МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

ВИПУСКНА РОБОТА

на тему:

«Комп'ютерна 3D -гра з використанням Unity Engine и C#»

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

Керівник роботи **Шутилєва О. В.**

Студента групи ІН – 71 **Трусова Б. О.**

СУМИ 2021

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Кафедра комп’ютерних наук

Затверджую _____

Зав. кафедрою Довбиш А.С.

“_____” _____2021 р.

ЗАВДАННЯ
до випускної роботи

Студента четвертого курсу, групи ІН-71 спеціальності “122 - комп’ютерні науки” денної форми навчання Трусова Богдана Олексійовича.

Тема: «Комп’ютерна 3D -гра з використанням Unity Engine и C#»

Затверджена наказом по СумДУ

№_____ от _____2021 р.

Зміст пояснювальної записки: 1) загальний огляд методів та підходів для створення комп'ютерних ігор; 2) постановка завдання й формулювання завдань роботи; 3) опис основних положень, підходів та методів, що використовуються у створенні комп’ютерних 3D ігор; 4) розробка прототипу комп’ютерної гри; 5) аналіз та тестування прототипу.

Дата видачі завдання “_____”_____2021 р.

Керівник випускної роботи _____ Шутилєва О. В.

Завдання прийняв до виконання _____ Трусов Б. О.

РЕФЕРАТ

Записка: 47 стор., 43 рис., 1 додаток, 5 джерел.

Об'єкт дослідження — процес створення 3D ігор.

Мета роботи — розробка прототипу комп'ютерної гри з використанням сучасних пайплайнів, для створення 3D графіки, а також з використанням Unity3d engine та C#.

Методи дослідження — використання сучасних пайплайнів та інструментарію рушія Unity3d.

Результати — розроблено прототип 3D гри. Протестовані та проаналізовані основні аспекти гри такі як: дизайн рівнів, геймплей, скрипти. Розроблений прототип реалізовано у формі програмного забезпечення, створеного за допомогою інструментарію Unity3d engine, використанням бібліотек C#, а також програмного забезпечення згідно з обраним пайплайном для створення 3D гарфіки : 3ds max, Substance painter, Photoshop.

КОМП'ЮТЕРНІ ІГРИ,3D МОДЕЛЬ,
ДИЗАЙН РІВНІВ, АНІМАЦІЯ,
ЛОГІКА СЦЕНАРІЇВ UNITY3D,
ІГРОВИЙ ДИЗАЙН ДОКУМЕНТ.

ЗМІСТ

ВСТУП

В наш час ігрова індустрія є невід'ємною частиною життя якщо не кожної, то багатьох людей. Це зумовлено швидким розвитком останньої: лише приблизно 20 років тому не було відкритого інструментарію, обчислювальна потужність була мізерною і максимум тих часів було відображення інформації на моніторі за допомогою консольних символів.

Проте, з плином часу технології почали прогресувати. Спочатку різні студії почали створювати власний інструментарій(рушій) для подальшого створення свого ігрового продукту. Траплялися досить вдалі спроби(Unreal engine) так і взагалі жахливі(Genom engine). Все залежало від підходу до створення, використаних технологій а також видавництва, з яким працювали розробники.

В сучасних же реаліях трапились суттєві зміни: технології досить швидко і сильно ступили вперед, що не кожна команда розробників може дозволити собі розробляти власний конкурентоспроможний рушій. На зустріч таким розробникам пішли компанії-гіганти і як результат більшість популярних рушіїв стали умовно безкоштовними, а також розробникам надали весь можливий інструментарій для створення власного ігрового продукту. Це дало можливість розробникам не відволікатись на низькорівневі процеси(відросивування трикутників 3д моделі), а сконцентруватись на логічному та естетичному наповненню ігрового процесу. Як наслідок все більше і більше виділяється і часу і фінансів на створення приємної графіки, чудового музикального супроводу, а що саме головне скриптовій логіці гри.

Саме на цій підставі ігри неважливо де: смартфон, приставка чи ПК стали пов’язувати в собі різноманітні аспекти мистецтва: образотворче, музикальне, словесне. Проте без сильної технічної частини неможливо гармонійно поєднати дані розділи.

Основою кожної гри як 2d так і 3d є рушій. Кожен рушій базується на бібліотеці певної мови програмування та використання SDK, бібліотеки яких

будуть використовуватись самим ігровим продуктом. Проте і SDK не є проблемою, адже більшість відкритих ігрових рушіїв кросплатформенні, що дає можливість створювати ігровий продукт на різні платформи за мінімум використаного часу.

Наступний аспект на який слід звернути увагу це графічна частина гри. Вона складається із трьох основних частин:

1. дизайн ігрових рівнів - це проектування та створення ігрових локацій, згідно з концептом та потребами розробників
2. дизайн ігрових персонажів - це аналіз стилістики, опрацювання та створення усіх анімованих ігрових персонажів гри
3. UI інтерфейс - це частина візуального аспекту гри, яка дає можливість гравцю напряму взаємодіяти з грою.

За допомогою цих двох потужних інструментів як графічна частина і технічна(логіка та сценарії) і буде створюватись прототип комп'ютерної гри.

АНАЛІЗ ІГРОВОЇ ІНДУСТРІЇ. ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз сучасних ігор та виділення трендових переваг та механік

Перед створенням власного продукту слід ретельно проаналізувати теперішній ринок відеоігор: які ігрові механіки або їх сукупність є самими розповсюдженими і бажаними. Потім слід розуміти, що необхідно змінити, або чим їх доповнити щоб створюваний продукт був унікальним та цікавим водночас.

Напевне найгучнішим ігровим трендом є хмарний геймінг. Для прикладу Google Stadia або ж Playstation Now можуть запускати будь-які по складності проекти, використовуючи свої гігантські серверні ресурси. Користувачі зможуть підключатись до ігрової хмари з будь-якого навіть дуже обмеженого у ресурсах пристрою и грати в складні, з технічної точки зору, продукти, не відчуваючи дискомфорту. З матеріальної точки зору такий підхід набирає обертів тому що юзери отримають продукт вищої якості за менші витрати. Яку користь можна винести з даної технології? Розроблюваний продукт повинен бути якомога доступнішим, не псуючи при цьому ігровий досвід юзера.

Іншим же розділом ігрової індустрії є онлайн ігри. Розповісти про всі ігри даного сегменту напевне неможливо, проте слід відмітити найпопулярніші із них. Проаналізуємо гру Fortnite. Перш за все гра цікава тим, що дає можливість змагатися з іншими реальними гравцями. При цьому гра надає безліч можливостей для змагання: різноманітні персонажі з уникальними здатностями, досить масштабна карта та велика кількість зброї. Подібні плюси має гра Dota2 та напевне будь яка інша онлайн гра, за основу якої взятий режим PvP(player versus player). Проте у цього режиму є один суттєвий мінус: для нього необхідний великий і головне стабільний онлайн

проекту, що досить складно отримати на невеликій інді грі. Щоб вирішити, нехай і тимчасово дану проблему слід звернути увагу на серію ігор Dark Souls. Основною масою суперників є звичайні рядові монстри, проте гра дає можливість реальним людям вторгатись в ігрові світи інших гравців. При цьому, якщо онлайн малий або ж його зовсім немає, то можна піти на хитрість: видавати звичайного суперника за реального, проте ним буде керувати штучний інтелект. Важливо: щоб ефект схожості був максимальним необхідно щоб рухи, швидкість, анімації та інші візуальні моменти були максимально схожими на гравця і не повторювались більше ні в яких інших суперників.

Наступна ігрова ніша, яку неможливо ігнорувати це гіпер казуальні ігри. Чому вони настільки популярні? Кожна гіпер казуальна гра приносить максимальну кількість задоволення при мінімально витрачених силах. Завжди приємно коли збирається велика кількість кристалів, лопається багато кульков одночас, швидко збирається пазл і тому подібні прості механіки. Як можна використати подібний досвід, створюючи власний прототип гри в іншому жанрі? Використати схожу механіку, оперуючи доступними об'єктами: самими слабкими суперниками, або ж предметами. Для прикладу: дати гравцю можливість перемагати велику зграю слабких створінь однією кнопкою.

Як протиставлення попередній механіці, необхідно вносити досить сильних створінь, для отримання перемоги над якими юзеру необхідно буде сконцентруватись і докласти зусиль. Завдяки подібному контрасту гравецю не буде нудно в створюваній грі. Проте не слід піднімати планку складності досить високо, адже це може повністю відбити інтерес до продукту. Подібним протиставленням складно-легко відомі ігри жанру hack&slash. Наприклад серія ігор Diablo, Grim Dawn, Path of Exile та багато інших.

Чим можна зацікавити гравця прогресувати далі і далі? Наприклад можна використати механіки role play games, а саме покращення здібностей свого героя. Нікому не буде цікаво використовувати одну і ту саму можливість, зброю чи прийом протягом тривалого часу. Саме для цього

розробники створюють прогресію покращення здібностей головного героя, вводять різноманітну зброю, обладунки і тому подібні речі, які будуть стимулювати гравця покращувати свого героя. Щось конкретне виділити досить складно, усе залежить від фантазії та бажань розробників. Для прикладу у прототипі гри можна ввести декілька параметрів головного героя: рівень, кількість очок здоров'я, кількість очок енергії та швидкість. З кожним рівнем кількість даних параметрів буде збільшуватись. Також гравцю будуть надаватись очки для покращення здібностей персонажу.

Описаних механік повністю достатньо для проектування та подальшого створення прототипу гри, тому можна перейти до створення game design document(GDD).

1.2 Ігровий дизайн документ

Найкращим способом постановки задачі для створення ігор є написання ігрового дизайн документу(gdd). Буде створюватись десятисторінник, про який описано в книзі Скота Роджерса "Level up. The guid to great video game design".

Десятисторінник - це документ який обмежений десятьма сторінками, та може бути написаний для різної цільової аудиторії. Якщо документ створюється для команди розробників, він повинен містити точний опис локацій, механік, персонажів, ігор якими надихався геймдизайнер при написанні даного документу, навіть якщо вони були невдалими. Сам Скот Роджерс писав, що найчастіше даний десятисторінник є основним набором правил для команди розробників, якими вони керуються, тому до його розроблення слід підійти максимально відповідально. Також в книзі була описана структура даного документу:

- перша сторінка: містить назву, платформи, вік цільової аудиторії, дата видання продукту, конкуренти та ігри, які надихали розробників, концепт арт, який передає дух гри та контактні дані
- друга сторінка: містить короткий опис сюжету від початку до його завершення та біглий опис як ігрових рівнів так і дій в ньому; ключове завдання цієї сторінки - донести цільовій аудиторії ігровий досвід та провести через нього від його початку до завершення
- третя сторінка: опис персонажу, його ім’я, раса, історія, цілі, характер, можливості; вагомим плюсом є наявний концепт арт персонажа
- четверта сторінка: детальний опис ігрового процесу враховуючи жанр розроблюваної гри, які будуть рівні, опис найскладніших чи цікавіших, різноманітні технічні обмеження, системні, на чому запускається та чим контролюється розроблюваний продукт
- п’ята сторінка: опис ігрового світу, кількість локацій, яким чином вони пов’язані між собою та завдяки чому гравець буде орієнтуватись в ньому, за необхідності концепт арт локацій, або загальної карти
- шоста сторінка: ігровий досвід. А саме як довести гравця від початку гри до її завершення, зберігаючи при цьому цілісність та завершеність. Що мотивує гравця рухатись від початку до кінця? За наявності аудіосупроводу опис звуків локацій
- сьома сторінка: описання ключових ігрових механік продукту: убивання, покращення, колекціонування, секрети, пастки і всі інші механіки якими оперує юзер в момент гри
- восьма сторінка: вороги, їх опис, різноманітність, кількість, ким або чим керуються та як перемагаються
- дев’ята сторінка: описання сюжетних роликів чи сцен якщо такі наявні, або ж якісь сценки подібні останнім
- десята сторінка: бонусний контент, яка винагорода буде чекати гравця після перемоги

Створений ігровий дизайн документ у вигляді десятисторінника(Додаток А) на основі якого і буде написана постановка задачі.

1.3 Постановка задачі

Необхідно створити прототип комп'ютерної гри відповідно до ігрового дизайн документу, виконавши такі етапи:

1. Проектування ігрового світу
2. Створення та анімація головного героя та його логіка(бій, та взаємодія з ворогами)
3. Створення та анімація ворогів і винагорода за перемогу над ними.
4. Система зберігання предметів.

МЕТОДИ ТА РІШЕННЯ ПРИ СТВОРЕННІ ІГОР

2.1 Вибір ігрового рушію

Основою будь-якої іграшки на всіх сучасних системах є ігровий рушій. Це набір інструментів, за допомогою яких розробники можуть поєднувати в одному продукті такі аспекти як відображення графіки синхронізоване зі звуковим супроводом та логічним змістом.

Проте не всі ігрові рушії однакові за своїм наповненням, інструментарієм, технічними потребами, а також результатом роботи який можна отримати за його використання. Розглянемо самі відомі рушії.

Unreal engine - кроссплатформенный рушій, створений американською компанією Epic Games. Спочатку даний рушій був закритим, і щоб розробники могли навчитись з ним працювати потрібно було придбати повну версію інструментарію. Проте з виходом 4 версії політика компанії змінилась: рушій може використовувати будь-хто і в будь яких цілях, якщо фінансові обороти розробників не перевищують певну суму. Завдяки цьому рішенню велика кількість людей може ознайомитись з інструментами та документацією, побачити як створюються ігри з середини і можлива в майбутньому стати частиною ігрової індустрії.

Стосовно конкурентів: чим саме цей рушій виділяється на відміну від Unity3d або ж CryEngine. А відмінності досить суттєві. Перш за все у кожного рушія свій список платформ, під які розробники можуть створювати свій продукт. Unreal може похизуватися такими: PC, PS4, Xbox, Android, SteamOS, iOS та Linux. Наступною перевагою даного рушію є використання технології BluePrint - візуальне програмування. У розробників є можливість створити особисту гру не написавши ні однієї строки коду. Приклад візуального програмування ви можете спостерігати нижче:

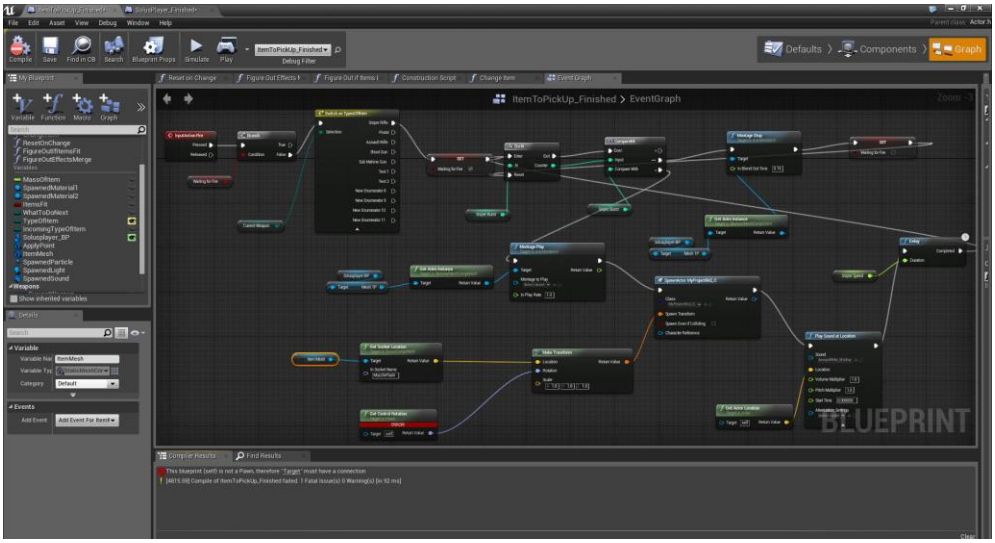


Рисунок 2.1 - приклад використання технології BluePrint

Проте якщо по тій чи іншій причині використовувати цю технологію не можна, на зустріч виходить класичний програмний код на мові C++. Більш того, готові blueprints можна конвертувати в C++ код без особливих зусиль.

Наступною перевагою даного рушія на відміну від конкурентів є досить гнучкий та потужний редактор матеріалів. Взагалі при створенні комп'ютерної графіки художники використовують різне програмне забезпечення для створення потрібної картинки: можна створити hand-paint текстури, можна скористатись програмою Substance Painter з великою бібліотекою уже готових та гнучких матеріалів, або ж створити власний матеріал зі своїми особливими фізичними якостями. Використати для цього можна як і програму Substance Design так і сам рушій. Налаштований матеріал зберігається в картах, які шляхом накладання одна на одну при взаємодії з ігровим освітленням створюють максимально реалістичну картинку.

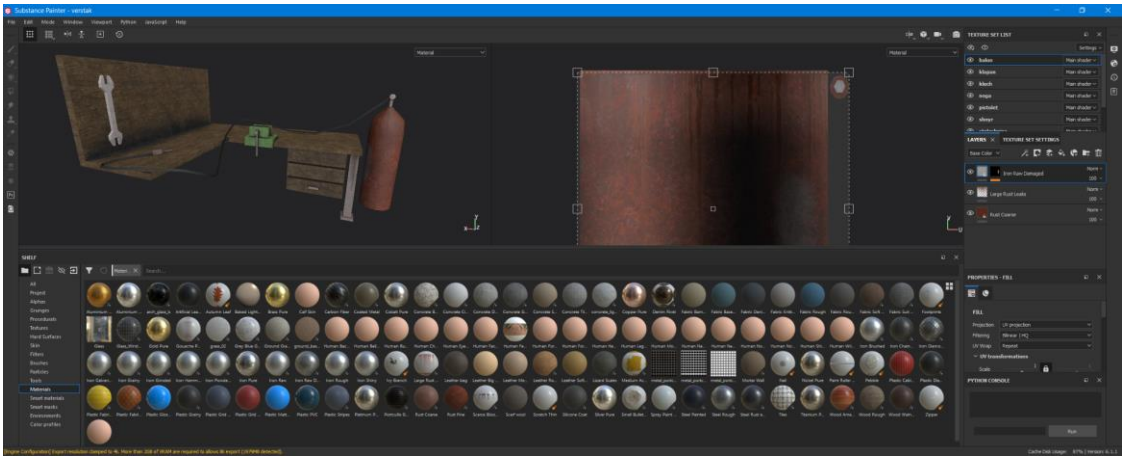


Рисунок 2.2 - приклад текстурування у програмі Substance Painter

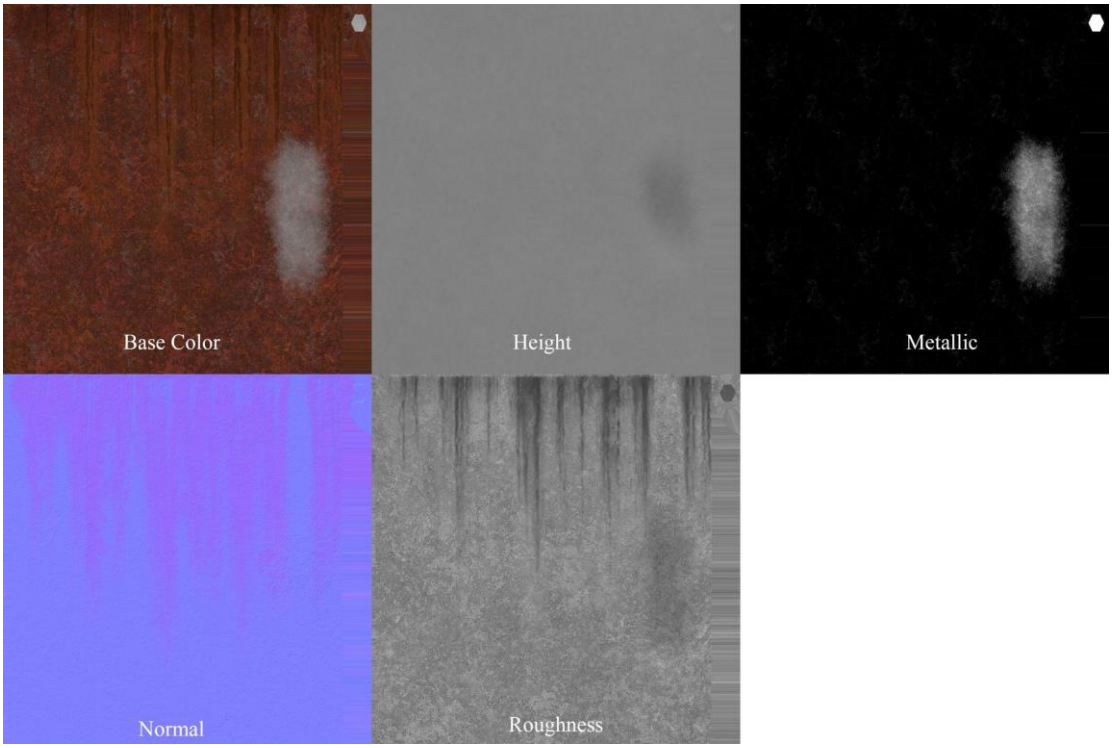


Рисунок 2.3 - pbr текстури

Вище описані інструменти є у деяких пайплайнах створення трьохмірної графіки, проте Unreal надає розробникам можливість користуватись внутрішнім редактором матеріалів, скорочуючи час на виробництво.

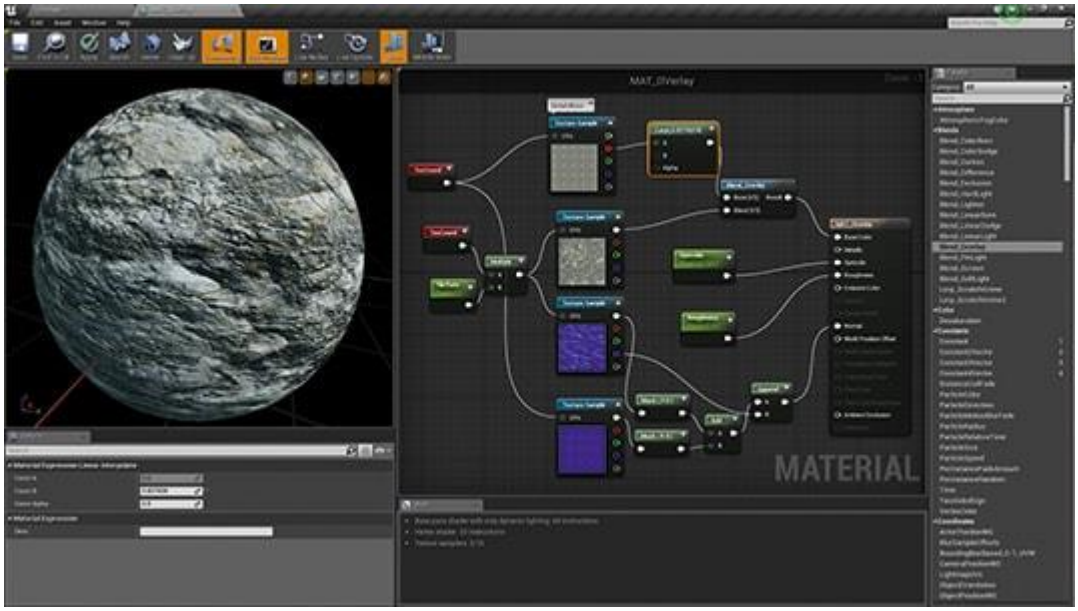


Рисунок 2.4 - редактор матеріалів Unreal Engine 4

Також слід згадати те, що вихідний код рушію відкритий, і к розробників є можливість змінювати уже готові частини рушію, а також з легкістю додавати свої. Для прикладу, було створено blueprint, що дозволяє використовувати C# замість C++. Із мінусів даного рушію є те що він потребує 4.2gb оперативної пам’яті при розрахунках освітлення та взагалі не підтримує 32-х розрядні операційні системи. Це вносить технічні обмеження як для розробників(необхідне сучасне обладнання для розробки) також для юзерів і із за цього обмеження кількість платформ для розробки різко зменшується.

Наступним рушієм-конкурентом є CryEngine. Його можно отримати безкоштовно для некомерційної діяльності, а також у цілях навчання. Проте для розробників команія пропонує досить агресивну політику цін, яку не кожен може собі дозволити. Головною перевагою даного рушію, окрім використання бібліотек DirectX 12 є реалістичний фізичний рушій, що дозволяє створювати реалістичні ефекти руйнування, вибухів та погоди. Даний рушій немає внутрішнього середовища візуального програмування та редактора матеріалів, тому потрібно використовувати сторонні матеріали при розробці візуальної частини. Також даний рушій потребує досить суттєвих

ресурсів системи(навіть більше ніж попередній), тому слід ретельно продумати вибір даного рушію.

Наступний і останній на розгляді рушій Unity3d Engine. Основною перевагою даного рушію є кросплатформеність та низькі потреби в ресурсах системи. Даний інструмент не підтримує бібліотеки DirectX 12, тому чекати від нього реалістичной картинки непотрібно. За рахунок своєї легкості (він займає 10gb простору, в той час як Unreal Engine 40gb) на ньому можна створювати як 2D так і 3D продукти під різні платформи. В останніх оновленнях в даному рушію було додано магазин об’єктів для розробників. За технічну і логічну частину в даному рушії відповідає C# та його бібліотеки. Проте щоб створити скрипт для рушію, необхідно щоб клас наслідувався від MonoBehavior - інструменту Unity3d. Цінова політика дещо схожа на Unreal, проте відкритий код для розробників доступним не стає.

2.2 Вибір пайплайну для візуальної частини продукту

Взагалі що таке пайплайн? Це ланцюжок виробництва 3D графіки від ідеї до кінцевого продукту. Самий відомий із таких чітко передає зображення Енді Бейна:

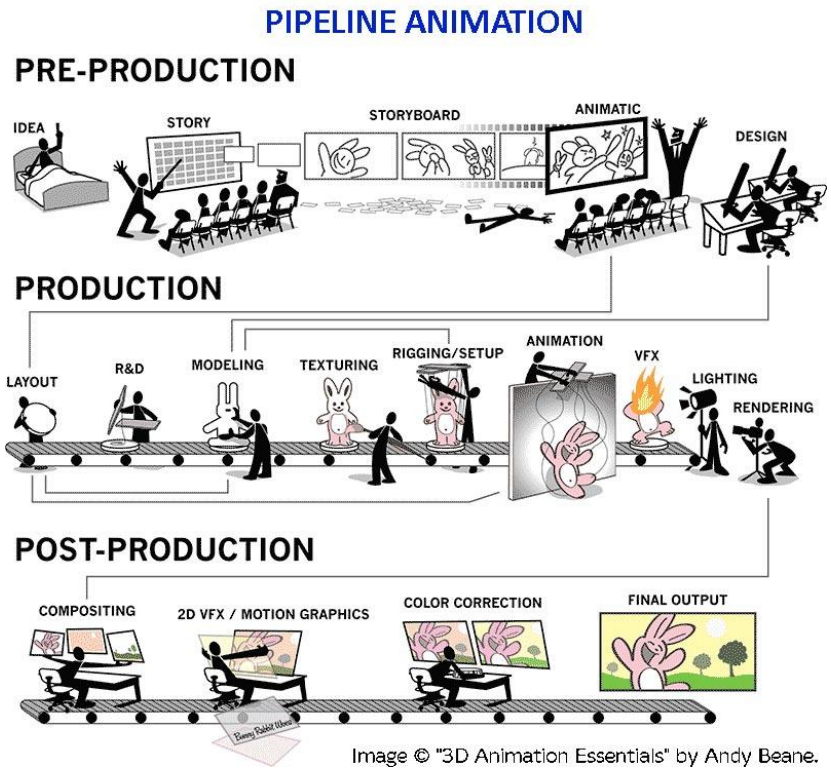


Рисунок 2.5 - схема самого розповсюдженого пайплайну

Спочатку створюється ідея, яка згодом збільшується, розширюється та покращується(pre-production). На основі цих напрацювань створюється дизайн чи концепт арт об’єкту, локації чи персонажу. Наступним етапом є тривимірне моделювання та подальше текстурування і анімування за необхідності. Це основний етап який називається production. На цьому етапі і використовують різні підходи та інструменти для створення 3D об’єктів. Наприклад для створення жорстких, структурно незмінних об’єктів таких як зброя, транспорт, будівлі використовують метод моделювання hard-surface modeling. В ньому моделювання відбувається за допомогою полігонів на основі креслень(якщо такі є), або концепту.

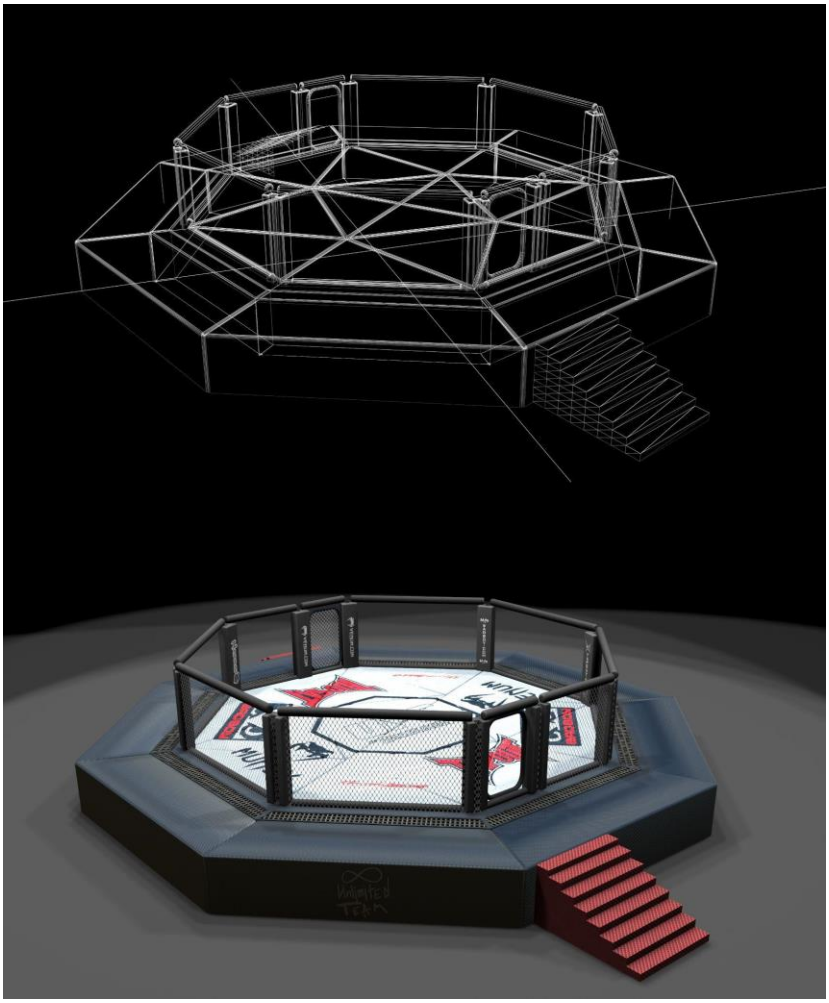


Рисунок 2.6 - приклад створеної моделі методом hard-surface modeling

Дана модель була створена за допомогою пакету 3Ds max, а також з використанням інструментарію Substance Painter.

Проте, якщо діло стосується персонажів усе різко змінюється. Перш за все створюється високополігональний скульпт майбутньої моделі. Найпоширеніші інструменти для цього: Zbrush або Blender3D. Далі необхідно створити ретопологію. Ретопологія - це процес зміни об’ємої сітки 3D моделі на менш деталізовану та оптимізовану. Перш за все це робиться для оптимізації майбутньой гри та зручності роботи з моделлю.

Рисунок 2.7 - візуальний приклад ретопології

Це можна виконати в багатьох 3д пакетах наприклад 3Ds max, Blender3D чи Maya, але досить часто для цих цілей використовують 3D Coat. Наступним етапом є запікання дрібної деталізації скульпта на готову низькополігональну модель в pbr карти. Саме цей підхід дає найбільший приріст продуктивності, при незначному погіршенні результату, або ж зовсім без нього. Це можна виконати в будь-якому 3D пакеті. Проте є і інший спосіб: створення дрібної деталізації на готову низькополігональну модель власноруч, з використанням Substance Painter, який може генерувати normal map, bump mup. Проте цей спосіб використовують у виключно стилізованій графіці, яку складно назвати реалістичною. Після цього іде етап текстурування та анімування моделі. Для першого можна використовувати як Photoshop так і будь який другий редактор, або ж візуалізацію за допомогою спеціалізованого рушію. Для анімування ААА студії використовують підхід motion capture. Камера за допомогою маркерів фіксує рухи актора, і ретранслює на скелет 3д моделі. Цей використовується як і в сучасній кіноіндустрії так і в ігровій.

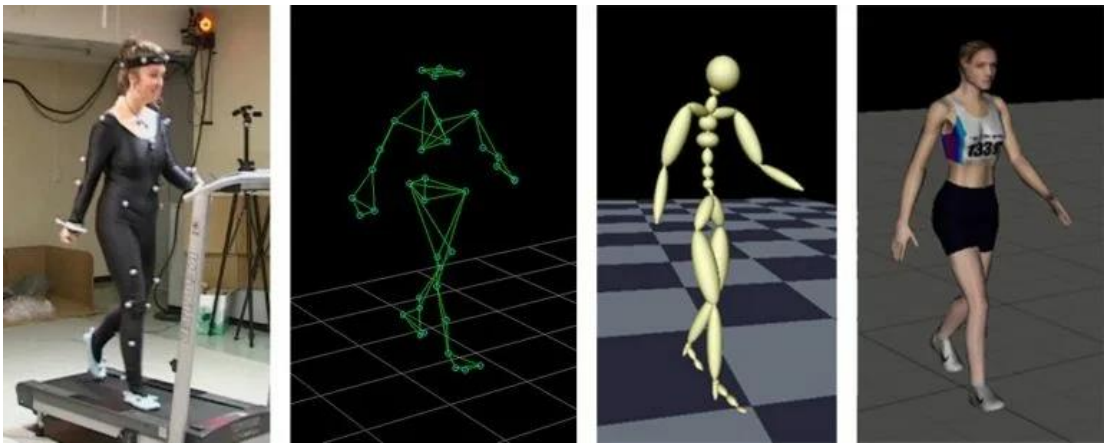


Рисунок 2.8 - приклад використання технології motion capture

Етап post-production в ігровій індустрії зазвичай виконується в самому ігровому рушії за допомогою текстурних шейдерів та пост-обробки. Шейдери надають матеріалам неповторного, та за необхідності стилізованого вигляду. Постобробка зазвичай накладається на вже готовий кадр і лише вносить невелику корекцію, змінює тон, настрій та атмосферу сцени. Наприклад за допомогою постобробки та vfx ефектів розробники можуть створювати цілі погодні цикли, які змінюються динамічно.

Рисунок 2.9 - приклад постобробки з ефектом туману на прикладі гри Valheim

Але що робити у випадку інді розробки? Адже не кожна студія може дозволити собі високоякісну акторську гру та технологію motion capture. На допомогу приходить безкоштовний сервіс mixamo з дуже обширною

бібліотекою анімацій. Крім того, туди можна завантажити власні об’єкти, та анімувати їх на свій смак. Але в цілому, структура та план розробки не змінюється, обирається лише інструментарій індивідуально для кожної команди розробників.

2.3 Дизайн ігрових рівнів

Який підхід буде використовуватись для створення ігрового світу? Все залежить від масштабу створюваної гри. Наприклад для малої інді гри на платформу PC та Android часто використовують prefabs для проектування ігрового світу. Це невелика кількість об’єктів які ми копіюємо та виставляємо на нашій сцені так як нам необхідно. Принцип схожий на конструктор лего:

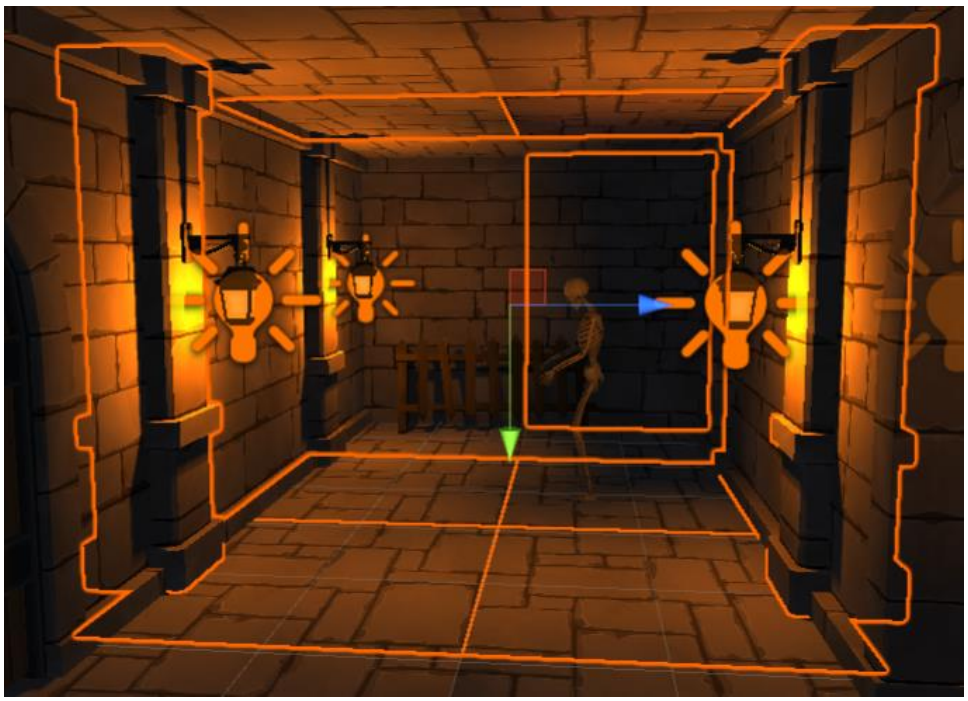


Рисунок 2.10 - приклад локації з використання префабів

Даний підхід використовується в тих випадках коли візуальна частина або потребує стилізації(як на рис. 2), або у випадку коли локація генерується

процедурно. У цьому випадку один префаб виступає tile - однією клітиною ігрової сітки. Але у цього підходу є і мінус: неможливість або дуже складна реалізація ландшафту з плавними перепадами по осі Z. Другий мінус: досить великі локації створені за допомогою префабів створюють суттєве навантаження на систему, тому це не самий кращий підхід для оптимізації.

Як альтернативу використовують змішаний тип створення ігрових рівнів. Це необхідно для створення відкритого ігрового світу(рівень у якому немає додаткових завантажень, підгрузок и тп.). За основу береться terrain - динамічна сітка. За допомогою пензлів різної форми є можливість змінювати форму ландашафту: видавлювати, втоплювати, скручувати згладжувати.

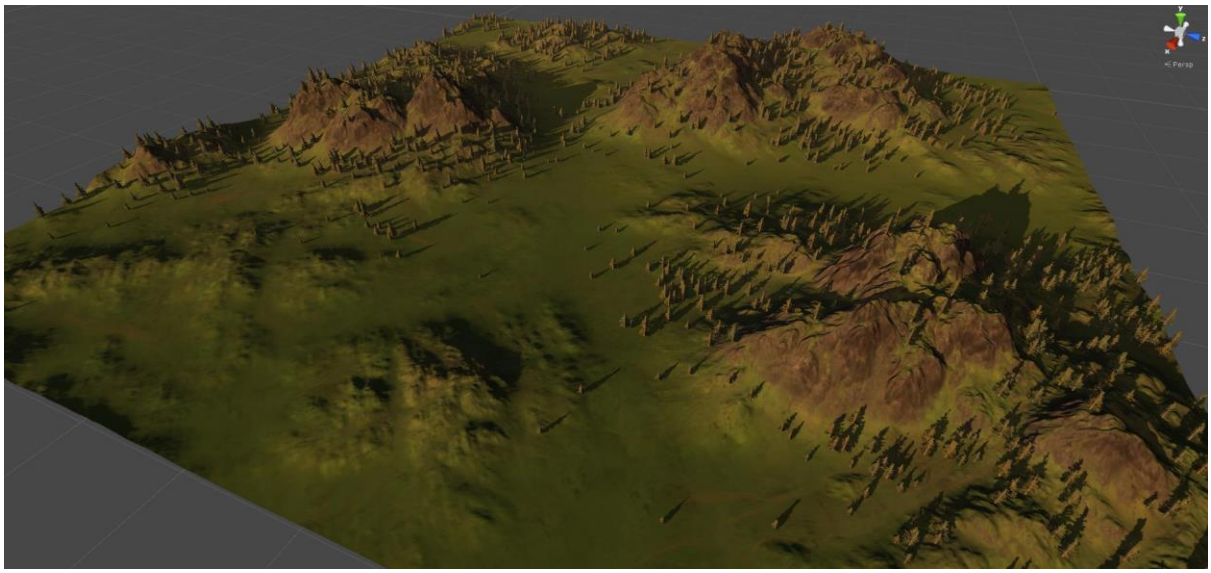


Рисунок 2.11 - приклад ігрового світу з використанням terrain

Проте чому ж цей підхід називається змішаним? Все тому, що префаби досі використовуються. Будь-який ігровий об’єкт ігрового світу дерево, паркан, квіти, зброя, усе що використовується більше одного разу слід використовувати як префаб а не як унікальний 3d mesh. Це необхідно для більш розумного використання ресурсів системи.

Наступне на що слід зробити акцент - корекція освітлення та postprocessing. Це технологія, яка дає можливість покращити вигляд візуальної

частини проекту уже після відрисовки кадру відеокартою. Постобробка це доволі важливий інструмент геймдизайнера. За його допомогою розробники можуть передати настрій ігрового рівню, його атмосферу. Безпечно там чи ні, холодно чи жарко сухо або волого усі ці параметри досить складно передати лише з використанням префабів.

ПРОГРАМНА РЕАЛІЗАЦІЯ ПРОДУКТУ

3.1 Дизайн ігрового рівня

Перш за все, створимо ігровий рівень, на якому буде можливість тестувати розроблювані у майбутньому ігрові модулі. Створювати його будемо за допомогою такого пулу префабів:

Рисунок 3.1 - пул префабів для створення локації

Але перед тим як приступити до розстановки об’єктів зв’яжемо об’єкт Lamp_02 з точковим освітленням та налаштуємо колір та інтенсивність.

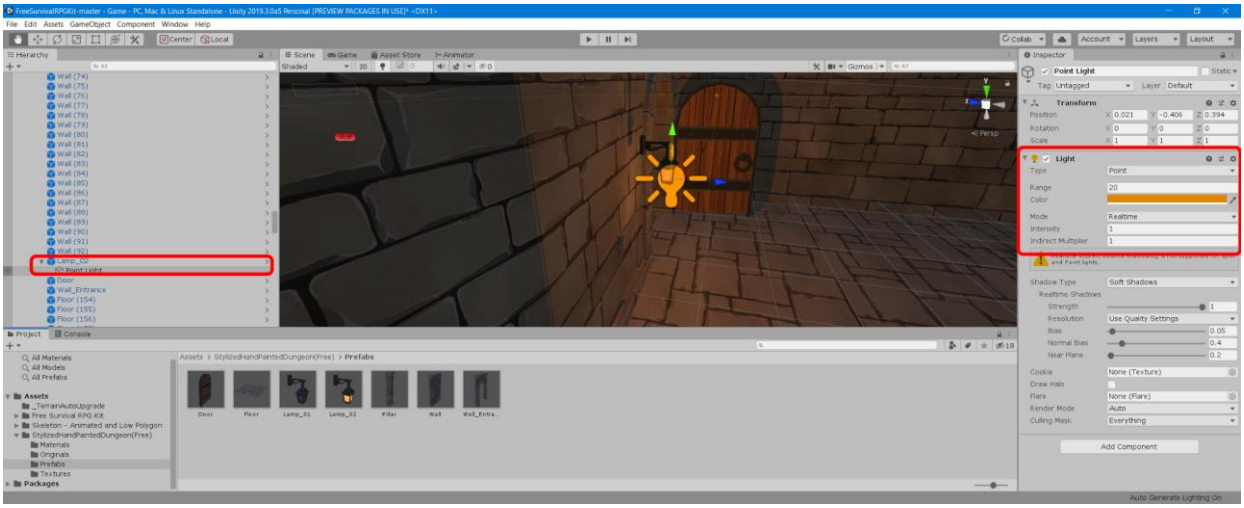


Рисунок 3.2 - налаштування освітлення

Наступним кроком створимо ігрову арену, для тестування основних дій головного героя та кімнату респавну, де гравець зможе з’являтися після невдалої спроби.

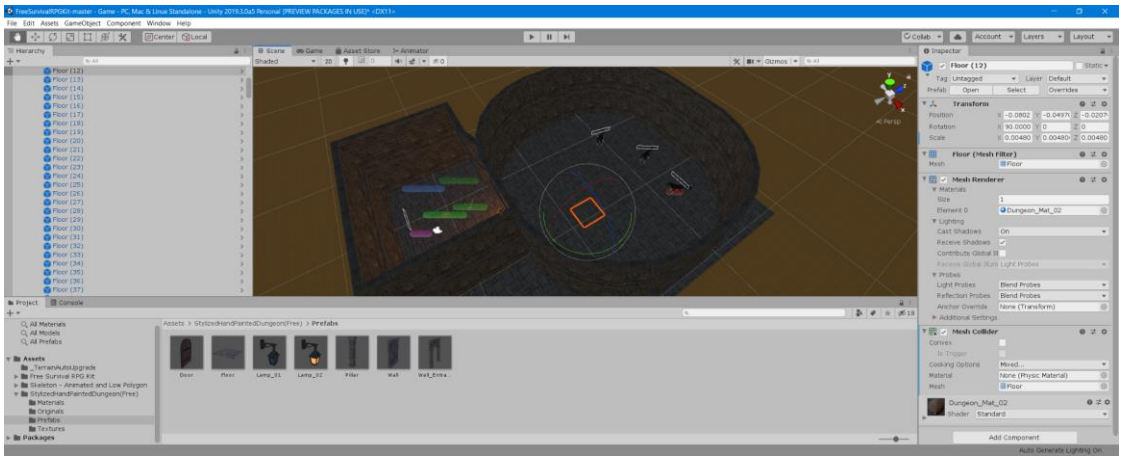


Рисунок 3.3 - Вигляд ігрової локації

Перед додаванням нових префабів та зміні інснуючих не потрібно забувати про обов’язкову властивість кожного статичного префабу, а саме про MeshCollider. Ця фізична властивість надає префабу об’єм і прибирає можливість проходити крізь нього наскрізь:

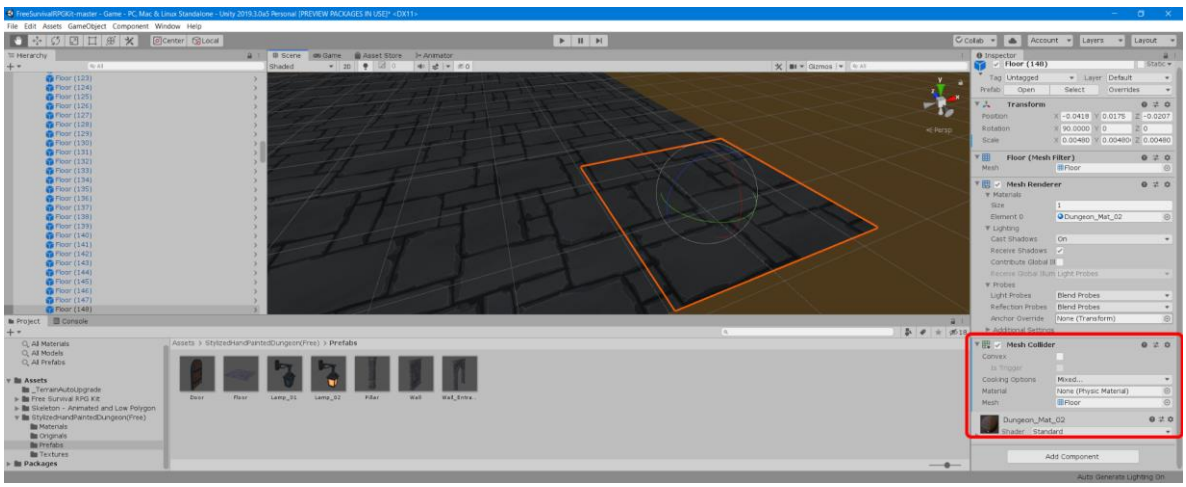


Рисунок 3.4 - властивість MeshCollider

3.2 Створення префабу персонажу гравця та логики його поведінки

Персонаж, яким буде керувати гравець створюється з пустого ігрового об’єкту(Create Empty). Далі необхідно додати властивість NavMeshAgent та виставити тип агенту Humanoid. Це необхідно для подальшого додавання скелету людського типу і відповідних анімацій.

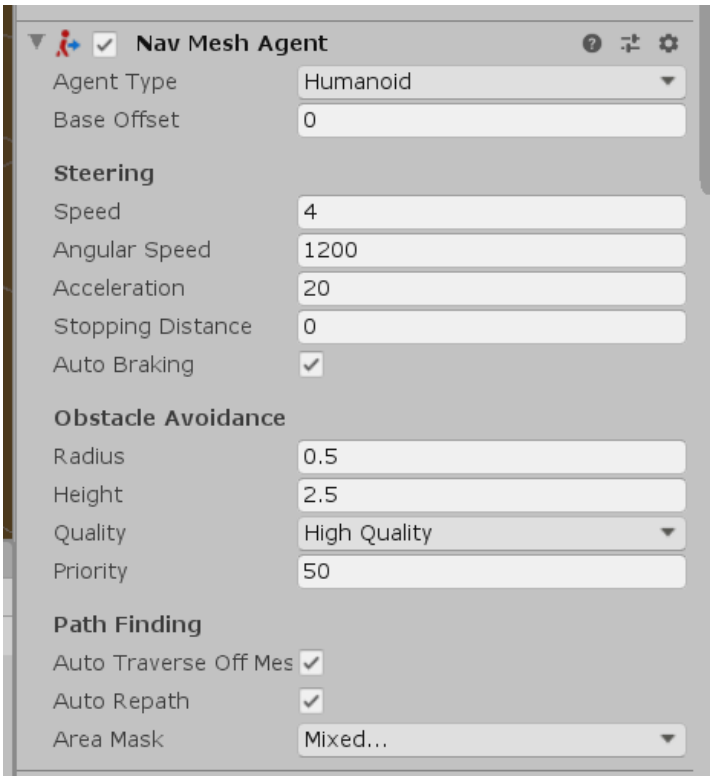


Рисунок 3.5 - властивість NavMeshAgent

На наступному етапі розробки необхідно змусити персонажа рухатись. Перш за все необхідно щоб fbx файл нашого персонажу мав всередині скелет, набір анімацій, та RIG самого мешу. Без цього набору неможливо створити рухи. Перевірити це можна таким чином: розгорнути fbx файл в папці з ресурсами гри та перевірити наявність необхідного.

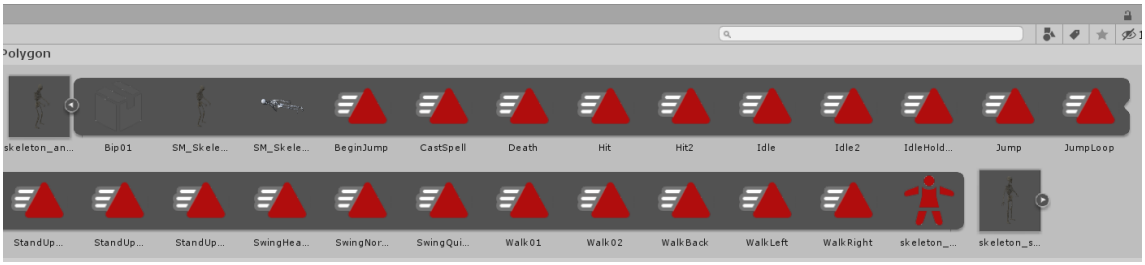


Рисунок 3.6 - розгорнутий fbx файл

Щоб наявні анімації могли змінюватись у правильному порядку, слід створити контроллер анімацій. Це набір блоків зі своїми зв'язками між ними які сигналізують про поточний стан персонажу. Якщо рушій фіксує будь-яку подію, то контроллер анімацій змінює необхідну булеву змінну(тригер) і це зумовлює перехід стану від поточного до наступного.

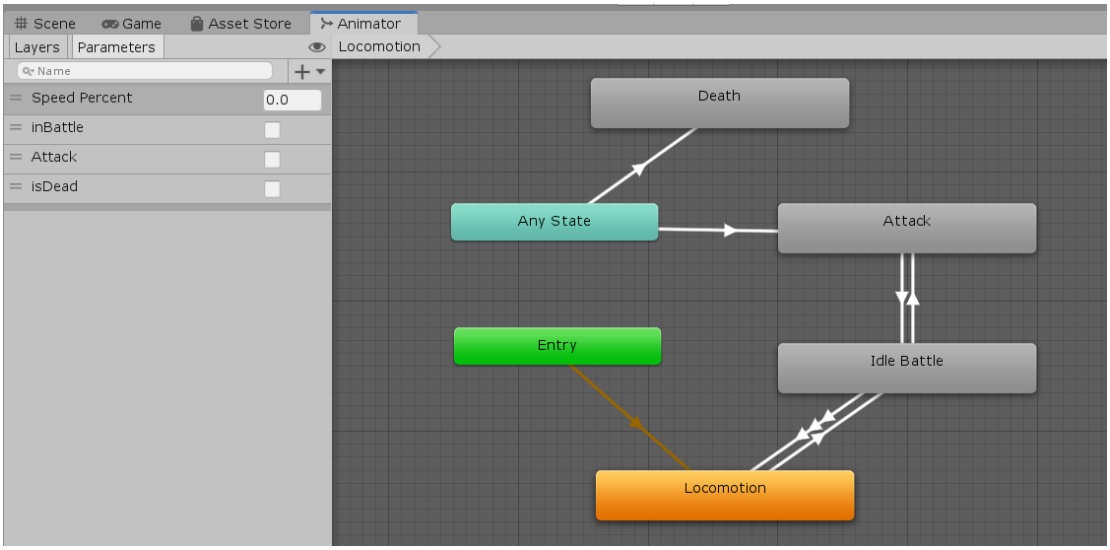


Рисунок 3.7 - Контроллер анімацій персонажа у розробленому прототипі

При ініціалізації рівню персонаж знаходиться в Т-позі, але через декілька секунд він переходить до Locomotion. Locomotion - це набір анімацій при русі та спокійному стані(без фіксованих подій).

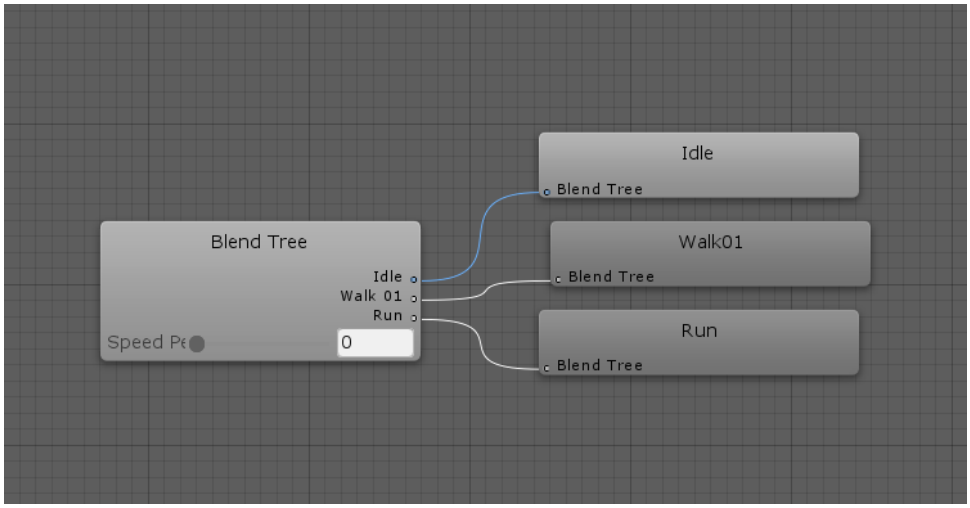


Рисунок 3.8 - Locomotion

В даному випадку персонаж має можливість програвати анімацію спокою(Idle), анімацію кроку та бігу відповідно. У разі наближення ворогів(входження ближче ніж радіус від pivot point префабу ворога) стан змінюється на Idle battle. В цьому стані персонаж має можливість програвати анімацію атаки, та у разі отримання великої кількості травм може перейти в стан смерті та подальшого респавну. Відбувається це через блок Any State.

У наступному етапі створюємо скрипти, за допомогою яких буде фіксуватись натискання кнопок, та змінюватися як координати персонажа та камери, так і тригери в контролері анімацій.

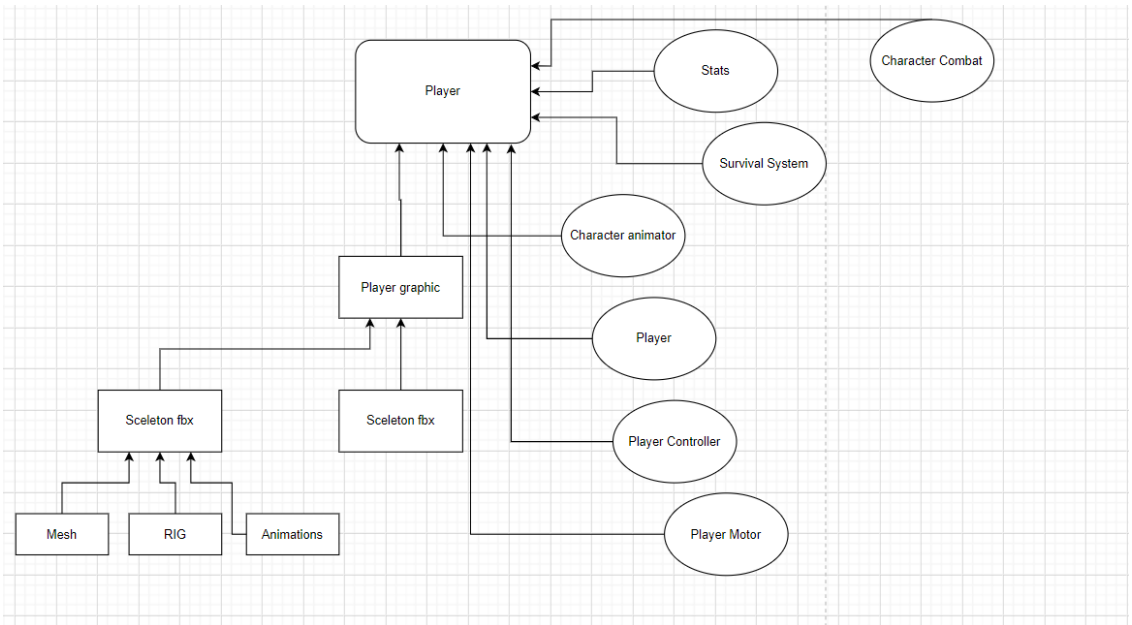


Рисунок 3.9 - Схема префабу Player

Перший складний модуль Player Graphics складається, як уже було описано вище з моделі анімацій та контроллера анімацій. Скрипти, які відповідають за логіку представлені у схемі у вигляді еліпсів. Розглянемо що робить кожен з них:

1. Character animator - скрипт що відповідає за зміну тригерів для анімацій(Рис. 3.10)
2. Player - відповідає за ініціалізацію персонажа та його скриптів, мінікарти, контролює стан смерті та подальшого респавну.(Рис. 3.11)
3. Player Motor - скрипт, що відповідає за рух персонажу, та поворот камери за допомогою миші(Рис. 3.12)
4. Player Stats - скрипт, що контролює основні параметри героя та їх динамічні зміни: здоров’я, енергія, зміна атаки при наявних обладунках. (Рис. 3.13)
5. Character combat - скрипт що регулює бойову систему, відтворює перехід від спокійного стану у бойовий, відповідає за можливість відтворювати удари героя, наносити та отримувати урон(Рис. 3.14)

```
protected virtual void OnAttack() {
    animator.SetBool("Attack", true);
    animator.SetBool("inBattle", false);
}

protected virtual void OnBattle()
{
    animator.SetBool("inBattle", true);
}

protected virtual void OnNormal()
{
    animator.SetBool("Attack", false);
    animator.SetBool("inBattle", false);
}

protected virtual void OnDeath()
{
    animator.SetBool("isDead", true);
}
```

Рис. 3.10 - Character animator

```
void Start()
{
    // встановлює положення мінікарти на старті
    GameObject.Find("MinimapCamera").GetComponent<Minimap>().player = this.gameObject.transform;

    // якщо параметр зрово'я рівний 0 герой переходить в стан смерті
    playerStats.OnHealthReachedZero += Die;

}

void Die()
{
    // функція смерті
    playerCombatManager.Death();

    // показ інтерфейсу при смерті
    GameObject.Find("GameManager").GetComponent<RespawnManager>().DeathUI.SetActive(true);

    // відключення ігрового інтерфейсу на момент смерті
    GetComponent<CharacterCombat>().healthSlider.gameObject.SetActive(false);
}
```

Рисунок 3.11 - Player

```
public void WASDMove(bool usingJoystick)
{
    // Перевірка з чого натискаємо кнопки
    if (!usingJoystick)
    {
        // WASD керування
        Vector3 goal = transform.position
              + Camera.main.transform.right * Input.GetAxis("Horizontal")
              + Camera.main.transform.forward * Input.GetAxis("Vertical");

        // саме переміщення за допомогою NavMeshAgent
        agent.SetDestination(goal);
    }
```

Рисунок 3.12 - Character motor

```
void OnEquipmentChanged(Equipment newItem, Equipment oldItem)
{
    if (newItem != null) {
        armor.AddModifier (newItem.armorModifier);
        damage.AddModifier (newItem.damageModifier);
    }

    if (oldItem != null)
    {
        armor.RemoveModifier(oldItem.armorModifier);
        damage.RemoveModifier(oldItem.armorModifier);
    }
}
```

Рисунок 3.13 - Character Stats

```
public void Attack (CharacterStats enemyStats)
{
    if (attackCountdown <= 0f)
    {
        this.enemyStats = enemyStats;
        attackCountdown = 1f / attackRate;

        StartCoroutine(DoDamage(enemyStats,.6f));

        if (OnAttack != null) {
            OnAttack ();
        }
    }
}
```

Рисунок 3.14 - функція атаки з класу Character Combat

3.3 Створення ворогів

Вороги також будуть створюватись за схожою структурою префабу, проте є деякі відмінності:

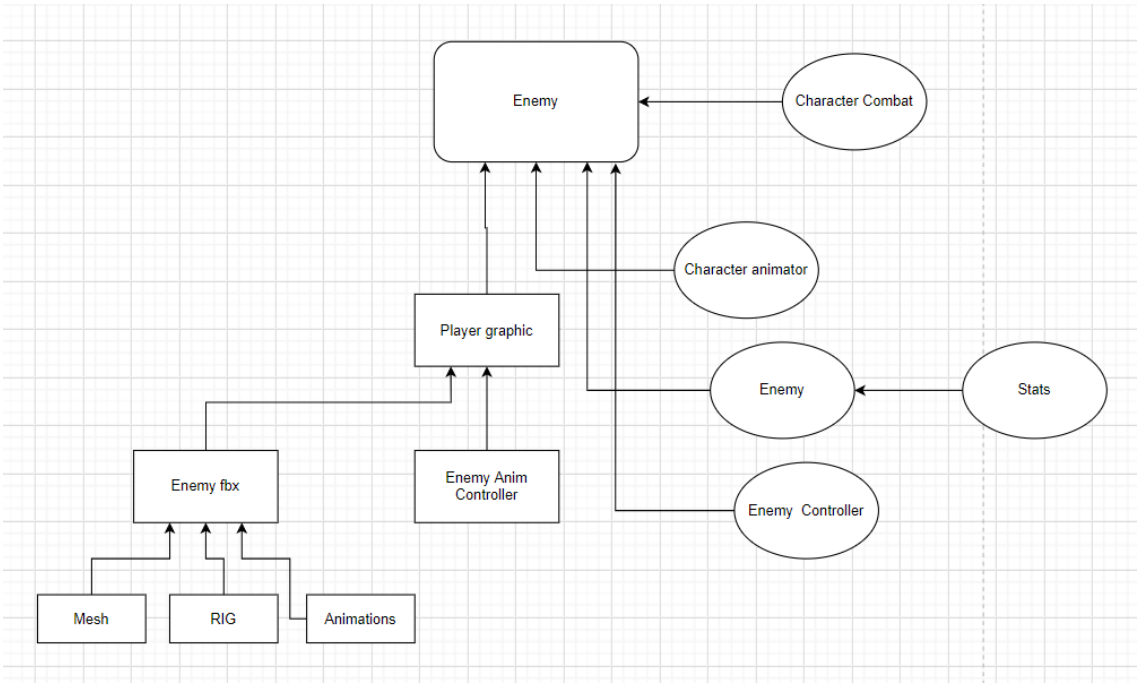


Рисунок 3.15 - схема префабу Enemy

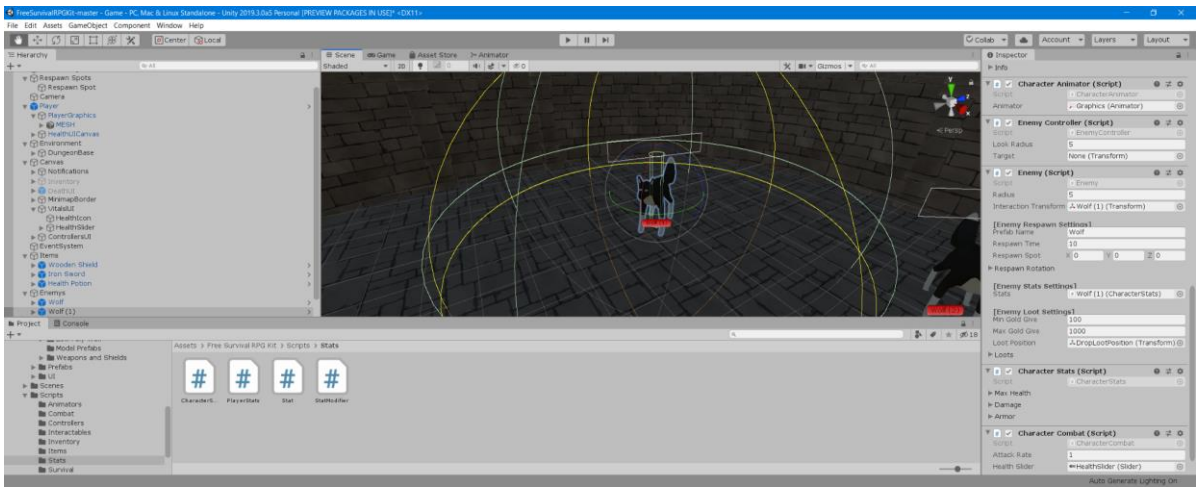


Рисунок 3.16 - відображення схеми Enemy у рушії

Логіка у даному випадку дещо простіша. Ворог також має можливість переміщатися з точки на точку. У разі, коли персонаж входить в зону його реакції він переходить у бойовий стан та починає атакувати головного героя. Системи предметів у ворогів немає тому клас Stats є частиною головного класу Enemy і динамічної зміни параметрів тут бути не може. Цікавим є те, що при смерті Enemy головний герой отримує деяку кількість золота, а також на землю випадає випадковий предмет, з пулу можливих. Реалізовано це таким чином:

```
void DropLoot()
{
    // вибір випадкової кількості золота та його видача
    int goldValue = Random.Range(minGoldGive, maxGoldGive);
    GameObject.Find("GameManager").GetComponent<Inventory>().gold += goldValue;

    // вибір випадкового предмету зі стеку предметів
    int lootIndex = Random.Range(0, loots.Length);
    GameObject currentLoot = loots[lootIndex];

    Instantiate(currentLoot, lootPosition.position, Quaternion.identity);
    //Debug.Log("Drop Loot: " + currentLoot.name + ", Gold Drop: " + goldValue);
}
```

Рисунок 3.17 - функція що відповідає за випадання предметів з Enemy

А ось так виглядає цей аспект у самій грі:

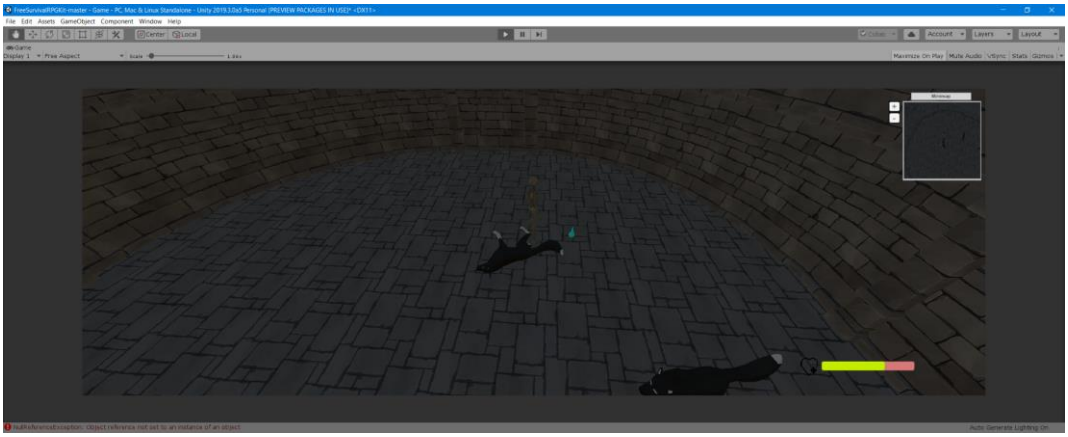


Рис. 3.18 - предмет з ворога

3.4 Створення інвентаря та предметів

Інвентар представлений у вигляді колекції яка може зберігати об’єкти класу Item. У разі натискання кнопки E у радіуси предмету викликається функція Interact, яка в свою чергу викликає функцію PickUp:

```
public class ItemPickup : Interactable {

    public Item item;   // об'єкт класу Item який ми можемо зберігати в інвентарі

    // при взаємодії з ним
    public override void Interact()
    {
        base.Interact();

        PickUp();
    }

    // функція що видповідає за підбор
    void PickUp ()
    {
        Debug.Log("Picking up " + item.name);
        Inventory.instance.Add(item);   // додавання поточного елементу в коллекцію

        Destroy(gameObject);    // видалення префабу який був підібраний
    }
```

Рисунок 3.19 - функція підбору та зберігання предметів

Сам інвентар у грі представлений у вигляді комірок. У разі якщо вільні комірки закінчаться, гравець отримає попередження, що більше нічого підняти та зберегти він не зможе.

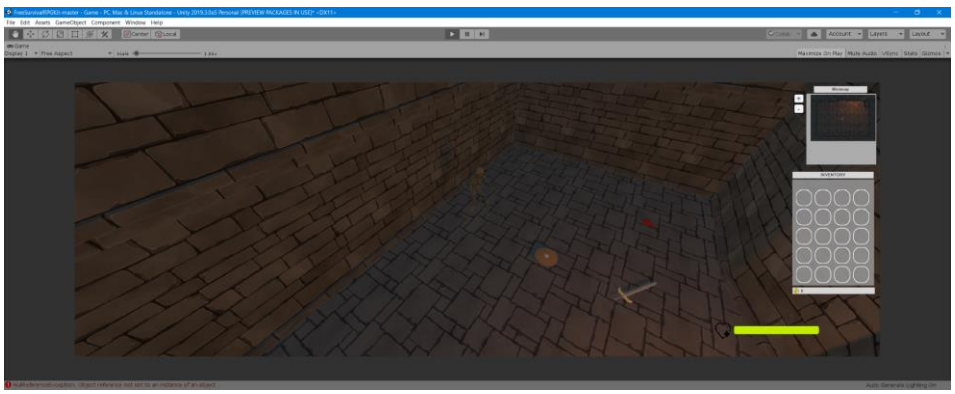


Рисунок 3.20 - вигляд пустого інвентаря головного героя

Прямо після початку гри гравець може підібрати початкові обладунки, та склянку, що відновлює здоров’я.

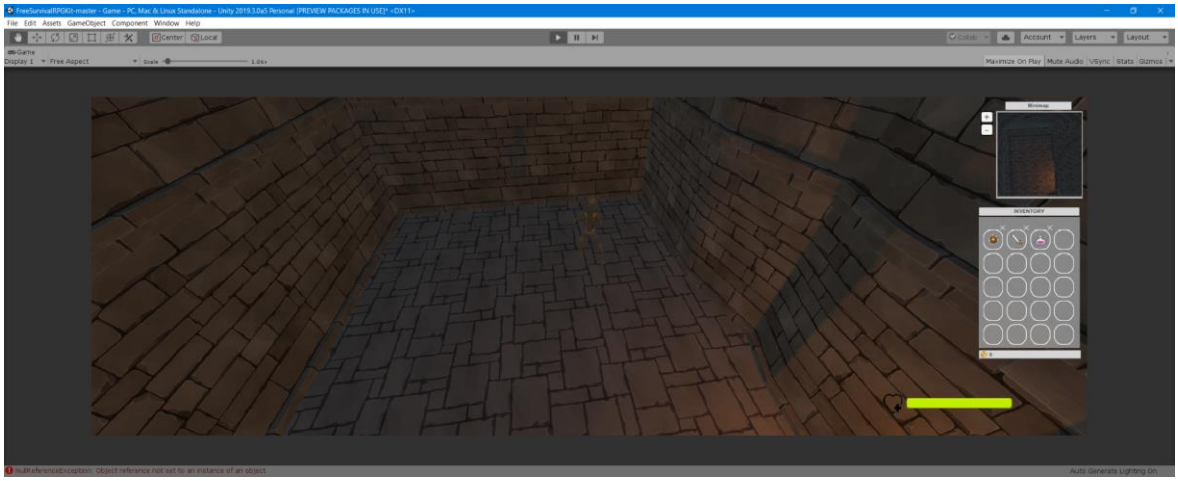


Рисунок 3.21 - вигляд інвентарю після підбору декількох предметів

```
// Додавання предметів і подальше зберігання
public void Add (Item item)
{
    if (item.showInInventory) {
        if (items.Count >= space) {
            Debug.Log ("Not enough room.");
            return;
        }

        items.Add (item);

        if (onItemChangedCallback != null)
            onItemChangedCallback.Invoke ();
    }
}

// Видалення з інвентарю
public void Remove (Item item)
{
    items.Remove(item);

    if (onItemChangedCallback != null)
        onItemChangedCallback.Invoke();
}
```

Рисунок 3.22 - функції класу Inventory відповідні за зберігання та видалення предметів

ВИСНОВОК

У даній роботі було проаналізовано сучасну ігрову індустрію, були розглянуті найвдаліші представники та розібрані на складові механіки.

Наступним етапом роботи було розглянуто приклади правильної ігрової документації та їх написання. Було створено власний ігровий дизайн документ у вигляді десятисторінника, на основі якої було поставлений план роботи.

Далі відбувався пошук доступних рішень, яких у сучасній ігровій індустрії безліч. Підбір було зупинено на рушії Unity3d Engin, із-за його легкості, малому навантаженні та безкоштовній основі у разі використання його для навчання.

Також були розглянуті актуальні пайплайни по розробці комп’ютерної 3d графіки, було розглянуто інструментарій для цього, а також обрано ті інструменти, за допомогою яких створювалась графіка для прототипу.

Прототип комп’ютерної гри було створено згідно поставленого плану. За основу було взято безкоштовний Asset, який створений самими розробниками Unity, та який знайомить з найпопулярнішими та базовими можливостями рушію.

ЛІТЕРАТУРА

1. Скота Роджерса “Level up. The guid to great video game design”
2. Джо Хокінг “Unity in Action: Multiplatform Game Development in C#”
3. Кріс Дікінсон “Unity 5 Game optimization”
4. Ернест Адамс “Fundamentals of Game Design”
5. Кріс Легаспі “Anatomy for 3D artists: the essential guide for CG professionals”

ДОДАТОК А

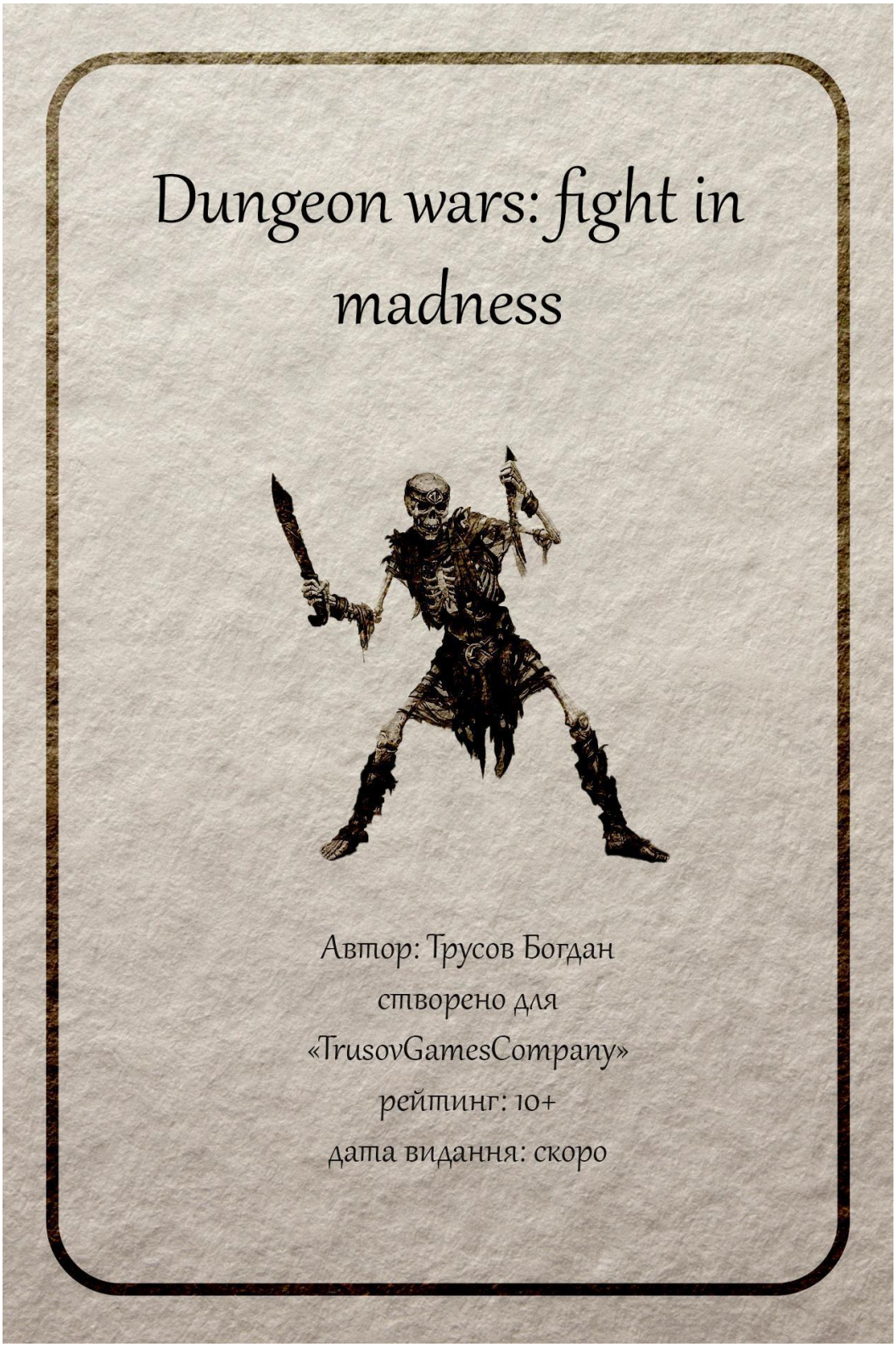
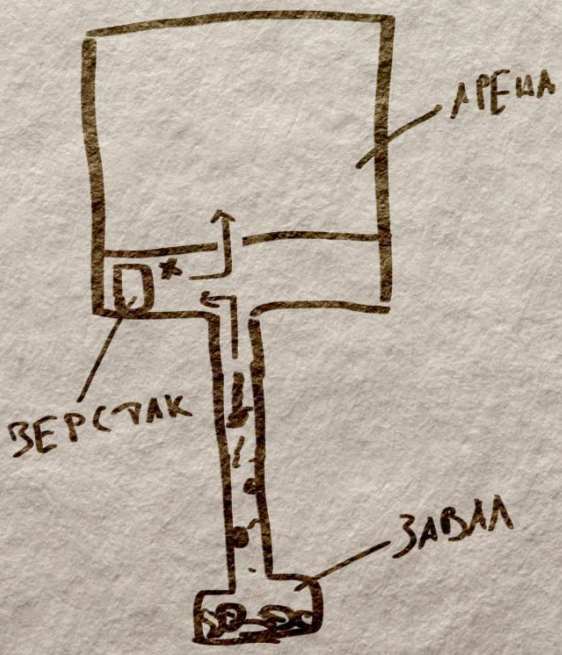


Рисунок 4.1 - перша сторінка

Як такого сюжету гра не матиме. Головний герой - безіменний скелет із расси вічних який прокидається на арені біля завалу. Він розуміє що тікати нікуди і єдиний шанс отримати спокій це вистояти 10 хвиль на арені та перемогти конкурентів.



Основним місцем подій є арена древні. Головний герой прокидається біля завалу, проходить коротке озайомлення з ігровим процесом, далі потрапляє в вестибюль де його чекає привид - командувач арени та верстак

Рисунок 4.2 - друга сторінка

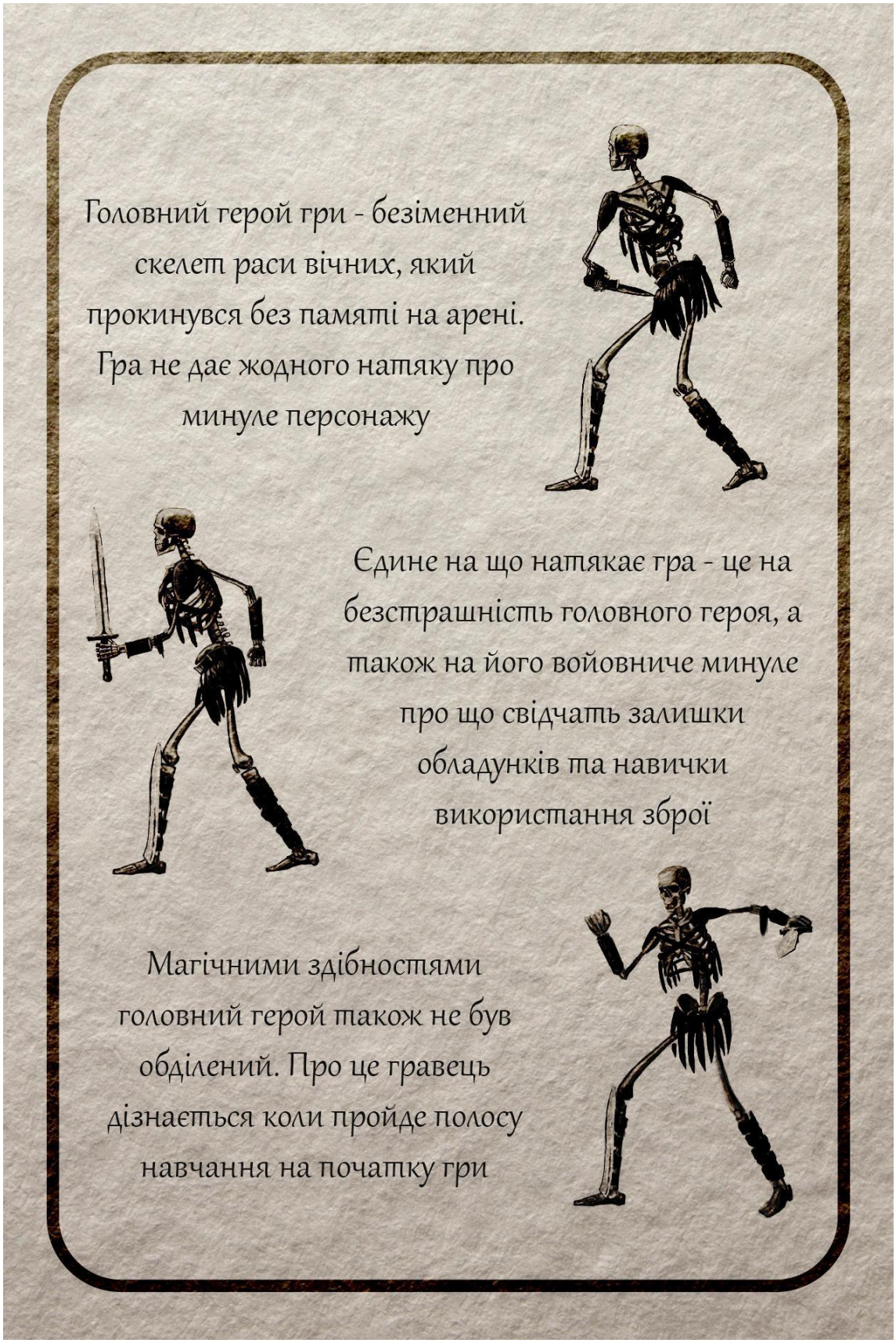


Рисунок 4.3 - третя сторінка

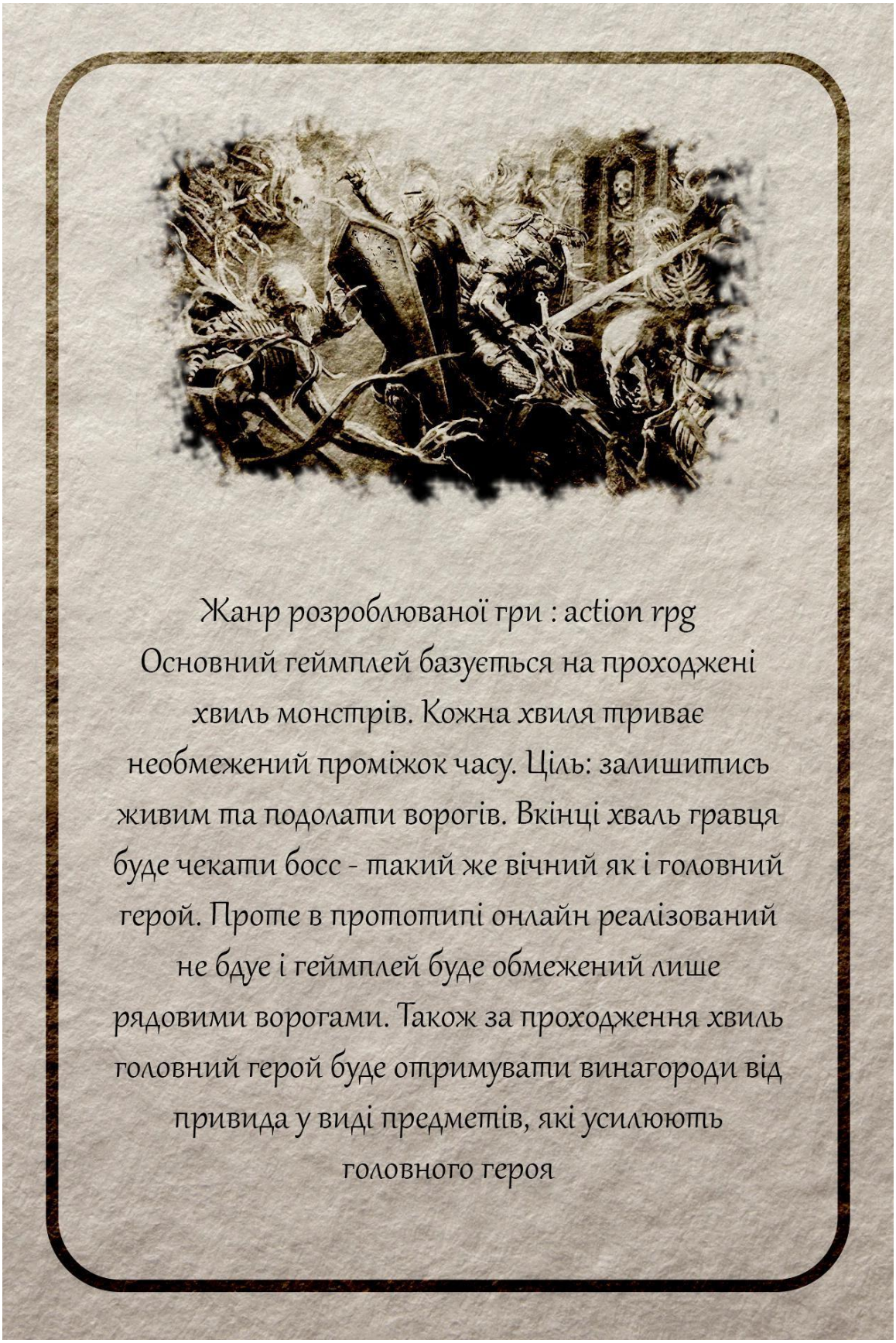


Рисунок 4.4 - четверта сторінка

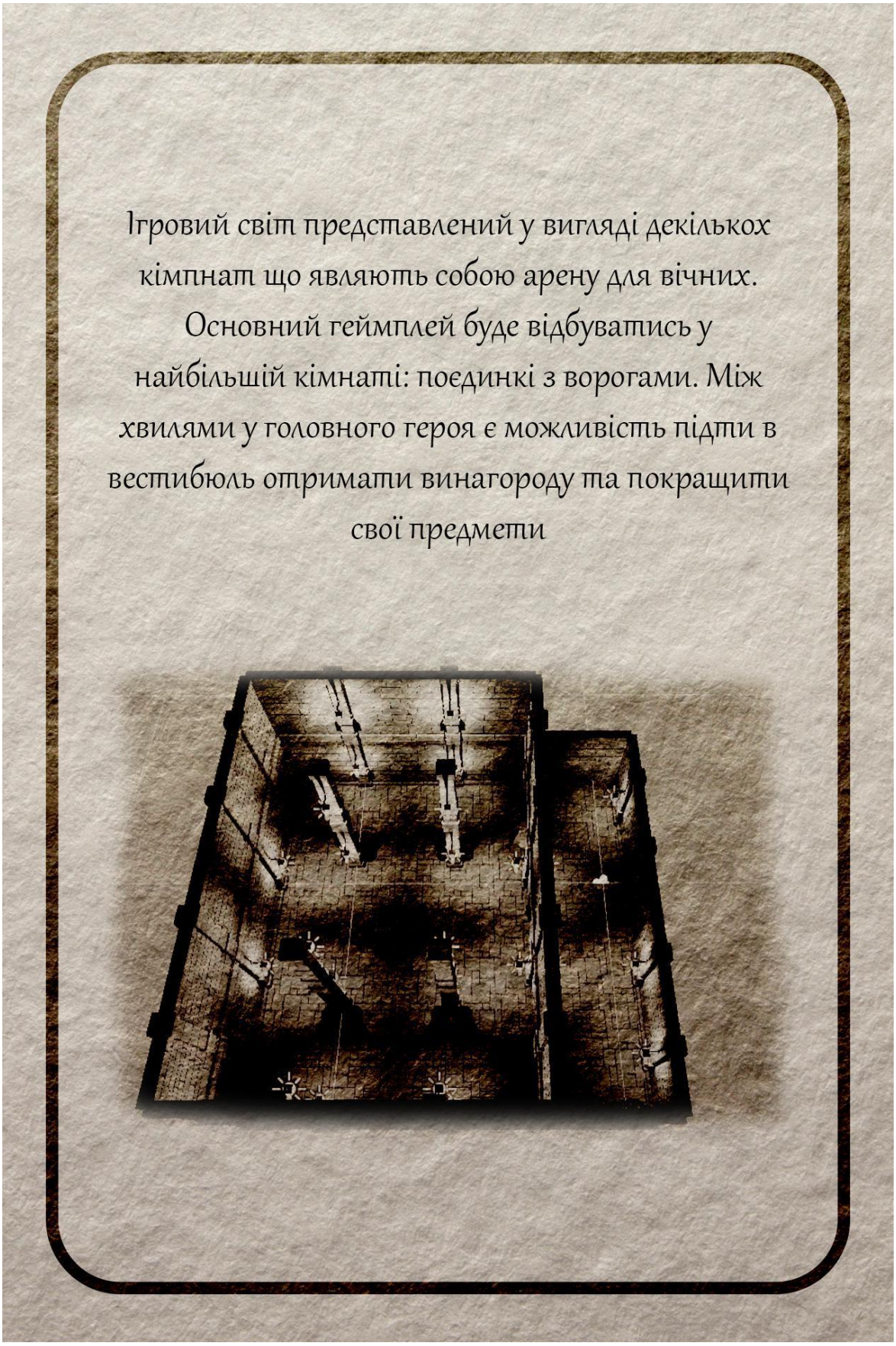


Рисунок 4.5 - п’ята сторінка

Основна мета головного героя від початку гри до її завершення це отримання спокою. Це говориться на початку і це повторюється в кінці після перемоги. Сам процес проходження ігрового шляху займає небагато часу і саме це спонукає гравця виконати вимогу головного героя. Також що слід пам'ятати: окрім мети персонажа гравцем керує жага покращити свого головного персонажа і зробити його якомога сильнішим і витривалішим.

Рисунок 4.6 - шоста сторінка

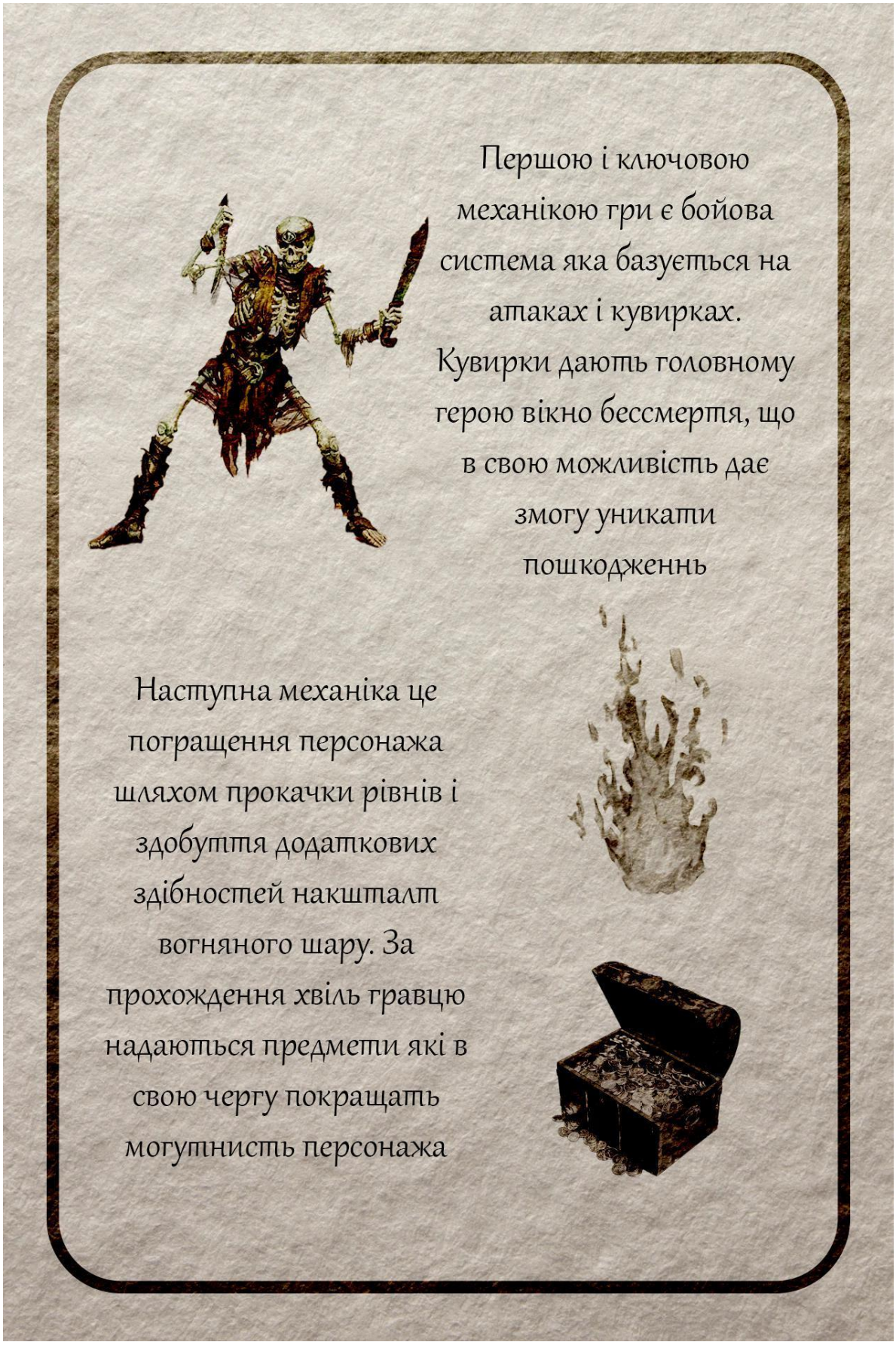


Рисунок 4.7 - сьома сторінка

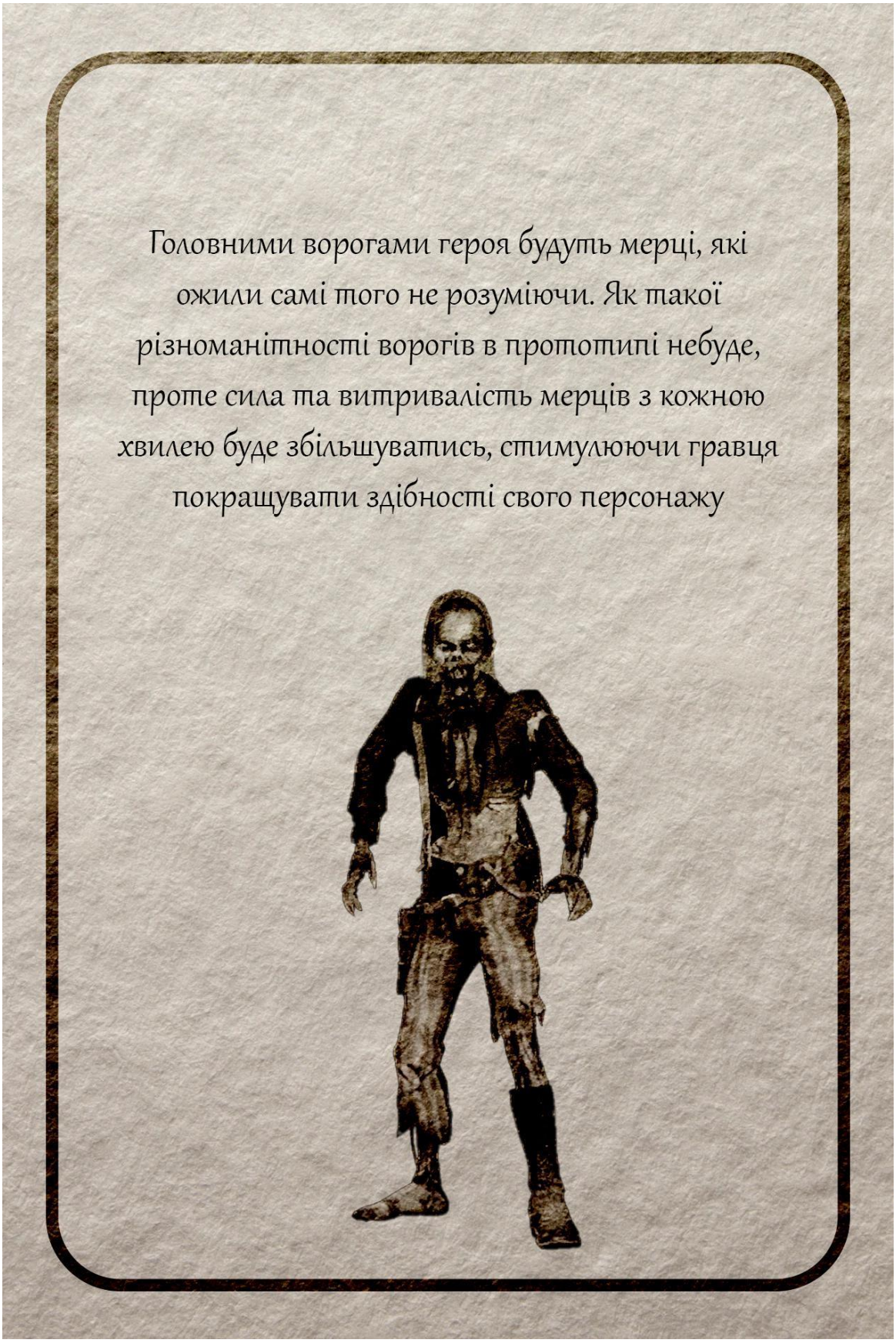


Рисунок 4.8 - восьма сторінка

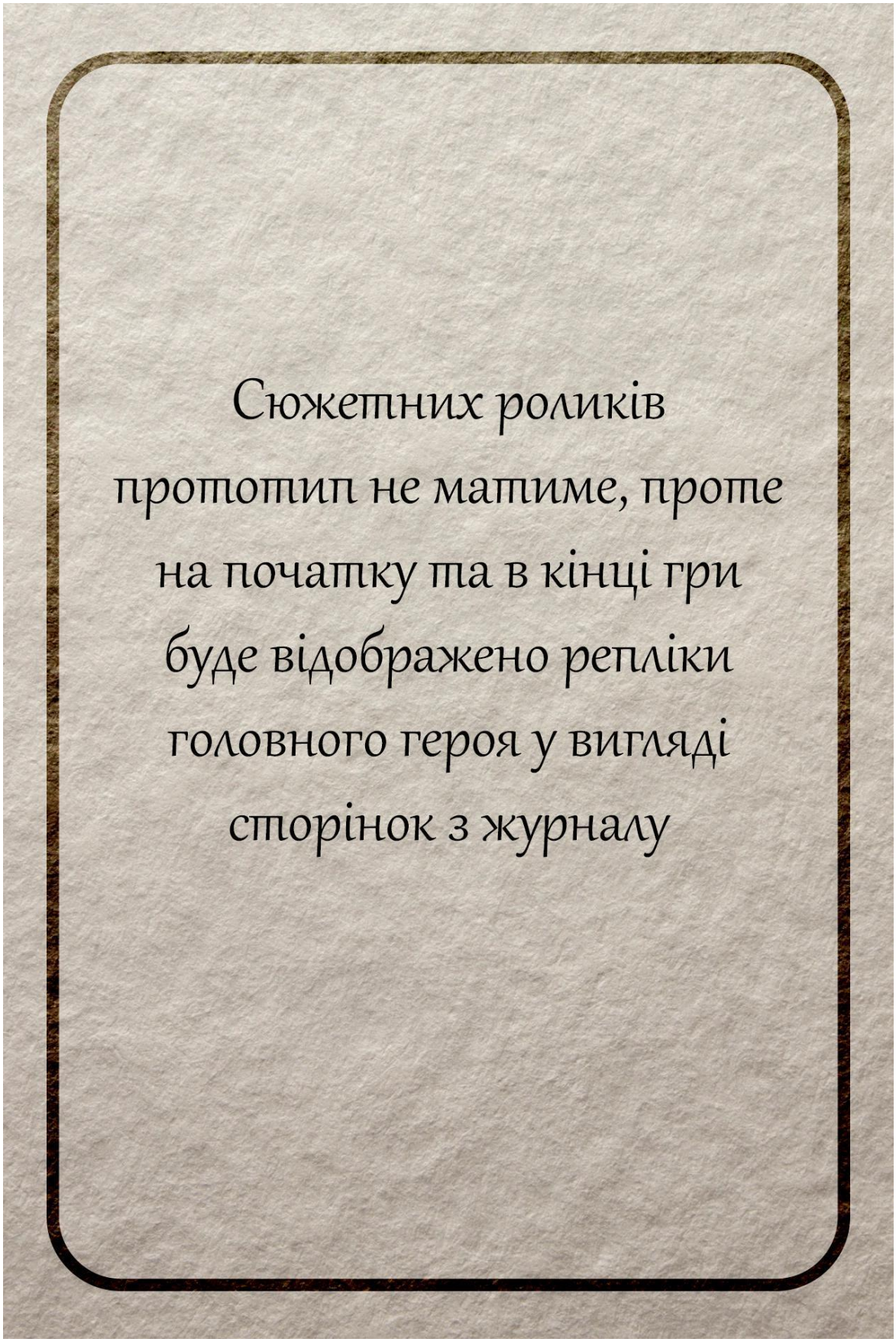


Рисунок 4.9 - дев’ята сторінка

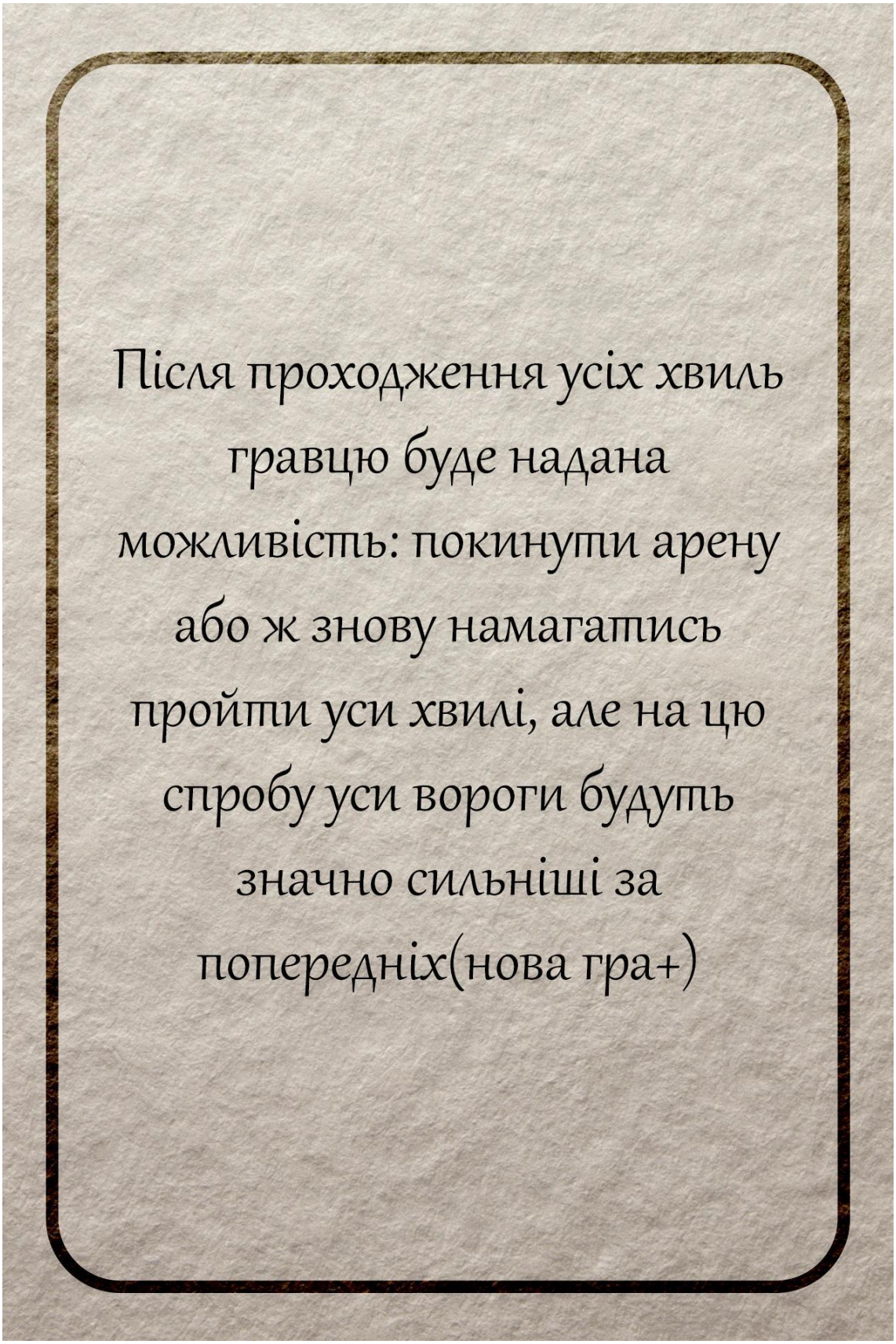


Рисунок 4.10 - десята сторінка

ДОДАТОК Б

```
public class Player : MonoBehaviour {

#region Singleton

public static Player instance;

void Awake ()
{
instance = this;
}

    #endregion

    public CharacterCombat playerCombatManager;
    public PlayerStats playerStats;

    void Start()
    {


GameObject.Find("MinimapCamera").GetComponent<Minimap>().player =
this.gameObject.transform;

        смерті
        playerStats.OnHealthReachedZero += Die;

    }

                                                                void Die()
    {
                playerCombatManager.Death();


GameObject.Find("GameManager").GetComponent<RespawnManager>().DeathUI.
SetActive(true);

GetComponent<CharacterCombat>().healthSlider.gameObject.SetActive(fals
e);
    }
}
```

```
public class PlayerAnimator : CharacterAnimator {

void Awake() {
EquipmentManager.instance.onEquipmentChanged += OnEquipmentChanged;
}

protected override void Start() {
base.Start ();
}

protected override void OnAttack() {

base.OnAttack ();
}

void OnEquipmentChanged(Equipment newItem, Equipment oldItem) {
if (oldItem != null) {
if (oldItem.equipSlot == EquipmentSlot.Weapon) {
}
if (oldItem.equipSlot == EquipmentSlot.Shield) {
           }
       }

if (newItem != null) {
if (newItem.equipSlot == EquipmentSlot.Weapon) {}

if (newItem.equipSlot == EquipmentSlot.Shield) {

            }
        }
                                                                                }
}
```

```
public class CharacterAnimator : MonoBehaviour {

                                                    public Animator animator;

                                                   NavMeshAgent navmeshAgent;
                                                      CharacterCombat combat;

                                                protected virtual void Start() {

                                    navmeshAgent = GetComponent<NavMeshAgent> ();

                                       combat = GetComponent<CharacterCombat> ();

                                                   combat.OnAttack += OnAttack;
        combat.OnBattle += OnBattle;
        combat.OnNormal += OnNormal;
        combat.OnDeath += OnDeath;
    }

                                             protected virtual void Update () {

                                            animator.SetFloat ("Speed Percent",
navmeshAgent.velocity.magnitude/navmeshAgent.speed,.1f,Time.deltaTime)
;
                                                                                }

                                            protected virtual void OnAttack() {
        animator.SetBool("Attack", true);
        animator.SetBool("inBattle", false);
    }

    protected virtual void OnBattle()
    {
        animator.SetBool("inBattle", true);
    }

    protected virtual void OnNormal()
    {
        animator.SetBool("Attack", false);
        animator.SetBool("inBattle", false);
    }

    protected virtual void OnDeath()
    {
        animator.SetBool("isDead", true);
    }
}
```

```
[RequireComponent(typeof(NavMeshAgent))]
[RequireComponent(typeof(PlayerController))]
public class PlayerMotor : MonoBehaviour {

Transform target;
public NavMeshAgent agent;

void Start ()
{
agent = GetComponent<NavMeshAgent>();
GetComponent<PlayerController>().onFocusChangedCallback +=
OnFocusChanged;
                                                                    }

    public void WASDMove(bool usingJoystick)
    {

        if (!usingJoystick)
        {
                 Vector3 goal = transform.position
                 + Camera.main.transform.right *
Input.GetAxis("Horizontal")
                 + Camera.main.transform.forward *
Input.GetAxis("Vertical");

            agent.SetDestination(goal);
        }
        else
        {
                Vector3 goal = transform.position
                + Camera.main.transform.right *
Input.GetAxis("Horizontal Joystick")
                + Camera.main.transform.forward *
Input.GetAxis("Vertical Joystick");


            agent.SetDestination(goal);
        }
    }

public void MoveToPoint (Vector3 point)
                                                                    {
        // point click move
        agent.SetDestination(point);
                                                                    }

void OnFocusChanged (Interactable newFocus)
                                                                    {
```

```
if (newFocus != null)
{
agent.stoppingDistance = newFocus.radius*.8f;
agent.updateRotation = false;
target = newFocus.interactionTransform;
}
else
{
agent.stoppingDistance = 0f;
agent.updateRotation = true;
target = null;
}
}

void Update ()
{
if (target != null)
{
MoveToPoint (target.position);
FaceTarget ();
}
}

void FaceTarget()
{

 Vector3 direction = (target.position - transform.position).normalized;

                  Quaternion lookRotation = Quaternion.LookRotation(new
Vector3(direction.x, 0, direction.z));

              transform.rotation = Quaternion.Slerp(transform.rotation,
lookRotation, Time.deltaTime * 5f);
                                                                          }
}
```

```
[RequireComponent(typeof(CharacterStats))]
public class CharacterCombat : MonoBehaviour {

public float attackRate = 1f;
private float attackCountdown = 0f;

public event System.Action OnAttack;
    public event System.Action OnBattle;
    public event System.Action OnNormal;
    public event System.Action OnDeath;

    CharacterStats myStats;
CharacterStats enemyStats;

    public Slider healthSlider;

void Start ()
{
       myStats = GetComponent<CharacterStats>();


        healthSlider.maxValue = myStats.maxHealth.GetValue();
        healthSlider.value = myStats.currentHealth;
    }

void Update ()
{

        if (myStats.currentHealth <= 0)
            return;


        healthSlider.value = myStats.currentHealth;


        attackCountdown -= Time.deltaTime;
}

public void Attack (CharacterStats enemyStats)
{
if (attackCountdown <= 0f)
{
this.enemyStats = enemyStats;
attackCountdown = 1f / attackRate;

StartCoroutine(DoDamage(enemyStats,.6f));
if (OnAttack != null) {
OnAttack ();}
```

```
}
}

    public void Battle()
    {
        if (OnBattle != null)
            OnBattle();
    }

    public void Normal()
    {
        if (OnNormal != null)
            OnNormal();
    }

    public void Death()
    {
        if (OnDeath != null)
            OnDeath();
    }

    IEnumerator DoDamage(CharacterStats stats, float delay)
    {
Debug.Log("Start combat damage system");
yield return new WaitForSeconds (delay);

Debug.Log (transform.name + " attacking for " +
myStats.damage.GetValue () + " damage");
enemyStats.TakeDamage (myStats.damage.GetValue ());
}
}
```