ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ТЕХНОЛОГІЙ ЦИФРОВОГО РОЗВИТКУ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Ігровий застосунок для симуляції життя динозаврів із використанням Unity»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Технології цифрового розвитку»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Яна САНКІНА
(підпис)

Виконала: здобувачка вищої освіти групи ТЦР-41

Яна САНКІНА

Керівник: Вадим ДЗЮБА
старший викладач кафедри ТЦР

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Технологій цифрового розвитку

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Технології цифрового розвитку»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Технологій цифрового розвитку
_____ Вікторія ЖЕБКА
« ____ » _____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Санкіній Яні Сергіївні

1. Тема кваліфікаційної роботи: «Ігровий застосунок для симуляції життя динозаврів із використанням Unity»
керівник кваліфікаційної роботи старший викладач кафедри ТЦР Вадим ДЗЮБА, затверджені наказом Державного університету інформаційно-комунікаційних технологій від «_20_» лютого 2026 року № _51_.

2. Строк подання кваліфікаційної роботи «_18_» травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про палеонтологічні дослідження, документація Unity, бібліотеки для реалізації ШІ та фізики гри.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Аналіз предметної області та існуючих ігрових аналогів, порівняння їх актуальності, визначення вимог і проблем, постановка задачі.
2. Проектування архітектури, ігрових механік та структури даних гри на Unity, огляд вибраних технічних засобів.
3. Програмна реалізація ігрового застосунку, опис загальних стадій розробки, перелік реалізованих механік і додавання штучного інтелекту.

4. Тестування програми, аналіз ступені виконання роботи, оцінка ігрового балансу і результати дослідження.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.
2. Вимоги до гри.
3. Програмні засоби реалізації.
4. Подієво-орієнтована архітектура системи.
5. Діаграма прецедентів
6. Моделювання ігрового балансу
7. Оцінка результатів.
8. Екранні форми
9. Апробація результатів
10. Висновки

6. Дата видачі завдання «_20__» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз документації Unity	02.03-03.03.2026	
2	Аналіз предметної галузі	04.03-10.03.2026	
3	Дослідження існуючих аналогів та їх проблем	11.03-14.03.2026	
4	Розробка архітектури гри	15.03-19.03.2026	
5	Програмування основних механік	20.03-01.04.2026	
6	Реалізація системи поведінки динозаврів	02.04-11.04.2026	
7	Створення інтерфейсу	12.04-14.04.2026	
8	Проведення тестування і виправлення помилок	15.04-17.04.2026	
9	Підготовка до захисту	18.04-01.6.2026	

Здобувачка вищої освіти _____ Яна САНКІНА
(підпис)

Керівник
кваліфікаційної роботи _____ Вадим ДЗЮБА
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: __46_ стор., __2__ табл., _17__ рис., __25__ джерел.

Мета роботи – покращення процесу програмного відтворення життєдіяльності палеоекосистеми за допомогою розробленого ігрового застосунку на основі методів математичного моделювання і об'єктно-орієнтованого проєктування.

Об'єкт дослідження – процес програмного відтворення та функціонування палеоекосистем в інтерактивному віртуальному середовищі.

Предмет дослідження – ігровий застосунок, розроблений на базі рушія Unity з використанням даних на основі палеонтологічних і палеоекологічних досліджень.

Короткий зміст роботи: Проаналізовано предметну область та існуючі аналоги у галузі ігрових продуктів на тематику доісторичних істот і екосистем. Розроблено архітектуру застосунку і програмно реалізовані ключові функціональні можливості. У їх перелік входять: вибір і керування персонажем, система унікальних механік ведення бою, система штучного інтелекту і взаємодія з навколишнім середовищем. Додаток протестовано, висновки задокументовані. В роботі використано ігровий рушій Unity для створення і редагування віртуального середовища, а також мову програмування C# для реалізації ігрових скриптів.

Сферою використання застосунку є розважальна індустрія. Застосунок також може використовуватись у цілях популяризації палеонтології і виступати орієнтиром для розробників, які планують у майбутньому поєднувати покращення користувацького досвіду з науковим підходом.

КЛЮЧОВІ СЛОВА: UNITY, C#, ДИНОЗАВРИ, 3D, МЕХАНІКИ, ПАЛЕОНТОЛОГІЯ, ШТУЧНИЙ ІНТЕЛЕКТ, ВІДЕОГРА

ЗМІСТ

ВСТУП

Ігрова індустрія останнім часом є однією з найбільших високотехнологічних IT галузей. Успіх продукту залежить не лише від візуальної складової гри, але й від унікальності, зручності механік та якості програмної реалізації. Створення подібних 3D-середовищ вимагає від розробників вирішення складних завдань з оптимізації рендерингу, реалізацію і налаштування неповторюваних механік і розробки систем штучного інтелекту для неігрових персонажів.

Проведений аналіз ринку свідчить про те, що у ніші проєктів про доісторичний світ наразі існує концептуальний розрив. Існуючі рішення схиляються або до перевантажених симуляцій з високим порогом входження, або до спрощених аркад. Це зумовлює потребу у дослідженні і розробці ігрового застосунку, здатного забезпечити баланс між науковою достовірністю й привабливим для гравця досвідом.

Мета роботи – покращення процесу програмного відтворення життєдіяльності палеоекосистеми за допомогою розробленого ігрового застосунку на основі методів математичного моделювання і об’єктно-орієнтованого проєктування.

Для досягнення поставленої мети потрібно виконати перелік завдань:

1. Проаналізувати предметну область і існуючі аналоги на наявність технічних і концептуальних недоліків.
2. Спроєктувати архітектуру застосунку: склад ігрової системи, структуру даних і методи взаємодії об’єктів.
3. Розробити системи керування персонажем, бою та штучного інтелекту на базі скінчених автоматів, для більш автономної поведінки динозаврів.
4. Провести функціональне та ітеративне тестування, під час якого буде оцінено відповідність поставленим задачам, а також валідовано математичний баланс між симуляцією та ігровим процесом.

Об'єкт дослідження – процес програмного відтворення та функціонування палеоекосистем в інтерактивному віртуальному середовищі.

Предмет дослідження – ігровий застосунок, розроблений на базі рушія Unity з використанням даних на основі палеонтологічних і палеоекологічних досліджень.

Методи дослідження – системний аналіз, націлений на дослідження предметної області, палеонтологічних документів; порівняльний аналіз для виявлення недоліків існуючих рішень; методи проєктування програмного забезпечення і розробки ігор за допомогою Unity; методи математичного моделювання та ітеративного тестування для перевірки працездатності застосунку і налаштування точного балансу.

Наукова новизна роботи полягає у розробці подієво-орієнтованої архітектури та впровадженні системи ігрових механік, які базуються на науково достовірних фактах і адаптовані через ітеративне тестування для забезпечення стабільної продуктивності системи.

Практична значущість результатів полягає у створенні робочої версії ігрового застосунку, що програмно забезпечує баланс між розважальним контентом і науково достовірними фактами. Це демонструє можливість підвищення освітнього потенціалу інструментами ігрової індустрії серед прихильників комп'ютерних ігор.

Апробація результатів і публікації. Основні положення і результати бакалаврської роботи доповідались на науково-практичних конференціях, які проходили на базі Державного університету інформаційно-комунікаційних технологій, зокрема:

1. На VII Всеукраїнській науково-технічній конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях» (Київ, 2026 р.). Тези доповідей: «Розробка науково-орієнтованого ігрового симулятора палеоекосистеми».
2. На конференції «Технології цифрового розвитку: програмні системи та децентралізація» (Київ, 2026 р.). Тези доповіді: «Проєктування слабкопов'язаної архітектури ігрового застосунку на базі подієвої моделі».

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Характеристика предметної галузі

Сучасні комп’ютерні ігри – це результат еволюції простих розважальних програм у складні інтерактивні системи. Вони здатні виконувати комунікаційні, симуляційні, інколи професійно-тренувальні і навіть освітні функції. З точки зору інженерії програмного забезпечення, розробка подібних систем потребує комплексного підходу до проєктування складної архітектури, програмної оптимізації використання ресурсів апаратного забезпечення і створення алгоритмів взаємодії, до яких легко адаптуватись. Через високий рівень інтерактивності, ігрові програмні продукти дозволяють користувачу зануритися у віртуальне середовище і взаємодіяти з ним у реальному часі. Це робить сферу розваг особливо привабливою для наукових досліджень і комерційної розробки.

Окрему нішу в згаданій вище індустрії займають проєкти, присвячені доісторичному світу, а особливо динозаврам. Для багатьох користувачів ці застосунки є інструментом, що допомагають більше дізнатись про вимерлих істот, дослідити атмосферу доісторичного світу або випробувати власні навички у незвичних сценаріях виживання.

Аналіз ринку демонструє, що наявні продукти, пов’язані із цією сферою, умовно поділяються на три основні категорії:

– аркадні: спрощена фізика, фокус на геймплей з високою динамікою;

– симулятори: максимальна увага направлена на мікроменеджмент параметрів життєдіяльності і реалістичність середовища;

– навчальні додатки: виражений пізнавальний компонент, але часто обмежений і однотипний функціонал;

Однак, більшість з них стикається з технічними або функціональними недоліками. Це або високобюджетні комерційні проєкти, які вимагають значних обчислювальних можливостей через перевантажений рендеринг, або ж, навпаки,

занадто примітивні рішення, які не задовольняють вимог сучасного користувача стосовно якості продукту.

Особливість розробки у цій сфері полягає у необхідності програмної інтеграції кількох складних підсистем. До них належать: система генерації та оптимізації віртуального середовища, контролери анімацій, системи обробки колізій, а також архітектура штучного інтелекту для управління поведінкою неігрових персонажів.

Отже, розробка ігрового застосунку про динозаврів перетворюється на комплексне завдання, яке об’єднує елементи комп’ютерної графіки, програмування взаємозв’язків між об’єктами, моделювання фізичних процесів і проєктування інтерфейсу. Це робить завдання зі створення такої гри актуальним і міждисциплінарним. Воно вимагає від розробника застосування великої кількості сучасних інженерних практик.

1.2 Аналіз і характеристика існуючих ігрових продуктів

Тема доісторичних екосистем у іграх має тривалу історію розвитку. Щоб визначити концептуальні недоліки найпопулярніших аналогів та обґрунтувати власний варіант рішення, було проаналізовано актуальні на 2025-2026 роки ігрові продукти. Сам аналіз сфокусовано на тому, як ці проєкти вирішили проблему дотримання балансу між науковою достовірністю і геймплеєм, як реалізували штучний інтелект та які мають ігрові механіки.

Одним із найпоказовіших прикладів у галузі є симулятор Saurian. Гра робила акцент на 100% відповідності палеонтологічним даним, починаючи з відтворення точної анатомії і закінчуючи реконструкцією екосистеми формації Хелл-Крік. Користувач у цій грі виступає у ролі динозавра, якому потрібно пройти весь життєвий цикл. Проте досвід цього проєкту демонструє ключову проблему галузі: повне ігнорування потреб гравців заради максимально точного відтворення наукової достовірності шкодить залученості користувачів. Станом на 2026 рік

проєкт фактично заморожений, і це доводить неефективність повної симуляції без якісного геймдизайну [1].

Також популярним варіантом є багатокористувацький симулятор виживання The Isle на актуальній гілці розробки Evrima. Архітектура гри орієнтована виключно на соціальну бойову взаємодію між гравцями. Вектор розвитку цього проєкту свідчить про повний відхід від наукової достовірності у бік жанру «хорор». У майбутньому планують додавати мутантів, до того ж у грі практично відсутній штучний інтелект інших динозаврів, оскільки велика частина взаємодії направлена на самих гравців [2].

Комерційно відомим прикладом також є ARK: Survival Ascended. Це масштабна гра в жанрі «пісочниця», в якій динозаври виступають як ресурс. З точки зору логіки, віртуальні істоти в грі є спрощеними об’єктами, які виконують функції транспорту або зброї. Наукова достовірність повністю принесена в жертву рольовим елементам, а інтелектуальна взаємодія між різними видами динозаврів у дикій природі відсутня [3].

Прикладом також є Jurassic World Evolution 2, яка належить до жанру економічних стратегій. Увага у грі приділяється мікроменеджменту парку динозаврів. З технічної точки зору, поведінка тварин реалізована через складні анімації, і взаємодія з ними є жорстко заскриптованою. Екосистема тут закрита, а гравець виступає лише спостерігачем, який не має можливості брати участь в управлінні персонажем. Це зменшує рівень особистого занурення [4].

Проведений аналіз дозволяє зробити висновок і визначити проблему – на ринку існує значний розрив. Проєкти, які орієнтовані на науковий підхід, є технічно нежиттєздатними через відсутність динаміки всередині гри. Багатокористувацькі і масштабні ігри ігнорують особливості екосистеми і відповідність палеонтологічним даним заради бойових взаємовідносин між користувачами. Стратегії ж не надають досвіду прямої взаємодії. Це, відповідно, звільняє нішу на ринку для застосунку, який поєднує науково обґрунтовані характеристики динозаврів, складні алгоритми штучного інтелекту, пряме

керування персонажем та гнучкі механіки виживання. Саме розробка такого продукту і є завданням цієї кваліфікаційної роботи.

1.3 Опис проблем галузі та обґрунтування наукової достовірності

Під час аналізу було виявлено, що головна проблема предметної області – це неможливість існуючих проєктів запропонувати рішення, яке поєднує наукові факти та ігрові механіки без втрати необхідної гравцю динаміки. З інженерної точки зору, перенесення абстрактних біологічних даних у програмне середовище вимагає створення чітких моделей та алгоритмів.

У розроблюваному застосунку, який має робочу назву Forgotten Era, відповідність палеонтологічним даним програмно реалізується через три ключові підсистеми:

Балансування фізичних параметрів і бойових навичок. Характеристики динозаврів, такі як швидкість, сила укусу, затримка між атаками та розміри самих персонажів, не є довільними. Вони обчислюються і масштабуються, враховуючи палеонтологічні оцінки маси та анатомії тварин. Наприклад, більші види мають вищий показник сили, але обмежену маневреність. Менші види, навпаки, покладаються на швидкість пересування і атак. Крім згаданого, кожен з трьох доступних для гри видів динозаврів має унікальні механіки, що базуються на їх фізіологічних особливостях і мають за собою прототип у наукових дослідженнях.

Моделювання онтогенезу і адаптивної прогресії. Усі ігрові види динозаврів мають унікальні життєві цикли індивідуального розвитку – росту. Його швидкість корелює з їх біологічними особливостями, наприклад, різна тривалість життя. Перехід між стадіями росту супроводжується зміною розміру динозавра та індивідуальним збільшенням базових характеристик. Задля поєднання наукової достовірності із залученістю гравця, було розроблено механіку адаптивної прогресії: за кожну нову життєву стадію надається можливість обрати шлях розвитку персонажа в одній із трьох характеристик.

Таким чином, ця ігрова абстракція імітує фізіологічну адаптацію особини до середовища, посилюючи ті сильні сторони, які цікаві користувачу.

Штучний інтелект. Екосистема застосунку дотримується принципу історичної збіжності: перелік неігрових персонажів, з якими доведеться зіткнутися гравцю, історично існували в один проміжок часу [5]. Для управління поведінкою ворогів розроблено архітектуру штучного інтелекту (Finite State Machine, FSM). Якщо стандартні ігрові супротивники атакують до повного знищення, то представлений у роботі ШІ імітує реальну поведінку тварини. Автомат включає зміну декількох станів: «Патрулювання» – «Переслідування» – «Атака» – «Повільний оберт». Критичною для наукової достовірності можна назвати стан «Втеча». Якщо рівень здоров’я неігрового персонажа падає нижче 50%, програмно спрацьовує інстинкт самозбереження, і динозавр намагається покинути зону конфлікту [6].

1.4 Порівняльна характеристика аналогів

Проведений аналіз відомих ігрових рішень, які присвячені тематиці доісторичних екосистем, дозволяє виділити кілька основних напрямів реалізації. Частина проєктів акцентують увагу на виживанні гравця в умовах відкритого світу, інша – зосереджена на економічних механіках управління парком. Є ті, хто намагається наблизити поведінку віртуальних істот до моделі, близької до наукової.

При цьому кожен із розглянутих застосунків має як переваги, так і недоліки, що впливають або на освітній, або на розважальний потенціал гри. Аналіз цих параметрів дозволяє систематизувати сильні і слабкі сторони існуючих варіантів. Після цього стає можливим визначити нішу, у якій розроблюваний проєкт може запропонувати оптимізовані механіки, підвищений рівень реалізму та освітню цінність.

Нижче наведена порівняльна характеристика згаданих у попередніх розділах аналогів, яка відображає основні критерії їхньої оцінки. (Таблиця 1.1).

Таблиця 1.1

Порівняльна характеристика ігрових застосунків-аналогів.

Назва гри	**Реалізм**	**Цільова аудиторія**	**Переваги**	**Недоліки**
ARK: Survival Ascended	Низький	Прихильники жанру survival-sandbox.	Великий вибір видів динозаврів, мережевий режим, розвинена система будівництва.	Складний інтерфейс, висока системна вимогливість до апаратних ресурсів, відсутність наукової складової
Jurassic World Evolution	Низький	Прихильники економічних стратегій і відомої кінофраншизи	Висока графічна якість, зручні механіки з менеджменту бази.	Відсутність прямого контролю над динозаврами
Saurian	Високий	Користувачі, глибоко зацікавлені палеонтологією	Науково достовірне середовище, детальні моделі росту динозаврів	Заморожена розробка, відсутність динамічного геймплею
The Isle	Низький	Онлайн-спільнота, яка цікавиться багатокористувацькими іграми	Динамічна взаємодія між користувачами, великий вибір динозаврів	Відсутність освітнього елементу і ухил в жанр «хорор»
Проєктований застосунок (Forgotten Era)	Високий	Онлайн-спільноти прихильників динозаврів	Науково достовірне середовище, реалістична поведінка ШІ	Обмежена кількість ігрових видів на етапі прототипу

Проведений аналіз демонструє, що більшість ігор орієнтовані або виключно на розважальну складову, або на вузький сегмент геймерів, які шукають абсолютну симуляцію. Разом із тим, жоден із розглянутих аналогів не пропонує стабільної золотої середини: поєднання інтуїтивно зрозумілого керування від третьої особи із реалістичною алгоритмічною поведінкою неігрових персонажів [7]. Саме вирішення цієї проблеми і становить практичну новизну запропонованого проєкту.

1.5 Постановка завдання

Головним завданням кваліфікаційної роботи залишається проєктування і розробка тривимірного ігрового застосунку, реалізовані в середовищі Unity. В застосунку моделюється поведінка доісторичних істот на основі наукових даних.

Для досягнення поставленої мети потрібно реалізувати декілька етапів:

1. Спроєктувати архітектуру гри та визначити взаємозв'язки між основними програмними модулями. Розробити систему збереження прогресу користувача.
2. Реалізувати систему управління ігровим динозавром від третьої особи. Впровадити бойові механіки, додати кожному із видів унікальні здібності.
3. Розробити універсальну підсистему штучного інтелекту неігрових персонажів з використанням FSM. Вона повинна враховувати механіки патрулювання, переслідування, атаки, а також інстинкт самозбереження як для хижих динозаврів, так і для травоїдних.
4. Програмно реалізувати дорослішання персонажа та систему адаптивної прогресії характеристик відповідно до вибору гравця. Характеристики включають такі види як: сила укусу, швидкість, витривалість.
5. Створити графічний інтерфейс для візуалізації показників потреб, меню вибору персонажа і керування процесом прогресії.

1.6 Визначення вимог до програмного продукту

Функціональні вимоги:

1. Ініціалізація ігрової сесії. Система повинна забезпечувати вибір та завантаження однієї з трьох зумовлених конфігурацій ігрових сутностей.
2. Обробка локомоції та вводу. Програмний модуль керування має обробляти команди користувача для реалізації переміщення у тривимірному просторі та активації бойових методів обраного персонажа.

3. Система динамічного росту. Програма повинна реалізувати алгоритм зміни станів росту сутності через лінійну інтерполяцію масштабів і модифікацію числових атрибутів – характеристик.

4. Модуль модифікації атрибутів. Система повинна забезпечувати інтерфейс для внесення змін у структуру даних персонажа через розподіл накопичених балів досвіду.

5. Збереження даних. Програма повинна реалізувати механізм серіалізації та десеріалізації поточного стану ігрових об’єктів через локальне сховище для збереження і відновлення прогресу.

6. Керування автономними агентами ШІ. Модуль штучного інтелекту повинен базуватися на скінченних автоматах для детермінованого перемикання між станами на основі вихідних даних від системи сприйняття.

Нефункціональні вимоги:

1. Технологічний стек навігації. Обчислення шляхів переміщення автономних агентів має здійснюватися засобами технології Unity NavMesh.

2. Продуктивність та сумісність. Застосунок має стабільно функціонувати на ОС Windows 10/11 x64 та підтримувати стабільну частоту кадрів не нижче 30 на комп’ютерах середньої конфігурації.

3. Стабільність графічного інтерфейсу. Програмна реалізація інтерфейсу користувача має забезпечувати коректне масштабування і позиціювання елементів при зміні роздільної здатності екрана.

4. Архітектурна зв’язність. Внутрішня взаємодія між підсистемами має базуватись на принципах слабкої пов’язаності через подієво-орієнтовану модель.

2 ПРОЄКТУВАННЯ СИСТЕМИ І АНАЛІЗ ЗАСОБІВ РЕАЛІЗАЦІЇ

2.1 Вибір та обґрунтування обраних технологій

Для успішної реалізації поставленого роботою завдання та дотримання виконання функціональних вимог до тривимірного застосунку необхідно було обрати оптимальний набір технологій. Основними критеріями вибору можна назвати підтримку роботи з 3D-графікою, наявність вбудованих інструментів для розробки штучного інтелекту, гнучкість архітектури і можливість швидкого прототипування.

В якості основного середовища розробки було обрано ігровий рушій Unity версії 6.0 LTS. Вибір обґрунтовується декількома перевагами цієї платформи [8]:

– кросплатформеність і продуктивність. Unity забезпечує високий рівень оптимізації рендерингу для 3D-сцен, а це критично важливо для відображення доісторичного світу та моделей персонажів;

– вбудована система навігації NavMesh. Інструмент є необхідним для реалізації алгоритмів переміщення неігрових персонажів і побудови архітектури штучного інтелекту на базі FSM [9];

– система анімацій. Animator надає гнучкий інструментарій задля плавного переходу між стадіями руху та атаки, і це дозволяє досягти реалістичної поведінки динозаврів;

Основною мовою програмування для написання логіки гри було обрано C#. Це сучасна об’єктно-орієнтована мова програмування, яка для Unity є нативною. Її використання дозволяє реалізувати складні архітектурні патерни, як, наприклад, управління системою подій [10], та забезпечує сувору типізацію. Це значно знижує кількість помилок на етапі компіляції.

Для створення підсистеми збереження прогресу користувача було обрано формат представлення даних JSON (JavaScript Object Notation). На відміну від бінарних форматів, використання цього варіанту полегшує процес налагодження

і робить процес збереження та зберігання інформації досить легким. Мова C# надає доступ до вбудованих бібліотек, які використовуються для швидкої серіалізації і десеріалізації об’єктів, що підлягають збереженню [11].

У якості інтегрованого середовища розробки використовувався програмний продукт JetBrains Rider через його глибоку інтеграцію з Unity, наявність інструментів рефакторингу та зручної системи підказок.

Вибір цих технологій повністю покриває технічні потреби проєкту, дозволяє сфокусуватись на розробці ігрових механік і алгоритмів поведінки [12].

2.2 Проєктування архітектури та ігрових механік

Під час проєктування архітектури розроблюваного ігрового застосунку було поставлене завдання – забезпечення високої продуктивності, масштабованості та уникнення жорстких зв’язків між системами. Це сильно спрощує рефакторинг і налагодження проєкта у майбутньому [13].

Архітектура проєкту концептуально поділена на декілька ключових підсистем. Взаємодія між ними відбувається за принципом слабкої пов’язаності.

У класичній розробці ігор іноді застосовується метод взаємодії, при якому різні системи перевіряють стан одна одної кожний кадр. Це створює значне навантаження на процесор і не є необхідною умовою, тому у проєктованому застосунку від цього підходу відмовилися. Завдяки спеціальному диспетчеру подій модулі гри, такі як, наприклад, система здоров’я динозавра та користувацький інтерфейс, залишаються автономними. Коли відбувається деяка ігрова зміна, система генерує подію і усі інші модулі, які на неї підписані, самостійно на неї реагують.

Також для збільшення стабільної передачі даних між різними ігровими сценами спроєктовано менеджер, відповідний за стан ігрової сесії. Він відповідає за маршрутизацію логіки ініціалізації гри: зберігає ідентифікатор обраного виду динозавра, визначає, потрібно створювати нову гру чи відновлювати збережений файл.

Для керування неігровими персонажами спроєктовано універсальну систему на базі патерну FSM. Автомат інкапсулює логіку прийняття рішень для супротивників [14], а перехід між поведінковими патернами відбувається на основі оцінки дистанції між ШІ та гравцем, а також перевірки поточного рівня здоров'я. При спрацьовуванні заданих тригерів, автомат змінює активний стан.

Механіка розвитку персонажа розширена і концептуально розділена на два логічні вектори, що взаємодоповнюють один одного – зростання персонажа та накопичення бойового досвіду. Система росту відстежує загальний час і прогрес виживання, створюючи можливість для переходу між життєвими стадіями. Це гарантовано надає фіксовану кількість універсальних балів розвитку. Система бойового досвіду ж відстежує взаємодію гравця з неігровими персонажами: за перемогу над супротивником користувачу нараховуються умовні одиниці досвіду. Це дозволяє підвищувати ігровий рівень динозавра і отримувати додаткові універсальні бали розвитку понад вікові ліміти.

Усі ці модулі повністю інтегровані з диспетчером подій, тому при досягненні нової стадії росту або нового рівня генерується відповідна подія. Вона автоматично оновлює масштаб візуальної моделі та розблоковує користувацький інтерфейс для розподілу балів між трьома основними характеристиками [15].

Така архітектурна модель забезпечує високу гнучкість системи до подальших модифікацій. Відокремлення логіки обчислень від логіки відображення дозволяє імплементувати нові ігрові механіки, змінювати параметри видів динозаврів без ризику порушення цілісності кодової бази проєкту. Це також знижує навантаження на апаратні ресурси, оскільки розробка відбувається реактивно – у відповідь на ігрові тригери. Підхід робить застосунок масштабованим, а це критично важливо для підтримки стабільної продуктивності і збільшенні кількості автономних агентів у сцені. Візуальну схему взаємодії між описаними модулями системи через центральний диспетчер подій наведено на рисунку 2.1.

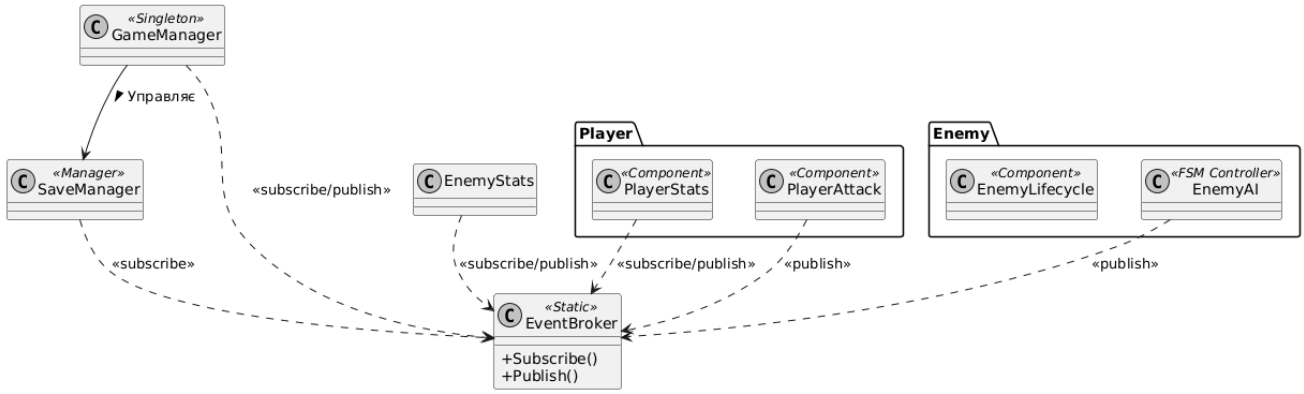


Рис. 2.1 UML-діаграма класів: архітектура підсистем ігрового застосунку

2.3 Проєктування структури даних

Ефективне функціонування систем збереження та прогресії вимагає чіткої структуризації даних, що підлягають серіалізації. Для забезпечення цілісності гри було спроєктовано єдину модель збереження. Вона виступає як універсальний контейнер, який відповідає за стан усіх автономних підсистем ігрового персонажа для подальшого запису у файл.

Для збереження ігрового прогресу було обрано формат текстового представлення об’єктів JSON [16]. Вибір можна обґрунтувати його ієрархічною структурою, котра добре підходить для опису складних сутностей. Також цей спосіб має просту інтеграцію з інструментами серіалізації Unity. Концептуально структура збереження поділена на декілька логічних блоків, які відображають різні аспекти ігрової сесії.

Метадані містять службову інформацію. В неї входять версія формату збереження для забезпечення зворотної сумісності у майбутньому, часова мітка створення файлу та ідентифікатор ігрового слота.

Просторові дані та ідентифікація відповідають за збереження технічної назви обраного динозавра, його точних координат і кута повороту для коректного відновлення позиції на локації.

Біологічні показники фіксують поточний фізичний стан персонажа. Зберігаються такі дані як, наприклад, точне значення росту, сили атаки, рівень здоров’я та показники базових потреб.

Прогресія та модифікатори акумулюють накопичений досвід, поточний рівень, кількість нерозподілених універсальних балів розвитку, і всі, вже застосовані, бонуси до базових характеристик.

Логіка роботи підсистеми збереження в цілому базується на принципі централізованого збору інформації. Спеціалізований під цю задачу менеджер під час ініціалізації процесу звертається до автономних компонентів динозавра, збирає їх актуальні дані в єдину структуру і записує їх у локальну пам’ять [17].

Для мінімізації ризику втрати прогресу ж створено механізм автоматичного збереження. Він функціонує на основі циклічного таймера, котрий у фоновому режимі ініціює перезапис даних у визначений слот. Запис спрацьовує через задані інтервали часу, але виключно за умови активного ігрового процесу [18].

2.4 Проєктування користувацьких сценаріїв

Внутрішню архітектуру і структуру даних ігрового застосунку було спроєктовано у попередніх підрозділах. Проте для повного розуміння функціонування системи необхідно розглянути її з нової перспективи – з точки зору кінцевого користувача. Для візуалізації того, як саме гравець взаємодіє із розробленими механіками, було створено UML-діаграму прецедентів, яка зображена на рисунку 2.2. Вона демонструє основні користувацькі сценарії, які забезпечує спроєктована архітектура, та визначає межі системи. Головним актором виступає гравець, і саме він ініціює більшість ігрових подій.

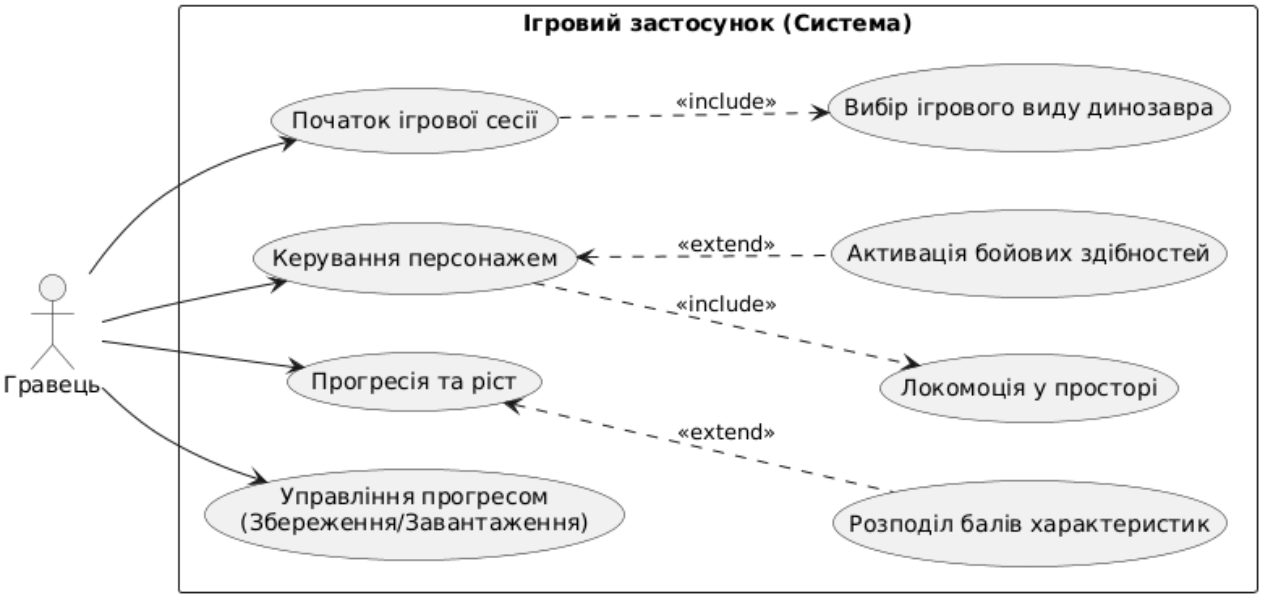


Рис. 2.2 UML-діаграма прецедентів взаємодії користувача з системою

До ключових прецедентів взаємодії відносяться:

– вибір динозавра та ініціалізація сесії. Гравець взаємодіє з головним меню застосунку і вибирає одного із доступних видів динозаврів. Сценарій визначає початкові характеристики персонажа, унікальні здібності і передає ці дані до менеджера ігрової сесії перед завантаженням основного рівня;

– керування персонажем. Користувач має можливість вільно переміщуватись у тривимірному просторі, контролює напрямок камери і взаємодіє з ігровою картою. Цей прецедент забезпечує модуль контролера від третьої особи [19];

– підтримка життєздатності. Такий сценарій передбачає контроль базових потреб динозавра зі сторони гравця, таких як голод і спрага, необхідних для його виживання в ігровому світі;

– бойова взаємодія. Сценарій охоплює використання базових атак і унікальних власних механік для боротьби з неігровими супротивниками. Взаємодія викликає події зниження рівня здоров’я у учасників;

– розвиток та розподіл характеристик. Успішні бойові зіткнення та проведений у грі час приносять умовні бали досвіду. При досягненні певних порогових значень користувач ініціює сценарій підвищення рівня або переходу

на наступну стадію росту, і за це отримує можливість власноруч розподілити універсальні бали характеристик;

– керування ігровим процесом. Гравець може в будь-який момент зберегти поточний прогрес або завантажити попередню збережену гру через відповідне меню;

Деталізація і формалізація наведених сценаріїв дозволила встановити чіткі межі між функціональними підсистемами застосунку і визначити необхідні інтерфейси взаємодії. Кожен із наведених прецедентів розглядається як завершена логічна транзакція, яка ініціюється гравцем і призводить до детермінованої зміни стану віртуального середовища. Використання UML-моделювання дозволило заздалегідь передбачити критичні шляхи передачі даних і оптимізувати порядок обробки через диспетчер подій.

Описана структура сценаріїв є фундаментом для подальшої програмної реалізації. Вона забезпечує повну простежуваність функціональних вимог від рівня концепту до конкретного програмного модуля. Системний підхід значно спрощує подальше тестування застосунку, дозволяє перетворити кожен прецедент у набір тестових сценаріїв.

Окрім ігрових функцій, розроблена модель враховує аспекти обробки виняткових ситуацій. Сценарій бойової взаємодії, наприклад, інкапсулює автоматичну перевірку поточних ресурсів персонажа, а сценарій розвитку перевіряє накопичення досвіду. Це запобігає виникненню логічних помилок у потоках даних.

3 РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Опис структури проєкту і ключові складові

Для забезпечення зручності навігації, підтримки і подальшого масштабування розроблюваного проєкту було створено строгу ієрархічну структуру зберігання файлів. Усі ресурси гри логічно розподілені за спеціалізованими каталогами всередині головної директорії Assets. Візуальне представлення файлової структури зображено на рисунку 3.1

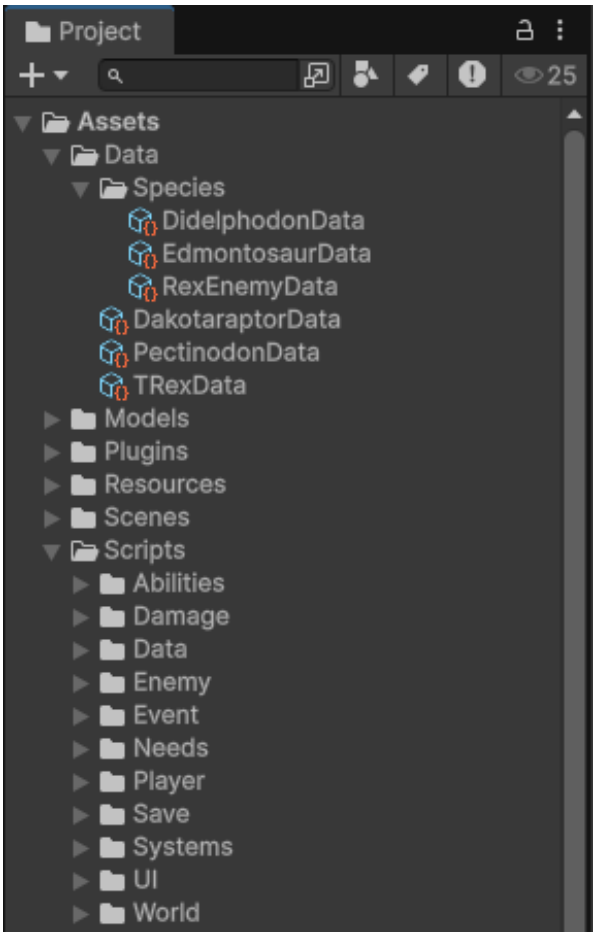


Рис. 3.1 Ієрархія файлів розроблюваного проєкту в Unity

Організація проєкту базується на декількох ключових підкаталогах: Scripts, Models, Scenes та ін. Далі розглянуто папку Scripts, як центральну папку

у проєкті, де зберігається вся основна логіка гри та її механіки. Вона містить всі вихідні коди, написані мовою C#. Щоб дотримуватись принципів чистої архітектури та єдиної відповідальності, скрипти розподілені за функціональними модулями:

1. Player – найбільш об’ємний модуль, який інкапсулює всю логіку головного персонажа. Там містяться контролери переміщення, анімацій, системи росту і прогресії, а також управління потребами динозавра.

2. Enemy зберігає логіку поведінки і штучний інтелект неігрових персонажів.

3. Needs зберігає скрипт оточення, який наділяє ресурси властивостями їжі та води, а Damage відповідає за обробку бойових зіткнень.

4. Abilities зберігає унікальні здібності кожного виду ігрових динозаврів.

5. Event – це каталог, який містить структури даних для всіх ігрових подій, циркулюючих в системі.

6. World, Save, UI є підсистемами взаємодії з навколишнім середовищем, збереження прогресу та користувацького інтерфейсу.

7. System є ядром ігрового застосунку, в якому зберігаються глобальні класи-менеджери. В них входять управління ігровою сесією, ініціалізація сцени та диспетчер подій.

Для того, щоб забезпечити модульність й уникнути неконтрольованих залежностей коду, глобальна архітектура спирається на кілька базових систем.

Система управління ігровим станом – статичний клас GameSession – відповідає за зберігання глобальних даних гравця під час завантаження нових сцен. До складу цих даних входять, наприклад, вид динозавра, обраного ще у головному меню. Це потрібно для того, щоб DinosaurInitializer міг зчитати його після завантаження відкритого світу. Логіку передачі даних і взаємодію між компонентами під час зміни ігрового контексту наведено на рисунку 3.2.

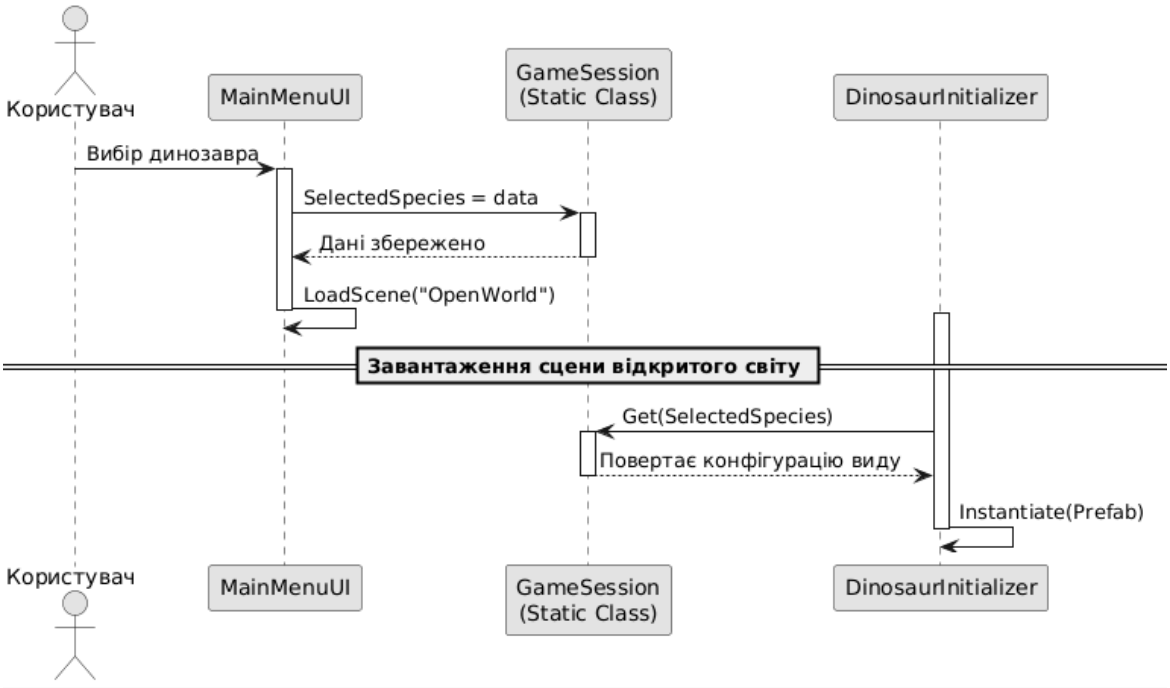


Рис. 3.2 UML-діаграма послідовності передачі даних

Диспетчер подій використовується для забезпечення слабкої пов’язанності між системами, використовуючи подієво-орієнтований патерн. Замість прямих посилань на об’єкти, що призводить до високої зв’язності коду, будь-який компонент може виступити ініціатором і опублікувати подію у загальну шину[20].

Програмна логіка побудована на базі словника типів. Використання хеш-таблиці гарантує швидкість пошуку необхідних підписників за алгоритмічний час. Це важливо для збереження стабільної частоти кадрів. В той момент, коли певна система ініціює підписку, диспетчер подій перевіряє наявність відповідного типу події у словнику і динамічно приєднує новий метод до ланцюга викликів. При публікації події диспетчер автономно викликає весь ланцюг делегатів, передаючи їм пакет даних. Архітектурну схему життєвого циклу підписки і публікації події наведено на рисунку 3.3.

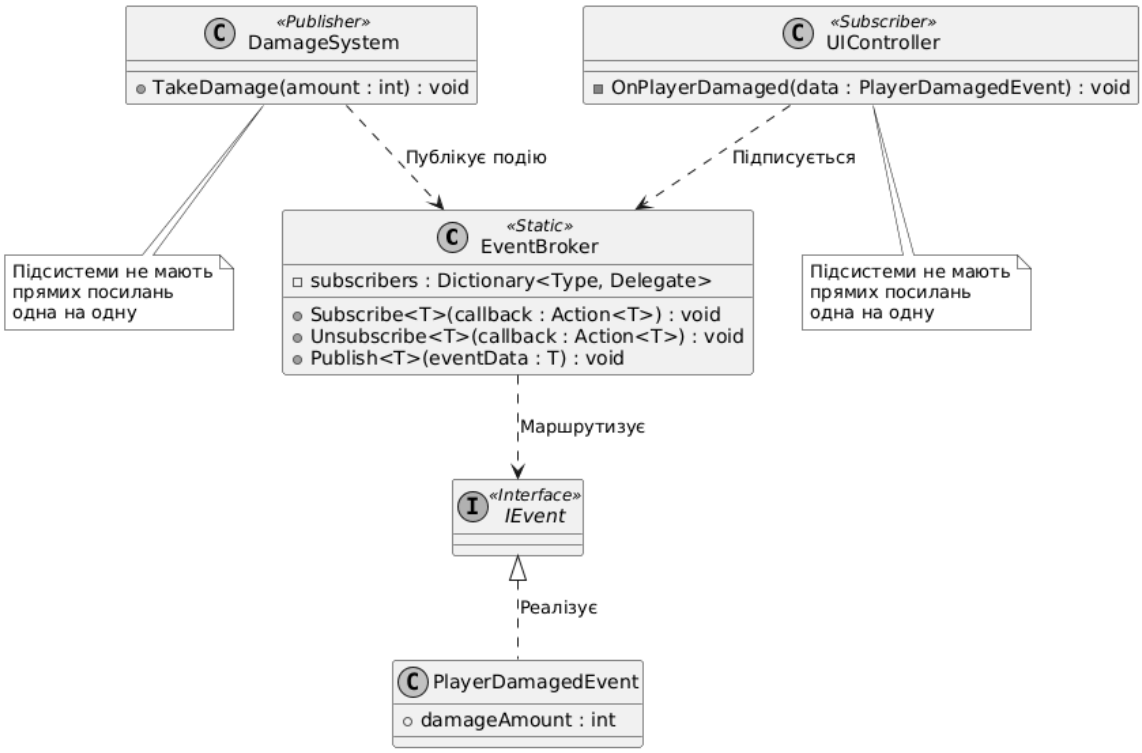


Рис. 3.3 UML-діаграма класів EventBroker

Окрім цього, у проєкті застосовано дата-орієнтований підхід. Усі балансні характеристики видів, такі як швидкість, здоров’я, сила атаки і потреби, винесені з програмного коду у конфігураційні ассети. Програмно це виконано шляхом створення класу-контейнера DinosaurSpeciesData, який успадковується від базового класу рушія – ScriptableObject.

Такий підхід дозволяє створювати численні варіації ігрових сутностей на основі єдиної структури без дублювання коду. Використання конфігураційних файлів також вирішує проблему налаштування ігрового балансу. Зміна параметрів динозавра відбувається через візуальний інспектор Unity, не вимагає втручання у вихідний код скриптів та їх подальшої перекомпіляції. Структуру класу даних та його зв’язок з основними ігровими системами наведено у вигляді діаграми класів на рисунку 3.4.

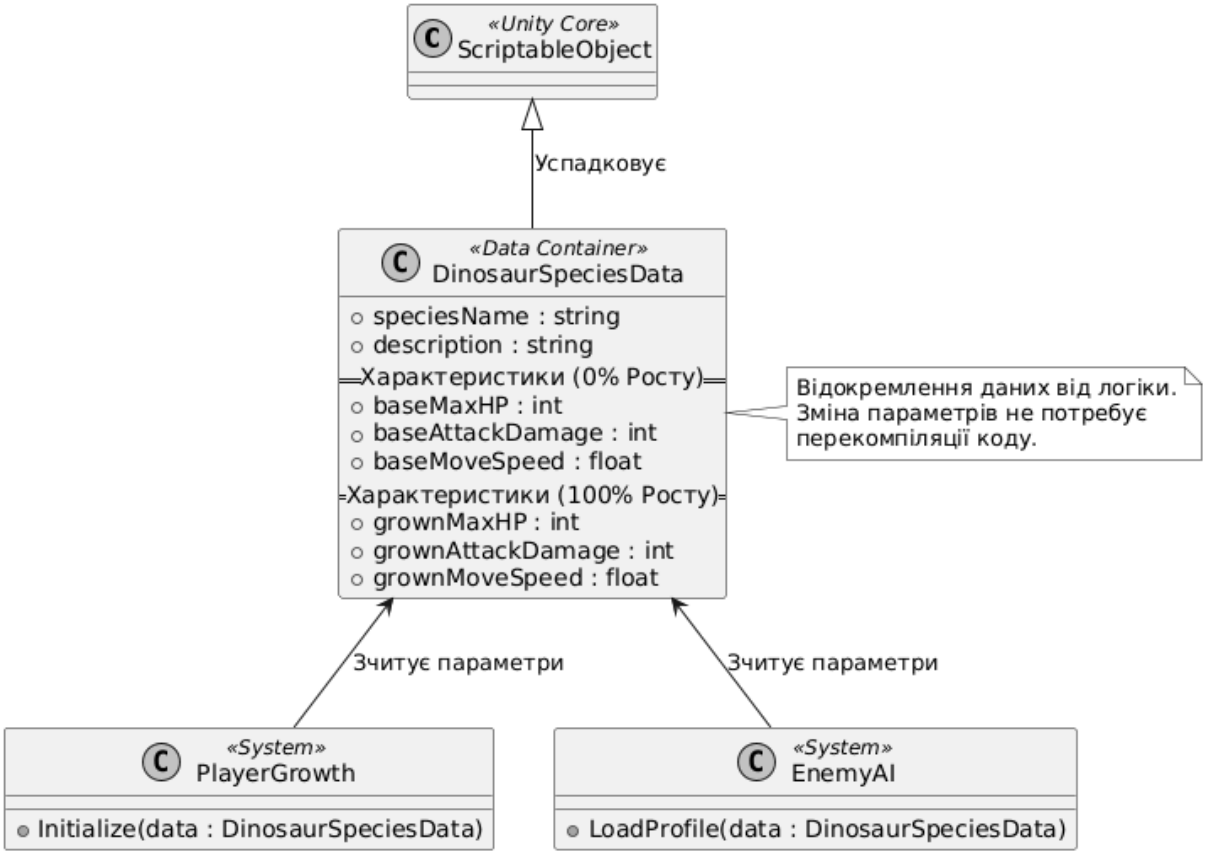


Рис. 3.4 UML-діаграма структури конфігураційних даних та їх залежностей

Така структура створює надійний та оптимізований фундамент для реалізованих у подальшому ігрових механік, таких як штучний інтелект і система виживання.

3.2 Реалізація штучного інтелекту для динозаврів

Для реалізації автономної поведінки неігрових персонажів було застосовано архітектурний патерн Finite State Machine (FSM). Подібний підхід дозволяє чітко розмежувати логіку поведінки об’єкта на дискретні стани та визначити суворі правила переходу між ними. Це допомагає запобігати виникненню логічних конфліктів під час ігрового процесу.

Ядром системи штучного інтелекту виступає компонент EnemyAI, що тісно взаємодіє з навігаційною сіткою Unity – NavMeshAgent. Логіка прийняття рішень у цьому випадку обробляється у головному циклі Update, в якому,

залежно від поточного стану, викликається потрібний метод обробки поведінки. Програмну реалізацію базового перемикача станів наведено на рисунку 3.5.

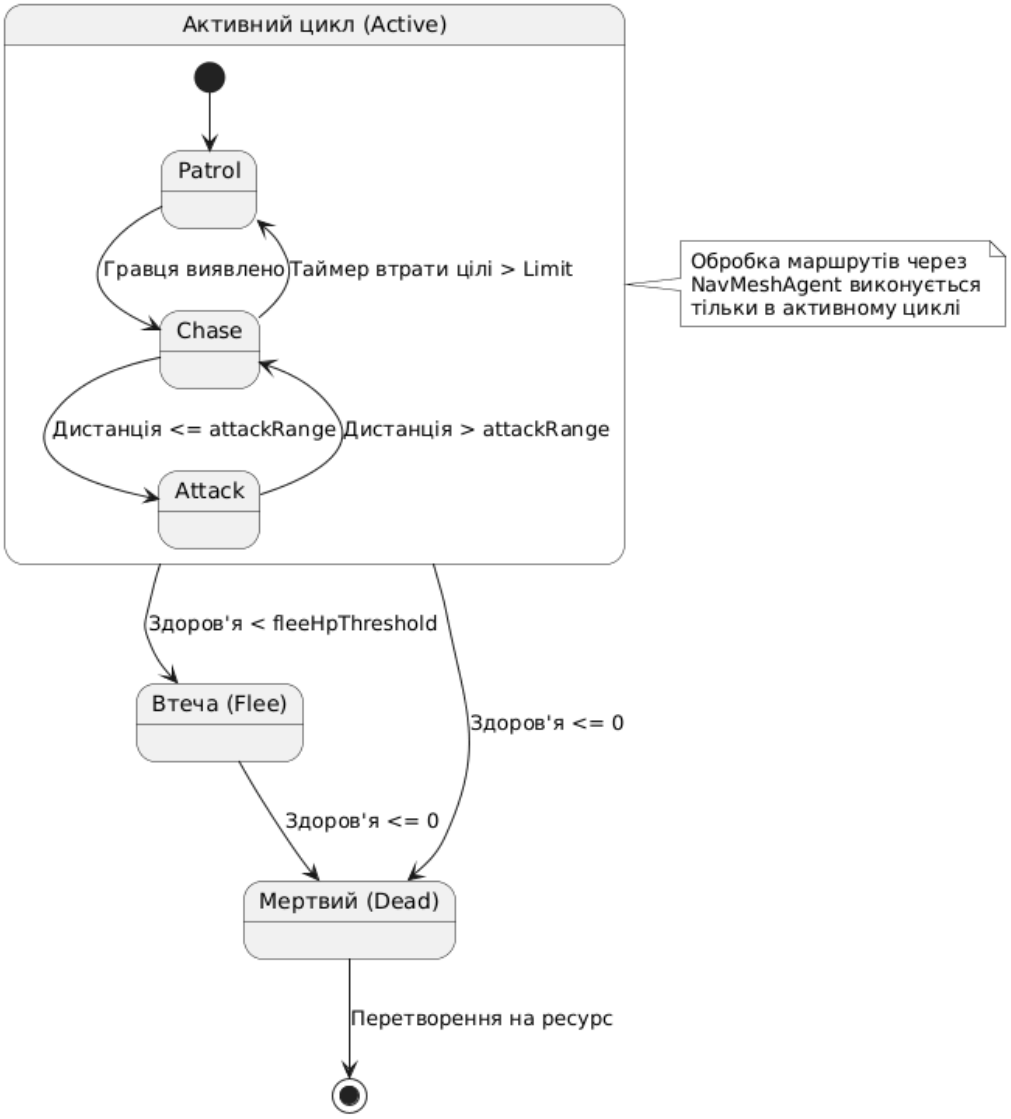


Рис. 3.5 UML-діаграма станів маршрутизації автономної поведішки ШІ

Кожен стан інкапсулює свою логіку. Розглядаючи, наприклад, стан переслідування – TickChase – можна помітити, що саме він створює найбільше навантаження на обчислювальні потужності процесора через необхідність постійного перерахунку маршруту на складній геометрії ігрового світу.

Для оптимізації алгоритм розділено на фази просторової валідації і оновлення маршруту. Робота починається із застосування швидкісних параметрів, взятих із конфігураційного профілю динозавра, і перевірки наявності

об’єкта цілі в пам’яті. Далі обчислюється двовимірна відстань до гравця, ігноруючи вісь Y для запобігання помилок на схилах. На підставі дистанції система приймає рішення: якщо ціль знаходиться у зоні ураження, супротивник миттєво переходить до бойового стану.

Якщо гравець занадто віддаляється і виходить за межі радіуса пошуку, запускається механізм акумуляції затримки. Доки ціль поза зоною видимості, таймер збільшується. Коли дистанція знов скорочується – лічильник скидається до нуля. Блок-схему першої фази алгоритму наведено на рисунку 3.6.

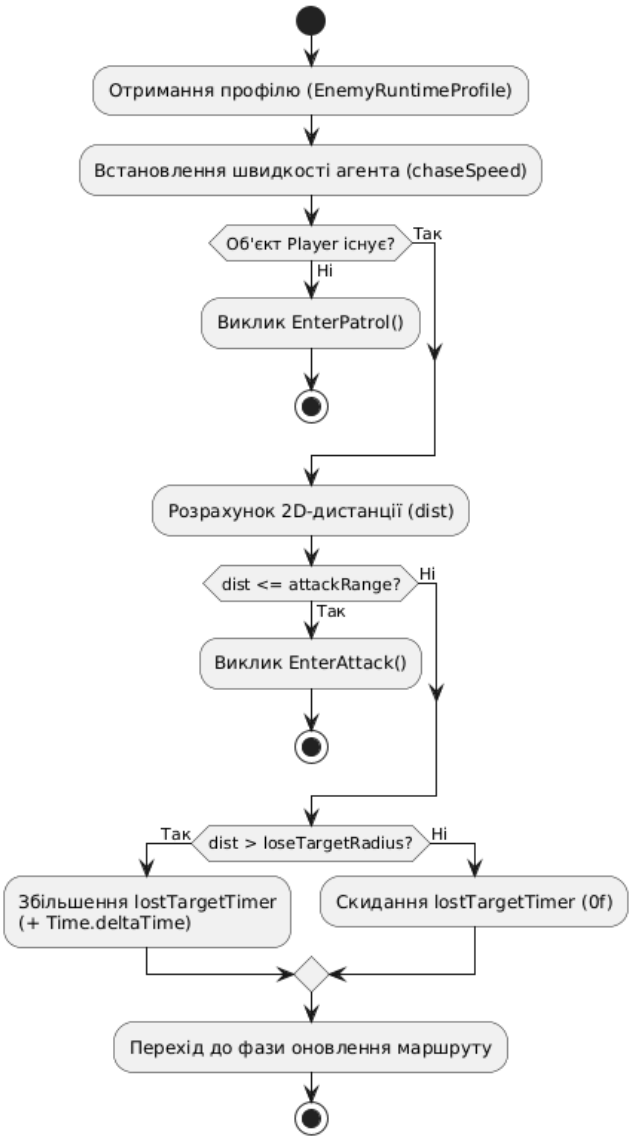


Рис. 3.6 UML-діаграма діяльності перевірки просторових умов та стану цілі під час переслідування

Якщо лічильник перевищує допустиму затримку, то ШІ припиняє переслідування і повертається на патрулювання.

Під час розробки було приділено увагу оптимізації. Стандартна реалізація оновлення цілі для компонента через NavMeshAgent у кожному кадрі створює надмірне навантаження. Це особливо помітно за умови складної геометрії рівня і великій кількості ворогів.

Для вирішення даної проблеми в проєкті було реалізовано механізм планування шляху за допомогою таймера. Розрахунок нової траєкторії та звернення до методів відбувається не кожної миті, а тільки після завершення визначеного часового інтервалу. Такий підхід дозволяє знизити кількість викликів ресурсомістких методів SetDestination, зберігаючи візуальну плавність руху неігрового персонажу. Це запобігає появі мікрозатримок під час ігрового процесу. Програмну реалізацію даного механізму винесено у Додаток Б.

Це рішення дозволяє системі штучного інтелекту залишатись продуктивною навіть у стресових ситуаціях, коли на сцені одночасно присутня велика кількість автономних персонажів.

Бойова система ШІ побудована на принципах абстракції і обробки колізій. Основним методом реалізації бойової системи є функція TryDealDamage для ворогів і TryApplyHit для гравця.. Вони відповідають за виялення цілей у зоні ураження і передачу розрахованого значення пошкоджень. Система використовує вбудований у рушій механізм фізичних тригерів – методи OnTriggerEnter і OnTriggerStay. Вони автоматично викликаються при перетині колайдерів.

Алгоритм обробки удару включає кілька етапів. По-перше, система перевіряє, чи не була ця ціль вже вражена під час поточної атаки. По-друге, для кожного знайденого об’єкта проводиться перевірка на наявність інтерфейсу Damageable або компоненту PlayerHealth. Це забезпечує гнучкість системи. Будь-який новий об’єкт, динозавр або елемент оточення, може бути

зруйнованим, якщо має відповідний компонент, але при цьому код базової системи не потребує змін.

Після успішної валідації цілі відбувається розрахунок фінальних пошкоджень і публікація через диспетчер подій. Це дозволяє іншим системам реагувати на факт удару незалежно від логіки самої жертви. Повну блок-схему алгоритму бойової взаємодії наведено на рисунку 3.7.

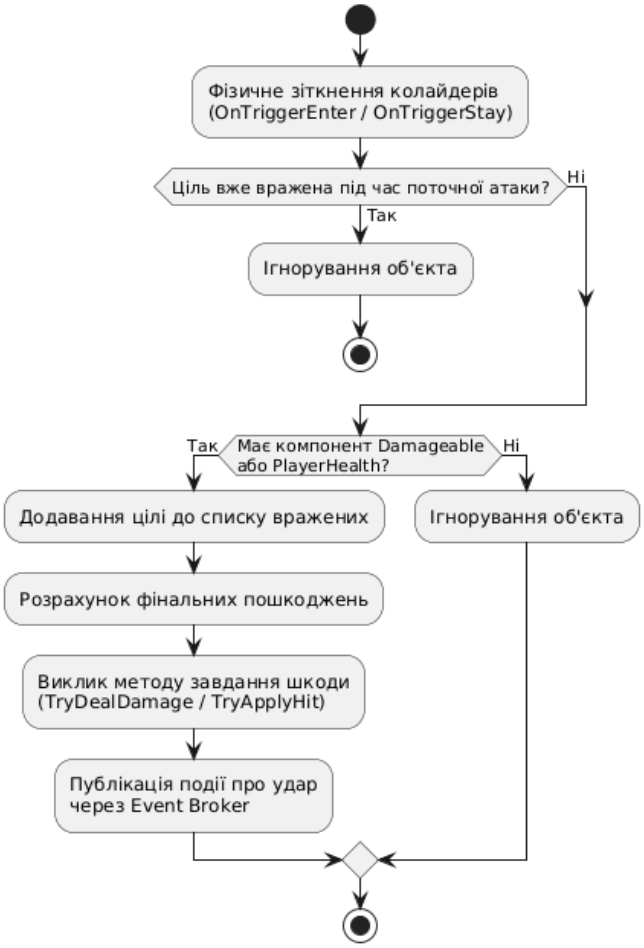


Рис. 3.7 UML-діаграма діяльності пошуку компонента здоров'я і розрахунку шкоди

Відслідковування вхідних пошкоджень абстраговано у базовий компонент Damageable. Щоб уникнути постійного опитування стану здоров’я, інтеграція штучного інтелекту з бойовою системою повністю переведена на реактивну

подієву модель. Компонент автономної поведінки виступає підписником на делегати компонента здоров’я.

У момент отримання пошкоджень викликається асинхроний обробник, який виконує валідацію нового стану: якщо відсоток залишкового здоров’я перетинає критичну межу, заздалегідь визначену у конфігураційному профілі, скінчений автомат примусово перериває поточний бойовий цикл. Ворог переходить у захисний стан – втечу. Це забезпечує достовірну імітацію інстинкту самозбереження.

Окрім поведінкових реакцій, життєвий цикл неігрового персонажа інтегрований в економічну модель симулятора виживання. Коли показник здоров’я сягає нульової позначки, система не викликає метод видалення об’єкта, запобігаючи стрибкам навантаження. Замість знищення, спеціалізований контролер EnemyLifecycle ініціює метод ConvertToCorpse. Алгоритм безпечно відключає всі бойові скрипти, зупиняє розрахунки навігаційної сітки і деактивує колізії, фіксуючи об’єкт. Після цього алгоритм динамічно ініціалізує на об’єкті компонент Consumable, перетворюючи його на джерело їжі. Повний код описаних алгоритмів бойової взаємодії і трансформації стану винесено у Додаток Б.

3.3 Інтерфейс користувача, опис ігрових сцен і основних механік

Архітектура користувача побудована на принципі динамічної конфігурації. Замість створення окремих статичних префабів, проєкт використовує єдиний базовий об’єкт-контейнер, котрий наповнюється логікою і даними в момент вибору динозавра і завантаження сцени.

Процес підготовки гравця розпочинається із головного меню, де обраний вид динозавра записується у GameSession. Коли основна ігрова локація завантажується, DinosaurInitializer знаходить відповідний файл і транслює задані там параметри у функціональні скрипти. Підхід дозволяє миттєво змінювати

модель персонажа без необхідності перезавантаження сцени або дублювання логіки в інспекторі.

Основою взаємодії гравця із ігровим світом є система переміщення. У проєкті вона базується на CharacterController. Замість використання жорсткої фізики Rigidbody, контролер забезпечує плавне ковзання по рельєфу і запобігає застряганню у різних місцях карти. Повну логіку обробки переміщення та фізичних перевірок наведено на рисунку 3.8.

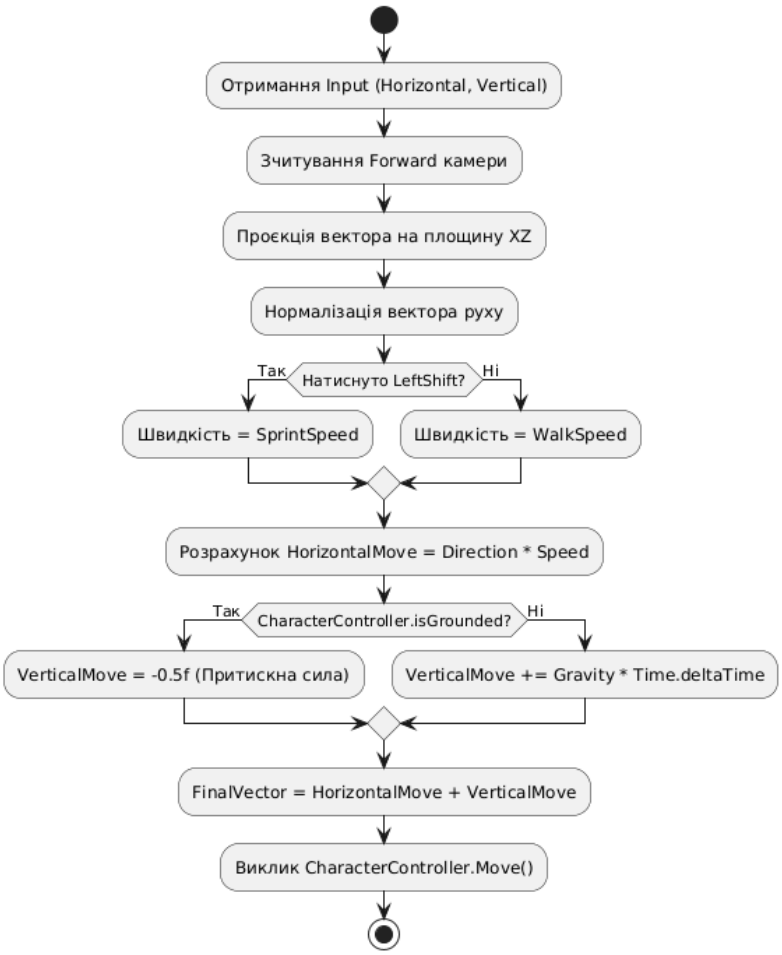


Рис. 3.8 UML-діаграма діяльності алгоритму обробки переміщення та гравітації.

Розрахунок вектору руху відбувається відносно напрямку камери. Скрипт отримує вектори forward і right об’єкта камери, нівелює їх по осі Y і нормалізує.

Завдяки цьому натискання клавіші «вперед» завжди направляє динозавра у напрямку, куда дивиться гравець.

Другий етап алгоритму відповідає за вертикальне переміщення. Для повноцінного бігу по складному рельєфу необхідно враховувати фізику падіння. Якщо динозавр знаходиться на землі, до нього застосовується невелика притискна сила, що активна завжди – вона запобігає підстрибуванню моделі під час спуску зі схилів. У повітрі на об’єкт діє повноцінна кастомна гравітація, тому лише після об’єднання горизонтального і вертикального векторів, об’єднуючий вектор передається у метод Move.

Швидкість переміщення залежить від двох факторів – глобальних характеристик динозавра і поточного стану витривалості. За контроль бігу відповідає скрипт PlayerStamina.

Для того, щоб уникнути миттєвого відновлення витривалості і змусити гравця краще планувати витрачання своїх ресурсів, у механіку впроваджено систему затримки. Для цього використовується асинхронний цикл RegenDelayCoroutine. Для запобіганню постійних перевірок, корутина запускається лише у момент припинення витрат енергії. В цей час алгоритм зупиняє своє виконання на заданий час – RegenDelay [21], після чого починає поступово відновлювати витривалість до максимального значення. Будь-яка повторна спроба побігти до завершення процесу – автоматично перериває цикл, скидаючи таймер регенерації.

Також у застосунку реалізована механіка росту. Розроблена архітектура відмовляється від статичних показників гравця, вони не є сталими величинами, а безперервно еволюціонують у реальному часі. За це відповідає PlayerGrowth.

Математична модель росту базується на обчисленні нормалізованого коєфіцієнта – відношення поточного віку до максимально можливого. Використовуючи його, скрипт звертається до конфігураційного файлу динозавра і застосовує функцію лінійної інтерполяції. Це дозволяє системі плавно вираховувати проміжні значення між характеристиками маленького і дорослого динозавра для будь-якого параметра.

Окрім візуального масштабування, компонент також виконує архітектурну функцію головного агрегатора характеристик. Прогресія посилюється системою рівнів – ExpirienceSystem, – яка функціонує на базі подієвої моделі. При отриманні досвіду за бойові досягнення алгоритм використовує ітераційну валідацію: циклічна перевірка залишку досвіду дозволяє коректно обробляти стрибки через кілька рівнів за один кадр. Отримані бали розвитку вкладаються гравцем через StatUpgradeSystem, де вони підсумовуються з базовими характеристиками поточного росту. Кінцевий результат циклу обчислень публікується як глобальна подія, що дозволяє інтерфейсу та фізичним колайдерам адаптуватися до нових габаритів персонажа. Детальні алгоритми розрахунку досвіду та інтерполяції характеристик наведені у Додатку Б.

3.4 Програмна реалізація системи збереження та серіалізації даних

Для забезпечення тривалого ігрового процесу та збереження прогресу гравця було розроблено багаторівневу систему зберігання даних. Архітектура підсистеми побудована на принципах поділу відповідальності і розділена на три логічні рівні: модель даних, менеджер серіалізації та контролер автоматичного збереження.

Основою системи є клас-контейнер SaveData, який виконує роль об’єкта передачі даних. Він має можливість перетворюватись на текстовий або бінарний файл. Для забезпечення максимальної стабільності та сумісності при серіалізації клас не містить посилань на складні ігрові об’єкти. Він інкапсулює примітивні типи даних, такі як координати, індекси стадій росту, показники потреб і метадані.

Головним вузлом обробки виступає статичний клас SaveSystem. Процес збереження починається зі зняття знімку стану поточного ігрового процесу. У момент ініціації збереження менеджер виконує опитування ключових компонентів гравця. Дані з цих модулів передаються у єдиний екземпляр

SaveData. Це гарантує, що підсистеми гравця залишаються незалежними і не потребують власної логіки запису в файл.

Сформований об’єкт даних передається до рівня серіалізації. Там відбувається його конвертація у формат JSON за допомогою вбудованого класу JsonUtility. Такий вибір обумовлений його невеликою вагою, швидкістю обробки і можливістю зручного відлагодження.

Для запису файлу на диск застосовується бібліотека System.IO, і критично важливим рішенням є використання властивості Application.persistentDataPath для динамічної генерації шляху до файлу збереження. На відміну від статичних шляхів, ця властивість автоматично визначає безпечну і дозволену директорію для запису даних залежно від операційної системи. Логіку взаємодії компонентів під час циклу збереження наведено на рисунку 3.9.

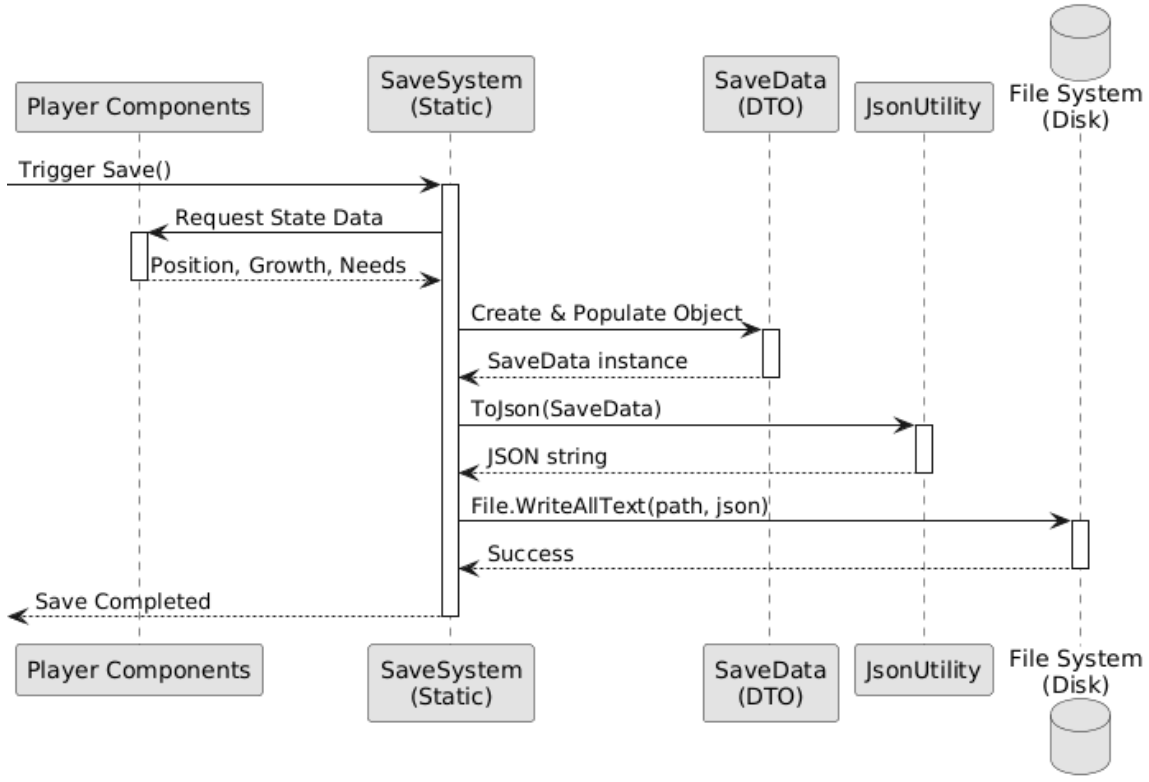


Рис. 3.9 UML-діаграма послідовності процесів агрегації та серіалізації ігрових даних

Для запобігання ризиків неочікуваної втрати прогресу систему доповнено AutoSaveManager. Він функціонує на базі асинхронних корутин, які з певною періодичністю ініціюють фонове збереження активного слота. Також реалізовано обробник системної події, який гарантовано записує фінальний стан гри безпосередньо перед закриттям ігрового вікна операційною системою. Повний код структур даних та методів запису винесено до Додатка Б.

3.5 Програмна реалізація графічного інтерфейсу користувача

Диспетчер подій також використовує і система розробленого графічного інтерфейсу. Це збільшує продуктивність і відсутність жорстких зв’язків між візуальною частиною гри і внутрішньою логікою.

Головний екран під час гри містить індикатори важливих для життя показників: шкалу здоров’я, росту, голоду, спраги і витривалості. Замість ресурсномісткого опитування характеристик гравця, контролер інтерфейсу, як і кожна згадана вище система, використовує підписку на системні події.

При настанні відповідної події, наприклад, втраті витривалості, інтерфейс отримує пакет даних із поточним і максимальним значенням параметра. Система нормалізує ці дані у діапазон від 0 до 1 та застосовує їх до візуальних елементів. Це гарантує, що перемальовування графіки відбувається виключно в моменти фактичної зміни стану, що суттєво економить ресурси процесора. Програмний лістинг логіки оновлення наведено в Додатку Б.

Точкою входу в саму програму є головне меню, яке веде на екран вибору персонажа, що керується компонентом SpeciesSelectionUI. Інтерфейс використовує динамічне зв’язування даних і звертається до візуальних карток динозаврів, пов’язаних з відповідними конфігураційними файлами. При натисканні на кнопку вибору, скрипт передає обраний об’єкт даних у глобальний клас GameSession та ініціює після цього завантаження основної сцени. Таким

чином об’єкт гравця формується динамічно і залежить від вибору в меню. Головне меню зображено на рисунку 3.26.

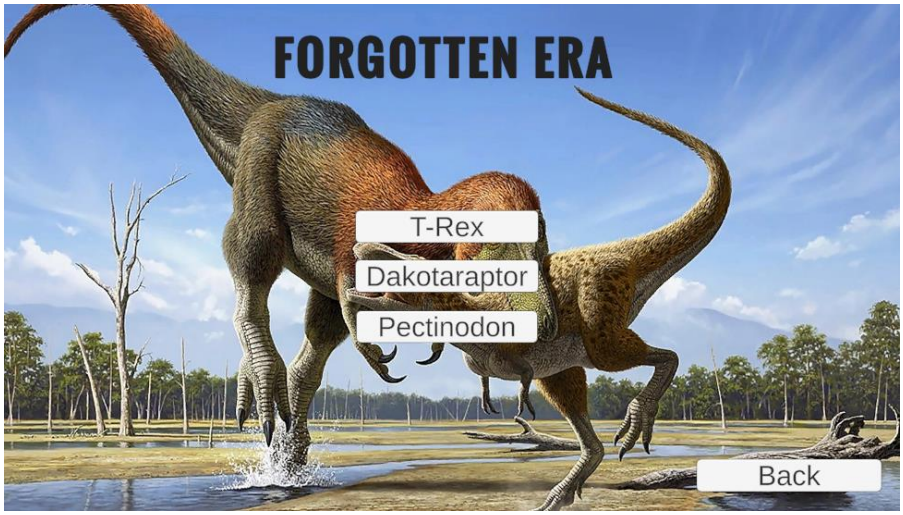


Рис. 3.10 Графічний інтерфейс вибору ігрового персонажа

У переліку підсистем графічного дизайну також є меню покращення характеристик. Цей компонент відображає поточні показники і разом з тим валідує користувацьке введення. Скрипт підписаний на подію StatPointsChangedEvent, а у методі оновлення інтерфейсу система перевіряє логічну умову наявності балів розвитку. Якщо гравець їх не має, кнопки для покращення характеристик автоматично блокуються. У разі ж успішної перевірки, після того, як користувач обирає характеристику для покращення, викликається відповідний метод системи. Лістинг валідації наведено в Додатку Б.

Крім цього, важливим елементом можна назвати можливість зупинки ігрової сесії. Для цього реалізовано меню паузи – система, яка при натисканні відповідної клавіші перехоплюплює подію та заморожує внутрішньоігровий час. Це реалізовано завдяки встановленню глобального параметра рушія – Time.timeScale зі значенням 0.

Разом із тим відбувається розблокування курсора миші. Це дозволяє користувачу взаємодіяти з елементами навігації, такими як кнопки продовження

гри, збереження або повернення до головного меню. Під час паузи також блокуються всі обчислення: ШІ припиняють пошук шляху, потреби гравця і його розвиток не змінюються. Після деактивації панелі перебіг часу відновлюється, а управління персонажем знову переходить в активний стан. Загальний вигляд меню паузи наведено на рисунку 3.28.

Рис. 3.11 Меню паузи під час ігрового процесу

У розділі було розглянуто повну реалізацію архітектури ігрового застосунку. В результаті, розроблено багаторівневу систему штучного інтелекту, яка відповідає за перемикання кількох станів існування супротивника, а також обробляє бойові зіткнення. Комбінування цих станів дозволяє продемонструвати найбільш реалістичну поведінку доісторичних істот.

Окрім цього, було спроєктовано систему, яка відповідає за взаємодію гравця зі світом. Реалізовано механіки динамічного масштабування росту, виживання та покращення через накопичення внутрішньоігрового досвіду.

Впроваджено надійну систему збереження ігрового прогресу, створено інтуїтивно зрозумілий інтерфейс. Усі підсистеми успішно інтегровані між собою через диспетчер подій, що забезпечує високу модульність, гнучкість коду і стабільність роботи програми.

4 ТЕСТУВАННЯ ТА ОЦІНКА РЕЗУЛЬТАТІВ

4.1 Обґрунтування достовірності симуляції

Головною відмінністю розробленого ігрового застосунку від аналогічних проєктів є максимальне наближення до палеонтологічної і історичної достовірності. Замість випадкового поєднання відомих і знайомих гравцям видів з різних епох, архітектура ігрової екосистеми повністю відповідає екосистемі часів формації Хелл-Крік близько 66 мільйонів років тому. Ціллю проєкту було поєднання наукового підходу до розробки ігор, не перетворюючи гру на повну симуляцію.

Усі представлені у грі істоти – як динозаври, доступні для вибору гравцем, так і їх неігрові супротивники – історично співіснували в одному часовому проміжку і географічному регіоні. Таким чином вони формували складну конкурентну мережу.

1. Вищі хижаки і макрофауна. Тиранозавр займає вищу точку харчового ланцюга, полюючи на велику здобич. Прикладом його базового супротивника є Едмонтозавр і інші тиранозаври.

2. Мезохижаки. Дакотараптор був швидким хижаком середнього розміру, котрий займав нішу полювання на відкритій місцевості. Він не міг конкурувати з тиранозаврами навіть полюючи у зграї, але мав можливість атакувати молодих тиранозаврів и діделькфодонів.

3. Мікрофауна і базові ресурси. Дрібний троодонтид – Пектинодон, та сумчатий ссавець Діделькфодон формують найнижчий рівень харчового ланцюга.

Особливістю гри є відображення зміни екологічних ролей істот під час їхнього розвитку. Згідно з палеонтологічними дослідженнями, молоді тиранозаври мали пропорції тіла, кардинально відмінні від дорослих особин. Вони не могли полювати на велику здобич. Замість цього вони займали нішу

мезохижаків, конкуруючи за ресурси з дорослими дромеозавридами, до яких відноситься Дакотараптор.

У грі цю особливість реалізовано через механіку росту і генерацію ШІ-тиранозаврів різного віку. Тепер ситуація, коли гравець, що грає за Дакотараптора, змушений рятуватися від молодого ворожого тиранозавра, є не лише елементом ігрового балансу, а і науково підтвердженим фактом міжвидової конкуренції. Користувач відчуває еволюційну боротьбу за виживання, а його місце в екосистемі динамічно змінюється із часом.

4.1.2 Математична трансляція палеоданих в ігровий баланс

Було відкинуто метод емпіричного підбору параметрів, як метод перенесення реальних палеонтологічних даних у простір рушія Unity. Натомість розроблено універсальну систему математичного масштабування. Вона застосовується до всіх об’єктів в екосистемі – і до ШІ і до ворогів. Баланс ігри тепер гарантовано є системним і саморегульованим.

Для розрахунку ігрової швидкості переміщення персонажів було застосовано універсальний кінематичний коєфіцієнт (K_{kin}). Формула конвертації (4.1) має вигляд:

$$V_{unity} = V_{real} * K_{kin}, \quad (4.1)$$

де V_{unity} – це ігрова швидкість об’єкта; V_{real} – реальна палеонтологічна швидкість динозавра; K_{kin} – кінематичний коефіцієнт простору;

Значення коефіцієнта 0.14 не є випадковим. Воно є добутком фізичної константи переведення км/год у м/с (0.277) та ігрового множника простору (0.5). Використання редуктора є стандартним методом у геймдизайні для ігор від третьої особи. При застосуванні цієї моделі, швидкості ігрових і неігрових персонажів коректно масштабовано.

Для визначення запасу здоров’я застосовано закон квадрата-куба, що описує співвідношення маси тіла і міцності біологічних тканин. Формула нелінійного масштабування (4.2) виглядає:

$$HP = \sqrt{Mass_{kg}} * 2, \quad (4.2)$$

де HP – запас здоров’я ігрового об’єкта; $\mathrm{Mass_{kg}}$ – реальна маса динозавра (кг); 2 – базова константа виживання;

Множник 2 виступає як аналітична константа показника Time-To-Kill. Виходячи з розрахункової сили атаки, саме цей множник забезпечує цільову тривалість бойового зіткнення великих особин у межах 10-15 секунд. Це відповідає стандартам проєктування систем виживання, унеможливлюючи миттєву загибель персонажів та надаючи гравцеві час на тактичний відступ.

Швидкість виснаження потреб динозавра не потребує штучних множників, оскільки і так базується на фундаментальному біологічному законі Клайбера [22]. Формула розрахунку (4.3) метаболізму:

$$M_{rate} = Mass^{-0.25}, \quad (4.3)$$

де M_{rate} – рівень метаболізму динозавра; Mass – маса тіла динозавра;

Ця модель створює природну асиметрію. Дрібний Пектинодон має в 4.5 раза вищу швидкість метаболізму порівняно з Тиранозавром, що змушує гравця змінювати свою стратегію гри в залежності від обраного виду динозавра. Дрібні види вимушені постійно шукати ресурси, тоді як великі хижаки зосереджені на не частому полюванні.

Наведені розрахунки є ключовими, але не єдиними прикладами застосування подібного моделювання у проєкті. Аналогічний підхід був застосований до інших характеристик ігрових об’єктів.

Зокрема, розрахунок сили атаки спирається на збереження пропорцій сили укусу згідно з палеомеханічними дослідженнями, а місткість витривалості на

оцінці бігового потенціалу видів [23]. Час досягнення дорослого віку – це пропорційне стиснення реального періоду життя.

4.1.3 Результат балансування

Наукова достовірність і захоплива ігрова динаміка знаходяться у стані конфлікту і часто заперечують одне одного. Відхилення в одну зі сторін перетворює гру на симуляцію або аркаду. І в цьому випадку, якби проєкт був на 100% симулятором, ігровий процес став би нестерпно повільним для користувача – реальні хижаки, наприклад, жили до тридцяти років у середньому, і могли перетравлювати їжу кілька днів. З іншого боку, чиста аркадна гра зруйнувала б наукову цінність і унеможливила розвиток освітнього потенціалу проєкту.

Головним висновком етапу проєктування стало знаходження балансу. За допомогою математичного апарату, описаного у розділі 4.1.2, було сформовано гібридну систему компромісів.

– просторово-часова адаптація, при якій застосований кінематичний редуктор не лише зберігає історичні пропорції швидкості видів, а і стискає час переміщення по мапі. Це запобігає безглуздому блуканню гравця;

– ігрова гіперболізація метаболізму. Формули закону Клайбера забезпечують біологічну відповідність дослідженням, демонструючи хто і скільки їжі споживає. Проте сам процес виснаження був суттєво прискорений. Це підвищує щільність подій на хвилину у грі, змушує гравця взаємодіяти з ШІ більш активно;

– баланс здоров’я і пошкоджень розрахований так, щоб уникнути ефекту багаточасового забивання ворогів. Такий підхід притаманний аркадам, але мало відповідає дійсності, проте обраний варіант рішення дозволяє уникнути, також, смертей від одного удару, притаманним хардкорним симуляціям;

Наочна різниця між описаними підходами та обраний шлях оптимізації наведені у таблиці 4.1.

Таблиця 4.1

Порівняння параметрів симулятивного, аркадного та гібридних підходів.

Параметр	Симуляція	Аркада	Forgotten Era
Швидкість переміщення	Реалістична. Тиранозавр перетинає ліс годинами	Гіперболізована. Більшість динозаврів бігають зі схожою швидкістю	Масштабована. Збережено пропорції видів, які прискорено заради динаміки
Цикл харчування	Хижак може їсти раз на декілька днів	Штучний таймер. Потрібно полювати постійно через штучне ускладнення	Біомеханічна. Закон Клайбера адаптовано під ігровий час
Time-To-Kill	Смерть може відбутись через дні після поранення	Дрібних ворогів потрібно бити довго, що не корелює з їх розмірами	Залежна від маси. Діделфодон помирає за 1 укус, великі динозаври потребують тактичного бою

Розроблений варіант і застосування гібридної архітектури дозволили створити продукт, який не перетворюється на повну симуляцію, втрачаючи притаманну іграм динаміку й розважальну частку, але і не стає повною аркадою, що підбирає значення змінних виключно емпіричним методом. Налаштовані змінні забезпечують динамічний ігровий процес, залишаючись при цьому вірними базовим палеонтологічним законам. Процес конвертації зображено на рисунку 4.1.

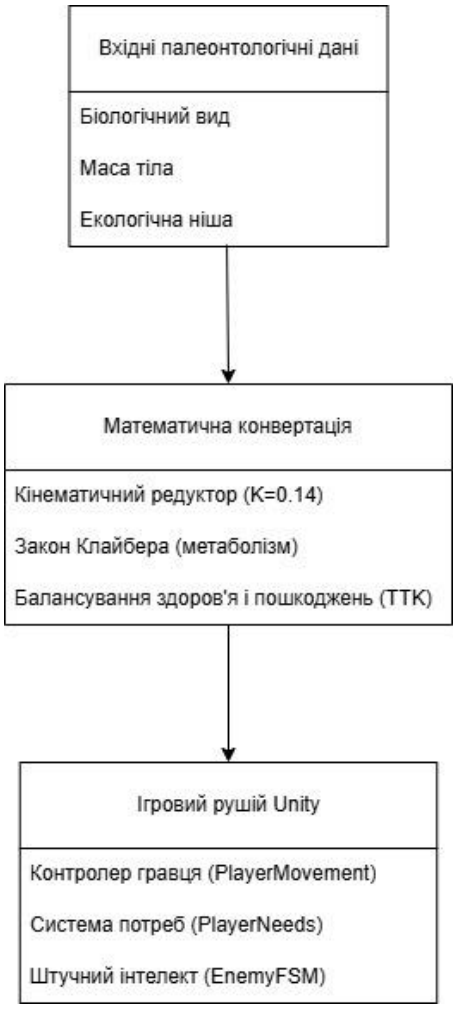


Рис. 4.1. Процес конвертації палеонтологічних даних в ігрові параметри

4.2 Стратегія та результати тестування програмного продукту

Для перевірки коректності роботи розробленого ігрового застосунку була сформована стратегія тестування, яка базується на функціональних і нефункціональних вимогах, визначених ще у першому розділі. Процес тестування розділений на чотири основні напрями: перевірка базових механік, тестування штучного інтелекту, перевірка збережених даних і перевірка продуктивності. Основний акцент було зроблено на ітераційному характері перевірок, що дозволило проводити гнучке налаштування числових коефіцієнтів балансу шляхом послідовних наближень. Використання циклічного підходу «тест-аналіз-корекція» забезпечило досягнення стабільності ігрового циклу і

високої точності відтворення розрахованих біологічних параметрів у віртуальному просторі.

4.2.1 Тестування ігрових механік

Мета цього етапу була у підтвердженні коректності роботи контролера гравця і внутрішньої логіки його розвитку.

Тестування підтвердило, що меню вибору коректно ініціалізує ігрову сесію з префабом обраного динозавра. Контролер переміщення працює без затримок, механіки атаки ініціюються відповідними тригерами.

Також було проведено інтеграційне тестування переходу між стадіями. При досягненні необхідної стадії росту система успішно збільшує масштаб моделі і нараховує універсальні бали розвитку. Розподіл цих балів через користувацький інтерфейс миттєво оновлює актуальне значення базових змінних персонажа без виникнення конфліктів у пам’яті. Інтерфейс екрану розподілу балів характеристик зображено на рисунку 4.2.

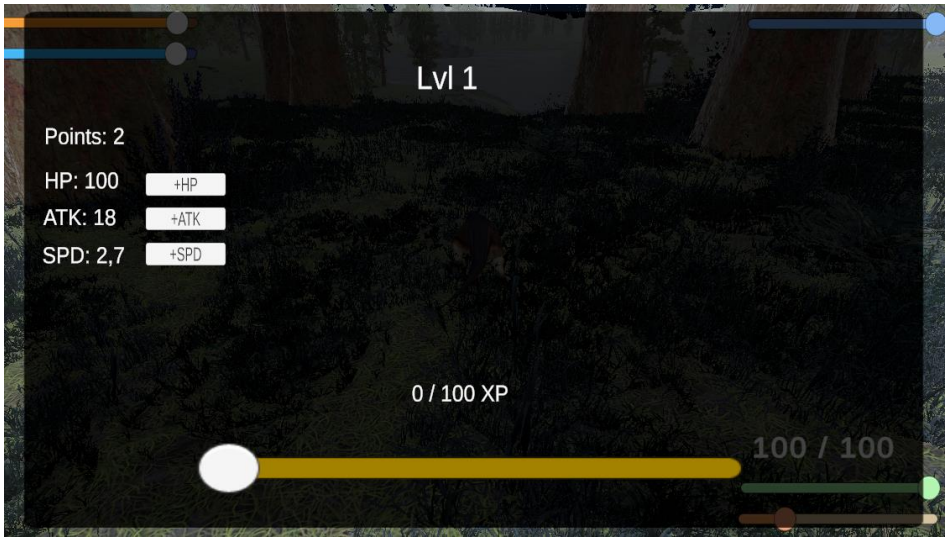


Рис. 4.2 Інтерфейс екрану росту та розподілу балів характеристик

За допомогою інструментів зневадження в інспекторі Unity фіксувалась швидкість падіння показника голоду і спраги. На цьому етапі було застосовано метод ітераційного тестування для валідації математичної моделі балансу. Сам

процес налаштування відбувався за алгоритмом поступового наближення. На початкових етапах коригувальні коефіцієнти змінювались із фіксованим кроком (±0,5) для визначення верхніх і нижніх меж норми. Після знаходження прийнятного діапазону крок ітерації зменшувався для точного налаштування параметрів. Ці кроки поступово збільшували або зменшували коефіцієнт, на який множились початкові константи.

Ітераційному коригуванню піддавалися усі ключові підсистеми гри. Наприклад, тестування бойової системи виявило необхідність зміни показників здоров'я та шкоди для дотримання цільового часу вбивства. Пряме перенесення пропорцій маси робило бої за участю великих хижаків занадто довгими, тому для їх показників шкоди було застосовано підвищувальні коефіцієнти. Для середніх видів початкове значення здоров'я виявилось занадто низьким для їхньої бойової ролі. Це також вимагало коригування для забезпечення достатнього вікна помилки гравця.

Окрему увагу також було приділено системі виживання та швидкості метаболізму. Перші ітерації тестів показали, що імплементований закон Клайбера працює коректно, проте виявили проблему ігрового балансу. Для видів з наднизькою масою, таких як Пектинодон, сире математичне значення параметру призводило до неіграбельного виснаження за п'ятнадцять секунд. Завдяки використаному підходу, ці формули були пропущені через алгоритм лінійної нормалізації [24]. Точні коригувальні коєфіцієнти дозволили обмежити таймери нормальним діапазоном від двох до десяти хвилин. При цьому зберіглися історично достовірні пропорції між різними видами динозаврів, що доводить єфективність ітераційного підходу для адаптації наукових даних до механік реального часу.

4.2.2 Тестування штучного інтелекту

Оскільки нефункціональні вимоги передбачали використання технології NavMesh для логіки навігації, було створено кілька тестових сценаріїв для перевірки поведінки ШІ.

Перевірка показала, що агенти коректно генерують шлях до цілі, обходячи перешкоди, такі як камні і дерева. Вони не виходять за межі навігаційної сітки.

Протестовано динамічну зміну станів неігрових персонажів. При наближенні до гравця на критичну дистанцію, ШІ успішно перериває стан патрулювання і переходить у стан переслідування або атаки в залежності від кількості здоров'я. Відображення переключення станів зображено на рисунку 4.3.

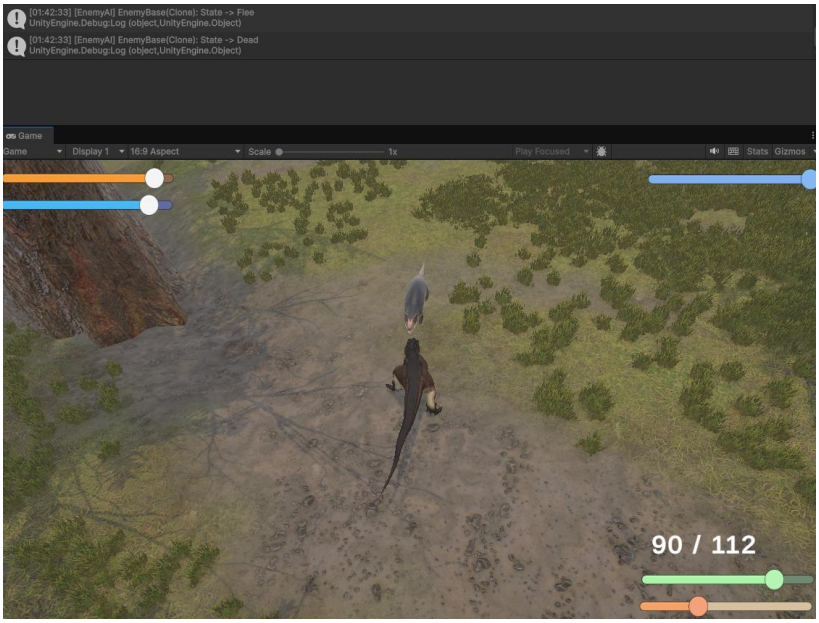


Рис. 4.3 Переключення станів штучного інтелекту під час гри

4.2.3 Тестування збереження прогресу та адаптивності дизайну

Заради забезпечення локального прогресу систему було протестовано на коректність серіалізації даних. Завершення ігрової сесії та її повторний запуск підтвердили, що збережені параметри успішно завантажуються. Гравець продовжує гру зі своїми покращеними характеристиками на тому етапі, на якому зупинився при попередньому збереженні.

Користувацький інтерфейс протестовано на різних розподільчих здатностях екрана. Елементи інтерфейсу адаптивні, надають доступ до прогресії, не перекриваючи ігровий огляд.

4.2.4 Профілювання та продуктивність

Згідно з нефункціональними вимогами, застосунок мав забезпечувати стабільну частоту кадрів на ОС Windows 10/11. Для замірів використовувався інструмент Unity Profiler [25]. Результати перевірки наведено на рисунку 4.4.

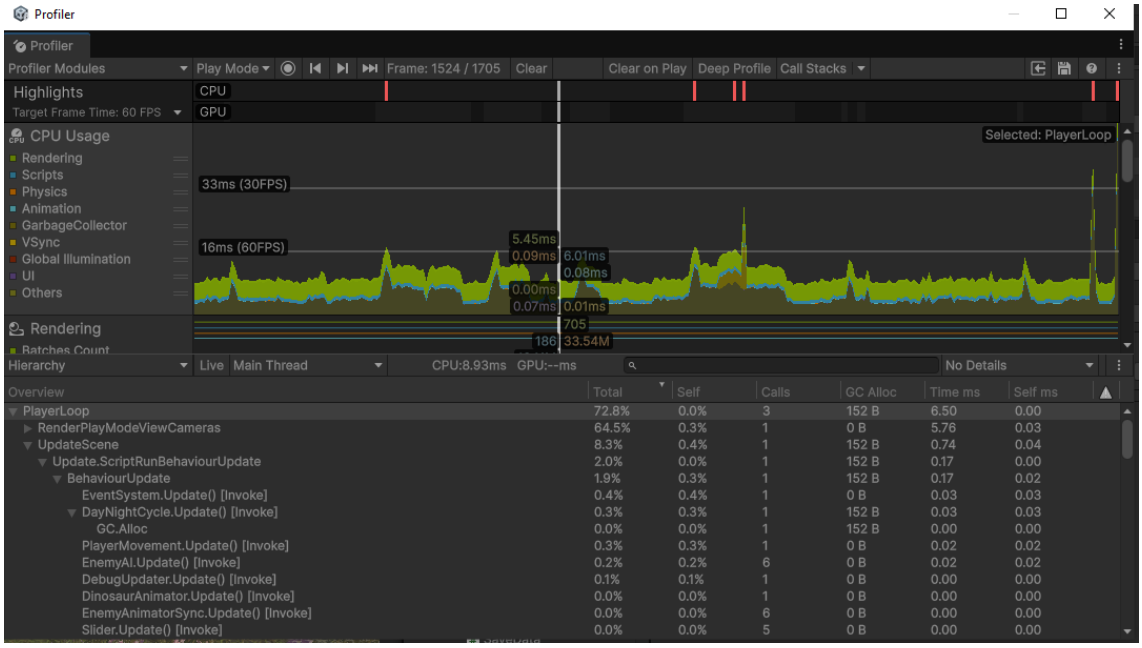


Рис. 4.4 Результати перевірки навантаження на CPU.

Результати профілювання на комп’ютері середньої конфігурації показали стабільну частоту кадрів на рівні 90+ FPS. Пікові навантаження на процесор під час розрахунку шляхів для кількох неігрових персонажів одночасно залишаються в межах норми і не викликають проблем. Витоків пам’яті під час переходу між стадіями росту також не виявлено.

Комплексне тестування доводить, що ігровий застосунок на фінальному етапі повністю відповідає всім поставленим функціональним і нефункціональним вимогам і залишає простір для покращення. Окрім базових сценаріїв, особливу увагу під час тестування приділено обробці граничних випадків та перевірці системи на відмовостійкість. Зокрема, успішно пройдено стрес-тести на одночасне ініціювання конфліктуючих ігрових подій, таких як, наприклад, отримання критичної шкоди в момент ініціалізації переходу на нову

стадію росту або спроба автоматичного збереження сесії безпосередньо під час смерті персонажа. Завдяки архітектурі диспетчера подій та чіткому розмежуванню станів у скінченному автоматі, система коректно ізолює і відхиляє невалідні транзакції. Це запобігає пошкодженню файлів збереження і винекнення критичних помилок під час виконання.

Результати профілювання підтвердили високий потенціал архітектури до подальшого масштабування. Завдяки принципам слабкої пов’язаності модулів, на майбутніх етапах розробки планується розширення тестового покриття за рахунок впровадження автоматизованих тестів.

Підсумовуючи результат, можна стверджувати, що застосований комплексний підхід до забезпечення якості проєкту дозволив створити не лише цікаву концептуально гру, а й технічно надійний застосунок. Проведена верифікація та валідація математичних моделей довели життєздатність гібридного підходу, в якому історичні палеонтологічні закони органічно поєднані з оптимізованими програмними алгоритмами. Розроблений проєкт демонструє стабільну роботу, володіє інтуїтивно зрозумілим інтерфейсом і повністю готовий до подальшого розширення контенту і впровадження нових ігрових механік. Фінальні випробування цільової збірки підтвердили відсутність критичних дефектів та падінь системи. Розроблений продукт демонструє стабільну роботу, володіє високою відмовостійкістю і повністю готовий до експлуатації як масштабований програмний фундамент для створення науково-орієнтованих ігор.

ВИСНОВКИ

У кваліфікаційній роботі успішно вирішено актуальне науково-практичне завдання – спроєктовано і розроблено прототип інтерактивного програмного продукту. Гра базується на основі палеонтологічних і палеоекологічних даних формації Хелл-Крік. У процесі виконання роботи було досягнуто всіх поставлених цілей та реалізовано визначені функціональні і нефункціональні вимоги.

Було проведено системний аналіз предметної області. Доведено, що інтеграція достовірних наукових даних у розважальний програмний продукт вимагає застосування специфічних архітектурних патернів для збереження балансу між симулятивністю і позитивним користувацьким досвідом.

Спроєктовано і впроваджено архітектуру продуктивного управління даними. Реалізація подієво-орієнтованої моделі дозволила досягти слабкої пов’язаності між модулями інтерфейсу, бойової системи і логіки росту. Використання ScriptableObject в середовищі розробки Unity дозволило повністю відокремити ігрову логіку від конфігураційних даних. Це гарантувало високу гнучкість налаштувань, оптимізацію застосунку і можливість легкого масштабування проєкту без втручань у вихідний код бази.

Реалізовано комплексну систему штучного інтелекту. Для неігрових персонажів використано логіку на базі скінченних автоматів, що у поєднанні з технологією динамічної навігації Unity NavMesh дозволило створити ШІ, який здатен адекватно реагувати на загрозу, генерувати шляхи обходу перешкод і перемикати стани поведінки в залежності від дій гравця.

Використано математичний апарат балансування системи. Для розрахунку ігрових параметрів були адаптовані реальні біологічні закономірності. Завдяки застосуванню методу ітераційного тестування з поступовим наближенням кроку коефіцієнтів, виявлений на етапі тестування конфлікт між достовірною симуляцією та ігровим циклом був успішно вирішений. Для цього використано

лінійну нормалізацію даних, що гарантувало комфортний геймплей зі збереженням історичних пропорцій.

Розроблено системи прогресії та збереження. Контролер гравця, система незалежного від роздільної здатності інтерфейсу, механіки росту – реалізовані. Впроваджено систему серіалізації даних, яка надійно зберігає та завантажує локальний прогрес користувача.

Проведено тестування і профілювання. Результати перевірки підтвердили стабільність програмного коду. Через правильний розподіл обчислювальних навантажень, таких як використання циклів замість постійного оновлення кожного кадру, продукт демонструє стабільну частоту кадрів і відповідає усім критеріям продуктивності.

Розроблений ігровий застосунок є повністю робочим прототипом, що поєднує інженерну надійність із сучасними підходами до геймдизайну. Архітектурні рішення, закладені в основу проєкту, залишають широкий простір для подальшого масштабування. Існує можливість додавання нових механік та покращення візуальної складової в майбутніх версіях.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Saurian [Електронний ресурс] // Steam. URL: https://store.steampowered.com/app/587450/Saurian/ (дата звернення: 17.04.2026).
2. The Isle [Електронний ресурс] // Steam. URL: https://store.steampowered.com/app/376210/The_Isle/ (дата звернення: 17.04.2026).
3. ARK: Survival Ascended [Електронний ресурс] // Steam. URL: https://store.steampowered.com/app/2399830/ARK_Survival_Ascended/ (дата звернення: 20.04.2026).
4. Jurassic World Evolution 2 [Електронний ресурс] // Steam. URL: https://store.steampowered.com/app/1244460/Jurassic_World_Evolution_2/ (дата звернення: 17.04.2026).
5. Fastovsky D. E., Bercovici A. The Hell Creek Formation and its contribution to the Cretaceous–Paleogene extinction: A short primer. Cretaceous Research. 2016. Vol. 57. P. 368–390.
6. Lyson T. R., Longrich N. R. Spatial niche partitioning in dinosaurs from the latest cretaceous (Maastrichtian) of North America. Proceedings of the Royal Society B: Biological Sciences. 2011. Vol. 278(1709). P. 1158–1164.
7. Adams E. Fundamentals of Game Design. 3rd ed. New Riders. 2013. 576 p.
8. Unity User Manual [Електронний ресурс] // Unity Documentation. URL: https://docs.unity3d.com/Manual/index.html (дата звернення: 17.04.2026).
9. Millington I. Artificial Intelligence for Games. 3rd ed. CRC Press. 2019. 896 p.
10. Freeman E., Robson E. Head First Design Patterns. O'Reilly Media. 2020. 672 p.
11. Програмування на C# [Електронний ресурс] // Microsoft Learn. URL: https://learn.microsoft.com/uk-ua/dotnet/csharp/ (дата звернення: 17.04.2026).
12. NavMesh Agent [Електронний ресурс] // Unity Documentation. URL: https://docs.unity3d.com/550/Documentation/Manual/class-NavMeshAgent.html (дата звернення: 18.04.2026).
13. Gregory J. Game Engine Architecture. 3rd ed. CRC Press. 2018. 1240 p.
14. Nystrom R. Game Programming Patterns. Genever Benning. 2014. 354 p.
15. ScriptableObject [Електронний ресурс] // Unity Documentation. URL: https://docs.unity3d.com/Manual/class-ScriptableObject.html (дата звернення: 18.04.2026).
16. JsonUtility [Електронний ресурс] // Unity Scripting API. URL: https://docs.unity3d.com/ScriptReference/JsonUtility.html (дата звернення: 19.04.2026).
17. Application.persistentDataPath [Електронний ресурс] // Unity Scripting API. URL: https://docs.unity3d.com/ScriptReference/Application-persistentDataPath.html (дата звернення: 19.04.2026).

18. System.IO Namespace [Електронний ресурс] // Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/api/system.io (дата звернення: 20.04.2026).
19. CharacterController [Електронний ресурс] // Unity Documentation. URL: https://docs.unity3d.com/Manual/class-CharacterController.html (дата звернення: 20.04.2026).
20. Events and routed events overview [Електронний ресурс] // Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/advanced/events-wpf (дата звернення: 20.04.2026).
21. Coroutines [Електронний ресурс] // Unity Documentation. URL: https://docs.unity3d.com/Manual/Coroutines.html (дата звернення: 20.04.2026).
22. Kleiber M. Body size and metabolism. Hilgardia. 1932. Vol. 6(11). P. 315–353.
23. Alexander R. M. Principles of Animal Locomotion. Princeton University Press. 2003. 384 p.
24. Mathf.Lerp [Електронний ресурс] // Unity Scripting API. URL: https://docs.unity3d.com/ScriptReference/Mathf.Lerp.html (дата звернення: 20.04.2026).
25. Unity Profiler [Електронний ресурс] // Unity Documentation. URL: https://docs.unity3d.com/Manual/Profiler.html (дата звернення: 20.04.2026).

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
Кафедра Технологій цифрового розвитку

Ігровий застосунок для симуляції життя динозаврів із використанням Unity

Виконала: Санкіна Яна Сергіївна
студентка групи ТЦР-41

Керівник: Дзюба Вадим Віталійович
старший викладач кафедри ТЦР

Київ-2026

АКТУАЛЬНІСТЬ

1. Зростання попиту на освітньо-розважальний контент для візуалізації наукових даних.
2. Потреба у достовірному відтворенні наукових фактів у реальному часі.
3. Необхідність оптимізації архітектури розроблених додатків.

МЕТА, ОБ'ЄКТ І ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи – покращення процесу програмного відтворення життєдіяльності палеоекосистеми за допомогою розробленого ігрового застосунку на основі методів математичного моделювання і об'єктно-орієнтованого проєктування.

Об'єкт дослідження – процес програмного відтворення та функціонування палеоекосистем в інтерактивному віртуальному середовищі.

Предмет дослідження – ігровий застосунок, розроблений на базі рушія Unity з використанням даних на основі палеонтологічних і палеоекологічних досліджень.

ЗАВДАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Для досягнення поставленої мети потрібно реалізувати декілька етапів:

1. Проаналізувати предметну область і існуючі аналоги на наявність технічних і концептуальних недоліків.
2. Спроєктувати архітектуру застосунку: склад ігрової системи, структуру даних і методи взаємодії об'єктів.
3. Розробити системи керування персонажем, бою та штучного інтелекту на базі скінчених автоматів, для більш автономної поведінки динозаврів.
4. Провести функціональне та ітеративне тестування, під час якого буде оцінено відповідність поставленим задачам, а також валідовано математичний баланс між симуляцією та ігровим процесом.

АНАЛІЗ АНАЛОГІВ

Назва гри	Реалізм	Цільова аудиторія	Переваги	Недоліки
ARK: Survival Ascended	Низький	Прихильники жанру survival-sandbox.	Великий вибір видів динозаврів, мережевий режим, розвинена система будівництва.	Складний інтерфейс, висока системна вимогливість до апаратних ресурсів, відсутність наукової складової
Jurassic World Evolution	Низький	Прихильники економічних стратегій і відомої кінофраншизи	Висока графічна якість, зручні механіки з менеджменту бази	Відсутність прямого контролю над динозаврами
Saurian	Високий	Користувачі, глибоко зацікавлені палеонтологією	Науково достовірне середовище, детальні моделі росту динозаврів	Заморожена розробка, відсутність динамічного геймплею
The Isle	Низький	Онлайн-спільнота, яка цікавиться багатокористувацькими іграми	Динамічна взаємодія між користувачами, великий вибір динозаврів	Відсутність освітнього елементу і ухил в жанр «хорор»
Проєктований застосунок (Forgotten Era)	Високий	Онлайн-спільноти прихильників динозаврів	Науково достовірне середовище, реалістична поведінка ШІ	Обмежена кількість ігрових видів на етапі прототипу

ВИМОГИ ДО ГРИ

Функціональні вимоги:

1. Ініціалізація ігрової сесії. Система повинна забезпечувати вибір та завантаження однієї з трьох зумовлених конфігурацій ігрових сутностей.
2. Обробка локомоції та вводу. Програмний модуль керування має обробляти команди користувача для реалізації переміщення у тривимірному просторі та активації бойових методів обраного персонажа.
3. Система динамічного росту. Програма повинна реалізувати алгоритм зміни станів росту сутності через лінійну інтерполяцію масштабів і модифікацію числових атрибутів – характеристик.
4. Модуль модифікації атрибутів. Система повинна забезпечувати інтерфейс для внесення змін у структуру даних персонажа через розподіл накопичених балів досвіду.
5. Збереження даних. Програма повинна реалізувати механізм серіалізації та десеріалізації поточного стану ігрових об'єктів через локальне сховище для збереження і відновлення прогресу.
6. Керування автономними агентами ШІ. Модуль штучного інтелекту повинен базуватися на скінченних автоматах для детермінованого перемикання між станами на основі вихідних даних від системи сприйняття.

Нефункціональні вимоги:

1. Технологічний стек навігації. Обчислення шляхів переміщення автономних агентів має здійснюватися засобами технології Unity NavMesh.
2. Продуктивність та сумісність. Застосунок має стабільно функціонувати на ОС Windows 10/11 x64 та підтримувати стабільну частоту кадрів не нижче 30 на комп'ютерах середньої конфігурації.
3. Стабільність графічного інтерфейсу. Програмна реалізація інтерфейсу користувача має забезпечувати коректне масштабування і позиціювання елементів при зміні роздільної здатності екрана.
4. Архітектурна зв'язність. Внутрішня взаємодія між підсистемами має базуватись на принципах слабкої пов'язаності через подієво-орієнтовану модель.

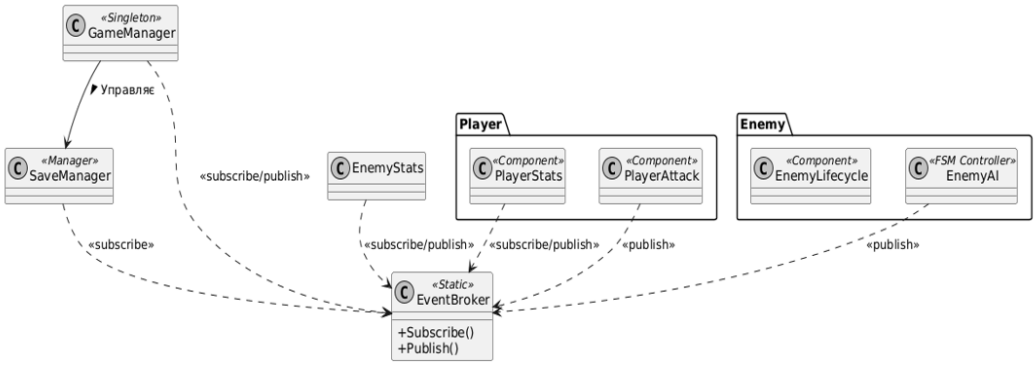
ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Компонентно-орієнтований підхід

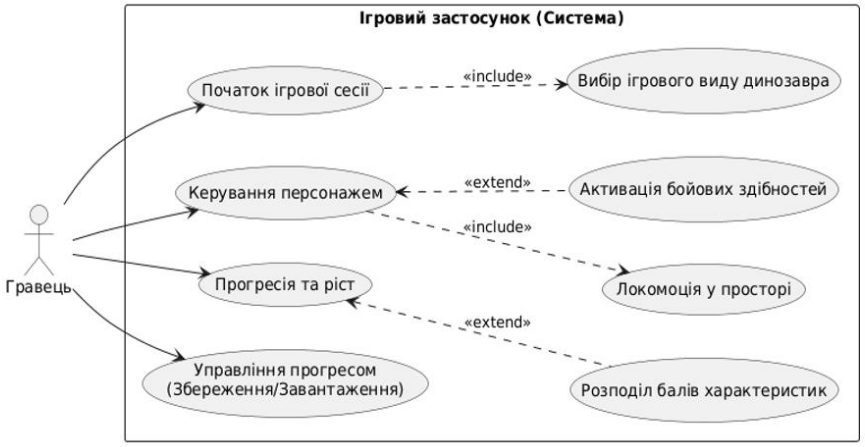
Вбудований NavMesh

Підтримка ScriptableObjects

ПОДІЄВО-ОРІЄНТОВАНА АРХІТЕКТУРА СИСТЕМИ



ДІАГРАМА ПРЕЦЕДЕНТІВ



МОДЕЛЮВАННЯ ІГРОВОГО БАЛАНСУ

Вид (Species)	Маса (кг)	Здоров'я (HP)	Шкода (ATK)	Швидкість (м/с)	Потреби (хв)
T-Rex	8 000	180	56	3.5	10.0
Dakotaraptor	300	50	21	6.3	7.8
Pectinodon	32	24	7	7.7	5.0

Масштабування метаболізму
Визначає швидкість виснаження базових потреб

$$M_{rate} = Mass^{-0.25}$$

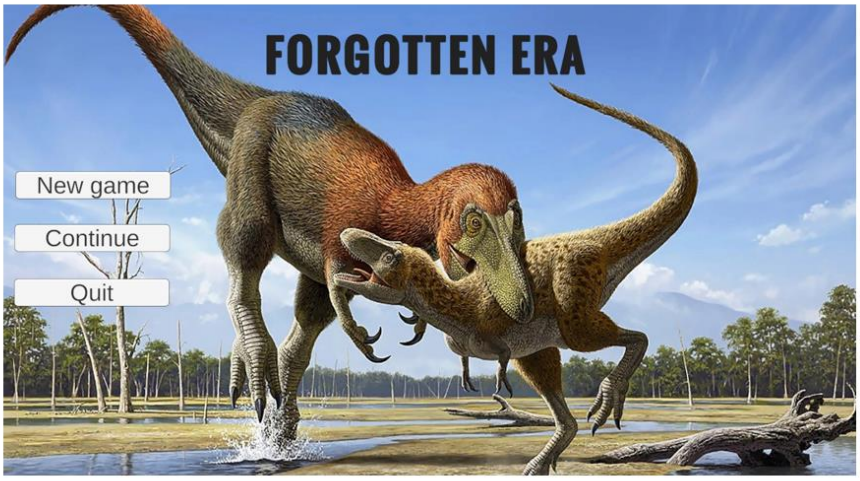
Кінематична редукція
Збереження історичних пропорцій швидкості видів, адаптованих до розмірів ігрової мапи

$$V_{unity} = V_{real} \times K_{kin}$$

ОЦІНКА РЕЗУЛЬТАТІВ

Параметр	Симуляція	Аркада	Forgotten Era
Швидкість переміщення	Реалістична. Тиранозавр перетинає ліс годинами	Гіперболізована. Більшість динозаврів бігають зі схожою швидкістю	Масштабована. Збережено пропорції видів, які прискорено заради динаміки
Цикл харчування	Хижак може їсти раз на декілька днів	Штучний таймер. Потрібно полювати постійно через штучне ускладнення	Біомеханічна. Закон Клайбера адаптовано під ігровий час
Time-To-Kill	Смерть може відбутись через дні після поранення	Дрібних ворогів потрібно бити довго, що не корелює з їх розмірами	Залежна від маси. Дідельфодон помирає за 1 укус, великі динозаври потребують тактичного бою

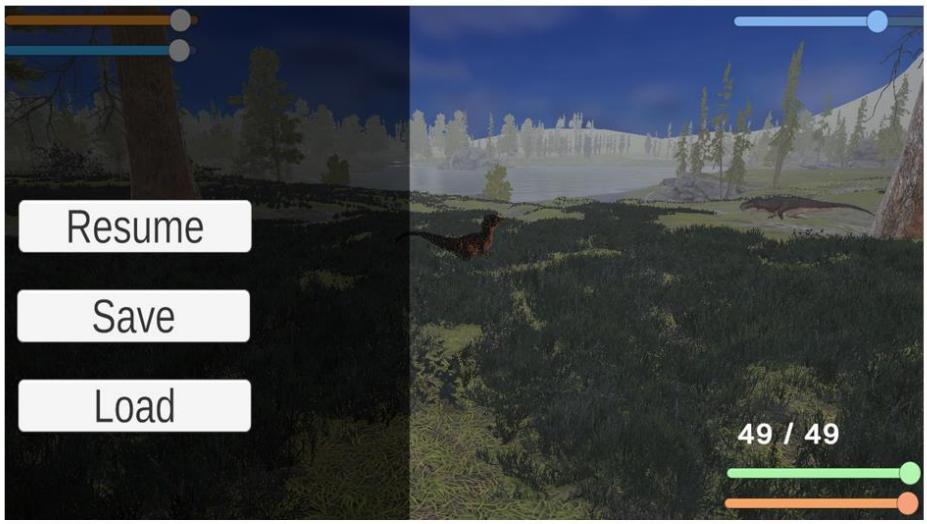
ЕКРАННІ ФОРМИ



ЕКРАННІ ФОРМИ

ЕКРАННІ ФОРМИ

ЕКРАННІ ФОРМИ



РЕЗУЛЬТАТИ РОБОТИ ПРОГРАМИ

АПРОБАЦІЯ РЕЗУЛЬТАТІВ

1. Санкіна Я.С, Дзюба В.В. Розробка науково-орієнтованого симулятора палеоекосистеми. // Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. – К.: ДУІКТ, 2026, (подано до друку).

2. Санкіна Я.С. Проєктування слабкопов'язаної архітектури ігрового застосунку на базі подієвої моделі. // Матеріали Всеукраїнської науково-технічної конференції «Технології цифрового розвитку: програмні системи та децентралізація». Збірник тез. – К.: ДУІКТ, 2026, (подано до друку).

ВИСНОВКИ

1. Проаналізовано предметну область, виявлено технічні й концептуальні недоліки існуючих аналогів

2. Спроєктовано масштабовану архітектуру із чіткою структурою даних та ізоляцією логіки обчислень.

3. Розроблено автономний ШІ, системи керування персонажем та унікальних бойових механік.

4. Проведено комплексне тестування, оптимізовано продуктивність та успішно валідовано математичний баланс гри.

ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

Лістинг Б.1 – Скрипт штучного інтелекту EnemyAI.cs

```
[RequireComponent(typeof(NavMeshAgent))]
[RequireComponent(typeof(Damageable))]
public class EnemyAI : MonoBehaviour
{
    private enum State { Patrol, Chase, Attack, Flee, Dead }

    [SerializeField] private float repathInterval = 0.25f;
    [SerializeField] private float stuckSpeedThreshold = 0.05f;
    [SerializeField] private float stuckTimeSeconds = 1.5f;

    private NavMeshAgent agent;
    private Damageable damageable;
    private EnemyHitbox[] enemyHitboxes;
    private EnemyRuntimeProfile profile;
    private Transform player;
    private State state = State.Patrol;
    private Vector3 spawnOrigin;
    private float repathTimer, stuckTimer, patrolPointTimer;
    private float lostTargetTimer, attackWindowTimer, attackCooldownTimer;
    private float attackTurnLockTimer, currentAttackTurnSpeed, fleeTimer;
    private Vector3 currentMoveTarget;
    private bool hasMoveTarget, isAttacking, isStunned;

    public void Configure(EnemyRuntimeProfile runtimeProfile,
        Transform playerTransform, EnemyHitbox[] hitboxes)
    {
        profile = runtimeProfile;
        if (playerTransform != null) player = playerTransform;
        enemyHitboxes = hitboxes;
        SetHitboxActive(false);
    }

    private void Awake()
    {
        agent = GetComponent<NavMeshAgent>();
        damageable = GetComponent<Damageable>();
        spawnOrigin = transform.position;
    }

    private void OnEnable()
    {
        if (damageable != null)
        {
            damageable.Died += OnDied;
            damageable.OnDamaged += OnDamaged;
            damageable.OnStunChanged += OnStunChanged;
            damageable.OnBecameCorpse += OnCorpse;
        }
    }

    // OnDisable — аналогічна відписка від подій

    private void Update()
    {
        if (state == State.Dead || isStunned) return;
        if (agent != null && !agent.isOnNavMesh)
        {
            EnsureOnNavMesh();
            if (!agent.isOnNavMesh) return;
        }
        EnsurePlayerReference();
        switch (state)
        {
            case State.Patrol: TickPatrol(); break;
            case State.Chase:  TickChase();  break;
            case State.Attack: TickAttack(); break;
            case State.Flee:   TickFlee();   break;
        }
    }

    private void TickChase()
    {
        var cfg = GetProfile();
        SetAgentSpeed(cfg.chaseSpeed);
```

```
    if (agent != null && agent.isOnNavMesh) agent.isStopped = false;
    if (player == null) { EnterPatrol(); return; }

    float dist = GetFlatDistanceToPlayer();
    if (dist <= cfg.attackRange) { EnterAttack(); return; }

    if (dist > cfg.loseTargetRadius) lostTargetTimer += Time.deltaTime;
    else lostTargetTimer = 0f;

    if (lostTargetTimer >= cfg.loseTargetDelaySeconds)
    { EnterPatrol(); return; }

    repathTimer -= Time.deltaTime;
    if (repathTimer <= 0f || IsStuck())
    {
        repathTimer = Mathf.Max(0.05f, repathInterval);
        MoveTo(player.position);
    }
}

private void OnDamaged(Damageable d, int amount)
{
    if (state == State.Dead) return;
    float hpPct = GetHpPercent();
    if (state != State.Flee && hpPct < GetProfile().fleeHpThreshold)
        EnterFlee();
}

private void TickFlee()
{
    var cfg = GetProfile();
    SetAgentSpeed(cfg.fleeSpeed);
    if (agent != null && agent.isOnNavMesh) agent.isStopped = false;
    fleeTimer -= Time.deltaTime;
    if (fleeTimer <= 0f)
    {
        if (player != null && GetFlatDistanceToPlayer() <= cfg.detectionRadius
            && GetHpPercent() >= cfg.reengageHpThreshold)
            EnterChase();
        else
            EnterPatrol();
        return;
    }
    if (!hasMoveTarget || HasReachedDestination() || IsStuck())
    {
        Vector3 away = (transform.position -
            (player != null ? player.position : transform.position));
        away.y = 0f;
        if (away.sqrMagnitude < 0.01f) away = Random.insideUnitSphere;
        away.Normalize();
        float fleeDist = Mathf.Max(cfg.fleeDistance, cfg.attackRange * 3f);
        Vector3 candidate = transform.position + away * fleeDist;
        if (TryPickRandomNavMeshPoint(candidate, cfg.fleePickRadius,
            out Vector3 point))
            MoveTo(point);
    }
}

// EnterPatrol, EnterChase, EnterAttack, EnterFlee, EnterDead —
// встановлюють state, скидають таймери, вмикають/вимикають хітбокси

// TickPatrol — перевірка detectionRadius, вибір випадкової точки
// TickAttack — таймер cooldown, активація хітбоксу, поворот до цілі

// Допоміжні методи:
private float GetHpPercent()
{
    if (damageable == null) return 1f;
    return Mathf.Clamp01(damageable.CurrentHP / (float)damageable.MaxHP);
}

private void MoveTo(Vector3 destination)
{
    if (agent == null || !agent.isOnNavMesh) return;
    agent.isStopped = false;
    agent.SetDestination(destination);
    currentMoveTarget = destination;
    hasMoveTarget = true;
}
```

```
    private float GetFlatDistanceToPlayer()
    {
        if (player == null) return float.MaxValue;
        Vector3 a = transform.position; Vector3 b = player.position;
        a.y = 0f; b.y = 0f;
        return Vector3.Distance(a, b);
    }
}
```

Лістинг Б.2 – Скрипт життєвого циклу ворога EnemyLifecycle.cs

```
[RequireComponent(typeof(Damageable))]
public class EnemyLifecycle : MonoBehaviour
{
    [SerializeField] private float defaultCorpseLifetimeSeconds = 600f;
    [SerializeField] private int defaultCorpseUses = 5;
    [SerializeField] private float defaultFoodPerBite = 25f;
    [SerializeField] private Vector3 corpseRotationEuler = new Vector3(0f, 0f, 90f);
    [SerializeField] private MonoBehaviour[] additionalBehavioursToDisable;

    private Damageable damageable;
    private bool isCorpse;
    private float configuredCorpseLifetime = -1f;
    private int configuredCorpseUses = -1;

    private void Awake() { damageable = GetComponent<Damageable>(); }

    private void OnEnable() { damageable.Died += OnDamageableDied; }
    private void OnDisable() { damageable.Died -= OnDamageableDied; }

    public void ConfigureCorpse(float lifetime, int uses)
    {
        configuredCorpseLifetime = Mathf.Max(1f, lifetime);
        configuredCorpseUses = Mathf.Max(1, uses);
    }

    private void OnDamageableDied(Damageable dead)
    {
        if (dead != damageable || isCorpse) return;
        ConvertToCorpse();
    }

    private void ConvertToCorpse()
    {
        isCorpse = true;
        damageable.MarkAsCorpse();
        DisableCombatBehaviours();
        ApplyCorpsePose();
        SetupConsumable();

        EventBroker.Publish(new EnemyCorpseSpawnedEvent
        {
            InstanceId = gameObject.GetInstanceID(),
            CorpseObject = gameObject
        });

        float lifetime = (configuredCorpseLifetime > 0f)
            ? configuredCorpseLifetime : defaultCorpseLifetimeSeconds;
        StartCoroutine(RemoveCorpseAfterDelay(lifetime));
    }

    private void DisableCombatBehaviours()
    {
        foreach (var hb in GetComponentsInChildren<EnemyHitbox>(true))
            hb.enabled = false;
        var ai = GetComponent<EnemyAI>();
        if (ai != null) ai.enabled = false;
        var nav = GetComponent<NavMeshAgent>();
        if (nav != null) nav.enabled = false;
        var anim = GetComponentInChildren<Animator>();
        if (anim != null) anim.enabled = false;
    }

    private void SetupConsumable()
    {
        Consumable c = gameObject.AddComponent<Consumable>();
        c.type = Consumable.ConsumableType.Food;
        c.restoreAmountPerBite = defaultFoodPerBite;
        c.usesLeft = (configuredCorpseUses > 0)
```

```
            ? configuredCorpseUses : defaultCorpseUses;
    }

    private IEnumerator RemoveCorpseAfterDelay(float delay)
    {
        yield return new WaitForSeconds(delay);
        if (this != null && isCorpse) Destroy(gameObject);
    }
}
```

Лістинг Б.3 – Скрипт обробки вхідних пошкоджень Damageable.cs

```
public class Damageable : MonoBehaviour
{
    [SerializeField] private int maxHP = 100;
    [SerializeField] private int xpReward = 25;

    private int currentHP;
    public bool IsDead { get; private set; }
    public bool IsCorpse { get; private set; }
    public bool IsStunned { get; private set; }
    public int MaxHP => maxHP;
    public int CurrentHP => currentHP;

    public event Action<Damageable> Died;
    public event Action<Damageable, int> OnDamaged;
    public event Action<Damageable, bool> OnStunChanged;
    public event Action<Damageable> OnBecameCorpse;

    private void Start() { currentHP = maxHP; }

    public void TakeDamage(int amount)
    {
        if (IsDead || IsCorpse) return;
        currentHP -= amount;
        OnDamaged?.Invoke(this, amount);
        if (currentHP <= 0) Die();
    }

    public void ApplyBleed(int damagePerTick, float duration)
    {
        if (IsDead || IsCorpse) return;
        StartCoroutine(BleedCoroutine(damagePerTick, duration));
    }

    public void ApplyStun(float duration)
    {
        if (IsDead || IsCorpse) return;
        StartCoroutine(StunCoroutine(duration));
    }

    public void SetRuntimeStats(int newMaxHP, int newXpReward)
    {
        maxHP = Mathf.Max(1, newMaxHP);
        xpReward = Mathf.Max(0, newXpReward);
        IsDead = false; IsCorpse = false; currentHP = maxHP;
    }

    public void MarkAsCorpse()
    {
        IsCorpse = true; IsStunned = false;
        OnBecameCorpse?.Invoke(this);
    }

    private void Die()
    {
        if (IsDead) return;
        IsDead = true;
        Died?.Invoke(this);
        EventBroker.Publish(new EnemyKilledEvent { XPReward = xpReward });
    }

    private IEnumerator BleedCoroutine(int dmg, float dur)
    {
        float elapsed = 0f;
        while (elapsed < dur && !IsDead)
        { yield return new WaitForSeconds(1f); elapsed++; TakeDamage(dmg); }
    }
```

```
    private IEnumerator StunCoroutine(float dur)
    {
        IsStunned = true; OnStunChanged?.Invoke(this, true);
        yield return new WaitForSeconds(dur);
        IsStunned = false; OnStunChanged?.Invoke(this, false);
    }
}
```

Лістинг Б.4 – Скрипт хітбоксу ворога EnemyHitbox.cs

```
public class EnemyHitbox : MonoBehaviour
{
    public int attackDamage = 5;
    private List<Collider> hitTargets;

    void OnEnable()
    {
        if (hitTargets == null) hitTargets = new List<Collider>();
        hitTargets.Clear();
    }

    void OnTriggerEnter(Collider other) { TryDealDamage(other); }
    void OnTriggerStay(Collider other) { TryDealDamage(other); }

    private void TryDealDamage(Collider other)
    {
        if (hitTargets.Contains(other)) return;

        PlayerHealth player = other.GetComponent<PlayerHealth>();
        if (player == null) player = other.GetComponentInParent<PlayerHealth>();
        if (player == null) return;

        Vector3 hitPoint = other.ClosestPoint(transform.position);
        player.TakeDamage(attackDamage);
        EventBroker.Publish(new PlayerDamagedEvent
        {
            Player = player, DamageAmount = attackDamage, HitPoint = hitPoint
        });
        hitTargets.Add(other);
    }
}
```

Лістинг Б.5 – Скрипт системи досвіду ExperienceSystem.cs

```
public class ExperienceSystem : MonoBehaviour
{
    [SerializeField] private int baseXPPerLevel = 100;
    [SerializeField] private int xpScaling = 150;
    [SerializeField] private int statPointsPerLevel = 1;

    private int currentXP = 0;
    private int currentLevel = 1;

    public int CurrentXP => currentXP;
    public int CurrentLevel => currentLevel;
    public int XPToNextLevel => baseXPPerLevel + (currentLevel - 1) * xpScaling;

    private void OnEnable()
    { EventBroker.Subscribe<EnemyKilledEvent>(OnEnemyKilled); }
    private void OnDisable()
    { EventBroker.Unsubscribe<EnemyKilledEvent>(OnEnemyKilled); }

    public void LoadState(int xp, int level)
    {
        currentXP = Mathf.Max(0, xp);
        currentLevel = Mathf.Max(1, level);
        PublishXPEvent();
    }

    private void OnEnemyKilled(EnemyKilledEvent e) { AddXP(e.XPReward); }

    public void AddXP(int amount)
    {
        currentXP += amount;
        while (currentXP >= XPToNextLevel)
        {
```

```
            currentXP -= XPToNextLevel;
            currentLevel++;
            EventBroker.Publish(new LevelUpEvent
                { NewLevel = currentLevel, StatPoints = statPointsPerLevel });
        }
        PublishXPEvent();
    }

    private void PublishXPEvent()
    {
        EventBroker.Publish(new ExperienceChangedEvent
        {
            CurrentXP = currentXP,
            XPToNextLevel = XPToNextLevel,
            Level = currentLevel
        });
    }
}
```

Лістинг Б.6 – Скрипт системи росту персонажа PlayerGrowth.cs

```
public class PlayerGrowth : MonoBehaviour
{
    [SerializeField] private DinosaurSpeciesData speciesData;
    private float currentGrowth = 0f;
    private int currentStageIndex = -1;
    private int bonusHP = 0, bonusATK = 0;
    private float bonusSPD = 0f;

    public float CurrentGrowth => currentGrowth;
    public int CurrentStageIndex => currentStageIndex;

    public void Initialize(DinosaurSpeciesData data)
    {
        speciesData = data;
        currentGrowth = Mathf.Max(speciesData.maxGrowth * 0.01f,
            speciesData.growthPerTick);
        currentStageIndex = -1;
        UpdateScaleAndStats();
        PublishGrowthEvent();
        StartCoroutine(GrowthLoop());
    }

    public void LoadState(float growth, int stageIndex)
    {
        StopAllCoroutines();
        currentGrowth = Mathf.Clamp(growth, 0f, speciesData.maxGrowth);
        currentStageIndex = stageIndex;
        UpdateScaleAndStats();
        if (currentGrowth < speciesData.maxGrowth)
            StartCoroutine(GrowthLoop());
    }

    private void OnEnable()
    { EventBroker.Subscribe<PlayerBonusStatsUpdatedEvent>(OnBonusStats); }
    private void OnDisable()
    { EventBroker.Unsubscribe<PlayerBonusStatsUpdatedEvent>(OnBonusStats); }

    private void OnBonusStats(PlayerBonusStatsUpdatedEvent e)
    {
        bonusHP = e.BonusHP; bonusATK = e.BonusATK; bonusSPD = e.BonusSPD;
        if (speciesData != null) UpdateScaleAndStats();
    }

    private IEnumerator GrowthLoop()
    {
        while (currentGrowth < speciesData.maxGrowth)
        {
            yield return new WaitForSeconds(speciesData.growthInterval);
            currentGrowth = Mathf.Min(currentGrowth + speciesData.growthPerTick,
                speciesData.maxGrowth);
            UpdateScaleAndStats();
            PublishGrowthEvent();
            CheckStageThresholds();
        }
    }

    private void CheckStageThresholds()
    {
```

```
        if (speciesData.growthStageThresholds == null) return;
        float pct = (currentGrowth / speciesData.maxGrowth) * 100f;
        for (int i = currentStageIndex + 1;
             i < speciesData.growthStageThresholds.Length; i++)
        {
            if (pct >= speciesData.growthStageThresholds[i])
            {
                currentStageIndex = i;
                int bonus = (i < speciesData.stageStatBonus.Length)
                    ? speciesData.stageStatBonus[i] : 1;
                EventBroker.Publish(new GrowthStageReachedEvent
                    { StageIndex = i + 1, BonusPoints = bonus });
            }
            else break;
        }
    }

    private void UpdateScaleAndStats()
    {
        float pct = currentGrowth / speciesData.maxGrowth;
        float s = speciesData.GetScale(pct);
        transform.localScale = new Vector3(s, s, s);

        EventBroker.Publish(new PlayerStatsUpdatedEvent
        {
            NewMaxHP = speciesData.GetMaxHP(pct) + bonusHP,
            NewAttackDamage = speciesData.GetAttackDamage(pct) + bonusATK,
            NewMoveSpeed = speciesData.GetMoveSpeed(pct) + bonusSPD
        });
        EventBroker.Publish(new PlayerNeedsCapacityUpdatedEvent
        {
            NewMaxHunger = speciesData.GetMaxHunger(pct),
            NewMaxThirst = speciesData.GetMaxThirst(pct),
            HungerDecayRate = speciesData.hungerDecayRate,
            ThirstDecayRate = speciesData.thirstDecayRate
        });
    }

    private void PublishGrowthEvent()
    {
        EventBroker.Publish(new PlayerGrowthChangedEvent
            { CurrentGrowth = currentGrowth, MaxGrowth = speciesData.maxGrowth });
    }
}
```

Лістинг Б.7 – Скрипт системи розподілу характеристик StatUpgradeSystem.cs

```
public class StatUpgradeSystem : MonoBehaviour
{
    [SerializeField] private int hpPerPoint = 15;
    [SerializeField] private int atkPerPoint = 3;
    [SerializeField] private float spdPerPoint = 0.3f;

    private int availablePoints = 0;
    private int bonusHP, bonusATK;
    private float bonusSPD;

    public int AvailablePoints => availablePoints;
    public int BonusHP => bonusHP;
    public int BonusATK => bonusATK;
    public float BonusSPD => bonusSPD;

    public void LoadState(int points, int hp, int atk, float spd)
    {
        availablePoints = Mathf.Max(0, points);
        bonusHP = hp; bonusATK = atk; bonusSPD = spd;
        PublishAll();
    }

    private void OnEnable()
    {
        EventBroker.Subscribe<GrowthStageReachedEvent>(OnStageReached);
        EventBroker.Subscribe<LevelUpEvent>(OnLevelUp);
    }
    private void OnDisable()
    {
        EventBroker.Unsubscribe<GrowthStageReachedEvent>(OnStageReached);
        EventBroker.Unsubscribe<LevelUpEvent>(OnLevelUp);
    }
```

```
    private void OnStageReached(GrowthStageReachedEvent e)
    { availablePoints += e.BonusPoints; PublishPointsEvent(); }
    private void OnLevelUp(LevelUpEvent e)
    { availablePoints += e.StatPoints; PublishPointsEvent(); }

    // Аналогічні методи: UpgradeATK(), UpgradeSPD()
    public bool UpgradeHP()
    {
        if (availablePoints <= 0) return false;
        availablePoints--;
        bonusHP += hpPerPoint;
        PublishAll();
        return true;
    }

    private void PublishAll()
    {
        PublishPointsEvent();
        EventBroker.Publish(new PlayerBonusStatsUpdatedEvent
            { BonusHP = bonusHP, BonusATK = bonusATK, BonusSPD = bonusSPD });
    }

    private void PublishPointsEvent()
    {
        EventBroker.Publish(new StatPointsChangedEvent
            { AvailablePoints = availablePoints });
    }
}
```

Лістинг Б.8 – Структура даних збереження SaveData.cs

```
[System.Serializable]
public class SaveData
{
    public int saveVersion;
    public string speciesName;
    public float posX, posY, posZ;
    public float rotationY;
    public float currentGrowth;
    public int growthStageIndex;
    public int currentHP;
    public int currentXP, currentLevel;
    public int availablePoints, bonusHP, bonusATK;
    public float bonusSPD;
    public float currentHunger, currentThirst;
    public string saveTimestamp;
    public int slotIndex;
}
```

Лістинг Б.9 – Скрипт системи збереження та серіалізації SaveSystem.cs

```
public static class SaveSystem
{
    public const int MAX_SLOTS = 3;
    private const int CURRENT_SAVE_VERSION = 1;
    private const string FILE_PREFIX = "save_slot_";

    public static bool Save(int slotIndex)
    {
        slotIndex = Mathf.Clamp(slotIndex, 0, MAX_SLOTS - 1);
        GameObject player = FindPlayer();
        if (player == null) return false;

        SaveData data = CollectPlayerData(player, slotIndex);
        string json = JsonUtility.ToJson(data, true);
        string path = GetSavePath(slotIndex);

        string dir = Path.GetDirectoryName(path);
        if (!Directory.Exists(dir)) Directory.CreateDirectory(dir);
        File.WriteAllText(path, json);
        return true;
    }

    public static SaveData Load(int slotIndex)
    {
        string path = GetSavePath(Mathf.Clamp(slotIndex, 0, MAX_SLOTS - 1));
```

```
        if (!File.Exists(path)) return null;
        string json = File.ReadAllText(path);
        return JsonUtility.FromJson<SaveData>(json);
    }

    public static bool HasSave(int slot)
        => File.Exists(GetSavePath(Mathf.Clamp(slot, 0, MAX_SLOTS - 1)));

    public static string GetSavePath(int slot)
        => Path.Combine(Application.persistentDataPath,
            FILE_PREFIX + slot + ".json");

    private static SaveData CollectPlayerData(GameObject player, int slot)
    {
        SaveData data = new SaveData();
        data.saveVersion = CURRENT_SAVE_VERSION;
        data.slotIndex = slot;
        data.saveTimestamp = DateTime.Now.ToString("dd.MM.yyyy HH:mm");
        data.posX = player.transform.position.x;
        data.posY = player.transform.position.y;
        data.posZ = player.transform.position.z;
        data.rotationY = player.transform.eulerAngles.y;

        if (GameSession.SelectedSpecies != null)
            data.speciesName = GameSession.SelectedSpecies.speciesName;

        var growth = player.GetComponent<PlayerGrowth>();
        if (growth != null)
        { data.currentGrowth = growth.CurrentGrowth;
          data.growthStageIndex = growth.CurrentStageIndex; }

        var hp = player.GetComponent<PlayerHealth>();
        if (hp != null) data.currentHP = hp.currentHP;

        var xp = player.GetComponent<ExperienceSystem>();
        if (xp != null)
        { data.currentXP = xp.CurrentXP; data.currentLevel = xp.CurrentLevel; }

        var stats = player.GetComponent<StatUpgradeSystem>();
        if (stats != null)
        { data.availablePoints = stats.AvailablePoints;
          data.bonusHP = stats.BonusHP; data.bonusATK = stats.BonusATK;
          data.bonusSPD = stats.BonusSPD; }

        var needs = player.GetComponent<PlayerNeeds>();
        if (needs != null)
        { data.currentHunger = needs.CurrentHunger;
          data.currentThirst = needs.CurrentThirst; }

        return data;
    }

    // FindPlayer — пошук гравця за тегом або компонентами
}
```

Лістинг Б.10 – Скрипт автоматичного збереження AutoSaveManager.cs

```
public class AutoSaveManager : MonoBehaviour
{
    [SerializeField] private float autoSaveInterval = 300f;
    [SerializeField] private int autoSaveSlot = 0;

    private void Start() { StartCoroutine(AutoSaveLoop()); }

    private IEnumerator AutoSaveLoop()
    {
        yield return new WaitForSeconds(5f);
        while (true)
        {
            yield return new WaitForSeconds(autoSaveInterval);
            if (Time.timeScale > 0f)
            {
                int slot = Mathf.Clamp(autoSaveSlot, 0, SaveSystem.MAX_SLOTS - 1);
                SaveSystem.Save(slot);
            }
        }
    }
}
```

Лістинг Б.11 – Скрипт контролера ігрового інтерфейсу PlayerUIController.cs

```
public class PlayerUIController : MonoBehaviour
{
    [SerializeField] private Slider healthBar;
    [SerializeField] private TMP_Text healthText;
    [SerializeField] private Slider growthBar;
    [SerializeField] private Slider hungerBar;
    [SerializeField] private Slider thirstBar;
    [SerializeField] private Slider staminaBar;
    [SerializeField] private GameObject deathScreenPanel;

    private void OnEnable()
    {
        EventBroker.Subscribe<PlayerHealthChangedEvent>(UpdateHealthUI);
        EventBroker.Subscribe<PlayerDiedEvent>(ShowDeathScreen);
        EventBroker.Subscribe<PlayerGrowthChangedEvent>(UpdateGrowthUI);
        EventBroker.Subscribe<PlayerHungerChangedEvent>(UpdateHungerUI);
        EventBroker.Subscribe<PlayerThirstChangedEvent>(UpdateThirstUI);
        EventBroker.Subscribe<PlayerStaminaChangedEvent>(UpdateStaminaUI);
    }
    // OnDisable — аналогічна відписка від подій

    private void UpdateHealthUI(PlayerHealthChangedEvent e)
    {
        if (healthBar != null)
            healthBar.value = (e.MaxHP > 0) ? ((float)e.CurrentHP / e.MaxHP) : 0;
        if (healthText != null)
            healthText.text = e.CurrentHP + " / " + e.MaxHP;
    }

    private void ShowDeathScreen(PlayerDiedEvent e)
    {
        if (deathScreenPanel != null) deathScreenPanel.SetActive(true);
        Cursor.visible = true;
        Cursor.lockState = CursorLockMode.None;
    }

    // Аналогічні методи оновлення: UpdateGrowthUI, UpdateHungerUI,
    // UpdateThirstUI, UpdateStaminaUI — нормалізують значення 0..1
    private void UpdateHungerUI(PlayerHungerChangedEvent e)
    {
        if (hungerBar != null)
            hungerBar.value = (e.Max > 0) ? (e.Current / e.Max) : 0;
    }
}
```

Лістинг Б.12 – Скрипт меню покращення характеристик UpgradeMenuUI.cs

```
public class UpgradeMenuUI : MonoBehaviour
{
    [SerializeField] private GameObject upgradePanel;
    [SerializeField] private TMP_Text levelText, xpText, pointsText;
    [SerializeField] private TMP_Text hpValueText, atkValueText, spdValueText;
    [SerializeField] private Button hpButton, atkButton, spdButton;
    [SerializeField] private Slider xpBar;

    private StatUpgradeSystem upgradeSystem;
    private PlayerInput playerInput;
    private bool isOpen = false;
    private int currentLevel = 1, currentXP = 0, xpToNext = 100;
    private int currentMaxHP = 0, currentATK = 0;
    private float currentSPD = 0f;

    private void Awake()
    {
        upgradeSystem = FindFirstObjectByType<StatUpgradeSystem>();
        playerInput = FindFirstObjectByType<PlayerInput>();
        if (upgradePanel != null) upgradePanel.SetActive(false);
    }

    private void OnEnable()
    {
        EventBroker.Subscribe<ExperienceChangedEvent>(OnXPChanged);
        EventBroker.Subscribe<StatPointsChangedEvent>(OnPointsChanged);
        EventBroker.Subscribe<PlayerStatsUpdatedEvent>(OnStatsUpdated);
        if (hpButton != null) hpButton.onClick.AddListener(OnHPClicked);
        if (atkButton != null) atkButton.onClick.AddListener(OnATKClicked);
```

```
        if (spdButton != null) spdButton.onClick.AddListener(OnSPDClicked);
    }
    // OnDisable — відписка від подій та видалення слухачів кнопок

    private void Update()
    {
        if (playerInput != null && playerInput.UpgradeMenuInput)
            ToggleMenu();
    }

    public void ToggleMenu()
    {
        isOpen = !isOpen;
        if (upgradePanel != null) upgradePanel.SetActive(isOpen);
        Cursor.visible = isOpen;
        Cursor.lockState = isOpen ? CursorLockMode.None : CursorLockMode.Locked;
        if (isOpen) UpdateAllTexts();
    }

    private void OnXPChanged(ExperienceChangedEvent e)
    { currentLevel = e.Level; currentXP = e.CurrentXP;
      xpToNext = e.XPToNextLevel; if (isOpen) UpdateAllTexts(); }

    private void OnStatsUpdated(PlayerStatsUpdatedEvent e)
    { currentMaxHP = e.NewMaxHP; currentATK = e.NewAttackDamage;
      currentSPD = e.NewMoveSpeed; if (isOpen) UpdateAllTexts(); }

    private void OnHPClicked()
    { if (upgradeSystem != null) upgradeSystem.UpgradeHP(); }
    // Аналогічно: OnATKClicked -> UpgradeATK, OnSPDClicked -> UpgradeSPD

    private void UpdateAllTexts()
    {
        int points = (upgradeSystem != null) ? upgradeSystem.AvailablePoints : 0;
        if (levelText != null) levelText.text = "Lvl " + currentLevel;
        if (xpText != null) xpText.text = currentXP + " / " + xpToNext + " XP";
        if (pointsText != null) pointsText.text = "Points: " + points;
        if (hpValueText != null) hpValueText.text = "HP: " + currentMaxHP;
        if (atkValueText != null) atkValueText.text = "ATK: " + currentATK;
        if (spdValueText != null) spdValueText.text = "SPD: " + currentSPD.ToString("F1");

        bool hasPoints = points > 0;
        if (hpButton != null) hpButton.interactable = hasPoints;
        if (atkButton != null) atkButton.interactable = hasPoints;
        if (spdButton != null) spdButton.interactable = hasPoints;
    }
}
```