Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій

Кафедра мережевих та інтернет технологій

ЗАТВЕРДЖУЮ
завідувач кафедри
мережевих та інтернет технологій
_____ Юрій КРАВЧЕНКО
«___»_____2023 року

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

галузі знань 17 «Електроніка та телекомунікації»
за спеціальністю 172 «Телекомунікації та радіотехніка»
освітньо-професійна програма «Мережеві та інтернет технології»
на тему:

РОЗРОБКА ДОДАТКУ З АЛГОРИТМОМ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ ДЛЯ РОЗМІЩЕННЯ НА ВЕБ-ПЛАТФОРМІ

Виконав: студент групи МІТ-41

Іван ШИЛО _____

Керівник: завідувач кафедри мережевих та інтернет технологій

д. т.н., професор Юрій КРАВЧЕНКО _____

Київ 2023

Міністерство освіти і науки України
«Київський національний університет імені Тараса Шевченка»

Факультет інформаційних технологій
Кафедра мережевих та інтернет технологій

ЗАТВЕРДЖУЮ
завідувач кафедри
мережевих та інтернет технологій
_____ Юрій КРАВЧЕНКО

«_____»_____2023 року

ЗАВДАННЯ
НА ДИПЛОМНУ РОБОТУ

Здобувачу вищої освіти Шилу Івану Костянтиновичу
(прізвище, ім'я, по батькові)

1. Тема роботи:
Розробка додатку з алгоритмом процедурної генерації для розміщення на веб-платформі

затверджена на засіданні кафедри МІТ «07» грудня 2022 р. протокол №5
2. Термін здачі закінченої роботи «31» травня 2023 р.
3. Вихідні дані до проекту (роботи)

Сучасні технології створення додатків

4. Зміст пояснювальної записки (перелік питань, що їх потрібно розробити, обсяг – 35-45 стор.)

Вступ
1. Теоретичне обґрунтування та огляд засобів
2. Аналіз та постановка задач
3.Розробка додатку
Висновки

5. Перелік графічного матеріалу 8-12 слайдів

Дата видачі завдання _____
Керівник роботи _____ Ю.В Кравченко
(підпис) (посада, прізвище, ім'я, по батькові)

Завдання прийняв до виконання
_____ І.К. Шило
(підпис)

КАЛЕНДАРНИЙ ПЛАН ВИКОНАННЯ РОБОТИ

Номер	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Підготовчий	20.01.2023	
2	Розділ 1	15.02.2023	
3	Розділ 2	15.03.2023	
4	Розділ 3	15.04.2023	
5	Доповідь та слайди	25.05.2023	
6	Пояснювальна записка	31.05.2023	

Здобувач вищої освіти _____ Іван ШИЛО
(підпис)

Керівник _____ Юрій КРАВЧЕНКО
(підпис)

ЗМІСТ

ВСТУП

Метою даної бакалаврської кваліфікаційної роботи є розробка програмного додатку, що виконуватиме процедурну генерацію карти з багатьох пов’язаних шляхами між собою локацій і буде доступним інструментом за рахунок розміщення на виділеній для нього сторінці в мережі Інтернет. Результатом роботи буде програмний код, що створюватиме відображення виводу алгоритму процедурної генерації ігрової мапи; додаток на його основі розміщено на веб-платформі для подальшого розповсюдження. Розроблений програмний код в подальшому може бути використаний як елемент генерації мапи ігрового світу або довершений в якості повноцінної цифрової версії настільної гри.

Методи процедурної генерації є особливо широковживаними у галузі розробки ігор. Зазвичай їх протиставлять ручному способу створення контенту, тобто вони використовується для автоматичного створення ігрових елементів без прямої участі художників, дизайнерів, аніматорів. В першу чергу це дозволяє зменшити об’єми ресурсів, які виділяються на створення тих самих ігрових елементів вручну та привнести різноманітності в гру, таким чином збільшити її перегравність [Replay value]. Результатами такої процедурної генерації можуть бути різні ігрові предмети, такі як спорядження та артефакти, неігрові персонажі ¬– вороги, послідовники, завдання та навіть ігрові рівні. Окрім того, зменшуються об’єми кінцевого продукту-гри: замість того, щоб зберігати всі однотипні об’єкти, як наприклад ігрові рівні, зберігаються елементи з яких за певним алгоритмом генерації ці рівні створюються під час гри.

В наш час, коли особливо сильно відчутним стає вклад глобальної мережі у подолання різноманітних перешкод спілкуванню людей, частиною нашого повсякденного життя стали різноманітні цифрові продукти для ведення бізнесу, організації діяльності та власне самого спілкування. Тож цифрові версії традиційних настільних ігор це послідовне продовження для

подальшого їхнього існування за нових часів. Шахи були однією з перших ігор, які стало можливим грати за комп’ютером. Нині власні електронні версії та цифрові доповнення мають такі відомі ігри як Монополія (Monopoly), Ризик (Risk), Поселенці Катана (Settlers of Catan). Остання навіть має власний мобільний додаток для генерації стартового ігрового поля, що і є програмною реалізацією процедурної генерації, яка використовується для гри. Особливо сильно автоматизації процесів потребують ігри, де створення ігрових сценаріїв покладається на окремого гравця, тобто ведучого гри. Тому для настільних ігор будь-яких жанрів, що так чи інакше потребують різноманіття ігрового досвіду будуть потрібні відповідні набори інструментів, до яких має бути зручний та швидкий доступ. Це забезпечить неповторність ігрового досвіду, а отже захопить увагу гравців на якомога довший час.

РОЗДІЛ 1. ТЕОРЕТИЧНЕ ОБҐРУНТУВАННЯ ТА ОГЛЯД ЗАСОБІВ

1.1. Поняття процедурної генерації

Процедурна генерація – це узагальнена назва для методів автоматичного створення наборів даних за допомогою алгоритмів, що включають в себе елемент випадковості. В даному терміні слово “процедура” позначає послідовне виконання певних дій, до яких входить обчислення певної математичної функції, в результаті яких буде згенеровано, тобто створено новий набір даних. Отримані в результаті виконання задачі процедурної генерації набори даних можуть бути визначені як процедурно створені або процедурно згенеровані.

Програма процедурної генерації приймає в якості вхідних даних як детерміновані параметри, так і випадкові початкові значення. Елемент випадковості в процедурній генерації забезпечує неповторність типових об’єктів, створення яких є задачею однієї програми процедурної генерації. Завчасно визначеними елементами можуть бути компонентами, котрі були вручну створені до початку виконання програми процедурної генерації.

1.2. Походження використання процедурної генерації в іграх

Використання процедурної генерації в іграх веде свій початок з настільних рольових ігор. Система Advanced Dungeons & Dragons забезпечувала майстра гри способами генерувати підземелля та місцевість за допомогою кидків гральних костей, за отриманими значеннями яких в наступних виданнях гри вже проводиться вибірка з таблиць, що містять у собі шаблони кімнат, монстрів і подібного. Компанія Strategic Simulations за ліцензією тодішнього правовласника D&D Tactical Studies Rules випустила Dungeon Master's Assistant, комп’ютерну програму, яка генерувала підземелля на основі тих самих таблиць. Подібні методи використовуються і для сучасних настільних ігор та доповнень до них; існують програмні застосунки

та веб-сервіси які задовольняють потребу у автоматизованій генерації ігрових локацій шляхом процедурної генерації. Ці засоби мають різну ступінь пропрацьованості і генерують сценарії на різну тематику. Витрати часу на підготовку гри та різноманітність наповнення сюжету напряму залежать від обраного інструментарію майстра гри, тому чим більш широким буде асортимент засобів, тим менше підготовка забиратиме часу.

Однією з перших комп'ютерних ігор, що використовувала процедурну генерацію аналогічно тому, як це використовувалося в настільних іграх була опублікована в 1980 році гра Rogue (англ. «Пройдисвіт»). В ній гравець мав змогу взяти на себе роль спритника, котрий досліджує підземелля з кімнат та коридорів та бореться з істотами, що їх населяють. Типовий рівень цієї гри представляв підземелля, котре складалося з процедурно створених кімнат, з'єднаних між собою коридорами[рисунок 1.2]. Кожна кімната заповнювалася предметами, такими як артефакти, золоті монети, ключі та іншими скарбами, а також супротивниками, котрі їх охороняли. Ця гра заснувала жанр ігор, де процедурна генерація рівнів є складовою частиною ігрового циклу [gameplay loop – a cycle of actions that you repeat over and over within a game] і в честь неї жанр отримав свою назву Rougelike – «rogue-подібні». Нині цей жанр налічує сотні відео-ігор різної тематики.

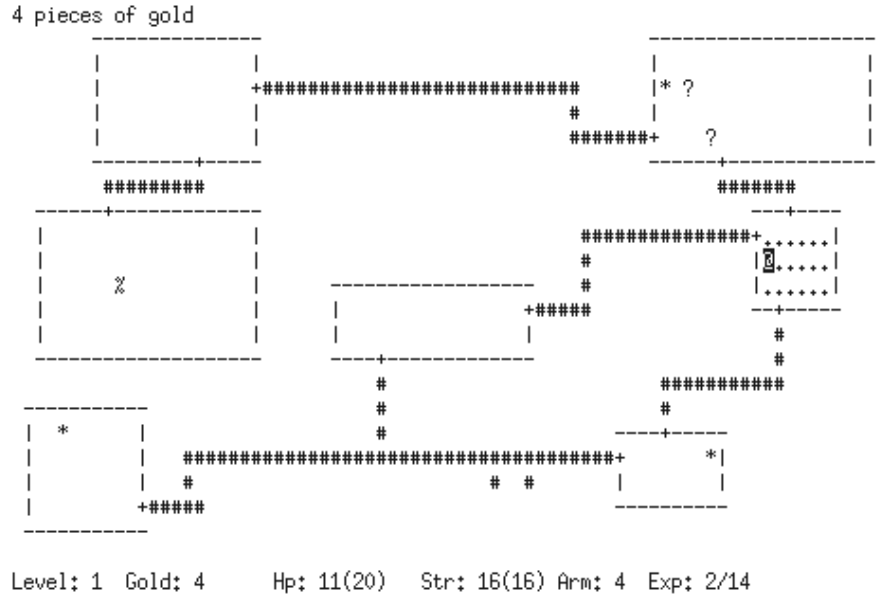

Рисунок 1.2 – вид гри Rogue, колір екрану інвертовано для зручності

1.3. Характерні приклади використання процедурної генерації

Процедурна генерація дуже широко використовується в галузі розробки відео-ігор. Окрім вже згаданої у вступі генерації ігрових рівнів, засобами процедурної генерації можуть бути створені інші ігрові одиниці такі як предмети екіпіровки, неігрові персонажі, завдання. Перелічені ігрові елементи всі входять до категорії таких, з якими гравець має змогу активно взаємодіяти. Але навіть якщо всі такі об’єктами створюються розробником вручну через умови жанру до якого належить гра (скажімо, гра жанру шутер, в якій зброя мусить мати постійні характеристики), засоби, що впроваджують процедурну генерацію, можуть бути використані для естетичних цілей: створення неоднакових хмар, покриття об’єктів неоднорідними текстурами та подібне слугує для створення відчуття реалістичності ігрового світу. Далі наведено приклади використання процедурної генерації, що відображають ці та інші способи використання методів процедурної генерації.

No Man's Sky (англ. «безлюдне небо» або «нічийне небо», дослівно «небо без людей») – гра про дослідження процедурно створеного всесвіту з мільярдами планет, кожна з яких має власні екосистеми, флору, фауну та ресурси, які також визначаються алгоритмами процедурної генерації. Розробка гри супроводжувалася довготривалими дискусіями та жорсткою критикою від гравців через недотримання розробниками обіцянок та врешті-решт на момент випуску гра знайшла свою аудиторію. Незадоволеність гравців виникала зокрема через ефект «процедурної вівсянки». Термін вигаданий письменницею Кейт Комптон передає позначає таку проблему: хоча можливо математично сформувати тисячі мисок вівсяних пластівців із допомогою процедурного генерування, користувач сприйматиме їх як однакові, бо їм бракує унікальності. Але гра, що створена вручну командою людей теж може страждати від одноманітності: кількість цілей які може створити розробник вручну обмежена його власними можливостями часу та фінансуванням проекту. Використання процедурної генерації вирішуватиме

цю проблему різноманітністю створюваного контенту, який відрізнятиметься при кожному запуску гри, при взаємодії з яким гравець не відчуватиме одноманітності.

Цікавим прикладом переваг використання ПГ може слугувати незвичайний шутер .kkriger. Ця гра у 2004 році посіла перше місце на конкурсі ігор розміром до 96 кб. Гра використовує лише 97,280 байтів дискового простору. За словами розробників, .kkrieger мав би займати 200-300 Мб місця на диску, якби він зберігався стандартним способом. Такий ступінь стиснення досягається завдяки тому, що всі ігрові ресурси — текстури, моделі, геометрія рівня, музика і звук — генеруються «на льоту» з допомогою різних алгоритмів ПГ, зокрема музика і звуки генеруються багатофункціональним синтезатором. .kkriger демонструє незвичайне використання ПГ для подолання технічних обмежень.

На відміну від попереднього прикладу, Dwarf Fortress (англ. «Фортеця гномів») — це складна та детальна гра-симулятор, яка випробовує розрахункові потужності комп’ютера гравця. Гра здатна процедурно генерувати цілий фантастичний світ зі своїми цивілізаціями, двохсот років його історії, географією, біомами та деталізованими підземними печерами. Алгоритм створення світу гри кожного разу створює новий багатий і унікальний світ, в якому гравець може привести обрану цивілізацію гномів до величі або до кошмарної загибелі. Процедурна генерація гри створює кожного окремого гнома в контрольованому гравцем загоні, так само як і їхніх супротивників, хоча з меншою кількістю деталей та характеристик. Враховуючи ступінь пропрацьованості цієї гри, вона демонструє здатність ПГ до моделювання реальності: генератори відтворюють життєві умови та можуть бути особливо ефективними в умовах розробки деталей, що симулюють безладдя реального життя.

Отже, використання методів процедурної генерації може забезпечити:

– Різноманітність;

- Відчуття реалістичності;
- Подолання технічних обмежень;
- Економія часу та ресурсів;

Варто зазначити, що при неправильному впровадженні процедурної генерації, гра отримає дзеркально протилежні до цих переваг недоліки. Занадто мала різниця між результатами призведе до процедурної вівсянки; занадто багато ускладнень алгоритму процедурної генерації вимагатиме більш потужного розрахункового обладнання та часу на розробку. Та все ж, при недосконалій реалізації подібні проблеми матиме будь-яка програма.

1.4. Огляд ігрового рушія Unity

Ігровий рушій (англ. Game engine) – центральна програмна частина будь-якої відеогри, що відповідає за її технічну сторону, дозволяє полегшити розробку гри шляхом уніфікації та систематизації її внутрішньої структури. Важливим значенням рушія є можливість створення ігор, в які можна грати на різних платформах. Unity – це інструмент для розробки та ігровий рушій, що дозволяє створювати відеоігри та застосунки, що можуть запускатися на платформах Windows, Mac, iOS, Android, WebGL, PlayStation та багатьох інших. Unity надає засоби для розробки ігор у форматах 2D, 3D та навіть VR. В основі розробки на Unity лежить принцип композиції, тобто об’єкти створюються поєднанням готових компонентів, що надають об’єкту нових властивостей та функціональності. Основною організаційною одиницею побудови об’єктів в Unity є GameObject (“ігровий об’єкт”), до якого можна доєднати програмні скрипти, відображення тексту, зображення, взаємодію з освітленням та інші готові компоненти. Щоб задати правила взаємодії та контролю ігрових об’єктів використовуються скрипти на мові програмування C#. Таким чином Unity дозволяє створювати комплексні об’єкти, для кожного елементів яких можна задати параметри та налаштувати взаємодію один з іншими компонентами.

1.5. Огляд та обґрунтування використання платформи itch.io

itch.io - це онлайн-платформа для дистрибуції, продажу і відтворення комп'ютерних ігор, а також інших видів цифрового контенту. Вона створена з метою надання можливості незалежним розробникам ігор і творчим людям виставляти на продаж свої твори безпосередньо споживачам. Як продавець ви самі відповідаєте за те, як це робиться: ви одноосібно встановлюєте ціну, самостійно керуєте продажами та можете розробляти веб-сторінки власних продуктів. Щоб отримати схвалення вашого вмісту, ніколи не обов’язково отримувати голоси, оцінки «подобається» або підписки, і ви можете змінювати спосіб поширення своєї роботи так часто, як забажаєте.

itch.io також є колекцією деяких найбільш унікальних, цікавих і незалежних творінь, які ви знайдете в Інтернеті. На itch.io можна знайти широкий спектр ігор і інших проектів, включаючи інді-ігри, експериментальні твори, демо-версії, хобі-проекти, арт-інсталяції, музику, комікси та інше.

itch.io зручний для пошуку, покупки і завантаження ігор. Користувачі можуть стежити за улюбленими розробниками, додавати ігри до списку бажань, а також залишати відгуки і оцінки для продуктів, що допомагає в розповсюдженні та рекомендаціях.

Окрім того, itch.io також використовується для проведення гейм-джемів (ігрових змагань) та інших заходів, спрямованих на сприяння творчості в галузі розробки ігор. Платформа стала популярною серед розробників ігор, які шукають можливість самостійно публікувати та продавати свої твори, не залежно від великих видавництв.

Одним з ключових можливостей, які надає дана онлайн-платформа це хостинг ігор. Ігри, що розроблені за допомогою таких технологій як Flash, Unity, Java, HTML5 можуть бути запущені на сторінці проекту як браузерна гра, що буде використаним в рамках цього проекту для розміщення створеного додатку. Таким чином будуть одночасно вирішені питання

розповсюдження додатку та розміщення його для прямої взаємодії через мережу Інтернет.

Висновки до розділу 1

1. Процедурна генерація – це методи створення наборів даних шляхом алгоритмів, що спрощує або повністю виключає ручний спосіб створення цих даних.

2. Процедурна генерація існує давно і використовувалася в іграх до того, як її методи були взяті на користування при розробці комп’ютерних програм.

3. Процедурна генерація здатна забезпечить різноманітність результатів, в окремих випадках зменшити об’єми не тільки роботи під час розробки, але й обійти обмеження обладнання на яких працює програма. Але як і будь-який програмний продукт, розробки з процедурною генерацією можуть мати певні недоліки і складнощі при реалізації.

4. Unity – потужний ігровий рушій, котрий надає все необхідне для створення програмних застосунків, і що може бути використаний для розробки проектів різних масштабів.

5. Itch.io – онлайн-платформа, що підтримує хостинг та відтворення ігор та інструментів певного типу, тому може бути використана для розповсюдження робіт практикуючих розробників.

РОЗДІЛ 2. АНАЛІЗ ТА ПОСТАНОВКА ЗАДАЧ

2.1. Алгоритм процедурної генерації

Основна ціль проекту – це створення програмного додатку, що виконуватиме і демонструватиме роботу алгоритму процедурної генерації, і може бути розміщений на веб-платформі для загального користування всіма бажаючими. Ця кваліфікаційна робота має під собою мету провести дослідження використання та продемонструвати впровадження алгоритму процедурної генерації. Відповідно, перед початком роботи необхідно мати певний алгоритм процедурної генерації і для досягнення мети найкращим кандидатом буде алгоритм, котрий по-перше, може бути оцінений як відносно простий для впровадження, а по-друге, буде достатньо цікавим для користувачів. Для задоволення необхідності в готовому алгоритмі процедурної генерації, було вирішено звернутися до класичного використання процедурної генерації – до настільний ігор. Скориставшись раніше згаданою платформою itch.io, було обрано гру, яка мала простий, але достатньо цікавий набір таблиць процедурної генерації.

2.2. Аналіз першоджерела обраного алгоритму

A Traveler’s Guide to the Echelon Forest (далі згадується як Echelon Forest) – системно-агностична настільна гра про подорож загадковим лісом. Розроблена Девідом Ломбардо для гейм-джему Cairn: Forests of Another Name та опублікована 13 квітня 2022 року, Echelon Forest надає простий генератор лісу, що налічує 36 різних точок інтересу, і базується на використанні звичайних грального кісток. Приставка “системно-агностична” має під собою ідею, що ця гра та засоби в ній будуть використовуватися в парі або як доповнення до іншої настільної гри, де будуть доречними запропоновані інструменти. Окрім власне генератора, гра містить також процедури для подорожі лісом, відслідковування часу в грі, дуже простий лічильник для зміни погодних умов та ще додаткові деталі. Гру можна

придбати на її сторінці на платформі itch.io за 5 доларів або за бажанням більше; в той же час на тій же сторінці автор розміщує громадські копії гри за кожну куплену копію, тому для ознайомлення повну версію гри можна отримати навіть безкоштовно. Розповсюдження гри та матеріалів до неї здійснюється за ліцензією CC-BY-4.0 [посилання перенести в список посилань: https://creativecommons.org/licenses/by/4.0/], яка дозволяє використання та адаптацію тексту гри, допоки вказані посилання на оригінал, автора, цю ліцензію, вказані автор та внесені зміни. Має сенс вказати, що завдяки цьому є можливим також створення повної цифрової копії цієї гри і розширення її новими локаціями та механіками, хоча використання гри в комерційних цілях вірогідно необхідно погодити з автором – буде необхідним розгляд юридичного аспекту цього питання.

2.3 Аналіз та ізоляція алгоритму процедурної генерації

Беручи за основу інструкції по стартовій генерації для гри Echelon Forest, англійський текст якої поданий у додатку А.1 [Текст А.1], алгоритм генерації карти матиме наступне текстове формулювання:

1. Кинути на чистий лист паперу щонайменше 8 гральних кісток – вони позначають точки інтересу.
2. З’єднати позиції на яких знаходяться кості лініями, що позначатимуть шляхи між точками інтересу, притримуючись наступних правил:
   - Одна лінія з’єднує дві точки-локації.
   - Кожна окремо взята локація не може мати більше ніж 3 з’єднуючі лінії до інших локацій.
   - Має існувати принаймні один шлях від однієї точки до іншої.
3. Для кожної точки інтересу підсумувати значення, що випали на гральних кістках, котрі відповідають її сусіднім точкам, тобто з’єднаних з нею лініями.
4. Для кожної точки інтересу призначити тип локації, відповідно до значення отриманої на попередньому етапі сумі, що відповідає призначенню в таблиці локацій.
5. Для кожної точки інтересу, звіряючись із таблицею до відповідного типу локації, шляхом підкидання гральної кості отримати значення, що відповідатиме одному з шести можливих варіантів локації.

Спираючись на виведе текстове формулювання, визначимо з якої послідовності дій або як доречно сказати на цьому етапі процедур, (в цьому випадку доречно вжити слово “процедура”) складається даний алгоритм генерації. Результат поданий у вигляді формальної блок-схеми на рисунку 2.1.

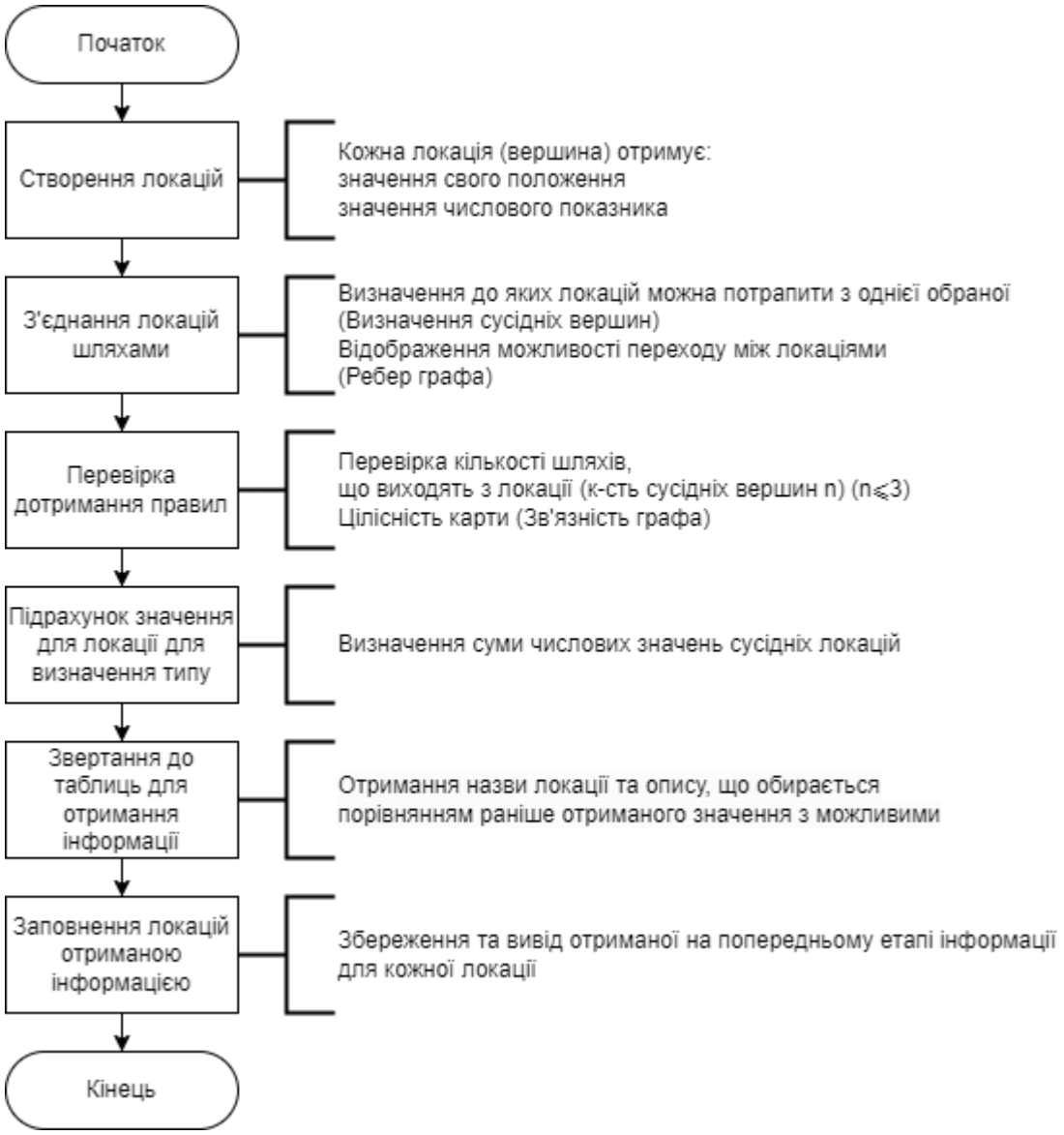

Рисунок 2.1 – формалізована блок-схема алгоритму

Варто вказати, що в описуючих кожний етап анотаціях на наведеній вище блок-схемі цілеспрямовано використовується термінологія з розділу теорії графів. В попередньо представленій текстовій інтерпретації алгоритму слово “шлях” використовується у сенсі ланцюга з ребер графа, що сполучає початкову та кінцеву вершину. Аналізуючи текстовий опис процедурної генерації, помітною стає схожість з побудовою графа. Справді, якщо переглянути приклад готової карти лісу, що представлений на сторінці Echelon Forest [рисунок 2.2] стає очевидним, що готова карта представляє з себе неорієнтований граф.

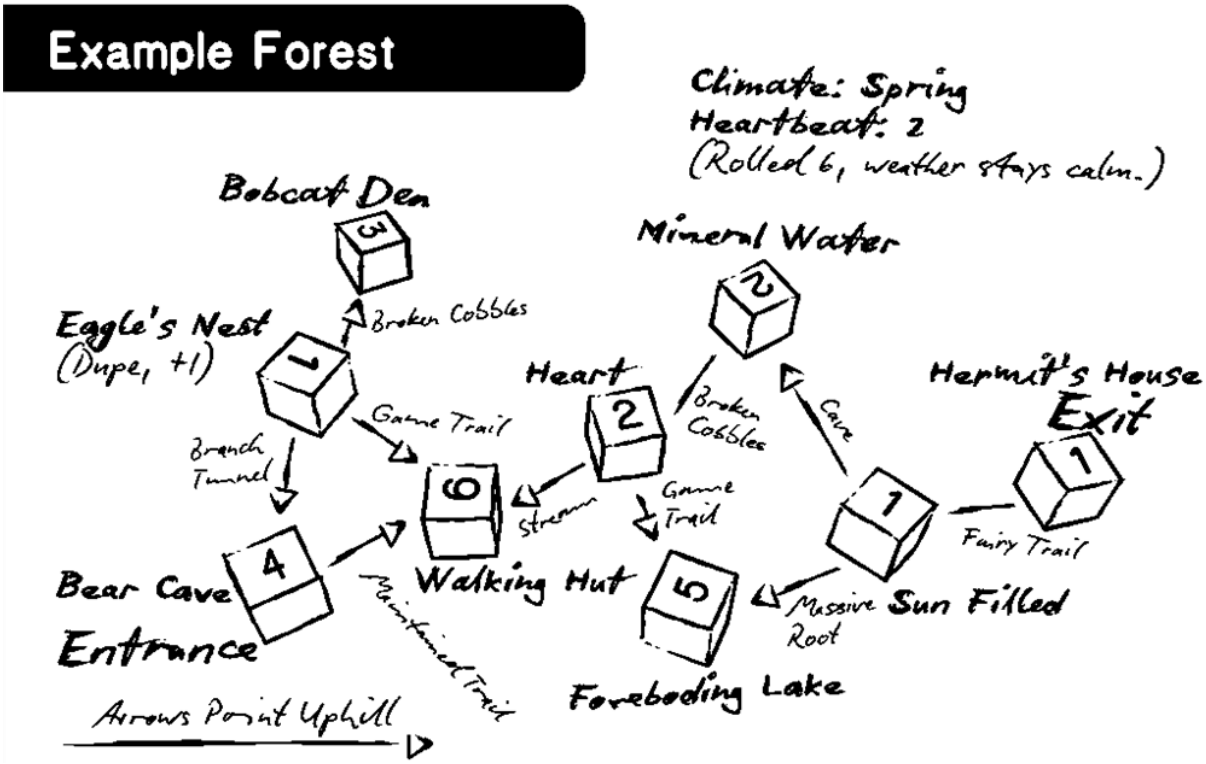

Рисунок 2.2 – приклад згенерованого лісу, намальований Браяном Стауффером [посилання на сторінку https://brstf.itch.io/] для Echelon Forest на стр. 28-29

Важливо уточнити, що не всі представлені на рисунку деталі відповідають механікам, що використані в рамках даної розробки. Автор малюнку також додав від себе вказівку на зміну висоту на шляхах певних локацій, що не варто сприймати як орієнтований граф [вставити посилання на визначення ор графу].

2.4. Постановка функціональних вимог

Тепер, маючи можливість оперувати отриманими шляхом аналізу алгоритму за яким відбуватиметься процедурна генерація та власне процедур, які мають бути реалізованими для мінімального виконання цього алгоритму, складемо список програмних функцій.

Функціональні вимоги:

- Вибір кількості локацій для генерації – правила вказують, що необхідно щонайменше 8 грального кісток, тобто для генерації потрібно щонайменше 8 точок та така ж кількість наборів значень позиції та призначеного числа від 1 до 6, що відповідатимуть кожній.
- Відображення позицій кожної локації та шляхів, котрі їх сполучають для забезпечення можливості перегляду результатів генерації користувачем.

- Визначення та збереження інформації про сусідні локації для подальшого визначення типу локації та дотримання умови про можливу кількість сусідніх вершин.
- Визначення типу локації шляхом отримання суми числових значень.
- Перевірка можливості досягнення кожної створеної локації – забезпечення можливості проходження всієї карти, по суті перевірка зв’язності графа.
- Отримання інформації (назви та опису) про локацію для генерації.
- Збереження та вивід інформації про згенеровані локації для перегляду користувачем.

2.5. Постановка нефункціональних вимог

Основуючись на викладених у попередньому розділі та сформованими на етапі аналізу особливостями розробки, визначено наступні нефункціональні вимоги:

- Візуальний інтерфейс користувача: користувач повинен мати засоби для введення змінних, наприклад, для зміни кількості локацій, що будуть згенеровані; для виведення результатів та можливість їх перегляду.
- Доступність до додатку: як було зазначено раніше, додатки, подібні до того, що розробляється, використовуються як доповнення до вже наявних інструментів для ігор, тому мають бути доступними швидко для кожного бажаючого. Найкращий спосіб досягнення цього – розміщення та відтворення додатку на сторінці в мережі Інтернет.

Висновки до розділу 2

В результаті проведеного аналізу виведено основні етапи роботи обраного алгоритму процедурної генерації; опис генерації подано у форматах тексту та блок-схеми; призначено функціональні та нефункціональні вимоги до програмного додатку.

РОЗДІЛ 3. РОЗРОБКА ДОДАТКУ

3.1. Процес розробки додатку на основі ігрового рушія Unity

Основний метод створення ігор на Unity це додавання ігрових об’єктів (GameObject) на сцену (Scene) яку в окремих випадках доцільно вважати ігровим рівнем. Як вже було згадано в підрозділі 1.4 розділу 1, ігровий об’єкт – це основна організаційна одиниця розглянутого ігрового рушія. Тому процес розробки додатку в даному випадку буде представляти з себе створення елементів гри, шляхом композиції ігрових об’єктів з компонентів різного функціоналу та їхнє розміщення на сцені.

На рисунку 3.1 зображено стандартне положення елементів інтерфейсу Unity, на якому:

1. Вікно Ієрархії. Містить в собі список всіх об’єктів, присутніх на сцені та, відповідно до власної назви, їхню ієрархію – приналежність компонентів ігровим об’єктам.
2. Сцена. Відображає положення та слугує для розміщення присутніх ігрових об’єктів. При запуску програми гри саме через сцену буде відбуватися взаємодія користувача та програми.
3. Проект. Це вікно показує всі ресурси, що використовуються у даному проекті, такі як тексти, зображення, аудіо, моделі, текстури тощо. Слугує для організації ресурсів проекту.
4. Інспектор. Відображає властивості та компоненти вибраного об'єкта зі списку ієрархії. Через нього можна налаштовувати різні параметри об'єкта, такі як координати для позиції, його розмір та інші параметри кожного його компонента.

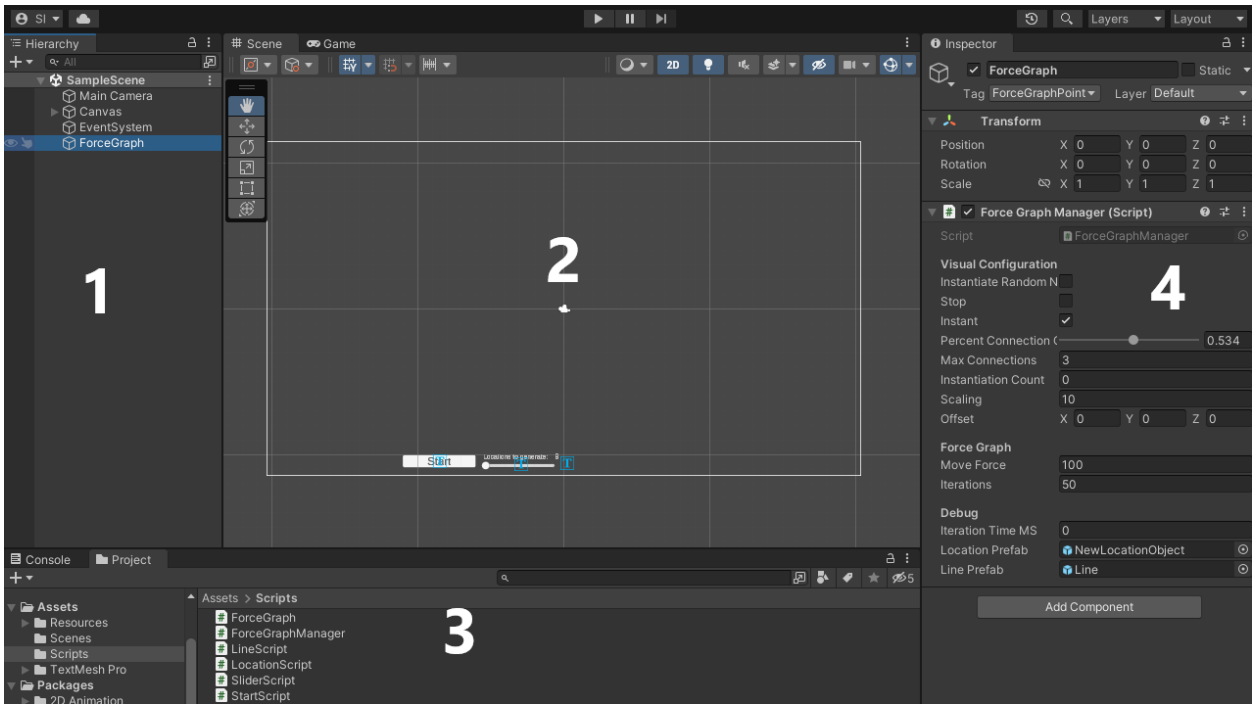

Рисунок 3.1 – знімок інтерфейсу середовища розробки Unity

На рисунку 3.1 не показана лише Консоль. Як і в будь-якому середовищі розробки консоль тут потрібна для виведення повідомлень про стан виконання програми.

Також іншою дією, що буде постійно повторюватися під час розробки, є додавання компонентів до ігрових об’єктів, а конкретно додавання скриптів. Справа в тому, що для того, щоб звертатися до створених власноруч програм та взагалі до активних об’єктів, вони повинні знаходитися на сцені.

Варто також згадати про використання префабів. У Unity префаби (Prefabs) використовуються для створення та повторного використання готових об'єктів у сцені. Префаби дозволяють створювати шаблони об'єктів, екземпляри яких можна надалі легко створювати через скрипти, або до них можна, як і до будь-якого ігрового об’єкті, додати власний скрипт.

3.2. Силовий граф

Силовий граф – це тип візуалізації, який представляє зв’язки між сутностями за допомогою вузлів (вершини графа) і зв’язків (ребра графа).

Його називають «силовим» графом, оскільки він імітує фізичні сили, що діють на вузли та ребра, щоб упорядкувати їх у візуально приємний макет.

У силовому графі вузли зазвичай представляють такі сутності, як об’єкти, індивідууми або поняття, тоді як зв’язки представляють відносини або зв’язки між ними. Компонування графа визначається застосуванням сил, які діють на вузли та ребра, змушуючи їх відштовхуватися або притягуватись.

Сили, які використовуються в симуляції на графіку сил, включають:

1. Притягання: вузли, з’єднані зв’язками, створюють силу притягання, зближуючи їх один з одним.
2. Відштовхування: вузли, які не з’єднані напряму, створюють силу відштовхування, розсовуючи їх, щоб зберегти відстань і уникнути перекриття.
3. Сила тяжіння (гравітація): спрямована до певної точки сила, яка тягне вузли графа себе, для відображення важливості вузла чи інших особливостей.

Алгоритм компонування силового графа ітераційно застосовує ці сили до вузлів і ребер, доки не буде досягнуто врівноважений стан, коли сили врівноважуються, а графік встановлюється в стабільну конфігурацію.

Силові графіки зазвичай використовуються для візуалізації складних мереж, соціальних мереж, біологічних систем і будь-яких наборів даних, де важливі зв’язки між об’єктами. Вони можуть виявити закономірності, кластери та зв’язки всередині даних, полегшуючи ідентифікацію центральних вузлів, спільнот або сфер інтересів.

Часто силові графи є інтерактивними, що дозволяє користувачам керувати графом і досліджувати його, перетягуючи вузли, збільшуючи/зменшуючи масштаб або фільтруючи певні вузли чи посилання. Ця гнучкість робить графіки сил потужним інструментом для аналізу, дослідження та представлення даних.

Використання алгоритму силового графа в даному проекті надаватиме можливість розміщувати локації: сусідні вершини будуть притягуватися одна до одної до певної міри, не утворюючи скупчень, рівномірно розподіляючись.

Силовий граф – це основний компонент розробки. Як граф, він вже зумовлює використання таких елементів, як вершини та ребра, що як було визначено на етапі аналізу та постановки задач, є невід'ємними компонентами для створення мапи локацій.

Алгоритм створення силового графа відтворює скрипт ForceGraph [скрипт Б.1], котрий описує клас силового графу. В ньому клас Node задає об'єкт вузла силового графа, що має свій ідентифікатор, список сусідніх до нього вузлів, числовий показник локації та числове значення для типу локації, що використовуються при генерації локацій, а також позиція. Методи ApplyPush, ApplyPull відповідають за симуляцію дій сил і відповідно підтягування та відштовхування вузлів. Метод SingleStepExecution виконує ці операції, відповідно до того чи є з'єднані між собою вузли, чи ні. Метод FullExecution викликає SingleStepExecution допоки не буде пройдена певна кількість ітерацій алгоритму.

3.3. Забезпечення перевірки зв'язності графа

Перевірка зв'язності графа за допомогою обходу вглиб є одним з основних алгоритмів для визначення, чи існує шлях між двома вершинами графа. Цей метод базується на рекурсивному проходженні по всіх вершинах графа, починаючи з певної стартової вершини і перевіряючи, чи можна досягти всіх інших вершин.

Основна ідея алгоритму полягає в тому, що для кожної вершини графа ми перевіряємо, чи вона була відвідана раніше. Якщо вершина вже була відвідана, то ми не продовжуємо її обробку. Якщо вершина ще не була відвідана, то ми відзначаємо її як відвідану і рекурсивно продовжуємо обхід по всіх сусідніх вершинах. Якщо під час обходу вглиб ми зможемо відвідати

всі вершини графа, то граф вважається зв'язним. У протилежному випадку, якщо є вершини, які залишилися невідвіданими, то граф буде незв'язним.

Для реалізації алгоритму перевірки зв’язності графа за допомогою обходу вглиб створено функцію, що представлена у фрагменті коду нижче на рисунку 3.2. Функція dfs реалізує алгоритм обходу вглиб. Вона приймає початкову вершину start і об'єкт типу ForceGraph, який містить інформацію про граф. Внутрішній стек використовується для зберігання вершин, які потрібно обробити. За допомогою циклу while вершини вибираються зі стека і перевіряються сусідні вершини, які ще не були відвідані. Відвідані вершини відзначаються у словнику visited.

```
Dictionary<ForceGraph.Node, bool> visited = new Dictionary<ForceGraph.Node, bool>();
1 reference
public void dfs(ForceGraph.Node start, ForceGraph ForceGraph)
{
    Stack<ForceGraph.Node> stack = new Stack<ForceGraph.Node>();

    stack.Push(start);
    visited[start] = true;

    while (stack.Count > 0)
    {
        ForceGraph.Node node = stack.Pop();

        foreach (var nodeI in ForceGraph.Nodes)
        {
            foreach (var neighbour in node.ConnectedNodes)
            {
                if (!visited[neighbour])
                {
                    stack.Push(neighbour);
                    visited[neighbour] = true;
                }
            }
        }
    }
}
```

Рисунок 3.2 – функція dfs

Власне перевірка графу на зв’язність відбувається в коді, що йде слідом: функція Is_connected перевіряє, чи є граф зв'язним. Вона викликає dfs для першої вершини графа та перевіряє, чи всі вершини були відвідані. Якщо яка-небудь вершина залишилася невідвіданою, то граф вважається незв'язним. Метод CheckGraphConnectivity по суті є точкою входу для перевірки зв'язності графа, викликає Is_connected і повертає результат перевірки. Після перевірки словник visited очищується. Код представлений на рисунку 3.3.

```
public bool Is_connected(ForceGraph ForceGraph)
{
    dfs(ForceGraph.Nodes.First(), ForceGraph);

    foreach (KeyValuePair<ForceGraph.Node, bool> entry in visited)
    {

        // If any vertex it not visited in any direction
        // Then graph is not connected
        if (entry.Value == false)
            return false;
    }

    // If graph is connected
    return true;
}
2 references
public bool CheckGraphConnectivity(ForceGraph ForceGraph)
{
    foreach (ForceGraph.Node node in ForceGraph.Nodes)
    { visited.Add(node, false); };

    if (Is_connected(ForceGraph) == true)
    {
        UnityEngine.Debug.Log("Yes");
        visited.Clear();
        return true;
            }
    else
    {
        UnityEngine.Debug.Log("No");
        visited.Clear();
        return false;

    };
}
```

Рисунок 3.3 – Is_connected, CheckGraphConnectivity

3.4. Створення екземплярів вузлів

Функція для випадкової генерації вершин графа [рис. 3.4], працює наступним чином:

1. Спочатку вона скидає кількість ітерацій до початкового значення Iterations.
2. Очищує список вершин графа ForceGraph.Nodes, щоб почати заново, якщо це не перший запуск.
3. У циклі for генерує випадкові вершини графа. Кількість вершин визначається змінною InstantiationCount. Кожна вершина отримує унікальний ідентифікатор id та випадкову позицію Position, яка обчислюється як випадкова точка в одиничному колі за допомогою Random.insideUnitCircle.
4. Після генерації вершин розпочинається створення з'єднань між вершинами. Перший вкладений цикл for перебирає вершини в графі, починаючи з першої ітерації.

5. У другому вкладеному циклі for перебираються наступні вершини, починаючи з наступної після поточної вершини.
6. Перевіряється, чи вже є з'єднання між двома вершинами. Якщо так, то переходиться до наступної ітерації циклу.
7. Виконується перевірка, чи має поточна вершина nodeI дозвіл на додавання нових з'єднань. Це обмежується distrubutionCap, який обчислюється як добуток PercentConnectionChance на InstantiationCount. Якщо лічильник з'єднань перевищує цей ліміт, то переходиться до наступної ітерації циклу.
8. За допомогою генератора випадкових чисел Random.Range, з'єднання додається між nodeI та nodeJ з ймовірністю PercentConnectionChance поділеною на кількість вже з'єднаних вершин у nodeI та обмеженою кількістю з’єднань _MaxConnections. Зв'язки додаються в обидва напрямки: від nodeI до nodeJ та від nodeJ до nodeI.

```
private void InstantiateRandomNodes(ForceGraph ForceGraph)
{
    IterationsRemaining = Iterations;

    ForceGraph.Nodes.Clear();

    for (int i = 0; i < InstantiationCount; i++)
    {
        ForceGraph.Nodes.Add(new ForceGraph.Node()

        {
            id = i,
            Position = Random.insideUnitCircle * 1f

        });
    }
    var distrobutionCap = PercentConnectionChance * InstantiationCount;
    for (int i = 0; i < InstantiationCount - 1; i++)
    {
        var nodeI = ForceGraph.Nodes[i];
        if (nodeI.ConnectedNodes.Count >= distrobutionCap) continue;

        for (int j = i + 1; j < InstantiationCount; j++)
        {
            var nodeJ = ForceGraph.Nodes[j];

            if (nodeI.ConnectedNodes.Contains(nodeJ)) continue;

            if (Random.Range(0f, 1f) <= PercentConnectionChance / nodeI.ConnectedNodes.Count && _MaxConnections > nodeI.ConnectedNodes.Count)
            {
                nodeI.ConnectedNodes.Add(nodeJ);
                nodeJ.ConnectedNodes.Add(nodeI);
            }
        }
    }
}
```

Рисунок 3.4 – InstantiateRandomNodes

3.5. Створення ліній за допомогою компоненту Line Renderer

Компонент LineRenderer в Unity використовується для малювання простих ліній та кривих у тривимірному просторі. Він дозволяє створювати візуальні зображення ліній або кривих, які можуть бути використані для відображення з'єднань, траєкторій руху об'єктів, контурів тощо. За рахунок нього створюється візуальне представлення з’єднань між вершинами графа. У фрагменті коду на рисунку 3.5.1 це реалізовано наступним чином:

1. Спочатку створюється список AllNodes, який містить всі вершини графа ForceGraph.Nodes.
2. Створюється порожній список NodesConnections, який буде містити пари вершин, які мають з'єднання між собою.
3. За допомогою вкладених циклів foreach, перебираються всі вершини nodeI графа, а потім перебираються їх сусідні вершини nodeJ зі списку ConnectedNodes.
4. Для кожної пари вершин nodeI та nodeJ, створюється новий масив int[] з ідентифікаторами вершин nodeI.id та nodeJ.id, і цей масив додається до списку NodesConnections. Таким чином, ми отримуємо список з'єднань між вершинами графа.
5. Після створення списку з'єднань, виконується перевірка на дублікати з'єднань. Зовнішній цикл for перебирає з'єднання в зворотньому порядку, щоб забезпечити коректне видалення дублікатів.
6. У внутрішньому циклі for перебираються з'єднання з індексами меншими за поточне з'єднання i. З'єднання NodesConnections[i] має дві вершини i1 та i2, а з'єднання NodesConnections[y] має вершини y1 та y2.
7. Перевіряється, чи мають з'єднання однакові вершини, але в протилежному порядку. Якщо таке з'єднання знайдено, воно видаляється зі списку NodesConnections, оскільки це є дублікатом.

8. Нарешті, за допомогою циклу foreach перебираються всі пари вершин у списку NodesConnections, і для кожної пари створюється графічний об'єкт (префаб) за допомогою функції Instantiate. Далі встановлюються початкова і кінцева точки лінії, використовуючи позиції вершин AllNodes[NodeCon[0]].Position та AllNodes[NodeCon[1]].Position.

```
private void ConnectionLines(ForceGraph ForceGraph)
{
    List<ForceGraph.Node> AllNodes = ForceGraph.Nodes;

    List<int[]> NodesConnections = new List<int[]>();

    foreach (ForceGraph.Node nodeI in AllNodes)
    {
        foreach (ForceGraph.Node nodeJ in nodeI.ConnectedNodes)
        {
            NodesConnections.Add(new int[] { nodeI.id, nodeJ.id });
        }
    }

    for (int i = NodesConnections.Count - 1; i >= 0; i = i-1)
    {
        int i1 = NodesConnections[i][0];
        int i2 = NodesConnections[i][1];

        for (int y = 0; y < i; y++)
        {

            int y1 = NodesConnections[y][0];


            int y2 = NodesConnections[y][1];
            //UnityEngine.Debug.Log($"{i1} {i2} {y1} {y2}");
            if (i1 == y2 && i2 == y1) { NodesConnections.RemoveAt(i);  }
            //UnityEngine.Debug.Log("Deleted");

        }
    }
    foreach (int[] NodeCon in NodesConnections)
    {
        GameObject LineCreate = Instantiate(LinePrefab, Vector3.zero, Quaternion.identity);
        LineScript Ls;
        Ls = LineCreate.GetComponent<LineScript>();
        Ls.LineUpdate(((AllNodes[NodeCon[0]].Position) * Scaling + Offset), ((AllNodes[NodeCon[1]].Position) * Scaling + Offset));
    }
}
```

Рисунок 3.5.1 – ConnectionLines

Префабом, екземпляр якого створюється наприкінці виконання згаданого скрипту, є LineRenderer, який в свою чергу містить скрипт LineScript, що визначає його поведінку [рис. 3.5.2]. Він працює наступним чином:

1. Змінні startPoint та endPoint: Вектори, що визначають початкову та кінцеву точки лінії.
2. Змінна lineRenderer: Зберігає посилання на компонент LineRenderer, що прикріплений до цього об'єкта.

3. Метод Awake(): Викликається при запуску скрипту. Він отримує посилання на компонент LineRenderer і встановлює кількість позицій (2) та початкові позиції лінії.
4. Метод LineUpdate(Vector3 x1, Vector3 x2): Оновлює позиції лінії, приймаючи нові значення для початкової (x1) та кінцевої (x2) точок. Викликається для зміни положення лінії.

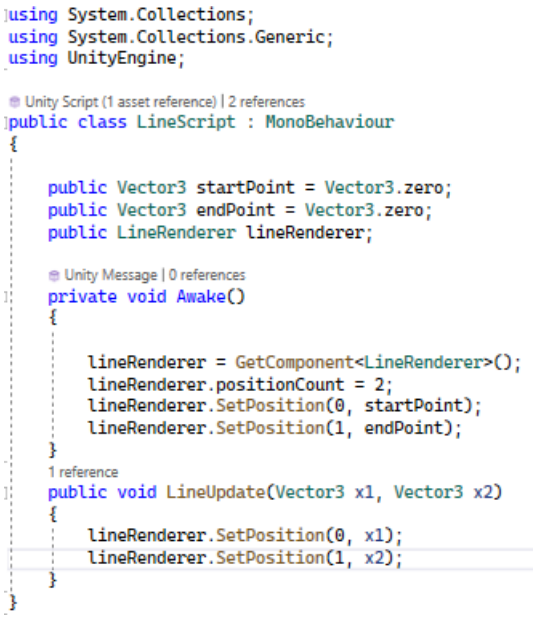

Рисунок 3.5.2 – LineScript

3.6. Отримання опису локацій

Опис локацій зберігається у текстових документах, у форматі “Назва?Текстовий опис”. Назва кожного документа відповідає типу локації. Фрагмент коду на рисунку 3.6.1 є функцією, що формує з отриманого текстового документу словник, що складається з назви і відповідного їй опису.

```
public Dictionary<string, string> Abodes;
public Dictionary<string, string> Glades;
public Dictionary<string, string> WaterFeature;
public Dictionary<string, string> OldRuins;
public Dictionary<string, string> OddStones;
public Dictionary<string, string> Assorted;

6 references
static Dictionary<string, string> CreateDictionary(string textAsset)
{
    Dictionary<string, string> dictionary = new Dictionary<string, string>();
    try
    {
        // Зчитуємо всі рядки з файлу
        string[] lines = textAsset.Split('\n');

        // Проходимося по кожному рядку
        foreach (string line in lines)
        {
            // Розбиваємо рядок на ключ і значення
            string[] parts = line.Split('?');

            if (parts.Length >= 2)
            {
                // Отримуємо ключ (перші два слова у рядку)
                string key = parts[0];

                // Отримуємо значення (решта рядка)
                string value = parts[1];

                // Додаємо пару ключ-значення у словник
                dictionary.Add(key, value);
            }
        }
    }
    catch (Exception ex)
    {
        UnityEngine.Debug.Log("Виникла помилка: " + ex.Message);
    }
    return dictionary;
}
```

Рисунок 3.5.1 – CreateDictionary

Отримання текстових файлів (завантаження їх з папки ресурсів проекту) можливе тільки перед початком основної програми, тому їх виклик розміщений у методі Start. Там же відбувається їх перетворення та переписування у словники [рис. 3.5.2].

```
public void Start()
{

    TextAsset LocDescAbodes = Resources.Load<TextAsset>("LocationsDescriptions/LocationsDescription_Abodes");
    Abodes = CreateDictionary(LocDescAbodes.text);

    TextAsset LocDescAssorted = Resources.Load<TextAsset>("LocationsDescriptions/LocationsDescription_Assorted");
    Assorted = CreateDictionary(LocDescAssorted.text);

    TextAsset LocDescGlades = Resources.Load<TextAsset>("LocationsDescriptions/LocationsDescription_Glades");
    Glades = CreateDictionary(LocDescGlades.text);

    TextAsset LocDescOddStones = Resources.Load<TextAsset>("LocationsDescriptions/LocationsDescription_OddStones");
    OddStones = CreateDictionary(LocDescOddStones.text);

    TextAsset LocDescOldRuins = Resources.Load<TextAsset>("LocationsDescriptions/LocationsDescription_OldRuins");
    OldRuins = CreateDictionary(LocDescOldRuins.text);

    TextAsset LocDescWaterFeature = Resources.Load<TextAsset>("LocationsDescriptions/LocationsDescription_WaterFeature");
    WaterFeature = CreateDictionary(LocDescWaterFeature.text);
```

Рисунок 3.5.2 – Завантаження ресурсів

3.6. Створення екземплярів локацій

Процес створення локацій поєднує в собі одразу декілька методів, які необхідно виконувати послідовно. Такі ускладнення зумовлені насамперед бажанням притримуватися процесу генерації, що виконується в грі-першоджерелі алгоритму.

Спочатку, для вершини, в залежності від значення суми числових значень її сусідніх вершин, призначається тип локації. Функція GetLocationType, представлена на рисунку 3.6.1 нижче, порівнює числове значення типу локації LocationValue із можливими значеннями і повертає відповідний словник, з якого буде виконуватися вибір локації.

```
public Dictionary<string, string> GetLocationType(int LocationValue)
{
    switch (LocationValue)
    {
        case 1:
        case 7:
        case 13:
            return Abodes;
        case 2:
        case 8:
        case 14:
            return Glades;
        case 3:
        case 9:
        case 15:
            return WaterFeature;
        case 4:
        case 10:
        case 16:
            return OldRuins;
        case 5:
        case 11:
        case 17:
            return OddStones;
        case 6:
        case 12:
        case 18:
            return Assorted;
        default:
            return null;
    }
}
```

Рисунок 3.6.1 – GetLocationType

Далі наступна функція GetRandomFromDictionary повертає з отриманого словника випадкову пару ключ-значення [рис. 3.6.2].

```
1 reference
public KeyValuePair<string, string> GetRandomEntryFromDictionary(Dictionary<string, string> dictionary)
{

    if (dictionary != null)
    {
        int randomIndex = Random.Range(1,dictionary.Count);
        return dictionary.ElementAt(randomIndex);
    }
    else UnityEngine.Debug.Log("smth wrong with random from dict");
    return default(KeyValuePair<string, string>);
}
```

Рисунок 3.6.2 – GetRandomFromDictionary

Функція InstantiateLocation на рисунку 3.6.3 поєднує в собі всі ці вище перелічені методи для створення екземплярів локацій на позиціях вузлів графу.

```
private void InstantiateLocations(ForceGraph ForceGraph)
{
    foreach (var node in ForceGraph.Nodes)
    {
        GameObject LocationToInst = Instantiate(LocationPrefab, (node.Position * Scaling + Offset), Quaternion.identity);
        node.LocationType = node.ConnectedNodes.Sum(nodeIn => nodeIn.LocationNumber);
        //UnityEngine.Debug.Log(node.LocationType);
        KeyValuePair<string, string> randomEntry = GetRandomEntryFromDictionary(GetLocationType(node.LocationType));

        LocationToInst.GetComponent<LocationScript>().LocationUpdate(randomEntry.Key, randomEntry.Value);
    }

}
```

Рисунок 3.6.3 – InstantiateLocation

Остання строка вище згаданої функції звертається до методу LocationUpdate, що скрипт якого є компонентом префабу локації. Цей код на рисунку 3.6.4 знаходить та звертається до текстових компонентів ігрового об’єкті локації та вносить в них зміни, а саме призначає текст в поля назви локації та опису. Остання строка всередині вже цієї функції створює фон для тексту, повторюючи контур опису.

```
public void LocationUpdate(string LocationName, string LocationDescription)
{

    transform.Find("LocationName").GetComponent<TMP_Text>().text = LocationName;
    transform.Find("LocationDescription").GetComponent<TMP_Text>().text = LocationDescription;
    transform.Find("LocDescBakground").GetComponent<TMP_Text>().text = $"<mark=#C8780Fff padding=\"10, 10, 10, 10\"> {LocationDescription} </mark>";

}
```

Рисунок 3.6.4 – LocationUpdate

3.8. Показ та приховання опису локації

Для забезпечення можливості зручного читання опису локацій, по замовчуванню текст є прихованим, допоки курсор мишки не буде наведено на назву локації, що виводиться над позначенням для кожної вершини графа. Це досягається за рахунок додавання на зайнятий назвою простір компоненту коллайдеру (Collider). Колайдер здатен отримувати дані про знаходження курсору мишки в межах свого контуру [рис. 3.8.1].

```
public class LocationScript : MonoBehaviour
{
    public TextMeshPro LocDescObject;
    public TextMeshPro LocDescBack;
    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()
    {

        LocDescBack.enabled = false;
        LocDescObject.enabled = false;

    }

    1 reference
    public void LocationUpdate(string LocationName, string LocationDescription)
    {

        transform.Find("LocationName").GetComponent<TMP_Text>().text = LocationName;
        transform.Find("LocationDescription").GetComponent<TMP_Text>().text = LocationDescription;
        transform.Find("LocDescBakground").GetComponent<TMP_Text>().text = $"<mark=#C8780fff padding=\"10, 10, 10, 10\"> {LocationDescription} </mark>";

    }


    Unity Message | 0 references
    private void OnMouseEnter()
    {
        // Set the visibility of the TextMeshPro component to true
        LocDescObject.enabled = true;
        LocDescBack.enabled = true;

    }

    Unity Message | 0 references
    private void OnMouseExit()
    {
        // Set the visibility of the TextMeshPro component to false
        LocDescObject.enabled = false;
        LocDescBack.enabled = false;

    }
}
```

Рисунок 3.8.1 – LocationScript, за приховування та показ тексту відповідають методи OnMouseEnter та OnMouseExit

Для того, щоб це працювало, необхідно не тільки приєднати скрипт як компонент ігрового об’єкту, але й вказати змінні які прийматиме скрипт. Як виглядає готове налаштування зображено на рисунку 3.8.2.

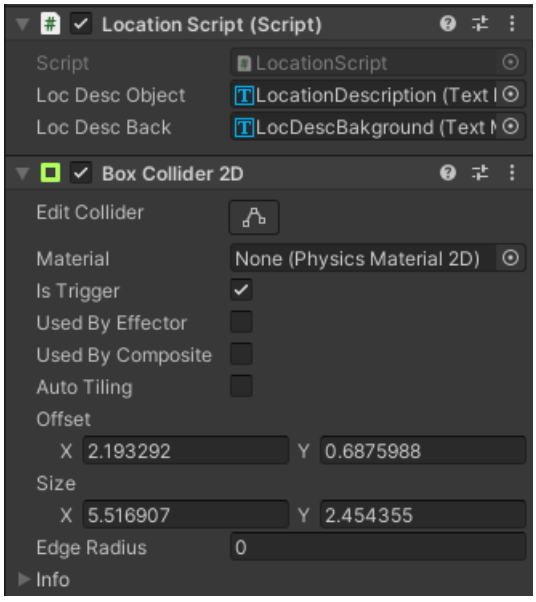

Рисунок 3.8.2 – налаштування для скрипту та коллайдер

3.9. Інтерфейс користувача: кнопка пуску та контроль кількості локацій до генерації

Для контролю запуску генерації та для можливості вводу кількості локацій для генерації створено кнопку «Start» та слайдер (Slider), значення останнього приймається скриптом ForceGraphManager – компіляцією зі всіх створених функцій, від якого наслідує StartScript, котрий в свою чергу запускається при натисканні на кнопку. Для цього необхідно налаштувати подію OnClick для кнопки таким чином, як це можна побачити на рисунку 3.9.

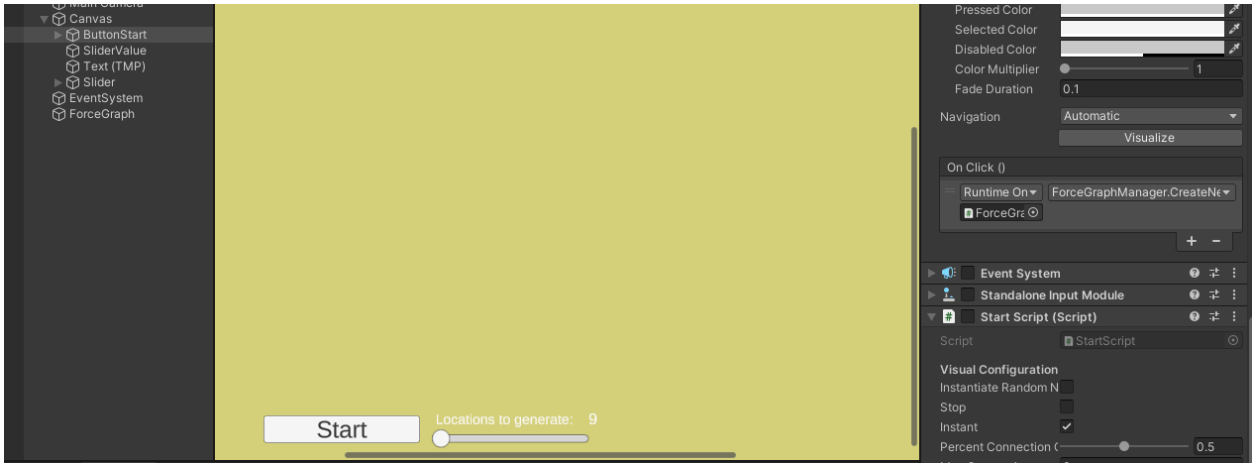

Рисунок 3.9 – кнопка, слайдер, налаштування кнопки у вікні інспектора

3.10. Алгоритм роботи програми

Після забезпечення виконання всіх функціональних вимог, забезпечимо послідовний виклик розроблених функцій через створення нової функції CreateNew[рис.3.10.1], алгоритм роботи якої переданий на блок-схемі 3.10.2.

```
public void CreateNew()
{

    ForceGraph NewGraph = new ForceGraph();


    ClearLocations();
    ClearLines();
    InstantiateRandomNodes(NewGraph);
    if (IterationsRemaining < 0) return;
    //var duration = new Stopwatch();
    //duration.Start();

    IterationsRemaining = -1;
    NewGraph.FullExecution();
    if (CheckGraphConnectivity(NewGraph))
    {
        InstantiateLocations(NewGraph);
        ConnectionLines(NewGraph);
    }
    else CreateNew();
    //duration.Stop();
}
```

Рисунок 3.10.1 – CreateNew

Рисунок 3.10.2 – блок-схема алгоритму виконання програми

3.11. Підготовка до розміщення на онлайн-платформі

Перед тим, як викласти проект у мережу, його необхідно правильно забілдити. Для цього з варіантів можливих платформ обрати платформу на якій буде відбуватися запуск додатку. У випадку цього проекту це WebGL – варіант білду для запуску у браузері. Варто відмітити, що готовий проект після білду без хостінгу на сервері не можна запустити локально просто

відкривши його файли. Для цього можна використовувати кнопку Build And Run у цьому ж самому меню [рис. 3.11.1].

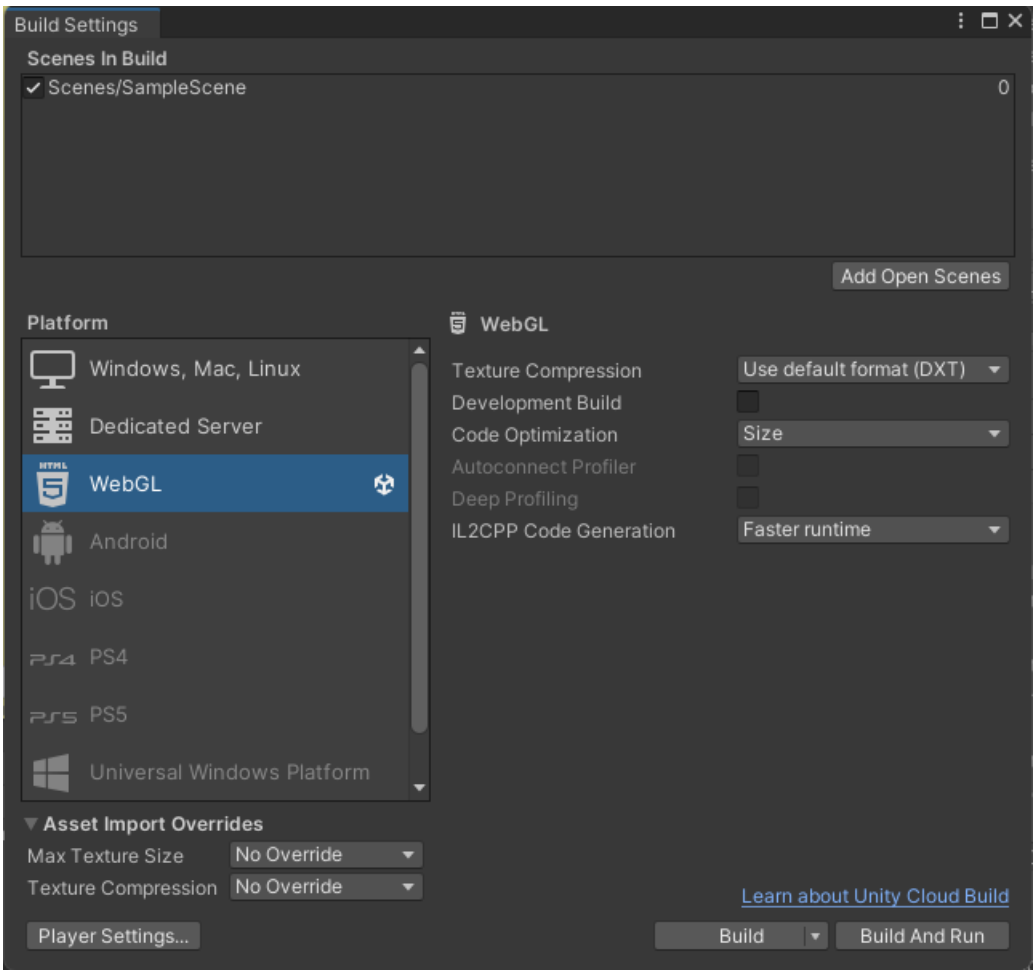

Рис. 3.11.1 – опції білду проекту

Отримавши справний білд проекту, для його завантаження на itch.io необхідно впевнитися, що папка з файлами проекту має в собі документ формату HTML під назвою index. Він слугує в якості точки входу і потрібен на стороні серверу itch.io щоб зібрати сторінку проекту з інтерактивним вікном для розміщеного на сторінці додатку.

Для того щоб розмістити будь-який свій проект на itch.io перш за все необхідно створити власний обліковий запис на цій платформі. Процес реєстрації аналогічний такому, що притаманний кожній платформі з подібним функціоналом. Обов’язково треба підтвердити електронну пошту, вказану при реєстрації – непідтверджені користувачі можуть писати

коментарі на сторінках чужих проектів, але не мають змоги завантажувати власні файли.

Після входу в обліковий запис, стає доступною можливість створити свій проект. Це можна зробити перейшовши по посиланню “Create new project”. Після перенаправлення, будуть доступні наступні налаштування проекту[рис. 3.11.2]: заголовок адреса, опис, класифікація, тип проекту, статус публікації та інші.

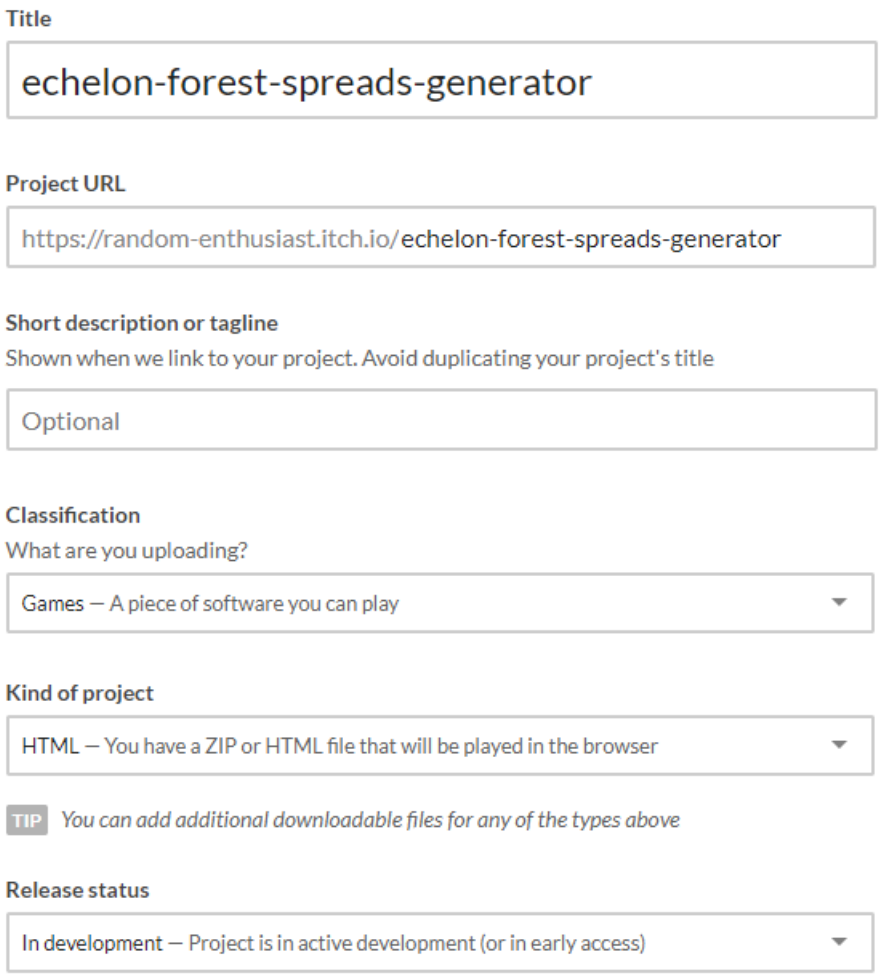

Рисунок 3.11.2 – деякі основні параметри налаштування сторінки проекту на itch.io

Далі необхідно завантажити стиснутий архів з файлами проекту. Це робиться у розділі Uploads, знімок якого показаний на рисунку 3.11.3.

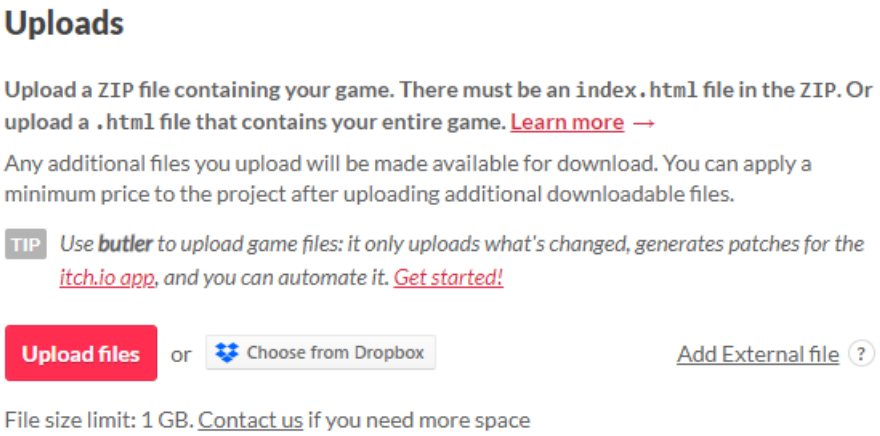

Рисунок 3.11.3 – розділ Uploads в налаштуваннях проекту

Після цього можна вибрати тип видимості сторінки[рис. 3.11.4]. На момент написання, сторінка не оформлена щоб бути презентабельною, тому обрано варіант “Draft” (Чорновик).

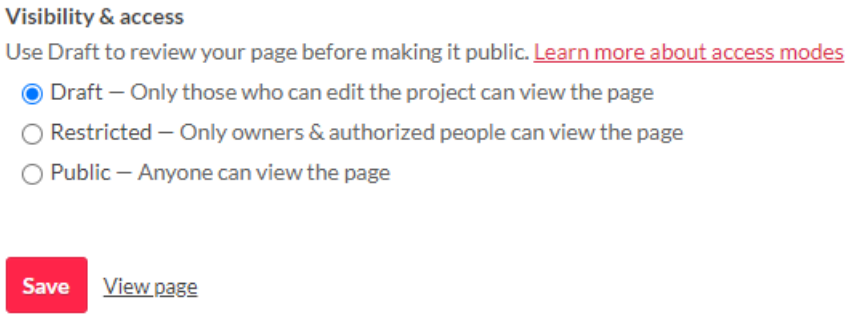

Рисунок 3.11.4 – вибір типу доступу та видимості проекту

ВИСНОВКИ

В результаті роботи над даною кваліфікаційною роботою було розроблено альфа-версію додатку генератора ігрових мап, в основі роботи якого лежить алгоритм процедурної генерації та алгоритм візуалізації даних типу силовий граф. Додаток побудовано на базі ігрового рушія Unity, що також включає використання мови програмування C#. На практиці досліджено методику розробки додатку засобами Unity, за рахунок чого було виявлені переваги використання ігрового рушія та конкретно використання ігрового рушія Unity. За рахунок застосування принципу композиції для створення ігрових об’єктів, створення елементів гри є досить інтуїтивним і може бути швидко опанованим до рівня, достатнього для створення та публікації власних ігор.

Було розглянуто онлайн-платформу itch.io як засіб для хостингу власних розробок. За допомогою цієї платформи кожен бажаючий може поділитися власним проектом та навіть за бажанням монетизувати його на власну користь. Також за рахунок можливості самостійного налаштування сторінки проекту можна зекономити власні ресурси, що в іншому випадку пішли б на розробки свого веб-сайту та організацію з орендуванням серверу в якості хостинг-платформи.

Досліджено та впроваджено в проекті використання методів силових алгоритмів візуалізації графів. Розроблений алгоритм має потенціал для використання в широкому спектрі інших проектів, наприклад заради візуалізації даних, таких як відстані, зв’язки у мережах тому, взаємодія об’єктів.

Додаток перебуває на стадії альфа-тестування. Наступними потенційними етапами його розробки можуть бути доопрацювання естетичної складової візуалізації, тобто створення унікальних зображень для локацій, додавання функцій для відтворення решти механік гри-першоджерела, розширення алгоритму процедурної генерації для забезпечення ще більшої різноманітності результатів генерації.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Документація Unity URL: https://docs.unity3d.com/Manual/index.html
2. Документація C# URL:https://learn.microsoft.com/en-us/dotnet/csharp/
3. Офіційний сайт Unity URL: https://unity.com/
4. Освітній ресурс Unity URL: https://learn.unity.com/
5. Офіційне співтовариство Unity URL: https://community.unity.com/
6. Кузьменко І.М. Теорія графів. Київ КПІ ім. Ігоря Сікорського 2020 URL: https://ela.kpi.ua/bitstream/123456789/35854/1/Teoriia_hrafiv.pdf
7. Боднарчук Ю.В., Олійник Б.В. ОСНОВИ ДИСКРЕТНОЇ МАТЕМАТИКИ (для студентів-інформатиків) Київ 2007 URL: https://www.ukma.edu.ua/~bogd/Discrete%20Mathematics/PosibnykNew.pdf
8. Сторінка проекту A Traveler’s Guide to Echelon Forest URL: https://awkwardturtle.itch.io/echelon-forest
9. Автор ілюстрації прикладу згенерованого лісу URL: https://brstf.itch.io/
10. Ліцензія Creative Commons URL: https://creativecommons.org/licenses/by/4.0/

ДОДАТКИ

ДОДАТОК А. ТЕКСТИ З ПЕРШОДЖЕРЕЛА

Текст А.1: Інструкція по генерації лісу – Echelon Forest сторінка 6 – Forest Generation та сторінка 7 Locations

Drop a handful of d6 (at least 8) on a piece of paper.

Draw a line through a single die such that there are equal numbers of dice on each side (or as close as you can get). That die is the Heart of the forest, the line is the Meridian.

Draw paths between the rest of the dice, following these rules:

1. A path connects two dice.
2. Paths cannot cross one another.
3. Paths cannot cross the meridian.
4. A given location cannot have more than 3 connected paths.
5. Note the furthest dice from the Heart on each side of the Meridian. These are the forest entrances.
6. There must be at least one route connecting the two forest entrances. Determine each location following the instructions on the opposite page.

Determine the nature of the paths between each location following the instructions Page 22.

Finally determine the current Season on Page 24 and the starting Heartbeat on Page 27.

Текст А.2: Приклад опису локацій типу Abodes

1. Hermit's House: A small hut with a well maintained herb garden beside. A kindly old lady who knows much about the forest has lived here for a great many years, but will offer only tea and enigmatic advice if asked for information. The forest and guardian will respond with immediate violence if the Hermit is threatened with harm.
2. Eagle's Nest: A pair of eagles make their nest in the crook of a large tree. They are highly territorial and will react violently to any attempt to approach the tree or climb it.
3. Bobcat Den: A fallen tree forms the home for a grey and white bobcat. Territorial, but easily bribed with gifts of meat.

4. Bear Cave: A shallow cave in the side of a hill, overgrown with moss and lichen. Contains a bear's den, 3 in 6 chance it's currently present.
5. Lean-to: A simple shelter constructed from local materials. Signs of a camp fire remain inside, but otherwise uninhabited.
6. Walking Hut: A small cottage standing on giant chicken legs exuding an oppressive aura. Empty, although in good repair despite not being currently occupied. Could possibly be coerced to provide transport, but will not leave the forest.

ДОДАТОК Б. РОЗРОБЛЕНІ СКРИПТИ C#

Скрипт Б.1 – ForceGraph

```
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using Unity.VisualScripting;
using UnityEngine;

public class ForceGraph
{

    public class Node
    {
        public int id;
        public List<Node> ConnectedNodes = new List<Node>();
        public int LocationNumber = Random.Range(1, 6);
        public int LocationType;
        public Vector3 Position;


    }

    public List<Node> Nodes = new List<Node>();

    public float MoveForce = 10f;
    public int IterationsToStop = 100;
    public float FrameTime = 1f / 60;
    public Vector3 AnchorPosition= Vector3.zero;
    public void FullExecution()
    {
        for(int i = 0; i < IterationsToStop; i++)
        {
            SingleStepExecution();

        }
    }
    public void SingleStepExecution()
    {
        foreach (var n1 in Nodes)
        {
            ApplyPull(n1, AnchorPosition, n1.Position.magnitude);
            foreach (var n2 in Nodes)
            {
```

```
                if (n1 == n2) continue;
                var connected = n1.ConnectedNodes.Contains(n2);
                var distance = (n1.Position - n2.Position).magnitude;

                if (connected)
                {
                    ApplyPull(n1, n2.Position, distance);

                }

                ApplyPush(n1, n2.Position, distance);
            }
        }

    }
    private void ApplyPush(Node n1, Vector3 toPosition, float distance)
    {
        var diff = n1.Position - toPosition;
        var dir = diff.normalized;
        var force =
            MoveForce
            * FrameTime
            * dir
            * (1f - (Mathf.Clamp(distance, 0, 1f)));
        n1.Position += force;
    }
    private void ApplyPull(Node n1, Vector3 toPosition, float distance)
    {
        var diff = n1.Position - toPosition;
        var dir = diff.normalized;
        var force =
            MoveForce
            * FrameTime
            * dir
            * Mathf.Clamp(distance, 0, 1f);
        n1.Position -= force;
    }
}
```

Скрипт Б.2 – LineScript

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class LineScript : MonoBehaviour
{

    public Vector3 startPoint = Vector3.zero;
    public Vector3 endPoint = Vector3.zero;
    public LineRenderer lineRenderer;

    private void Awake()
    {

        lineRenderer = GetComponent<LineRenderer>();
        lineRenderer.positionCount = 2;
        lineRenderer.SetPosition(0, startPoint);
        lineRenderer.SetPosition(1, endPoint);
    }
    public void LineUpdate(Vector3 x1, Vector3 x2)
    {
        lineRenderer.SetPosition(0, x1);
        lineRenderer.SetPosition(1, x2);}}
```

Скрипт Б.3 – LocationScript

```
using System.Collections;
using System.Collections.Generic;
using TMPro;
using Unity.VisualScripting;
using UnityEngine;
using UnityEngine.EventSystems;
using UnityEngine.UI;

public class LocationScript : MonoBehaviour
{
    public TextMeshPro LocDescObject;
    public TextMeshPro LocDescBack;
    //public GameObject LocDescObject;
    // Start is called before the first frame update
    void Start()
    {
        LocDescBack.enabled = false;
        LocDescObject.enabled = false;

    }

    public void LocationUpdate(string LocationName, string LocationDescription)
    {

        transform.Find("LocationName").GetComponent<TMP_Text>().text =
LocationName;
        transform.Find("LocationDescription").GetComponent<TMP_Text>().text =
LocationDescription;
        transform.Find("LocDescBakground").GetComponent<TMP_Text>().text =
$"<mark=#C8780Fff padding=\"10, 10, 10, 10\"> {LocationDescription} </mark>";

    }
    private void OnMouseEnter()
    {
        // Set the visibility of the TextMeshPro component to true
        LocDescObject.enabled = true;
        LocDescBack.enabled = true;
    }

    private void OnMouseExit()
    {
        // Set the visibility of the TextMeshPro component to false
        LocDescObject.enabled = false;
        LocDescBack.enabled = false;
    }

}
```

Скрипт Б.4 – SliderScript

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using TMPro;

public class SliderScript : MonoBehaviour
{
    [SerializeField] private Slider slider;
    [SerializeField] private TMP_Text slider_text;
    public static int InstValue;
    // Start is called before the first frame update
    void Start()
    {
        slider.onValueChanged.AddListener((v) =>
        {
```

```
                slider_text.text = v.ToString();
                InstValue = Mathf.RoundToInt(v);
            });

        }
}
```

Скрипт Б.5 – StartScript

```
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;
using UnityEngine;

public class StartScript : ForceGraphManager
{
    public void ButtomStart()
    {
        ForceGraph forceGraph = new ForceGraph();

        CreateNew();
    }
}
```

Скрипт Б.6 – ForceGraphManager

```
using System.Collections.Generic;
using System.Diagnostics;
using UnityEngine;
using System.Linq;
using System;
using Random = UnityEngine.Random;
using UnityEngine.UI;

public class ForceGraphManager : MonoBehaviour
{

    [Header("Visual Configuration")]
    public bool _InstantiateRandomNodes = false;
    public bool _Stop = false;
    public bool _Instant = true;
    [Range(0f, 1f)]
    public float PercentConnectionChance = 0.5f;
    public int _MaxConnections = 3;

    public int InstantiationCount = SliderScript.InstValue;//9;
    public float Scaling = 10;//1f;
    public Vector3 Offset = Vector3.zero;
    [Header("Force Graph")]
    public float MoveForce = 100; //10f;
    public int Iterations = 50; //100;
    [Header("Debug")]
    public float IterationTimeMS = 0;
    private int IterationsRemaining = 0;
    private ForceGraph ForceGraph = new ForceGraph();

    public GameObject LocationPrefab;
    public GameObject LinePrefab;

    public Dictionary<string, string> Abodes;
    public Dictionary<string, string> Glades;
    public Dictionary<string, string> WaterFeature;
    public Dictionary<string, string> OldRuins;
```

```
    public Dictionary<string, string> OddStones;
    public Dictionary<string, string> Assorted;
    static Dictionary<string, string> CreateDictionary(string textAsset)
    {
        Dictionary<string, string> dictionary = new Dictionary<string, string>();
        try
        {
            // Зчитуємо всі рядки з файлу
            string[] lines = textAsset.Split('\n');
            // Проходимося по кожному рядку
            foreach (string line in lines)
            {
                // Розбиваємо рядок на ключ і значення
                string[] parts = line.Split('?');
                if (parts.Length >= 2)
                {
                    // Отримуємо ключ (перші два слова у рядку)
                    string key = parts[0];

                    // Отримуємо значення (решта рядка)
                    string value = parts[1];
                    // Додаємо пару ключ-значення у словник
                    dictionary.Add(key, value);
                }
            }
        }
        catch (Exception ex)
        {
            UnityEngine.Debug.Log("Виникла помилка: " + ex.Message);
        }
        return dictionary;
    }
    public Dictionary<string, string> GetLocationType(int LocationValue)
    {
        switch (LocationValue)
        {
            case 1:
            case 7:
            case 13:
                return Abodes;
            case 2:
            case 8:
            case 14:
                return Glades;
            case 3:
            case 9:
            case 15:
                return WaterFeature;
            case 4:
            case 10:
            case 16:
                return OldRuins;
            case 5:
            case 11:
            case 17:
                return OddStones;
            case 6:
            case 12:
            case 18:
                return Assorted;
            default:
                return null;
        }
    }
    public KeyValuePair<string, string>
GetRandomEntryFromDictionary(Dictionary<string, string> dictionary)
```

```
    {
        if (dictionary != null)
        {
            int randomIndex = Random.Range(1,dictionary.Count);
            return dictionary.ElementAt(randomIndex);
        }
        else UnityEngine.Debug.Log("smth wrong with random from dict");
        return default(KeyValuePair<string, string>);
    }
    private void OnValidate()
    {
        ForceGraph.FrameTime = Time.deltaTime;
        ForceGraph.MoveForce = MoveForce;
        ForceGraph.IterationsToStop = Iterations;
        if (_Stop)
        {
            _Stop = false;
            UnityEngine.Debug.Log("Force Graph ran " + (Iterations -
IterationsRemaining));
            IterationsRemaining = -1;
        }
    }
    private void InstantiateRandomNodes(ForceGraph ForceGraph)
    {
        IterationsRemaining = Iterations;
        ForceGraph.Nodes.Clear();
        for (int i = 0; i < InstantiationCount; i++)
        {
            ForceGraph.Nodes.Add(new ForceGraph.Node()
            {
                id = i,
                Position = Random.insideUnitCircle * 1f
            });
        }
        var distrobutionCap = PercentConnectionChance * InstantiationCount;
        for (int i = 0; i < InstantiationCount - 1; i++)
        {
            var nodeI = ForceGraph.Nodes[i];
            if (nodeI.ConnectedNodes.Count >= distrobutionCap) continue;
            for (int j = i + 1; j < InstantiationCount; j++)
            {
                var nodeJ = ForceGraph.Nodes[j];
                if (nodeI.ConnectedNodes.Contains(nodeJ)) continue;
                if (Random.Range(0f, 1f) <= PercentConnectionChance /
nodeI.ConnectedNodes.Count && _MaxConnections > nodeI.ConnectedNodes.Count)
                {
                    nodeI.ConnectedNodes.Add(nodeJ);
                    nodeJ.ConnectedNodes.Add(nodeI);
                }
            }
        }
    }
    private void OnDrawGizmos()//служить для відображення gizmos - підказки про
розташування елементів: тексту, точок і подібного
                               //видно тільки якщо увімкнути gizmos для виду гри
    {
        Gizmos.color = Color.green;
        Gizmos.DrawSphere(Offset, .05f);

        Gizmos.color = new Color(0, 1, 0, 1);
        foreach (var node in ForceGraph.Nodes)
        {
            foreach (var n2 in node.ConnectedNodes)
            {
                Gizmos.DrawLine(node.Position * Scaling + Offset, n2.Position *
Scaling + Offset);
```

```
            }
        }
        Gizmos.color = Color.white;
        foreach (var node in ForceGraph.Nodes)
        {
            Gizmos.DrawSphere(node.Position * Scaling + Offset, .05f);
        }
    }

    #region CheckGraphConnectivity()
    Dictionary<ForceGraph.Node, bool> visited = new Dictionary<ForceGraph.Node,
bool>();
    public void dfs(ForceGraph.Node start, ForceGraph ForceGraph)
    {
        Stack<ForceGraph.Node> stack = new Stack<ForceGraph.Node>();
        stack.Push(start);
        visited[start] = true;
        while (stack.Count > 0)
        {
            ForceGraph.Node node = stack.Pop();
            foreach (var nodeI in ForceGraph.Nodes)
            {
                foreach (var neighbour in node.ConnectedNodes)
                {
                    if (!visited[neighbour])
                    {
                        stack.Push(neighbour);
                        visited[neighbour] = true;
                    }
                }
            }
        }
    }
    public bool Is_connected(ForceGraph ForceGraph)
    {
        dfs(ForceGraph.Nodes.First(), ForceGraph);

        foreach (KeyValuePair<ForceGraph.Node, bool> entry in visited)
        {
            // If any vertex it not visited in any direction
            // Then graph is not connected
            if (entry.Value == false)
                return false;
        }
        return true;
    }
    public bool CheckGraphConnectivity(ForceGraph ForceGraph)
    {
        foreach (ForceGraph.Node node in ForceGraph.Nodes)
        { visited.Add(node, false); };
        if (Is_connected(ForceGraph) == true)
        {
            UnityEngine.Debug.Log("Yes");
            visited.Clear();
            return true;
                }
        else
        {
            UnityEngine.Debug.Log("No");
            visited.Clear();
            return false;
        };
    }
    #endregion
    private void ClearLocations()
    {
```

```
        GameObject[] locations;
        locations = GameObject.FindGameObjectsWithTag("LocationPoint");
        if (locations.Length > 0)
            foreach (GameObject location in locations) { Destroy(location); }
    }
    private void ClearLines()
    {
        GameObject[] lines;
        lines = GameObject.FindGameObjectsWithTag("Line");
        if (lines.Length > 0)
            foreach (GameObject line in lines) { Destroy(line); }
    }
    #region Lines
    private void ConnectionLines(ForceGraph ForceGraph)
    {
        List<ForceGraph.Node> AllNodes = ForceGraph.Nodes;
        List<int[]> NodesConnections = new List<int[]>();
        foreach (ForceGraph.Node nodeI in AllNodes)
        {
            foreach (ForceGraph.Node nodeJ in nodeI.ConnectedNodes)
            {
                NodesConnections.Add(new int[] { nodeI.id, nodeJ.id });
            }
        }
        for (int i = NodesConnections.Count - 1; i >= 0; i = i-1)
        {
            int i1 = NodesConnections[i][0];
            int i2 = NodesConnections[i][1];
            for (int y = 0; y < i; y++)
            {
                int y2 = NodesConnections[y][1];
                if (i1 == y2 && i2 == y1) { NodesConnections.RemoveAt(i);  }
            }
        }
        foreach (int[] NodeCon in NodesConnections)
        {
            GameObject LineCreate = Instantiate(LinePrefab, Vector3.zero,
Quaternion.identity);
            LineScript Ls;
            Ls = LineCreate.GetComponent<LineScript>();
            Ls.LineUpdate(((AllNodes[NodeCon[0]].Position) * Scaling + Offset),
((AllNodes[NodeCon[1]].Position) * Scaling + Offset));
        }
    }
    #endregion
    private void InstantiateLocations(ForceGraph ForceGraph)
    {
        foreach (var node in ForceGraph.Nodes)
        {
            GameObject LocationToInst = Instantiate(LocationPrefab, (node.Position
* Scaling + Offset), Quaternion.identity);
            node.LocationType = node.ConnectedNodes.Sum(nodeIn =>
nodeIn.LocationNumber);
            KeyValuePair<string, string> randomEntry =
GetRandomEntryFromDictionary(GetLocationType(node.LocationType));
LocationToInst.GetComponent<LocationScript>().LocationUpdate(randomEntry.Key,
randomEntry.Value);
        }
    }
    public void Start()
    {
        TextAsset LocDescAbodes =
Resources.Load<TextAsset>("LocationsDescriptions/LocationsDescription_Abodes");
        Abodes = CreateDictionary(LocDescAbodes.text);
        TextAsset LocDescAssorted =
Resources.Load<TextAsset>("LocationsDescriptions/LocationsDescription_Assorted");
```

```
        Assorted = CreateDictionary(LocDescAssorted.text);
        TextAsset LocDescGlades =
Resources.Load<TextAsset>("LocationsDescriptions/LocationsDescription_Glades");
        Glades = CreateDictionary(LocDescGlades.text);
        TextAsset LocDescOddStones =
Resources.Load<TextAsset>("LocationsDescriptions/LocationsDescription_OddStones");
        OddStones = CreateDictionary(LocDescOddStones.text);
        TextAsset LocDescOldRuins =
Resources.Load<TextAsset>("LocationsDescriptions/LocationsDescription_OldRuins");
        OldRuins = CreateDictionary(LocDescOldRuins.text);
        TextAsset LocDescWaterFeature =
Resources.Load<TextAsset>("LocationsDescriptions/LocationsDescription_WaterFeature
");
        WaterFeature = CreateDictionary(LocDescWaterFeature.text);
    }
    public void CreateNew()
    {
        ForceGraph NewGraph = new ForceGraph();
        ClearLocations();
        ClearLines();
        InstantiateRandomNodes(NewGraph);
        if (IterationsRemaining < 0) return;

        IterationsRemaining = -1;
        NewGraph.FullExecution();
        if (CheckGraphConnectivity(NewGraph))
        {
            InstantiateLocations(NewGraph);
            ConnectionLines(NewGraph);
        }
        else CreateNew();
    }
    public void Update()
    {
        InstantiationCount = SliderScript.InstValue;
        if (_InstantiateRandomNodes)
        {
            _InstantiateRandomNodes= false;
            ClearLocations();
            ClearLines();
            InstantiateRandomNodes(ForceGraph);
        }
        if (IterationsRemaining < 0) return;
        var duration = new Stopwatch();
        duration.Start();
            IterationsRemaining = -1;
            ForceGraph.FullExecution();
        if (CheckGraphConnectivity(ForceGraph))
        {
            InstantiateLocations(ForceGraph);
            ConnectionLines(ForceGraph);
        }
        duration.Stop();
        this.IterationTimeMS = (float)duration.ElapsedMilliseconds;
    }
}
```

РЕФЕРАТ

Іван Шило

РОЗРОБКА ДОДАТКУ З АЛГОРИТМОМ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ ДЛЯ РОЗМІЩЕННЯ НА ВЕБ-ПЛАТФОРМІ

Пояснювальна записка: 55 с, 23 рис, 2 додатки, 11 джерел.

Об’єкт дослідження: алгоритм процедурної генерації.

Мета роботи (проекту): розробка програмного додатку, що виконуватиме процедурну генерацію карти та його подальше розміщення на веб-платформі для доступу до нього потенціальних користувачів.

Методи використані при дослідженні: структурний аналіз, створення блок-схем, методи дискретної математики (теорії графів), аналіз тексту.

Розроблено: функціонуючий додаток у стані альфа-тестування

Практичне значення роботи: кінцевий програмний продукт створений для рекреаційної діяльності, досліджені методи можуть бути використані для проектів, що включають в себе використання силового графа для цілей візуалізації взаємозв’язків елементів систем, процедурної генерації задля обмеженої симуляцію реальності; розробка ігор на основі цієї роботи.

Результати здійснених у дипломному проекті досліджень можуть бути використані як приклад створення додатків на ігровому рушії Unity, використання мови програмування C# для створення алгоритму силової візуалізації графів, розробки браузерних та настільних ігор.

Ключові слова: UNITY, ІГРОВИЙ РУШІЙ, ПРОЦЕДУРНА ГЕНЕРАЦІЯ, СИЛОВИЙ ГРАФ, ТЕОРІЯ ГРАФІВ, НАСТІЛЬНІ ІГРИ

ABSTRACT

Ivan Shylo

DEVELOPMENT OF AN APPLICATION WITH A PROCEDURAL GENERATION ALGORITHM FOR PLACEMENT ON WEB PLATFORMS

Explanatory note: 55 p., 23 figures, 2 appendices, 11 sources.

Research object: procedural generation algorithm.

The purpose of the work (project): development of a software application that will perform the procedural generation of the map and its subsequent placement on the web platform for access to it by potential users.

Methods used in the research: structural analysis, creation of block diagrams, methods of discrete mathematics (graph theory), text analysis.

Developed: A working app in alpha testing state

The practical significance of the work: the final software product is created for recreational activities, the researched methods can be used for projects that include the use of a force graph for the purpose of visualizing the interrelationships of system elements, procedural generation for the limited simulation of reality; game development based on this work.

The results of the research carried out in the diploma project can be used as an example of creating applications on the Unity game engine, using the C# programming language to create an algorithm for power visualization of graphs, and developing browser and board games.

Keywords: UNITY, GAME ENGINE, PROCEDURAL GENERATION, POWER GRAPH, GRAPH THEORY, BOARD GAMES