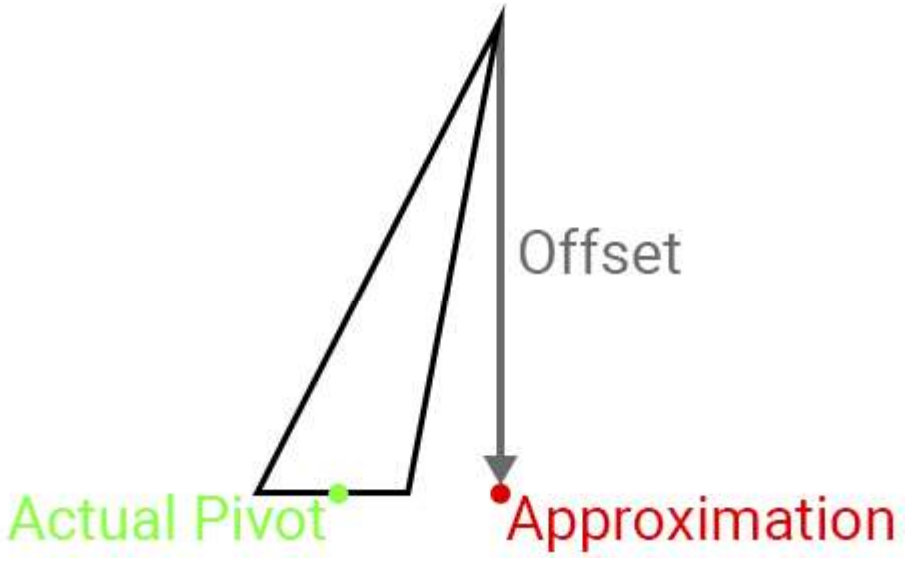

Рисунок 3.37 – Аппроксимація точки обертання

Далі нам потрібно виконати дві маски. Перша маска гарантує, що корінь стебла не буде рухатися. Друга маска гарантує, що на траву за межами області захоплення не діятиме векторне поле.

4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

4.1 Опис візуальної складової роботи алгоритму

Виходячи з принципів формування алгоритми поведінки рослинності ми повинні враховувати такі параметри, як структура мешей моделей рослин, кількість вертексів на моделі, розмір сфери злому, та вітер, який реалізує гнучкість рослин в грі, надаваючи реалізму сцені.

Принципи роботи алгоритму розібрані в третьому розділі, де детально описані всі етапи реалізації алгоритму. В цьому розділі ми звернемо увагу безпосередньо на реалізації та оптимізації алгоритму поведінки рослинності безпосередньо в нашій сцені. На рисунку 4.1 можна побачити вплив всих цих факторів.

Рисунок 4.1 – Ілюстрація впливу факторів на сцену

Розглядаючи детальніше можна побачити, що на поведінку рослини обночасно впливає декілька факторів. Прижата до землі рослина продовжує реагувати як на сусідні, так і на вітер. Але вплив вітру регулюється системно,

виходячи з його базової інтенсивності. На рисунку 4.2 можна побачити, що маса об’єкту, що діє на рослини впливаю на те, як сильно трава опуститься до землі і як швитко підніметься.

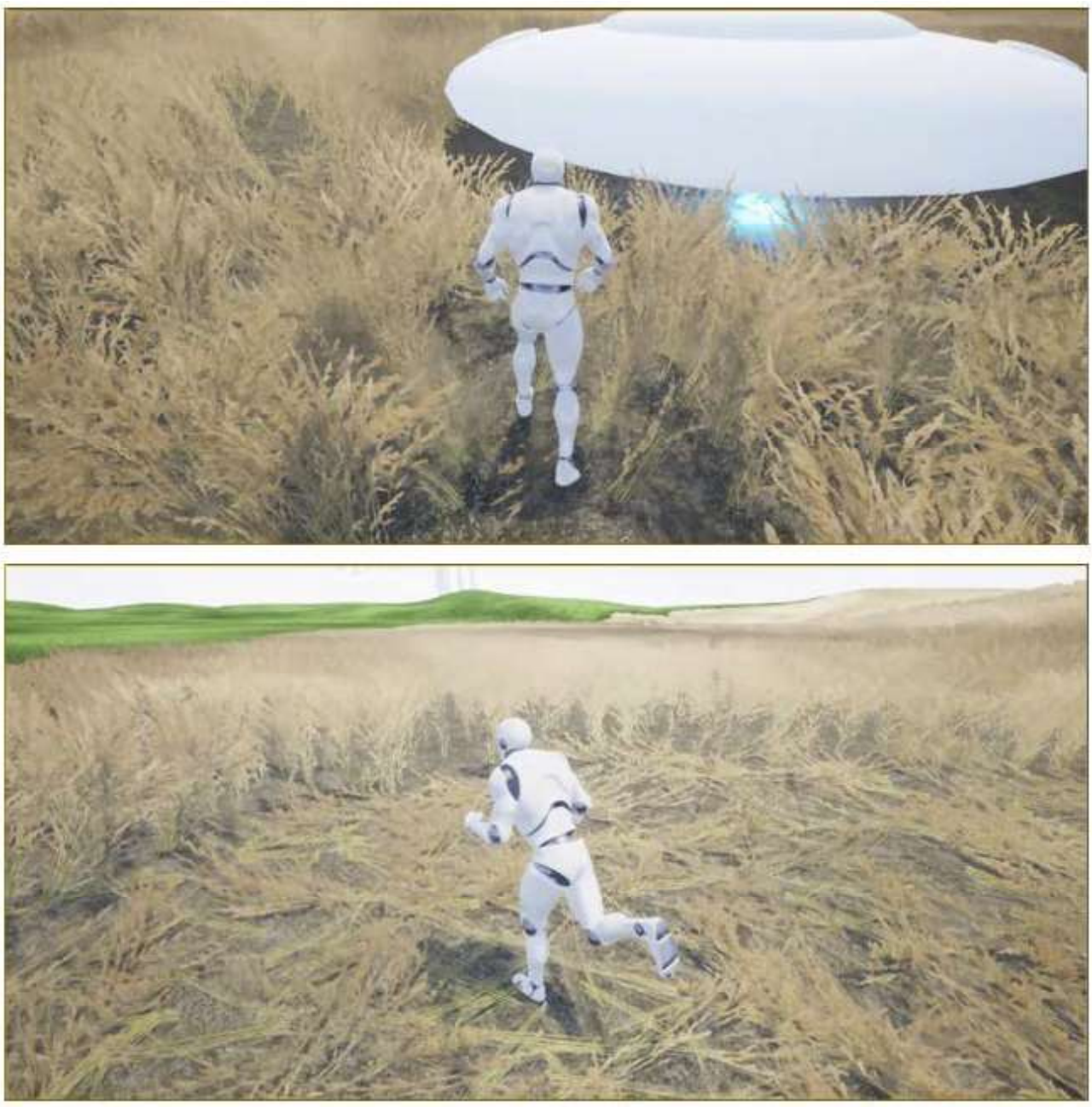

Рисунок 4.2 – Ілюстрація впливу маси об’єкту на рослини

З цього ми можемо зробити висновок, що алгоритм реалізує взаємодію рослин не тільки з персонажем, но і з ігровим оточенням, що надає сцені реалістичності.

Але ми бажали би, щоб рослини реагували також на ігрові ефекти, такі як магічні, або фізичні руйнування. На рисунку 4.3 можно побачити, як трава реагує на закляття та фізичний взрив.

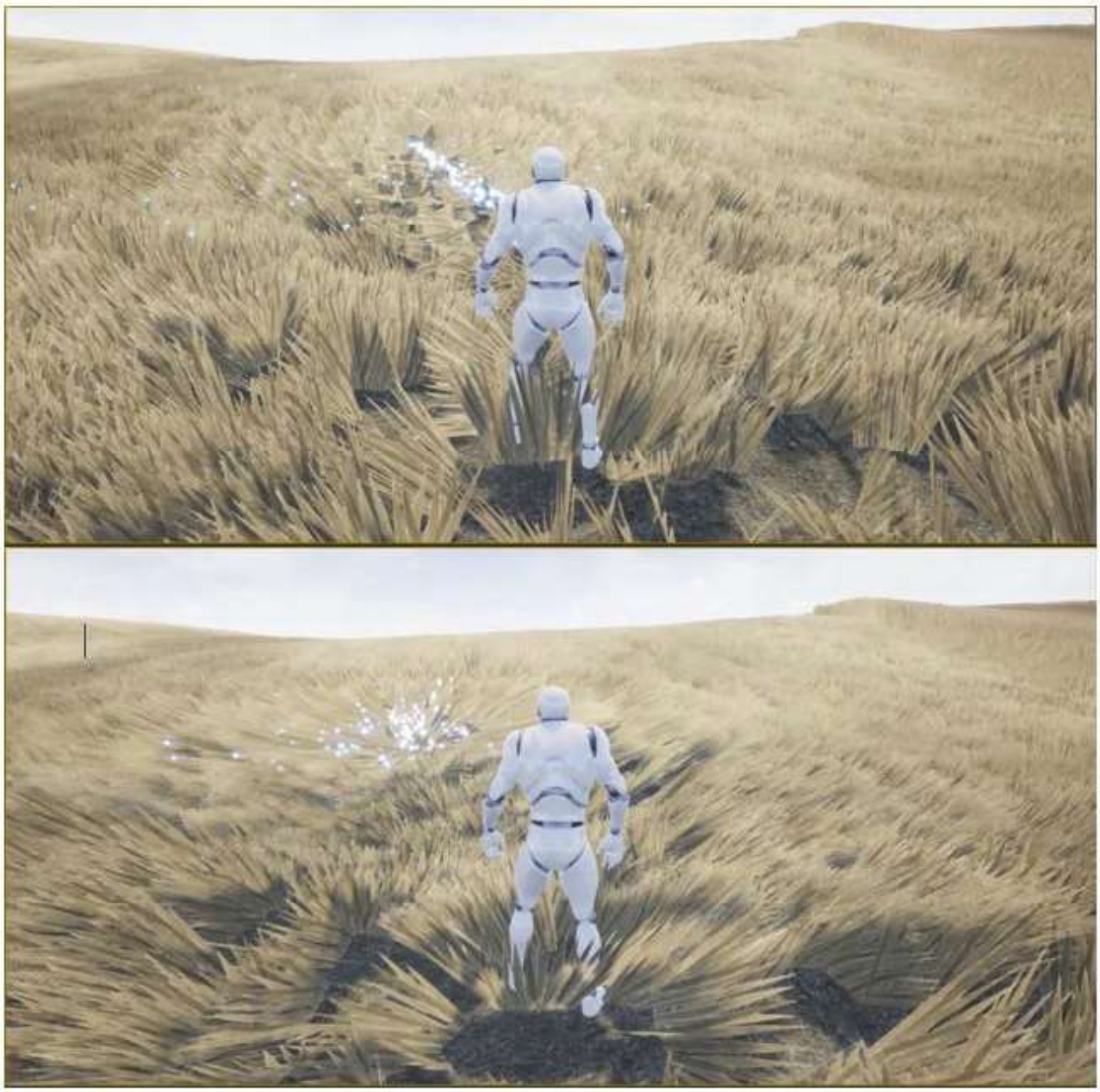

Рисунок 4.3 – Ілюстрація впливу закляття та фізичного взриву

Вихлдячи з цого можна констатувани, що алгоритм працює відповідно завдання, та реалівує усі функціональні потреби для реалістичного моделювання поведінки рослинності в іграх.

4.2 Опис програмної реалізації алгоритму

На сам перед необхідно розібрати реалізацію матеріалу трави. Для цього обераємо, як показано на рисунку 4.4 матеріал Grass2_M_Inst, який можна знайти в папці Materials во вкладеній папці Grass2.

Рисунок 4.4 – Матеріал Grass2_M_Inst

Далі розберемо декилька блюпринтів цого матеріалу, за допомогою яких створюється симуляція поведінки рослин.

Весь процес симуляції поведінки базується на декількох сферах, що мають власну модель поведінки, та власні параметри.

Рослина взаємодіє з такими сферами та вигинається видповідно куту обічній, що проходить між нимию.

Для ілюстрації такої сфери персонажу розглянемо рисунок 4.5, який ілюструє блюпрінти, котри реалізують цей функціонал.

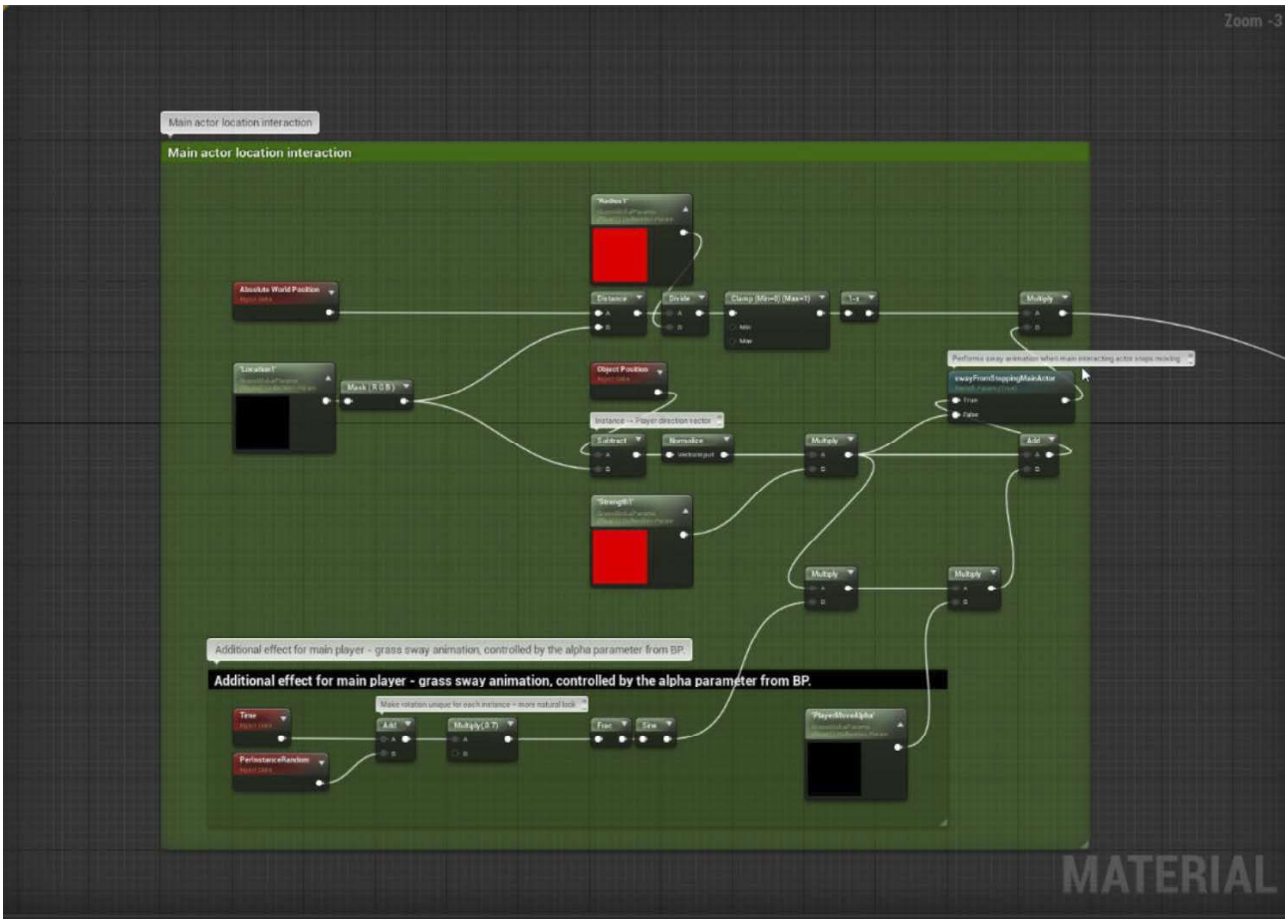

Рисунок 4.5 – Блюпринт Main actor location interaction

Розглянемо детальніше цей блюпрінт. Нода Location1 відповідає за позицію цієї сфери. Ноди Radius1 та Strenght1 відповідають за радіус та силу відповідно.

Такі сфери сворюються для кожного об’єкту на сцені, будь то ефект от фізичного взриву, чи від магічного ефекту. Регулюючи позицію, радіус та силу ми можемо створювати свої унікальні ефекти для кожного впливу на рослинність в сцені.

Наступним важливим вактором є вітер, що повинен постійно впливати на рослини та створювати ефект реалізму на сцені. Це дозволить нам створити для ігрока живий досвід, подібний до того, що створюється в таких іграх, як «Ghost of Tsushima». Для цого ми створюємо наступний блюпрінт, зображений на рисунку 4.6, який буде створювати ефект вітру на сцені.

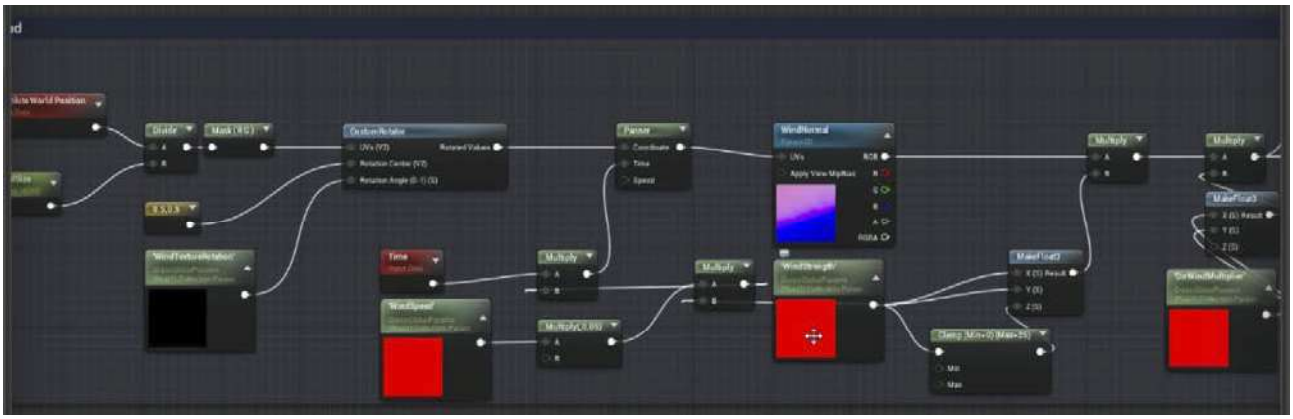

Рисунок 4.6 – Блюпринт поведінки вітру

Реалізован цей блюпринт через накладення Normal Map, які дозволяють створити ефект реалізму.

Normal Mapping – це технологія, яка використовується для імітації нерівностей поверхні на об'єкті. Вона застосовується, щоб зробити фінальну модель більш схожою на її HP (High Poly) версію. З її допомогою можна добити різні деталі, які не можна передати через геометрію з-за обмежень полигонажа на вашому проекті, і змусити вашу модель виглядати більш округленої для кращої передачі освітленості і більшої реалістичності.

Карти нормалей – це RGB-зображення, де кожен з каналів (червоний, зелений, синій) інтерпретується в X, Y і Z координати нормалей поверхні відповідно. Червоний канал простору дотичних карти нормалей відповідає за вісь X (нормалі спрямовані вліво або вправо), зелений канал за вісь Y (нормалі спрямовані вгору або вниз) та синій канал за вісь Z (нормалі спрямовані прямо від поверхні).

Осі U, V та N позначають напрями, в яких їх значення змінюються вздовж поверхні, так само як X, Y, Z представляють напрями, в яких змінюються їх значення у світових координатах.

Це означає, що ми можемо передавати координати простору дотичних через канали RGB карти нормалей. Червоний канал відповідальний за вісь U, синій через N, а зелений за V.

Для більш реалістичної картинки трема врахувати тінь від пропливаючих небом хмар. Для цього створемо блюпринт «Cloud shadows – direction, shape & tint», який зображен на рисунку 4.7.

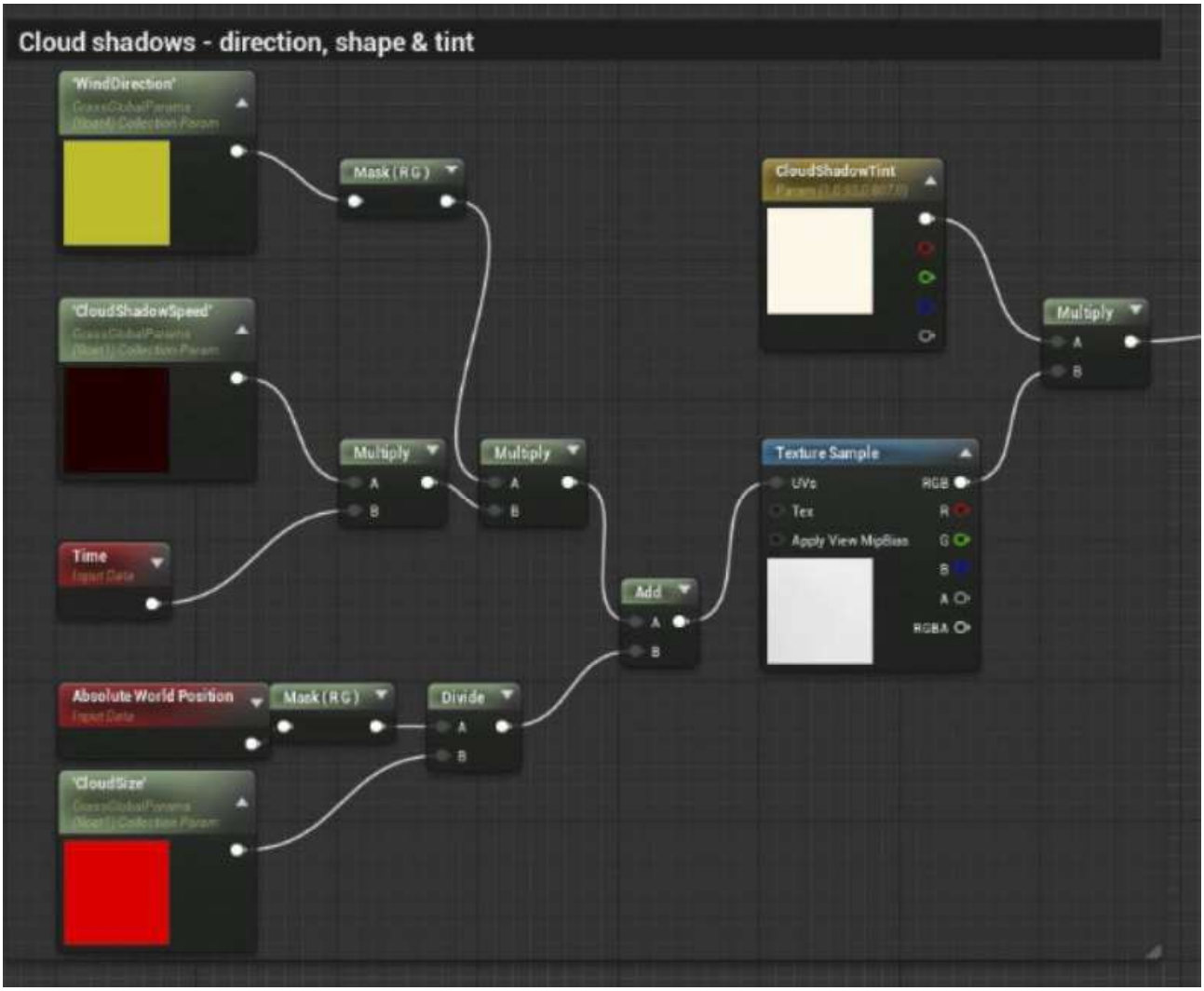

Рисунок 4.7 – Блюпринт Cloud shadows – direction, shape & tint

В якому за допомогою масок створюємо реалістичні тіні хмар, що рухаються по поверхні рослинного покрову. Цей ефект дає можливість ігроку усвідомлювати себе майже як в реальному світі.

Наприкінці нам необхідно провести бленд усіх створенних раніше блюпринтів, щоб вони мали вожливість впливати на наш матеріал гармонійно.

Режими накладення матеріалів описують як матеріал буде себе вести з навколишнім тлом. Перефразовуючи, це буде звучати приблизно так — режим

накладення дозволяє контролювати те, як движок буде комбінувати матеріал (джерело кольору) з тим, який вже знаходиться в буфері (цільової колір) при візуалізації цього матеріалу опиняється перед іншими пікселями.

Masked Blend Mode — режим змішування для об'єктів, в яких ми можемо контролювати видимість в бинароном вигляді.

Процес блендінгу ілюстрован на рисунку 4.8, де зображена нода, що відповідає за цей процес.

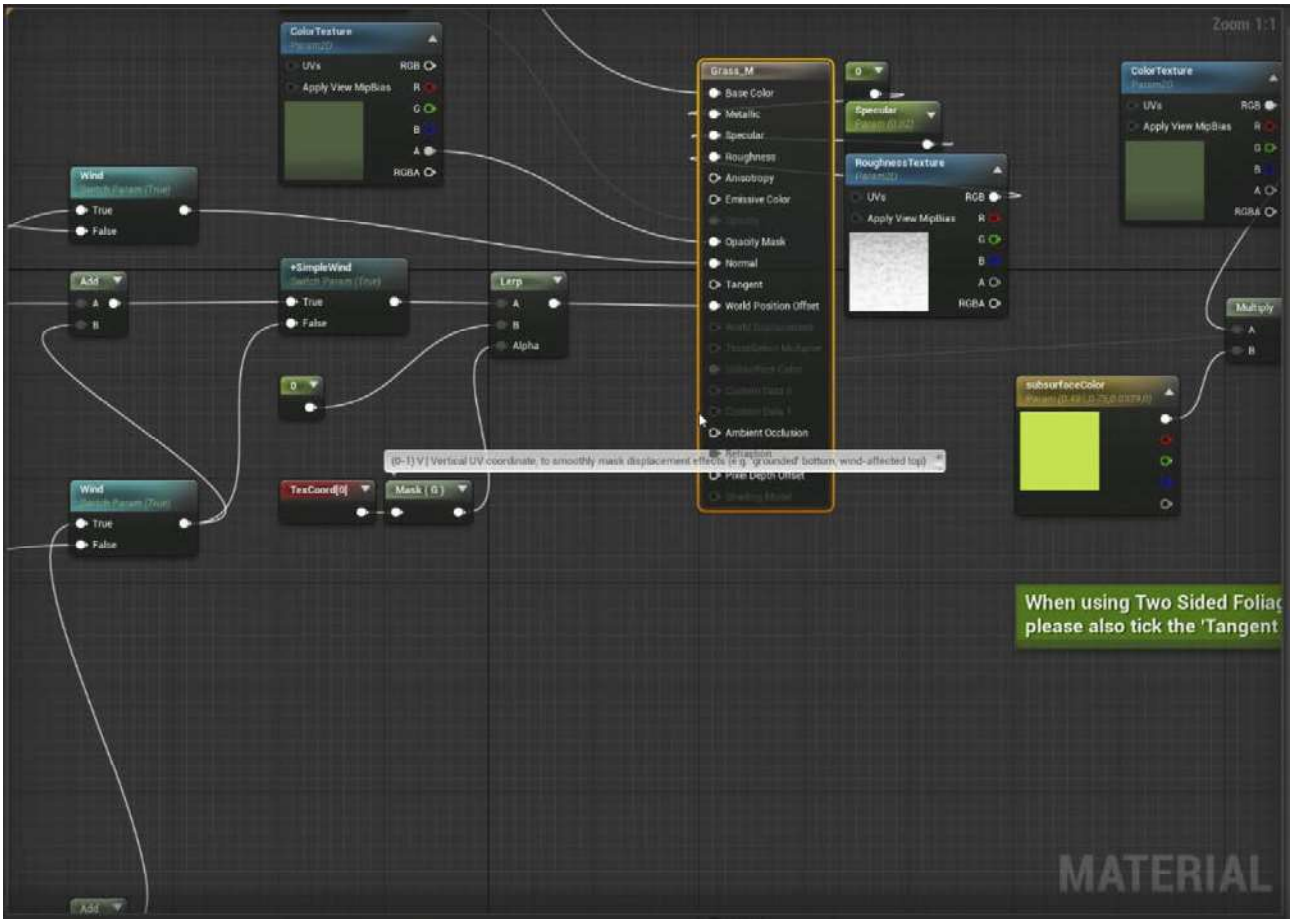

Рисунок 4.7 – Нода Grass_M

Во вхід World Position Offset передається позиція вертекса в просторі, сюди ми блендимо ефект сфери впливу та вітру

5 АНАЛІЗ МОЖЛИВИХ ЗАСТОСУВАНЬ ТА ТЕСТУВАННЯ ПРОГРАМНОІ СИСТЕМИ

5.1 Аналіз оптимізації програмного модулю

Найпростіше, з чого ми почнемо в цій статті - це базові поняття Frame Per Second і Milliseconds.

Зрозуміло, що Frame Per Second (далі FPS) - це кількість кадрів в секунду. Так само зрозуміло і те, що кількість кадрів визначається часом обробки цих кадрів, яке вимірюється в мілісекундах. Так, наприклад, щоб домогтися 60 кадрів в секунду, вам треба, щоб ваша гра оброблялася в середньому за один прохід в 16мс (в одній секунді 1000 мс).

Але чому саме 16мс? Чому не 30мс, наприклад? До чого таке прагнення? Адже в фільмах 24-29 кадрів використовують, і все нормально, а в іграх 30 кадрів - це погано.

Це така очевидна річ, що до однієї зі статей про геймдева я ніколи не замислювався, а чому всі хочуть 60 кадрів в секунду, і чому 30 кадрів в секунду в іграх це не круто, а в кіно - цього не помічаєш, і картинка плавна.

Все виявилося дуже просто - у фільмах кадр створюється з урахуванням руху. Камера фіксує не тільки поточний стан об'єктів, а й їх зміщення за час захоплення кадру. Тобто, кадри на плівці ніколи не бувають ідеально чистими - всі об'єкти так чи інакше змащені в русі, і коли це програється - створюється відчуття плавності руху навіть при 24 кадрах в секунду. А при нових технологіях і 120 кадрах в секунду (фільм "Gemini" був першим, хто показав в такому FPS) і зовсім здається, що дивишся якусь реальну сцену в театрі, а не в кіно.

В іграх же ситуація кардинально інша. Кожен кадр промальовується з нуля без урахування рухів. Тобто, всі об'єкти в цьому кадрі завжди статичні, і кадр не фіксує рух об'єктів за відведений йому час. В результаті 30 ігрових

кадрів в секунду здаються смикати, недостатніми для ока, щоб задовольнятися картинкою, і звідси прагнення до 60 кадрів і вище.

Так, це очевидна річ, якої мені не вистачало колись, щоб відчути саму суть проблеми. Воно було інтуїтивно зрозуміло, але ось так от чітко я не зустрічав опису до певної статті. Може бути, у вас так само буде і з цією.

У VR іграх потрібно 90 кадрів в секунду. Тут вимога закономірно. Якщо в 30 кадрів в секунду ви граєте і бачите смикання, і це в цілому нормально, то в VR вас не оточує квартира / офіс або де ви граєте. У VR вас оточує тільки гра і чорний фон. Тому чутливість до кількості кадрів зростає на порядок, а можливість вертіти головою в просторі створює додатковий дискомфорт при низькому FPS. А деяких і зовсім вивертає.

Що стосується об'єктів в грі, то тут дуже багато нюансів, але всі вони зводяться до 2 моментам:

- вертекси. Кількість вертексов має бути розумним. Пам'ятайте, що створюючи шви на розгортці, фарбуючи фейси по Vertex Color'у, роблячи модель з жорстких заходів - ви пропорційно збільшуєте кількість вертексов на рівному місці. Все повинно бути в міру, зважено і кожне рішення повинно бути обдумано;
- трикутники. Як ми всі знаємо - всі полігони (4хугольние n-вугільні) все одно діляться відкритий (а перед нею і самим двигуном) на трикутники. І якщо ви не порізали потрібні вам полігони заздалегідь, за вас це зробить движок в рандомном напрямку.

Кількість вертексов обтяжує ваше залізо додаткової обробкою. Тут все просто - чим більше вертексов, тим більше потрібно врахувати інформації для коректного відображення моделі.

Якщо ви робите модель з урахуванням запікання фасок в Normal Map, то ви прекрасно повинні розуміти, що ваші вертекси на жорстких гранях будуть розщеплюватися для того, щоб зберегти в собі інформацію про направлення Vertex Normal.

І ми всі прекрасно розуміємо, що на швах розгортки точки так само розщеплюються.

Розщеплення вертексов - дуже цікава тема і прі не вмілому зверненні може створити вам додатково x2 точок. Важливо пам'ятати всього 1 правило, яке дозволить вам трохи краще управляти моделлю:

Тобто, якщо ми в одній точці зробимо розріз шва розгортки, кордон фарбування кольору і зробимо жорсткі межі - це буде все ще одне розщеплення на 2 точки, тому що точка здатна зберігати в собі всі ці параметри одночасно.

Виходячи з цього розуміння можна прогнозувати, скільки буде реально займати ваша модель вертексов і як це можна контролювати.

Так само пам'ятайте, що другий і наступні шари розгортки так само збільшують кількість вертексов.

Трикутники - це візуальна площа, створена за допомогою трьох вертексов і яка буде відображатися на вашому екрані.

Кількість трикутників у вашій моделі визначає ступінь завантаженості вашої відеокарти для рендера цих поверхонь.

Здавалося б, круто, зрозуміло, потрібно робити меншу кількість трикутників (полігонів і т.д.) і рендер буде швидше обробляти модель. Але це не зовсім так. Розмір і положення трикутника так само важливо, як і те, як зібрана модель.

Справа в тому, що GPU (графічний процесор відеокарти) обробляє пікселі вашого монітора групами - по 4 пікселя. Тобто, не можна обробити 1 піксель для відображення трикутника, якщо він дуже вузький або закінчується на якомусь пікселі - завжди буде запущений цикл обробки на 4 пікселя.

LOD розшифровується, як Levels Of Detail - рівні деталізації. Це копії ваших моделей, але зі зменшеною кількістю вертексов, які замінюють вашу модель на віддаленій відстані[15].

У UE4 вони генеруються автоматично. Ну, тобто, потрібно натиснути на кнопки, але руками модель правити не потрібно, і це прискорює процес до пари кліків.

LOD'и - це те, з чим можна і потрібно дружити і завжди враховувати їх існування при роботі над об'єктами. Тобто, ви завжди можете використовувати високо деталізовані об'єкти, але з відстанню добре порізати їх, і ніхто цього не помітить, а ваша картинка буде при цьому на висоті і поміщатися в 16мс на обробку.

Третій не маловажний пункт в роботі з об'єктами - це розуміння того, як буде різатися ваш об'єкт при частковому відображенні камери.

Справа в тому, що якщо об'єкт якимось чином потрапляє в камеру і повинен відображатися, то він буде прорахований повністю весь і весь буде відображатися, навіть якщо потрапив в кадр малесенький трикутник.

Тому великі об'ємні об'єкти повинні бути зібрані таким чином, щоб їх можна було нарізати на шматки.

Обрізаючи відділи за певними правилами ми отримуємо хороший результат для рендера - нічого зайвого не обробляється за межами кадру.

LOD'и дозволяють нам знижувати навантаження на рендер.

Оптимізація Вертекс дозволяє нам знижувати розрахунки для відображення моделі.

Пам'ятайте - вертекси у великій кількості не так страшні, як зведення всіх граней в одну точку. Кількість вертексов на екрані легко зрізати LOD'амі, а неправильно зведені межі приведуть до ускладнення генерації LOD'ов і до навантажень для рендера.

Шейдери в UE4 - в першу чергу це матеріали. Те, в чому ви створюєте розрахунки текстур і накладаєте на об'єкти, щоб вони виглядали красиво, стилізовано, реалістично або якось ще.

У UE4 є безліч способів оптимізувати шейдери. Але я розгляну тільки основні моменти.

Шейдер прораховується на кожен займаний ним піксель. Займані пікселі визначаються на етапі растеризації, коли починає прораховуватимуться модель і те, де її видно. Якщо модель частково прикрита якимось об'єктом - прикрита

зона не потрапляє в розрахунок і не враховується при обробці і відображенні шейдеров.

Таким чином, якщо ви створите суперскладний шейдер, який жере мільйон інструкцій і вимагає ядерного реактора для розрахунків, але при цьому сам об'єкт буде займати площу 40 на 40 пікселів, то відображення цього шейдера буде займати мізерно мало часу і практично не зжере FPS.

При цьому, якщо ми збільшимо об'єкт до 100% площі екрану - ми отримаємо найдовший рендер кадру.

Інакше кажучи, чим більше площа об'єкта, тим менше навантаженим повинен бути шейдер.

Це говорить нам про те, що потрібно враховувати, в який момент ми можемо собі дозволити складний шейдер з сотнею розрахунків і 1000 інструкцій, а в який момент нам потрібно створити найпростіший матеріал, щоб він не віджиратись ресурси.

І немає, це не означає, що тепер на олівчик, який займає 20 пікселів, ви можете накласти суперскладний матеріал. Одного разу в який-небудь кат-сцені він може бути ключовим об'єктом в кадрі, і ваш GPU вас подякує.

Виходячи з вище написаного ми тепер розуміємо, що шейдер обробляється і відображається тільки в тих пікселях, де повинен. Але що ж відбувається з шейдером, коли в його коді є розгалуження?

Шейдеру плювати насправді, за якими правилами він повинен відобразитися. У будь-якому випадку при будь-якому результаті, навіть якщо ви пропишіть жорстку логіку - Весь код шейдера буде прорахований від і до для відображення в пікселі[16].

Потрібно розуміти, що якщо ви робите матеріал з поділом на 2 і більше об'єктів по UV координатам (так звані Атласовим шейдери з Атласовим текстурами), а в кадр потрапив тільки 1 об'єкт, то все одно ВЕСЬ код буде прорахований для відображення пікселя і завантажені будуть все текстури, навіть які не беруть участі в поточній роботі шейдера.

У зв'язку з цим потрібно кілька разів подумати, а чи варто запихати кілька об'єктів в один шейдер? Може бути, простіше буде, якщо на кожен об'єкт буде за власним шейдеру?

У UE4 існує потужна фіча - створювати складний матеріал і робити з нього максимально швидко різні копії, які можна дуже легко і так само швидко налаштовувати.

Для цього в шейдерах існує тип даних званий "Параметром". Цей тип даних дозволяє динамічно змінювати щось у шейдерах, наприклад, колір поверхні, при цьому не прораховуючи з нуля весь шейдер.

Так само існує такий тип об'єкта, як інстанси шейдера (матеріалу). Це копія шейдера, в якій ви можете змінювати параметри шейдера, і вони будуть моментально прораховуватимуться і відображатися. В цьому випадку вам не потрібно чекати, коли шейдер буде повністю перераховано.

Наприклад, ви створили шейдер, який базово можна накласти на будь-який об'єкт. Ви додали в нього параметр текстури (можна замінювати текстуру) і параметр кольору, який буде змінювати колір текстури.

Далі ви створюєте інстанси цього шейдера і тепер можете підставити в нього іншу текстуру і колір. Все легко, швидко і круто.

Але є серйозні нюанси, про які якось не прийнято говорити, тому що це малозначне і ускладнює життя.

У шейдерах можна використовувати константні значення (ті, які не можна змінювати в реальному часі) і параметри (динамічні значення, які можна змінювати в реальному часі).

Коли ми створюємо шейдер тільки на константних значеннях, то він прораховується повністю заздалегідь і окремо записує свої прорахунки. Так відбувається до тих пір, поки в коді не народжуються параметри.

Параметри в даному випадку стають невідомою величиною - ніхто не може передбачити, який саме колір в який момент ігрового часу вибере гравець, або вкаже програміст.

Тому з моменту, як в коді з'являється параметр, шейдер починає ділитися на 2 шматки коду - той, який він може прорахувати і зафіксувати залізно і той, який він не може передбачити.

У першому випадку шматок коду прораховується, і результати записуються, а в другому випадку - все залишається в динамічному стані і буде розраховане в режимі реального часу.

Це не означає, що тепер шейдер буде так само довго рендери, як прораховується в редакторі. Але це означає, що трохи навантаження на шейдер з'явиться. Вона практично не помітна в одному шейдера. Але якщо додавати в кожен шейдер по параметру - це може коштувати вам заповітні мілісекунди.

Що стосується інстанси, то у них є інша проблема - це параметри текстур. Коли ви створюєте параметр текстури в шейдера, то там обов'язково потрібно вказати якусь текстуру. А коли ви замінюєте текстуру в інстанси на іншу, то перша нікуди не пропадає.

Справа в тому, що, як я вже писав вище - весь код прораховується повністю в не залежності від того, яка в ньому логіка. І текстури, які ви вказали в базовому шейдера, будуть завантажені так само.

Тому. Якщо ви використовуєте інстанси з можливістю заміни текстур, то переконаєтеся, що ви не використовуєте сам базовий шейдер на інших об'єктах. Це дозволить вам підключити в усі його параметри текстур одну і ту ж текстуру-затичку, а її розмір рекомендую зробити в 1 піксель. Це буде легко для підвантаження, не займає пам'ять і розширить можливості підключення текстур через інстанси.

Пам'ятайте, що максимальне число текстур дорівнює 16 штук. І це разом з текстурами з базового шейдера і використовуваного інстанси.

З розумінням, як працюють параметри, інстанси і в цілому шейдери, можна тепер створювати трохи більше оптимізований код і робити трохи швидший рендер.

Шейдери - це та штука, до якої потрібно підходити максимально усвідомлено й обдумано. Іноді варто подивитися на текстури і зробити

заготовки чогось, ніж писати динамічний код, який буде робити все те ж саме, але в 5 разів дорожче. А іноді потрібно пошукати рішення, яке дозволить вам бути більш гнучкими і легше.

Draw Calls - це набір команд по відображенні об'єктів. Кожен об'єкт вимагає 2 дій для відтворення:

- обробка об'єкта для рендера;
- прорахунок текстури / шейдера для відображення об'єкта.

Перемикання між цими діями вимагає часу і сил CPU і GPU одночасно. І чим більше у нас об'єктів, чим більше у нас окремих текстур, тим більше у нас Draw Calls. А чим більше DC, тим менше FPS.

В першу чергу DC навантажує CPU. Умовно кажучи, процесор обробляє заявки на рендер, становить список об'єктів, які повинні відмалювати за один прогін і відправляє список відеокарти, яка швидко малює меши. Потім процесор визначає шейдер, який повинен наложиться на ці об'єкти і знову відправляє інформацію в GPU. А потім заново з подальшими об'єктами.

Як бачите, це займає певну кількість завдань, які повинен виконати процесор. А як відомо, швидкість центрального процесора набагато нижче бажаного, і чим менше у нас буде об'єктів, тим менше часу буде витрачатися на рендер[17].

Колись там в далекі роки середнім значенням DC було близько 1к запитів. Це вважалося нормою, при якій центральний і графічний процесори могли комфортно обробляти моделі і відображати їх, щоб не просідав FPS.

У сучасній грі це значення піднімається далеко за 3к і може перевищувати 5к (GTA V), або ж зовсім бути за межами 40к (Watch Dogs 2). Наприклад, середнє значення в Відьмак 3 - це 3.5к, іноді вище, іноді нижче.

У сучасних двигунах з'явилося таке поняття, як інстанси. Це копії об'єктів, які ми можемо розташувати на сцені, і які будуть проганяти процесором під один DC. Наприклад, трава. Трави дуже багато, десятки, сотні тисяч об'єктів на сцені, але всі вони Рендер через десяток DC і накриваються одним шейдером.

Але варто зазначити, що це працює тільки з одним об'єктом. Ви не можете загнати в один шейдер текстури для двох різних об'єктів (наприклад, куб і конус) і розраховувати, що у вас буде всього 1 DC. Їх буде мінімум 3:

- отрісовка першого об'єкта;
- отрісовка другого об'єкта;
- накладення шейдера на перший і другий об'єкт.

Якщо у ви використовуєте динамічний шейдер (інстанси або шейдер з параметрами), то тоді у вас буде 4 DC, так як додасться накладення шейдера на другий об'єкт окремо.

Чому? Тому що параметри в шейдера - це елемент невідомості, і необхідно прораховувати його при кожному використанні окремо. І якщо у вас є хоч 1 відрізняється параметр, то значить, у вас новий DC.

Тоді як же бути? Невже тільки траву і дерева можна засунути в один DC? Ні, є певні хитрощі, якими ви можете скористатися.

Для початку визначимо, що 1 DC може бути на 1000 об'єктів, якщо це копія одного і того ж об'єкта. Тепер, знаючи це ми можемо розставити на сцені 1000 однакових об'єктів, і вони всі будуть намальовані під одним DC з деякими застереженнями.

Об'єкти повинні бути вказані, як статичні (Static). Якщо якийсь із об'єктів динамічний, то він буде враховуватися окремо. Dynamic і Stationary об'єкти не групуються для DC, і кожен промальовується окремо.

Один шейдер для всіх об'єктів. Якщо там є параметри, то вони не повинні відрізнятися і / або редагуватися в реальному часі.

Можна використовувати компонент Instance Static Mesh для явної вказівки, які об'єкти повинні бути намальовані в один DC.

Якщо використання компонента Instance Static Mesh насильно запихає все копії об'єкта в один DC, то використання розставлених вручну статичних об'єктів і одного шейдера не гарантує вам одного DC. І це пов'язано з оптимізацією роботи рендеру і обробки об'єктів.

Движок UE4 - розумна штука. Він визначає оптимальну кількість статичних однакових об'єктів, яке можна записати в один DC і відправляє їх через CPU в GPU. Це необхідно, щоб CPU неспроможний перевантажився великим списком об'єктів, які йому потрібно обробити перш, ніж відправити GPU.

У зв'язку з цим, невелика кількість однакових об'єктів (в межах 10-20 штук) має сенс не засовувати в Instance Static Mesh (далі ISM), щоб у движка був самостійний вибір - як коректно це обробити.

Просто розстановка 100-1000 однакових статичних об'єктів без використання ISM призводить до інших проблем - скоріше за все всі об'єкти будуть намальовані під 2-5 DC (і це добре), але вони будуть займати пам'ять кожен окремо (а це не добре).

Якщо ми візьмемо, наприклад, кубики землі з нашої гри (на яких стоять наші відділи) і просто раскопіруем звичайними статичними об'єктами по грі, то DC ми збережемо в межах 1-2 значень, а ось поле з 300x300 кубиків зжере 7 гігабайт оперативної пам'яті . Без жартів =)

ISM ж дозволяє нам не тільки економити насильно в DC, а й економити пам'ять, так як 1 компонент ISM може зберігати собі тільки 1 об'єкт і займає пам'ять відповідно 1-му об'єкту. Навіть якщо ми створимо поле 300x300 кубиків, в оперативній пам'яті це буде все ще дорівнює 50 кбайт.

Але ISM має свій недолік. Він жорстко обмежує DC. Тобто, не можна буде поділити групу об'єктів на 2 або 3 DC - все буде довантажуючи через один DC, а значить, вам потрібно переконатися, щоб процесор не захлинувся в даних, які йому потрібно підготувати для відправки GPU. Можливо, має сенс розділити на кілька компонентів, якщо кількість вертексов у копіюється дуже велике.

Так само ISM не дозволяє об'єктам перемикати свої LOD'и на відстані. Тому для масового використання об'єктів існує другий компонент Hierarchical Instance Static Mesh Component (далі HISM). Він завантажує не тільки основний об'єкт, а й LOD'и цього об'єкта і коректно перемикає їх.

Ну і на прикладі поля 300 на 300 кубиків (90000 об'єктів) варто пам'ятати, що кількість вертексов відіграє особливу роль. Коли на кубику було 350 вертексов (здавалося б), а шейдер трави був напханий непотрібним і не використовуваним кодом, таке поле в 90.000 кубиків роняю FPS до 20 кадрів в секунду. Обрізавши кількість вертексов до 120 і поправивши трохи шейдер, ми зараз маємо стабільні 90-120 кадрів в секунду при дозволі 1920x1080. Залишилося дооптімізіровать шейдер, і наші кубики практично не будуть віднімати час на рендер.

Draw Call'и - це важлива частина оптимізації, яку не можна ігнорувати. Подивитися кількість DC в UE4 можна, запровадивши в консоль команду 'stat rhi ", де передостанній рядком буде число DC.

Оптимізація DC - це в першу чергу оптимізація навантаження на CPU. Занадто великий обсяг об'єктів може навантажити CPU, а надто частий - змусити його виконувати купу непотрібних інструкцій. Тут потрібно шукати баланс і слідувати йому.

5.2 Опис тестування розробленої програмної системи

TDD в геймдева застосовують досить рідко. Зазвичай простіше найняти тестувальника, ніж виділити розробника для написання тестів - так економляться і ресурси, і час. Тому кожен успішний приклад використання TDD стає цікавіше. Під катом переклад матеріалу, де цю техніку розробки застосували при створенні пересування персонажів в грі ElemenTerra.

Test-driven development або TDD (розробка через тестування) - це техніка розробки ПО, при якій весь процес розбивається на безліч невеликих циклів. Пишуться unit-тести, потім пишеться код, який ці тести проходить, а після робиться рефакторинг. І алгоритм повторюється.

З TDD в геймдева є дві проблеми. По-перше, багато ігрові функції мають суб'єктивні цілі, які не піддаються вимірюванню. А по-друге, важко написати тести, що охоплюють всі можливості простору світів, які сповнені складних взаємодіючих об'єктів. Розробникам, які хочуть, щоб руху їх персонажів «виглядали добре» або фізичні симуляції «не виглядали смиканими», буде важко висловити ці метрики у вигляді детермінованих умов «пройдено / не пройдене».

Проте, техніка TDD застосовна до складних і суб'єктивним особливостям - наприклад, до руху персонажів [18].

Як і unit-тести, ці рівні налагодження можуть використовуватися для відтворення та діагностики помилок. Але де в чому вони відрізняються:

Unit-тести ділять системи на частини і оцінюють кожну окремо, в той час як налагоджувальні рівні проводять тести в більш цілісному вигляді. Після знаходження помилки на отладочном рівні, розробникам все ще може знадобитися пошук помилковою точки вручну.

Unit-тести автоматизовані і повинні кожен раз давати детерміновані результати, в той час як багато налагоджувальні рівні «управляються» гравцем. Це створює різницю в сесіях.

Але це не означає, що unit-тести краще налагоджувальних рівнів. Останні часто більш практичні. Однак unit-тестування можна застосовувати навіть в тих системах, де воно традиційно не було присутнє.

Використання TDD для пересування на сцені було величезним плюсом, але підхід мав кілька обмежень.

Unit-тести оцінювали кожну особливість руху окремо, тому помилки з комбінаціями декількох особливостей не вважалися. Іноді доводилося доповнювати unit-тести традиційними рівнями налагодження.

Розробники можуть витратити занадто багато сил на рівні для unit-тестів, які гравець ніколи не оцінить. Такі внутрішні функції можуть забрати багато часу і поставити під загрозу важливіші Майлстоун. Щоб цього не сталося - ретельно вивчіть, де і коли варто використовувати unit-тести.

Перш ніж витрачати час на автоматизоване тестування, потрібно перевірити, чи зможемо ми оцінити функцію за допомогою звичайних ігрових елементів управління. Якщо ви хочете переконатися, що ваші ключі відмикають двері, заспавніте ключ і відкрийте їм двері. Створення unit-тестів для цієї функції було б втратою часу - ручне тестування займає всього кілька секунд.

Автоматизовані unit-тести виправдані, коли є відомі і важко відтворювані випадки. Така ситуація може бути важким або неможливим для відтворення за допомогою ігрових елементів управління, а тести - легко.

Ігровий дизайн повністю заснований на ітераціях, тому цілі фичей можуть змінюватися в міру того, як ваша гра буде перероблятися. Навіть невеликі зміни можуть анулювати метрики, за якими ви оцінюєте свої фічі, і, отже, будь-які unit-тести. Якщо поведінка істот під час їжі, сну і взаємодії з гравцем кілька разів змінювалися, то перехід з пункту А в пункт Б залишався незмінним. Тому код пересування і його unit-тести залишалися актуальними протягом всієї розробки.

Була у вас ситуація, коли ви завершуєте один з останніх ТАСК перед відправкою гри, і раптом виявляєте помилку, яка ламає правила? Причому в функції, яку ви закінчили багато років назад. Ігри є гігантські взаємопов'язані системи, і тому цілком природно, що додавання нової функції В може привести до виходу з ладу старої функції А.

Це не так погано, коли зламана функція використовується повсюдно (наприклад стрибок) - ви повинні негайно помітити поломку механіки. Помилки, виявлені в пізньої розробці, можуть порушити розклад, а після запуску можуть нашкодити ігрового процесу[19].

Якщо для створення тесту фичи потрібно багато часу, вона нескладна і навряд чи буде змінена (або з нею можна буде впоратися, якщо вона зламається в пізньої розробці), то unit-тести можуть доставити більше проблем, ніж користі. Якщо тести зробити легко, функція нестабільна і взаємопов'язана (або її помилки будуть займуть багато часу), то тести допоможуть.

ВИСНОВКИ

Під час кваліфікаційної магістерської роботи було досліджено алгоритми поведінки рослин в комп’ютерних іграх на рушії UE4.

Було вирішено практичні задачі моделювання поведінки рослинного оточення необхідно створити програмний модуль на рушії UE4, які складалися з наступних частин:

- ігрове середовище на якому будуть розміщуватися рослини;
- рослини, поведінку яких ми будемо моделювати;
- об’єкти та ефекти, що будуть впливати на поведінку рослин;
- вітер, шо симулює поведінку рослин без зовнішнього втручання;
- тіні від хмар, що створюють реалістичну атмосферу на сцені.

Програмний продукт реалізовує наступні функції:

- поведінка рослин в спокої;
- вплив актора на рослини;
- вплив вибухів та магічних ефектів на рослини;
- вплив вітру на рослини;
- бленд усіх ефектів;
- реалістичні тіні від хмар.

Програмне забезпечення відповідає основним вимогам, що висуваються до подібних програмних модулів, такі як стійкість до навантаження, оптимальне використання ресурсів та оптимізація графічної складової.

Таким чином, усі поставлені задачі при виконанні кваліфікаційної роботи магістра виконані в повному обсязі.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Mitch McCaffrey Unreal Engine VR Cookbook: Developing Virtual Reality with UE4. Addison-Wesley Professional, 2017 – 406 с.

2. Spatial interpretation of the notion of relation and its application in the system of artificial intelligence Proniuk, G., Geseleva, N., Kyrychenko, I., Tereshchenko, G. CEUR Workshop Proceedings, 2019, 2362

3. Software System for Virtual Laboratory Works. XV International Scientific and Technical Conference Computer Science and Information Technologies – CSIT-2020, Zbarazh Castle, Ukraine.- 23-26 September, 2020.-P.396-400.

4. A Study of Optimization Models for Creation of Artificial Intelligence for The Computer Game in The Tower Defense Genre. Problem of Infocommunications. Science and Technology (PIC S&T'2020), Kharkiv, Ukraine.- 6-9 October 2020.John Blain The Complete Guide to Blender Graphics. CRC, 2017–557 с

5. Chowdhury K. Mastering Visual Studio 2017. New York, 2017. 466 с.

6. Monografia pokonferencyjna: Science , research ,development #28 technics and technology. Diamond Trading Tour Warszawa, 2020 – 72с.

7. ITC: веб-сайт. – URL: itc.ua/blogs/steam-podvel-itogi-2019-goda-95-mln-aktivnyhpolzovatelej-20-mlrd-chasov-v-igre-i-70-tys-obzorov-v-den/ (Дата звернення: 14.05.2021)

8. Unreal Engine FAQ: веб-сайт. – URL: www.unrealengine.com/faq (Дата звернення 20.03.2021)

9. Unreal Engine Features: веб-сайт. – URL: www.unrealengine.com/unreal-engine-4 (Дата звернення 21.03.2021)

10. Офіційний канал Unreal Engine: веб-сайт. – URL: www.youtube.com/channel/UCBobmJyzsJ6Ll7UbfhI4iwQ

11. Альфред В. Ахо, Джон Э. Хопкрофт, Джеффри Д. Ульман. Структуры данных и алгоритмы. – Издательство «Диалектика-Вильямс». – 2019. – 400 с.

12. Герберт Шилдт. C++. Полное руководство. Классическое издание. – Издательство «Диалектика». – 2020. – 800 с.

13. Jamis Buck. Mazes for Programmers: Code Your Own Twisty Little Passages. – Pragmatic Bookshelf. 2015. – 286 с.

14. Aram Cookson, Ryan Dowling Soka, Clinton Crumpler. Unreal Engine 4 Game Development in 24 Hours, Sams Teach Yourself. Sams Publishing. – 2016. – 496 с.

15. Unreal Property System: веб-сайт. – URL. www.unrealengine.com/blog/unrealproperty-system-reflection (дата звернення 20.05.2020)

16. Static Meshes | Unreal Engine: веб-сайт. – URL. docs.unrealengine.com/latest/INT/Engine/Content/Types/StaticMeshes/ (дата звернення 28.04.2020)

17. Рэшка Джефф, Дастин Элфрид, Пол Джон. Тестирование программного обеспечения. - Лори, 2012. - 568 с.

18. Виды Тестирования Программного Обеспечения / Тестирование ПроТестинг – URL: http://www.protesting.ru/testing/testtypes.html – (дата звернення: 05.04.2020).