Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

ВИКОРИСТАННЯ ШЕЙДЕРНИХ ПРОГРАМ ТА ЕФЕКТІВ ДЛЯ ПОКРАЩЕННЯ 2D-ГРАФІКИ В UNITY

Текстова частина до курсової роботи

за спеціальністю „Інженерія програмного запезпечення ” 121

Керівник курсової роботи
ст. в.   Бучко О. А.

____
(підпис)
“____” __________ 2020 р.

Виконалп студентка
Дєточка А.Р.
“____” __________ 2020 р.

Київ 2020

Зміст

Вступ. Формулювання задачі та проблеми. 3

Теорія .... **Помилка! Закладку не визначено.**

Про шейдеры в Юнити. Чистые, графы и дефолтные. 5

Build-in shaders. Universal render pipeline. 7

Shader Graph 8

Custom shaders. 9

Bumpmapping .... **Помилка! Закладку не визначено.**

Tangent Space .... **Помилка! Закладку не визначено.**

Normal Maps .... **Помилка! Закладку не визначено.**

Displacement Maps 12

Using These Maps Together .... **Помилка! Закладку не визначено.**

Normal mapping in unity .... **Помилка! Закладку не визначено.**

Particle systems 15

Particle system in Unity 18

Практика .... **Помилка! Закладку не визначено.**

Освітлення в Unity 22

Шейдер ефекту колихання від вітру на Shader Graph 24

Система часток в Unity 26

Кастомна система часток 28

Висновки. 31

Вступ. Формулювання задачі та проблеми.

Зі збільшенням потреби у використанні комп'ютерної графіки збільшилось і кількість інструментів для її покращення. Нині розроблена велика кількість таких технологій. А також з'явилися нові технології, що дозволяють використовувати вже набуті знання та полегшити розробку.

Одна з найбільших складностей в розробці ефектів полягає в тестуванні функцій, у складності чи неможливості перевірки проміжних даних. Інша полягає у низькорівневій роботі з даними. Нині доступні способи обійти такі складності і полегшити розробку та прискорити її ефективність.

Метою цієї роботи було пошук та дослідження таких засобів для роботи з 2Д графікою. Специфіка такої графіки полягає в тому, що більша частина засобів та методів покращення графіки створюється для 3Д простору, для наближення графіки до реалізму. Але 2Д графіку частіше використовують не для наближення до реалізму, а для художньої інтерпретації світу. І тому засоби для покращення такої графіки мають не стільки створювати максимальне наближення до реальності, скільки чіткіше та ясніше виражати художню інтерпретацію.

Результатом цієї роботи є сцена з макету гри із використанням досліджених та розроблених ефектів, а також окремо розроблена програма для створення ефектів з частинками. Методами дослідження є огляд існуючих алгоритмів та реалізація їх за допомогою мови C# та платформою розробки Unity.

Джерелами дослідження виступають переважно літературні й веб ресурси з інформацією про ефекти, шейдери та Unity.

Праця поділена на два розділи. Перший розділ містить визначення, опис ефектів та засобів їх розробки на Unity.

Другий розділ містить розгляд окремих методів та реалізація таких ефектів як освітлення, рух за допомогою шуму та система часток.

Постановка задачі:

1. Дослідити поняття ефектів та засоби їх реалізації.

2. Розглянути існуючі алгоритми створення ефектів для графіки в іграх.

3. Реалізувати макет сцени гри із реалізованими ефектами.

Розділ 1

Шейдери в Unity. Види шейдерів.

Шейдери в Unity можна записати одним із трьох різних способів:

Поверхневі шейдери (Surface Shaders)

Поверхневі шейдери - найкращий варіант, якщо на шейдер має впливати світло і тіні. Поверхневі шейдери дозволяють легко писати складні шейдери щільно - це вищий рівень абстракції для взаємодії з освітлювальним конвеєром Unity. Більшість поверхневих шейдерів автоматично підтримують освітлення вперед і відкладене (forward and deferred). Поверхневі шейдери пишуться в пару рядків Cg / HLSL, і набагато більше коду отримується автоматично з цього.

Не слід використовувати поверхневі шейдери, якщо шейдер нічого не робить із світлом. Для післяоброблених ефектів або багатьох шейдерів зі спеціальними ефектами поверхневі шейдери - це неоптимальний варіант, оскільки вони роблять безліч обчислень освітлення без поважних причин.

Вершинні та фрагментні шейдери (Vertex and Fragment Shaders)

Вершинні та фрагментні шейдери необхідні, якщо шейдеру не потрібно взаємодіяти з освітленням або якщо потрібні дуже екзотичні ефекти, з якими поверхневі шейдери не можуть впоратися. Програми шейдерів, написані таким чином, є найбільш гнучким способом створення потрібного ефекту, але це доводиться ціною: потрібно написати більше коду, і важче змусити його взаємодіяти з освітленням. Ці шейдери також написані в Cg / HLSL.

Шейдери з фіксованою функцією (Fixed Function Shaders)

Шейдери з фіксованою функцією є застарілим синтаксисом Shader для дуже простих ефектів. Доцільно писати програмовані шейдери, оскільки це дозволяє значно більше гнучкості. Шейдери з фіксованою функцією повністю написані мовою, що називається ShaderLab, яка схожа на файли .FX Microsoft або файли CgFX NVIDIA. Внутрішньо всі шейдери з фіксованою функцією перетворюються в шейдери вершин та фрагментів під час імпорту шейдера.

ShaderLab

Незалежно від того, який тип обрано, фактичний код Shader завжди загорнутий у ShaderLab, який використовується для організації структури Shader. Це виглядає приблизно так:

```
Shader "MyShader" {
  Properties {
    _MyTexture ("My Texture", 2D) = "white" { }
    // Place other properties like colors or vectors here as well
  }
  SubShader {
    // here goes your
    // - Surface Shader or
    // - Vertex and Fragment Shader or
    // - Fixed Function Shader
  }
  SubShader {
    // Place a simpler "fallback" version of the SubShader above
    // that can run on older graphics cards here
```

```
    }
}
```

Вбудовані шейдери. Universal render pipeline.

Unity містить велике різноманіття реалізованих шейдерів та інших засобів покращення графіки. Для своєї задачі я розглядала бібліотеку Universal Render Pipeline.

“The Universal Render Pipeline (URP) - це попередньо вбудований Scriptable Render Pipeline, виконаний компанією Unity. URP надає зручні для виконавців робочі процеси, які дозволяють швидко та легко створювати оптимізовану графіку на різних платформах - від мобільних до високих класів консолей та ПК.”

URP має широкий функціонал для роботи з 3Д графікою. Але мене цікавить його робота з 2Д.

Нижче наведено 2D пов'язані функції, які використовують універсальний рендер-конвеєр (URP):

Під час використання URP з обраним 2D рендерером компонент Light 2D вводить спосіб застосування 2D оптимізованого освітлення для спрайтів.

Можна вибрати один з декількох типів світла за допомогою компонента Light 2D. Доступні такі типи світла:

- Freeform: Можна редагувати форму цього типу Light за допомогою редактора контурів.
- Sprite: Можна вибрати спрайт для створення цього типу Light.
- Parametric: Можна використовувати n-однобічний багатокутник для створення цього типу 2D-світла.

- Point: Можна керувати внутрішнім і зовнішнім радіусом, напрямком і кутом цього типу Світла.
- Global: Це 2D-світло впливає на всі надані спрайти на всіх цільових сортувальних шарах.

Пакет включає 2D Renderer Data Asset, який містить параметри стилів змішування і дозволяє створити до чотирьох спеціальних Light операцій для проекту.

2D Lights можуть взаємодіяти зі картою нормалей та маскою текстур, пов'язані зі спрайтами, щоб створити передові освітлювальні ефекти. Щоб зв'язати ці додаткові текстури зі спрайтом, потрібно обрати спрайт і відкрити редактор спрайта. Потім обрати модуль "Вторинні текстури" зі спадного меню у верхньому лівому куті вікна Редактора спрайт.

Його вбудовані світло і тіні зручно використовувати для додавання глибини кадрів гри.

Також URP підтримує інтеграцію з іншим модулем, Shader Graph, який значно полегшує написання шейдерів.

Shader Graph

“Shader Graph enables you to build shaders visually. Instead of writing code, you create and connect nodes in a graph framework. Shader Graph gives instant feedback that reflects your changes, and it’s simple enough for users who are new to shader creation.”

Shader Graph доступний у вікні Керування пакетами у версіях Unity 2018.1 та новіших версій. Якщо встановлено попередньо Scriptable Render Pipeline (SRP), такий як Universal Render Pipeline (URP) або конвеєр рендерингу

високої чіткості (HDRP), Unity автоматично встановлює Shader Graph у проект.

Створення шейдерів в Unity з нуля.

Написання шейдерів вершин та фрагментів

ShaderLab шейдери описують властивості, які відображає Material Inspector, містять кілька реалізацій шейдерів для різних графічних апаратних засобів та налаштовують стан апаратних засобів з фіксованою функцією. Програмовані шейдери, такі як вершинні та фрагментні програми, є лише частиною концепції шейдера ShaderLab.

Програми шейдерів написані мовою HLSL шляхом вбудовування фрагментів у текст шейдера, всередині команди Pass. Зазвичай вони виглядають так:

```
Pass {
    // ... the usual pass state setup ...
    CGPROGRAM
    // compilation directives for this snippet, e.g.:
    #pragma vertex vert
    #pragma fragment frag
    // the Cg/HLSL code itself

    ENDCG
    // ... the rest of pass setup ...
}
```

Фрагменти програми на HLSL записуються між ключовими словами CGPROGRAM та ENDCG, або ж між HLSLPROGRAM та ENDHLSL.

Остання форма не включає автоматично вбудовані файли заголовків HLSLSupport та UnityShaderVariables.

Карта нормалей та інші засоби для деталізації графіки

Є різні способи показати деталізацію без великого навантаження систем, і тут будуть розглянуті три методи - рельєфне текстурування (Bumpmapping), карта нормалей (normal map) та displacement map.

Можна сказати, що вони роблять майже те саме. Кожна з цих трьох типів мап створює щось подібне до додаткової роздільної здатності чи деталі на поверхні геометрії. Деякі з цих деталей справжні, а деякі - ні.

Всі ці додаткові деталі можна було б представити геометрією, але це створило б величезну кількість полігонів, які б дуже ускладнили роботу з графікою. Іншим рішенням проблеми є розгляд впливу доданої деталі. Оскільки додаткові деталі геометрії невеликі, вони насправді не впливають на форму предмета. Однак вплив здебільшого можна побачити на нормалі поверхні.

Рельєфне текстурування

Додаткові геометричні деталі можуть бути представлені прямолінійно. У цій формі можна створити напівтонову текстуру, яка представляє висоту кожного пікселя відносно один одного, при цьому білі пікселі вище ніж чорні пікселі.

Хоча таку текстуру легко створити, вона не настільки корисна при візуалізації. Нас цікавить нормальна поверхня для певного пікселя, а не його висота. Однак можна попередньо обробити текстуру висоти, щоб отримати зміщення U та V для кожного пікселя.

Замість того, щоб зберігати отримане нормальне значення при попередній обробці, можна просто виявити зміщення U і V, що представляє різницю нормалізованого вектора.

Відносний простір

Маючи справу зі normal maps та bump maps, кожен тексель відноситься до поточної поверхні нормалі, яка зазвичай знаходиться в просторі об'єктів або у просторі світу. Однак, оскільки текстурується об'єкт за допомогою текстури або bumpmap, яка не обов'язково створена для цього об’єкта, не існує способу, щоб bumpmap або normal map визначали нову нормаль без будь-якого знання поверхні об'єкта.

Одне рішення - переконатися, що текстура співвідноситься з об'єктом, а потім переконатися, що normal map або bump map посилається на нормалі в об'єктному просторі. Однак це недоцільно, оскільки воно перешкоджає повторному використанню текстури та потребує спеціального створення текстури для кожного зіткнутого об’єкта у сцені. Той факт, що bump текстура має знання поверхні об'єктів, також забороняє анімувати або деформувати поверхню, ще більше обмежує використання bumpmapping.

Більш практичним підходом є створення нової рівномірної системи координат, яка однакова для кожного пікселя на об'єкті і може бути побудована з bump/нормальної текстури. Для цього потрібно врахувати, що якби об’єкт не був оброблений, нормальна точка переднього руху на поверхні була б перпендикулярною до поверхні.

Застосовуючи bumpmap або normal map до поверхні, по суті, модифікується нормаль уздовж двох векторів, що йдуть уздовж координат текстури U та V Якщо скласти все це разом, виходить тривекторна система координат, яку можна обчислити на основі пікселя.

Карти нормалей

Найбільш поширене використання дотичного простору при роботі з освітленням попіксельно - це використання мап нормалей для додання деталей, що нагадують удари на поверхню об’єкта.

Хоча результат схожий із застосуванням bumpmap, засоби, що застосовуються для досягнення результатів, різні. Під час використання bumpmap використовувались для кодування зміщення від інтерпольованого нормального за допомогою двох кольорових компонентів. У випадку карти нормалей текстура використовує триколірний компонент, який представляє фактичну нормаль в дотичному просторі. Це представлення передбачає синій компонент, або Z, являє собою нормальне положення за замовчуванням у дотичному просторі.

Такі текстури складно створити вручну, оскільки мова йде про фарбування векторів за допомогою кольорів. Однак такі інструменти, як плагін NVIDIA Photoshop, можуть генерувати такі звичайні текстури карти з текстури висоти сірого кольору.

Displacement Maps

Що стосується створення додаткових деталей для сіток з низькою роздільною здатністю, Displacement Maps є найкращим варіантом. Ці типи карт фізично витісняють (як випливає з назви) сітку, до якої вони застосовуються. Для того, щоб деталізація була створена на основі карти переміщення, зазвичай сітка повинна бути підрозділена або тессельована, щоб була створена реальна геометрія.

Чудова річ у картках переміщення полягає в тому, що вони насправді можуть бути випечені з моделі високої роздільної здатності або намальовані вручну. Як і bump map, карта зміщення складається з значень масштабування сірого. Хоча можна використовувати 8-бітну карту переміщення, майже завжди

будуть кращі результати, використовуючи 16- або 32-бітну карту переміщення. Незважаючи на те, що 8-бітні файли можуть виглядати добре у двовимірному просторі, коли їх впроваджують у 3D, вони іноді можуть спричиняти смуги чи інші артефакти внаслідок недостатнього діапазону значень.

Але у цього метода є недоліки. Створити всю цю додаткову геометрію в режимі реального часу надзвичайно складно і важко для системи. Через це більшість 3D-програм обчислюють кінцеві результати переміщення під час візуалізації. У порівнянні зі bump або normal картами карта переміщення також додасть значного часу рендерам. Внаслідок цієї додаткової геометрії важко замінити результати карти переміщення. Оскільки поверхня фактично модифікована, силует відображає додаткову геометрію. Потрібно зважувати витрати карти переміщення проти додаткової вигоди, перш ніж вирішити використовувати її.

Використання карт разом

У деяких випадках можливо комбінувати або bump, або normal карту із картою переміщення на одному ресурсі. Найкращим способом зробити це було б використовувати зміщення для великих змін геометрії, а потім нормальне або bump для дрібних деталей. Незалежно від того, яку карту ви вирішено використовувати, розуміння того, як працює кожна карта та як її сильні, так і слабкі сторони, лише полегшить рішення. Зрештою, обрана карта повинна бути тією, яка найкраще відповідає потребам сценарію, з яким стикаєтесь.

Використання карт нормалей в Unity

Мапу нормалей можна імпортувати, помістивши файл текстури у папку ассетів. Однак потрібно сказати Unity, що ця текстура - це мапа нормалей. Це можна зробити, змінивши налаштування "Тип текстури" на "Нормальна карта" в налаштуваннях інспектора імпорту.

Вторинні звичайні карти

В інспекторі матеріалів для стандартного шейдера внизу вниз є другий слот Normal Map. Це дозволяє використовувати додаткову звичайну карту для створення додаткових деталей. Можна додати звичайну карту до цього слота так само, як і звичайний слот для карти нормалей, але тут потрібно використовувати інший масштаб або частоту плитки, щоб дві нормальні карти разом створювали високий рівень деталізації на різні масштаби. Наприклад, головна звичайна карта може визначати деталі обшивки стіни або транспортного засобу з вирізками для країв панелі. Вторинна нормальна карта може забезпечити дуже тонку деталізацію для подряпин та зносу на поверхні, яка може бути обложена плиткою в 5 - 10 разів більше масштабу основної карти звичайного. Ці деталі можуть бути настільки тонкими, що видно лише при уважному огляді. Щоб ця детальна інформація була розміщена на базовій нормальній карті, потрібно мати базову звичайну карту неймовірно великою, проте, поєднуючи дві в різних масштабах, можна досягти високого загального рівня деталізації з двома відносно невеликими нормальними текстурами карти.

Particle systems

Система частинок - це техніка у фізиці ігор, графіці руху та комп’ютерній графіці, яка використовує спрайти, 3D-моделі чи інші графічні об’єкти для імітації певних видів "нечітких" явищ, які в іншому випадку дуже важко відтворити за допомогою звичайних методів візуалізації - зазвичай сильно хаотичні системи, природні явища або процеси, викликані хімічними реакціями.

Приклади їх використання включають повторення явищ вогню, вибухів, диму, рухомої води (наприклад, водоспад), іскор, падаючого листя , падіння каміння, хмари, туман, сніг, пил, метеорні хвости, зірки та галактики, або абстрактні візуальні ефекти, такі як світлові сліди, магічні заклинання тощо - для цього використовуються частинки, які швидко згасають і потім знову виникають з джерела ефекту. Також техніка може бути використана для речей, які містять багато ниток - таких як хутро, волосся та трава - що включає рендер одразу всього циклу життя цілих частинок за один раз, які потім можуть бути намальовані та маніпульовані як окрема нитка відповідного матеріалу.

Системи частинок можуть бути двовимірними або тривимірними.

Типова реалізація

Зазвичай положення і рух системи частинок в 3D-просторі контролюється тим, що називається випромінювачем (emitter). Випромінювач виступає джерелом частинок, а його розташування в 3D-просторі визначає, де вони утворюються та куди вони рухаються. Звичайний 3D-сітчастий об’єкт, наприклад куб або площина, може використовуватися як випромінювач. Випромінювач додав до нього набір параметрів поведінки частинок. Ці параметри можуть включати швидкість появи (скільки частинок утворюється за одиницю часу), початковий вектор швидкості частинок (напрямок, який

вони випромінюються при створенні), час життя частинок (тривалість часу існування кожної окремої частинки до зникнення), колір частинок та багато іншого. Загально, що всі або більшість цих параметрів є "нечіткими" - замість точного числового значення художник визначає центральне значення та ступінь випадковості, дозволену по обидва боки від центру (тобто середній час життя частинки може становити 50 кадри ± 20%). При використанні сітчастого об'єкта в якості випромінювача початковий вектор швидкості часто встановлюється як нормальний для окремих граней об'єкта, тому частинки, здається, "розпорошуються" безпосередньо з кожної грані, але необов'язково.

Типовий цикл оновлення системи частинок (який виконується для кожного кадру анімації) можна розділити на два різних етапи, етап оновлення / моделювання параметрів та етап візуалізації.

Етап моделювання

Під час етапу моделювання кількість нових частинок, які необхідно створити, обчислюється на основі швидкості появи та інтервалу між оновленнями, і кожна з них породжується у певному положенні в 3D-просторі, виходячи з положення випромінювача та визначеної області появи.

Кожен з параметрів частинки (тобто швидкість, колір тощо) ініціалізується відповідно до параметрів випромінювача. Під час кожного оновлення перевіряються всі наявні частинки, щоб побачити, чи не перевищив їх термін життя: в такому випадку вони вилучаються з моделювання. В іншому випадку положення частинок та інші характеристики висуваються на основі фізичного моделювання, яке може бути таким же простим, як переклад їх поточного положення, або таким же складним, як виконання фізично точних обчислень траєкторії, що враховують зовнішні сили (сила тяжіння, тертя, вітер, тощо).

Не рідко виконується виявлення зіткнень між частинками і конкретизовано 3D об'єктами в сцені, щоб частинки відскакували чи інакше взаємодіяли з перешкодами в навколишньому середовищі. Зіткнення між частинками застосовуються рідко, оскільки вони обчислювально дорогі і не є візуально актуальними для більшості моделей.

Етап візуалізації

Після завершення оновлення кожна частинка виводиться, як правило, у вигляді текстурованого чотирьохбордингового квадрата (тобто чотирикутника, який завжди звернений до глядача). Однак іноді це не потрібно для ігор; частинка може бути виведена як один піксель у невеликих роздільних здатностях / обмеженому середовищі потужності обробки. І навпаки, частинки графіки в русі мають тенденцію бути повноцінними, але дрібномасштабними та легкими для надання 3D-моделями, щоб забезпечити вірність навіть при високій роздільній здатності.

Частинки можуть бути виведені як Metaballs в режимі офлайн-візуалізації; Ізоповерхності, обчислені з частинок-метаболів, утворюють досить переконливі рідини. Нарешті, 3D-сітчасті об’єкти можуть «підмінювати» частинки - снігова буря може складатися з єдиної 3D-сніжинки, яка дублюється та обертається, щоб відповідати позиціям тисяч чи мільйонів частинок.

"Snowflakes" проти "Hair"

Системи частинок можуть бути анімованими або статичними; тобто тривалість життя кожної частинки може бути розподілена за часом або виведені всі відразу. Наслідок цього розрізнення є схожим на різницю між сніжинками та волоссям - анімовані частинки схожі на сніжинки, які рухаються навколо як окремі точки у просторі, а статичні частинки схожі на волосся, що складається з чіткої кількості кривих.

Сам термін "система частинок" часто нагадує лише анімований аспект, який зазвичай використовується для створення імітації рухомих частинок - іскри, дощу, вогню тощо. У цих реалізаціях кожен кадр анімації містить кожну частинку в певній позиції у своєму життєвому циклі, і кожна частинка займає єдину точкову позицію в просторі. Для таких ефектів, як вогонь або дим, які розсіюються, кожній частинці надається час зникнення або фіксований термін експлуатації; такі ефекти, як снігопади або дощ, замість цього зазвичай припиняють термін експлуатації частинки, як тільки вона виходить з певного поля зору.

Однак якщо весь життєвий цикл кожної частинки відображається одночасно, результатом є статичні частинки - нитки матеріалу, які показують загальну траєкторію частинок, а не точкові частинки. Ці пасма можна використовувати для імітації волосся, хутра, трави тощо.

Нитки можна керувати тими ж векторами швидкості, силовими полями, швидкостями появи і параметрами відхилення, яким підкоряються анімовані частинки. Крім того, виведена товщина ниток може контролюватися і в деяких варіантах може змінюватись по довжині пасма. Різні комбінації параметрів можуть надавати жорсткість, м'якість, важкість, щетину або будь-яку кількість інших властивостей. Пасма можуть також використовувати відображення текстури для зміни кольору, довжини та інших властивостей на всій поверхні випромінювача.

Системи часток в Unity

Вбудована система частинок Unity дозволяє створювати ефекти для кожної платформи, яку Unity підтримує. Вбудована система частинок імітує поведінку частинок на процесорі, що дозволяє отримати такі основні переваги:

• Можна використовувати сценарії C # для взаємодії з системою та окремими частинками в ній.

• Системи частинок можуть використовувати базову фізичну систему Unity і, таким чином, взаємодіяти з колайдерами у вашій сцені.

Компонент системи частинок імітує рідкі утворення, такі як рідина, хмари та полум'я, генеруючи та анімуючи велику кількість маленьких двовимірних зображень на сцені. Повне ознайомлення з системами частинок та їх використанням див. У подальшій документації щодо систем частинок.

Динаміка системи

Кожна частка має певний час життя (зазвичай кілька секунд), протягом якого частка може зазнати різні зміни. Частка починає своє існування, коли вона створена, або видана своєю системою частинок. Система випускає частки в випадкових точках в просторі, обмеженому регіоном в формі сфери, півсфери, конусом, прямокутним паралелепіпедом або мешем довільної форми. Частка відображається до тих пір, поки не закінчиться час її життя, потім вона видаляється з системи. Частота випускання частинок системи жорстко визначає кількість частинок, що випускаються в секунду, хоча точний час випускання містить невеликий фактор випадковості. Частота випускання в сукупності із середнім часом життя частинки визначають кількість "стабільних" частинок і час, необхідний системі для досягнення такого стану.

Динаміка частинок

Налаштування випускання і часу життя впливають на загальну поведінку системи, однак окремі частинки теж можуть змінюватися з часом. Кожна частка має вектор швидкості, що визначає напрямок і відстань, яку проходить частинка за один кадр. Швидкість може бути змінена за допомогою сил і гравітації, що застосовуються самою системою або коли

частки здуваються в зонах вітру над Terrain (назва об'єкта в Unity, що представляє земну поверхню). Колір, розмір і обертання кожної частки також можуть змінюватися протягом часу життя або пропорційно її поточної швидкості руху. Колір містить альфа-канал (для прозорості), так що частка може плавно зникати і з'являтися замість різкої зміни видимості в таких випадках.

При використанні в сукупності, динаміка частинок виразно може застосовуватися для симуляції багатьох типів текучих ефектів. Наприклад, водоспад може бути симулювати за допомогою вузької області випускання, з якої падають частки води під дією гравітації, прискорюючись по ходу руху. Дим від вогню зазвичай піднімається вгору, розширюється і в кінці розсіюється, так що система повинна застосовувати висхідну силу до частинкам диму, збільшувати їх в розмірі і збільшувати прозорість з плином їх часу життя.

Властивості

Компонент системи частинок має безліч властивостей, і для зручності Інспектор організовує їх у розбірні розділи під назвою «модулі».

Зміна властивостей з плином часу

Багато числових властивостей частинок або навіть всієї системи можуть змінюватися відносно до часу. Unity надає кілька різних способів вказівки як буде відбуватися зміна:

- Константа: Значення властивості не змінюється з часом.
- Крива: Значення змінюється по кривій / графу.

Random between two constants: Два постійних значення визначають верхню і нижню межі для підсумкового значення, яке вибирається випадково з усіх значень, що потрапляють в ці рамки.

Random between two curves: Дві криві визначають верхню і нижню межі діапазону значень в певний момент часу; поточне значення випадковим чином вибирається з цього діапазону.

Для властивостей кольору, таких як Color over lifetime, є два окремі варіанти:

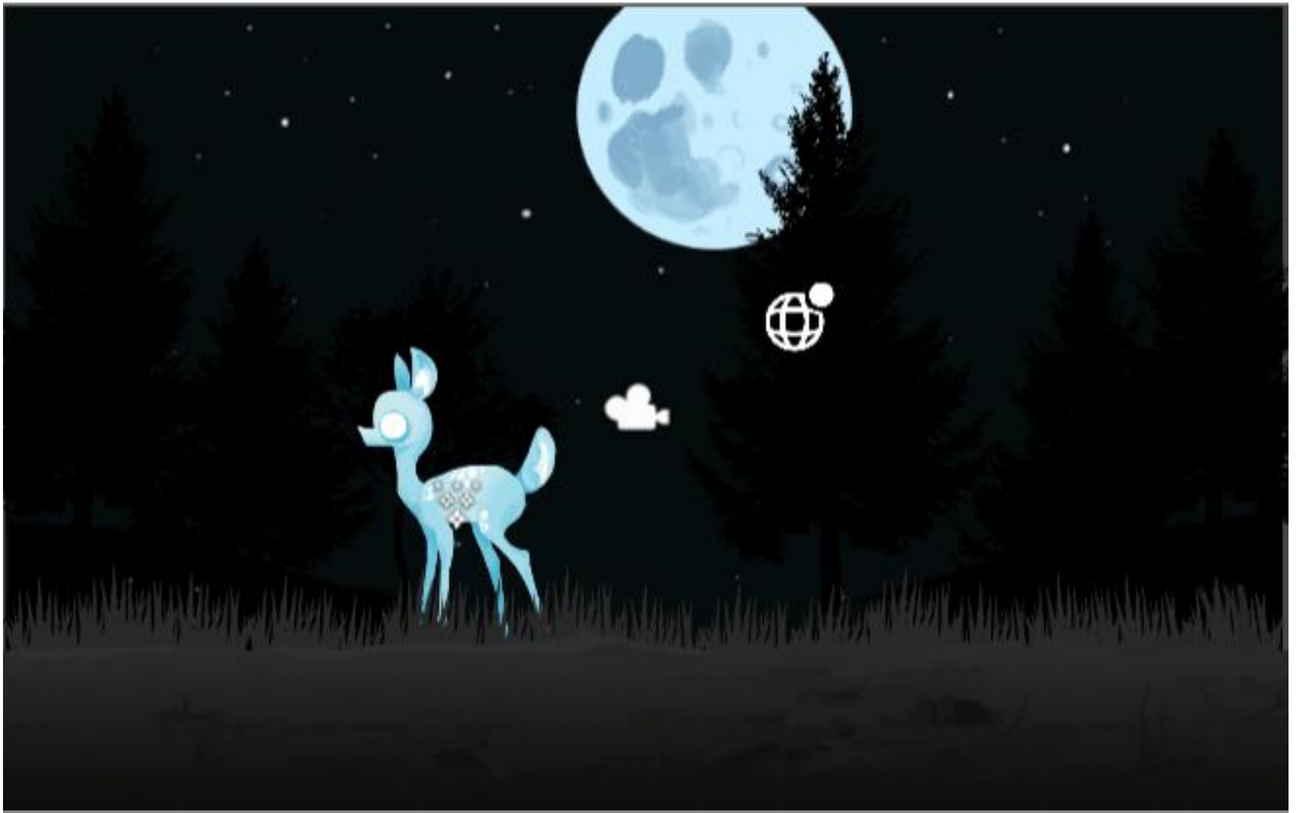
Градієнт: Значення кольору вибирається з градієнта.

Random between two gradients: Два градієнта визначають верхню і нижню межі значення кольору в певний момент часу; поточне значення є випадковим середньозваженими з квітів на кордонах.

Розділ 2

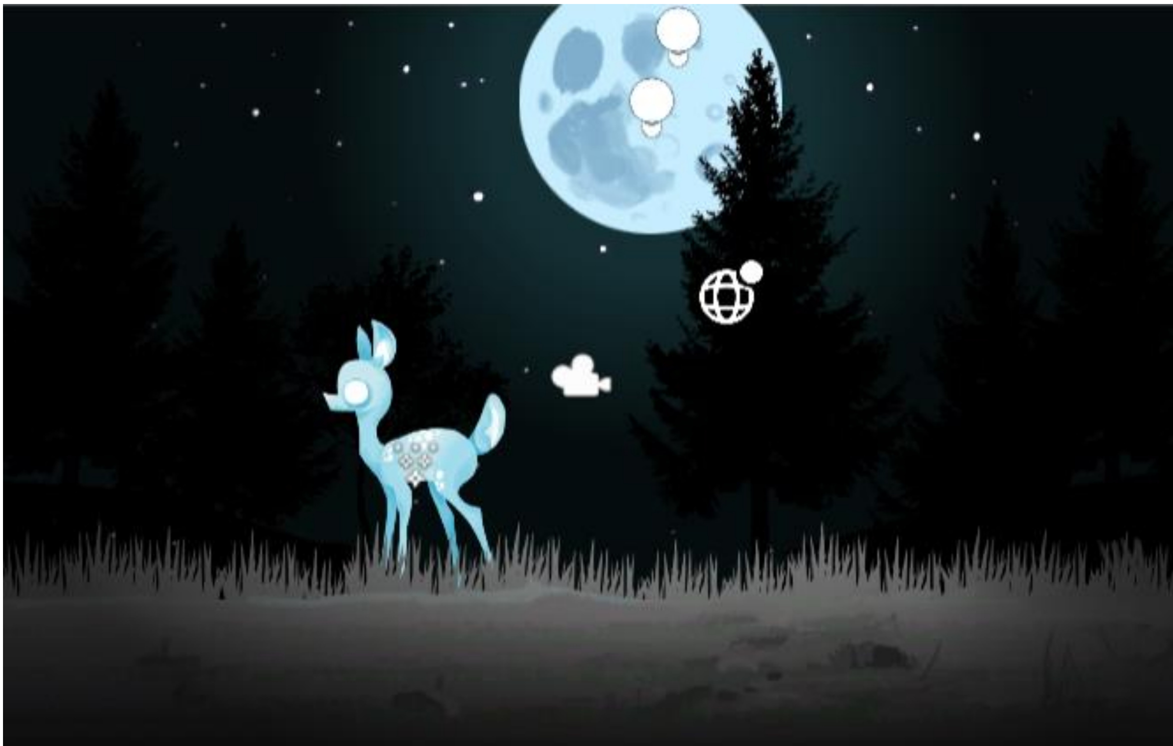
Освітлення в Unity

Освітлення має велику роль для створення гарної та влучної графіки для ігр. Для цього проекту потрібно створити ефекти освітлення в темному лісі.



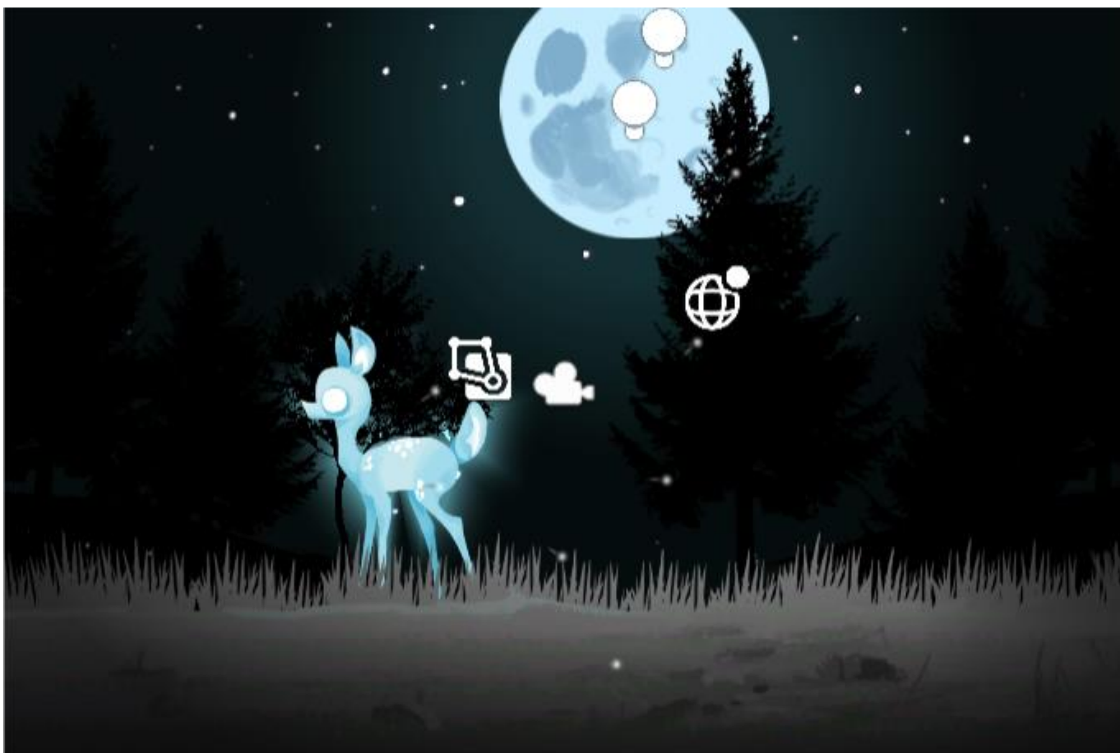
Сцена з Global light

Без джерел освітлення по дефолту сцена залишається повністю чорною. Тому спочатку визначається Global light. Воно дає тьмяне освітлення сцени.



Додавання кількох Point Light

Наступними додаються Point Light. В даній сцені вони створені для імітації освітлення місяця. Їх створено два – один для освітлення неба, інший для освітлення переднього плану. Кожен с джерел освітлення діє лише на обрані шари сцени та ігнорує інші.



Додавання FreeForm Light

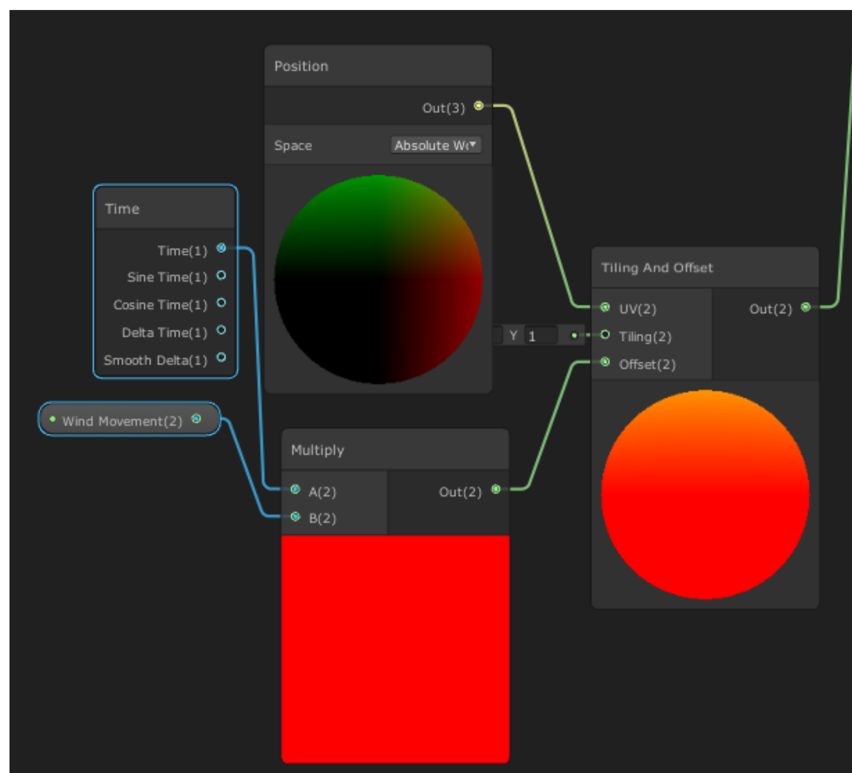
Для створення сяйва навколо оленя використано FreeForm Light. Воно дає змогу для складної форми джерела освітлення, в даному випадку – для симуляції свічення магічного оленя по його контуру.

Шейдер ефекту колихання від вітру на Shader Graph

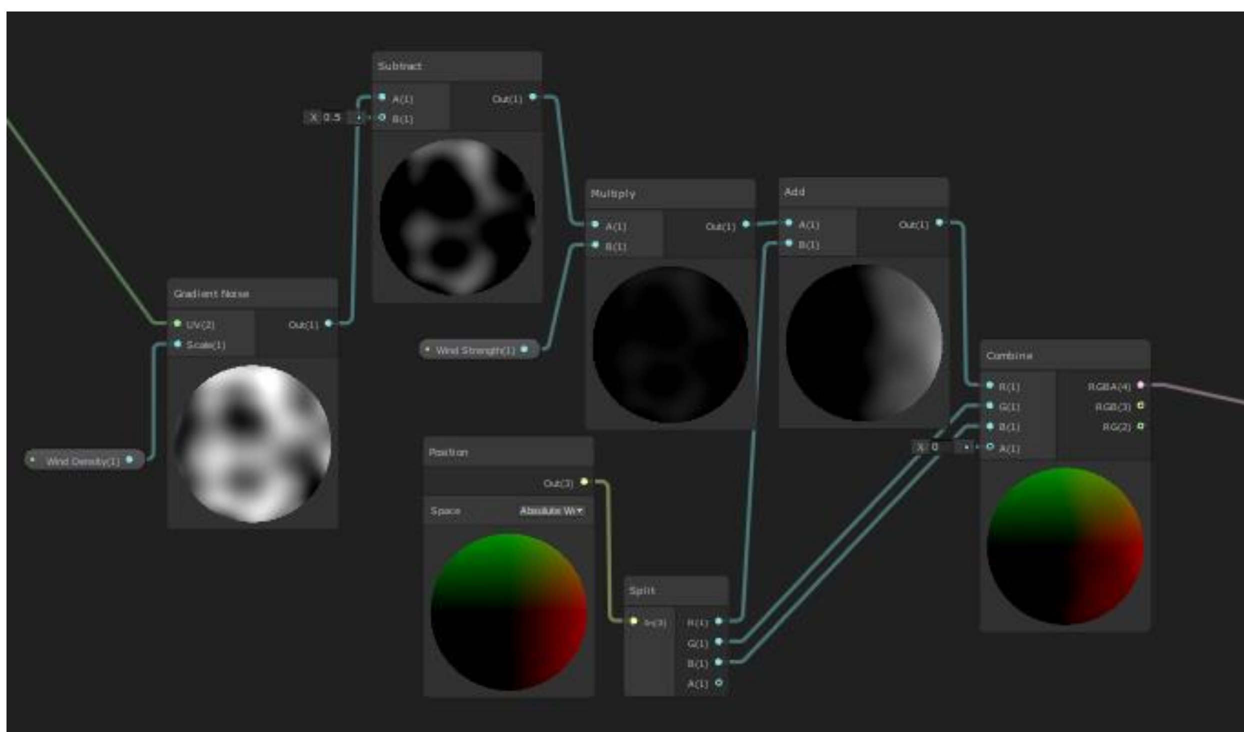
Цей шейдер можна використовувати для ефекту колихання від вітру різних об’єктів у грі. Він може застосовуватись до будь-яких об’єктів, але найкраще виглядає із спрайтами рослинності, такі як трава чи дерева.

Ефект природнього руху створюється за допомогою *шуму* та зсуву точок. Шейдер має такий алгоритм:

- отримуються координати в координатах “світу” відносно центу сцени, їх нормалізовані дані зсуваються з часом



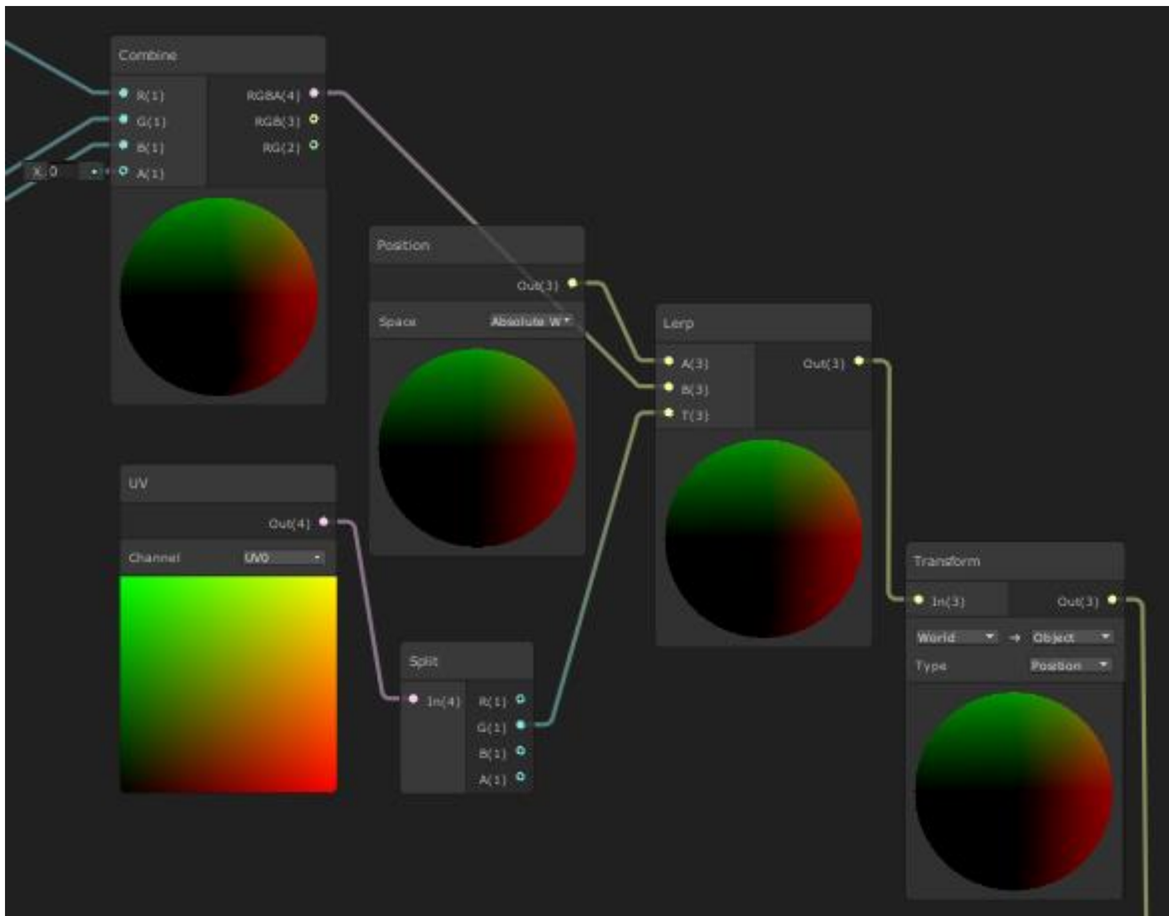
- генерується шум в координатах “світу” відносно центу сцени, на основі нормалізованих координат, який зсувається по осі *x* відносно часу
    - на низьких значеннях (чорний колір) об’єкт зсувається по осі x за напрямком вітру
    - на високих значеннях (білий колір) об’єкт зсувається по осі x проти напрямку вітру



- сила зсуву буде залежати від висоти пікселя: нормалізоване значення координати *y* помножується на зсув; таким чином “корінь” об’єкта не рухається взагалі, а “верхівка” об’єкта зсувається найбільше

Також додані додаткові змінні для більшого контролю ефекту. Це:

- *Wind Movement* – швидкість зсуву по осі *x,* швидкість вітру
- *Wind Density* – масштаб шуму, щільність вітру
- *Wind Strength* – помножується на значення шуму для регулювання сили вітру, тобто як сильно віте впливає на об’єкт

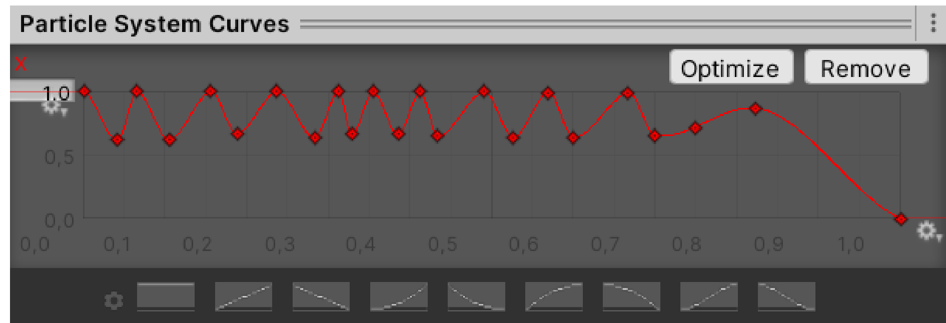


Система часток в Unity

Завдання цього ефекту – створити слід вогників, які злітають з оленя, і, як світляки, розлітаються по лісу.

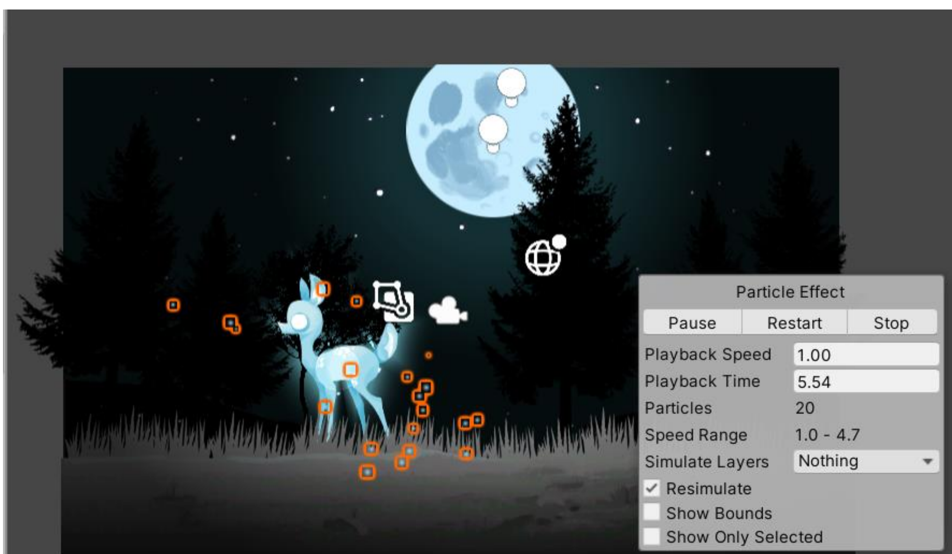
Потрібний ефект завдається за допомогою зміни різних параметрів, таких як:

- Шум:
  - шум надає руху часток природнього виду, ніби хаотичний політ комахи
- Velocity over LifeTime
  - зокрема параметр Orbital velocity, завдяки частки обертаються навколо центра системи
- Limit Velocity over Lifetime
  - допомагає знизити прискорення відносно життя частки; так у даній системі частки сповільнюються до їх повного згасання
- Size over LifeTime

- дозволяє змінювати розмір частки, зокрема використати криву замість числа; такий спосіб дозволяє створити ефект миготіння, а також плавного повного зникнення до кінця життя
- 

Particle System
Emission
Shape
Velocity over Lifetime
Limit Velocity over Lifetime
Inherit Velocity
Force over Lifetime
Color over Lifetime
Color by Speed
Size over Lifetime
Size by Speed
Rotation over Lifetime
Rotation by Speed
External Forces
Noise
Collision
Triggers
Sub Emitters
Texture Sheet Animation
Lights
Trails
Custom Data
Renderer

-



Створення та програмування системи часток

Спочатку потрібно створити клас, щоб зберігати інформацію про місце розташування та напрямок частинки. Оскільки система часток має працювати у 3D-світі, використаємо простий 3D-вектор. Клас Вектор інкапсулює три змінних з плаваючою крапкою, містить функції додавання, віднімання та множення векторів.

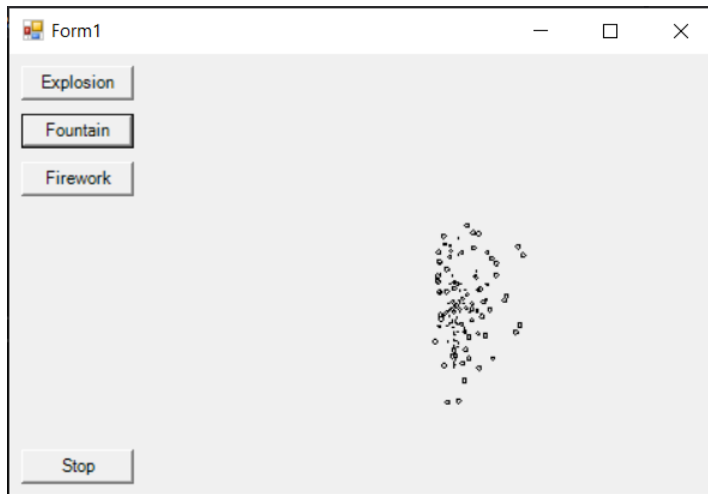
Оскільки Частинка є лише маленькою сутністю, на неї впливають зовнішні сили, такі як сила тяжіння та вітер. Ці сили будуть реалізовані в класі Environment.

Середовище включає всі зовнішні сили, які впливатимуть на всі частинки у різних системах. Такі сили включають тривіальну гравітацію і вітер, але можуть також включати такі сили, як температура або будь-яка інша ідея. Оскільки ми хочемо лише один екземпляр для середовища, доцільно реалізувати це як Singleton.

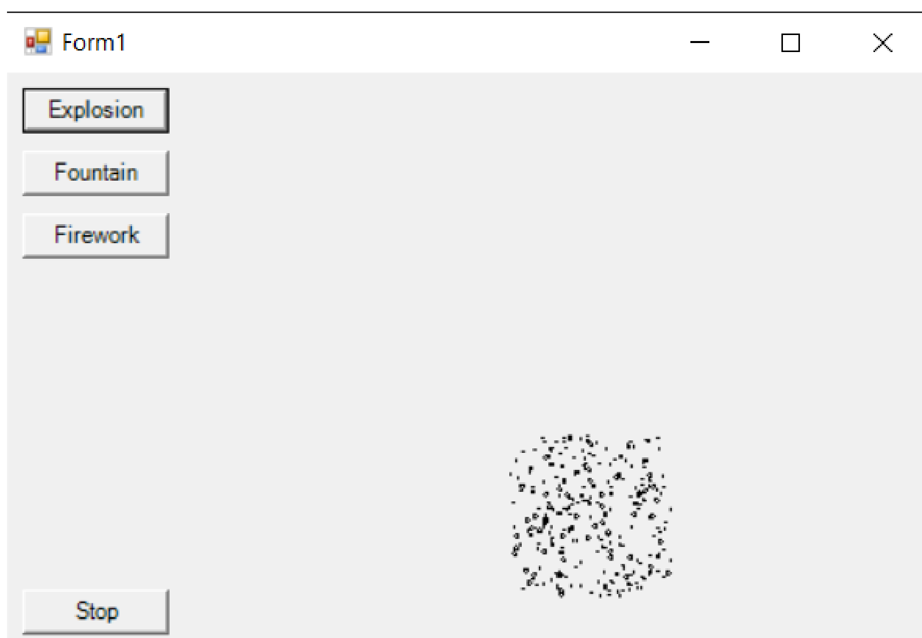
Наступним було створемо базовий клас для системи - ParticleSystem. Цей клас, який насправді є абстрактним класом, буде обробляти список частинок і вимагатиме від кожного класу, який успадковує його, для реалізації функції створення нових частинок та функції оновлення цих частинок. Функція GenerateParticle () буде використовуватися, коли створюється нова частинка, будь то абсолютно нова частинка, або коли частинка відмирає, і ми хочемо замінити її новою. Update () буде використовуватися для оновлення частинок у системі. Update () повинен вирішити, чи і коли створювати нові частинки. І останнє, Draw () буде використовуватися для відображення системи частинок на заданому об’єкті Graphics.

Наступними будуть створені системи часток, які імплементують абстрактний клас. Це будуть ефекти вибуху, фонтану та салюту. При вибуху частинки просто летять скрізь. Це нескладно здійснити - просто потрібно встановити всі частинки, щоб вони починалися в центрі системи, і рухалися у

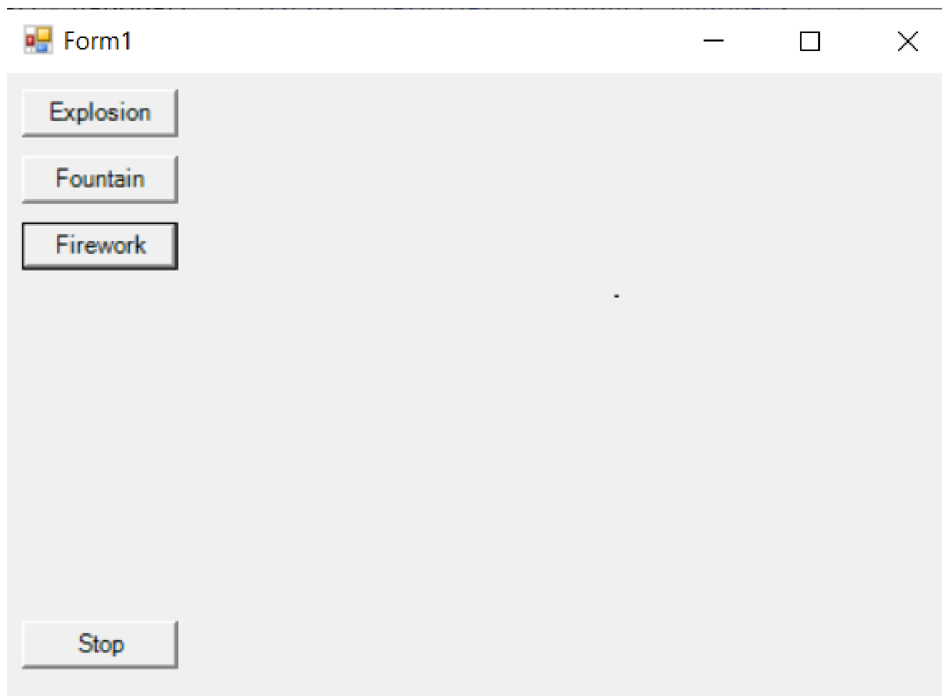
випадковому напрямку зі випадковою швидкістю. Фонтан відрізняється не сильно. По-перше, фонтан регенерує частинки, які гинуть, щоб продовжувати ефект. По-друге, не всі частинки створюються одразу - спочатку створюються кілька частинок, а з часом йде все більше і більше частинок до системи.

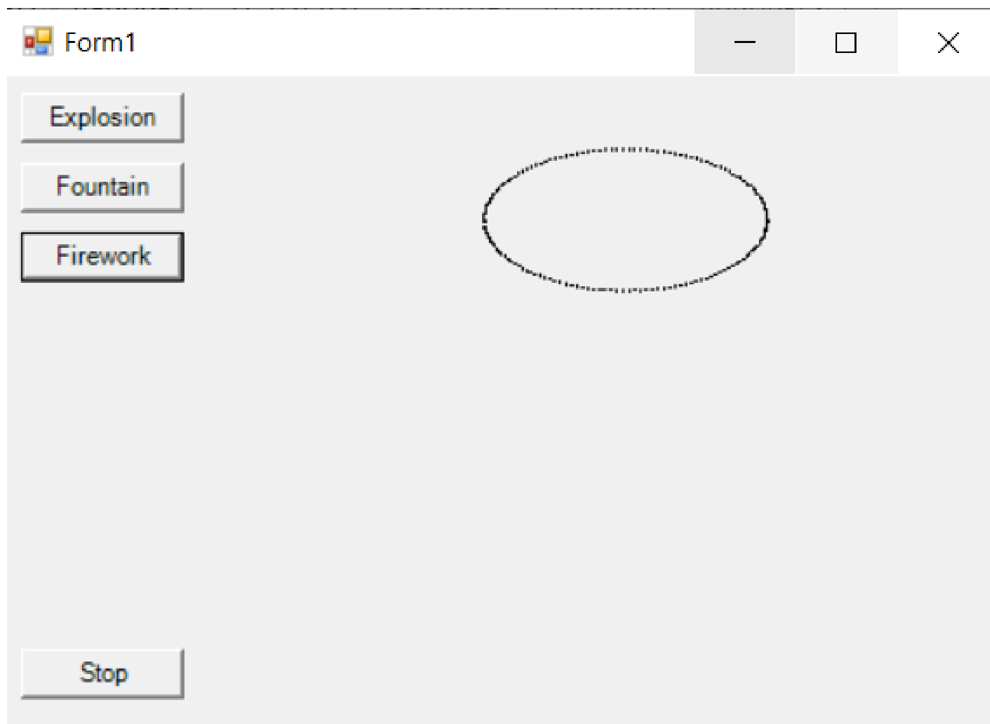


Реалізація фонтану



Реалізація взриву





Реалізація салюту (фаза 1 – ракета, фаза 2 - салют)

Висновки.

Нині існує багато засобів для розробки ефектів. Unity є чудовим інструментом, оскільки він надає багато засобів для використання вже готових рішень, а також для створення нових з нуля.

Реалізовані ефекти показують як різними засобами можна покращити графіку для 2Д ігор.

Цінність цієї роботи – це розгляд основ теорії та засобів для покращення графіки для подальшого ускладнення розроблених ефектів. Використання такими засобами значно покращує якість застосувань (особливо відеоігор), тому володіння навичками розробки їх досить важливе.

Список використаних джерел

1. Sebastien St-Laurent. Shaders for Game Programmers and Artists. - Thomson Course Technology PTR, 2004.

2. Mike Bailey, Steve Cunningham. Graphic Shaders. - Taylor & Francis Group, LLC, 2012.

3. Unity User Manual (2019.3)[Електронний ресурс] – Режим доступу: https://docs.unity3d.com/Manual/index.html.

4. Patricio Gonzalez Vivo, Jen Lowe. Book of Shaders. [Електронний ресурс] – Режим доступу: https://thebookofshaders.com/

Додаток А

Приклади програмного коду виконаних ефектів

```
using System;

using System.Drawing;

namespace Particles
{
	public class Particle
	{
		private Vector coords;
	//velocity
		private Vector v;
		//lifespan of the particle
		private int lifespan;
		private Color clr;


		public Particle() : this(Vector.Zero, Vector.Zero, Color.Aqua, 0)
		{
		}

		public Particle(Vector coords, Vector v, Color clr, int ls)
		{
			this.coords = coords;
			this.v = v;
			this.clr = clr;
			// valid life
			if (ls > 0)
				this.lifespan = ls;
```

```
		}

		public virtual void UpdateP()
		{
			v = v-
Environment.GetInst().GravityValue+Environment.GetInst().WindValue;
	lifespan++;
	coords = coords + v;
		}
		#region Accesors

		public Vector Coords
		{
			get { return coords; }
		}
		public Vector Vel
		{
			get { return v; }
		}
		public int Life
		{
			get { return lifespan; }
		}
		public Color Color
		{
			get { return clr; }
		}
		#endregion Accessors
	}
}
```

```
using System;


namespace Particles
{
    public class Environment
    {
        // instance of  environment
        private static Environment inst = new Environment();

        private Vector gravityval = Vector.Zero;
        private Vector windval = Vector.Zero;

        protected Environment()
        {
        }

        public static Environment GetInst()
        {
            return inst;
        }
      //gravity getter and setter
      public Vector GravityValue
        {
            get { return gravityval; }
            set { gravityval = value; }
        }
      //wind getter and setter
        public Vector WindValue
        {
```

```
				get { return windval; }
				set { windval = value; }
			}
		}
}

using System;
using System.Collections;
using System.Drawing;


namespace Particles
{
	public abstract class ParticleSystem
	{
		protected ArrayList partics = new ArrayList();
		protected bool regen;
		protected Vector coords;
		protected int maxlifespan = 150;
		protected Color clr;



		//generate one particle

		protected abstract Particle GenParticles();

		public abstract bool Update();

		//draw all  particles
		public virtual void DrawParticles(Graphics graphics)
```

```
		{
			Pen p;
			int intensity;
			Particle particle;

			for (int i = 0; i < partics.Count; i++)
			{
				particle = this[i];
				//count  intensity
				intensity = (int)((float)particle.Life / maxLife);
				//gen pen
				p = new
Pen(Color.FromArgb(intensity*clr.R,intensity*clr.G,intensity*clr.B));
				//draw
				graphics.DrawEllipse(p, particle.Coords.XVal,
particle.Coords.YVal,Math.Max(1,4 * particle.Life / maxLife),Math.Max(1,4 *
particle.Life / maxLife));
				p.Dispose();
			}
		}




		//for acces
		public Particle this[int index]
		{
			get
			{
				return (Particle)partics[index];
			}
		}
```

```
		public int count
		{
			get { return partics.Count; }
		}

		public virtual int maxLife
		{
			get { return maxlifespan; }
		}

	}
}

using System;
using System.Drawing;

//100 but check
namespace Particles
{
	public class FireworkSys : ParticleSystem
	{
		private static readonly int defaultnum = 150;
		protected new int particleMaxLife = 80;

		private Random random = new Random();
		private bool rocketstage = true;

		public FireworkSys() : this(Vector.Zero, Color.Black){}
		public FireworkSys(Vector pos) : this(pos, Color.Black){}
```

```
		public FireworkSys(Vector coords, Color clr)
		{
			base.coords = coords;
			base.clr = clr;
			InitSystem();
		}

		private void InitSystem()
		{
			float x = 1 * ((float)random.NextDouble() - 0.5f);
			float y = -2f - 1 * ((float)random.NextDouble());
			float z = 1 * ((float)random.NextDouble() - 0.5f);

			Particle part = new Particle(coords,new Vector(x, y, z),
clr,random.Next(50));

			partics.Add(part);
		}

		public override bool Update()
		{
			Particle p;
			int num = partics.Count;

			if (rocketstage)
			{
				p = (Particle)partics[0];
				p.UpdateP();
				if (p.Life > particleMaxLife)
				{
```

```
                        rocketstage = !rocketstage;
                        for (int i = 0; i < defaultnum; i++)
                        {
                            Particle newPart = new Particle(p.Coords,
                                new Vector((float)Math.Cos(Math.PI
* 2 * i/defaultnum),((float)Math.Sin(Math.PI * 2 * i/defaultnum)/2)-0.75f, 0), clr,
0);
                            partics.Add(newPart);
                        }
                        // Remove "rocket" particle
                        partics.RemoveAt(0);
                    }
                    return true;
                }

                // in firework stage

                for (int i=0; i<num; i++)
                {
                    p = (Particle)partics[i];
                    p.UpdateP();
                    if (p.Life > particleMaxLife)
                    {
                        partics.RemoveAt(i);
                        i--;
                        num = partics.Count;
                    }
                }
                if (partics.Count <= 0)
                    return false;
                return true;
```

```
        }

        protected override Particle GenParticles()
        {
            return null;
        }
    }
}i
```