НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій

ПОГОДЖЕНО	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ**
Декан факультету	**Завідувач кафедри комп'ютерних наук**
_____Ігор БОЛБОТ	**_____ Белла ГОЛУБ**
(підпис)	(підпис)
(підпис)	(підпис)
«___» _____ 2026 р.	«___» _____ 2026 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Програмне забезпечення піксельної 2D RPG-гри з елементами конструктора»
Спеціальність «121 Інженерія програмного забезпечення»
Освітньо-професійна програма «Інженерія програмного забезпечення»

Гарант освітньої програми
к.т.н., доцент _____ **Ганна ВАЙГАНГ**
(підпис)
(підпис)

Керівник бакалаврської кваліфікаційної роботи
к.т.н., доцент _____ **Белла ГОЛУБ**
(підпис)
(підпис)

Виконала _____ **Тетяна МАЛИНКА**
(підпис)
(підпис)

Київ-2026

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерних наук

к.т.н., доцент _____ Белла ГОЛУБ

«___» _____ 2025 р.

ЗАВДАННЯ
ДО ВИКОНАННЯ БАКАЛАВРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ
ЗДОБУВАЧУ

Малинці Тетяні Олександрівні

Спеціальність «121 Інженерія програмного забезпечення»
Освітньо-професійна програма «Інженерія програмного забезпечення»
Тема бакалаврської кваліфікаційної роботи
«Програмне забезпечення піксельної 2D RPG-гри з елементами конструктора»
затверджена наказом від «19» грудня 2025 р. № 3204 «С»
Термін подання завершеної роботи на кафедру «22» травня 2026 р.
Вихідні дані до бакалаврської кваліфікаційної роботи:
матеріали з проєктування та розробки комп'ютерних ігор; документація середовища GameMaker Studio 2; вимоги до програмного забезпечення піксельної 2D RPG-гри.
Перелік питань, що підлягають дослідженню: аналіз предметної області та існуючих ігрових рішень; визначення функціональних і нефункціональних вимог до програмного продукту; проєктування архітектури 2D RPG-гри; розробка системи керування персонажем; визначення вимог до впровадження та експлуатації програмного продукту.

Керівник бакалаврської кваліфікаційної роботи _____ **Белла ГОЛУБ**

Завдання взяла до виконання _____ **Тетяна МАЛИНКА**

Зміст

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

2D — двовимірний простір, two-dimensional. У роботі використовується для позначення двовимірної графіки та ігрового середовища.

RPG — рольова гра, Role-Playing Game. Жанр гри, у якому гравець керує персонажем, взаємодіє з ігровим світом, виконує завдання та впливає на проходження через власні дії.

NPC — неігровий персонаж, Non-Player Character. Персонаж, яким не керує гравець, але з яким можна взаємодіяти під час проходження гри.

GML — мова GameMaker, GameMaker Language. Мова програмування, яка використовується для реалізації логіки гри в середовищі «GameMaker Studio 2».

GUI — графічний інтерфейс користувача, Graphical User Interface. Частина програми, через яку користувач бачить меню, діалоги, інвентар, повідомлення та інші візуальні елементи.

HUD — екранний ігровий інтерфейс, Heads-Up Display. Частина інтерфейсу, яка показує основні дані про стан персонажа, наприклад рівень, HP, шкоду та досвід.

JSON — текстовий формат обміну даними, JavaScript Object Notation. У роботі використовується для збереження та завантаження прогресу гри.

HP — очки здоров’я, Health Points. Показник здоров’я персонажа або ворога.

EXP — досвід, Experience Points. Показник досвіду персонажа, який може нараховуватися після перемоги над ворогами або виконання ігрових дій.

UML — уніфікована мова моделювання, Unified Modeling Language. Мова графічного опису структури та поведінки системи, яка використовується для побудови діаграм.

obj_ — префікс назв об’єктів у проєкті «GameMaker Studio 2». Наприклад, obj_player_controller, obj_npc_parent, obj_enemy_parent.

scr_ — префікс назв скриптів у проєкті «GameMaker Studio 2». Наприклад, scr_save_game() або scr_load_game().

ВСТУП

Комп’ютерні ігри можна використовувати не тільки для розваг. Вони підходять і для навчальних задач, бо користувач не просто читає інформацію, а діє всередині змодельованої ситуації. Він рухається, обирає варіанти, перевіряє предмети, виконує завдання і бачить наслідки своїх рішень [15, 19].

Для такої роботи підходить формат 2D RPG-гри. У ній гравець керує персонажем, досліджує локації, спілкується з неігровими персонажами, бере предмети, виконує квести та проходить бойові епізоди. Ці механіки пов’язані між собою. Наприклад, розмова може відкрити завдання, предмет може знадобитися для його завершення, а перемога над ворогом може змінити подальший стан гри [12].

Темою бакалаврської кваліфікаційної роботи є програмне забезпечення піксельної 2D RPG-гри з елементами конструктора. У грі користувач проходить невелику сюжетну історію через дослідження світу, діалоги, квести, інвентар, бойову систему, читабельні об’єкти, катсцени та збереження прогресу.

Актуальність роботи пов’язана з потребою у простих інтерактивних засобах навчання. Не кожен такий засіб має бути великою освітньою платформою. Невелика гра також може виконувати навчальну роль, якщо вона ставить перед користувачем зрозумілу задачу, вимагає уважності та дає результат після виконаних дій.

Основна складність такої гри полягає не тільки у створенні локацій або персонажів. Потрібно правильно організувати взаємодію систем. Рух персонажа, NPC, діалоги, квести, предмети, інвентар, вороги, переходи між кімнатами та збереження прогресу не можуть працювати як повністю окремі частини. Якщо не передбачити спільну структуру, додавання нового квесту, предмета або персонажа швидко ускладнює проєкт.

Окремі положення цієї роботи були попередньо висвітлені автором у тезах на тему «Програмне забезпечення піксельної 2D RPG-гри з елементами

конструктора». У них було коротко розглянуто актуальність теми, основні ігрові механіки, використання 2D RPG-формату для інтерактивної взаємодії та загальну ідею програмного продукту.

Метою роботи є розробка програмного забезпечення піксельної 2D RPG-гри з елементами конструктора, яка забезпечує інтерактивну взаємодію користувача з ігровим світом і сприяє розвитку логічного мислення, уважності та навичок прийняття рішень.

Для досягнення мети потрібно виконати такі завдання:

- проаналізувати особливості 2D RPG-ігор як програмних систем;
- визначити основні механіки, потрібні для навчальної RPG-гри;
- розглянути існуючі ігрові рішення зі схожими елементами взаємодії;
- обґрунтувати вибір засобів розробки;
- визначити функціональні та нефункціональні вимоги до гри;
- спроєктувати загальну архітектуру програмного продукту;
- розробити об’єктно-орієнтовану структуру гри;
- спроєктувати системи діалогів, квестів, інвентаря, бою, переходів між кімнатами та збереження прогресу;
- реалізувати основні ігрові механіки та інтерфейс користувача;
- провести тестування основних сценаріїв роботи гри.

Об’єктом роботи є процес розробки програмного забезпечення 2D RPG-гри.

Предметом роботи є архітектура та програмна реалізація ігрових систем, які забезпечують керування персонажем, взаємодію з NPC, діалоги, квести, інвентар, бойову систему, переходи між локаціями, збереження прогресу та інтерфейс користувача.

Практичне значення роботи полягає у створенні працездатного навчального ігрового застосунку. Його можна використовувати як приклад побудови 2D RPG-гри, у якій окремі системи не існують ізольовано, а працюють разом через спільну структуру. Такий проєкт можна доповнювати новими

локаціями, персонажами, предметами, ворогами, квестами та сюжетними подіями без повного переписування основної логіки.

У результаті виконання роботи розроблено програмний продукт, який поєднує навчальну спрямованість, просту взаємодію з користувачем і базові механіки 2D RPG-гри. Гра може використовуватися як допоміжний інтерактивний інструмент і як основа для подальшого розвитку ігрового проєкту.

Бакалаврська кваліфікаційна робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел і додатків.

У першому розділі розглянуто предметну область, проаналізовано існуючі ігрові рішення, визначено функціональні та нефункціональні вимоги і сформульовано задачу розробки.

У другому розділі виконано проєктування програмного забезпечення. Описано загальну архітектуру гри, об’єктно-орієнтовану структуру, логіку переходів між локаціями, системи NPC, діалогів, квестів, інвентаря, бойової взаємодії та збереження прогресу.

У третьому розділі описано реалізацію програмного продукту. Розглянуто використані інструменти, контролери гри, керування персонажем, роботу NPC, діалогів, квестів, предметів, інвентаря, бойової системи, переходів між кімнатами та збереження прогресу.

У четвертому розділі наведено рекомендації щодо впровадження та експлуатації системи. Описано тестування основних сценаріїв, перевірку інтерфейсу та ігрових станів, вимоги до апаратного і програмного забезпечення, склад інсталяційного пакету та перспективи розвитку програмного продукту.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз предметної області

Предметна область цієї роботи пов’язана з розробкою 2D RPG-гри. Такий тип гри не обмежується тільки переміщенням персонажа по локації. У ньому поєднуються кілька систем: дослідження світу, діалоги, квести, предмети, інвентар, бойова взаємодія, переходи між локаціями та збереження прогресу [12].

2D RPG зручна для навчального застосунку, бо гравець не просто спостерігає за подіями. Він сам виконує дії. Наприклад, читає репліки персонажів, шукає потрібні предмети, перевіряє умови завдання, приймає рішення і бачить результат. Через це гра може працювати як просте інтерактивне середовище, у якому користувач діє послідовно і краще розуміє зв’язок між власними діями та змінами у світі гри [15, 19].

Основою RPG-гри є взаємодія з ігровим світом. Гравець переміщується між локаціями, знаходить об’єкти, спілкується з неігровими персонажами та виконує завдання. Кожна така дія має впливати на подальше проходження. Якщо користувач підібрав предмет, він має з’явитися в інвентарі. Якщо поговорив з персонажем, може змінитися стан завдання. Якщо переміг ворога, гра повинна зафіксувати цей результат [2, 20].

Для такої гри важлива система станів. Стан показує, на якому етапі перебуває певний об’єкт або подія. Це може бути стан квесту, NPC, предмета, ворога або переходу між локаціями. Без цього гра не зможе правильно реагувати на дії користувача. Наприклад, персонаж не повинен однаково говорити до початку завдання, під час його виконання і після завершення.

Окрему роль мають квести. Вони задають гравцю конкретну мету і пов’язують між собою різні механіки. Завдання може вимагати знайти предмет,

поговорити з персонажем, перемогти ворога або перейти до іншої локації. Завдяки цьому користувач не просто рухається по карті, а виконує послідовність дій, які мають логічний результат [11].

Інвентар потрібний для збереження предметів, які гравець отримує під час проходження. У RPG-грі предмети часто не є декоративними. Вони можуть бути частиною квесту, нагородою або засобом для відновлення стану персонажа. Тому система інвентаря має бути пов’язана з квестами, предметами на локаціях і збереженням прогресу.

Бойова система додає до гри активну взаємодію. Гравець має реагувати на ворогів, атакувати, отримувати шкоду і стежити за станом персонажа. При цьому бій також не існує окремо. Перемога над ворогом може впливати на квест, відкривати наступну подію або змінювати стан локації.

Ще одна важлива частина — переходи між локаціями. Ігровий світ у 2D RPG зазвичай складається з окремих кімнат або сцен. Гравець переходить між ними, а гра повинна правильно переносити персонажа в потрібну точку. Деякі переходи можуть бути доступні не одразу. Наприклад, шлях може відкриватися тільки після виконання завдання або певної сюжетної дії.

Для навчальної гри також важливий зрозумілий інтерфейс. Користувач має бачити, коли можна взаємодіяти з об’єктом, які предмети є в інвентарі, що відбувається під час діалогу і який стан має персонаж. Інтерфейс не повинен заважати проходженню, але має давати достатньо інформації для наступної дії [11].

Особливість цієї предметної області полягає в тому, що навіть невелика 2D RPG-гра складається з багатьох пов’язаних частин. Якщо ці частини створювати окремо і без спільної логіки, проєкт швидко стане складним для підтримки. Тому під час розробки потрібно одразу враховувати архітектуру, зв’язки між системами та можливість подальшого розширення гри.

У цій роботі 2D RPG розглядається як програмний продукт, у якому ігрові механіки працюють разом. Діалоги пов’язані з квестами, квести — з предметами

та ворогами, інвентар — зі станом гравця, а переходи між локаціями — із сюжетним прогресом. Саме така структура дозволяє створити гру, де користувач не просто проходить набір екранів, а взаємодіє з системою, яка реагує на його дії.

1.2 Аналіз існуючих ігрових рішень

Для аналізу було обрано чотири ігрові рішення: Undertale [21], Kindergarten [22], Hello Charlotte [23] та Stardew Valley [24]. Вони відрізняються жанром і масштабом, але мають окремі механіки, близькі до теми роботи: діалоги, взаємодію з NPC, завдання, предмети, дослідження локацій і залежність подій від дій гравця.

Мета аналізу — визначити, які рішення можна врахувати під час розробки власної 2D RPG-гри. Порівняння не зводиться до графіки або популярності. Важливішим є те, як у цих іграх організовано взаємодію між користувачем, персонажами, предметами та ігровими подіями [2, 11].

Undertale є прикладом 2D RPG, у якій важливу роль мають діалоги, бойові ситуації та вибір дій гравця [21]. Гравець спілкується з персонажами, проходить бойові ситуації та приймає рішення, які впливають на подальше проходження. Для розроблюваної гри важливим є саме використання діалогу як способу передавання інформації та зміни ставлення персонажів до гравця.

Для аналізу Undertale використано приклад діалогового вікна, оскільки діалоги є однією з основних систем і в розроблюваній грі.

Рис. 1.2.1 Діалогове вікно гри Undertale

На рис. 1.2.1 показано спосіб подання реплік персонажа через окреме діалогове вікно. Для цієї роботи важливим є те, що текст не існує окремо від ігрової події, а підтримує взаємодію між гравцем і персонажем.

Kindergarten не є класичною RPG, але добре показує причинно-наслідкову логіку та залежність подій від послідовності дій гравця [22]. Гравець взаємодіє з персонажами, обирає дії та поступово розуміє, як вони впливають на подальші події. Неправильний порядок дій може змінити результат проходження.

Для розроблюваної гри Kindergarten корисна як приклад побудови завдань, де результат залежить не тільки від факту виконання дії, а й від її місця у загальній послідовності.

Рис. 1.2.2 Взаємодія з персонажем у грі Kindergarten

На рис. 1.2.2 показано сцену взаємодії гравця з персонажем. У межах аналізу важливим є зв’язок між вибором дії, діалогом і подальшим розвитком подій.

Hello Charlotte можна розглядати як приклад сюжетної гри з акцентом на діалоги, дослідження простору та текстові сцени. У ній інформація відкривається поступово: через кімнати, об’єкти, репліки персонажів і зміну сцен [23].

Для власного проєкту це корисно тим, що частину сюжету також можна подавати не прямим поясненням, а через взаємодію з локацією, записками, NPC та катсценами.

Рис. 1.2.3 Дослідження кімнати в грі Hello Charlotte

На рис. 1.2.3 показано фрагмент ігрової кімнати. Для аналізу важливе використання простору як частини сюжетної подачі: гравець отримує інформацію не тільки з діалогів, а й через об’єкти в локації.

Stardew Valley має більший масштаб, ніж розроблювана гра, тому її не потрібно порівнювати напряму, але вона корисна як приклад системної взаємодії світу, NPC, предметів і прогресу гравця [24].

Для цієї роботи важливий не фермерський напрям гри, а те, що окремі системи працюють разом. Діалог із NPC може бути пов’язаний із завданням, предметами, подальшими діями та розвитком ігрового світу.

Рис. 1.2.4 Діалог з NPC у грі Stardew Valley

На рис. 1.2.4 показано діалог гравця з неігровим персонажем. Такий приклад важливий для аналізу NPC як активної частини ігрової системи, а не просто декоративного об’єкта.

Для узагальнення результатів аналізу наведено табл. 1.

Таблиця 1

Порівняння існуючих ігрових рішень

Гра	**Жанрові особливості**	**Корисні рішення**	**Обмеження відносно теми роботи**
Undertale	2D RPG з діалогами, боями, вибором дій і сюжетними наслідками	Подання інформації через діалоги, реакція персонажів на дії гравця	Немає акценту на створенні нових ігрових об’єктів через спільну архітектурну основу
Kindergarten	Puzzle-adventure з подіями, що залежать від порядку дій	Причинно-наслідкова логіка, залежність результату від вибору користувача	Гра не є класичною RPG; менше уваги до розвитку персонажа та бойової системи
Hello Charlotte	Сюжетна гра з діалогами, текстовими сценами та дослідженням простору	Подання історії через локації, об’єкти, персонажів і короткі текстові взаємодії	Основний акцент зроблено на сюжеті, а не на розширюваній RPG-архітектурі
Stardew Valley	RPG/simulation з відкритим світом, NPC, предметами, прогресом і завданнями	Взаємодія багатьох систем: NPC, предметів, локацій, завдань і розвитку персонажа	Значно більший масштаб; фермерські механіки не відповідають напряму розроблюваної гри

Після аналізу можна виділити кілька рішень, корисних для власного проєкту. Від Undertale важливо взяти роль діалогу як частини проходження. Kindergarten показує залежність подій від послідовності дій. Hello Charlotte демонструє подання історії через кімнати, об’єкти та текстові сцени. Stardew Valley є прикладом того, як NPC, предмети, завдання і прогрес можуть працювати як пов’язана система [2, 11].

Розроблювана гра не копіює ці рішення. Вона використовує схожі принципи у меншому форматі: діалоги з NPC, квести, предмети, бойові епізоди, дослідження локацій і збереження стану проходження. Окремою відмінністю є елементи конструктора. Вони не винесені в окремий редактор для користувача, а закладені у структуру гри: нові NPC, предмети, вороги, квести та інші елементи.

1.3 Визначення функціональних і нефункціональних вимог

Після аналізу предметної області та існуючих ігрових рішень потрібно визначити вимоги до розроблюваного програмного продукту. Вимоги показують, які дії має підтримувати гра і які умови потрібно врахувати під час її створення [11].

Функціональні вимоги описують конкретні можливості гри. У цьому проєкті вони пов’язані з основними діями гравця: керуванням персонажем, взаємодією з NPC, діалогами, завданнями, предметами, інвентарем, бойовою системою та переходами між локаціями.

Загальну взаємодію користувача з основними системами гри подано на рис. 1.3.1.

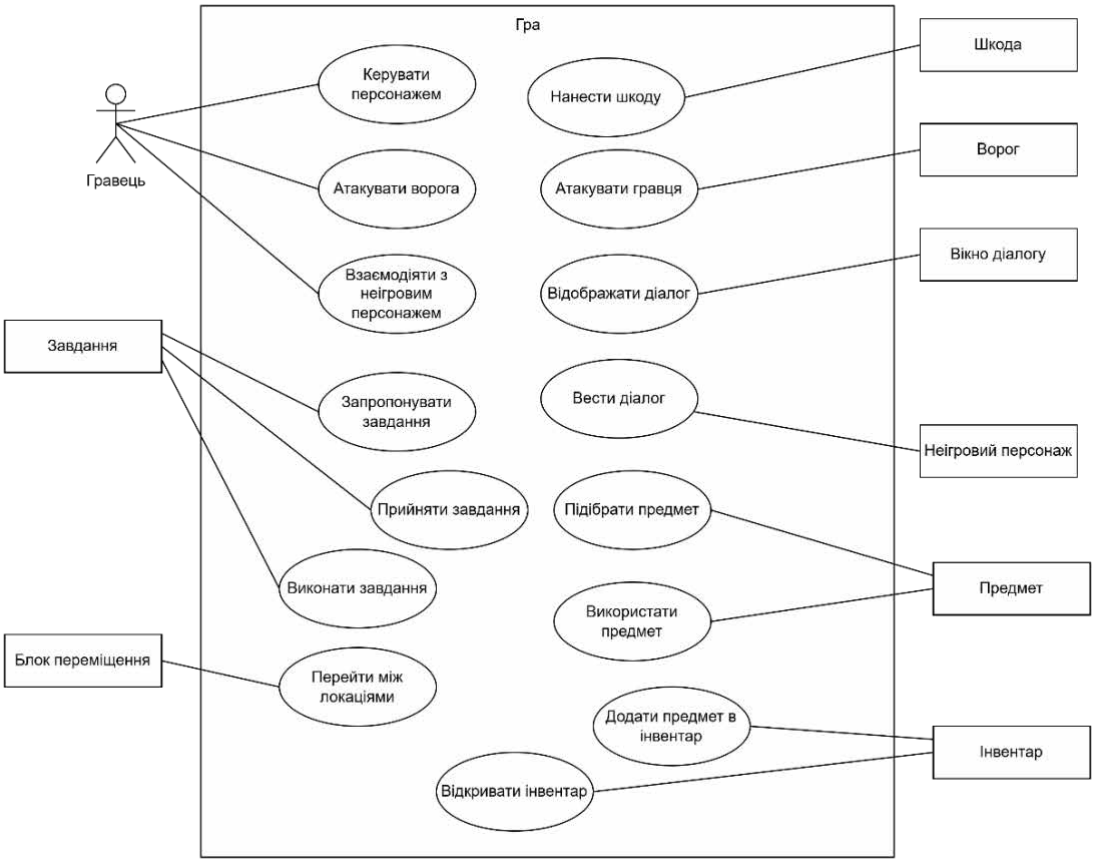


Рис. 1.3.1 Діаграма прецедентів, що ілюструє взаємодію гравця з основними системами гри

На рис. 1.3.1 показано основні дії, які може виконувати гравець, і частини гри, з якими ці дії пов’язані. Гравець керує персонажем, атакує ворога, взаємодіє з неігровим персонажем, приймає та виконує завдання, підбирає предмети, відкриває інвентар і переходить між локаціями.

Окремо на діаграмі показано системи, які реагують на ці дії. Ворог може атакувати гравця, система шкоди обробляє нанесення ушкодження, вікно діалогу відображає репліки, NPC веде діалог і може запропонувати завдання. Предмет пов’язаний із підбором, використанням і додаванням до інвентаря. Блок переміщення відповідає за перехід між локаціями.

На основі цієї діаграми можна сформувати основні функціональні вимоги до гри:

- гра має підтримувати керування персонажем;
- гравець повинен мати можливість атакувати ворога;

- ворог повинен мати можливість атакувати гравця;
- система має обробляти нанесення шкоди;
- гравець повинен мати можливість взаємодіяти з NPC;
- NPC має запускати діалогове вікно;
- діалогова система повинна відображати репліки персонажів;
- NPC може запропонувати гравцю завдання;
- гравець повинен мати можливість прийняти завдання;
- гра має підтримувати виконання завдань;
- гравець повинен мати можливість підібрати предмет;
- підібраний предмет має додаватися до інвентаря;
- інвентар повинен відкриватися під час гри;
- предмети з інвентаря мають використовуватися, якщо це передбачено їх логікою;
- гра має підтримувати перехід між локаціями через блок переміщення;
- прогрес гри має зберігатися і завантажуватися для продовження проходження.

Остання вимога не показана на поточній діаграмі прецедентів, але вона потрібна для повноцінної RPG-гри. Без збереження прогресу користувач не зможе повернутися до вже виконаних завдань, зібраних предметів, поточної локації та зміненого стану світу. Тому цю вимогу потрібно або додати на діаграму окремим прецедентом, або залишити в тексті як окрему функціональну вимогу.

Нефункціональні вимоги описують не окремі дії, а якість роботи програмного продукту. Для цієї гри важливо, щоб взаємодія була зрозумілою, а основні системи працювали стабільно.

До нефункціональних вимог належать:

- інтерфейс має бути зрозумілим для користувача;
- підказки взаємодії повинні чітко показувати, коли можна говорити з NPC, підібрати предмет або перейти до іншої локації;

- гра має стабільно працювати під час переходів між локаціями, діалогів, бою та відкриття інвентаря;
- текст у діалогах, інвентарі та повідомленнях має підтримувати українську мову;
- після завантаження прогресу стан гри має відновлюватися коректно;
- структура гри повинна дозволяти додавати нових NPC, ворогів, предмети, завдання та локації без повного переписування основної логіки;
- основні дії гравця мають виконуватися без зайвих затримок, щоб керування не відчувалося повільним.

Визначені вимоги потрібні для подальшого проєктування. На їх основі можна переходити до побудови архітектури програмного продукту, об’єктної структури гри, систем діалогів, квестів, інвентаря, бойової взаємодії, переходів між локаціями та збереження прогресу.

1.4 Постановка задачі

На основі аналізу предметної області, існуючих ігрових рішень та вимог потрібно розробити програмне забезпечення піксельної 2D RPG-гри з елементами конструктора. Гра має поєднувати дослідження локацій, діалоги з NPC, виконання завдань, роботу з предметами, інвентар, бойову взаємодію, переходи між локаціями та збереження прогресу [12, 15].

Основна задача полягає не лише у створенні ігрового процесу. Потрібно спроєктувати таку структуру гри, у якій окремі системи не працюють ізольовано. Діалоги мають бути пов’язані з квестами, предмети — з інвентарем і завданнями, вороги — з бойовою системою та прогресом проходження, а переходи між локаціями — зі станом гри.

Окремо потрібно передбачити елементи конструктора. У цій роботі вони не розглядаються як окремий редактор рівнів для користувача. Їхня суть полягає у внутрішній структурі гри: нові NPC, предмети, вороги, квести, точки

збереження та інші об’єкти мають створюватися на основі спільної логіки. Це потрібно для зменшення дублювання і спрощення подальшого розширення проєкту.

Програмний продукт повинен забезпечувати такі можливості:

- керування персонажем у межах ігрових локацій;
- взаємодію з неігровими персонажами;
- відображення діалогів;
- прийняття та виконання завдань;
- підбір предметів;
- додавання предметів до інвентаря;
- використання предметів, якщо це передбачено їх логікою;
- бойову взаємодію з ворогами;
- нанесення та отримання шкоди;
- перехід між локаціями;
- збереження і завантаження прогресу;
- відображення основних елементів інтерфейсу.

Під час розробки потрібно врахувати, що гра має бути зрозумілою для користувача. Гравець повинен бачити, коли можна взаємодіяти з NPC, підібрати предмет, перейти до іншої локації або відкрити інвентар. Інтерфейс має допомагати проходженню, а не відволікати від нього.

Також потрібно забезпечити коректне збереження стану гри. Після завантаження мають відновлюватися не тільки позиція персонажа, а й виконані завдання, предмети в інвентарі, переможені вороги, поточна локація та інші важливі зміни. Без цього проходження буде непослідовним.

Результатом виконання задачі має стати працездатна 2D RPG-гра, у якій користувач може проходити сюжет, взаємодіяти з персонажами, виконувати завдання, збирати предмети, брати участь у боях і продовжувати гру після збереження. Структура проєкту повинна залишатися придатною для подальшого доповнення новими ігровими елементами.

Висновки до розділу 1

У першому розділі було розглянуто предметну область розробки 2D RPG-гри. Визначено, що така гра складається з кількох пов’язаних систем: переміщення персонажа, діалогів, квестів, предметів, інвентаря, бойової взаємодії, переходів між локаціями та збереження прогресу.

Під час аналізу існуючих ігрових рішень було розглянуто Undertale, Kindergarten, Hello Charlotte та Stardew Valley. У цих іграх виділено рішення, корисні для власного проєкту: подання інформації через діалоги, залежність подій від дій гравця, використання простору для сюжетної подачі та взаємодію кількох ігрових систем.

Також було визначено функціональні та нефункціональні вимоги до програмного продукту. Функціональні вимоги описують основні можливості гри, а нефункціональні — умови її зручної та стабільної роботи.

На основі проведеного аналізу сформульовано задачу розробки. Потрібно створити піксельну 2D RPG-гру з елементами конструктора, у якій основні системи працюють разом і можуть бути розширені новими об’єктами без повного переписування логіки.

2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Загальна архітектура програмного продукту

Під час проєктування програмний продукт потрібно розглядати як набір пов’язаних компонентів. Для 2D RPG-гри це важливо, бо ігровий процес складається не з однієї дії, а з кількох систем. Персонаж рухається, взаємодіє з NPC, бере участь у боях, отримує предмети, виконує завдання, переходить між локаціями та зберігає прогрес [1, 6, 9].

Загальну структуру основних компонентів програмного продукту подано на рис. 2.1.1.

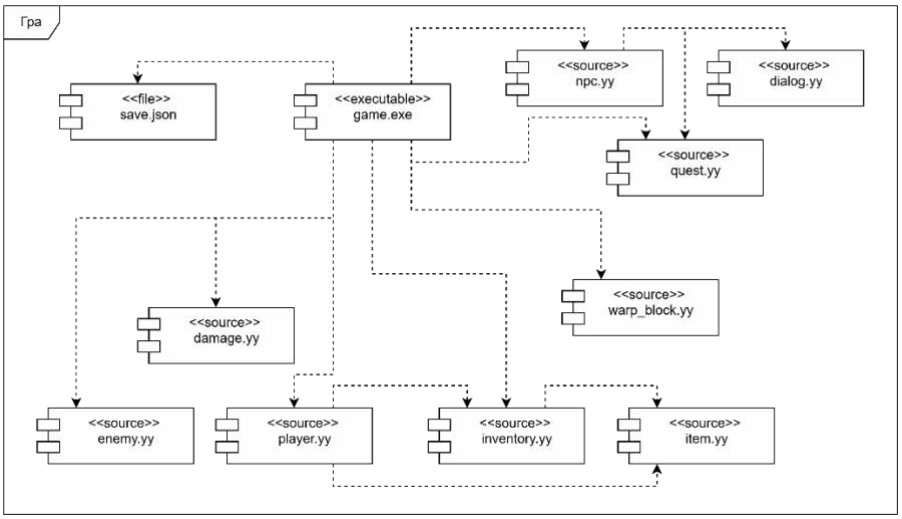


Рис. 2.1.1 Діаграма компонентів, що ілюструє загальну структуру програмного продукту

На рис. 2.1.1 показано виконуваний файл «game.exe», файл збереження «save.json» та основні компоненти проєкту, які відповідають за окремі частини гри. Така схема не описує всі об’єкти детально. Її задача — показати, з яких частин складається програмний продукт і як ці частини пов’язані між собою на загальному рівні.

Центральним компонентом є «game.exe». Він представляє зібрану гру, яка запускається користувачем. Саме через нього виконуються всі ігрові системи:

керування персонажем, діалоги, квести, інвентар, бойова логіка, предмети, переходи між кімнатами та робота зі збереженням.

Окремо на діаграмі показано файл «save.json». Він використовується для збереження прогресу. До нього мають записуватися дані, потрібні для продовження гри: поточний стан проходження, локація, інвентар, виконані дії та інші важливі зміни. Через це файл збереження не можна розглядати як другорядний елемент. Він напряму пов’язаний із логікою проходження.

Компонент «player.yy» відповідає за гравця. Він пов’язаний із системами, які потрібні для основної взаємодії: інвентарем, предметами та бойовою частиною. Персонаж не існує окремо від інших систем, бо саме через нього користувач підбирає предмети, атакує ворогів, переходить між локаціями та запускає більшість ігрових подій.

Компонент «enemy.yy» відповідає за ворогів. Його робота пов’язана з бойовою системою, оскільки вороги мають реагувати на гравця, отримувати шкоду та впливати на проходження. Для обробки ударів використовується компонент «damage.yy». Він потрібний як окрема частина, щоб не змішувати логіку персонажа, ворога і нанесення шкоди в одному місці.

Компонент «npc.yy» відповідає за неігрових персонажів. Він пов’язаний із «dialog.yy» та «quest.yy», бо NPC у грі не просто стоять на локаціях. Вони запускають діалоги, можуть пропонувати завдання і реагують на поточний стан проходження. Через це NPC є зв’язувальним компонентом між сюжетною частиною, діалогами та квестами.

Компонент «dialog.yy» відповідає за діалогову систему. Він потрібний для виведення реплік, показу тексту та організації розмови між гравцем і персонажем. Винесення діалогів в окремий компонент дозволяє не дублювати логіку показу тексту в кожному NPC.

Компонент «quest.yy» відповідає за завдання. Він зберігає логіку прийняття, виконання та завершення квестів. Квестова система пов’язана з NPC,

предметами, ворогами і загальним станом проходження. Наприклад, завдання може вимагати підібрати предмет або перемогти певного ворога.

Компонент «item.yy» відповідає за предмети. Він пов’язаний з інвентарем і гравцем, бо предмети можуть бути підібрані, додані до списку речей або використані під час гри. Предмети також можуть впливати на квести, якщо вони потрібні для виконання завдання.

Компонент «inventory.yy» відповідає за інвентар. Він зберігає отримані предмети та дає користувачу можливість переглядати їх під час проходження. Інвентар пов’язаний із предметами напряму, бо саме туди потрапляють об’єкти після підбору.

Компонент «warp_block.yy» відповідає за переходи між локаціями. Він потрібний для перенесення гравця з однієї кімнати в іншу. Такі переходи можуть залежати від стану гри, тому цей компонент важливий не тільки для переміщення, а й для побудови послідовного проходження.

Загальна архітектура будується так, щоб кожен компонент мав свою зону відповідальності. Гравець відповідає за основну взаємодію, NPC — за розмови та сюжетні події, діалогова система — за текст, квести — за завдання, інвентар і предмети — за роботу з речами, вороги та шкода — за бойову частину, а блок переходу — за переміщення між кімнатами.

Такий поділ спрощує подальше проєктування. Якщо всі ці частини змішати в одному об’єкті, будь-яка зміна швидко зачіпатиме інші системи. Окремі компоненти дозволяють легше розширювати гру: додавати нових NPC, предмети, ворогів, завдання або локації без повного переписування основної логіки.

2.2 Об’єктно-орієнтована структура гри

Для проєктування внутрішньої структури гри використано об’єктно-орієнтований підхід. Його суть полягає в тому, що основні елементи гри

розглядаються як окремі об’єкти зі своїми даними та діями. Такий підхід зручний для RPG-гри, бо в ній є багато сутностей, які постійно взаємодіють між собою: гравець, NPC, вороги, предмети, інвентар, квести, діалоги та переходи між локаціями [6].

На рис. 2.2.1 подано діаграму класів, яка показує основні об’єкти гри та зв’язки між ними.

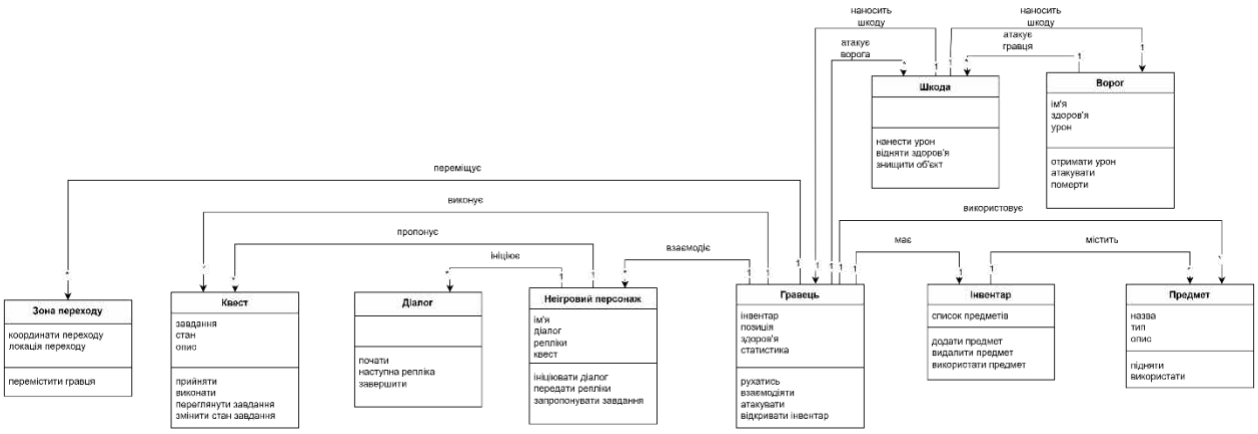


Рис. 2.2.1 Діаграма класів, що ілюструє зв’язки між основними об’єктами гри

На рис. 2.2.1 центральним об’єктом є «Гравець». Він має позицію, здоров’я, статистику та інвентар. Також гравець виконує основні дії: рухається, взаємодіє з іншими об’єктами, атакує та відкриває інвентар. Через це більшість інших класів прямо або опосередковано пов’язані саме з ним.

Клас «Неігровий персонаж» описує NPC, з якими гравець може взаємодіяти під час проходження. Такий об’єкт має ім’я, діалог, репліки та може бути пов’язаний із квестом. Його основні дії — ініціювати діалог, передавати репліки та пропонувати завдання. Це дозволяє використовувати NPC не тільки як декоративний елемент, а як частину сюжетної і квестової логіки.

Клас «Діалог» відповідає за роботу з репліками. Він містить дії початку розмови, переходу до наступної репліки та завершення діалогу. Винесення діалогу в окремий клас потрібне для того, щоб NPC не зберігали всю логіку показу тексту всередині себе. NPC лише запускає діалог і передає потрібні репліки.

Клас «Квест» описує завдання, яке може отримати гравець. Він містить назву або текст завдання, стан і опис. Основні дії квесту — прийняти, виконати, переглянути завдання та змінити його стан. Така структура потрібна, бо завдання в RPG-грі не є простим текстом. Воно має власний стан і може змінюватися після дій гравця.

Клас «Інвентар» зберігає список предметів. Він дає можливість додавати предмет, видаляти його та використовувати. Інвентар пов’язаний із гравцем і предметами: гравець має один інвентар, а сам інвентар може містити багато предметів. Такий зв’язок дозволяє відокремити логіку зберігання речей від логіки персонажа.

Клас «Предмет» описує об’єкти, які гравець може підняти або використати. Предмет має назву, тип і опис. Його основні дії — підняти та використати. У грі предмет може бути частиною квесту, нагородою або засобом для відновлення стану персонажа, тому він пов’язаний не тільки з інвентарем, а й із загальним проходженням.

Клас «Ворог» описує противників. Він має ім’я, здоров’я та урон. Основні дії ворога — отримати урон, атакувати та померти. Ворог пов’язаний із класом Шкода, бо бойова взаємодія потребує окремої обробки нанесення урону. Це дозволяє не змішувати рух ворога, його стан і саму логіку нанесення шкоди.

Клас «Шкода» відповідає за бойовий вплив на об’єкти. Він має дії нанесення урону, віднімання здоров’я та знищення об’єкта після виконання своєї задачі. На діаграмі видно, що шкода може бути пов’язана і з атакою гравця, і з атакою ворога. Тому цей клас потрібен як проміжний елемент між атакуючим об’єктом і тим, хто отримує урон.

Клас «Зона переходу» відповідає за переміщення між локаціями. Він містить координати переходу та локацію, куди потрібно перенести гравця. Основна дія цього класу — перемістити гравця. Такий об’єкт потрібний для поділу ігрового світу на окремі кімнати або сцени.

Зв’язки на діаграмі показують, що основні системи гри не працюють окремо. Гравець взаємодіє з NPC, квестами, предметами, інвентарем, ворогами та зонами переходу. NPC може запускати діалог і пропонувати квест. Інвентар містить предмети. Ворог і гравець можуть бути пов’язані через об’єкт шкоди. Зона переходу переносить гравця до іншої локації.

Така структура потрібна для подальшої реалізації гри. Вона дозволяє розділити відповідальність між об’єктами: гравець відповідає за основні дії користувача, NPC — за взаємодію та діалоги, квест — за стан завдання, інвентар — за предмети, ворог — за бойову логіку, а зона переходу — за переміщення між локаціями. Завдяки цьому нові ігрові елементи можна додавати без змішування всієї логіки в одному об’єкті.

2.3 Проєктування структури ігрового світу та переходів між локаціями

Ігровий світ поділяється на окремі локації. Гравець не переміщується по одному великому безперервному простору, а переходить між кімнатами. Для цього потрібен окремий механізм переходу, який визначає, у яку локацію треба перенести персонажа і в якій точці він має з’явитися.

У структурі гри за перехід відповідає блок переміщення. Він не є декоративним об’єктом на мапі. Його задача — відстежити, чи гравець знаходиться поруч, прийняти запит на перехід і виконати перенесення в задану локацію. Такий підхід потрібний, щоб кожен перехід мав конкретну ціль, а гравець після зміни кімнати з’являвся не випадково, а в потрібній позиції.

Логіку переходу між локаціями подано на рис. 2.3.1.

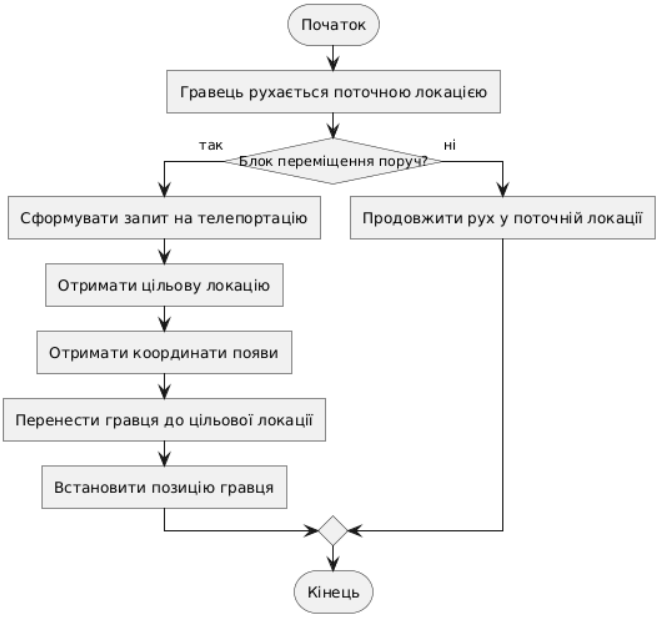


Рис. 2.3.1 Блок-схема, що ілюструє перехід гравця між локаціями

На рис. 2.3.1 показано послідовність роботи переходу. Спочатку гравець рухається поточною локацією. Після цього система перевіряє, чи знаходиться поруч блок переміщення. Якщо такого блоку поруч немає, гра продовжує виконувати звичайну логіку руху в поточній кімнаті.

Якщо блок переміщення поруч, формується запит на телепортацію. Далі визначається цільова локація та координати появи персонажа. Після цього гравець переноситься до іншої кімнати, а його позиція встановлюється у заздалегідь визначеній точці.

Такий алгоритм робить переходи між локаціями керованими. Наприклад, якщо персонаж виходить з будинку, він має з’явитися біля входу зовні. Якщо повертається назад, то повинен опинитися біля дверей усередині. Це зменшує плутанину для користувача і робить переміщення між кімнатами логічним.

У загальній структурі гри блок переміщення пов’язаний з гравцем і локаціями. Гравець ініціює перехід через наближення до відповідної зони, а сам блок переміщення задає, куди саме потрібно перенести персонажа. Завдяки цьому локації можна розглядати як окремі частини світу, які з’єднуються між собою через спеціальні точки переходу.

2.4 Проєктування системи NPC та діалогів

NPC у грі потрібні не тільки для візуального заповнення локацій. Вони беруть участь у проходженні: можуть передавати інформацію, запускати діалог, пропонувати завдання або змінювати свою поведінку після певних дій гравця. Тому систему NPC потрібно проєктувати разом із діалоговою системою [11, 17].

Основна взаємодія починається з перевірки відстані між гравцем і неігровим персонажем. Якщо гравець не знаходиться поруч із NPC, підказка не показується і діалог не запускається. Це потрібно, щоб користувач не міг випадково почати розмову з персонажем, який знаходиться далеко або взагалі не бере участі в поточній взаємодії.

Алгоритм взаємодії гравця з NPC подано на рис. 2.4.1.

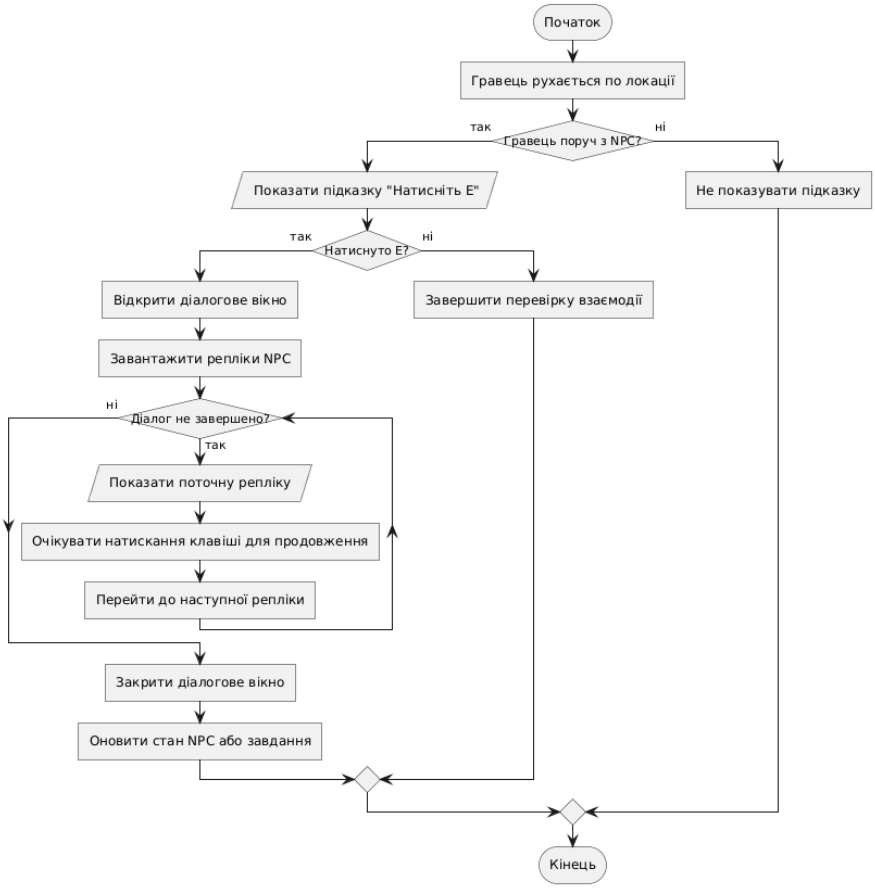


Рис. 2.4.1 Блок-схема, що ілюструє взаємодію гравця з NPC

На рис. 2.4.1 показано послідовність роботи системи NPC та діалогів. Спочатку гравець рухається по локації. Гра перевіряє, чи знаходиться він поруч

із неігровим персонажем. Якщо NPC поруч немає, підказка не виводиться, а система завершує перевірку взаємодії.

Якщо гравець знаходиться поруч із NPC, на екрані з’являється підказка для взаємодії. Вона показує, що з цим персонажем можна почати розмову. Після натискання клавіші взаємодії відкривається діалогове вікно і завантажуються репліки, які належать конкретному NPC.

Діалог працює послідовно. Спочатку показується поточна репліка. Після натискання клавіші для продовження система переходить до наступної репліки. Цей цикл повторюється доти, поки всі репліки не будуть показані.

Після завершення діалогу вікно закривається. Якщо розмова пов’язана із завданням або станом персонажа, система може оновити відповідний стан. Наприклад, після розмови NPC може перейти до іншого етапу діалогу або активувати завдання для гравця.

Такий підхід розділяє відповідальність між NPC і діалоговим вікном. NPC визначає, чи можна почати розмову і які дані потрібно передати. Діалогове вікно відповідає за показ реплік та перехід між ними. Завдяки цьому логіка розмови не змішується з логікою переміщення гравця або іншими системами гри.

Система NPC також пов’язана з квестами. Не кожен діалог має запускати завдання, але така можливість повинна бути передбачена. Якщо персонаж бере участь у квесті, після діалогу може змінитися стан завдання або самого NPC. Це дозволяє робити розмови частиною проходження, а не окремим текстовим елементом.

2.5 Проєктування квестової системи

Квестова система потрібна для організації завдань, які гравець отримує під час проходження. Вона пов’язує між собою NPC, діалоги, предмети, ворогів і стан гри. Завдання не повинно бути просто текстом у діалоговому вікні. Воно має мати власний стан: запропоноване, прийняте, активне або виконане [11].

У розроблюваній грі початок завдання пов’язаний із взаємодією гравця з NPC. Спочатку гравець підходить до неігрового персонажа і запускає діалог. NPC передає потрібні діалогові дані у вікно діалогу. Якщо персонаж має завдання, після початку розмови гравцю може бути показана інформація про нього.

Логіку роботи квестової системи подано на рис. 2.5.1.

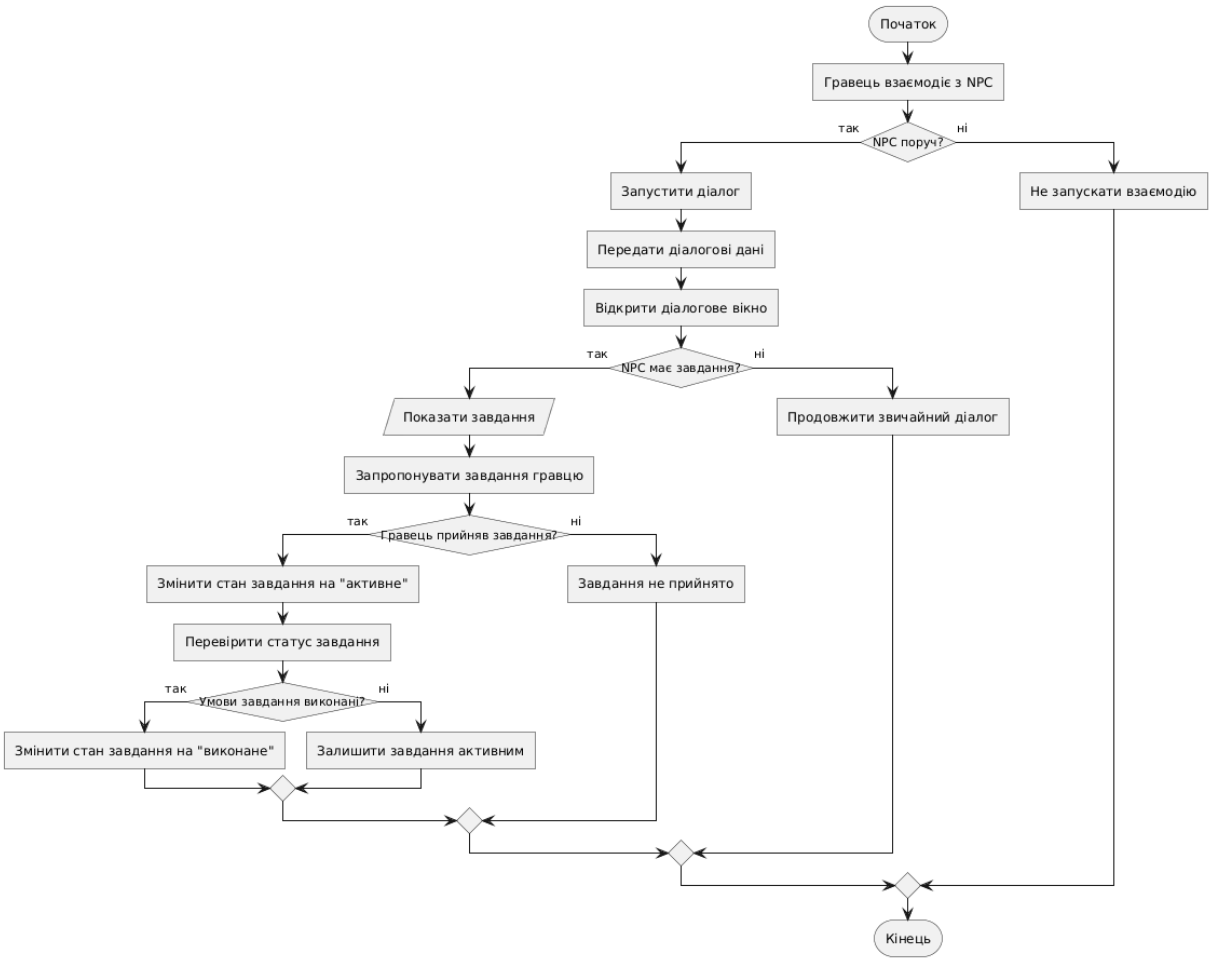


Рис. 2.5.1 Блок-схема, що ілюструє роботу квестової системи

На рис. 2.5.1 показано, що квестова система починає роботу не окремо, а через NPC і діалог. Якщо гравець не знаходиться поруч із NPC, взаємодія не запускається. Якщо NPC поруч, відкривається діалогове вікно, і система перевіряє, чи пов’язаний цей персонаж із завданням.

Якщо NPC має завдання, воно показується гравцю через діалогову взаємодію. Після цього гравець може прийняти завдання. У разі прийняття стан

завдання змінюється на активний. Це потрібно, щоб гра могла надалі перевіряти його виконання і не пропонувала те саме завдання як нове.

Після прийняття завдання система перевіряє його статус. Якщо умови виконані, завдання переходить у стан виконаного. Якщо умови ще не виконані, воно залишається активним. Такий стан дозволяє гравцю повернутися до NPC пізніше, коли потрібні дії вже будуть виконані.

Якщо гравець не приймає завдання, воно не переходить в активний стан. У такому випадку NPC може залишити звичайний діалог або повторно запропонувати завдання пізніше, якщо це передбачено логікою проходження.

Квестова система не працює ізольовано. Вона залежить від діалогів, бо саме через них гравець отримує завдання. Також вона пов’язана з іншими частинами гри: предметами, ворогами та переходами між локаціями. Наприклад, для виконання завдання може бути потрібно знайти предмет, перемогти ворога або перейти до іншої кімнати.

Такий підхід дозволяє зробити завдання частиною загального проходження. Гравець не просто читає текст квесту, а виконує дії, після яких змінюється стан завдання і подальша поведінка NPC.

2.6 Проєктування інвентаря та предметів

Система предметів потрібна для взаємодії гравця з об’єктами, які можна отримати під час проходження. Предмети можуть бути частиною квесту, нагородою або засобом для відновлення стану персонажа. Тому вони не повинні існувати окремо від інвентаря, квестів і загального стану гри [11].

Під час проєктування предмет розглядається як окремий ігровий об’єкт. Він знаходиться в локації, може показувати підказку для взаємодії і після підбору має потрапити до інвентаря. Якщо предмет уже підібраний, він не повинен залишатися на карті як активний об’єкт.

Логіку підбору предмета подано на рис. 2.6.1.

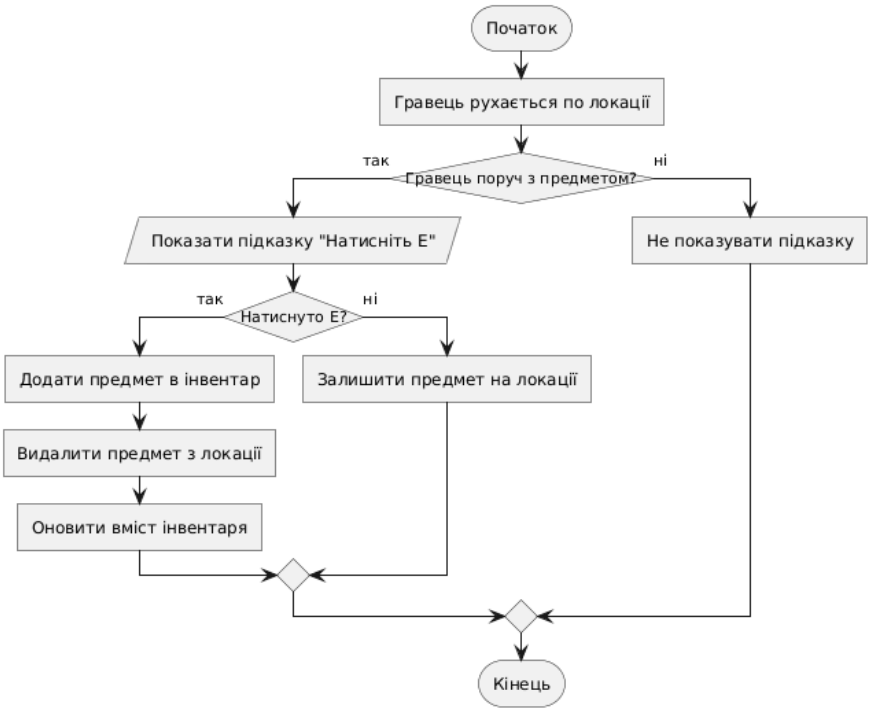


Рис. 2.6.1 Блок-схема, що ілюструє підбір предмета та додавання його до інвентаря

На рис. 2.6.1 показано, що спочатку гравець рухається по локації. Система перевіряє, чи знаходиться він поруч із предметом. Якщо предмета поруч немає, підказка не показується, і взаємодія не запускається.

Якщо гравець підійшов до предмета, на екрані з’являється підказка для взаємодії. Після натискання клавіші предмет додається до інвентаря. Потім він видаляється з локації, щоб користувач не міг підібрати один і той самий об’єкт кілька разів. Після цього оновлюється вміст інвентаря.

Інвентар потрібний для збереження предметів, які вже належать гравцю. Він має показувати список речей і давати можливість вибрати предмет для використання, якщо така дія передбачена його логікою. Не кожен предмет обов’язково має використовуватися одразу. Частина предметів може бути потрібна для квестів або сюжетних подій.

Логіку використання предмета з інвентаря подано на рис. 2.6.2.

Рис. 2.6.2 Блок-схема, що ілюструє використання предмета з інвентаря

На рис. 2.6.2 показано роботу інвентаря після його відкриття. Користувач відкриває вікно інвентаря, переглядає список предметів і обирає потрібний об’єкт. Якщо натиснуто дію використання, система перевіряє, чи можна застосувати цей предмет у поточній ситуації.

Якщо предмет можна використати, його дія застосовується. Після цього кількість предмета зменшується. Якщо кількість стає нульовою, предмет видаляється з інвентаря. Якщо ще залишаються однакові предмети, він продовжує відображатися у списку.

Якщо предмет використати не можна, система не змінює інвентар і показує повідомлення про неможливість використання. Це потрібно, щоб користувач розумів, чому дія не виконалася.

Інвентар пов’язаний з іншими системами гри. Підібраний предмет може впливати на квест, відновлювати стан гравця або використовуватися пізніше в іншій локації. Через це інвентар не є просто списком речей. Він зберігає частину

прогресу проходження і має працювати разом із предметами, квестами та системою збереження.

2.7 Проєктування бойової системи

Бойова система потрібна для взаємодії гравця з ворогами. У цій системі важливо розділити сам факт атаки і процес нанесення шкоди. Гравець або ворог ініціює атаку, а окремий об’єкт шкоди перевіряє, чи є поруч ціль, і зменшує її здоров’я [2, 11].

Такий поділ потрібний, щоб не змішувати логіку руху, стан персонажа і саму обробку удару. Гравець відповідає за початок атаки. Ворог також може ініціювати атаку. Об’єкт шкоди виконує окрему задачу: перевіряє наявність цілі поруч і застосовує урон.

Логіку атаки гравця подано на рис. 2.7.1.

Рис. 2.7.1 Блок-схема, що ілюструє атаку гравця по ворогу

На рис. 2.7.1 показано послідовність атаки з боку гравця. Спочатку гравець ініціює атаку. Після цього створюється об’єкт шкоди, який перевіряє, чи знаходиться ворог поруч із зоною удару.

Якщо ворог потрапляє в зону шкоди, система наносить йому урон і зменшує його здоров’я. Якщо ворога поруч немає, шкода не застосовується. Після завершення атаки об’єкт шкоди має бути знищений або деактивований, щоб удар не спрацьовував постійно.

Логіку атаки ворога подано на рис. 2.7.2.

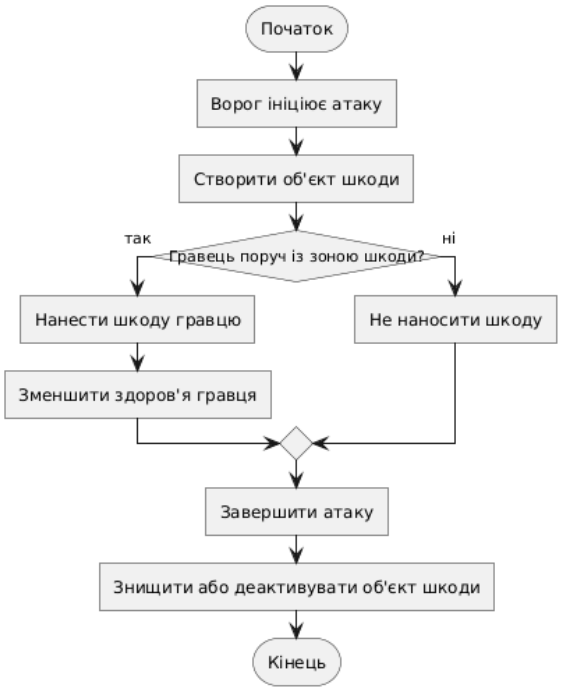


Рис. 2.7.2 Блок-схема, що ілюструє атаку ворога по гравцю

На рис. 2.7.2 показано схожий процес, але ініціатором атаки є ворог. Він створює об’єкт шкоди, після чого система перевіряє, чи знаходиться гравець поруч із зоною атаки. Якщо гравець потрапляє в цю зону, його здоров’я зменшується.

Якщо гравець не перебуває в зоні шкоди, атака не змінює його стан. Це потрібно, щоб удар ворога не спрацьовував без фактичного контакту або наближення до гравця.

У бойовій системі об’єкт шкоди працює як проміжна частина між атакуючим об’єктом і ціллю. Завдяки цьому гравець і ворог не повинні напряму змінювати здоров’я одне одного. Вони тільки запускають атаку, а перевірка попадання і зменшення здоров’я виконуються через окрему логіку.

Такий підхід спрощує подальше розширення бойової системи. Якщо потрібно додати нового ворога або змінити силу атаки, не потрібно повністю переписувати взаємодію між гравцем і всіма противниками. Достатньо зберігати спільний принцип: атака створює зону шкоди, зона перевіряє ціль, після чого змінюється здоров’я відповідного об’єкта.

2.8 Проєктування системи збереження та завантаження прогресу

Система збереження потрібна для того, щоб гравець міг продовжити проходження після виходу з гри. Для RPG-гри недостатньо зберегти тільки координати персонажа. Потрібно також фіксувати стан світу, бо під час проходження змінюються квести, інвентар, NPC, прочитані об’єкти та інші частини гри.

У проєкті збереження має містити дані про гравця, інвентар, неігрових персонажів і завдання. Такі дані потрібні для правильного відновлення гри після завантаження. Якщо частину станів не зберегти, після повернення до гри можуть виникнути помилки в проходженні: предмети з’являться повторно, NPC повернеться до старого діалогу або завдання втратить свій поточний стан [10].

Модель даних, які потрібно зберігати для відновлення прогресу, подано на рис. 2.8.1.

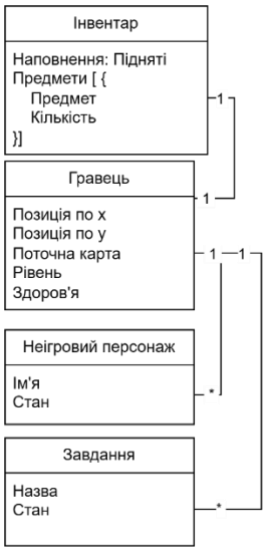


Рис. 2.8.1 Модель даних, що ілюструє структуру збереження прогресу гри

На рис. 2.8.1 показано основні групи даних, які входять до збереження. До блоку Гравець належать позиція по осі x, позиція по осі y, поточна карта, рівень і здоров’я. Ці дані потрібні, щоб після завантаження персонаж з’явився в правильній локації, у правильній точці та з актуальними характеристиками.

Блок Інвентар зберігає підняті предмети. Для кожного предмета важливо фіксувати сам предмет і його кількість. Це дозволяє відновити список речей після завантаження і не втратити предмети, які гравець уже отримав під час проходження.

Блок Неігровий персонаж містить ім’я та стан NPC. Стан потрібний для того, щоб після завантаження персонаж не повертався до початкової поведінки. Наприклад, якщо NPC уже видав завдання або змінив репліки після певної події, гра має відновити саме цей етап.

Блок Завдання містить назву та стан квесту. Стан завдання показує, чи воно ще не почате, активне або вже завершене. Це важливо для послідовного проходження, бо квести пов’язані з NPC, предметами, ворогами та подальшими подіями гри.

Зв’язки між блоками показують, що збереження не є окремим списком випадкових значень. Дані гравця пов’язані з інвентарем, NPC та завданнями. Один гравець може мати багато предметів в інвентарі, взаємодіяти з кількома

NPC і виконувати кілька завдань. Тому структура збереження має підтримувати не тільки одиничні значення, а й набори даних.

Під час завантаження гри ці дані мають відновлюватися разом. Спочатку потрібно повернути гравця в потрібну локацію та встановити його позицію. Потім відновлюється інвентар, стани NPC і стани завдань. Завдяки цьому гра продовжується з того моменту, на якому користувач зупинився.

Така структура збереження потрібна для коректної роботи всієї RPG-системи. Вона дозволяє не втрачати прогрес, не повторювати вже виконані дії та підтримувати зв’язок між квестами, NPC, предметами й поточним станом гравця.

Висновки до розділу 2

У другому розділі було спроєктовано структуру програмного забезпечення 2D RPG-гри. Визначено основні компоненти програмного продукту та показано, як між собою пов’язані гравець, NPC, діалоги, квести, предмети, інвентар, вороги, бойова система, переходи між локаціями та збереження прогресу.

Також було розглянуто об’єктно-орієнтовану структуру гри. Основні елементи подано як окремі об’єкти зі своїми даними та діями. Це дозволяє розділити відповідальність між частинами гри та не змішувати всю логіку в одному об’єкті.

Для ключових систем було побудовано блок-схеми. Окремо описано логіку переходу між локаціями, взаємодію гравця з NPC, роботу квестів, підбір і використання предметів, а також бойову взаємодію між гравцем і ворогом.

Наприкінці розділу було визначено модель даних для збереження прогресу гри. До неї входять дані гравця, інвентар, стани NPC і завдань. Така структура потрібна для коректного відновлення проходження після завантаження гри.

У результаті проєктування було визначено, як мають працювати основні системи гри та як вони повинні взаємодіяти між собою у межах єдиної структури програмного продукту.

3 РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТУ

3.1 Вибір інструментів розробки

Для реалізації програмного продукту було використано середовище розробки «GameMaker Studio 2». Це середовище підходить для створення 2D-ігор, оскільки має вбудовану роботу з кімнатами, об’єктами, спрайтами, подіями, зіткненнями та графічним інтерфейсом. Для цього проєкту важливим було те, що «GameMaker Studio 2» дозволяє будувати гру через систему об’єктів і наслідування. Завдяки цьому в проєкті можна створювати батьківські об’єкти для NPC, ворогів, предметів, квестів і переходів, а потім налаштовувати дочірні об’єкти під конкретні ігрові ситуації [1, 6].

Логіку гри реалізовано мовою «GML». Вона використовується для опису поведінки персонажа, NPC, ворогів, предметів, квестів, переходів між кімнатами, інвентаря, діалогів і збереження прогресу. У межах проєкту через «GML» реалізовано обробку клавіатури, зміну станів гри, перевірку колізій, створення об’єктів шкоди, роботу з глобальними змінними, списками, мапами та JSON-файлом збереження [1, 7, 8, 10].

Для створення та редагування піксельної графіки було використано програмне середовище «Aseprite». У ньому створювалися або редагувалися спрайти персонажів, предметів, ворогів, елементів локацій та інших графічних ресурсів. Використання піксельної графіки відповідає стилю 2D RPG-гри та дозволяє підтримувати єдиний візуальний вигляд проєкту [13, 14, 16].

Для тексту в інтерфейсі гри використано шрифт «HaxrCorp 4089 Cyrillic AltGr». Він підтримує кирилицю, тому підходить для українських діалогів, підказок, меню, інвентаря та інших текстових елементів. Шрифт створено на основі «PIXELA CYR» від «sanyapolecat». Його використання також відповідає піксельному стилю гри, бо текст виглядає узгоджено з графікою локацій і персонажів.

Отже, для розробки гри було обрано набір інструментів, який відповідає задачам проєкту: «GameMaker Studio 2» використано як основне середовище розробки, «GML» — для реалізації ігрової логіки, «Aseprite» — для роботи з піксельною графікою, а шрифт «HaxrCorp 4089 Cyrillic AltGr» — для відображення українського тексту в інтерфейсі гри.

3.2 Реалізація загальної структури проєкту та контролерів гри

Реалізація гри побудована через набір контролерів. Це потрібно для того, щоб не зберігати всю логіку в одному об’єкті. Окремі контролери відповідають за стан гри, меню, діалоги, інвентар, читання текстових об’єктів, екран смерті та масштабування інтерфейсу.

У проєкті використано такі основні контролери: «obj_game_controller», «obj_display_controller», «obj_dialog_controller», «obj_inventory_controller», «obj_read_controller», «obj_main_menu_controller» та «obj_death_controller». Кожен із них має свою зону відповідальності і взаємодіє з іншими системами через глобальні стани, змінні або виклик окремих скриптів [6, 7].

Центральним контролером є «obj_game_controller». Він задає основні режими роботи гри та початкові параметри персонажа. Завдяки цьому інші об’єкти можуть перевіряти, у якому стані зараз перебуває гра: вільне переміщення, діалог, вибір відповіді, інвентар або катсцена.

У події Create об’єкта «obj_game_controller» задаються глобальні стани, які використовуються в інших частинах проєкту.

```
persistent = true;
// стани гри
global.GAME_FREE = 0;
global.GAME_DIALOG = 1;
global.GAME_CHOICE = 2;
global.GAME_INVENTORY = 3;
global.GAME_CUTSCENE = 4;
global.interact_ready = true;
// статистика гравця
global.player_level = 1;
global.player_max_level = 5;
global.player_exp = 0;
global.player_exp_to_next = 5;
global.player_max_hp = 5;
global.player_hp = global.player_max_hp;
```

Рис. 3.2.1 Фрагмент коду опису глобальних станів гри в об’єкті «obj_game_controller», подія Create

У фрагменті коду на рисунку 3.2.1 показано, що гра має кілька режимів роботи. Наприклад, у стані «GAME_FREE» гравець може вільно рухатися, а під час «GAME_DIALOG» керування передається діалоговій системі. Такий поділ потрібний, щоб гравець не міг одночасно рухатися, відкривати інвентар або запускати іншу взаємодію під час діалогу чи катсцени.

Контролер «obj_display_controller» відповідає за масштабування та правильне розміщення GUI. Він потрібний для того, щоб меню, діалоги, інвентар, вікно читання та екран смерті не залежали напряму від фактичного розміру вікна гри.

Контролер «obj_dialog_controller» керує діалоговою системою. Він зберігає поточного NPC, номер репліки, кількість показаних символів, режим вибору відповіді та параметри діалогового вікна. Сам NPC не малює діалог самостійно, а тільки передає репліки в контролер.

Контролер «obj_inventory_controller» відповідає за відкриття інвентаря, вибір предметів, групування однакових речей і використання предметів. Він працює з глобальним списком інвентаря, який поповнюється під час підбору предметів у локаціях.

Контролер «obj_read_controller» використовується для читабельних об’єктів: записок, листів та інших текстових елементів. Він працює окремо від

діалогової системи, тому звичайні розмови з NPC і читання предметів не змішуються в одному механізмі.

Окремо реалізовано «obj_main_menu_controller». Він відповідає за головне меню, запуск нової гри, вихід із гри та перевірку наявності файлу збереження. Якщо файл «save_game.json» існує, користувач може продовжити проходження. Якщо збереження немає, пункт продовження не повинен запускати завантаження.

```
save_path = "The Dream Where I Save You/save_game.json";
save_exists = file_exists(save_path);
buttons = ["Нова гра", "Продовжити", "Вихід"];
if (keyboard_check_pressed(vk_enter)
    || keyboard_check_pressed(ord("E"))
    || keyboard_check_pressed(ord("У"))) {
    if (selected_index == 0) {
        scr_ensure_game_controllers();
        global.current_run_has_save = false;
        room_goto(rm_players_house_entrance_and_living_room);
    }
    if (selected_index == 1 && save_exists) {
        scr_load_game();
    }
    if (selected_index == 2) {
        game_end();
    }
}
```

Рис. 3.2.2 Фрагмент коду перевірки збереження та вибору пункту меню в об’єкті «obj_main_menu_controller», подія Step

У фрагменті коду на рисунку 3.2.2 показано, що меню не просто відображає список пунктів. Воно перевіряє наявність файлу збереження і залежно від вибору користувача запускає нову гру, завантажує прогрес або завершує роботу програми. Перед початком нової гри викликається «scr_ensure_game_controllers()», щоб потрібні контролери були створені перед переходом у першу кімнату.

Контролер «obj_death_controller» відповідає за ситуацію після смерті персонажа. Він перевіряє, чи є збереження в поточному проходженні, і залежно від цього пропонує або завантажити гру, або почати проходження заново. Сам персонаж не обробляє це меню напряму, а тільки відкриває відповідний контролер після втрати всього здоров’я.

Загальна структура контролерів зменшує дублювання коду. NPC не малюють діалоги самостійно, предмети не відкривають інвентар напряму, а

екран смерті не зберігається всередині персонажа. Для кожної задачі є окремий контролер, який виконує свою частину роботи. Завдяки цьому нові NPC, предмети, квести або інтерфейсні вікна можна додавати без повного переписування основної логіки гри.

3.3 Реалізація керування персонажем та взаємодії з ігровим світом

Керування персонажем реалізовано в об’єкті «obj_player_controller». Він відповідає за переміщення гравця по кімнаті, збереження останнього напрямку руху, перемикання анімацій, запуск атаки, отримання шкоди та реакцію на смерть персонажа. Через цей об’єкт користувач виконує основні дії в грі, тому «obj_player_controller» пов’язаний із бойовою системою, переходами між кімнатами, екраном смерті та глобальними станами гри.

Під час створення об’єкта персонажа задаються основні параметри руху, атаки та невразливості після отримання шкоди. Також у цьому об’єкті описано функцію «take_damage», яка зменшує здоров’я гравця тільки тоді, коли таймер невразливості вже завершився.

```
if (instance_number(obj_player_controller) > 1) {
    instance_destroy();
    exit;
}
persistent = true;
last_direction = "down";
simple_speed = 2;
attack_cooldown = 0;
attack_cooldown_max = 38;
is_attacking = false;
attack_timer = 0;
attack_duration = 34;
player_invuln_timer = 0;
player_invuln_duration = 40;
take_damage = function(_damage) {
    if (player_invuln_timer > 0) return;
    global.player_hp -= _damage;
    player_invuln_timer = player_invuln_duration;
};
```

Рис. 3.3.1 Фрагмент коду ініціалізації параметрів персонажа в об’єкті «obj_player_controller», подія Create

У фрагменті коду на рисунку 3.3.1 спочатку перевіряється кількість екземплярів «obj_player_controller». Це потрібно через те, що персонаж є

persistent-об’єктом і не повинен дублюватися після переходів між кімнатами. Якщо таких об’єктів стає більше одного, зайвий екземпляр знищується.

Змінна «last_direction» зберігає останній напрямок погляду персонажа. Вона використовується не тільки для idle-анімацій, а й для атаки, бо зона удару створюється з того боку, куди дивиться гравець. Таймери «attack_cooldown» і «attack_timer» обмежують частоту атак і тривалість бойової анімації. Функція «take_damage» захищає персонажа від повторного миттєвого отримання шкоди, оскільки після удару вмикається короткий період невразливості.

У події Step обробляється поточний стан персонажа. Якщо здоров’я стає нульовим, рух блокується, гра переходить у стан катсцени, а контролер «obj_death_controller» відкриває меню смерті. Окремо перевіряється змінна «global.player_can_move», яка використовується для блокування керування під час діалогів, інвентаря, катсцен або інших режимів, коли гравець не повинен пересуватися.

```
if (player_invuln_timer > 0) {
    player_invuln_timer--;
}
if (global.player_hp <= 0) {
    global.player_hp = 0;
    global.player_can_move = false;
    scr_set_game_state(global.GAME_CUTSCENE);
    if (instance_exists(obj_death_controller)) {
        obj_death_controller.open = true;
        obj_death_controller.selected_index = 0;
    }
    exit;
}
if (!global.player_can_move) {
    switch (last_direction) {
        case "right": sprite_index = player_simple_idle_right; break;
        case "left":  sprite_index = player_simple_idle_left;  break;
        case "up":    sprite_index = player_simple_idle_up;    break;
        case "down":  sprite_index = player_simple_idle_down;  break;
    }
    depth = -bbox_bottom;
    exit;
}
```

Рис. 3.3.2 Фрагмент коду блокування руху та обробки смерті персонажа в об’єкті «obj_player_controller», подія Step

У фрагменті коду на рисунку 3.3.2 показано зв’язок персонажа з іншими системами. «obj_player_controller» не малює екран смерті самостійно, а лише визначає факт смерті та відкриває «obj_death_controller». Завдяки цьому логіка смерті не змішується з логікою руху.

Перевірка «global.player_can_move» потрібна для спільного блокування керування. Її можуть використовувати різні системи: діалоги, інвентар, читання об’єктів або катсцени. Якщо рух заблокований, персонаж не змінює координати, але залишається в idle-анімації відповідно до останнього напрямку.

Сам рух персонажа реалізовано через перевірку клавіш і зміну координат «x» та «y». Перед переміщенням виконується перевірка зіткнення з об’єктом «obj_wall» через «place_meeting». Тому персонаж не проходить крізь стіни, меблі або інші непрохідні об’єкти. Для кожного напрямку використовується однакова схема: перевірка клавіші, перевірка колізії, зміна координати, оновлення анімації та запис останнього напрямку.

Атака персонажа також залежить від «last_direction». Після натискання клавіші атаки створюється об’єкт «obj_damage_controller». Він з’являється не в центрі персонажа, а зі зміщенням у напрямку удару. Саме цей об’єкт потім перевіряє попадання і наносить шкоду ворогу.

```
if ((keyboard_check_pressed(ord("X")) || keyboard_check_pressed(ord("J")))
    && attack_cooldown <= 0 && !is_attacking) {
    is_attacking = true;
    attack_timer = attack_duration;
    attack_cooldown = attack_cooldown_max;
    var hitbox_x = x;
    var hitbox_y = y;
    switch (last_direction) {
        case "right": hitbox_x = x + 12; break;
        case "left":  hitbox_x = x - 12; break;
        case "up":    hitbox_y = y - 12; break;
        case "down":  hitbox_y = y + 12; break;
    }
    var hitbox = instance_create_depth(hitbox_x, hitbox_y, 0, obj_damage_controller);
    hitbox.damage = global.player_damage;
    hitbox.owner = id;
    hitbox.attack_direction = last_direction;
}
```

Рис. 3.3.3 Фрагмент коду створення зони удару в об’єкті «obj_player_controller», подія Step

У фрагменті коду на рисунку 3.3.3 атака не виконує пряме зменшення здоров’я ворога. Персонаж тільки створює зону шкоди, передає їй значення «global.player_damage», власний ідентифікатор і напрямок атаки. Такий поділ важливий для бойової системи: «obj_player_controller» відповідає за дію гравця, а «obj_damage_controller» — за перевірку попадання та нанесення шкоди.

Окремо в «obj_player_controller» реалізовано встановлення позиції після переходу або завантаження. Якщо змінна «global.use_spawn_point» активна, у події Room Start персонаж отримує координати з «global.spawn_x» та «global.spawn_y». Після цього прапорець вимикається. Завдяки цьому після переходу між кімнатами або завантаження гри персонаж з'являється в потрібній точці.

Таким чином, «obj_player_controller» поєднує кілька важливих частин ігрової логіки: рух, напрямок, анімації, атаку, отримання шкоди, блокування керування та встановлення позиції після переходів. При цьому він не бере на себе всю логіку гри. Діалоги, інвентар, смерть, збереження і нанесення шкоди винесені в окремі об'єкти та контролери, тому персонаж залишається центральним об'єктом керування, але не перетворюється на єдине місце зберігання всієї логіки проєкту.

3.4 Реалізація системи NPC, діалогів та квестових станів

Системи NPC, діалогів і квестів у проєкті тісно пов'язані між собою. NPC не тільки запускає розмову з гравцем, а й може змінювати свій стан, пропонувати завдання, перевіряти прогрес квесту або переводити гравця до наступного етапу проходження. Через це ці системи реалізовано не як окремі незалежні частини, а як спільний механізм взаємодії [2, 11].

Базова логіка NPC винесена в батьківський об'єкт «obj_npc_parent». Він відповідає за перевірку активності NPC, відстані до гравця та запуск діалогу. Конкретні персонажі, наприклад «obj_npc_grape» або «obj_npc_alicia», наслідують цю логіку і задають власні репліки, портрети, стани та реакції на вибір гравця.

У події Step об'єкта «obj_npc_parent» перевіряється, чи може NPC бути активним у поточному стані проходження. Якщо персонаж активний, система

визначає, чи знаходиться гравець поруч, і після натискання клавіші взаємодії запускає діалог.

```
player_near = false;
npc_state_active = true;
if (!allow_all_npc_states) {
    npc_state_active = false;
    var current_story_state = get_story_state();
    for (var i = 0; i < array_length(allowed_npc_states); i++) {
        if (allowed_npc_states[i] == current_story_state) {
            npc_state_active = true;
            break;
        }
    }
}
visible = npc_state_active;
can_interact = npc_state_active;
if (!npc_state_active) exit;
if (global.game_state != global.GAME_FREE) exit;
if (instance_exists(obj_player_controller)) {
    var nearest_npc = instance_nearest(
        obj_player_controller.x,
        obj_player_controller.y,
        obj_npc_parent
    );
    var dist = point_distance(x, y, obj_player_controller.x, obj_player_controller.y);
    if (nearest_npc == id && dist <= interact_distance && can_interact) {
        player_near = true;
    }
}
if (player_near && global.interact_ready
    && (keyboard_check_pressed(ord("Z")) || keyboard_check_pressed(ord("E")))) {
    refresh_dialog();
    scr_start_dialog(id);
}
```

Рис. 3.4.1 Фрагмент коду перевірки активності NPC та запуску діалогу в об’єкті «obj_npc_parent», подія Step

У фрагменті коду на рисунку 3.4.1 показано, що NPC спочатку перевіряє власну активність. Якщо поточний стан проходження не входить до списку «allowed_npc_states», персонаж не відображається і не може запускати діалог. Також діалог дозволено починати тільки тоді, коли гра перебуває у стані «GAME_FREE». Це захищає систему від запуску нової розмови під час іншого діалогу, катсцени або відкритого інтерфейсного вікна.

Для конкретних NPC використовується дочірня реалізація. Наприклад, «obj_npc_grape» працює зі станом «global.grape_state». Цей стан визначає, на

якому етапі проходження перебуває Виноградинка: перша зустріч, очікування яблук, перехід до іншої локації або етап із квестом про орків.

```
event_inherited();
if (!variable_global_exists("grape_state")) {
    global.grape_state = 0;
}
allowed_npc_states = [0, 1, 2, 3, 10, 11, 12, 13, 14, 15, 21];
allow_all_npc_states = false;
get_dialog_state = function() {
    return global.grape_state;
};
set_dialog_state = function(_value) {
    global.grape_state = _value;
};
get_story_state = function() {
    return global.grape_state;
};
```

Рис. 3.4.2 Фрагмент коду налаштування станів NPC в об’єкті «obj_npc_grape», подія Create

У фрагменті коду на рисунку 3.4.2 об’єкт «obj_npc_grape» спочатку викликає «event_inherited()», тобто отримує базову логіку з «obj_npc_parent». Після цього він перевизначає функції роботи зі станом. Завдяки цьому батьківський об’єкт не знає деталей конкретного персонажа, але все одно може перевіряти його активність і запускати діалог.

Діалогова система працює через окремий контролер «obj_dialog_controller». NPC формує масив «dialog_lines», а контролер уже відповідає за показ реплік, поступовий друк тексту, перехід між рядками та режим вибору відповіді. Такий поділ потрібний, щоб кожен NPC не малював діалогове вікно самостійно. Код малювання діалогового вікна в події Draw GUI End об’єкта «obj_dialog_controller» наведено в додатку А.

```
if (!active) exit;
var _line = current_npc.dialog_lines[current_line];
var _text = _line.text;
var _text_len = string_length(_text);
if (visible_chars < _text_len) {
    type_timer++;
    if (type_timer >= type_delay) {
        visible_chars++;
        type_timer = 0;
    }
}
if (input_lock <= 0 && (keyboard_check_pressed(ord("Z")) ||
keyboard_check_pressed(ord("E")))) {
    if (visible_chars < _text_len) {
        visible_chars = _text_len;
    }
    else if (variable_struct_exists(_line, "choices")) {
        choice_mode = true;
        pending_choice_tag = _line.choice_tag;
        scr_set_game_state(global.GAME_CHOICE);
    }
    else if (current_line < array_length(current_npc.dialog_lines) - 1) {
        current_line++;
        visible_chars = 0;
        type_timer = 0;
    }
    else {
        current_npc.on_dialog_finished();
        active = false;
        current_npc = noone;
        scr_set_game_state(global.GAME_FREE);
    }
}
```

Рис. 3.4.3 Фрагмент коду обробки реплік у контролері «obj_dialog_controller», подія Step

У фрагменті коду на рисунку 3.4.3 реалізовано поступове відкриття тексту. Якщо гравець натискає клавішу під час друку, репліка одразу показується повністю. Якщо текст уже відкритий, контролер переходить до наступної репліки або завершує діалог. Якщо в поточній репліці є поле «choices», гра переходить у стан «GAME_CHOICE», а керування персонажем тимчасово блокується.

Квестова система реалізована через батьківський об’єкт «obj_quest_parent». Його задача — зберігати спільну логіку для всіх завдань: ідентифікатор, назву, опис, отримання стану, зміну стану та перевірку етапу виконання. Стан квестів зберігається в глобальній мапі «global.quest_states».

У проєкті використано такі стани квесту: «0» — завдання ще не почато, «1» — завдання активне, «-1» — завдання відхилено, «2» — завдання завершено.

```
quest_id = "";
quest_name = "";
quest_description = "";
get_state = function() {
    if (!ds_exists(global.quest_states, ds_type_map)) {
        return 0;
    }
    if (ds_map_exists(global.quest_states, quest_id)) {
        return global.quest_states[? quest_id];
    }
    return 0;
};
set_state = function(_value) {
    if (!ds_exists(global.quest_states, ds_type_map)) {
        return;
    }
    global.quest_states[? quest_id] = _value;
};
accept_quest = function() {
    set_state(1);
};
decline_quest = function() {
    set_state(-1);
};
complete_quest = function() {
    set_state(2);
};
```

Рис. 3.4.4 Фрагмент коду збереження та зміни стану квесту в об’єкті «obj_quest_parent», подія Create

У фрагменті коду на рисунку 3.4.4 стан квесту не зберігається напряму в самому об’єкті. Кожне завдання має власний «quest_id», а його значення записується в «global.quest_states». Це важливо для системи збереження, бо під час запису прогресу достатньо зберегти одну мапу станів, щоб після завантаження відновити всі активні, відхилені та завершені завдання.

Як приклад конкретного квесту можна розглянути «obj_quest_grape_apples». У цьому завданні гравець має принести Виноградинці три червоні яблука. Квест успадковує базову логіку з «obj_quest_parent», але задає власну умову виконання: перевірку кількості предметів в інвентарі.

```
event_inherited();
persistent = true;
global.quest_grape_apples = id;
quest_id = "grape_apples";
quest_name = "Принести 3 яблука Виноградинці";
quest_description = "Принести 3 яблука Виноградинці";
required_item_name = "Червоне яблуко";
required_item_count = 3;
has_enough_items = function() {
    return scr_inventory_count(required_item_name) >= required_item_count;
};
try_complete_quest = function() {
    if (!is_active()) {
        return false;
    }
    if (has_enough_items()) {
        scr_inventory_remove(required_item_name, required_item_count);
        complete_quest();
        scr_add_inventory_item_from_object(obj_gold_coin, 1);
        scr_add_player_exp(3);
        return true;
    }
    return false;
};
```

Рис. 3.4.5 Фрагмент коду квесту з яблуками в об’єкті «obj_quest_grape_apples», подія Create

У фрагменті коду на рисунку 3.4.5 показано зв’язок квесту з інвентарем. Спочатку перевіряється, чи активне завдання. Після цього функція «has_enough_items()» визначає, чи є в інвентарі потрібна кількість яблук. Якщо умова виконана, яблука видаляються, квест переходить у стан завершеного, а гравець отримує нагороду.

Інші квести працюють за тією самою схемою, але мають інші умови виконання. Наприклад, «obj_quest_grape_orcs» перевіряє не предмети, а кількість переможених орків. Фінальний квест «obj_quest_final_knights» також використовує лічильник переможених ворогів, але пов’язаний уже з Алісією та завершенням проходження. Їхній код не дублюється в основному тексті, оскільки принцип роботи залишається однаковим: квест має стан, перевіряє умову, змінює прогрес і після виконання переходить у стан «2».

У результаті NPC, діалоги та квести працюють як одна пов’язана система. «obj_npc_parent» відповідає за загальну взаємодію з гравцем, дочірні NPC задають конкретні стани та репліки, «obj_dialog_controller» показує текст і обробляє вибір, а «obj_quest_parent» зберігає спільну логіку завдань. Завдяки цьому нові NPC або квести можна додавати через дочірні об’єкти, не переписуючи базову систему взаємодії.

3.5 Реалізація предметів, інвентаря та читабельних об’єктів

Система предметів у грі пов’язана з інвентарем, квестами та збереженням прогресу. Після підбору предмет не просто зникає з кімнати, а додається до глобального списку інвентаря. Також його ідентифікатор записується до списку вже підібраних об’єктів. Це потрібно для того, щоб після переходу між кімнатами або завантаження гри той самий предмет не з’являвся повторно.

Для спільної логіки предметів використано батьківський об’єкт «obj_pickable_parent». Він відповідає за перевірку відстані до гравця, взаємодію з предметом, додавання даних предмета до інвентаря та збереження факту підбору. Дочірні об’єкти, наприклад «obj_red_apple» і «obj_gold_coin», задають тільки конкретні властивості предмета: назву, спрайт, опис, текст використання та можливий ефект.

Основна логіка підбору предмета розміщена в події Step об’єкта «obj_pickable_parent».

```
if (ds_map_exists(global.picked_items, item_id)) {
    instance_destroy();
    exit;
}
if (player_near && global.interact_ready
    && (keyboard_check_pressed(ord("Z")) || keyboard_check_pressed(ord("E")))) {
    var item_data = {
        name : item_name,
        sprite : item_sprite,
        description : item_description,
        use_text : item_use_text,
        can_use : item_can_use,
        heal_amount : item_heal_amount
    };
    ds_list_add(global.inventory, item_data);
    ds_map_replace(global.picked_items, item_id, true);
    global.interact_ready = false;
    instance_destroy();
}
```

Рис. 3.5.1 Фрагмент коду підбору предмета в об’єкті «obj_pickable_parent», подія Step

У фрагменті коду на рисунку 3.5.1 спочатку перевіряється, чи був предмет уже підібраний раніше. Якщо його ідентифікатор є в «global.picked_items», об’єкт знищується і не залишається активним у кімнаті. Після взаємодії з гравцем створюється структура «item_data», у якій зберігаються дані предмета. Саме ця

структура додається до «global.inventory», тому інвентар працює не з об’єктом у кімнаті, а з його збереженим описом.

Дочірні предмети використовують логіку «obj_pickable_parent», але задають власні параметри. Наприклад, «obj_red_apple» є предметом, який можна підібрати та використати для відновлення здоров’я. «obj_gold_coin» використовується як нагорода за виконання квесту і також зберігається в інвентарі.

Інвентар реалізовано в контролері «obj_inventory_controller». Він відповідає за відкриття інвентаря, вибір предмета, групування однакових речей і використання предметів. Групування потрібне для того, щоб кілька однакових предметів не займали окремі рядки, а показувалися як один запис із кількістю.

```
grouped_items = [];
grouped_counts = [];
for (var i = 0; i < ds_list_size(global.inventory); i++) {
    var item = global.inventory[| i];
    var found = -1;
    for (var j = 0; j < array_length(grouped_items); j++) {
        if (grouped_items[j].name == item.name) {
            found = j;
            break;
        }
    }
    if (found == -1) {
        var idx = array_length(grouped_items);
        grouped_items[idx] = item;
        grouped_counts[idx] = 1;
    } else {
        grouped_counts[found] += 1;
    }
}
```

Рис. 3.5.2 Фрагмент коду групування предметів в об’єкті «obj_inventory_controller», подія Step

У фрагменті коду на рисунку 3.5.2 інвентар проходить по списку «global.inventory» і формує два масиви: «grouped_items» та «grouped_counts». Перший масив зберігає унікальні предмети, другий — їх кількість. Це важливо не тільки для відображення інвентаря, а й для квестів, де потрібно перевіряти конкретну кількість предметів, наприклад три червоні яблука для завдання Виноградинки.

Використання предметів також обробляється в «obj_inventory_controller». Якщо предмет має лікувальний ефект, контролер змінює здоров’я гравця, але не дозволяє перевищити максимальне значення «global.player_max_hp».

```
if (keyboard_check_pressed(ord("Z")) || keyboard_check_pressed(ord("E"))) {
    var selected_item = grouped_items[selected_index];
    if (!selected_item.can_use) {
        use_message = selected_item.use_text;
        use_message_timer = 90;
        exit;
    }
    if (selected_item.heal_amount > 0) {
        if (global.player_hp >= global.player_max_hp) {
            use_message = "HP вже повне";
            use_message_timer = 90;
            exit;
        }
        global.player_hp = min(
            global.player_hp + selected_item.heal_amount,
            global.player_max_hp
        );
        use_message = selected_item.use_text;
        use_message_timer = 90;
    }
}
```

Рис. 3.5.3 Фрагмент коду використання предмета в об’єкті «obj_inventory_controller», подія Step

У фрагменті коду на рисунку 3.5.3 показано використання предмета з інвентаря. Якщо предмет не можна застосувати, система показує відповідне повідомлення. Якщо предмет відновлює здоров’я, значення «global.player_hp» збільшується на «heal_amount», але не може стати більшим за «global.player_max_hp». Так інвентар пов’язується з характеристиками персонажа.

Окремо в грі реалізовано читабельні об’єкти. Вони використовуються для записок, листів та інших текстових елементів, через які гравець отримує сюжетну інформацію. Для цього використовується батьківський об’єкт «obj_readable_parent» і контролер «obj_read_controller». Сам об’єкт у кімнаті не малює панель читання. Він передає заголовок і текст у контролер, а «obj_read_controller» відкриває інтерфейс читання та повертає гру у звичайний стан після закриття. Код малювання панелі читання в події Draw GUI End об’єкта «obj_read_controller» наведено в додатку А.

```
if (!open) exit;
if (input_lock > 0) {
    input_lock--;
}
if (input_lock <= 0
    && (keyboard_check_pressed(ord("Z")) || keyboard_check_pressed(ord("E")))) {
    open = false;
    current_title = "";
    current_text = "";
    global.interact_ready = false;
    scr_set_game_state(global.GAME_FREE);
}
```

Рис. 3.5.4 Фрагмент коду закриття вікна читання в об’єкті «obj_read_controller», подія Step

У фрагменті коду на рисунку 3.5.4 видно, що після закриття читабельного об’єкта очищаються «current_title» і «current_text», а стан гри повертається до «GAME_FREE». Змінна «global.interact_ready» тимчасово блокує повторну взаємодію, щоб записка не відкривалася знову одразу після закриття.

У результаті система предметів, інвентаря та читабельних об’єктів працює як частина загального проходження. Предмети додаються до «global.inventory», факт їх підбору зберігається в «global.picked_items», інвентар дозволяє групувати й використовувати отримані речі, а читабельні об’єкти передають сюжетну інформацію через окремий контролер. Завдяки цьому предмети й записки не існують ізольовано, а пов’язані з квестами, станом гравця, сюжетом і системою збереження прогресу.

3.6 Реалізація бойової системи та поведінки ворогів

Бойова система в грі побудована через взаємодію кількох об’єктів: «obj_player_controller», «obj_damage_controller», «obj_enemy_parent» та дочірніх об’єктів ворогів. Гравець не змінює здоров’я ворога напряму. Під час атаки він створює окрему зону шкоди, а вже ця зона перевіряє попадання і передає ворогу кількість шкоди.

Такий поділ потрібний для того, щоб не змішувати керування персонажем, бойову анімацію, перевірку зіткнення та зміну стану ворога в одному місці. «obj_player_controller» відповідає за запуск атаки, «obj_damage_controller» — за

перевірку попадання, а «obj_enemy_parent» — за здоров’я ворога, отримання шкоди, смерть, досвід і зв’язок із квестами.

Спільна логіка ворогів винесена в батьківський об’єкт «obj_enemy_parent». Він містить загальні параметри та поведінку ворога: здоров’я, швидкість руху, дистанцію бачення, дистанцію атаки, отримання шкоди, коротку невразливість після удару, смерть і оновлення квестового прогресу. Дочірні об’єкти ворогів не дублюють цю логіку, а тільки задають власні характеристики.

Наприклад, простий орк реалізований як дочірній об’єкт «obj_enemy_simple_orc». У ньому задаються параметри конкретного ворога: кількість здоров’я, шкода, швидкість, дистанція бачення, дистанція атаки та можлива нагорода після перемоги.

```
max_hp = 3;
hp = max_hp;
exp_reward = 3;
enemy_damage = 1;
move_speed = 1;
vision_range = 100;
stop_distance = 20;
attack_range = 24;
attack_prepare_duration = 60;
attack_cooldown_max = 120;
attack_anim_duration = 25;
loot_table = [
    { item_object : obj_red_apple, chance : 50 },
    { item_object : obj_gold_coin, chance : 5 }
];
```

Рис. 3.6.1 Фрагмент коду налаштування параметрів орка в об’єкті «obj_enemy_simple_orc», подія Create

У фрагменті коду на рисунку 3.6.1 спочатку викликається «event_inherited()», тому орк отримує базову поведінку з «obj_enemy_parent». Після цього задаються тільки його власні параметри. Завдяки цьому для створення іншого ворога не потрібно переписувати переслідування, атаку або отримання шкоди. Достатньо створити дочірній об’єкт і змінити характеристики.

Для орка задано 3 одиниці здоров’я, 1 одиницю шкоди, швидкість руху та дистанцію бачення. Також налаштовано таймери атаки: час підготовки удару, затримка між атаками та тривалість бойової анімації. Окремо задано «loot_table»,

через яку ворог після смерті може залишити яблуко або золоту монету. Так бойова система пов’язується з предметами та інвентарем.

За перевірку попадання відповідає «obj_damage_controller». Цей об’єкт створюється під час атаки гравця або ворога і короткий час існує як зона ураження. Він перевіряє, чи перетинається з ціллю, і викликає в неї функцію отримання шкоди.

```
with (obj_enemy_parent) {
    if (id != other.owner && place_meeting(x, y, other)) {
        if (ds_list_find_index(other.hit_enemies, id) == -1) {
            ds_list_add(other.hit_enemies, id);
            if (variable_instance_exists(id, "take_damage")) {
                take_damage(other.damage);
            }
        }
    }
}
```

Рис. 3.6.2 Фрагмент коду перевірки попадання по ворогу в об’єкті «obj_damage_controller», подія Step

У фрагменті коду на рисунку 3.6.2 зона шкоди перевіряє всі екземпляри «obj_enemy_parent». Якщо ворог перетинається із зоною удару і ще не був оброблений цією атакою, його ідентифікатор додається до списку «hit_enemies». Це потрібно, щоб один удар не наносив шкоду багато разів за кілька кадрів.

Після цього викликається функція «take_damage()», яка належить самому ворогу. Тобто «obj_damage_controller» не вирішує, як ворог має реагувати на удар. Він тільки передає факт попадання і кількість шкоди. Ворог уже сам зменшує здоров’я, перевіряє смерть і виконує дії після перемоги над ним.

Дочірні вороги відрізняються не тільки параметрами, а й анімаціями. Наприклад, «obj_enemy_simple_orc» після виконання батьківської логіки вибирає потрібний спрайт залежно від стану ворога та останнього напрямку руху.

```
event_inherited();
if (enemy_state == ENEMY_CHASE
    || enemy_state == ENEMY_WANDER
    || enemy_state == ENEMY_ATTACK) {

    switch (last_direction) {
        case "right": sprite_index = enemy_simple_orc_move_right; break;
        case "left":  sprite_index = enemy_simple_orc_move_left;  break;
        case "up":    sprite_index = enemy_simple_orc_move_up;    break;
        case "down":  sprite_index = enemy_simple_orc_move_down;  break;
    }
}
else {
    switch (last_direction) {
        case "right": sprite_index = enemy_simple_orc_idle_right; break;
        case "left":  sprite_index = enemy_simple_orc_idle_left;  break;
        case "up":    sprite_index = enemy_simple_orc_idle_up;    break;
        case "down":  sprite_index = enemy_simple_orc_idle_down;  break;
    }
}
```

Рис. 3.6.3 Фрагмент коду вибору анімації орка в об’єкті «obj_enemy_simple_orc», подія Step

У фрагменті коду на рисунку 3.6.3 дочірній ворог не обробляє переслідування або атаку самостійно. Спочатку виконується «event_inherited()», тобто працює логіка «obj_enemy_parent». Після цього дочірній об’єкт тільки вибирає потрібну анімацію. Якщо ворог переслідує, рухається або атакує, використовується анімація руху. Якщо він не виконує активної дії, показується idle-анімація відповідно до останнього напрямку.

У грі використано кілька типів ворогів. «obj_enemy_simple_orc» є простішим ворогом для проміжних локацій і використовується в квесті Виноградинки. «obj_enemy_knight» має сильніші параметри і використовується у фінальній частині гри, де гравець має перемогти лицарів для завершення квесту порятунку Алісії. Обидва типи ворогів використовують спільну батьківську логіку, але мають різні характеристики, спрайти та роль у проходженні.

Окремо для квестових ворогів реалізовано мітку цілі. Вона показується тільки тоді, коли ворог належить до активного квесту. Це допомагає гравцю зрозуміти, які противники потрібні для виконання завдання.

```
if (show_target_marker && is_quest_target) {
    var show_marker = false;
    if (quest_target_id != "") {
        var quest_var_name = "quest_" + quest_target_id;

        if (variable_global_exists(quest_var_name)) {
            var q = variable_global_get(quest_var_name);

            if (instance_exists(q) && variable_instance_exists(q, "is_active")) {
                show_marker = q.is_active();
            }
        }
    }
    if (show_marker) {
        draw_set_halign(fa_center);
        draw_set_valign(fa_middle);
        draw_set_color(c_red);
        draw_text(x, y - (sprite_get_height(sprite_index) / 2) - 8, "!");
        draw_set_color(c_white);
        draw_set_halign(fa_left);
        draw_set_valign(fa_top);
    }
}
```

Рис. 3.6.4 Фрагмент коду відображення квестової мітки в об’єкті «obj_enemy_parent», подія Draw

У фрагменті коду на рисунку 3.6.4 мітка не показується постійно. Спочатку перевіряється, чи ворог є квестовою ціллю. Потім за «quest_target_id» формується назва глобальної змінної квесту. Якщо відповідний квест існує і зараз активний, над ворогом малюється знак оклику.

Таке рішення зберігає поділ відповідальності між системами. Ворог не зберігає всю логіку квесту всередині себе. Він лише має ознаку, що може бути ціллю певного завдання. Сам стан завдання перевіряється через квестовий об’єкт.

У результаті бойова система працює як набір пов’язаних частин. Гравець створює зону удару, «obj_damage_controller» перевіряє попадання, «obj_enemy_parent» обробляє здоров’я, смерть і нагороди, а квестова система реагує на перемогу над потрібними ворогами. Завдяки батьківському об’єкту «obj_enemy_parent» спільна логіка ворогів не дублюється, а дочірні об’єкти можуть відрізнятися параметрами, анімаціями та роллю в проходженні гри.

3.7 Реалізація переходів між кімнатами, колізій та глибини малювання

Ігровий світ у проєкті складається з окремих кімнат. Щоб ці кімнати сприймалися як частини одного світу, у грі реалізовано переходи між локаціями, колізії з перешкодами та зміну глибини малювання об’єктів. За переходи відповідає об’єкт «obj_warp_block», за фізичне обмеження руху — «obj_wall», а порядок відображення персонажів і декорацій визначається через значення «depth» [9].

Переходи між кімнатами виконуються через «obj_warp_block». Такий об’єкт розміщується біля дверей, краю дороги або іншого місця, де гравець має перейти до наступної локації. Для звичайного переходу достатньо задати цільову кімнату та координати появи персонажа.

```
event_inherited();
target_room = rm_way_down_1;
target_x = 159;
target_y = 14 - 14;
```

Рис. 3.7.1 Фрагмент коду налаштування звичайного переходу в об’єкті «obj_warp_block», подія Create

У фрагменті коду на рисунку 3.7.1 показано, що перехід зберігає назву цільової кімнати та координати появи гравця. Завдяки цьому після зміни кімнати персонаж з’являється не у випадковій точці, а біля логічно пов’язаного входу або проходу.

Не всі переходи доступні одразу. Частина з них залежить від сюжетного стану. Для цього в «obj_warp_block» використовується функція «can_warp()». Вона повертає «true», якщо перехід дозволено, або «false», якщо гравець ще не виконав потрібну дію.

```
target_room = rm_way_1;
target_x = 3;
target_y = 494 - 14;
can_warp = function() {
    if (global.grape_state < 1) {
        locked_message = "Огляньте ситуацію навколо будинку.";
        return false;
    }
    if (global.grape_state < 100) {
        locked_message = "Закінчіть розмову з Виноградинкою.";
        return false;
    }
    return true;
};
```

Рис. 3.7.2 Фрагмент коду умовного переходу в об’єкті «obj_warp_block»

У фрагменті коду на рисунку 3.7.2 перехід залежить від значення «global.grape_state». Якщо гравець ще не дійшов до потрібного етапу проходження, перехід блокується, а в «locked_message» записується причина. Це потрібно для послідовного проходження, щоб гравець не міг перейти до наступної локації раніше, ніж завершить важливу розмову або сюжетну дію.

Колізії реалізовано через об’єкт «obj_wall». Він використовується як перешкода для руху. Перед зміною координат персонаж перевіряє, чи немає попереду такого об’єкта через функцію «place_meeting()». Якщо зіткнення є, координата не змінюється [25].

```
if (right_move) {
    if (!place_meeting(x + simple_speed, y, obj_wall)) {
        x += simple_speed;
    }
    if (!is_attacking) {
        sprite_index = player_simple_move_right;
        image_speed = 1;
    }
    last_direction = "right";
    is_moving = true;
}
```

Рис. 3.7.3 Фрагмент коду перевірки колізії з «obj_wall» під час руху персонажа

У фрагменті коду на рисунку 3.7.3 показано рух праворуч. Перед зміною координати «x» викликається «place_meeting()». Якщо попереду немає «obj_wall», персонаж рухається. Якщо перешкода є, координата не змінюється. Такий самий принцип використовується для інших напрямків руху.

Окремо реалізовано глибину малювання об’єктів. У 2D RPG важливо, щоб персонаж міг правильно перекриватися з NPC, ворогами та декораціями. Якщо

об’єкт стоїть нижче на екрані, він має відображатися ближче до переднього плану. Для цього використовується «bbox_bottom».

```
depth = -bbox_bottom;
```

Рис. 3.7.4 Фрагмент коду оновлення глибини малювання персонажа

У фрагменті коду на рисунку 3.7.4 значення «depth» залежить від нижньої межі об’єкта. Чим нижче персонаж знаходиться на екрані, тим ближче він малюється до переднього плану. Це дозволяє зробити перекриття об’єктів природнішим.

Для декорацій використовується схожий принцип, але з додатковим зміщенням «depth_offset». Воно потрібне для об’єктів, у яких порядок відображення має залежати не тільки від нижньої межі спрайта.

```
event_inherited();
visible = true;
depth_offset = 12;
```

Рис. 3.7.5 Фрагмент коду налаштування зміщення глибини декоративного об’єкта

```
scr_update_depth(depth_offset);
```

Рис. 3.7.6 Фрагмент коду оновлення глибини декоративного об’єкта

У фрагментах коду на рисунках 3.7.5 і 3.7.6 показано, що декоративні об’єкти можуть мати власне зміщення глибини. Це корисно для столів, дерев, меблів та інших великих об’єктів, які мають перекривати персонажа не завжди точно за нижньою межею спрайта.

У результаті переходи, колізії та глибина малювання працюють як частини однієї системи побудови локацій. «obj_warp_block» переносить гравця між кімнатами і може враховувати сюжетні умови, «obj_wall» не дозволяє проходити крізь перешкоди, а «depth», «bbox_bottom» і «depth_offset» забезпечують правильний порядок відображення персонажа, NPC, ворогів і декорацій.

3.8 Реалізація системи збереження та завантаження прогресу

Система збереження потрібна для того, щоб гравець міг продовжити проходження після виходу з гри. Для RPG-гри недостатньо зберегти тільки координати персонажа, тому в проєкті записується повніший стан проходження: поточна кімната, позиція гравця, характеристики персонажа, сюжетні стани, квести, інвентар, підібрані предмети та переможені вороги.

Збереження реалізовано у форматі JSON. Для цього використовується скрипт «scr_save_game()». Він формує загальну структуру даних і записує її у файл «save_game.json». У файлі збереження дані поділено на кілька логічних блоків: «player», «story», «quests», «inventory», «picked_items» і «killed_enemies» [10].


```
{
  "player": {
    "room": "rm_way_3",
    "x": 161,
    "y": 138,
    "level": 2,
    "exp": 7,
    "hp": 6,
    "max_hp": 6,
    "damage": 2
  },
  "story": {
    "story_state": 1,
    "grape_state": 101
  },
  "quests": {
    "grape_apples": 2,
    "grape_orcs": 1,
    "final_knights": 1
  },
  "inventory": [
    {
      "name": "Червоне яблуко",
      "sprite": "red_apple",
      "can_use": true,
      "heal_amount": 2
    }
  ],
  "picked_items": [
    "rm_players_house_kitchen_84_121"
  ],
  "killed_enemies": [
    "rm_way_2_obj_enemy_simple_orc_704_288"
  ]
}
```


Рис. 3.8.1 Фрагмент структури JSON-файлу збереження «save_game.json»

У фрагменті на рисунку 3.8.1 показано основну структуру файлу збереження. Блок «player» містить кімнату, координати та характеристики персонажа. Блок «story» зберігає сюжетні змінні, які впливають на появу NPC, доступність переходів і подальший розвиток подій. Блок «quests» містить стани квестів. Наприклад, значення «2» означає завершене завдання, а «1» — активне.

Інвентар зберігається як масив предметів. Для кожного предмета записується назва, назва спрайта, можливість використання та значення лікування. У файл не записується сам об’єкт предмета з кімнати, бо після підбору він уже не існує в локації. Натомість зберігаються його дані, які потім відновлюються в «global.inventory».

Окремо зберігаються списки «picked_items» і «killed_enemies». Перший список потрібний для того, щоб уже підібрані предмети не з’являлися повторно після завантаження. Другий список зберігає переможених ворогів, щоб вони не відновлювалися в кімнатах після повернення до гри.

Завантаження прогресу виконується у скрипті «scr_load_game()». Спочатку перевіряється наявність файлу «save_game.json». Якщо файл існує, він читається як текст і перетворюється назад у структуру даних через JSON. Після цього гра поступово відновлює всі частини прогресу: характеристики гравця, сюжетні стани, квести, інвентар, підібрані предмети та переможених ворогів.

Для відновлення позиції персонажа використовується назва кімнати та координати, записані у блоці «player». Назва кімнати перетворюється на ресурс гри, після чого виконується перехід до потрібної кімнати. Координати записуються в «global.spawn_x» і «global.spawn_y», а змінна «global.use_spawn_point» повідомляє персонажу, що після переходу потрібно з’явитися саме в цій точці.

Повний код скриптів «scr_save_game()» і «scr_load_game()» наведено в додатку Б. У цьому підрозділі показано не весь код, а принцип роботи системи: які дані записуються, як вони групуються у JSON-файлі та як після завантаження відновлюється стан гри.

У результаті система збереження та завантаження відновлює не окрему позицію персонажа, а повний стан проходження. Після завантаження зберігаються виконані квести, предмети в інвентарі, сюжетний прогрес, переможені вороги та вже підібрані об’єкти. Завдяки цьому гра продовжується з того етапу, на якому користувач зробив збереження.

Висновки до розділу 3

У третьому розділі було описано реалізацію програмного продукту піксельної 2D RPG-гри. Для розробки використано середовище «GameMaker Studio 2», мову «GML», редактор піксельної графіки «Aseprite» та шрифт «HaxrCorp 4089 Cyrillic AltGr» для відображення українського тексту в інтерфейсі.

Було реалізовано загальну структуру гри на основі контролерів. Окремі контролери відповідають за стан гри, меню, діалоги, інвентар, читання текстових об’єктів, екран смерті та масштабування інтерфейсу. Такий підхід дозволив не змішувати всю логіку в одному об’єкті та спростив подальше розширення проєкту.

Також було реалізовано керування персонажем через об’єкт «obj_player_controller». У ньому обробляються рух, напрямок, атака, отримання шкоди, блокування керування та встановлення позиції після переходів між кімнатами або завантаження гри. Для бойової взаємодії персонаж створює окремий об’єкт шкоди, що дозволяє відокремити запуск атаки від перевірки попадання.

Система NPC, діалогів і квестів реалізована через батьківські та дочірні об’єкти. «obj_npc_parent» відповідає за базову взаємодію з гравцем, «obj_dialog_controller» — за показ реплік і вибір відповіді, а «obj_quest_parent» — за спільну логіку станів завдань. Завдяки цьому NPC можуть запускати діалоги, змінювати сюжетний стан і впливати на квестовий прогрес.

Окремо було реалізовано систему предметів, інвентаря та читабельних об’єктів. Предмети додаються до «global.inventory», факт їх підбору зберігається в «global.picked_items», а читабельні об’єкти передають текст у контролер читання. Це дозволяє пов’язати предмети й записки з квестами, сюжетом і системою збереження.

Бойова система побудована через взаємодію «obj_damage_controller», «obj_enemy_parent» і дочірніх ворогів. Вороги мають спільну базову логіку, але можуть відрізнятися характеристиками, анімаціями, нагородами та роллю в проходженні. Для квестових ворогів реалізовано мітку, яка показується тільки під час активного завдання.

Також було реалізовано переходи між кімнатами, колізії та глибину малювання. «obj_warp_block» відповідає за перенесення гравця між локаціями, «obj_wall» обмежує рух, а значення «depth», «bbox_bottom» і «depth_offset» забезпечують правильне перекриття персонажа, NPC, ворогів і декорацій.

Наприкінці розділу було описано систему збереження та завантаження прогресу. Дані записуються у файл «save_game.json» у форматі JSON. Зберігаються характеристики гравця, поточна кімната, позиція, сюжетні стани, квести, інвентар, підібрані предмети та переможені вороги. Завдяки цьому після завантаження відновлюється не окрема координата персонажа, а повний стан проходження гри.

4 РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЇ СИСТЕМИ

4.1 Тестування основних сценаріїв роботи гри

Після реалізації основних систем було проведено тестування гри. Його метою була перевірка того, чи правильно працюють ключові сценарії проходження: запуск гри, керування персонажем, взаємодія з NPC, виконання квестів, бій, інвентар, переходи між кімнатами та збереження прогресу. Тестування виконувалося вручну, оскільки програмний продукт є 2D RPG-грою, де важливо перевірити не тільки окремі функції, а й послідовність дій гравця в реальному проходженні.

Першим перевірявся запуск гри та робота головного меню. Під час тестування було перевірено, що користувач може розпочати нову гру, вийти з програми або продовжити проходження за наявності файлу збереження. Якщо файл «save_game.json» відсутній, пункт продовження не повинен запускати завантаження. Якщо збереження вже створене, гра має правильно переходити до відновлення прогресу.

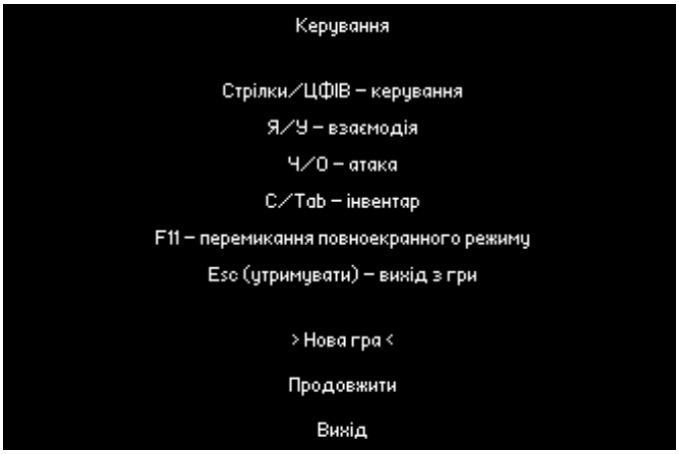


Рис. 4.1.1 Головне меню гри з пунктами запуску нового проходження та продовження гри

Далі перевірялося керування персонажем. Було протестовано рух у чотирьох напрямках, зміну анімацій, збереження останнього напрямку погляду та блокування проходу через об’єкти колізії. Окремо перевірялося, що персонаж не може рухатися під час діалогу, відкритого інвентаря, читання записки або екрану смерті. Це підтверджує правильну роботу глобальних станів гри та змінної «global.player_can_move».

Рис. 4.1.2 Персонаж у локації під час переміщення і взаємодії з ігровим світом

Також було протестовано систему NPC і діалогів. Перевірялося, що підказка взаємодії з’являється тільки тоді, коли персонаж знаходиться поруч із NPC. Після натискання клавіші взаємодії відкривається діалогове вікно, текст виводиться поступово, а після повторного натискання репліка або відкривається повністю, або перемикається на наступну. Для діалогів із вибором перевірялося, що обраний варіант змінює стан NPC або квесту.

Рис. 4.1.3 Діалогове вікно з NPC під час проходження сюжетної частини гри

Окремо тестувалися квести. Для квесту з яблуками перевірялося, що завдання можна прийняти через діалог, після чого гра очікує потрібну кількість

предметів в інвентарі. Якщо яблук недостатньо, квест не завершується. Якщо потрібна кількість предметів є, вони видаляються з інвентаря, стан квесту змінюється на завершений, а гравець отримує нагороду. Для квесту з орками перевірялося, що прогрес збільшується тільки після перемоги над потрібними ворогами і тільки тоді, коли завдання активне.

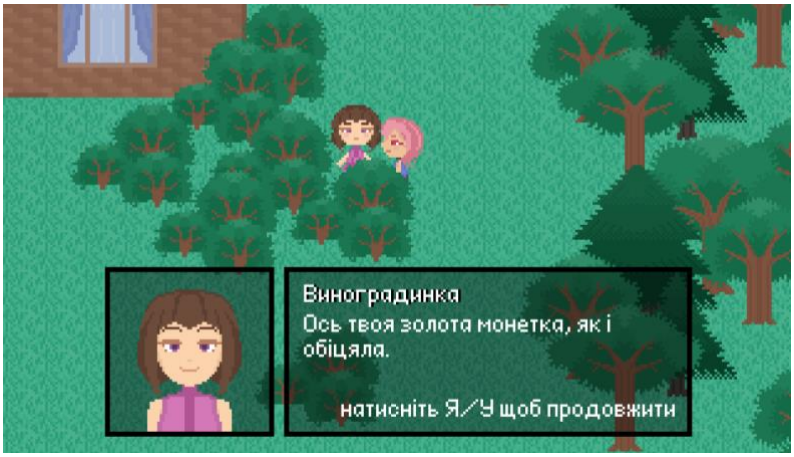


Рис. 4.1.4 Приклад отримання або завершення квесту через діалог з NPC

Було перевірено роботу предметів та інвентаря. Після підбору предмет додається до списку інвентаря, а сам об’єкт зникає з кімнати. Також перевірялося, що вже підібрані предмети не з’являються повторно після переходу між кімнатами або після завантаження гри. В інвентарі однакові предмети групуються, а предмети з лікувальним ефектом відновлюють здоров’я персонажа, але не збільшують його понад максимальне значення.

Під час тестування читабельних об’єктів перевірялося відкриття записки, показ заголовка і тексту, блокування руху під час читання та повернення гри у звичайний стан після закриття вікна. Це потрібно для сюжетних об’єктів, які не є звичайними предметами інвентаря, але впливають на проходження.

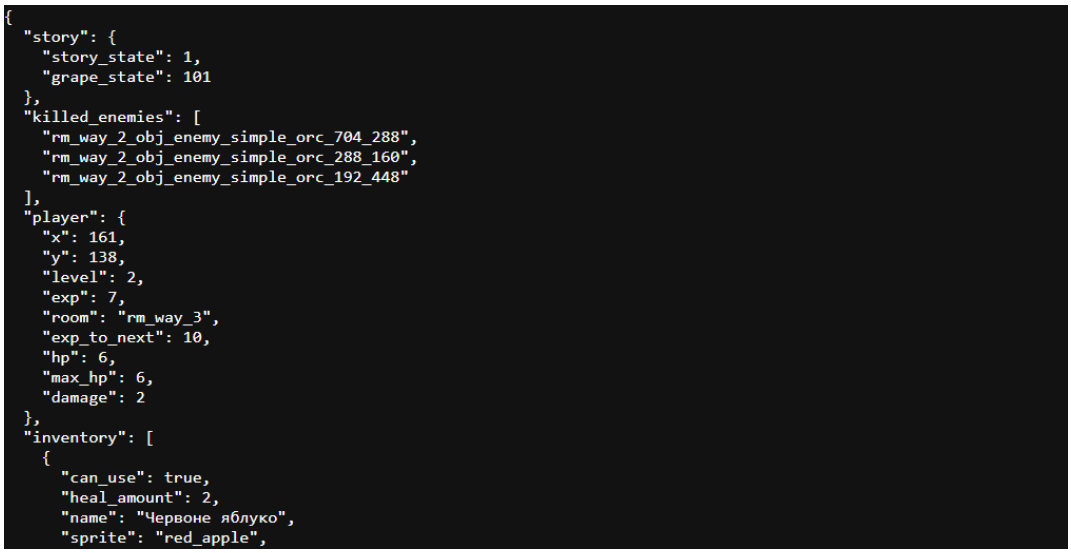
Бойова система тестувалася через зіткнення з ворогами, атаку гравця та отримання шкоди. Було перевірено, що під час атаки створюється зона шкоди, яка наносить урон тільки при попаданні у ворога. Також перевірялося, що ворог не отримує повторну шкоду багато разів від одного удару. Для ворогів перевірялися переслідування гравця, атака, смерть, нарахування досвіду, випадіння предметів і оновлення прогресу квесту.

Рис. 4.1.5 Бій з ворогом у локації гри

Для переходів між кімнатами перевірялося, що персонаж переноситься до правильної локації та з’являється в заданих координатах. Окремо тестувалися умовні переходи. Якщо гравець ще не виконав потрібну сюжетну дію, перехід блокується і показується відповідне повідомлення. Після виконання умови перехід стає доступним.

Рис. 4.1.6 Приклад переходу між кімнатами через прохід у локації

Останнім тестувався сценарій збереження та завантаження прогресу. Після створення збереження перевірялося, що у файлі «save_game.json» записуються кімната, координати персонажа, характеристики, сюжетні стани, квести, інвентар, підібрані предмети та переможені вороги. Після перезапуску гри і вибору пункту продовження перевірялося, що персонаж з’являється у правильній кімнаті, має збережені характеристики, інвентар не втрачається, завершені квести не скидаються, а вже переможені вороги та підібрані предмети не з’являються повторно.



```
{
  "story": {
    "story_state": 1,
    "grape_state": 101
  },
  "killed_enemies": [
    "rm_way_2_obj_enemy_simple_orc_704_288",
    "rm_way_2_obj_enemy_simple_orc_288_160",
    "rm_way_2_obj_enemy_simple_orc_192_448"
  ],
  "player": {
    "x": 161,
    "y": 138,
    "level": 2,
    "exp": 7,
    "room": "rm_way_3",
    "exp_to_next": 10,
    "hp": 6,
    "max_hp": 6,
    "damage": 2
  },
  "inventory": [
    {
      "can_use": true,
      "heal_amount": 2,
      "name": "Червоне яблуко",
      "sprite": "red_apple",
```


Рис. 4.1.7 Фрагмент JSON-файлу збереження, що ілюструє запис прогресу гри

Результати тестування основних сценаріїв подано в таблиці 2.

Таблиця 2

Результати тестування основних сценаріїв роботи гри

№	Сценарій тестування	Очікуваний результат	Результат
1	Запуск нової гри	Гра переходить до початкової кімнати	Виконано
2	Продовження гри	За наявності файлу збереження прогрес завантажується	Виконано
3	Рух персонажа	Персонаж рухається в чотирьох напрямках і не проходить крізь перешкоди	Виконано
4	Блокування руху	Під час діалогу, інвентаря або читання рух блокується	Виконано
5	Діалог з NPC	Відкривається діалогове вікно, репліки перемикаються коректно	Виконано
6	Вибір у діалозі	Обраний варіант змінює стан NPC або квесту	Виконано
7	Підбір предмета	Предмет додається до інвентаря і не з’являється повторно	Виконано
8	Використання предмета	Предмет відновлює HP у межах максимального значення	Виконано
9	Виконання квесту з предметами	За наявності потрібних предметів квест завершується	Виконано
10	Виконання квесту з ворогами	Після перемоги над потрібними ворогами прогрес оновлюється	Виконано
11	Бій з ворогом	Зона шкоди наносить урон при попаданні	Виконано

Таблиця 2 (продовження)

№	Сценарій тестування	Очікуваний результат	Результат
12	Перехід між кімнатами	Персонаж переноситься до потрібної кімнати і координат	Виконано
13	Умовний перехід	Перехід блокується до виконання сюжетної умови	Виконано
14	Збереження гри	Дані записуються у файл «save_game.json»	Виконано
15	Завантаження гри	Відновлюються кімната, позиція, інвентар, квести і стан світу	Виконано

За результатами тестування основні сценарії роботи гри виконуються коректно. Персонаж керується правильно, NPC запускають діалоги, квести змінюють стан залежно від дій гравця, предмети додаються до інвентаря, бойова система реагує на попадання, а збереження відновлює стан проходження після повторного запуску гри. Це підтверджує, що реалізовані системи працюють узгоджено між собою і можуть використовуватися в готовому програмному продукті.

4.2 Перевірка інтерфейсу та ігрових станів

Окремо було перевірено роботу інтерфейсу гри та перемикання ігрових станів. Це важливо, оскільки в проєкті є кілька режимів роботи: вільне переміщення, діалог, вибір відповіді, інвентар, читання текстового об’єкта, бій, екран смерті та збереження прогресу. Якщо ці стани працюють неправильно, гравець може одночасно виконувати дії, які не повинні поєднуватися, наприклад рухатися під час діалогу або відкривати інвентар під час катсцени.

Під час перевірки інтерфейсу було протестовано відображення головного меню, HUD, діалогового вікна, інвентаря, вікна читання записки, повідомлень про заблоковані переходи та екрана смерті. Основна увага приділялася тому, щоб

елементи інтерфейсу правильно масштабувалися, залишалися читабельними та не заважали основним діям гравця.

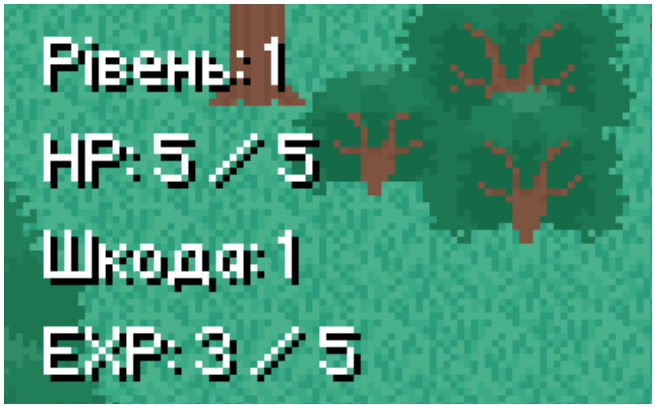


Рис. 4.2.1 Відображення HUD під час перебування персонажа в локації

Також перевірялося, чи правильно гра блокує керування персонажем під час відкриття інтерфейсних вікон. У стані вільного переміщення гравець може рухатися, атакувати, взаємодіяти з NPC, предметами та переходами. Під час діалогу рух персонажа блокується, а натискання клавіш використовується для перемикання реплік. Під час вибору відповіді керування також не повертається персонажу, доки гравець не обере один із варіантів.

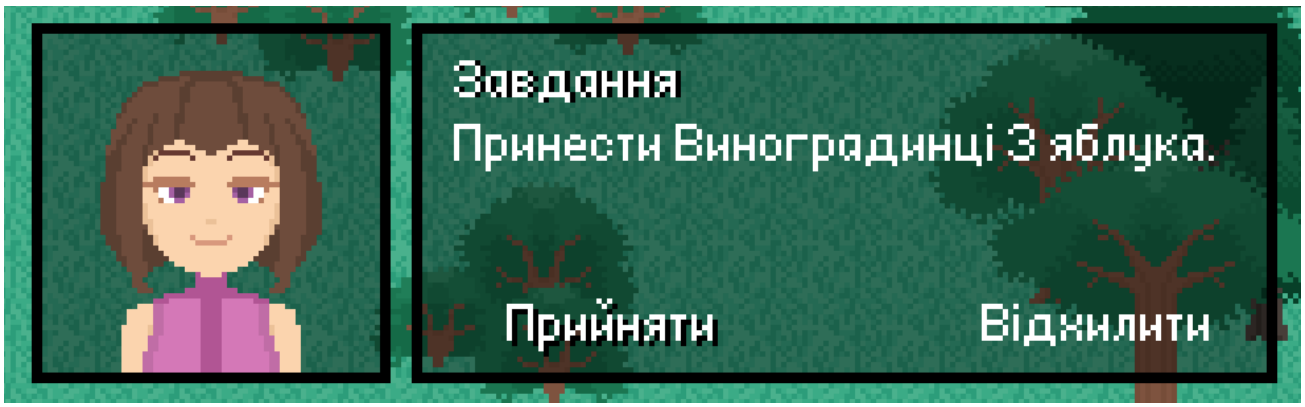


Рис. 4.2.2 Діалогове вікно з варіантами відповіді

Інвентар перевірявся окремо. Було важливо переконатися, що під час його відкриття персонаж не рухається по локації, а клавіші використовуються саме для навігації по списку предметів. Після закриття інвентаря гра має повертатися до звичайного стану, і гравець знову отримує контроль над персонажем.

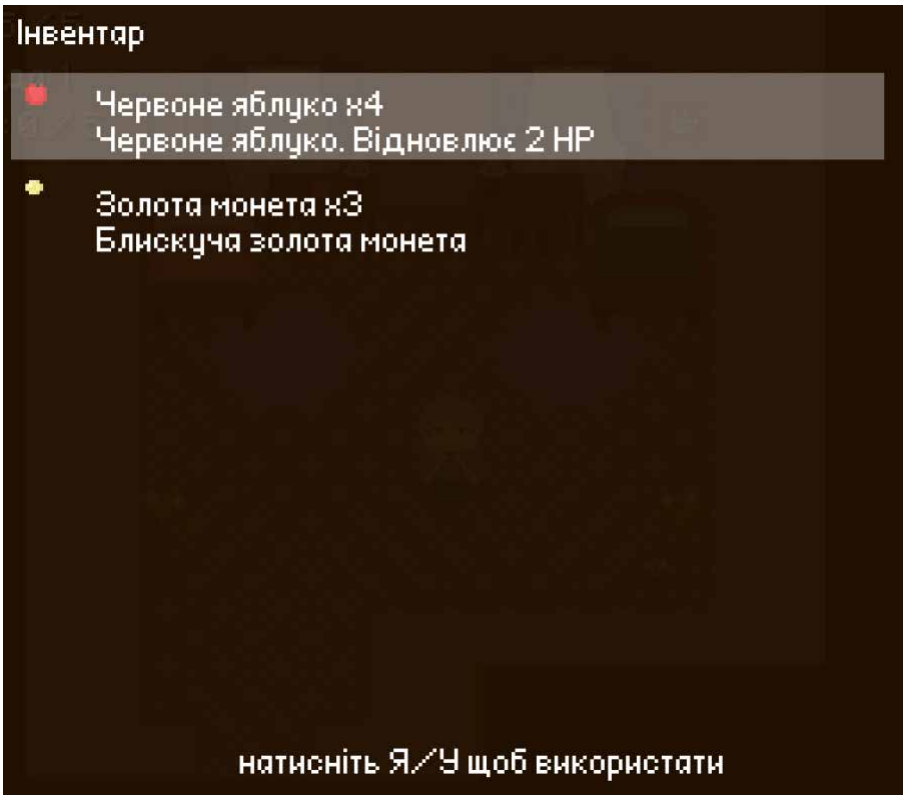


Рис. 4.2.3 Вікно інвентаря з підібраними предметами

Для читабельних об’єктів перевірялося відкриття та закриття вікна читання. Після взаємодії із запискою текст має з’явитися поверх ігрової сцени, рух персонажа блокується, а після закриття вікна гра повертається до стану вільного переміщення.

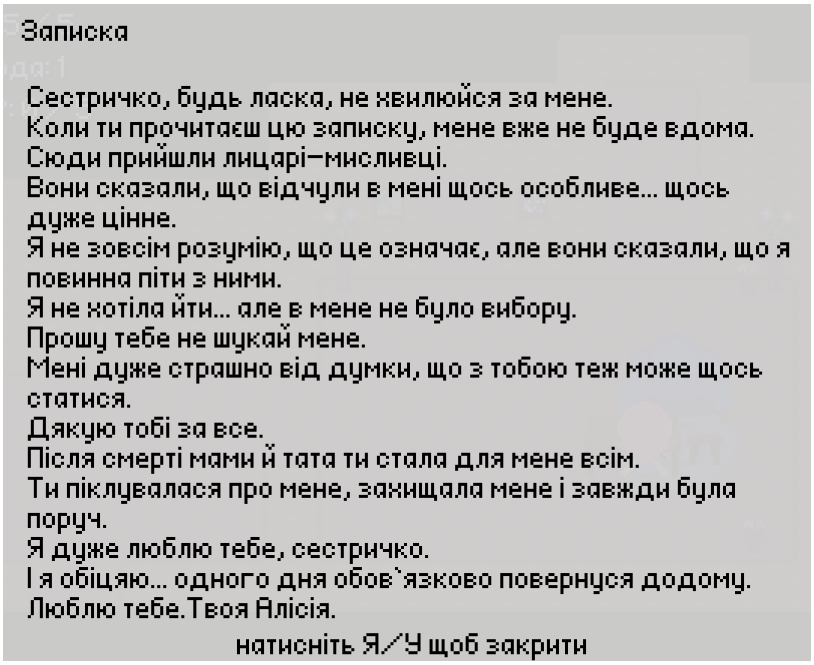


Рис. 4.2.4 Вікно читання записки

Результати перевірки інтерфейсу та ігрових станів подано в таблиці 3.

Таблиця 3

Результати перевірки інтерфейсу та ігрових станів

№	Елемент перевірки	Очікуваний результат	Результат
1	Головне меню	Пункти меню відображаються і реагують на вибір	Виконано
2	HUD	Показуються рівень, HP, шкода та досвід гравця	Виконано
3	Діалогове вікно	Репліки відображаються послідовно	Виконано
4	Вибір у діалозі	Гравець може обрати один із варіантів відповіді	Виконано
5	Інвентар	Предмети відображаються списком і групуються	Виконано
6	Вікно читання	Записка відкривається і закривається коректно	Виконано
7	Блокування руху	Під час діалогу, інвентаря та читання персонаж не рухається	Виконано
8	Повернення до гри	Після закриття вікон керування повертається гравцю	Виконано
9	Екран смерті	Після втрати HP відкривається меню подальшої дії	Виконано
10	Повідомлення переходів	Заблокований перехід показує причину блокування	Виконано

За результатами перевірки інтерфейсні елементи працюють коректно, а ігрові стани не конфліктують між собою. Гра правильно перемикається між вільним переміщенням, діалогом, вибором відповіді, інвентарем, читанням об’єктів та екраном смерті. Це підтверджує, що користувач взаємодіє з грою послідовно, без одночасного запуску несумісних дій.

4.3 Вимоги до апаратного та програмного забезпечення

Для запуску гри користувачу не потрібно встановлювати середовище розробки «GameMaker Studio 2». Гра постачається як готовий виконуваний файл у форматі .exe, а також може передаватися в архіві разом із потрібними файлами

проєкту. Користувач запускає програму без відкриття редактора, вихідного коду або додаткового налаштування середовища розробки [1].

Програмний продукт призначений для запуску на персональному комп’ютері з операційною системою Windows. Оскільки гра виконана у форматі 2D pixel-art RPG і не використовує складні тривимірні сцени, вимоги до апаратного забезпечення залишаються невисокими. Для роботи достатньо базового сучасного комп’ютера або ноутбука, який підтримує запуск Windows-застосунків.

Для керування грою потрібна клавіатура. Вона використовується для переміщення персонажа, взаємодії з NPC і предметами, атаки, переходу між репліками діалогу, відкриття інвентаря та вибору пунктів меню. Миша для основного проходження не є обов’язковою, оскільки основна взаємодія реалізована через клавіші.

Мінімальні вимоги до апаратного та програмного забезпечення подано в таблиці 4.

Таблиця 4

Вимоги до апаратного та програмного забезпечення

№	Компонент	Мінімальна вимога
1	Операційна система	Windows 10 або новіша
2	Процесор	Двоядерний процесор з частотою від 1.5 ГГц
3	Оперативна пам’ять	Від 2 ГБ
4	Відеосистема	Вбудована або дискретна відеокарта з підтримкою 2D-графіки
5	Вільне місце на диску	Від 200 МБ
6	Пристрої введення	Клавіатура
7	Додаткове ПЗ	Не потребує встановлення «GameMaker Studio 2»

Для впровадження гри достатньо передати користувачу виконуваний .exe-файл або архів із грою. Після розпакування архіву користувач може запустити гру через основний файл програми. Файл збереження створюється автоматично під час проходження, тому окреме налаштування папки збереження не потрібне.

Таким чином, програмний продукт можна використовувати на звичайному комп’ютері з Windows без додаткового встановлення середовища розробки. Це спрощує експлуатацію гри та дозволяє запускати її як готовий користувацький застосунок.

4.4 Склад інсталяційного пакету

Для передавання гри користувачу підготовлено інсталяційний пакет. Він може бути поданий у двох варіантах: як інсталяційний .exe-файл або як архів із зібраною грою. В обох випадках користувачу не потрібно відкривати проєкт у «GameMaker Studio 2» або працювати з вихідним кодом.

Перший варіант передбачає запуск інсталяційного .exe-файлу. Користувач відкриває файл, виконує встановлення гри, після чого може запускати її як звичайну програму Windows. Такий варіант зручний для кінцевого користувача, оскільки не потребує ручного пошуку основного файлу гри.

Другий варіант передбачає передавання гри у вигляді архіву. У цьому випадку користувач розпаковує архів у зручну папку та запускає основний виконуваний файл гри. Такий спосіб також не потребує встановлення середовища розробки, але вимагає попереднього розархівування файлів.

Склад інсталяційного пакету подано в таблиці 5.

Таблиця 5

Склад інсталяційного пакету гри

№	Компонент	Призначення
1	Виконуваний .exe-файл	Запуск або встановлення гри
2	Архів із грою	Передавання зібраного проєкту у стиснутому вигляді
3	Файли ресурсів гри	Збереження потрібних даних для роботи програми
4	Файл збереження save_game.json	Створюється автоматично після збереження прогресу

Файл збереження не обов’язково входить до початкового пакету. Він створюється після того, як користувач уперше збереже гру. У Windows файл збереження розміщується в локальній папці користувача за шляхом: C:\Users\UserName\AppData\Local\The_Dream_Where_I_Save_You_0_3\The Dream Where I Save You\save_game.json

Для запуску гри користувачу потрібно виконати такі дії: запустити .exe-файл або розпакувати архів із грою та відкрити основний виконуваний файл. Після цього гра запускається без додаткового налаштування. Якщо під час проходження буде створено збереження, пункт продовження гри стане доступним у головному меню.

4.5 Перспективи розвитку програмного продукту

У подальшому програмний продукт можна розширювати без повної зміни його основної структури, оскільки більшість систем побудована через батьківські та дочірні об’єкти. Це дозволяє додавати нові локації, NPC, ворогів, предмети та квести на основі вже реалізованої логіки [6].

Одним із напрямів розвитку є розширення ігрового світу. До гри можна додати нові кімнати, переходи між ними, додаткові сюжетні зони та побічні локації. Це дозволить зробити проходження довшим і різноманітнішим.

Також можна розширити квестову систему. На основі вже реалізованого «obj_quest_parent» можна створювати нові завдання з різними умовами виконання: збір предметів, перемога над ворогами, розмова з NPC, відкриття певної локації або виконання кількох дій у потрібній послідовності.

Окремим напрямом розвитку є покращення бойової системи. У майбутньому можна додати нові типи ворогів, різні варіанти атак, сильніших босів, додаткові ефекти шкоди та розширену систему нагород після перемоги.

Також доцільно розвивати інвентар. Наприклад, можна додати екіпірування, ключові предмети, сортування речей, окремі категорії предметів та більше варіантів їх використання під час проходження.

Ще одним можливим покращенням є розширення графічного інтерфейсу. Можна додати журнал завдань, карту локацій, екран характеристик персонажа та окреме меню налаштувань. Це зробить гру зручнішою для користувача.

Перспективи розвитку програмного продукту подано в таблиці 6.

Таблиця 6

Перспективи розвитку програмного продукту

№	Напрям розвитку	Очікуваний результат
1	Додавання нових локацій	Розширення ігрового світу
2	Створення нових NPC	Збільшення кількості сюжетних взаємодій
3	Розширення квестової системи	Більше варіантів проходження
4	Додавання нових ворогів	Різноманітніша бойова система
5	Покращення інвентаря	Зручніша робота з предметами
6	Додавання журналу завдань	Краща орієнтація гравця в проходженні
7	Розширення меню налаштувань	Зручніша експлуатація гри

Отже, програмний продукт має можливість подальшого розвитку. Його структура дозволяє поступово додавати новий контент і покращувати окремі системи без повної переробки вже реалізованої логіки.

Висновки до розділу 4

У четвертому розділі було розглянуто питання впровадження та експлуатації розробленої 2D RPG-гри. Основну увагу приділено перевірці роботи готового програмного продукту, вимогам до запуску гри, складу інсталяційного пакету та можливим напрямам подальшого розвитку.

Було проведено ручне тестування основних сценаріїв роботи гри. Перевірялися запуск нового проходження, продовження гри зі збереження, керування персонажем, взаємодія з NPC, виконання квестів, робота інвентаря,

бойова система, переходи між кімнатами та збереження прогресу. За результатами перевірки основні сценарії виконуються коректно, а реалізовані системи працюють узгоджено між собою.

Окремо було перевірено інтерфейс та ігрові стани. Під час тестування підтверджено, що HUD, діалоги, інвентар, вікно читання, повідомлення переходів і екран смерті відображаються правильно. Також встановлено, що гра коректно блокує керування персонажем під час діалогів, вибору відповіді, відкритого інвентаря або читання текстових об’єктів.

Було визначено мінімальні вимоги до апаратного та програмного забезпечення. Гра призначена для запуску на комп’ютері з операційною системою Windows, не потребує встановлення «GameMaker Studio 2» і може використовуватися як готовий виконуваний застосунок. Для керування достатньо клавіатури.

Також було описано склад інсталяційного пакету. Програмний продукт може передаватися користувачу як .exe-файл або як архів із зібраною грою. Файл збереження створюється автоматично після збереження прогресу, тому користувачу не потрібно вручну налаштовувати папку для збережень.

У підрозділі перспектив розвитку визначено можливі напрями подальшого покращення гри: додавання нових локацій, NPC, квестів, ворогів, розширення інвентаря, створення журналу завдань і меню налаштувань. Завдяки використанню батьківських і дочірніх об’єктів ці зміни можна впроваджувати поступово, без повної переробки основної структури проєкту.

Отже, розроблений програмний продукт готовий до запуску та базової експлуатації. Проведене тестування підтвердило працездатність основних ігрових систем, а визначені вимоги й склад пакету дозволяють передати гру користувачу у вигляді готового Windows-застосунку.

ВИСНОВКИ

У бакалаврській кваліфікаційній роботі було розроблено програмне забезпечення піксельної 2D RPG-гри з елементами конструктора. У межах роботи створено ігровий застосунок, у якому користувач може керувати персонажем, досліджувати локації, взаємодіяти з NPC, проходити діалоги, виконувати квести, підбирати предмети, використовувати інвентар, брати участь у боях, переходити між кімнатами та зберігати прогрес проходження.

Під час аналізу предметної області було визначено, що 2D RPG-гра складається не з однієї окремої механіки, а з кількох пов’язаних систем. Було розглянуто роль діалогів, квестів, предметів, інвентаря, бойової взаємодії, переходів між локаціями та збереження прогресу. Також було проаналізовано існуючі ігрові рішення, зокрема Undertale, Kindergarten, Hello Charlotte та Stardew Valley. На основі цього аналізу визначено рішення, які можна використати у власному проєкті: подання інформації через діалоги, залежність подій від дій користувача, зв’язок NPC із завданнями та використання простору як частини проходження.

Було сформовано функціональні та нефункціональні вимоги до програмного продукту. До основних функціональних вимог належать керування персонажем, взаємодія з NPC, робота діалогової системи, прийняття і виконання квестів, підбір та використання предметів, бойова взаємодія, переходи між локаціями, збереження і завантаження прогресу. Нефункціональні вимоги стосувалися зрозумілості інтерфейсу, стабільності роботи гри, підтримки українського тексту та можливості подальшого розширення проєкту.

На етапі проєктування було визначено загальну архітектуру гри. Основні системи поділено на окремі компоненти: гравець, NPC, діалоги, квести, предмети, інвентар, вороги, бойова система, переходи між кімнатами та збереження. Також було описано об’єктно-орієнтовану структуру гри, у якій важливу роль мають батьківські та дочірні об’єкти. Такий підхід дав змогу

передбачити елементи конструктора: нові NPC, вороги, предмети, квести й інші об’єкти можуть створюватися на основі спільної логіки без повного переписування систем.

Реалізацію програмного продукту виконано в середовищі GameMaker Studio 2 мовою GML. Для графічних ресурсів використано піксельну графіку, створену або відредаговану в Aseprite. Для українського тексту в інтерфейсі використано шрифт, який підтримує кирилицю та відповідає загальному піксельному стилю гри.

У процесі розробки було реалізовано основні контролери гри. Окремі контролери відповідають за загальний стан гри, меню, діалоги, інвентар, читання текстових об’єктів, екран смерті та масштабування інтерфейсу. Завдяки цьому логіка не зберігається в одному об’єкті, а розподілена між частинами, які мають власну відповідальність.

Було реалізовано керування персонажем, обробку руху, анімацій, атак, отримання шкоди, блокування керування під час діалогів або відкритих інтерфейсних вікон. Бойова система побудована через окремий об’єкт шкоди, який перевіряє попадання та передає цілі значення урону. Це дозволило не змішувати запуск атаки, перевірку зіткнення та зміну здоров’я ворога в одному місці.

Системи NPC, діалогів і квестів реалізовано як пов’язані частини проходження. NPC запускають діалоги, діалоги можуть змінювати стан персонажів або завдань, а квести перевіряють виконання потрібних умов. Предмети та інвентар також пов’язані з квестами і системою збереження. Після підбору предмет додається до інвентаря, а факт підбору зберігається, щоб об’єкт не з’являвся повторно після переходу між кімнатами або завантаження гри.

Окремо було реалізовано переходи між кімнатами, колізії та глибину малювання. Переходи дають змогу переносити персонажа до потрібної локації та встановлювати його в задану позицію. Умовні переходи залежать від сюжетного стану, тому гравець не може перейти далі раніше, ніж виконає

потрібну дію. Колізії не дозволяють проходити крізь перешкоди, а система глибини малювання забезпечує правильне перекриття персонажа, NPC, ворогів і декорацій.

Система збереження та завантаження реалізована через JSON-файл. У ньому зберігаються дані гравця, поточна кімната, позиція, характеристики, сюжетні стани, квести, інвентар, підібрані предмети та переможені вороги. Після завантаження відновлюється не лише координата персонажа, а повний стан проходження. Це дає змогу продовжити гру з того моменту, на якому користувач зробив збереження.

Для перевірки працездатності програмного продукту було проведено ручне тестування основних сценаріїв. Перевірялися запуск гри, продовження проходження зі збереження, рух персонажа, взаємодія з NPC, робота діалогів, виконання квестів, підбір і використання предметів, бойова система, переходи між кімнатами та завантаження прогресу. За результатами тестування основні сценарії виконуються коректно.

Також було визначено вимоги до запуску гри та склад інсталяційного пакету. Гра може запускатися на комп’ютері з операційною системою Windows як готовий виконуваний застосунок. Користувачу не потрібно встановлювати GameMaker Studio 2 або відкривати вихідний код проєкту. Для керування достатньо клавіатури, а файл збереження створюється автоматично після запису прогресу.

Отже, поставлену мету роботи досягнуто. Розроблено працездатну піксельну 2D RPG-гру з елементами конструктора, у якій основні системи працюють разом і можуть бути розширені в майбутньому. Програмний продукт можна використовувати як приклад побудови невеликої RPG-гри з діалогами, квестами, інвентарем, бойовою системою, переходами між локаціями та збереженням прогресу.

ЛІТЕРАТУРНІ ДЖЕРЕЛА

1. GameMaker Manual. YoYo Games, 2024. URL: https://manual.gamemaker.io/monthly/en/index.htm (дата звернення: 18.04.2026).
2. Salen K., Zimmerman E. Rules of Play: Game Design Fundamentals. Cambridge: MIT Press, 2004.
3. Хто такий геймдизайнер в ігровій індустрії. GameDev DOU, 2022. URL: https://gamedev.dou.ua/articles/game-designer-in-gamedev/ (дата звернення: 18.04.2026).
4. Rules, Mechanics, Gameplay and Logic of Game Development. Medium, 2023. URL: https://rctai.medium.com/rules-mechanics-gameplays-and-logic-of-game-development-3573b9d730aa (дата звернення: 18.04.2026).
5. Burnham P. How to Make an RPG from Scratch in GameMaker Studio 2 [Відеоплейлист]. YouTube, 2021. URL: https://surl.li/xcnhht (дата звернення: 20.04.2026).
6. YoYo Games. GameMaker Manual: Objects. URL: https://manual.gamemaker.io/lts/en/GameMaker_Language/GML_Reference/Asset_Management/Objects/Objects.htm (дата звернення: 21.04.2026).
7. YoYo Games. GameMaker Manual: Object Events. URL: https://manual.gamemaker.io/lts/en/The_Asset_Editors/Object_Properties/Object_Events.htm (дата звернення: 22.04.2026).
8. YoYo Games. GameMaker Manual: Other Events. URL: https://manual.gamemaker.io/lts/en/The_Asset_Editors/Object_Properties/Other_Events.htm (дата звернення: 23.04.2026).
9. YoYo Games. GameMaker Manual: Rooms. URL: https://manual.gamemaker.io/lts/en/GameMaker_Language/GML_Reference/Asset_Management/Rooms/Rooms.htm (дата звернення: 24.04.2026).

10. YoYo Games. GameMaker Manual: JSON Parse. URL: https://surl.li/exuesh (дата звернення: 24.04.2026).
11. Rogers S. Level Up! The Guide to Great Video Game Design. 2nd ed. Hoboken: Wiley, 2014.
12. Рольова відеогра. Вікіпедія. URL: https://uk.wikipedia.org/wiki/Рольова_відеогра (дата звернення: 26.04.2026).
13. Піксельна графіка. Вікіпедія. URL: https://uk.wikipedia.org/wiki/Піксельна_графіка (дата звернення: 27.04.2026).
14. Самусь Ю., Чемерис Г. Ігри в стилі піксель-арт. Дизайн, візуальне мистецтво та творчість: сучасні тенденції та технології: матеріали II міжнародної науково-практичної конференції. URL: https://zenodo.org/records/10372025 (дата звернення: 29.04.2026).
15. Лугова Т. А., Блажко О. А. Проектування комп'ютерних ігор для навчання: навчальний підручник. Одеса: Одеський національний політехнічний університет, 2018. 209 с. URL: https://surl.li/scfvei (дата звернення: 05.05.2026).
16. Aseprite. Aseprite Docs. URL: https://www.aseprite.org/docs/ (дата звернення: 10.05.2026).
17. Freeman W. Creating Emotion in Games: The Craft and Art of Emotioneering. Indianapolis: New Riders, 2003.
18. Як організовують свою працю українські соло-розробники. GameDev DOU, 2025. URL: https://gamedev.dou.ua/articles/how-ukrainian-solo-developers-working/ (дата звернення: 14.05.2026).
19. Використання комп'ютерних ігор в навчанні та вихованні дітей дошкільного віку. Освітологічний дискурс, 2022. URL: https://oip-journal.org/index.php/oip/article/view/30 (дата звернення: 17.05.2026).
20. Koster R. A Theory of Fun for Game Design. 2nd ed. Sebastopol: O'Reilly Media, 2013.

21. Fox T. Undertale. URL: https://undertale.com/ (дата звернення: 21.05.2026).
22. Kindergarten. Steam. URL: https://store.steampowered.com/app/589590/Kindergarten/ (дата звернення: 21.05.2026).
23. Etherane. Hello Charlotte EP1: Junk Food, Gods and Teddy Bears. itch.io. URL: https://etherane.itch.io/hello-charlotte-ep1 (дата звернення: 21.05.2026).
24. Stardew Valley. ConcernedApe. URL: https://www.stardewvalley.net/ (дата звернення: 21.05.2026).
25. YoYo Games. GameMaker Manual: place_meeting. URL: https://manual.gamemaker.io/lts/en/GameMaker_Language/GML_Reference/Movement_And_Collisions/Collisions/place_meeting.htm (дата звернення: 21.05.2026).

ДОДАТОК А

ФРАГМЕНТИ ПРОГРАМНОГО КОДУ GUI-СИСТЕМ

А.1 Код малювання діалогового вікна в об’єкті obj_dialog_controller, подія Draw GUI End

```
if (!active) exit;
if (!instance_exists(current_npc)) exit;
var gui_data = scr_get_gui_offset();
var scale = gui_data.scale;
var offset_x = gui_data.offset_x;
var offset_y = gui_data.offset_y;
draw_set_font(font_dialog);
draw_set_halign(fa_left);
draw_set_valign(fa_top);
var _line = current_npc.dialog_lines[current_line];
var _speaker = _line.speaker;
var _portrait = _line.portrait;
var _full_text = _line.text;
var _shown_text = string_copy(_full_text, 1, visible_chars);
var bx = offset_x + box_x * scale;
var by = offset_y + box_y * scale;
var px = offset_x + portrait_x * scale;
var py = offset_y + portrait_y * scale;
var tx = offset_x + text_x * scale;
var ty_name = offset_y + name_y * scale;
var ty_phrase = offset_y + phrase_y * scale;
draw_sprite_stretched(dialog_window, 0, bx, by, box_w * scale, box_h * scale);
if (_portrait != noone) {
    draw_sprite_stretched(_portrait, 0, px, py, portrait_w * scale, portrait_h *
scale);
}
draw_set_color(c_black);
draw_text_transformed(tx + scale, ty_name, _speaker, scale, scale, 0);
draw_set_color(c_white);
draw_text_transformed(tx, ty_name, _speaker, scale, scale, 0);
draw_set_color(c_white);
draw_text_ext_transformed(tx, ty_phrase, _shown_text, line_sep, text_w, scale, scale,
0);
if (!choice_mode && visible_chars >= string_length(_full_text)) {
    draw_text_transformed(
        offset_x + 137 * scale,
        offset_y + 222 * scale,
        "натисніть Я/У щоб продовжити",
        scale,
        scale,
        0
    );
}
if (choice_mode && array_length(choice_options) > 0) {
    var choice_y = offset_y + 222 * scale;
    for (var i = 0; i < array_length(choice_options); i++) {
        var choice_x = offset_x + (132 + i * 90) * scale;
        if (i == choice_index) {
            draw_set_color(c_black);
```

```
            draw_text_transformed(
                choice_x + scale,
                choice_y + scale,
                choice_options[i],
                scale,
                scale,
                0
            );
            draw_set_color(c_white);
            draw_text_transformed(
                choice_x,
                choice_y,
                choice_options[i],
                scale,
                scale,
                0
            );
        }
        else {
            draw_set_color(c_white);
            draw_text_transformed(
                choice_x,
                choice_y,
                choice_options[i],
                scale,
                scale,
                0
            );
        }
    }
}
```

А.2 Код малювання панелі читання в об’єкті obj_read_controller, подія Draw GUI End

```
if (!open) exit;
var gui_data = scr_get_gui_offset();
var scale = gui_data.scale;
var offset_x = gui_data.offset_x;
var offset_y = gui_data.offset_y;
draw_set_font(font_read);
draw_set_halign(fa_left);
draw_set_valign(fa_top);
var px = offset_x + panel_x * scale;
var py = offset_y + panel_y * scale;
var pw = panel_w * scale;
var ph = panel_h * scale;
draw_set_alpha(0.96);
draw_set_color(make_colour_rgb(210, 210, 210));
draw_rectangle(px, py, px + pw, py + ph, false);
draw_set_alpha(1);
draw_set_color(c_black);
draw_rectangle(px, py, px + pw, py + ph, true);
draw_set_color(c_black);
draw_text_transformed(
    offset_x + title_x * scale,
    offset_y + title_y * scale,
    current_title,
    scale,
    scale,
```

```
    0
);
draw_text_ext_transformed(
    offset_x + text_x * scale,
    offset_y + text_y * scale,
    current_text,
    line_sep,
    text_w,
    scale,
    scale,
    0
);
draw_set_halign(fa_center);
draw_text_transformed(
    offset_x + (gui_data.base_w div 2) * scale,
    offset_y + 228 * scale,
    "натисніть Я/У щоб закрити",
    scale,
    scale,
    0
);
draw_set_halign(fa_left);
```

ДОДАТОК Б

ФРАГМЕНТИ ПРОГРАМНОГО КОДУ СИСТЕМИ ЗБЕРЕЖЕННЯ ТА ЗАВАНТАЖЕННЯ

Б.1 Код скрипта «scr_save_game()»

```
function scr_save_game() {
    var save_data = {};
    var player_data = {};
    if (instance_exists(obj_player_controller)) {
        var player = instance_find(obj_player_controller, 0);
        player_data.room = room_get_name(room);
        player_data.x = player.x;
        player_data.y = player.y;
    }
    else {
        player_data.room = room_get_name(room);
        player_data.x = 0;
        player_data.y = 0;
    }
    player_data.level = global.player_level;
    player_data.exp = global.player_exp;
    player_data.exp_to_next = global.player_exp_to_next;
    player_data.hp = global.player_hp;
    player_data.max_hp = global.player_max_hp;
    player_data.damage = global.player_damage;
    save_data.player = player_data;
    var story_data = {};
    story_data.story_state = global.story_state;
    story_data.grape_state = global.grape_state;
    save_data.story = story_data;
    var quests_data = {};
    if (variable_global_exists("quest_states") && ds_exists(global.quest_states,
ds_type_map)) {
        var key = ds_map_find_first(global.quest_states);

        while (!is_undefined(key)) {
            quests_data[$ key] = global.quest_states[? key];

            key = ds_map_find_next(global.quest_states, key);
        }
    }
    save_data.quests = quests_data;
      var inventory_data = [];
      if (variable_global_exists("inventory") && ds_exists(global.inventory,
ds_type_list)) {
          for (var i = 0; i < ds_list_size(global.inventory); i++) {
              var item = global.inventory[| i];

              array_push(inventory_data, {
                  name : item.name,
                  sprite : sprite_get_name(item.sprite),
                  description : item.description,
                  use_text : item.use_text,
                  can_use : item.can_use,
```

```
                heal_amount : item.heal_amount
            });
        }
    }
    save_data.inventory = inventory_data;
    var picked_items_data = [];
    if (variable_global_exists("picked_items") && ds_exists(global.picked_items,
ds_type_map)) {
        var picked_key = ds_map_find_first(global.picked_items);

        while (!is_undefined(picked_key)) {
            array_push(picked_items_data, picked_key);
            picked_key = ds_map_find_next(global.picked_items, picked_key);
        }
    }
    save_data.picked_items = picked_items_data;
    var killed_enemies_data = [];
    if (variable_global_exists("killed_enemies") && ds_exists(global.killed_enemies,
ds_type_map)) {
        var enemy_key = ds_map_find_first(global.killed_enemies);
        while (!is_undefined(enemy_key)) {
            array_push(killed_enemies_data, enemy_key);
            enemy_key = ds_map_find_next(global.killed_enemies, enemy_key);
        }
    }
    save_data.killed_enemies = killed_enemies_data;
    var json_string = json_stringify(save_data);
    var save_dir = "The Dream Where I Save You";
    if (!directory_exists(save_dir)) {
        directory_create(save_dir);
    }
    var save_path = save_dir + "/save_game.json";
    var file = file_text_open_write(save_path);
    file_text_write_string(file, json_string);
    file_text_close(file);
    global.current_run_has_save = true;
    show_debug_message("GAME SAVED:");
    show_debug_message(json_string);
}
```

Б.2 Код скрипта «scr_load_game()»

```
function scr_load_game() {
    scr_ensure_game_controllers();
    if (!file_exists("The Dream Where I Save You/save_game.json")) {
        show_debug_message("SAVE FILE NOT FOUND");
        return false;
    }
    global.current_run_has_save = true;
    var file = file_text_open_read("The Dream Where I Save You/save_game.json");
    var json_string = "";
    while (!file_text_eof(file)) {
        json_string += file_text_read_string(file);
        file_text_readln(file);
    }

    file_text_close(file);
    var save_data = json_parse(json_string);
    if (variable_struct_exists(save_data, "player")) {
        var player_data = save_data.player;
        if (variable_struct_exists(player_data, "level")) {
```

```
            global.player_level = player_data.level;
        }
        if (variable_struct_exists(player_data, "exp")) {
            global.player_exp = player_data.exp;
        }
        if (variable_struct_exists(player_data, "exp_to_next")) {
            global.player_exp_to_next = player_data.exp_to_next;
        }
        if (variable_struct_exists(player_data, "hp")) {
            global.player_hp = player_data.hp;
        }
        if (variable_struct_exists(player_data, "max_hp")) {
            global.player_max_hp = player_data.max_hp;
        }
        if (variable_struct_exists(player_data, "damage")) {
            global.player_damage = player_data.damage;
        }
        if (
            variable_struct_exists(player_data, "x") &&
            variable_struct_exists(player_data, "y")
        ) {
            global.spawn_x = player_data.x;
            global.spawn_y = player_data.y;
            global.use_spawn_point = true;
        }
        if (variable_struct_exists(player_data, "room")) {
            var target_room = asset_get_index(player_data.room);
            if (target_room != -1) {
                room_goto(target_room);
            }
            else {
                show_debug_message("ROOM NOT FOUND: " + string(player_data.room));
                room_goto(rm_players_house_entrance_and_living_room);
            }
        }
    }
    if (variable_struct_exists(save_data, "story")) {
        var story_data = save_data.story;

        if (variable_struct_exists(story_data, "story_state")) {
            global.story_state = story_data.story_state;
        }

        if (variable_struct_exists(story_data, "grape_state")) {
            global.grape_state = story_data.grape_state;
        }
    }
    if (!variable_global_exists("quest_states") || !ds_exists(global.quest_states,
ds_type_map)) {
        global.quest_states = ds_map_create();
    }
    if (variable_struct_exists(save_data, "quests")) {
        var quests_data = save_data.quests;
        var keys = variable_struct_get_names(quests_data);
        for (var i = 0; i < array_length(keys); i++) {
            var quest_id = keys[i];
            var quest_state = quests_data[$ quest_id];
            global.quest_states[? quest_id] = quest_state;
        }
    }
```

```
        if (!variable_global_exists("inventory") || !ds_exists(global.inventory,
ds_type_list)) {
            global.inventory = ds_list_create();
        }
        else {
            ds_list_clear(global.inventory);
        }
        if (variable_struct_exists(save_data, "inventory")) {
            var inventory_data = save_data.inventory;
            for (var i = 0; i < array_length(inventory_data); i++) {
                var saved_item = inventory_data[i];
                var item_sprite = noone;
                if (variable_struct_exists(saved_item, "sprite")) {
                    item_sprite = asset_get_index(saved_item.sprite);
                }
                var item_data = {
                    name : saved_item.name,
                    sprite : item_sprite,
                    description : saved_item.description,
                    use_text : saved_item.use_text,
                    can_use : saved_item.can_use,
                    heal_amount : saved_item.heal_amount
                };
                ds_list_add(global.inventory, item_data);
            }
        }
        if (!variable_global_exists("picked_items") || !ds_exists(global.picked_items,
ds_type_map)) {
            global.picked_items = ds_map_create();
        }
        else {
            ds_map_clear(global.picked_items);
        }
        if (variable_struct_exists(save_data, "picked_items")) {
            var picked_items_data = save_data.picked_items;
            for (var p = 0; p < array_length(picked_items_data); p++) {
                var picked_id = picked_items_data[p];
                global.picked_items[? picked_id] = true;
            }
        }
        if (!variable_global_exists("killed_enemies") ||
!ds_exists(global.killed_enemies, ds_type_map)) {
            global.killed_enemies = ds_map_create();
        }
        else {
            ds_map_clear(global.killed_enemies);
        }
        if (variable_struct_exists(save_data, "killed_enemies")) {
            var killed_enemies_data = save_data.killed_enemies;
            for (var k = 0; k < array_length(killed_enemies_data); k++) {
                var killed_enemy_id = killed_enemies_data[k];
                global.killed_enemies[? killed_enemy_id] = true;
            }
        }
    show_debug_message("GAME LOADED");
    return true;
}
```

ДОДАТОК В

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

Декларація про академічну доброчесність та використання ШІ

Я, Малинка Тетяна Олександрівна, здобувачка НУБіП України, усвідомлюючи відповідальність за порушення академічної доброчесності, цією заявою підтверджую, що:

1. Моя бакалаврська кваліфікаційна робота на тему «Програмне забезпечення піксельної 2D RPG-гри з елементами конструктора» є результатом моєї особистої праці.
2. Усі запозичення з текстів інших авторів мають відповідні посилання згідно з чинними стандартами.
3. Щодо використання інструментів ШІ зазначаю, що (обрати необхідне):

- [✓] інструменти генеративного ШІ не використовувалися.
- [ ] інструменти ШІ _____ були використані для:
  - [ ] перекладу іншомовних джерел;
  - [ ] стилістичного редагування та перевірки граматики;
  - [ ] допомоги у написанні/відлагодженні програмного коду;
  - [ ] інше: _____.

Я розумію, що надання недостовірної інформації може призвести до скасування результатів захисту та відрахування з числа здобувачів НУБіП України.

«___»_____ 2026 р. _____ **Тетяна МАЛИНКА**

(підпис)