МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

КВАЛІФІКАЦІЙНА МАГІСТЕРСЬКА РОБОТА

на тему:

«Інформаційне та програмне забезпечення ігрової системи з елементами штучного інтелекту»

Завідувач випускаючої кафедри **Довбиш А.С.**

Керівник роботи **Шелехов І.В.**

Студентка групи ІН.мдн-91С **Череповська С.М.**

СУМИ 2020

Сумський державний університет
(назва вузу)

Факультет *ЦЗДВН* Кафедра *Комп'ютерних наук.*
Спеціальність *122 «Комп'ютерні науки»*

Затверджую:
зав. кафедри _____

"_____" _____ 20___р.

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЕКТ (РОБОТУ) СТУДЕНТОВІ

Череповської Соф'ї Миколаївни
(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) *Інформаційне та програмне забезпечення ігрової системи з елементами штучного інтелекту*
затверджую наказом по інституту від "____" _____ 20 ___ р. № _____

2. Термін здачі студентом закінченого проекту (роботи) _____

3. Вхідні данні до проекту (роботи) _____
_____
_____
_____

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити) *1) Аналіз проблеми та постановка задачі. 2) Проектування комп'ютерної гри. 3) Інформаційне та програмне забезпечення*

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) _____
_____
_____

6. Консультанти до проекту (роботи), із значенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання _____

Керівник

_____

(підпис)

Завдання прийняв до виконання _____

(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів дипломного проекту (роботи)	Термін виконання проекту (роботи)	Примітка
1	*Аналіз проблеми та постановка задачі*		
2	*Проектування комп'ютерної гри*		
3	*Інформаційне та програмне забезпечення*		
4	*Оформлення кваліфікаційної магістерської роботи*		

Студент – дипломник _____

(підпис)

Керівник проекту ____ _____

(підпис

РЕФЕРАТ

Записка: 44 стор., 27 рис., 1 додаток, 15 джерел.

Об’єкт дослідження — процес проектування та реалізації інформаційного та програмного забезпечення ігрової системи

Мета роботи — розробка та програмна реалізація ігрової системи.

Методи дослідження — методи проектування ігрових систем, методи процедурної генерації.

Результати — була створена ігрова система. В роботі було проведено аналіз найліпших ігрових рушіїв, представлено плюси та мінуси їх використання. Також був проведений аналіз можливих алгоритмів генерування ігрового середовища. На основі цього аналізу було розроблено ігрову концепцію, створено ігрову логіку та правила, спроектовано ігрові механіки та прототип дизайну гри. Для генерування ігрового середовища був створений власний алгоритм на основі алгоритму випадкових чисел. Для реалізації проекту було використано середовище розробки Unity. Для перевірки валідності ігрової системи було проведено закрите Альфа та Бета - тестування.

ІГРОВИЙ РУШІЙ, СЕРЕДОВИЩЕ РОЗРОБКИ, ШТУЧНИЙ ІНТЕЛЕКТ,
НЕЙРОННА МЕРЕЖА, КОМП’ЮТЕРНА ГРА, UNITY

ЗМІСТ

ВСТУП

Роками відеоігри критикували за те, що вони роблять людей більш асоціальними, малорухомими або депресивними. Але зараз дослідники виявляють, що ігри насправді можуть змінити нас на краще та покращити як наше тіло, так і розум.

Ігри можуть допомогти розвинути фізичні навички. Діти дошкільного віку, які грали в інтерактивні ігри показали, що вони покращили моторику, наприклад, діти, які захоплюються відеоіграми стали краще бити, ловити та кидати м'яч, ніж діти, які в них не грають. Дослідження хірургів, які займаються мікрохірургією в Бостоні, показало, що ті, хто грав у відеоігри, були на 27 відсотків швидшими і допускали на 37 відсотків менше помилок, ніж ті, хто цього не робив. Гострота зору також поліпшується, особливо це свідчить про різницю між відтінками сірого[1].

Ігри також корисні для покращення функцій мозку, включаючи прийняття рішень. За даними одного дослідження, люди, які грають в ігри на основі дій, приймають рішення на 25 відсотків швидше за інших і не менш точні. Також було встановлено, що найкращі геймери можуть робити вибір і діяти на них до шести разів на секунду, в чотири рази швидше, ніж більшість людей. В іншому дослідженні з Університету Рочестера в Нью-Йорку досвідченим геймерам було показано, що вони можуть звертати увагу на більш ніж шість речей одночасно, не заплутуючись, порівняно з тими чотирма, які більшість людей зазвичай можуть мати на увазі. Крім того, відеоігри здатні згладжувати гендерні відмінності. Вчені з’ясували, що жінки, які грають в ігри, краще здатні подумки маніпулювати 3D-об’єктами.

Є також дані, що ігри можуть допомогти у вирішенні психологічних проблем. В Оклендському університеті в Новій Зеландії дослідники попросили 94 молодих людей з діагнозом депресія пограти у 3D-фантастичну гру SPARX, і в багатьох випадках ця гра зменшила симптоми депресії більше, ніж звичайне лікування[11]. Інша дослідницька група з Оксфордського

університету виявила, що гра в Тетріс незабаром після впливу чогось дуже засмучуючого - в експерименті був використаний фільм травматичних сцен поранення та смерті - насправді може запобігти людям, що викликають тривожні спалахи.

В рамках даної роботи буде описано створення гри під назвою “CapsuleShooter” на движку Unity. Даний продукт буде створений на базі ОС Windows з можливістю портувати проект на інші платформи.

1 АНАЛІЗ ПРОБЛЕМИ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Огляд ігрових рушіїв

Ідея ігрового рушія - проста і доступна розумінню людини полягає в абстрагуванні деталей виконання спільних ігрових задач таких як фізика, введення даних та рендеринг, що дає можливість розробникам зосередитися на деталях власних ігор, які надають їм унікальності. [2]

Ігровий рушій є цілим комплексом прикладних програм, до якого входять фізичний рушій, візуалізатор для 2D або 3D графіки, звук, скриптинг, графічні сцени, анімація, штучний інтелект, управління пам'яттю і графічні сцени. Ігровий процес визначається функціями, реалізованими в програмах написаними програмістами.[3]

Unity і Unreal Engine посіли нішу найкращих та найпопулярніших ігрових рушіїв, архітектура рендерингу яких дає розробникам можливість відтворити приголомшливі візуальні ефекти в проектах, які зможуть працювати навіть на пристроях з невисокою продуктивністю. [4]

Unreal Engine 4 і Unity3d є відносно безкоштовними. В Unity3d ви маєте можливість придбати платну версію та користуватися безкоштовною, але за умови, що в кожному додатку або грі логотип Unity буде відображений при кожному відкритті. Також Unity претендує на 10% від прибутку гри, якщо він складає більше ніж 100 тисяч доларів. Unreal Engine 4 є безоплатною доки дохід від вашого продукту не перевищує три тисячі доларів, проте у разі прибутковості проекту 5% від прибутку буде належати їм.[5]

Кожен з рушіїв має свою власну крамницю додатків, де можна знайти готові 3d моделі персонажів і оточення, текстури, а також звуки і системи частинок. [2] Перевага Unity в наявності більшої кількості ассетів тож для новачків вона зручніша.

У Unity3d пишуть мовою програмування C # або JavaScript, а в Unreal Engine 4 користуються C ++. [2]

Unity - кросплатформенне середовище розробки комп'ютерних ігор, що дає можливість створювати додатки в понад 20 операційних системах.

Редактор Unity має простий Drag & Drop інтерфейс, який легко налаштовувати, він складається з різних вікон і дозволяє здійснювати налаштування гри прямо в редакторі. Для 2D-розробки в Unity можна користуватися такими інструменти як sprite creator, sprite editor і sprite packer, а також окремий фізичний рушій для 2D-об'єктів.

Unity має дві основні переваги, що відрізняє його від інших ігрових рушіїв: наявність візуального середовища розробки та кросплатформенна підтримка. Перший фактор окрім інструментарію візуального моделювання має інтегроване середовище, лінію складання, що направлено на підвищення продуктивності розробників, зокрема, етапів створення прототипів і тестування.

У Unity Asset Store є Playmaker, завдяки якому розробники без знання програмування мають можливість спроектувати власну гру. Останнім часом розробники рушія активно розвиваються в цій сфері. [3]

В Unity містяться інтуїтивно зрозумілий інструментарій, магазин готових ассетів та безкоштовна версія без явних обмежень, що робить його унікальним середовищем розробки для початківців.

1.2 Кросплатформне середовище розробки Unity

Unity має широкий набір інструментів для розробки ігор, що націлений не тільки на безпосередньо розробку, а й на покращення та оптимізацію вже готового продукту:

- Редактор "все-в-одному": доступний у Windows, Linux і Mac, він включає в себе ряд практичних для виконавців інструментів для проектування ігрових середовищ, а також набір інструментів для розробки логіки гри та високопродуктивного геймплея. [6]

• Ефективні робочі процеси: Unity Prefabs, які є заздалегідь налаштованими ігровими об'єктами, що дозволяють вам працювати впевнено, не думаючи про тимчасові помилки.

• Фізичні рушії: існує можливість використання переваг Box2D, нової системи фізики DOTS і підтримкою NVIDIA PhysX для реалістичного і високопродуктивного ігрового процесу.

• 2D & 3D: Unity підтримує як 2D, так і 3D-розробку з функціями та функціональністю для конкретних потреб у різних жанрах.

• Інтерфейси користувача: Вбудована система користувальницького інтерфейсу дає можливість швидко і інтуїтивно створювати інтерфейси користувача.

• ШІ для пошуку шляху (AI pathfinding tools) : Unity включає в себе навігаційну систему для створення NPC, які можуть розумно переміщатися всередині ігровому простору. Система використовує навігаційні сітки, які створюються автоматично з геометрії сцени, або навіть динамічні перешкоди, щоб змінити навігацію персонажів під час виконання

• Спеціальні інструменти: Редактор можна розширити за допомогою інструментів так, щоб він відповідав вимогам робочого процесу. Можна створювати та додавати спеціальні розширення або знайти те, чого не вистачає, у магазині активів, який містить тисячі ресурсів, інструментів і розширень для прискорення ваших проектів.

• Підтримка більшої кількості платформ у порівнянні з іншим рушієм.

• Нова високомодульна програма Unity дозволяє створювати миттєві ігри, які є малими, легкими та швидкими. Миттєвий доступ до контенту є суттєво важливим для максимального охоплення аудиторії на мобільних пристроях.

• Інструменти вдосконаленого профілювання: це дає можливість постійно оптимізувати вміст гри у процесі розробки з функціями профілювання Unity. Доступна перевірка, чи вміст, наприклад, пов'язаний з

процесором або графічним процесором. Це визначає ті області, які потребують покращення, щоб надати аудиторії безперебійний досвід роботи.

• Дизайнерські інструменти: Unity є творчим осередком для художників, дизайнерів, розробників та інших членів команди. Він включає в себе 2D та 3D засоби проектування сцени, оповідання історії та кінематографії, освітлення, аудіосистем, інструменти управління спрайтами, ефекти частинок та потужну система анімації персонажів.

• Ігровий сервер хостингу для Multiplay: масштабовані і стійкі хостинг-рішення для ігрових серверів. Multiplay підтримує студії, такі як Respawn Entertainment і PUBG Corp., надаючи найкращий досвід для деяких з найбільших ігор на планеті.

• Онлайн ігри: від динамічних одиночних ігор до багатокористувацьких ігор в реальному часі і поза ними, онлайн ігри є найбільш популярними і успішними. Unity має набір інструментів, служби рушіїв і інфраструктуру, необхідні для створення цих видів ігор, і масштабування до будь-якого рівня успіху.

• Командна співпраця. Unity Teams дають змогу творчим групам працювати більш ефективно разом із функціями, які дозволяють співпрацювати та спрощувати робочі процеси.

• Хмарна діагностика. Це набір інструментів, які підтримують хмару та допомагають ідентифікувати, збирати та встановлювати пріоритети даних про продуктивність та відгуки кінцевих користувачів.

1.3 Порівняння процедурного генерування та ручного дизайну

Переваги процедурної генерації[8]:

1. Можна легко масштабувати карти / проекти до дійсно великих розмірів, набагато більших, ніж можливо було б створити власноруч.
2. Створюючи систему, в якій шматки ландшафту створюються “на льоту”, можна уникнути необхідності писати фрагменти коду для завантаження шматками з постійної пам'яті.

3. У довгостроковій перспективі можливо знайти більш життєздатні рівні, використовуючи процедурний дизайн, ніж при ручному редагуванні карти.
4. Різнобарвність та непередбачуваність є плюсом для майбутнього користування. Користувач не занудьгує на згенерованій карті на відміну від тієї, яка була створена заздалегідь.

Недоліки процедурного покоління:

1. Потрібно багато роботи, щоб гарантувати, що згенерований ландшафт мав пристойний вигляд.
2. Створення «тестових рівнів» стає проблемою.
3. Розміщення предметів і інших подібних речей має бути зроблено обережно, хоча для гри згори до низу це, ймовірно, буде меншою проблемою.
4. Якщо не буде проведено велике тестування, можна отримати сценарій, в якому ігровий процес повністю переривається з певним процедурним рівнем, в більшій мірі, ніж якби було зроблено рівень вручну.
5. Для дійсно складних ландшафтів[9] витрачається більше часу на виправлення помилок і роботу з проектування, ніж на створення базового інструменту для розробки карти і завантажувача файлів.

Переваги ручного дизайну

1. Розробник гри може бути більш впевнений, що ігровий процес функціонує так як і повинно бути в контексті кожного рівня.
2. Можна створювати і розміщувати «дрібні деталі» простіше, ніж за допомогою процедурної генерації.
3. Деякі особливості рельєфу місцевості важко отримати процедурно, наприклад, вода, що стікає з гори. Це зажадає дуже ретельної оцінки карти висот, яка може бути важким для розробки.
4. Простіше направити гравця «лінійним шляхом».

Недоліки ручного дизайну

1. Застосування широко поширених змін у місцевості буде займати багато часу, в той час як при процедурній генерації просто зміна декількох значень призведе до створення зовсім іншої місцевості.
2. Вам доведеться зберігати кожну карту в пам'яті, що може бути прийнято для невеликих рівнів, але для великих рівнів процедурна генерація може бути єдиним варіантом. Збільшує постійні вимоги до пам'яті для гри.
3. Може знадобитися створення редактора карт і системи завантаження файлів.

1.4 Огляд алгоритмів процедурної генерації

Процедурна генерація включає в себе безліч генеративних алгоритмів, принцип роботи яких полягає в створенні даних не вручну, а алгоритмічно: замість ручного виготовлення пишеться алгоритм, який успішно може створювати різні приклади без багаторазового виконання того ж процесу. Особливо корисний такий підхід у відеоіграх, де випадковим чином може генеруватися ціла карта або рівень.

1. Алгоритм колапсу хвильової функції[12]

Алгоритм вирішує, які модулі підбирати в кожну клітинку ігрового світу. Масив осередків вважається хвильовою функцією в неспостережуваному вигляді. Таким чином, кожному осередку відповідає безліч модулів, які можуть в ній опинитися. У термінах квантової механіки можна було б сказати, «осередок знаходиться в суперпозиції всіх модулів». Існування світу починається в повністю неспостережуваному вигляді, де в кожному осередку може перебувати будь-модуль. Далі всі осередки з’єднуються, одна за одною. Це означає, що для кожного осередку випадковим чином вибирається по одному модулю з усіх можливих.

Після цього йде етап поширення обмежень (constraint propagation). Для кожного модуля підбирається таке підмножина модулів, яким дозволено бути суміжними з ним. Всякий раз при з’єднанні модуля оновлюються підмножини інших модулів, які як і раніше допускаються в якості суміжних йому. Етап

поширення обмежень – найбільш ресурсовитратна частина алгоритму з точки зору обчислювальної потужності.

Важливий аспект алгоритму полягає у визначенні того, яку комірку з'єднувати. Алгоритм завжди з'єднує осередок з найменшою ентропією. Це осередок, який припускає мінімальну кількість варіантів вибору (осередок з найменшою хаотичністю). Якщо у всіх модулів ймовірність з'єднання однакова, то найменша ентропія буде у тому осередку, який відповідає мінімальній кількості можливих модулів. Як правило, ймовірності потрапити під вибір для різних наявних модулів відрізняються. Осередок з двома можливими модулями, що мають однакову ймовірність, передбачає більш широкий вибір (більшу ентропію), ніж той, в якому два модуля, і для одного з них ймовірність потрапити під вибір дуже велика, а для іншого - дуже мала.

Як працює алгоритм колапсу хвильової функції можна побачити на малюнку (1.1)

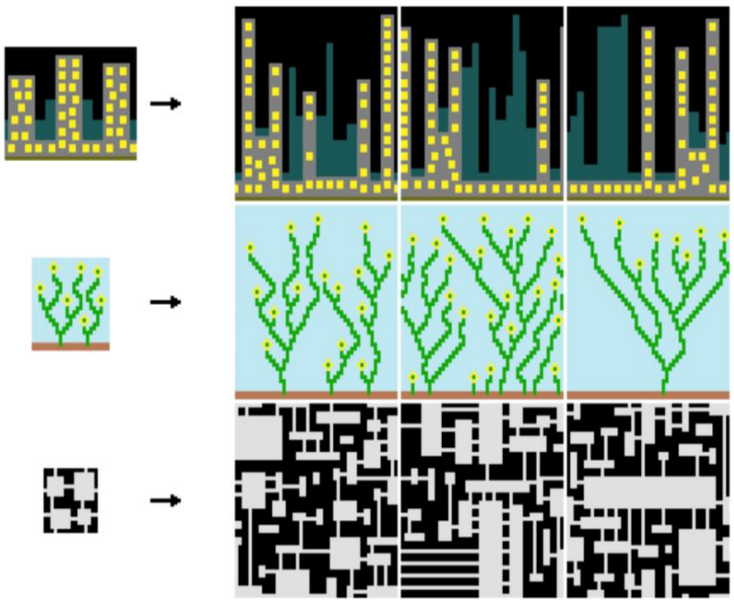

Рисунок1.1 – Алгоритм колапсу хвильової функції

2. Ланцюг Маркова[13]

Ланцюг Маркова - інструмент з теорії випадкових процесів, що складається з послідовності n кількості станів. Зв'язки між вузлами

(значеннями) ланцюжка при цьому створюються, тільки якщо стани стоять строго поруч один з одним.

Імовірність настання деякої нового стану в ланцюжку залежить тільки від справжнього стану і математично не враховує досвід минулих станів => Марковська ланцюг - це ланцюг без пам'яті.

Інакше кажучи, нове значення завжди “танцює” від того, яке його безпосередньо “тримає за ручку”.

Формально, ланцюг Маркова - це імовірнісний автомат. Розподіл ймовірностей переходів зазвичай представляється у вигляді матриці. Якщо ланцюг Маркова має N можливих станів, то матриця буде мати вигляд N x N, в якій запис (I, J) буде ймовірністю переходу зі стану I у стан J. Крім того, така матриця повинна бути стохастичною, тобто рядки або стовпці в сумі повинні давати одиницю. У такій матриці кожний рядок буде мати власне розподіл ймовірностей.

3. Алгоритм Diamond-Square[10]

Для Diamond-Square потрібно 2D-масив розміром $2^n + 1$ (5x5, 17x17, 33x33 і т. д.). Він починається з попередньо заповнених значень в чотирьох кутах масиву. Потім він циклічно перебирає поступово дедалі менші розміри кроків, виконуючи "алмазний крок", а потім "квадратний крок", поки кожне значення в масиві не буде встановлено.

Алмазний крок бере квадрат, знаходить середню точку і встановлює середню точку в середнє з чотирьох кутів плюс випадкове значення в деякому діапазоні. Уявіть собі, що ви малюєте лінії від чотирьох точок до середини для кожного квадрата в масиві: ви створите ромбоподібний візерунок (звідси і назва!).

Квадратний крок бере “алмаз”, знаходить середню точку і витягує середнє зі значень точок, що утворюють Diamond-Square (плюс випадкове значення).

4. Алгоритм Marching squares [14]

На вхід алгоритм отримує регулярну сітку, в кожному вузлі якої відомо значення поля. Вихідна сітка (на малюнку позначена синім кольором) може мати меншу роздільну здатність (в цьому випадку втрачається точність, але зменшується ступінчастість). Далі для кожного вузла вихідний сітки перевіряється, чи вище значення в ньому, ніж на ізоповерхні. Всім вузлам, які вище, приписується "+", іншим "-". Далі розглядаються квадратики вихідний сітки, вершини яких лежать в зазначених вузлах. Всього виходить 16 різних випадків, які з урахуванням симетрій і поворотів можна звести до чотирьох:

- Випадок 1: всі вершини мають один знак;
- Випадок 2: в однієї вершини знак відрізняється;
- Випадок 3: вершини з однаковими знаками мають загальне ребро;
- Випадок 4: вершини з однаковими знаками не мають загального ребра.

У четвертому випадку неможливо однозначно визначити форму сегмента ізолінії, тому додатково проглядається значення в центрі квадрата (якщо вхідні дані це дозволяють). При неможливості дізнатися значення в центрі квадрата прийняте рішення може вплинути на зв'язність ізолінії.

На рисунку 1.2 зображене схематичне зображення алгоритму: колір квадрата позначає значення в даній клітині регулярної сітки, чим темніше, тим значення ближче до ізолініях. Червоним показані отримані ізолінії.

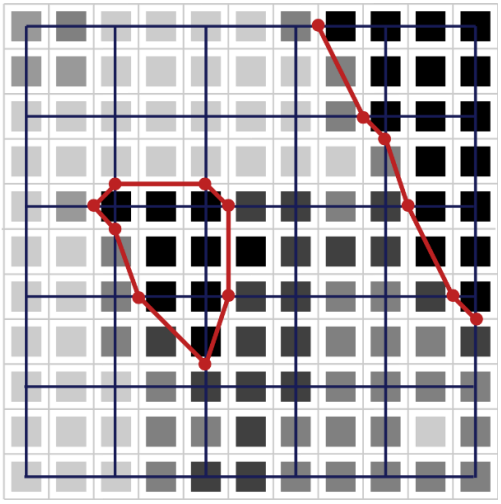

Рисунок1.2 – Алгоритм Marching squares

5. Алгоритм Perlin noise

Перліновий шум - це процедурний примітив текстури, тип градієнтного шуму, який використовується художниками візуальних ефектів для збільшення видимості реалізму в комп’ютерній графіці.[7] Функція має псевдовипадковий вигляд, але всі її візуальні деталі однакового розміру. Ця властивість дозволяє йому легко керувати; кілька математичних копій шуму Перліна можна вставити в математичні вирази, щоб створити велику різноманітність процедурних текстур[15].

Синтетичні текстури з використанням шуму Перліна часто використовуються в CGI, щоб створити візуальні елементи, створені комп’ютером, наприклад, поверхні об’єктів, вогонь, дим або хмари - більш природними, імітуючи контрольований випадковий вигляд текстур у природі. Він також часто використовується для створення текстур, коли пам’ять надзвичайно обмежена, наприклад, в демонстраційних версіях. Його наступники, такі як фрактальний шум і симплексний шум, стали майже всюдисущими в модулях обробки графіки як для графіки в режимі реального часу, так і для процедурних текстур, що не в реальному часі, у всіх видах комп'ютерної графіки.

Шум Перліна найчастіше реалізується як дво-, три- або чотиривимірна функція, але може бути визначений для будь-якої кількості вимірів. Реалізація зазвичай включає три етапи: визначення сітки випадкових векторів градієнта, обчислення точкового добутку між градієнтними векторами та їх зміщеннями та інтерполяція між цими значеннями.

Визначте n-мірну сітку(Рисунок 1.3), де кожен перетин сітки пов'язаний з нею фіксованим випадковим n-мірним одиничним вектором градієнта, за винятком одновимірного випадку, коли градієнти є випадковими скалярами від -1 до 1.

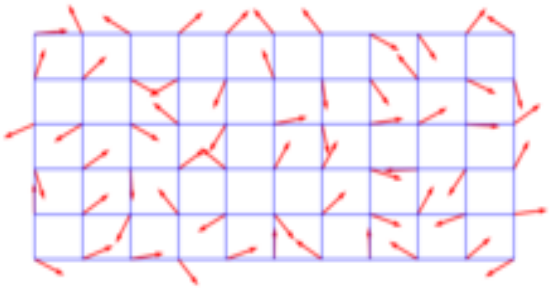

Рисунок1.3 – Двовимірна сітка градієнтних векторів

Для опрацювання значення будь-якої точки-кандидата спочатку розташована унікальна комірка сітки, в якій знаходиться точка. Потім ідентифікуються 2^n кути цієї комірки та пов'язані з ними градієнтні вектори. Далі для кожного кута обчислюється вектор зміщення, який є вектором переміщення від точки кандидата до цього кута.

Для кожного кута ми беремо точковий добуток(рисунок 1.4) між його вектором градієнта та вектором зміщення до точки кандидата. Цей точковий добуток дорівнюватиме нулю, якщо точка-кандидат знаходиться точно в куті сітки.

Для точки в двовимірній сітці для цього потрібно буде обчислити 4 вектори зсуву та крапкові добутки, тоді як у трьох вимірах - 8 зміщених векторів та 8 крапкових добутків. Загалом, алгоритм має масштабування складності O (2^n)

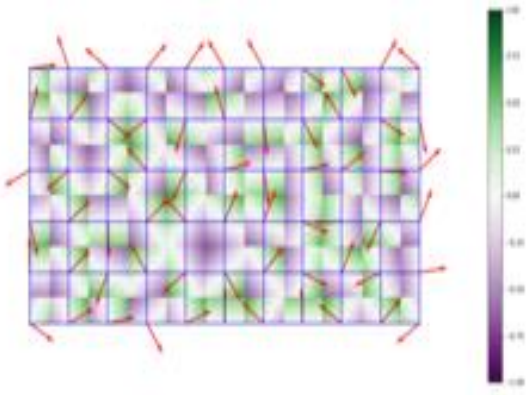

Рисунок 1.4 – Точковий добуток кожної точки з найближчим значенням градієнта вузла сітки.

Останній крок - інтерполяція між крапковими продуктами 2^n Інтерполяція виконується за допомогою функції, яка має нульову першу похідну (а можливо, також другу похідну) у вузлах сітки 2^n. Отже, у точках, близьких до вузлів сітки, на виході буде наближено точковий добуток вектора градієнта вузла та вектора зміщення до вузла. Це означає, що шумова функція буде проходити через нуль на кожному вузлі, надаючи шуму Перліна характерний вигляд.

Якщо n = 1, приклад функції, яка інтерполює значення a_0 у вузлі сітки 0 та значення a_1 у вузлі сітки 1 є

$$f(x) = a_0 + smoothstep(x)(a_1 - a_0)\ for\ 0 \leq x \leq 1$$

де була використана функція плавного кроку.

Остаточний інтерпольований результат зображений на рисунку 1.5.

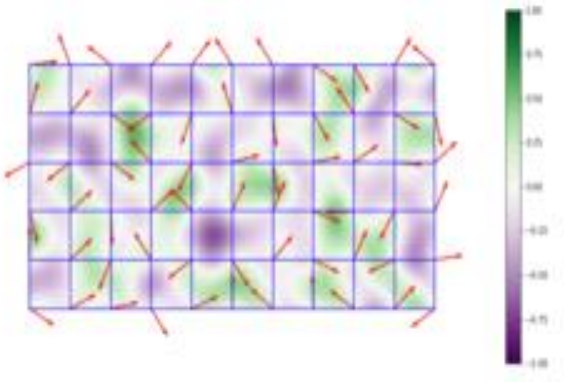

Рисунок 1.5 – Остаточний інтерпольований результат.

Функції шуму для використання в комп’ютерній графіці зазвичай видають значення в діапазоні [-1,0,1,0] і можуть бути відповідно масштабовані.

1.5 Постановка задачі

Метою роботи є програмна реалізація кооперативної ігрової системи для тренування штучного інтелекту. Для досягнення поставленої мети необхідно виконати такі завдання:

1. Розробка ігрової концепції:
   a) створення ігрової логіки та правил;
   b) проектування ігрових механік;
   c) створення дизайну гри.
2. Прототипування ігрового середовища:
   a) розробка алгоритмів генерування ігрового середовища;
   b) розробка алгоритмів оцінки ігрового процесу.
3. Програмна реалізація ігрової системи:

a) вибір програмного середовища;
b) імплементація ігрової логіки і правил;
c) програмна реалізація алгоритмів генерування ігрового середовища і оцінки ігрового процесу;
d) реалізація дизайну ігрового оточення;
e) реалізація штучного інтелекту.

4. Тестування ігрової системи.

2 ПРОЕКТУВАННЯ КОМП'ЮТЕРНОЇ ГРИ

2.1 Алгоритм генерування середовища

Для виконання проекту був створений власний алгоритм, який був построєний на базі алгоритму генерування випадкових чисел. Цей алгоритм є досить ресурсозатратним, але це не є пріоритетом даної роботи. Плюсом є його легкість у проектуванні та швидкість дії.

Основний принцип роботи полягає в рандомізації чисел з подальшою побудовою перешкод і перевірок на замкнуті простору. Далі буде представлено основні моменти генерації з прикладами.

На початку створюється сітка(поле) в даному випадку 5X5 для візуалізації пронумеровану(рисунок 2.1).

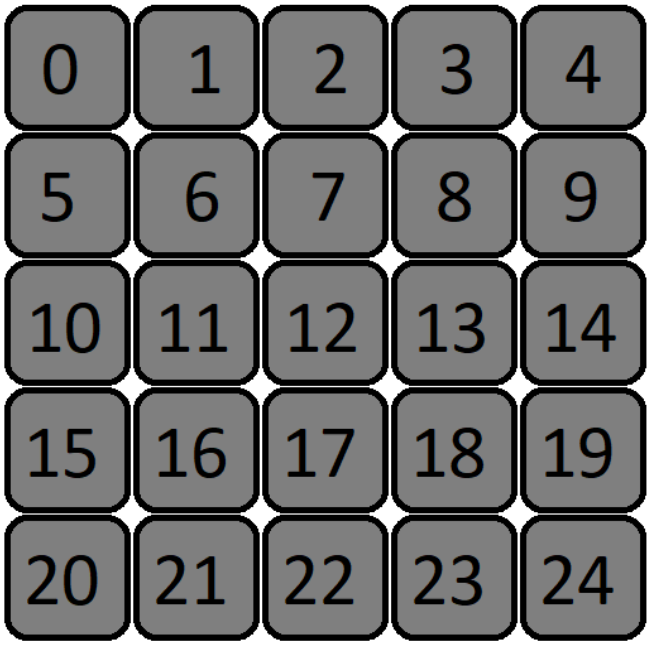

Рисунок 2.1 –Початкова сітка

Числа комірок записуються в стек після чого перемішуються, за допомогою алгоритму випадкових чисел (рисунок 2.2).

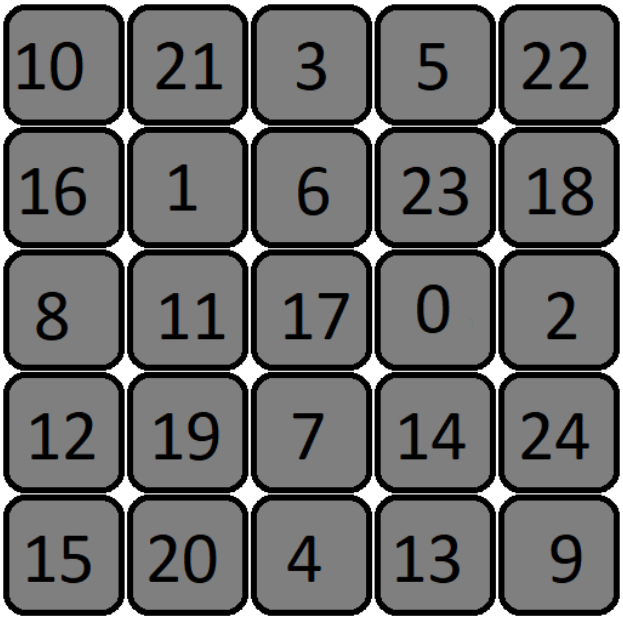

Рисунок 2.2 –Сітка після алгоритму випадкових чисел

Далі починаючи з 0-ї комірки створюється перешкода(рисунок 2.3).

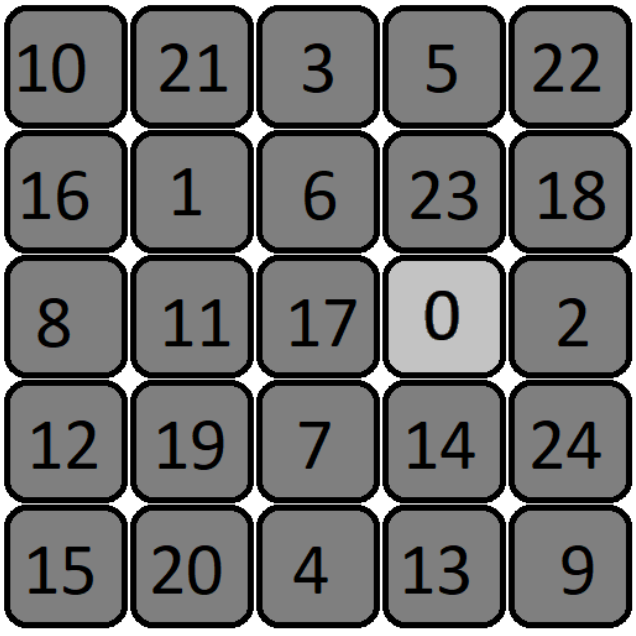

Рисунок 2.3 –Перша перешкода

Перед кожною спробою поставити перешкоду, карта перевіряється на наявність недосяжних місць(рисунок 2.4).

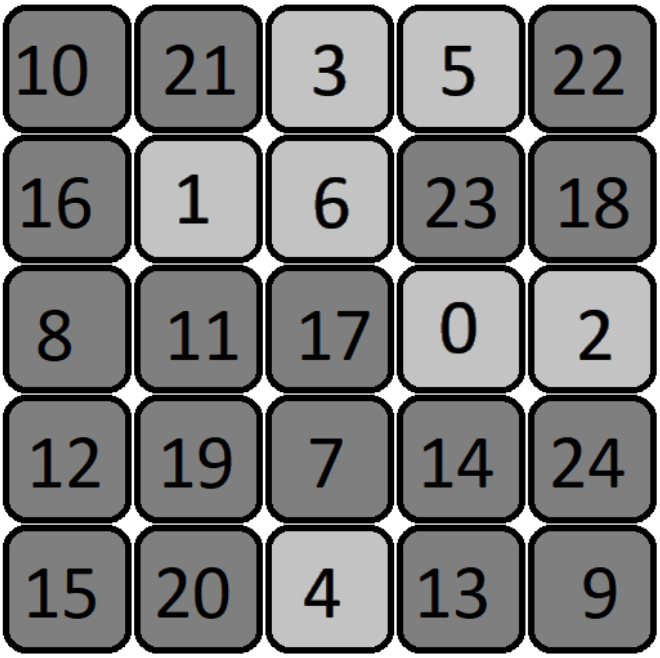

Рисунок 2.4 –Перша замкнута область

Перевірка проходить починаючи з центру, помічаючи, які з комірок є пустими. Якщо у кінці підрахунку різниця між загальною кількістю та кількістю вільних комірок не дорівнює сумі вже поставлених комірок, це означає, що є недосяжне місце і комірка не може бути перешкодою.

В кінці виходить лабіринт так як кількість перешкод була максимальною (рисунок 2.5)

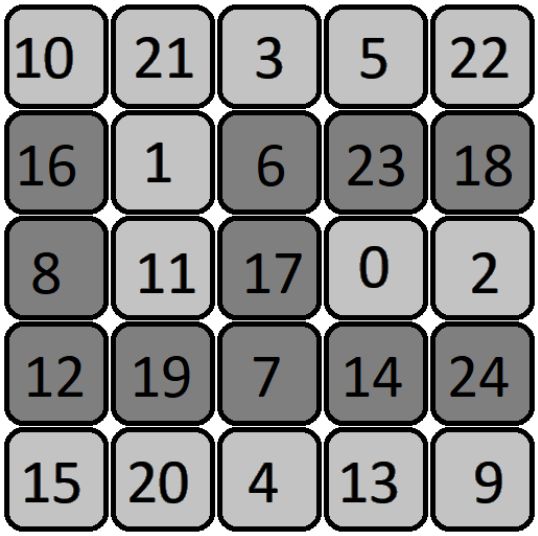

Рисунок 2.5 –Готовий лабіринт

Також кількість перешкод можна контролювати в процентному еквіваленті. Наприклад, якщо процент перешкод визначити на 40% завантаженість лабіринту буде не такою явною, як це видно на рисунку 2.6.

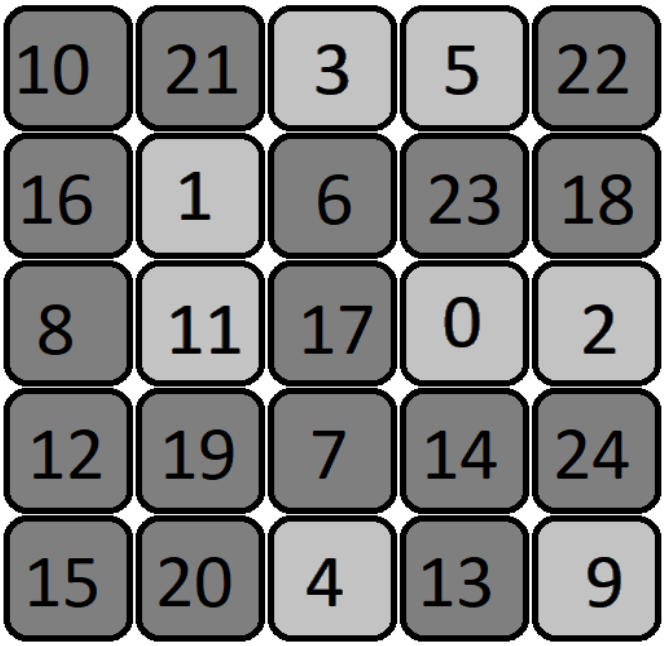

Рисунок 2.5 –Готовий лабіринт з завантаженістю на 40%.

2.2 Штучний інтелект ігрової системи

В цьому проекті був доданий штучний інтелект, завдяки якому на гравця будуть нападати вороги. В ігровій системі ворожі елементи будуть йти до гравця по найкоротшому вибраному шляху, поки не доберуться до нього, тим самим завдавши шкоди.

Пошук шляху – це класичне завдання для штучного інтелекту. «Академічними» варіантами пошуку шляху є алгоритм Дійкстра і алгоритм А, і вони є, по суті, рішенням оптимізаційної задачі на графах. Алгоритм Дійкстра надійний але повільний, алгоритм А швидший, але жадібний до пам'яті. І той і інший добре працюють на маленьких масштабах, але з ростом кількості вершин графа починають стрімко гальмувати, тому в іграх вони застосовуються в сильно модифікованому вигляд. Крім них іноді

застосовуються хвильової алгоритм, банальний «обхід перешкод», і численний комбінації з них.

В випадку поточного проекту найліпшим варіантом для вибору найкоротшого шляху була вибрана система NavMesh, яка вбудована в Unity.

Приклад роботи NavMesh-у представлений на рисунку 2.6

Рисунок 2.6 – Вибраний шлях до точки.

Алгоритм роботи NavMesh, якщо Q = набір вузлів для пошуку; S = початковий стан пошуку:

1. Ініціалізує Q, використовуючи вузол (S) як єдиний вхід;
2. Якщо Q порожній, зупиніть пошук. В іншому випадку виберіть найкращий елемент Q;
3. Якщо стан (n) є ціллю, поверніть n;
4. (Інакше) Видаляє n з Q;
5. Знаходить нащадків стану (n), які ще не перевірені, і створює розширені шляхи до n від кожного нащадка;
6. Додає всі розширені шляхи до Q і повертається до кроку 2;

2.3 Штучна нейронна мережа бота

Нейронна мережа - це низка алгоритмів, які намагаються розпізнати основні взаємозв'язки в наборі даних за допомогою процесу, що імітує роботу

людського мозку. У цьому сенсі нейронні мережі відносяться до систем нейронів, органічних або штучних за своєю природою.

Для навчання бота було використано Unity ML-Agents.

Алгоритм роботи машинного навчання. бот аналізує навколишній світ, пускаючи 12 променів в радіусі 100 юнітів. Вхідні дані можуть бути 3 типів: ворог, порожнеча або перешкода. Далі дані посилаються на ваги, де вони аналізують інформацію і в залежності від кількості прогонів, вибирають все більш точні дані. На виході вийдуть команда, куди боту треба йти, стріляти і чи стріляти взагалі.

Для керуванням персонажем гравця, будемо використовувати повнозв'язну нейронну мережу з трьома шарами. Топологія 12-12-12-6.

Число вхідних нейронів - 12, число вихідних – 6. Візуалізація представлена у вигляді діаграми на рисунку 2.7.

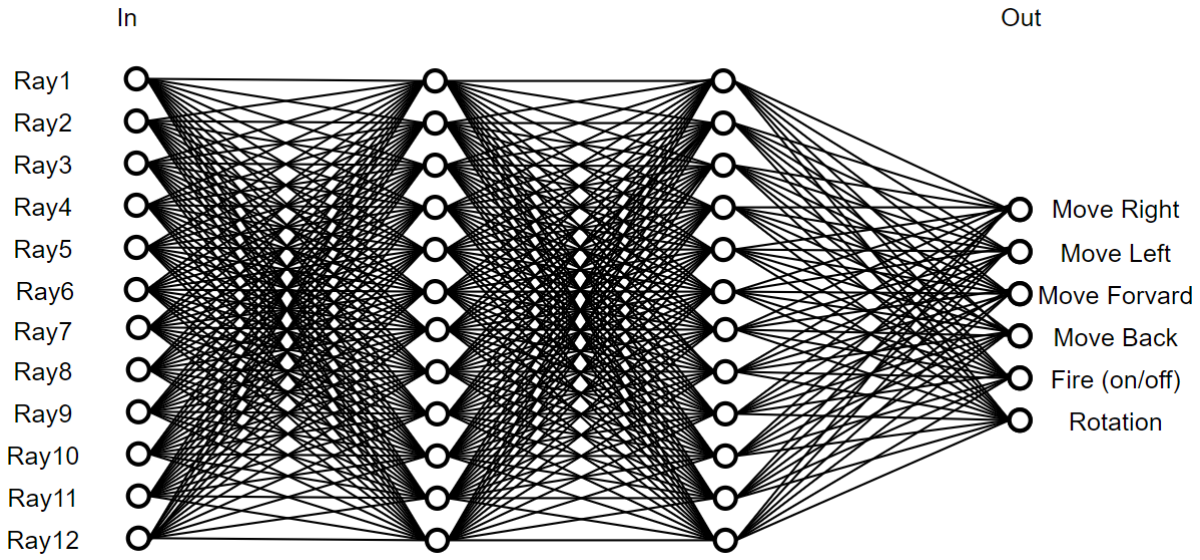

Рисунок 2.7 – Діаграма нейронної мережі.

Unity ML-Agents контролює вхідні та вихідні дані в залежності від того, як на реагує середовище(рисунок 2.8).

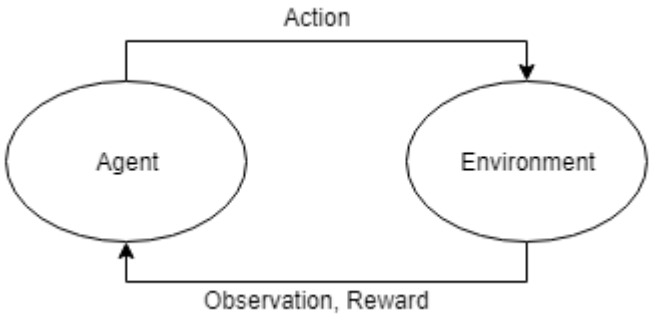

Рисунок 2.8 – Діаграма взаємодії агента та середовища.

Навчання нейронної мережі відбувається за допомогою генетичного алгоритму.

Алгоритм полягає в генерованих наборах різних випадкових нейронних мереж і тестуванні їх. Відбувається оцінка ефективності кожної мережі за допомогою якого-небудь визнання, наприклад, за якийсь час підтримувався бот на арені, до того, як його знищили вороги.

Далі виникають процеси відбору найкращих ботів. Із них формується нова популяція, ваги зв'язків у нейронах випадковим чином змінюються (за допомогою мутацій та рекомбінації генів). Нова популяція знову тестується в новій арені та процес повторюється.

У результаті, з кожним новим поколінням відбираються найкращі боти, а еволюційний процес дає всі найкращі та найкращі результати, які будуть показувати популяризацію ботів.

Кожне нове покоління ботів формується за допомогою трьох процедур - селекція, рекомбінації та мутації. Детальніше про них:

- Селекція

Завдання селекції - відібрати кращі генотипи з популяції і створити на їх основі нову популяцію такого ж розміру як і вихідна. Для відбору використовується метод, який називається Remainder Stochastic Sampling.

Це відбувається так: після того як пройшов процес моделювання, для всіх особин задається оцінка їх успішності. Далі підраховується середня оцінка по популяції. Потім їх популяції відбираються тільки ті особини, у яких

оцінка вище середньої. І далі ці особини клонуються, при чому число клонів - пропорційно оцінці батька. Підсумкове число популяції при цьому робиться рівним розміру вихідної популяції. Таким чином ми відбираємо і розмножуємо кращі генотипи, і при цьому зберігаємо розмір популяції.

- Мутація

Це процес в якому з популяції вибирається певне число НС і в них випадкові зв'язки змінюються на випадкову невелику величину. В результаті ми отримуємо нову НС з трохи іншими вагами(рисунок 2.9):

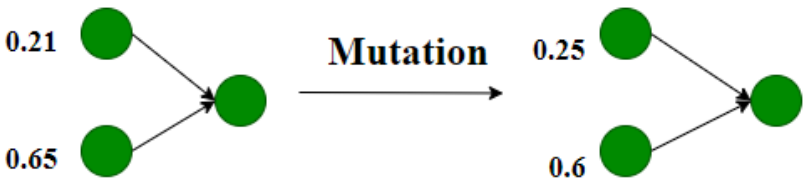

Рисунок 2.9 – Діаграма мутації.

- Рекомбінація

Це процес обміну генами (тобто вагами нейронних зв'язків) між двома ботами. Для рекомбінації з популяційного набору відбираються випадковим чином дві НС і відбувається обмін вагами між ними(рисунок 2.10):

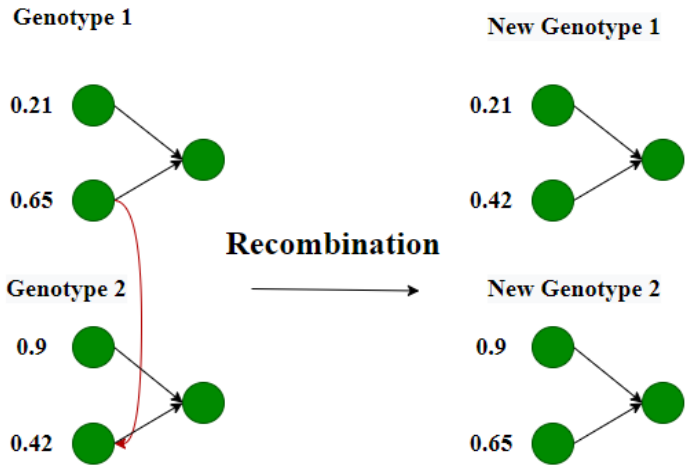

Рисунок 2.10 – Діаграма рекомбінації.

В результаті ми отримуємо дві нові особини, які поєднують в собі зв'язку двох предків.

3 ІНФОРМАЦІЙНЕ ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Вибір програмного середовища

Як середовище для розробки програми була обрана Unity, а редагування та компілювання коду - VisualStudio. Існує безліч середовищ розробки, але дана була обрана за численними рекомендаціями серед програмістів і з огляду на її зручного графічного інтерфейсу і засобів налагодження.

При розробці був використаний базовий функціонал Unity. Він дає можливість налаштування ігрового інтерфейсу та зручну інтеграцію у повноцінний програмний код. В Unity також є дуже зручна система префабів, яка спрощує роботу у редакторі. Настроєний об’єкт в будь-який час можна зберегти і використовувати його у подальшій роботі і в програмному коді і у редакторі. Усі ваші сутності розроблені як збірні, з компонентами, які з'єднуються в редакторі та зберігаються разом із збірною.

В Unity всі класи є нащадками Monobehaviour, він гарантує інтеграцію ігрової сцени з класом-спадкоємця, це досягається з допомогою реалізації подій та методів з класу Monobehaviour.

На рисунку 3.1 представлений зовнішній вигляд середовища розробки:

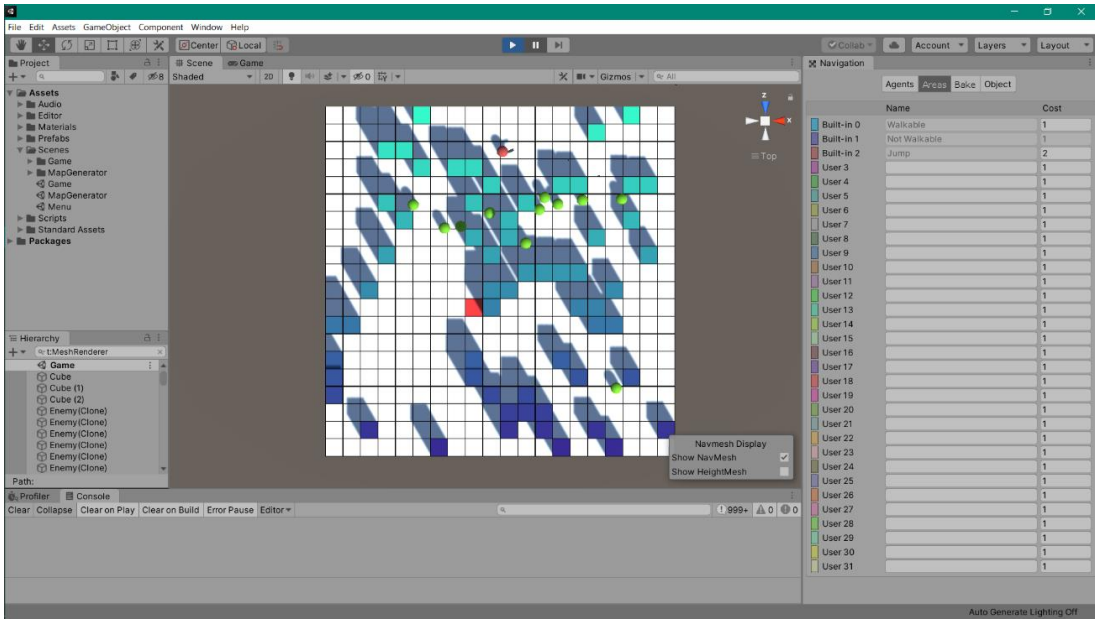

Рисунок 3.1 – Середовище розробки

На рисунку 3.1 представлений зовнішній вигляд середовища розробки:

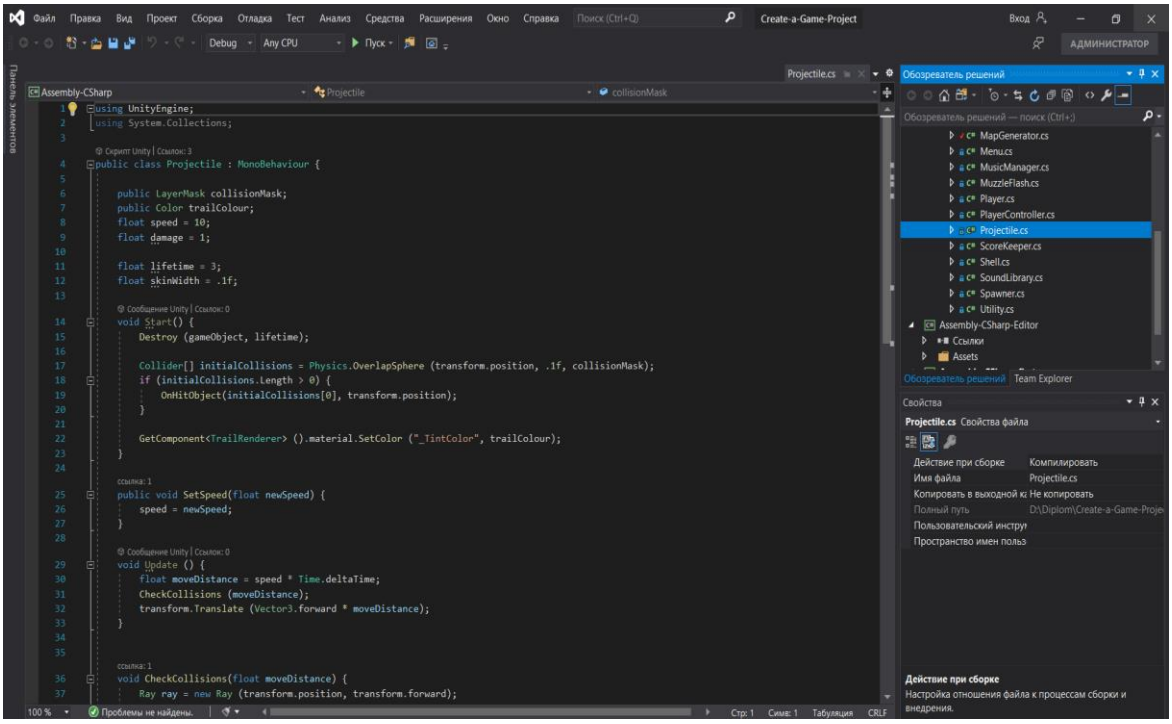

Рисунок 3.2 – Редактор додатку.

3.2 Короткий опис програмної реалізації

Діаграма класів є ключовим елементом в об'єктно-орієнтованому моделюванні. Діаграма класів проекту подана на рисунку 3.3.

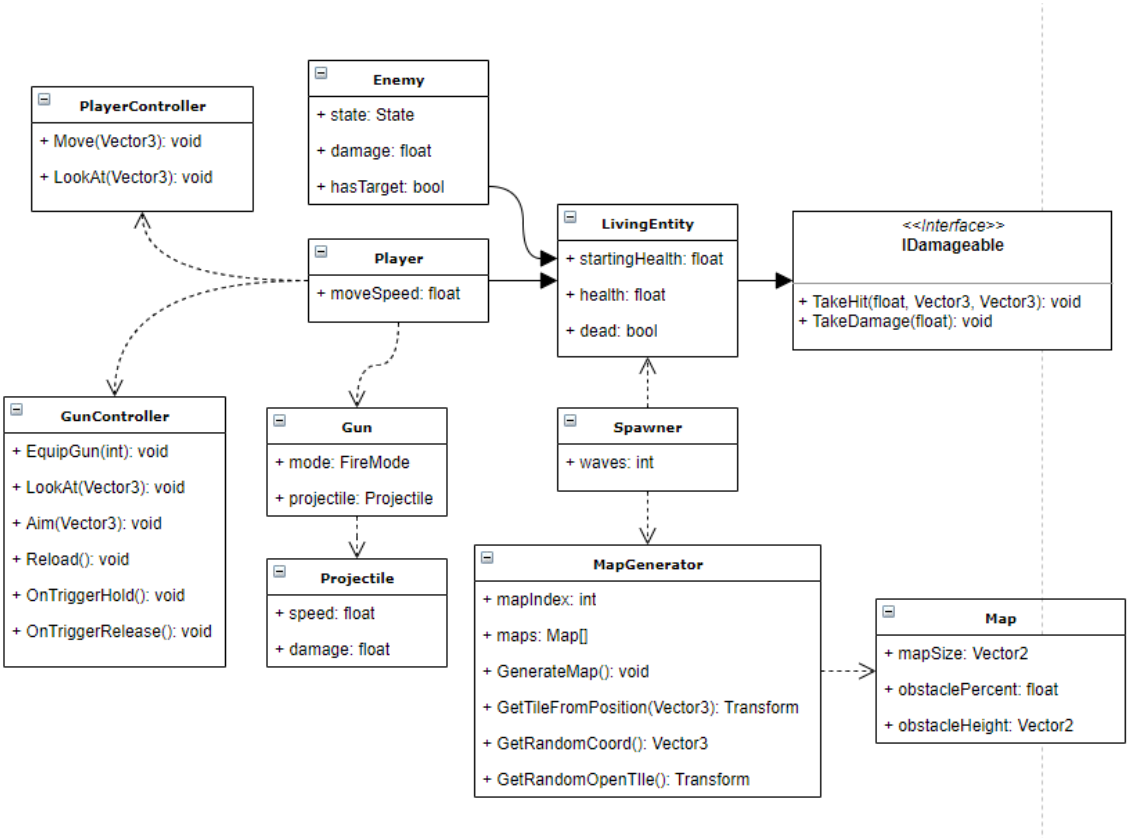

Рисунок 3.3 – Діаграма класів

В коді програми було реалізовано такі класи:

- Map: клас карти, він зберігає в собі її розмір, відсоток перешкод та їх висоту;
- MapGenerator: клас, що генерує карту, що містить розмір карти та методи її генерування;
- Projectile: класс, що відповідає за кулю, містить її швидкість та рівень нанесення шкоди;
- Gun: клас, що відповідає за зброю, він наслідується класом кулі;
- GunController: клас, що містить напрямок прицілу, та відповідає за методи пострілу;
- Player: клас, що містить швидкість гравця та наслідує клас LivingEntity;
- PlayerController: клас, що зберігає в собі вектор напрямку прицілу та вектор напрямку руху гравця;
- Enemy: клас моба, який атакує. Цей клас наслідує LivingEntity;
- LivingEntity: загальний клас для живих об’єктів таких, як гравець або ворог, він зберігає ;
- IDamageble: інтерфейс, що має функції прийому урону та його підрахунку;
- Spawner: клас, який відповідає за хвилі ворогів, він зберіє в собі кількість хвиль.

3.3 Алгоритми формування ігрового простору

При створенні гри було використано три алгоритми: алгоритм випадкових чисел, алгоритм генерування перешкод та алгоритм перевірка доступності середовища. Далі буде представлено декілька блок-схем для наглядного розуміння алгоритмів(рисунок 3.4).

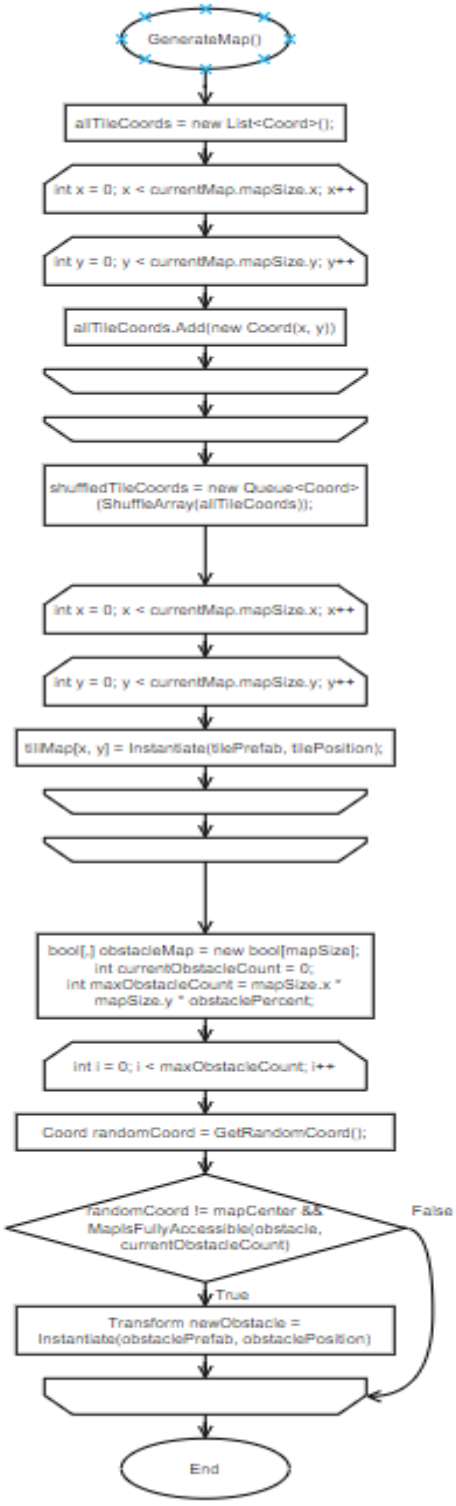

Рисунок 3.4 – Блок-схема функції GenerateMap.

Метод GenerateMap реалізує алгоритм алгоритм генерування перешкод.

Метод ShuffleArray реалізує алгоритм випадкових чисел його блок-схема представлена на рисунку 3.5.

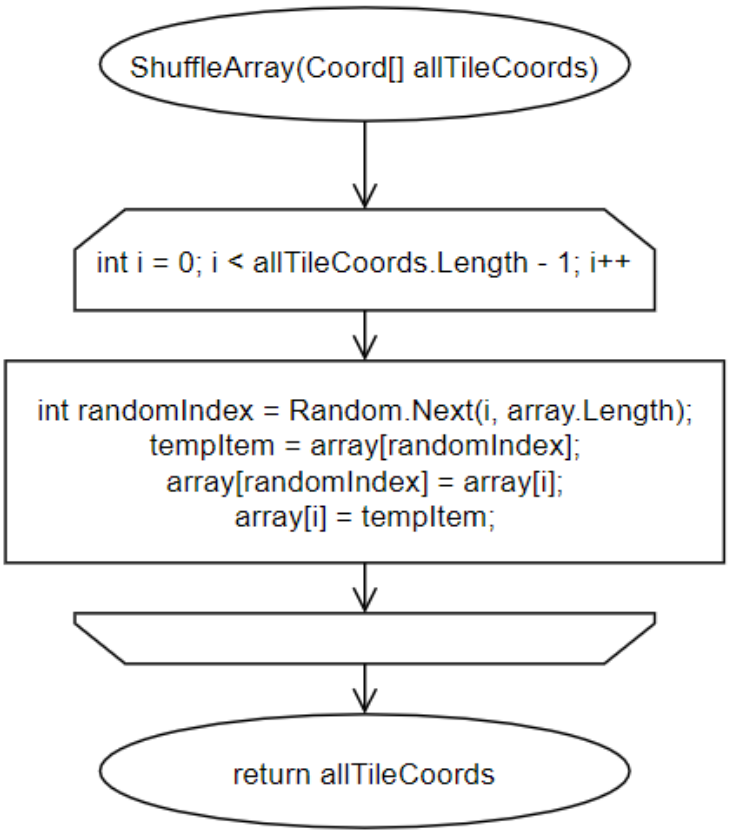

Рисунок 3.5 – Блок-схема функції ShuffleArray.

Метод MapIsFullyAccessible реалізує алгоритм перевірки замкнутого простору. Його реалізація представлена на рисунку 3.6.

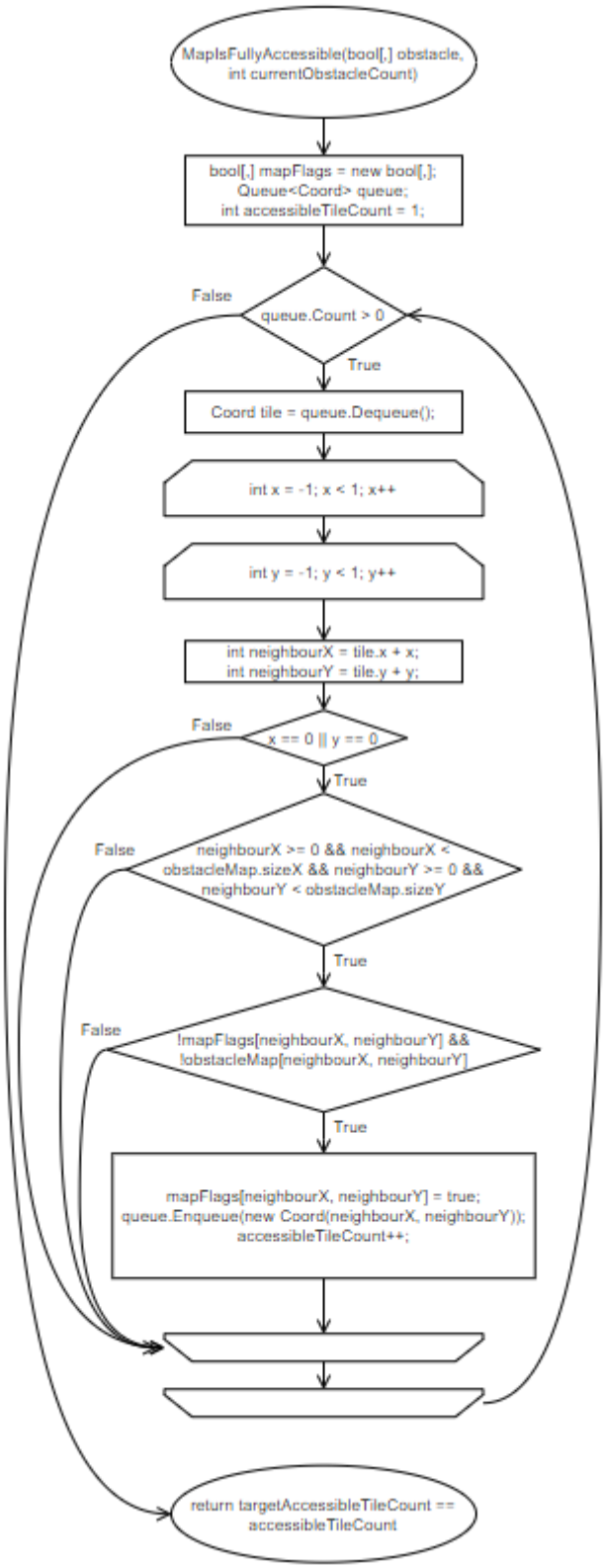

Рисунок 3.6 – Блок-схема функції MapIsFullyAccessible.

3.4 Тестування програмного забезпечення

Для перевірки працездатності ігрової системи була створена нова ігрова сесія. Була сгенерована сцена розміром 20 на 20. На рисунку 3.7 продемонстровано, як виглядає гра на початку.

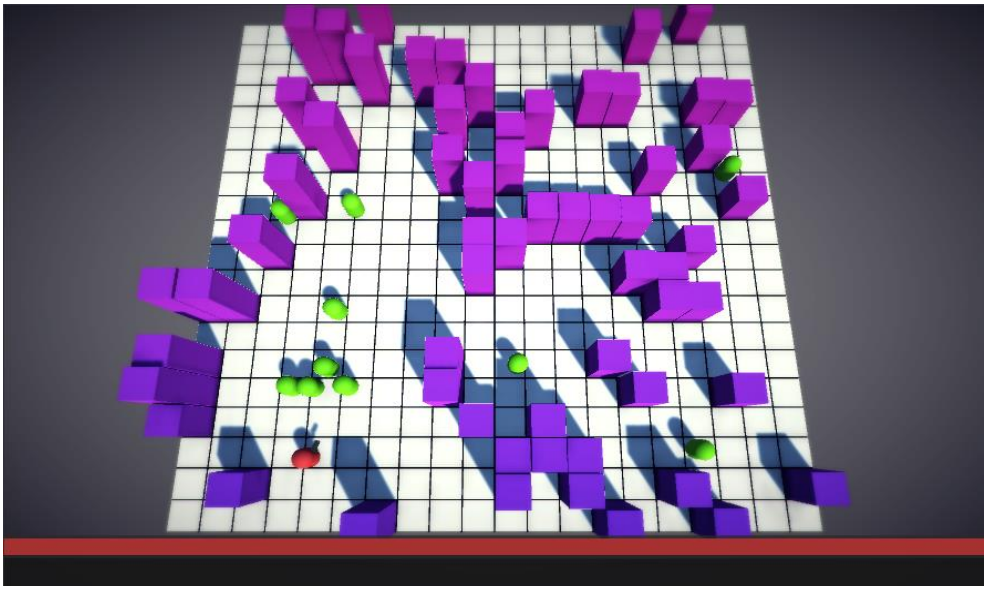

Рисунок 3.7 – Сцена гри на початку.

Спаун ворогів є випадковим, тож за 3 секунди виникає попередження про те, що на цьому місці з’явиться ворог. Це відбувається шляхом підсвічування червоним клітинки спауну. На рисунках 3.8, 3.9 продемонстровано як відбувається спаун.

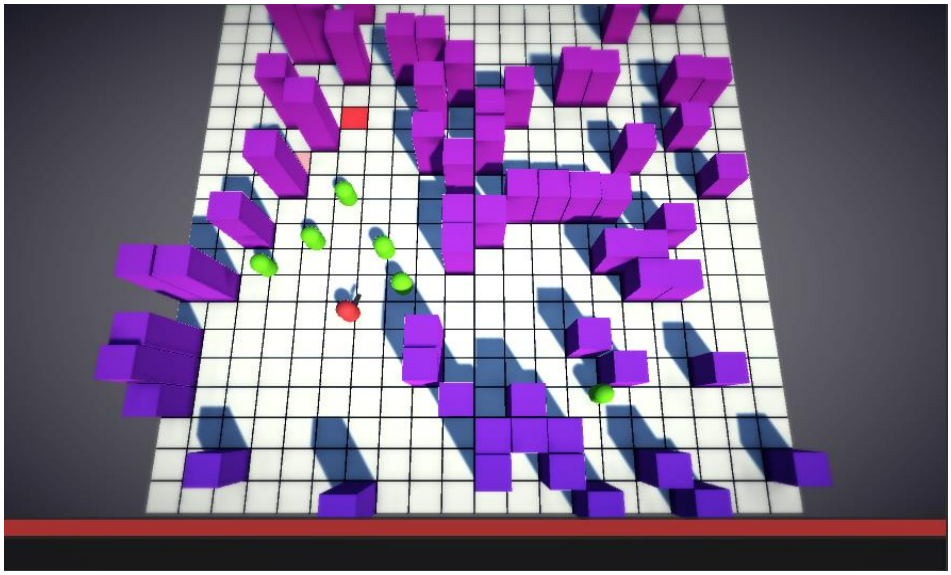

Рисунок 3.8 – Підсвічення червоним місця спауну.

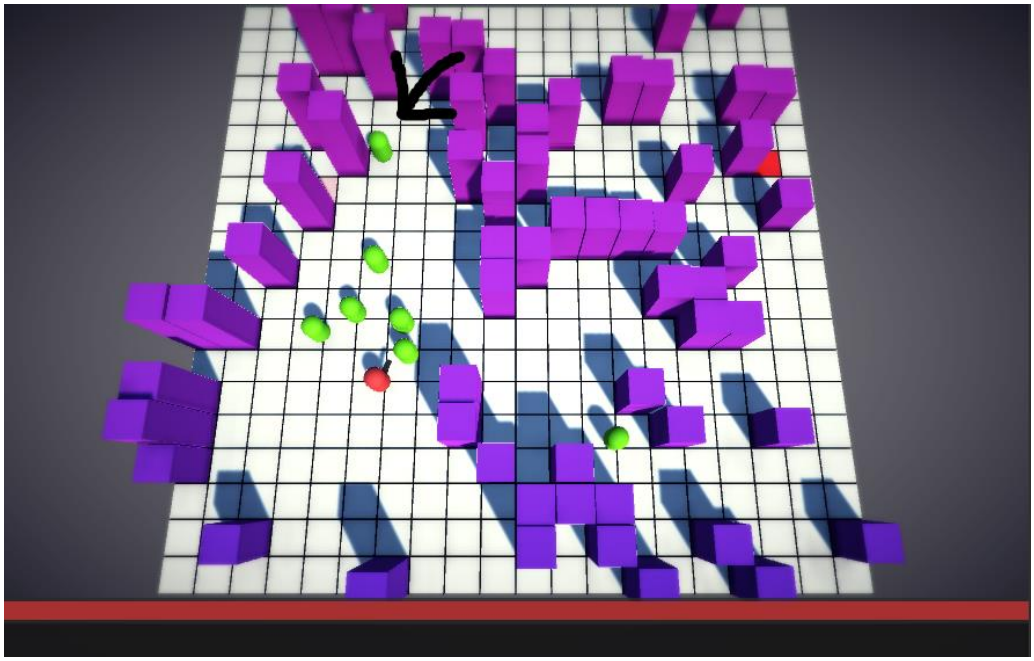

Рисунок 3.9 – Спаун ворога.

Коли ви вистрілюєте в ворога, він розлітається на полігони, це зображено на рисунку 3.10.

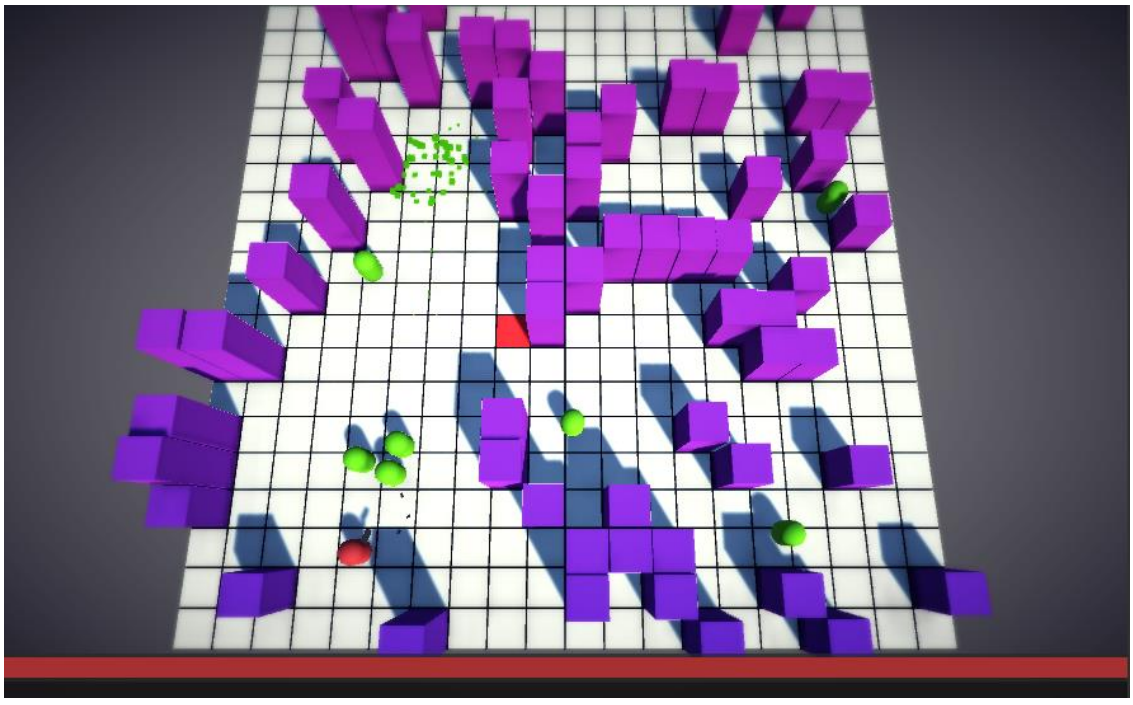

Рисунок 3.10 – Вбивство ворога.

Коли ворог досягає гравця, він наносить йому урон(рисунок 3.11). Шкалу здоров’я показано унизу сцени красною лінією.

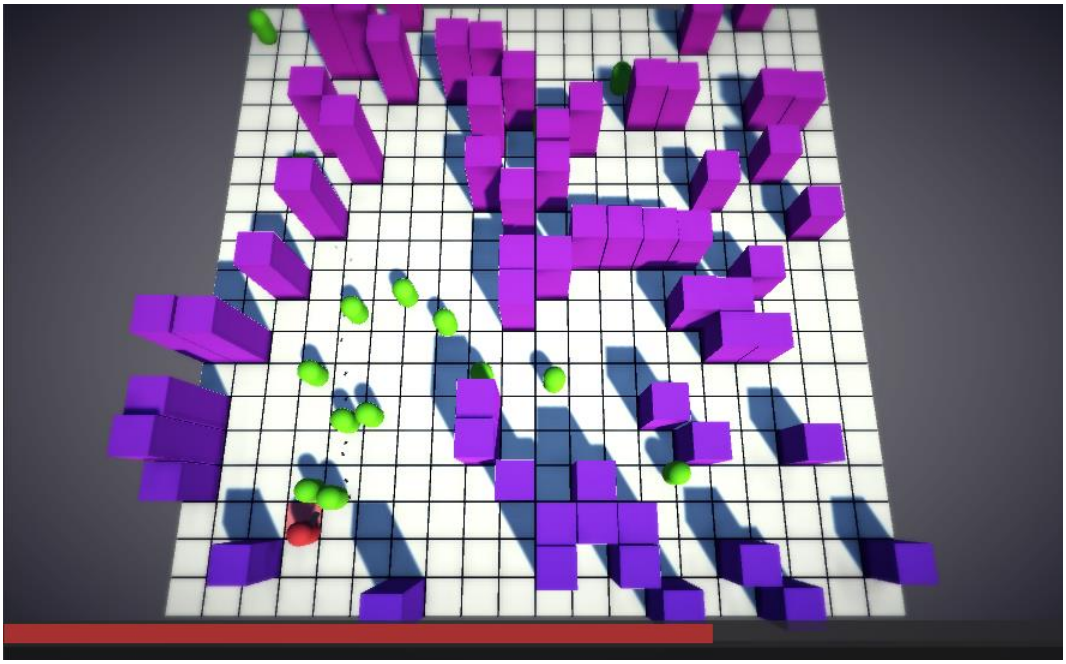

Рисунок 3.11 – Втрата частини здоров’я.

Коли кількість здоров’я досягне нуля гра закінчиться і буде показано підрахунок ваших балів(рисунок 3.12).

Рисунок 3.12 –Кінець гри.

ВИСНОВКИ

В випускній роботі була створена ігрова система. В роботі було проведено аналіз найліпших ігрових рушіїв, представлено плюси та мінуси їх використання. Також був проведений аналіз можливих алгоритмів генерування ігрового середовища. На основі цього аналізу було розроблено ігрову концепцію, створено ігрову логіку та правила, спроектовано ігрові механіки та прототип дизайну гри. Для генерування ігрового середовища був створений власний алгоритм на основі алгоритму випадкових чисел. Для реалізації проекту було використано середовище розробки Unity. Для перевірки валідності ігрової системи було проведено закрите Альфа та Бета - тестування.

СПИСОК ЛІТЕРАТУРИ

1. Bowen L.,Video game - Режим доступу https://www.apa.org/monitor/2016/02/video-game
2. Latushko V., What’s New in Unity3D 2020- Режим доступу: https://www.visartech.com/blog/unity3d-2020-what-is-new-and-how-business-benefits/
3. Unity3D или Unreal Engine 4 - Режим доступу: https://stfalcon.com/ru/blog/post/unity3d-vs-unreal-engine-4
4. Buckley D., Unity vs. Unreal – Choosing a Game Engine - Режим доступу: https://gamedevacademy.org/unity-vs-unreal/
5. UNITY PRO - быстрый старт - Режим доступу: http://www.fb.asu.in.ua/realizacia-platformy/modicon/unity-bystryj-start
6. Dealessandri M., What is the best game engine: is Unreal Engine right for you? - Режим доступу: https://www.gamesindustry.biz/articles/2020-01-16-what-is-the-best-game-engine-is-unreal-engine-4-the-right-game-engine-for-you
7. Understanding Perlin Noise - Режим доступу: https://adrianb.io/2014/08/09/perlinnoise.html
8. Nystrom R., Game Programming Patterns – Kindle Edition, 2014р. –295ст.
9. Schell J.,The Art of Game Design: A Book of Lenses - Kindle Edition, 2018 р. – 213ст.
10. Procedural Terrain Generation: Diamond-Square - Режим доступу: http://jmecom.github.io/blog/2015/diamond-square/
11. Perry D., David Perry on Game Design: A Brainstorming Toolbox Rusel – DeMaria , 2008 р. – 304 ст.
12. The Wavefunction Collapse Algorithm explained very clearly - Режим доступу: https://robertheaton.com/2018/12/17/wavefunction-collapse-algorithm/

13. Powell V., Markov Chains - Режим доступу: https://setosa.io/ev/markov-chains/
14. Wong J., Metaballs and Marching Squares - Режим доступу: http://jamie-wong.com/2014/08/19/metaballs-and-marching-squares/
15. Huttar L., Learning how Perlin noise works- Режим доступу: http://www.huttar.net/lars-kathy/graphics/perlin-noise/perlin-noise.html

ДОДАТОК

Клас MapGenerator реалізує логіку генерації всього сцени у вигляді двовимірного масиву об'єктів елементів Tile

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

public class MapGenerator : MonoBehaviour {

     public Map[] mapis;
     public int mapIndex;

     public Transform tile_prefab;
     public Transform obstacle_prefab;
     public Transform map_floor;
     public Transform navmesh_floor;
     public Transform navmesh_mask_prefab;
     public Vector2 max_map_size;

     [Range(0,1)]
     public float outline_percent;

     public float tile_size;
     List<Coord> all_tile_coords;
     Queue<Coord> shuffled_tile_coords;
     Queue<Coord> shuffled_open_tile_coords;
     Transform[,] tileMap;

     Map current_map;

     void Awake() {
          Find_object_of_type<Spawner> ().On_new_wave +=
On_new_wave;
     }

     void On_new_wave(int wave_number) {
          map_index = wave_number - 1;
          Generate_map ();
     }

     public void Generate_map() {
          current_map = maps[map_index];
          tile_map = new
Transform[current_map.map_size.x,current_map.map_size.y];
          System.Random prng = new System.Random
(current_map.seed);

          // Generating coords
          all_tile_coords = new List<Coord> ();
```

```
          for (int x = 0; x < current_map.map_size.x; x ++) {
               for (int y = 0; y < current_map.map_size.y; y ++)
{
                    all_tile_coords.Add(new Coord(x,y));
               }
          }
          shuffled_tile_coords = new Queue<Coord>
(Utility.Shuffle_array (all_tile_coords.ToArray (),
current_map.seed));

          string holderName = "Generated Map";
          if (transform.Find (holderName)) {
               DestroyImmediate (transform.Find
(holder_name).game_object);
          }

          Transform map_holder = new Game_object
(holder_name).transform;
          Map_holder.parent = transform;
          for (int x = 0; x < current_map.map_size.x; x ++) {
               for (int y = 0; y < current_map.map_size.y; y ++)
{
                    Vector3 tilePosition = CoordToPosition(x,y);
                    Transform newTile = Instantiate
(tile_prefab, tile_position, Quaternion.Euler (Vector3.right *
90)) as Transform;
                    newTile.local_scale = Vector3.one * (1 -
outline_percent) * tile_size;
                    new_tile.parent = map_holder;
                    tile_map[x,y] = new_tile;
               }
          }
          bool[,] obstacle_map = new
bool[(int)current_map.map_size.x,(int)current_map.map_size.y];

          int obstacleCount = (int)(current_map.map_size.x *
current_map.map_size.y * current_map.obstaclePercent);
          int current_obstacle_count = 0;
          List<Coord> all_open_coords = new List<Coord>
(all_tile_coords);

          for (int i =0; i < obstacleCount; i ++) {
               Coord random_coord = GetRandom_coord();
               obstacle_map[random_coord.x,random_coord.y] =
true;
               current_obstacle_count ++;

               if (random_coord != current_map.map_centre &&
MapIsFullyAccessible(obstacle_map, current_obstacle_count)) {
                    float obstacle_height =
Mathf.Lerp(current_map.minObstacle_height,current_map.maxObstacl
e_height,(float)prng.NextDouble());
```

```
                    Vector3 obstaclePosition =
CoordToPosition(random_coord.x,random_coord.y);

                    Transform newObstacle =
Instantiate(obstaclePrefab, obstaclePosition + Vector3.up *
obstacle_height/2, Quaternion.identity) as Transform;
                    newObstacle.parent = map_holder;
                    newObstacle.local_scale = new Vector3((1 -
outlinePercent) * tile_size, obstacle_height, (1 -
outlinePercent) * tile_size);

                    Renderer obstacleRenderer =
newObstacle.GetComponent<Renderer>();
                    Material obstacleMaterial = new
Material(obstacleRenderer.sharedMaterial);
                    float colourPercent = random_coord.y /
(float)current_map.map_size.y;
                    obstacleMaterial.color =
Color.Lerp(current_map.foregroundColour,current_map.backgroundCo
lour,colourPercent);
                    obstacleRenderer.sharedMaterial =
obstacleMaterial;

                    all_open_coords.Remove(random_coord);
               }
               else {
                    obstacle_map[random_coord.x,random_coord.y]
= false;
                    current_obstacle_count --;
               }
          }

          shuffled_open_tile_coords = new Queue<Coord>
(Utility.Shuffle_array (all_open_coords.ToArray (),
current_map.seed));

          // Creating navmesh mask
          Transform mask_left = Instantiate
(navmesh_mask_prefab, Vector3.left * (current_map.map_size.x +
max_map_size.x) / 4f * tile_size, Quaternion.identity) as
Transform;
          mask_left.parent = map_holder;
          mask_left.local_scale = new Vector3 ((max_map_size.x -
current_map.map_size.x) / 2f, 1, current_map.map_size.y) *
tile_size;

          Transform mask_right = Instantiate
(navmesh_mask_prefab, Vector3.right * (current_map.map_size.x +
max_map_size.x) / 4f * tile_size, Quaternion.identity) as
Transform;
          mask_right.parent = map_holder;
```

```
          mask_right.local_scale = new Vector3 ((max_map_size.x
- current_map.map_size.x) / 2f, 1, current_map.map_size.y) *
tile_size;

          Transform mask_top = Instantiate (navmesh_mask_prefab,
Vector3.forward * (current_map.map_size.y + max_map_size.y) / 4f
* tile_size, Quaternion.identity) as Transform;
          mask_top.parent = map_holder;
          mask_top.local_scale = new Vector3 (max_map_size.x, 1,
(max_map_size.y-current_map.map_size.y)/2f) * tile_size;

          Transform maskBottom = Instantiate
(navmesh_mask_prefab, Vector3.back * (current_map.map_size.y +
max_map_size.y) / 4f * tile_size, Quaternion.identity) as
Transform;
          maskBottom.parent = map_holder;
          maskBottom.local_scale = new Vector3 (max_map_size.x,
1, (max_map_size.y-current_map.map_size.y)/2f) * tile_size;

          navmeshFloor.local_scale = new Vector3
(max_map_size.x, max_map_size.y) * tile_size;
          mapFloor.local_scale =  new Vector3
(current_map.map_size.x * tile_size, current_map.map_size.y *
tile_size);
     }

     bool MapIsFullyAccessible(bool[,] obstacle_map, int
current_obstacle_count) {
          bool[,] mapFlags = new
bool[obstacle_map.GetLength(0),obstacle_map.GetLength(1)];
          Queue<Coord> queue = new Queue<Coord> ();
          queue.Enqueue (current_map.map_centre);
          mapFlags [current_map.map_centre.x,
current_map.map_centre.y] = true;

          int accessible_tile_count = 1;

          while (queue.Count > 0) {
               Coord tile = queue.Dequeue();

               for (int x = -1; x <= 1; x ++) {
                    for (int y = -1; y <= 1; y ++) {
                         int neighbourX = tile.x + x;
                         int neighbourY = tile.y + y;
                         if (x == 0 || y == 0) {
                              if (neighbourX >= 0 && neighbourX
< obstacle_map.GetLength(0) && neighbourY >= 0 && neighbourY <
obstacle_map.GetLength(1)) {
                                   if
(!mapFlags[neighbourX,neighbourY] &&
!obstacle_map[neighbourX,neighbourY]) {

     mapFlags[neighbourX,neighbourY] = true;
```

```
                                        queue.Enqueue(new
Coord(neighbourX,neighbourY));
                                        accessible_tile_count
++;
                                    }
                                }
                            }
                        }
                    }
                }

            int targetAccessible_tile_count =
(int)(current_map.map_size.x * current_map.map_size.y -
current_obstacle_count);
            return targetAccessible_tile_count ==
accessible_tile_count;
        }

        Vector3 CoordToPosition(int x, int y) {
            return new Vector3 (-current_map.map_size.x / 2f +
0.5f + x, 0, -current_map.map_size.y / 2f + 0.5f + y) *
tile_size;
        }

        public Transform GetTileFromPosition(Vector3 position) {
            int x = Mathf.RoundToInt(position.x / tile_size +
(current_map.map_size.x - 1) / 2f);
            int y = Mathf.RoundToInt(position.z / tile_size +
(current_map.map_size.y - 1) / 2f);
            x = Mathf.Clamp (x, 0, tileMap.GetLength (0) -1);
            y = Mathf.Clamp (y, 0, tileMap.GetLength (1) -1);
            return tileMap [x, y];
        }
}
```